
YosysHQ Yosys

Version 0.51

YosysHQ GmbH

Mar 26, 2025

CONTENTS

1	What is Yosys	11
1.1	What you can do with Yosys	12
1.1.1	Typical applications for Yosys	12
1.1.2	Things you can't do	12
1.2	The Yosys family	12
1.3	The original thesis abstract	13
1.3.1	Benefits of open source HDL synthesis	13
1.3.2	History of Yosys	14
2	Getting started with Yosys	17
2.1	Installation	17
2.1.1	CAD suite(s)	17
2.1.2	Building from source	18
2.1.3	Source tree and build system	20
2.2	Synthesis starter	21
2.2.1	Demo design	21
2.2.2	Loading the design	23
2.2.3	Elaboration	23
2.2.4	Flattening	29
2.2.5	The coarse-grain representation	30
2.2.6	Hardware mapping	37
2.2.7	Final steps	45
2.3	Scripting in Yosys	48
2.3.1	Script parsing	48
2.3.2	The synthesis starter script	48
3	Using Yosys (advanced)	55
3.1	Synthesis in detail	55
3.1.1	Synth commands	55
3.1.2	Converting process blocks	57
3.1.3	FSM handling	60
3.1.4	Memory handling	62
3.1.5	Optimization passes	78
3.1.6	Technology mapping	85
3.1.7	The extract pass	86
3.1.8	The ABC toolbox	100
3.1.9	Mapping to cell libraries	103
3.2	More scripting	107
3.2.1	Loading a design	108
3.2.2	Selections	109

3.2.3	Interactive design investigation	122
3.2.4	Symbolic model checking	140
4	Yosys internals	145
4.1	Internal flow	145
4.1.1	Flow overview	146
4.1.2	Control and data flow	146
4.1.3	The Verilog and AST frontends	148
4.2	Internal formats	157
4.2.1	The RTL Intermediate Language (RTLIL)	158
4.3	Working with the Yosys codebase	164
4.3.1	Writing extensions	165
4.3.2	Compiling with Verific library	173
4.3.3	Writing a new backend using FunctionalIR	176
4.3.4	Contributing to Yosys	189
4.3.5	Testing Yosys	189
4.4	Techmap by example	190
4.4.1	Mapping OR3X1	190
4.4.2	Conditional techmap	192
4.4.3	Scripting in map modules	193
4.4.4	Handling constant inputs	195
4.4.5	Handling shorted inputs	197
4.4.6	Notes on using techmap	198
4.5	Notes on Verilog support in Yosys	199
4.5.1	Unsupported Verilog-2005 Features	199
4.5.2	Verilog Attributes and non-standard features	199
4.5.3	Non-standard or SystemVerilog features for formal verification	203
4.5.4	Supported features from SystemVerilog	203
4.6	Hashing and associative data structures in Yosys	204
4.6.1	Container classes based on hashing	204
4.6.2	The hash function	205
4.6.3	Making a type hashable	205
4.6.4	Porting plugins from the legacy interface	206
5	A primer on digital circuit synthesis	207
5.1	Levels of abstraction	207
5.1.1	System level	208
5.1.2	High level	208
5.1.3	Behavioural level	208
5.1.4	Register-Transfer Level (RTL)	209
5.1.5	Logical gate level	209
5.1.6	Physical gate level	210
5.1.7	Switch level	210
5.1.8	Yosys	210
5.2	Features of synthesizable Verilog	210
5.2.1	Structural Verilog	211
5.2.2	Expressions in Verilog	211
5.2.3	Behavioural modelling	211
5.2.4	Functions and tasks	213
5.2.5	Conditionals, loops and generate-statements	213
5.2.6	Arrays and memories	213
5.3	Challenges in digital circuit synthesis	214
5.3.1	Standards compliance	214
5.3.2	Optimizations	214

5.3.3	Technology mapping	214
5.4	Script-based synthesis flows	215
5.5	Methods from compiler design	216
5.5.1	Lexing and parsing	216
5.5.2	Multi-pass compilation	217
5.5.3	Static Single Assignment (SSA) form	218
6	RTLIL text representation	221
6.1	Lexical elements	221
6.1.1	Characters	221
6.1.2	Identifiers	221
6.1.3	Values	222
6.1.4	Strings	222
6.1.5	Comments	222
6.2	File	222
6.2.1	Autoindex statements	223
6.2.2	Modules	223
6.2.3	Attribute statements	223
6.2.4	Signal specifications	223
6.2.5	Connections	224
6.2.6	Wires	224
6.2.7	Memories	224
6.2.8	Cells	224
6.2.9	Processes	225
6.2.10	Switches	225
6.2.11	Syncs	225
7	Auxiliary libraries	227
7.1	BigInt	227
7.2	dlfcn-win32	227
7.3	ezSAT	227
7.4	fst	227
7.5	json11	228
7.6	MiniSAT	228
7.7	SHA1	228
7.8	SubCircuit	228
8	Auxiliary programs	229
8.1	yosys-config	229
8.2	yosys-filterlib	230
8.3	yosys-abc	230
8.4	yosys-smtbmc	230
8.5	yosys-witness	234
9	Internal cell library	235
9.1	Word-level cells	235
9.1.1	Unary operators	235
9.1.2	Binary operators	241
9.1.3	Multiplexers	261
9.1.4	Registers	265
9.1.5	Memories	275
9.1.6	Finite state machines	287
9.1.7	Coarse arithmetics	289
9.1.8	Arbitrary logic functions	295
9.1.9	Specify rules	297

9.1.10	Formal verification cells	302
9.1.11	Debugging cells	309
9.1.12	Wire cells	312
9.2	Gate-level cells	313
9.2.1	Combinatorial cells (simple)	313
9.2.2	Combinatorial cells (combined)	318
9.2.3	Flip-flop cells	324
9.2.4	Latch cells	379
9.2.5	Other gate-level cells	391
9.3	Cell properties	391
10	Command line reference	393
10.1	Yosys environment variables	394
10.2	abc - use ABC for technology mapping	395
10.3	abc9 - use ABC9 for technology mapping	399
10.4	abc9_exe - use ABC9 for technology mapping	402
10.5	abc9_ops - helper functions for ABC9	403
10.6	abc_new - (experimental) use ABC for SC technology mapping (new)	405
10.7	abstract - replace signals with abstract values during formal verification	406
10.8	add - add objects to the design	408
10.9	aigmap - map logic to and-inverter-graph circuit	408
10.10	alumacc - extract ALU and MACC cells	409
10.11	anlogic_eqn - Anlogic: Calculate equations for luts	409
10.12	anlogic_fixcarry - Anlogic: fix carry chain	409
10.13	assertpmux - adds asserts for parallel muxes	409
10.14	async2sync - convert async FF inputs to sync circuits	410
10.15	attrmap - renaming attributes	410
10.16	attrmvcp - move or copy attributes from wires to driving cells	411
10.17	autoname - automatically assign names to objects	411
10.18	blackbox - convert modules into blackbox modules	411
10.19	bmuxmap - transform \$bmux cells to trees of \$mux cells	412
10.20	booth - map \$mul cells to Booth multipliers	412
10.21	box_derive - derive box modules	412
10.22	bufnorm - (experimental) convert design into buffered-normalized form	413
10.23	bugpoint - minimize testcases	414
10.24	bwmuxmap - replace \$bwmux cells with equivalent logic	416
10.25	cd - a shortcut for 'select -module <name>'	416
10.26	cellmatch - match cells to their targets in cell library	416
10.27	check - check for obvious problems in the design	417
10.28	chformal - change formal constraints of the design	418
10.29	chparam - re-evaluate modules with new parameters	419
10.30	chtype - change type of cells in the design	419
10.31	clean - remove unused cells and wires	419
10.32	clean_zerowidth - clean zero-width connections from the design	420
10.33	clk2fflogic - convert clocked FFs to generic \$ff cells	420
10.34	clkbufmap - insert clock buffers on clock networks	420
10.35	clockgate - extract clock gating out of flip flops	421
10.36	connect - create or remove connections	422
10.37	connect_rpc - connect to RPC frontend	422
10.38	connwrappers - match width of input-output port pairs	423
10.39	coolrunner2_fixup - insert necessary buffer cells for CoolRunner-II architecture	424
10.40	coolrunner2_sop - break \$sop cells into ANDTERM/ORTERM cells	424
10.41	copy - copy modules in the design	424
10.42	cover - print code coverage counters	424

10.43	cutpoint - adds formal cut points to the design	425
10.44	debug - run command with debug log messages enabled	426
10.45	delete - delete objects in the design	426
10.46	deminout - demote inout ports to input or output	426
10.47	demuxmap - transform \$demux cells to \$eq + \$mux cells	426
10.48	design - save, restore and reset current design	427
10.49	dffinit - set INIT param on FF cells	428
10.50	dfflegalize - convert FFs to types supported by the target	429
10.51	dfflibmap - technology mapping of flip-flops	430
10.52	dffunmap - unmap clock enable and synchronous reset from FFs	431
10.53	dft_tag - create tagging logic for data flow tracking	431
10.54	dump - print parts of the design in RTLIL format	431
10.55	echo - turning echoing back of commands on and off	432
10.56	edgetypes - list all types of edges in selection	432
10.57	efinix_fixcarry - Efinix: fix carry chain	432
10.58	equiv_add - add a \$equiv cell	433
10.59	equiv_induct - proving \$equiv cells using temporal induction	433
10.60	equiv_make - prepare a circuit for equivalence checking	434
10.61	equiv_mark - mark equivalence checking regions	434
10.62	equiv_miter - extract miter from equiv circuit	434
10.63	equiv_opt - prove equivalence for optimized circuit	435
10.64	equiv_purge - purge equivalence checking module	436
10.65	equiv_remove - remove \$equiv cells	436
10.66	equiv_simple - try proving simple \$equiv instances	437
10.67	equiv_status - print status of equivalent checking module	437
10.68	equiv_struct - structural equivalence checking	437
10.69	eval - evaluate the circuit given an input	438
10.70	example_dt - drivertools example	438
10.71	exec - execute commands in the operating system shell	439
10.72	expose - convert internal signals to module ports	439
10.73	extract - find subcircuits and replace them with cells	440
10.74	extract_counter - Extract GreenPak4 counter cells	442
10.75	extract_fa - find and extract full/half adders	442
10.76	extract_reduce - converts gate chains into \$reduce_* cells	443
10.77	extractinv - extract explicit inverter cells for invertible cell pins	443
10.78	flatten - flatten design	443
10.79	flowmap - pack LUTs with FlowMap	444
10.80	fmcombine - combine two instances of a cell into one	445
10.81	fminit - set init values/sequences for formal	446
10.82	formalff - prepare FFs for formal	446
10.83	freduce - perform functional reduction	447
10.84	fsm - extract and optimize finite state machines	448
10.85	fsm_detect - finding FSMs in design	449
10.86	fsm_expand - expand FSM cells by merging logic into it	449
10.87	fsm_export - exporting FSMs to KISS2 files	449
10.88	fsm_extract - extracting FSMs in design	450
10.89	fsm_info - print information on finite state machines	450
10.90	fsm_map - mapping FSMs to basic logic	450
10.91	fsm_opt - optimize finite state machines	451
10.92	fsm_recode - recoding finite state machines	451
10.93	fst2tb - generate testbench out of fst file	451
10.94	future - resolve future sampled value functions	452
10.95	gatemate_foldinv - fold inverters into Gatemate LUT trees	452
10.96	glift - create GLIFT models and optimization problems	452

10.97	greenpak4_dffinv - merge greenpak4 inverters and DFF/latches	454
10.98	help - display help messages	454
10.99	hierarchy - check, expand and clean up design hierarchy	455
10.100	hilomap - technology mapping of constant hi- and/or lo-drivers	456
10.101	history - show last interactive commands	457
10.102	ice40_braminit - iCE40: perform SB_RAM40_4K initialization from file	457
10.103	ice40_dsp - iCE40: map multipliers	457
10.104	ice40_opt - iCE40: perform simple optimizations	458
10.105	ice40_wrapcarry - iCE40: wrap carries	458
10.106	insbuf - insert buffer cells for connected wires	458
10.107	internal_stats - print internal statistics	459
10.108	opadmap - technology mapping of i/o pads (or buffers)	459
10.109	ny - write design and metadata	460
10.110	json - write design in JSON format	460
10.111	keep_hierarchy - selectively add the keep_hierarchy attribute	461
10.112	attice_gsr - Lattice: handle GSR	461
10.113	icense - print license terms	461
10.114	og - print text and log files	462
10.115	ogger - set logger properties	463
10.116	s - list modules or objects in modules	464
10.117	tp - print longest topological path	464
10.118	lut2mux - convert \$lut to \$__MUX__	464
10.119	maccmap - mapping macc cells	464
10.120	memory - translate memories to basic cells	465
10.121	memory_bmux2rom - convert muxes to ROMs	465
10.122	memory_bram - map memories to block rams	465
10.123	memory_collect - creating multi-port memory cells	467
10.124	memory_dff - merge input/output DFFs into memory read ports	468
10.125	memory_libmap - map memories to cells	468
10.126	memory_map - translate multiport memories to basic cells	469
10.127	memory_memx - emulate vlog sim behavior for mem ports	469
10.128	memory_narrow - split up wide memory ports	470
10.129	memory_nordff - extract read port FFs from memories	470
10.130	memory_share - consolidate memory ports	470
10.131	memory_unpack - unpack multi-port memory cells	471
10.132	microchip_dffopt - MICROCHIP: optimize FF control signal usage	471
10.133	microchip_dsp - MICROCHIP: pack resources into DSPs	471
10.134	miter - automatically create a miter circuit	472
10.135	mutate - generate or apply design mutations	473
10.136	muxcover - cover trees of MUX cells with wider MUXes	474
10.137	muxpack - \$mux/\$pmux cascades to \$pmux	475
10.138	lutmap - map to LUTs of different sizes	475
10.139	nx_carry - NanoXplore: create carry cells	475
10.140	onehot - optimize \$eq cells for onehot signals	476
10.141	opt - perform simple optimizations	476
10.142	opt_clean - remove unused cells and wires	477
10.143	opt_demorgan - Optimize reductions with DeMorgan equivalents	477
10.144	opt_dff - perform DFF optimizations	477
10.145	opt_expr - perform const folding and simple expression rewriting	478
10.146	opt_ffinv - push inverters through FFs	479
10.147	opt_lut - optimize LUT cells	479
10.148	opt_lut_ins - discard unused LUT inputs	479
10.149	opt_mem - optimize memories	480
10.150	opt_mem_feedback - convert memory read-to-write port feedback paths to write enables	480

10.151	opt_mem_priority	- remove priority relations between write ports that can never collide	480
10.152	opt_mem_widen	- optimize memories where all ports are wide	480
10.153	opt_merge	- consolidate identical cells	481
10.154	opt_muxtree	- eliminate dead trees in multiplexer trees	481
10.155	opt_reduce	- simplify large MUXes and AND/OR gates	481
10.156	opt_share	- merge mutually exclusive cells of the same type that share an input signal	482
10.157	paramap	- renaming cell parameters	482
10.158	peepopt	- collection of peephole optimizers	483
10.159	plugin	- load and list loaded plugins	483
10.160	pmux2shiftx	- transform \$pmux cells to \$shiftx cells	484
10.161	pmuxtree	- transform \$pmux cells to trees of \$mux cells	484
10.162	portarcs	- derive port arcs for propagation delay	484
10.163	portlist	- list (top-level) ports	485
10.164	prep	- generic synthesis script	485
10.165	printattrs	- print attributes of selected objects	486
10.166	proc	- translate processes to netlists	487
10.167	proc_arst	- detect asynchronous resets	487
10.168	proc_clean	- remove empty parts of processes	488
10.169	proc_dff	- extract flip-flops from processes	488
10.170	proc_dlatch	- extract latches from processes	488
10.171	proc_init	- convert initial block to init attributes	489
10.172	proc_memwr	- extract memory writes from processes	489
10.173	proc_mux	- convert decision trees to multiplexers	489
10.174	proc_prune	- remove redundant assignments	489
10.175	proc_rmdead	- eliminate dead trees in decision trees	490
10.176	proc_rom	- convert switches to ROMs	490
10.177	qbfsat	- solve a 2QBF-SAT problem in the circuit	490
10.178	ql_bram_merge	- Infers QuickLogic k6n10f BRAM pairs that can operate independently	492
10.179	ql_bram_types	- Change TDP36K type to subtypes	492
10.180	ql_dsp_io_regs	- change types of QL_DSP2 depending on configuration	492
10.181	ql_dsp_macc	- infer QuickLogic multiplier-accumulator DSP cells	492
10.182	ql_dsp_simd	- merge QuickLogic K6N10f DSP pairs to operate in SIMD mode	493
10.183	ql_ioff	- Infer I/O FFs for qlf_k6n10f architecture	493
10.184	read	- load HDL designs	493
10.185	read_aiger	- read AIGER file	494
10.186	read_blif	- read BLIF file	495
10.187	read_json	- read JSON file	495
10.188	read_liberty	- read cells from liberty file	495
10.189	read_rtlil	- read modules from RTLIL file	496
10.190	read_verilog	- read modules from Verilog file	496
10.191	read_xaiger2	- (experimental) read XAIGER file	500
10.192	recover_names	- Execute a lossy mapping command and recover original netnames	500
10.193	rename	- rename object in the design	500
10.194	rmports	- remove module ports with no connections	502
10.195	sat	- solve a SAT problem in the circuit	502
10.196	scatter	- add additional intermediate nets	505
10.197	scc	- detect strongly connected components (logic loops)	505
10.198	scratchpad	- get/set values in the scratchpad	506
10.199	script	- execute commands from file or wire	507
10.200	select	- modify and view the list of selected objects	507
10.201	setattr	- set/unset attributes on objects	512
10.202	setenv	- set an environment variable	512
10.203	setparam	- set/unset parameters on objects	513
10.204	setundef	- replace undef values with defined constants	513

10.205	share - perform sat-based resource sharing	514
10.206	shell - enter interactive command mode	514
10.207	show - generate schematics using graphviz	515
10.208	shregmap - map shift registers	517
10.209	sim - simulate the circuit	518
10.210	simplemap - mapping simple coarse-grain cells	520
10.211	splice - create explicit splicing cells	520
10.212	splitcells - split up multi-bit cells	521
10.213	splitnets - split up multi-bit nets	522
10.214	sta - perform static timing analysis	522
10.215	stat - print some statistics	522
10.216	submod - moving part of a module to a new submodule	523
10.217	supercover - add hi/lo cover cells for each wire bit	524
10.218	synth - generic synthesis script	524
10.219	synth_achronix - synthesis for Achronix Speedster22i FPGAs	526
10.220	synth_anlogic - synthesis for Anlogic FPGAs	527
10.221	synth_coolrunner2 - synthesis for Xilinx Coolrunner-II CPLDs	529
10.222	synth_easic - synthesis for eASIC platform	531
10.223	synth_ecp5 - synthesis for ECP5 FPGAs	532
10.224	synth_efinix - synthesis for Efinix FPGAs	536
10.225	synth_fabulous - FABulous synthesis script	538
10.226	synth_gatamate - synthesis for Cologne Chip GateMate FPGAs	541
10.227	synth_gowin - synthesis for Gowin FPGAs	544
10.228	synth_greenpak4 - synthesis for GreenPAK4 FPGAs	546
10.229	synth_ice40 - synthesis for iCE40 FPGAs	548
10.230	synth_intel - synthesis for Intel (Altera) FPGAs	552
10.231	synth_intel_alm - synthesis for ALM-based Intel (Altera) FPGAs	554
10.232	synth_lattice - synthesis for Lattice FPGAs	557
10.233	synth_microchip - synthesis for Microchip FPGAs	560
10.234	synth_nanoxplore - synthesis for NanoXplore FPGAs	563
10.235	synth_nexus - synthesis for Lattice Nexus FPGAs	566
10.236	synth_quicklogic - Synthesis for QuickLogic FPGAs	569
10.237	synth_sf2 - synthesis for SmartFusion2 and IGLOO2 FPGAs	572
10.238	synth_xilinx - synthesis for Xilinx FPGAs	574
10.239	synthprop - synthesize SVA properties	578
10.240	tbl - execute a TCL script file	579
10.241	techmap - generic technology mapper	579
10.242	tee - redirect command output to file	583
10.243	test_abcloop - automatically test handling of loops in abc command	583
10.244	test_autotb - generate simple test benches	583
10.245	test_cell - automatically test the implementation of a cell type	584
10.246	test_generic - test the generic compute graph	585
10.247	test_pmgen - test pass for pmgen	586
10.248	order - print cells in topological order	586
10.249	race - redirect command output to file	587
10.250	tribuf - infer tri-state buffers	587
10.251	uniquify - create unique copies of modules	587
10.252	verific - load Verilog and VHDL designs using Verific	588
10.253	verilog_defaults - set default options for read_verilog	590
10.254	verilog_defines - define and undefine verilog defines	590
10.255	viz - visualize data flow graph	591
10.256	wbflip - flip the whitebox attribute	592
10.257	wrapcell - wrap individual cells into new modules	592
10.258	wreduce - reduce the word size of operations if possible	593

10.259	write_aiger - write design to AIGER file	593
10.260	write_aiger2 - (experimental) write design to AIGER file	594
10.261	write_blif - write design to BLIF file	595
10.262	write_btor - write design to BTOR file	596
10.263	write_cxxrtl - convert design to C++ RTL simulation	597
10.264	write_edif - write design to EDIF netlist file	601
10.265	write_file - write a text to a file	602
10.266	write_firrtl - write design to a FIRRTL file	603
10.267	write_functional_cxx - convert design to C++ using the functional backend	603
10.268	write_functional_rosette - Generate Rosette compatible Racket from Functional IR	603
10.269	write_functional_smt2 - Generate SMT-LIB from Functional IR	603
10.270	write_intersynth - write design to InterSynth netlist file	604
10.271	write_jny - generate design metadata	604
10.272	write_json - write design to a JSON file	605
10.273	write_rtlil - write design to RTLIL file	610
10.274	write_simplec - convert design to simple C code	610
10.275	write_smt2 - write design to SMT-LIBv2 file	610
10.276	write_smv - write design to SMV file	613
10.277	write_spice - write design to SPICE netlist file	614
10.278	write_table - write design as connectivity table	614
10.279	write_verilog - write design to Verilog file	615
10.280	write_xaiger - write design to XAIGER file	617
10.281	write_xaiger2 - (experimental) write module to XAIGER file	617
10.282	xilinx_dffopt - Xilinx: optimize FF control signal usage	618
10.283	xilinx_dsp - Xilinx: pack resources into DSPs	618
10.284	xilinx_srl - Xilinx shift register extraction	619
10.285	xprop - formal x propagation	619
10.286	zinit - add inverters so all FF are zero-initialized	620
Bibliography		621
Tag Index		623
Command Reference		627
Property Index		631
Internal cell reference		633

Yosys is an open source framework for RTL synthesis. To learn more about Yosys, see [What is Yosys](#). For a quick guide on how to get started using Yosys, check out [Getting started with Yosys](#). For the complete list of commands available, go to [commandindex](#).

Todo: look into command ref improvements

- Search bar with live drop down suggestions for matching on title / autocompleting commands
 - Scroll the left sidebar to the current location on page load
 - Also the formatting in pdf uses link formatting instead of code formatting
-

Todo: how does a filterlib rules-file work?

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/appendix/auxprogs.rst`, line 22.)

Todo: see if we can get the two hanging appnotes as lit references

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/bib.rst`, line 10.)

Todo: flip-flops with async load, `$_ALDFFE?_[NP]{2,3}_`

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/cell/gate_reg_ff.rst`, line 226.)

Todo: Add information about `$alu`, `$fa`, and `$lcu` cells.

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/cell/word_arith.rst`, line 4.)

Todo: Describe formal cells

`$check`, `$assert`, `$assume`, `$live`, `$fair`, `$cover`, `$equiv`, `$initstate`, `$anyconst`, `$anyseq`, `$anyinit`, `$allconst`, and `$allseq`.

Also `$ff` and `$_FF_` cells.

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/cell/word_formal.rst`, line 13.)

Todo: Describe `$fsm` cell

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/cell/word_fsm.rst`, line 4.)

Todo: *\$specify2*, *\$specify3*, and *\$specrule* cells.

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/cell/word_spec.rst`, line 4.)

Todo: Add information about *\$slice* and *\$concat* cells.

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/cell/word_wire.rst`, line 4.)

Todo: reconsider including the whole (~77 line) design like this

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/getting_started/example_synth.rst`, line 23.)

Todo: `fifo.v` description

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/getting_started/example_synth.rst`, line 31.)

Todo: consider a brief glossary for terms like `adff`

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/getting_started/example_synth.rst`, line 199.)

Todo: *hierarchy* failure modes

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/getting_started/example_synth.rst`, line 237.)

Todo: pending bugfix in *wreduce* and/or *opt_clean*

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/getting_started/example_synth.rst`, line 427.)

Todo: `ice40_dsp` is pmgen

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/getting_started/example_synth.rst`, line 501.)

Todo: look into command ref improvements

- Search bar with live drop down suggestions for matching on title / autocompleting commands
- Scroll the left sidebar to the current location on page load

- Also the formatting in pdf uses link formatting instead of code formatting
-

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/index.rst`, line 10.)

Todo: nextpnr for FPGAs, consider mentioning openlane, vpr, coriolis

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/introduction.rst`, line 76.)

Todo: Consider a less academic version of the History of Yosys

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/introduction.rst`, line 206.)

Todo: brief overview for the more scripting index

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/index.rst`, line 4.)

Todo: troubleshooting document(?)

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/index.rst`, line 6.)

Todo: interactive design opening text

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/interactive_investigation.rst`, line 4.)

Todo: merge into *Scripting in Yosys* show section

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/interactive_investigation.rst`, line 14.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/interactive_investigation.rst`, line 363.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/interactive_investigation.rst`, line 395.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/interactive_investigation.rst`, line 483.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/interactive_investigation.rst`, line 553.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/interactive_investigation.rst`, line 597.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/interactive_investigation.rst`, line 643.)

Todo: replace inline code?

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/interactive_investigation.rst`, line 696.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/interactive_investigation.rst`, line 721.)

Todo: include `read_verilog <<EOF`, also other methods of loading designs

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/load_design.rst`, line 8.)

Todo: more info on other `read_*` commands, also is this the first time we mention `verific`?

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/load_design.rst`, line 27.)

Todo: check text context

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/model_checking.rst`, line 4.)

Todo: add/expand supporting text

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/model_checking.rst`, line 25.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/model_checking.rst`, line 47.)

Todo: add/expand supporting text

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/model_checking.rst`, line 91.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/model_checking.rst`, line 107.)

Todo: reduce overlap with *Scripting in Yosys* select section

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/selections.rst`, line 10.)

Todo: pending discussion on whether rule ordering is a bug or a feature

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/selections.rst`, line 340.)

Todo: reflow for not presentation

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/more_scripting/selections.rst`, line 385.)

Todo: more about logic minimization & register balancing et al with ABC

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/abc.rst`, line 102.)

Todo: find a Liberty pygments style?

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/cell_libs.rst`, line 93.)

Todo: add/expand supporting text, also mention custom pattern matching and pmgen

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/extract.rst`, line 12.)

Todo: add/expand supporting text

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/extract.rst`, line 103.)

Todo: Make `memory_*` notes less quick

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/memory.rst`, line 18.)

Todo: describe `memory` images

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/memory.rst`, line 34.)

Todo: assorted enables, e.g. `cen`, `wen+ren`

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/memory.rst`, line 233.)

Todo: “outlines these optimizations” or “outlines *some..*”?

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/opt.rst`, line 7.)

Todo: unsure if this is too much detail and should be in *Yosys internals*

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/opt.rst`, line 27.)

Todo: `$_DFF_` isn’t a valid cell

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/opt.rst`, line 185.)

Todo: more on the other optimizations

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/opt.rst`, line 227.)

Todo: describe `proc` images

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/proc.rst`, line 29.)

Todo: comment on common `synth_*` options, like `-run`

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/synth.rst`, line 4.)

Todo: less academic, check text is coherent

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/techmap_synth.rst`, line 4.)

Todo: newer techmap libraries appear to be largely `.v` instead of `.lib`

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/using_yosys/synthesis/techmap_synth.rst`, line 76.)

Todo: check text is coherent

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/extending_yosys/extensions.rst`, line 7.)

Todo: update to use `/code_examples/extensions/test*.log`

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/extending_yosys/extensions.rst`, line 9.)

Todo: mention coding guide

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/extending_yosys/extensions.rst`, line 14.)

Todo: consider replacing inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/extending_yosys/extensions.rst`, line 44.)

Todo: add/expand supporting text

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/extending_yosys/extensions.rst`, line 62.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/extending_yosys/extensions.rst`, line 84.)

Todo: use `my_cmd.cc` literalincludes

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/extending_yosys/extensions.rst`, line 171.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/extending_yosys/extensions.rst`, line 213.)

Todo: replace inline code

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/extending_yosys/extensions.rst`, line 228.)

Todo: more about the included test suite and how to add tests

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/extending_yosys/test_suites.rst`, line 4.)

Todo: are unit tests currently working

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/extending_yosys/test_suites.rst`, line 27.)

Todo: less academic

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/flow/control_and_data.rst`, line 4.)

Todo: less academic

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/flow/overview.rst`, line 4.)

Todo: Synthesizing Verilog arrays

Add some information on the generation of `$memrd` and `$memwr` cells and how they are processed in the memory pass.

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/flow/verilog_frontend.rst`, line 647.)

Todo: Synthesizing parametric designs

Add some information on the `RTLIL::Module::derive()` method and how it is used to synthesize parametric modules via the hierarchy pass.

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/flow/verilog_frontend.rst`, line 653.)

Todo: less academic

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/index.rst`, line 6.)

Todo: add RISC-V core example

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/index.rst`, line 18.)

Todo: add/expand supporting text

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/techmap.rst`, line 25.)

Todo: how much of this is specific to the `read_verilog` and should be in *The Verilog and AST frontends*?

(The [original entry](#) is located in `/builddir/build/BUILD/yosys-0.51-build/yosys-c4b5190229616f7ebf8197f43990b4429de3e420/docs/source/yosys_internals/verilog.rst`, line 4.)

WHAT IS YOSYS

Yosys began as a BSc thesis project by Claire Wolf intended to support synthesis for a CGRA (coarse-grained reconfigurable architecture). It then expanded into more general infrastructure for research on synthesis.

Modern Yosys has full support for the synthesizable subset of Verilog-2005 and has been described as “the GCC of hardware synthesis.” Freely available and [open source](#), Yosys finds use across hobbyist and commercial applications as well as academic.

Note: Yosys is released under the ISC License:

A permissive license lets people do anything with your code with proper attribution and without warranty. The ISC license is functionally equivalent to the BSD 2-Clause and MIT licenses, removing some language that is no longer necessary.

Together with the place and route tool [nextpnr](#), Yosys can be used to program some FPGAs with a fully end-to-end open source flow (Lattice iCE40 and ECP5). It also does the synthesis portion for the [OpenLane flow](#), targeting the SkyWater 130nm open source PDK for fully open source ASIC design. Yosys can also do formal verification with backends for solver formats like [SMT2](#).

Yosys, and the accompanying Open Source EDA ecosystem, is currently maintained by [Yosys Headquarters](#), with many of the core developers employed by [YosysHQ GmbH](#). A commercial extension, [Tabby CAD Suite](#), includes the Verific frontend for industry-grade SystemVerilog and VHDL support, formal verification with SVA, and formal apps.



1.1 What you can do with Yosys

- Read and process (most of) modern Verilog-2005 code
- Perform all kinds of operations on netlist (RTL, Logic, Gate)
- Perform logic optimizations and gate mapping with ABC

1.1.1 Typical applications for Yosys

- Synthesis of final production designs
- Pre-production synthesis (trial runs before investing in other tools)
- Conversion of full-featured Verilog to simple Verilog
- Conversion of Verilog to other formats (BLIF, BTOR, etc)
- Demonstrating synthesis algorithms (e.g. for educational purposes)
- Framework for experimenting with new algorithms
- Framework for building custom flows (Not limited to synthesis but also formal verification, reverse engineering, ...)

1.1.2 Things you can't do

- Process high-level languages such as C/C++/SystemC
- Create physical layouts (place&route)
 - Check out [nextpnr](#) for that
- Rely on built-in syntax checking
 - Use an external tool like [verilator](#) instead

Todo: [nextpnr](#) for FPGAs, consider mentioning [openlane](#), [vpr](#), [coriolis](#)

1.2 The Yosys family

As mentioned above, [YosysHQ](#) maintains not just Yosys but an entire family of tools built around it. In no particular order:

SBY for formal verification

Yosys provides input parsing and conversion to the formats used by the solver engines. Yosys also provides a unified witness framework for providing cover traces and counter examples for engines which don't natively support this. [SBY source](#) | [SBY docs](#)

EQY for equivalence checking

In addition to input parsing and preparation, Yosys provides the plugin support enabling EQY to operate on designs directly. [EQY source](#) | [EQY docs](#)

MCY for mutation coverage

Yosys is used to read the source design, generate a list of possible mutations to maximise design coverage, and then perform selected mutations. [MCY source](#) | [MCY docs](#)

SCY for deep formal traces

Since SCY generates and runs SBY, Yosys provides the same utility for SCY as it does for SBY. Yosys additionally provides the trace concatenation needed for outputting the deep traces. [SCY source](#)

1.3 The original thesis abstract

The first version of the Yosys documentation was published as a bachelor thesis at the Vienna University of Technology [Wol13].

Abstract

Most of today's digital design is done in HDL code (mostly Verilog or VHDL) and with the help of HDL synthesis tools.

In special cases such as synthesis for coarse-grain cell libraries or when testing new synthesis algorithms it might be necessary to write a custom HDL synthesis tool or add new features to an existing one. In these cases the availability of a Free and Open Source (FOSS) synthesis tool that can be used as basis for custom tools would be helpful.

In the absence of such a tool, the Yosys Open SYnthesis Suite (Yosys) was developed. This document covers the design and implementation of this tool. At the moment the main focus of Yosys lies on the high-level aspects of digital synthesis. The pre-existing FOSS logic-synthesis tool ABC is used by Yosys to perform advanced gate-level optimizations.

An evaluation of Yosys based on real-world designs is included. It is shown that Yosys can be used as-is to synthesize such designs. The results produced by Yosys in this tests were successfully verified using formal verification and are comparable in quality to the results produced by a commercial synthesis tool.

Yosys is a Verilog HDL synthesis tool. This means that it takes a behavioural design description as input and generates an RTL, logical gate or physical gate level description of the design as output. Yosys' main strengths are behavioural and RTL synthesis. A wide range of commands (synthesis passes) exist within Yosys that can be used to perform a wide range of synthesis tasks within the domain of behavioural, rtl and logic synthesis. Yosys is designed to be extensible and therefore is a good basis for implementing custom synthesis tools for specialised tasks.

1.3.1 Benefits of open source HDL synthesis

- Cost (also applies to **free as in free beer** solutions):

Today the cost for a mask set in 180nm technology is far less than the cost for the design tools needed to design the mask layouts. Open Source ASIC flows are an important enabler for ASIC-level Open Source Hardware.

- Availability and Reproducibility:

If you are a researcher who is publishing, you want to use tools that everyone else can also use. Even if most universities have access to all major commercial tools, you usually do not have easy access to the version that was used in a research project a couple of years ago. With Open Source tools you can even release the source code of the tool you have used alongside your data.

- Framework:

Yosys is not only a tool. It is a framework that can be used as basis for other developments, so researchers and hackers alike do not need to re-invent the basic functionality. Extensibility was one of Yosys' design goals.



Fig. 1.1: Where Yosys exists in the layers of abstraction

- All-in-one:

Because of the framework characteristics of Yosys, an increasing number of features become available in one tool. Yosys not only can be used for circuit synthesis but also for formal equivalence checking, SAT solving, and for circuit analysis, to name just a few other application domains. With proprietary software one needs to learn a new tool for each of these applications.

- Educational Tool:

Proprietary synthesis tools are at times very secretive about their inner workings. They often are **black boxes**. Yosys is very open about its internals and it is easy to observe the different steps of synthesis.

1.3.2 History of Yosys

Todo: Consider a less academic version of the History of Yosys

A Hardware Description Language (HDL) is a computer language used to describe circuits. A HDL synthesis tool is a computer program that takes a formal description of a circuit written in an HDL as input and generates a netlist that implements the given circuit as output.

Currently the most widely used and supported HDLs for digital circuits are Verilog [A+02, A+06] and VHDL (VHSIC HDL, where VHSIC is an acronym for Very-High-Speed Integrated Circuits) [A+04, A+09]. Both HDLs are used for test and verification purposes as well as logic synthesis, resulting in a set of synthesizable and a set of non-synthesizable language features. In this document we only look at the synthesizable subset of the language features.

In recent work on heterogeneous coarse-grain reconfigurable logic [WGS+12] the need for a custom application-specific HDL synthesis tool emerged. It was soon realised that a synthesis tool that understood Verilog or VHDL would be preferred over a synthesis tool for a custom HDL. Given an existing Verilog or VHDL front end, the work for writing the necessary additional features and integrating them in an existing

tool can be estimated to be about the same as writing a new tool with support for a minimalistic custom HDL.

The proposed custom HDL synthesis tool should be licensed under a Free and Open Source Software (FOSS) licence. So an existing FOSS Verilog or VHDL synthesis tool would have been needed as basis to build upon. The main advantages of choosing Verilog or VHDL is the ability to synthesize existing HDL code and to mitigate the requirement for circuit-designers to learn a new language. In order to take full advantage of any existing FOSS Verilog or VHDL tool, such a tool would have to provide a feature-complete implementation of the synthesizable HDL subset.

Basic RTL synthesis is a well understood field [HS96]. Lexing, parsing and processing of computer languages [ASU86] is a thoroughly researched field. All the information required to write such tools has been openly available for a long time, and it is therefore likely that a FOSS HDL synthesis tool with a feature-complete Verilog or VHDL front end must exist which can be used as a basis for a custom RTL synthesis tool.

Due to the author's preference for Verilog over VHDL it was decided early on to go for Verilog instead of VHDL¹. So the existing FOSS Verilog synthesis tools were evaluated. The results of this evaluation are utterly devastating. Therefore a completely new Verilog synthesis tool was implemented and is recommended as basis for custom synthesis tools. This is the tool that is discussed in this document.

¹ A quick investigation into FOSS VHDL tools yielded similar grim results for FOSS VHDL synthesis tools.

GETTING STARTED WITH YOSYS

This section covers how to get started with Yosys, from installation to a guided walkthrough of synthesizing a design for hardware, and finishing with an introduction to writing re-usable Yosys scripts.

2.1 Installation

This document will guide you through the process of installing Yosys.

2.1.1 CAD suite(s)

Yosys is part of the [Tabby CAD Suite](#) and the [OSS CAD Suite](#)! The easiest way to use yosys is to install the binary software suite, which contains all required dependencies and related tools.

- [Contact YosysHQ](#) for a [Tabby CAD Suite](#) Evaluation License and download link
- OR go to <https://github.com/YosysHQ/oss-cad-suite-build/releases> to download the free OSS CAD Suite
- Follow the [Install Instructions on GitHub](#)

Make sure to get a [Tabby CAD Suite](#) Evaluation License if you need features such as industry-grade SystemVerilog and VHDL parsers!

For more information about the difference between [Tabby CAD Suite](#) and the [OSS CAD Suite](#), please visit <https://www.yosyshq.com/tabby-cad-datasheet>

Many Linux distributions also provide Yosys binaries, some more up to date than others. Check with your package manager!

Targeted architectures

The [OSS CAD Suite](#) releases [nightly builds](#) for the following architectures:

2.1.2 Building from source

The Yosys source files can be obtained from the [YosysHQ/Yosys git repository](https://github.com/YosysHQ/yosys). ABC and some of the other libraries used are included as git submodules. To clone these submodules at the same time, use e.g.:

```
git clone --recurse-submodules https://github.com/YosysHQ/yosys.git # ..or..
git clone https://github.com/YosysHQ/yosys.git
cd yosys
git submodule update --init --recursive
```

Note: As of Yosys v0.47, releases include a `yosys.tar.gz` file which includes all source code and all sub-modules in a single archive. This can be used as an alternative which does not rely on `git`.

Supported platforms

The following platforms are supported and regularly tested:

- Linux
- macOS

Other platforms which may work, but instructions may not be up to date and are not regularly tested:

- FreeBSD
- WSL
- Windows with (e.g.) Cygwin

Build prerequisites

A C++ compiler with C++17 support is required as well as some standard tools such as GNU Flex, GNU Bison, Make and Python. Some additional tools: readline, libffi, Tcl and zlib; are optional but enabled by default (see `ENABLE_*` settings in Makefile). Graphviz and Xdot are used by the `show` command to display schematics.

Installing all prerequisites for Ubuntu 20.04:

```
sudo apt-get install gperf build-essential bison flex \
  libreadline-dev gawk tcl-dev libffi-dev git graphviz \
  xdot pkg-config python3 libboost-system-dev \
  libboost-python-dev libboost-filesystem-dev zlib1g-dev
```

Installing all prerequisites for macOS 13 (with Homebrew):

```
brew tap Homebrew/bundle && brew bundle
```

or MacPorts:

```
sudo port install bison flex readline gawk libffi graphviz \
  pkgconfig python311 boost zlib tcl
```

On FreeBSD use the following command to install all prerequisites:

```
pkg install bison flex readline gawk libffi graphviz \
  pkgconf python311 tcl-wrapper boost-libs
```

Note: On FreeBSD system use `gmake` instead of `make`. To run tests use: `MAKE=gmake CXX=cxx CC=cc gmake test`

For Cygwin use the following command to install all prerequisites, or select these additional packages:

```
setup-x86_64.exe -q --packages=bison,flex,gcc-core,gcc-g++,git,libffi-devel,libreadline-
  devel,make,pkg-config,python3,tcl-devel,boost-build,zlib-devel
```

Warning: As of this writing, Cygwin only supports up to Python 3.9.16 while the minimum required version of Python is 3.11. This means that Cygwin is not compatible with many of the Python-based frontends. While this does not currently prevent Yosys itself from working, no guarantees are made for continued support. It is instead recommended to use Windows Subsystem for Linux (WSL) and follow the instructions for Ubuntu.

Build configuration

The Yosys build is based solely on Makefiles, and uses a number of variables which influence the build process. The recommended method for configuring builds is with a `Makefile.conf` file in the root `yosys` directory. The following commands will clean the directory and provide an initial configuration file:

```
make config-clang    # ..or..
make config-gcc
```

Check the root Makefile to see what other configuration targets are available. Other variables can then be added to the `Makefile.conf` as needed, for example:

```
echo "ENABLE_ZLIB := 0" >> Makefile.conf
```

Using one of these targets will set the `CONFIG` variable to something other than `none`, and will override the environment variable for `CXX`. To use a different compiler than the default when building, use:

```
make CXX=$CXX        # ..or..
make CXX="g++-11"
```

Note: Setting the compiler in this way will prevent some other options such as `ENABLE_CCACHE` from working as expected.

If you have clang, and (a compatible version of) `ld.lld` available in `PATH`, it's recommended to speed up incremental builds with lld by enabling LTO with `ENABLE_LTO=1`. On macOS, LTO requires using clang from homebrew rather than clang from xcode. For example:

```
make ENABLE_LTO=1 CXX=$(brew --prefix)/opt/llvm/bin/clang++
```

By default, building (and installing) `yosys` will build (and install) `ABC`, using `yosys-abc` as the executable name. To use an existing ABC executable instead, set the `ABCEXTERNAL` make variable to point to the desired executable.

Running the build system

From the root `yosys` directory, call the following commands:

```
make
sudo make install
```

To use a separate (out-of-tree) build directory, provide a path to the Makefile.

```
mkdir build; cd build
make -f ../Makefile
```

Out-of-tree builds require a clean source tree.

See also:

Refer to *Testing Yosys* for details on testing Yosys once compiled.

2.1.3 Source tree and build system

The Yosys source tree is organized into the following top-level directories:

backends/

This directory contains a subdirectory for each of the backend modules.

docs/

Contains the source for this documentation, including images and sample code.

examples/

Contains example code for using Yosys with some other tools including a demo of the Yosys Python api, and synthesizing for various toolchains such as Intel and Anlogic.

frontends/

This directory contains a subdirectory for each of the frontend modules.

kernel/

This directory contains all the core functionality of Yosys. This includes the functions and definitions for working with the RTLIL data structures (`rtlil.h/cc`), the `main()` function (`driver.cc`), the internal framework for generating log messages (`log.h/cc`), the internal framework for registering and calling passes (`register.h/cc`), some core commands that are not really passes (`select.cc`, `show.cc`, ...) and a couple of other small utility libraries.

libs/

Libraries packaged with Yosys builds are contained in this folder. See *Auxiliary libraries*.

misc/

Other miscellany which doesn't fit anywhere else.

passes/

This directory contains a subdirectory for each pass or group of passes. For example as of this writing the directory `passes/hierarchy/` contains the code for three passes: *hierarchy*, *submod*, and *uniquify*.

techlibs/

This directory contains simulation models and standard implementations for the cells from the internal cell library.

tests/

This directory contains the suite of unit tests and regression tests used by Yosys. See *Testing Yosys*.

The top-level Makefile includes `frontends/*/Makefile.inc`, `passes/*/Makefile.inc` and `backends/*/Makefile.inc`. So when extending Yosys it is enough to create a new directory in `frontends/`, `passes/` or `backends/` with your sources and a `Makefile.inc`. The Yosys kernel automatically detects all commands linked with Yosys. So it is not needed to add additional commands to a central list of commands.

Good starting points for reading example source code to learn how to write passes are `passes/opt/opt_dff.cc` and `passes/opt/opt_merge.cc`.

Users of the Qt Creator IDE can generate a QT Creator project file using `make qtcreator`. Users of the Eclipse IDE can use the “Makefile Project with Existing Code” project type in the Eclipse “New Project” dialog (only available after the CDT plugin has been installed) to create an Eclipse project in order to programming extensions to Yosys or just browse the Yosys code base.

2.2 Synthesis starter

This page will be a guided walkthrough of the prepackaged iCE40 FPGA synthesis script - `synth_ice40`. We will take a simple design through each step, looking at the commands being called and what they do to the design. While `synth_ice40` is specific to the iCE40 platform, most of the operations we will be discussing are common across the majority of FPGA synthesis scripts. Thus, this document will provide a good foundational understanding of how synthesis in Yosys is performed, regardless of the actual architecture being used.

See also:

Advanced usage docs for *Synth commands*

2.2.1 Demo design

First, let’s quickly look at the design we’ll be synthesizing:

Todo: reconsider including the whole (~77 line) design like this

Listing 2.1: `fifo.v`

```

1 // address generator/counter
2 module addr_gen
3 #(   parameter MAX_DATA=256,
4       localparam AWIDTH = $clog2(MAX_DATA)
5 ) (   input en, clk, rst,
6       output reg [AWIDTH-1:0] addr
7 );
8     initial addr = 0;
9
10    // async reset
11    // increment address when enabled
12    always @(posedge clk or posedge rst)
13        if (rst)
14            addr <= 0;
15        else if (en) begin
16            if ({'0, addr} == MAX_DATA-1)
17                addr <= 0;

```

(continues on next page)

(continued from previous page)

```

18         else
19             addr <= addr + 1;
20         end
21     endmodule //addr_gen
22
23     // Define our top level fifo entity
24     module fifo
25     #( parameter MAX_DATA=256,
26         localparam AWIDTH = $clog2(MAX_DATA)
27     ) ( input wen, ren, clk, rst,
28         input [7:0] wdata,
29         output reg [7:0] rdata,
30         output reg [AWIDTH:0] count
31     );
32
33     // fifo storage
34     // sync read before write
35     wire [AWIDTH-1:0] waddr, raddr;
36     reg [7:0] data [MAX_DATA-1:0];
37     always @(posedge clk) begin
38         if (wen)
39             data[waddr] <= wdata;
40         rdata <= data[raddr];
41     end // storage
42
43     // addr_gen for both write and read addresses
44     addr_gen #(.MAX_DATA(MAX_DATA))
45     fifo_writer (
46         .en      (wen),
47         .clk      (clk),
48         .rst      (rst),
49         .addr     (waddr)
50     );
51
52     addr_gen #(.MAX_DATA(MAX_DATA))
53     fifo_reader (
54         .en      (ren),
55         .clk      (clk),
56         .rst      (rst),
57         .addr     (raddr)
58     );
59
60     // status signals
61     initial count = 0;
62
63     always @(posedge clk or posedge rst) begin
64         if (rst)
65             count <= 0;
66         else if (wen && !ren)
67             count <= count + 1;
68         else if (ren && !wen)
69             count <= count - 1;
70     end

```

(continues on next page)

(continued from previous page)

```

70
71 endmodule

```

Todo: fifo.v description

While the open source *read_verilog* frontend generally does a pretty good job at processing valid Verilog input, it does not provide very good error handling or reporting. Using an external tool such as *verilator* before running Yosys is highly recommended. We can quickly check the Verilog syntax of our design by calling *verilator --lint-only fifo.v*.

2.2.2 Loading the design

Let's load the design into Yosys. From the command line, we can call *yosys fifo.v*. This will open an interactive Yosys shell session and immediately parse the code from *fifo.v* and convert it into an Abstract Syntax Tree (AST). If you are interested in how this happens, there is more information in the document, *The Verilog and AST frontends*. For now, suffice it to say that we do this to simplify further processing of the design. You should see something like the following:

```

$ yosys fifo.v

-- Parsing `fifo.v' using frontend ` -vlog2k' --

1. Executing Verilog-2005 frontend: fifo.v
Parsing Verilog input from `fifo.v' to AST representation.
Storing AST representation for module `$abstract\addr_gen'.
Storing AST representation for module `$abstract\fifo'.
Successfully finished Verilog frontend.

```

See also:

Advanced usage docs for *Loading a design*

2.2.3 Elaboration

Now that we are in the interactive shell, we can call Yosys commands directly. Our overall goal is to call *synth_ice40 -top fifo*, but for now we can run each of the commands individually for a better sense of how each part contributes to the flow. We will also start with just a single module; *addr_gen*.

At the bottom of the *help* output for *synth_ice40* is the complete list of commands called by this script. Let's start with the section labeled *begin*:

Listing 2.2: *begin* section

```

read_verilog -D ICE40_HX -lib -specify +/ice40/cells_sim.v
hierarchy -check -top <top>
proc

```

read_verilog -D ICE40_HX -lib -specify +/ice40/cells_sim.v loads the iCE40 cell models which allows us to include platform specific IP blocks in our design. PLLs are a common example of this, where we might need to reference *SB_PLL40_CORE* directly rather than being able to rely on mapping passes later.

Since our simple design doesn't use any of these IP blocks, we can skip this command for now. Because these cell models will also be needed once we start mapping to hardware we will still need to load them later.

Note: `+/` is a dynamic reference to the Yosys `share` directory. By default, this is `/usr/local/share/yosys`. If using a locally built version of Yosys from the source directory, this will be the `share` folder in the same directory.

The `addr_gen` module

Since we're just getting started, let's instead begin with `hierarchy -top addr_gen`. This command declares that the top level module is `addr_gen`, and everything else can be discarded.

Listing 2.3: `addr_gen` module source

```
2 module addr_gen
3   #( parameter MAX_DATA=256,
4       localparam AWIDTH = $clog2(MAX_DATA)
5   ) ( input en, clk, rst,
6       output reg [AWIDTH-1:0] addr
7   );
8       initial addr = 0;
9
10      // async reset
11      // increment address when enabled
12      always @(posedge clk or posedge rst)
13          if (rst)
14              addr <= 0;
15          else if (en) begin
16              if ({'0, addr} == MAX_DATA-1)
17                  addr <= 0;
18              else
19                  addr <= addr + 1;
20          end
21 endmodule //addr_gen
```

Note: `hierarchy` should always be the first command after the design has been read. By specifying the top module, `hierarchy` will also set the `(* top *)` attribute on it. This is used by other commands that need to know which module is the top.

Listing 2.4: `hierarchy -top addr_gen` output

```
yosys> hierarchy -top addr_gen

2. Executing HIERARCHY pass (managing design hierarchy).

3. Executing AST frontend in derive mode using pre-parsed AST for module `addr_gen'.
Generating RTLIL representation for module `addr_gen'.

3.1. Analyzing design hierarchy..
Top module:  addr_gen
```

(continues on next page)

(continued from previous page)

```

3.2. Analyzing design hierarchy..
Top module:  \addr_gen
Removing unused module `$_abstract\fifo'.
Removing unused module `$_abstract\addr_gen'.
Removed 2 unused modules.

```

Our `addr_gen` circuit now looks like this:

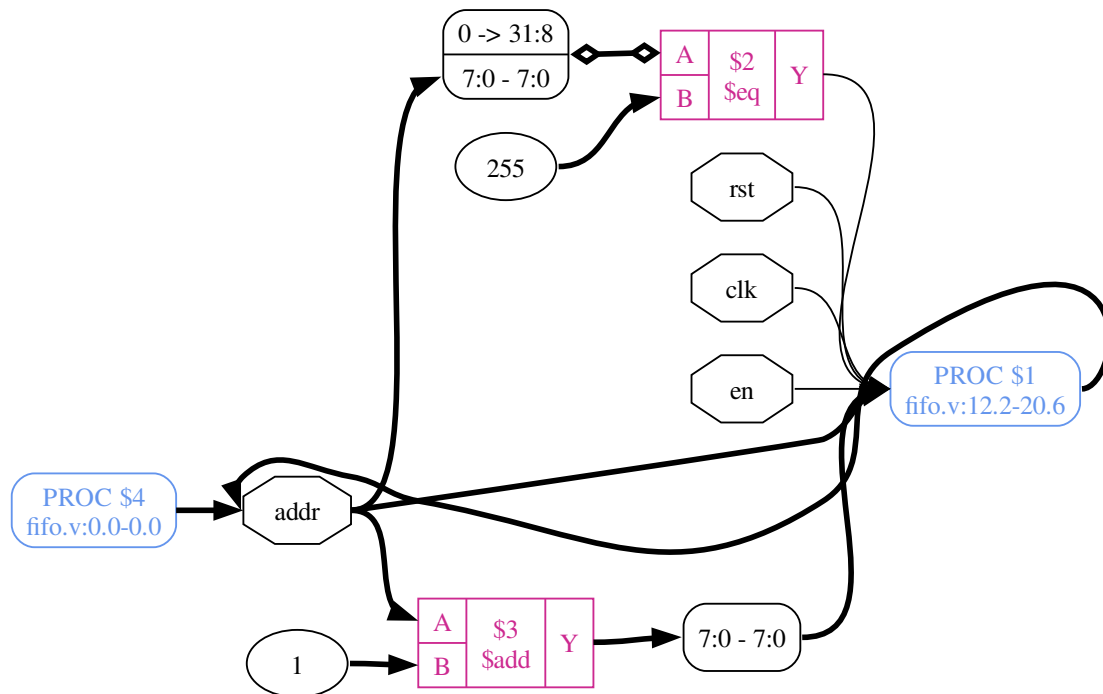


Fig. 2.1: `addr_gen` module after *hierarchy*

Simple operations like `addr + 1` and `addr == MAX_DATA-1` can be extracted from our `always @` block in *addr_gen module source*. This gives us the highlighted `$add` and `$eq` cells we see. But control logic (like the `if .. else`) and memory elements (like the `addr <= 0`) are not so straightforward. These get put into “processes”, shown in the schematic as PROC. Note how the second line refers to the line numbers of the start/end of the corresponding `always @` block. In the case of an `initial` block, we instead see the PROC referring to line 0.

To handle these, let us now introduce the next command: *proc - translate processes to netlists*. `proc` is a macro command like `synth_ice40`. Rather than modifying the design directly, it instead calls a series of other commands. In the case of `proc`, these sub-commands work to convert the behavioral logic of processes into multiplexers and registers. Let’s see what happens when we run it. For now, we will call `proc -noopt` to prevent some automatic optimizations which would normally happen.

There are now a few new cells from our `always @`, which have been highlighted. The `if` statements are now modeled with `$mux` cells, while the register uses an `$adff` cell. If we look at the terminal output we can

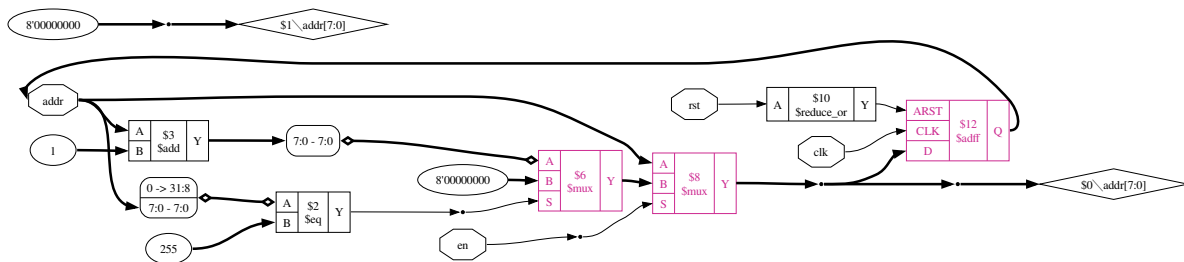


Fig. 2.2: addr_gen module after proc -noopt

also see all of the different `proc_*` commands being called. We will look at each of these in more detail in *Converting process blocks*.

Notice how in the top left of *addr_gen module after proc -noopt* we have a floating wire, generated from the initial assignment of 0 to the `addr` wire. However, this initial assignment is not synthesizable, so this will need to be cleaned up before we can generate the physical hardware. We can do this now by calling `clean`. We're also going to call `opt_expr` now, which would normally be called at the end of `proc`. We can call both commands at the same time by separating them with a colon and space: `opt_expr; clean`.

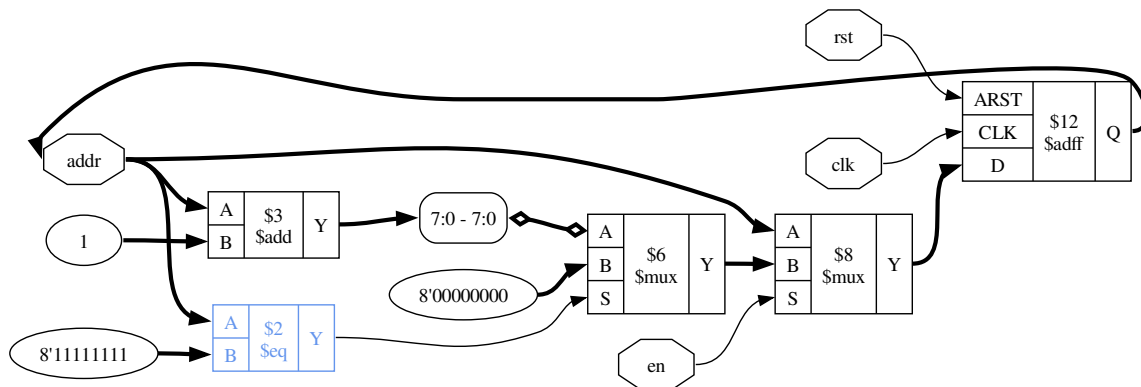


Fig. 2.3: addr_gen module after opt_expr; clean

You may also notice that the highlighted `$eq` cell input of 255 has changed to `8'11111111`. Constant values are presented in the format `<bit_width>'<bits>`, with 32-bit values instead using the decimal number. This indicates that the constant input has been reduced from 32-bit wide to 8-bit wide. This is a side-effect of running `opt_expr`, which performs constant folding and simple expression rewriting. For more on why this happens, refer to *Optimization passes* and the *section on opt_expr*.

Note: `clean - remove unused cells and wires` can also be called with two semicolons after any command, for example we could have called `opt_expr;;` instead of `opt_expr; clean`. You may notice some scripts will end each line with `;;`. It is beneficial to run `clean` before inspecting intermediate products to remove disconnected parts of the circuit which have been left over, and in some cases can reduce the processing required in subsequent commands.

Advanced usage docs for

- ## The full example

Note that if we tried to run this command now then we would get an error. This is because we already removed all of the modules other than `addr_gen`. We could restart our shell session, but instead let's use two new commands:

- Listing 2.5: reloading `fifo.v` and running `hierarchy -check -top fifo`

```
yosys> design -reset
```

```
yosys> read_verilog fifo.v
```

11. Executing Verilog-2005 frontend: fifo.v
Parsing Verilog input from `fifo.v' to AST representation.
Generating RTLIL representation for module `\addr_gen'.
Generating RTLIL representation for module `\fifo'.
Successfully finished Verilog frontend.

```
yosys> hierarchy -check -top fifo
```

12. Executing HIERARCHY pass (managing design hierarchy).

12.1. Analyzing design hierarchy..
Top module: \fifo
Used module: \addr_gen
Parameter \MAX_DATA = 256

12.2. Executing AST frontend in derive mode using pre-parsed AST for module `\addr_gen'.
Parameter \MAX_DATA = 256
Generating RTLIL representation for module `\$paramod\addr_gen\MAX_DATA=s32'
`<->'0000000000000000000000000100000000'`.
Parameter \MAX_DATA = 256
Found cached RTLIL representation for module `\$paramod\addr_gen\MAX_DATA=s32'
`<->'0000000000000000000000000100000000'`.

2.2. Synthesis starter

(continued from previous page)

```
12.3. Analyzing design hierarchy..
Top module:  \fifo
Used module:      $paramod\addr_gen\MAX_DATA=s32'000000000000000000000000100000000

12.4. Analyzing design hierarchy..
Top module:  \fifo
Used module:      $paramod\addr_gen\MAX_DATA=s32'000000000000000000000000100000000
Removing unused module  `\addr_gen'.
Removed 1 unused modules.
```

Notice how this time we didn't see any of those `$abstract` modules? That's because when we ran `yosys fifo.v`, the first command Yosys called was `read_verilog -defer fifo.v`. The `-defer` option there tells `read_verilog` only read the abstract syntax tree and defer actual compilation to a later `hierarchy` command. This is useful in cases where the default parameters of modules yield invalid code which is not synthesizable. This is why Yosys defers compilation automatically and is one of the reasons why `hierarchy` should always be the first command after loading the design. If we know that our design won't run into this issue, we can skip the `-defer`.

Todo: *hierarchy* failure modes

Note: The number before a command's output increments with each command run. Don't worry if your numbers don't match ours! The output you are seeing comes from the same script that was used to generate the images in this document, included in the source as `fifo.js`. There are extra commands being run which you don't see, but feel free to try them yourself, or play around with different commands. You can always start over with a clean slate by calling `exit` or hitting `ctrl+d` (i.e. EOF) and re-launching the Yosys interactive terminal. `ctrl+c` (i.e. SIGINT) will also end the terminal session but will return an error code rather than exiting gracefully.

We can also run *proc* now to finish off the full *begin section*. Because the design schematic is quite large, we will be showing just the data path for the `rdata` output. If you would like to see the entire design for yourself, you can do so with *show - generate schematics using graphviz*. Note that the *show* command only works with a single module, so you may need to call it with `show fifo`. *Displaying schematics* section in *Scripting in Yosys* has more on how to use *show*.

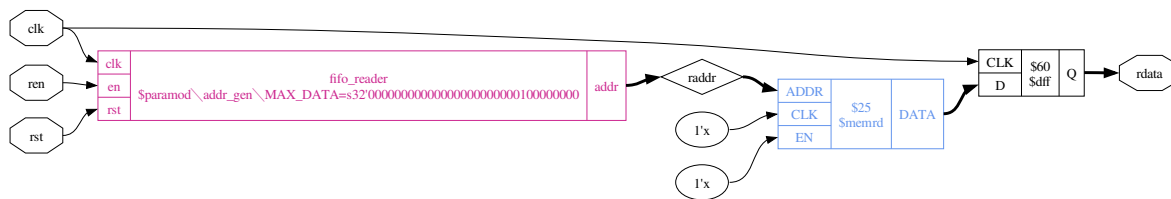


Fig. 2.4: rdata output after *proc*

The highlighted `fifo_reader` block contains an instance of the *addr_gen module after proc -noopt* that we looked at earlier. Notice how the type is shown as `$paramod\\addr_gen\\MAX_DATA=s32'...`. This is a “parametric module”: an instance of the `addr_gen` module with the `MAX_DATA` parameter set to the given value.

See also:

renamed to `fifo_reader.addr` and merged with the `raddr` wire feeding into the `$memrd` cell. This wire merging happened during the call to `clean` which we can see in the *output of flatten*;;.

Note: `flatten` and `clean` would normally be combined into a single `yosys> flatten;;` output, but they appear separately here as a side effect of using `echo` for generating the terminal style output.

Depending on the target architecture, this stage of synthesis might also see commands such as `tribuf` with the `-logic` option and `deminout`. These remove tristate and inout constructs respectively, replacing them with logic suitable for mapping to an FPGA. Since we do not have any such constructs in our example running these commands does not change our design.

2.2.5 The coarse-grain representation

At this stage, the design is in coarse-grain representation. It still looks recognizable, and cells are word-level operators with parametrizable width. This is the stage of synthesis where we do things like const propagation, expression rewriting, and trimming unused parts of wires.

This is also where we convert our FSMs and hard blocks like DSPs or memories. Such elements have to be inferred from patterns in the design and there are special passes for each. Detection of these patterns can also be affected by optimizations and other transformations done previously.

Note: While the iCE40 flow had a *flatten section* and put `proc` in the *begin section*, some synthesis scripts will instead include these in this section.

Part 1

In the iCE40 flow, we start with the following commands:

Listing 2.8: `coarse` section (part 1)

```
opt_expr
opt_clean
check
opt -nodffe -nosdff
fsm
opt
```

We've already come across `opt_expr`, and `opt_clean` is the same as `clean` but with more verbose output. The `check` pass identifies a few obvious problems which will cause errors later. Calling it here lets us fail faster rather than wasting time on something we know is impossible.

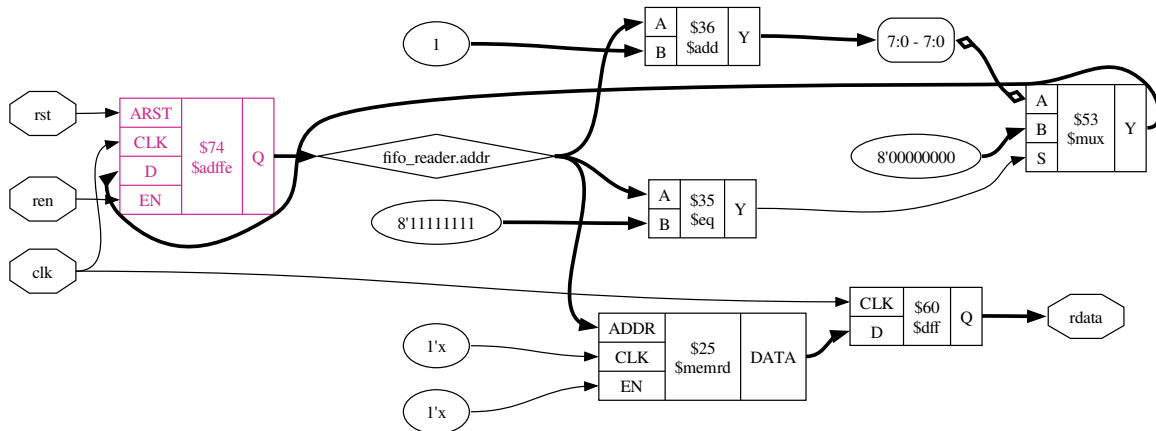
Next up is `opt -nodffe -nosdff` performing a set of simple optimizations on the design. This command also ensures that only a specific subset of FF types are included, in preparation for the next command: `fsm` - *extract and optimize finite state machines*. Both `opt` and `fsm` are macro commands which are explored in more detail in *Optimization passes* and *FSM handling* respectively.

Up until now, the data path for `rdata` has remained the same since *rdata output after flatten*;;. However the next call to `opt` does cause a change. Specifically, the call to `opt_dff` without the `-nodffe -nosdff` options is able to fold one of the `$mux` cells into the `$adff` to form an `$adffe` cell; highlighted below:

Listing 2.9: output of *opt_dff*

```
yosys> opt_dff
```

```
17. Executing OPT_DFF pass (perform DFF optimizations).
Adding EN signal on $procdff$59 ($adff) from module fifo (D = $0\count[8:0], Q = \count).
Adding EN signal on $flatten\fifo_writer.$procdff$66 ($adff) from module fifo (D =
↳ $flatten\fifo_writer.$procmux$53_Y, Q = \fifo_writer.addr).
Adding EN signal on $flatten\fifo_reader.$procdff$66 ($adff) from module fifo (D =
↳ $flatten\fifo_reader.$procmux$53_Y, Q = \fifo_reader.addr).
```

Fig. 2.6: rdata output after *opt_dff*

See also:

Advanced usage docs for

- *FSM handling*
- *Optimization passes*

Part 2

The next group of commands performs a series of optimizations:

Listing 2.10: coarse section (part 2)

```
wreduce
peepopt
opt_clean
share
techmap -map +/cmp2lut.v -D LUT_WIDTH=4
opt_expr
opt_clean
memory_dff [-no-rw-check]
```

First up is *wreduce* - *reduce the word size of operations if possible*. If we run this we get the following:

Listing 2.11: output of *wreduce*

```
yosys> wreduce

19. Executing WREDUCE pass (reducing word size of cells).
Removed top 31 bits (of 32) from port B of cell fifo.$add$fifo.v:66$29 ($add).
Removed top 23 bits (of 32) from port Y of cell fifo.$add$fifo.v:66$29 ($add).
Removed top 31 bits (of 32) from port B of cell fifo.$sub$fifo.v:68$32 ($sub).
Removed top 23 bits (of 32) from port Y of cell fifo.$sub$fifo.v:68$32 ($sub).
Removed top 1 bits (of 2) from port B of cell fifo.$auto$opt_dff.cc:195:make_patterns_
↳logic$72 ($ne).
Removed cell fifo.$flatten\fifo_writer.$procmux$55 ($mux).
Removed top 31 bits (of 32) from port B of cell fifo.$flatten\fifo_writer.$add$fifo.v:19
↳$36 ($add).
Removed top 24 bits (of 32) from port Y of cell fifo.$flatten\fifo_writer.$add$fifo.v:19
↳$36 ($add).
Removed cell fifo.$flatten\fifo_reader.$procmux$55 ($mux).
Removed top 31 bits (of 32) from port B of cell fifo.$flatten\fifo_reader.$add$fifo.v:19
↳$36 ($add).
Removed top 24 bits (of 32) from port Y of cell fifo.$flatten\fifo_reader.$add$fifo.v:19
↳$36 ($add).
Removed top 23 bits (of 32) from wire fifo.$add$fifo.v:66$29_Y.
Removed top 24 bits (of 32) from wire fifo.$flatten\fifo_reader.$add$fifo.v:19$36_Y.
Removed top 24 bits (of 32) from wire fifo.$flatten\fifo_writer.$add$fifo.v:19$36_Y.

yosys> show -notitle -format dot -prefix rdata_wreduce o:rdata %ci*

20. Generating Graphviz representation of design.
Writing dot description to `rdata_wreduce.dot'.
Dumping selected parts of module fifo to page 1.

yosys> opt_clean

21. Executing OPT_CLEAN pass (remove unused cells and wires).
Finding unused cells or wires in module \fifo..
Removed 0 unused cells and 5 unused wires.
<suppressed ~1 debug messages>

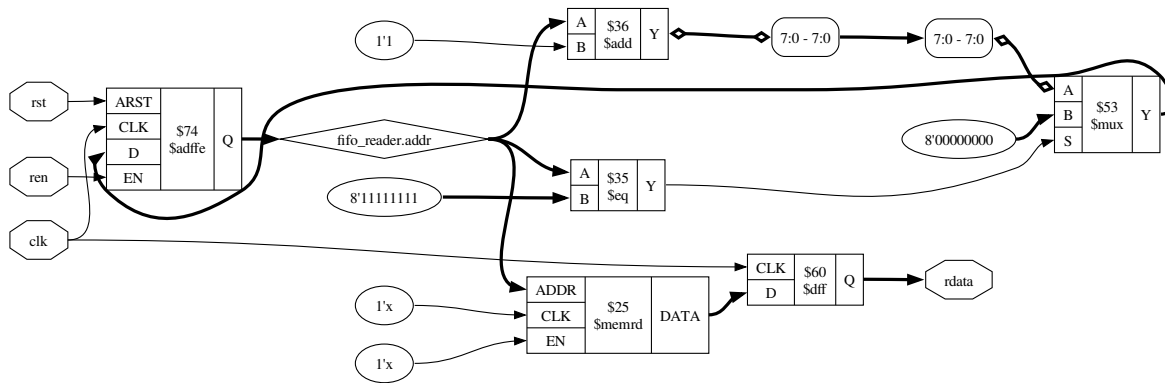
yosys> memory_dff

22. Executing MEMORY_DFF pass (merging $dff cells to $memrd).
Checking read port `data'[0] in module `fifo': merging output FF to cell.
Write port 0: non-transparent.
```

Looking at the data path for `rdata`, the most relevant of these width reductions are the ones affecting `fifo`. `$flatten\fifo_reader.$add$fifo.v`. That is the `$add` cell incrementing the `fifo_reader` address. We can look at the schematic and see the output of that cell has now changed.

Todo: pending bugfix in *wreduce* and/or *opt_clean*

The next two (new) commands are *peepopt* - *collection of peephole optimizers* and *share* - *perform sat-based resource sharing*. Neither of these affect our design, and they're explored in more detail in *Optimization*

Fig. 2.7: rdata output after *wreduce*

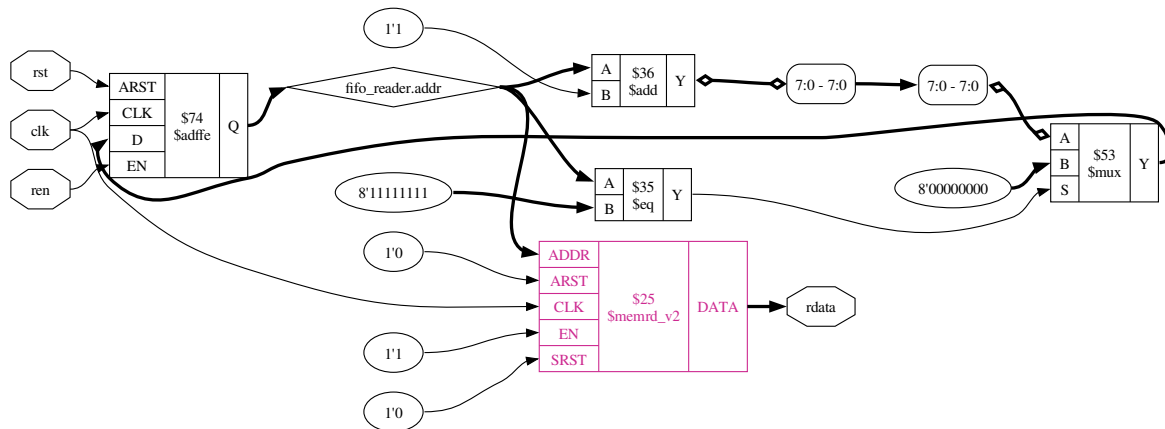
passes, so let's skip over them. `techmap -map +/cmp2lut.v -D LUT_WIDTH=4` optimizes certain comparison operators by converting them to LUTs instead. The usage of *techmap* is explored more in *Technology mapping*.

Our next command to run is *memory_dff* - merge input/output DFFs into memory read ports.

Listing 2.12: output of *memory_dff*

```
yosys> memory_dff
```

```
22. Executing MEMORY_DFF pass (merging $dff cells to $memrd).
Checking read port `data'[0] in module `fifo': merging output FF to cell.
Write port 0: non-transparent.
```

Fig. 2.8: rdata output after *memory_dff*

As the title suggests, *memory_dff* has merged the output *\$dff* into the *\$memrd* cell and converted it to a *\$memrd_v2* (highlighted). This has also connected the CLK port to the clk input as it is now a synchronous memory read with appropriate enable (EN=1'1) and reset (ARST=1'0 and SRST=1'0) inputs.

See also:

Advanced usage docs for

- [Optimization passes](#)
- [Technology mapping](#)
- [Memory handling](#)

Part 3

The third part of the `synth_ice40` flow is a series of commands for mapping to DSPs. By default, the iCE40 flow will not map to the hardware DSP blocks and will only be performed if called with the `-dsp` flag: `synth_ice40 -dsp`. While our example has nothing that could be mapped to DSPs we can still take a quick look at the commands here and describe what they do.

Listing 2.13: `coarse` section (part 3)

```
wreduce t:$mul
techmap -map +/mul2dsp.v -map +/ice40/dsp_map.v -D DSP_A_MAXWIDTH=16 -D DSP_B_
↳MAXWIDTH=16 -D DSP_A_MINWIDTH=2 -D DSP_B_MINWIDTH=2 -D DSP_Y_MINWIDTH=11 -D DSP_NAME=$_
↳_MUL16X16 (if -dsp)
select a:mul2dsp (if -dsp)
setattr -unset mul2dsp (if -dsp)
opt_expr -fine (if -dsp)
wreduce (if -dsp)
select -clear (if -dsp)
ice40_dsp (if -dsp)
chtype -set $mul t:$__soft_mul (if -dsp)
```

`wreduce t:$mul` performs width reduction again, this time targetting only cells of type `$mul`. `techmap -map +/mul2dsp.v -map +/ice40/dsp_map.v ... -D DSP_NAME=$__MUL16X16` uses `techmap` to map `$mul` cells to `$__MUL16X16` which are, in turn, mapped to the iCE40 SB_MAC16. Any multipliers which aren't compatible with conversion to `$__MUL16X16` are relabelled to `$__soft_mul` before `chtype` changes them back to `$mul`.

During the `mul2dsp` conversion, some of the intermediate signals are marked with the attribute `mul2dsp`. By calling `select a:mul2dsp` we restrict the following commands to only operate on the cells and wires used for these signals. `setattr` removes the now unnecessary `mul2dsp` attribute. `opt_expr` we've already come across for const folding and simple expression rewriting, the `-fine` option just enables more fine-grain optimizations. Then we perform width reduction a final time and clear the selection.

Todo: `ice40_dsp` is pmgen

Finally we have `ice40_dsp`: similar to the `memory_dff` command we saw in the previous section, this merges any surrounding registers into the SB_MAC16 cell. This includes not just the input/output registers, but also pipeline registers and even a post-adder where applicable: turning a multiply + add into a single multiply-accumulate.

See also:

Advanced usage docs for [Technology mapping](#)

Part 4

That brings us to the fourth and final part for the iCE40 synthesis flow:

Listing 2.14: `coarse` section (part 4)

```
alumacc
opt
memory -nomap [-no-rw-check]
opt_clean
```

Where before each type of arithmetic operation had its own cell, e.g. `$add`, we now want to extract these into `$alu` and `$macc` cells which can help identify opportunities for reusing logic. We do this by running `alumacc`, which we can see produce the following changes in our example design:

Listing 2.15: output of `alumacc`

```
yosys> alumacc

24. Executing ALUMACC pass (create $alu and $macc cells).
Extracting $alu and $macc cells in module fifo:
  creating $macc model for $add$fifo.v:66$29 ($add).
  creating $macc model for $flatten\fifo_reader.$add$fifo.v:19$36 ($add).
  creating $macc model for $flatten\fifo_writer.$add$fifo.v:19$36 ($add).
  creating $macc model for $sub$fifo.v:68$32 ($sub).
  creating $alu model for $macc $sub$fifo.v:68$32.
  creating $alu model for $macc $flatten\fifo_writer.$add$fifo.v:19$36.
  creating $alu model for $macc $flatten\fifo_reader.$add$fifo.v:19$36.
  creating $alu model for $macc $add$fifo.v:66$29.
  creating $alu cell for $add$fifo.v:66$29: $auto$alumacc.cc:495:replace_alu$87
  creating $alu cell for $flatten\fifo_reader.$add$fifo.v:19$36: $auto$alumacc.
→cc:495:replace_alu$90
  creating $alu cell for $flatten\fifo_writer.$add$fifo.v:19$36: $auto$alumacc.
→cc:495:replace_alu$93
  creating $alu cell for $sub$fifo.v:68$32: $auto$alumacc.cc:495:replace_alu$96
  created 4 $alu and 0 $macc cells.
```

Once these cells have been inserted, the call to `opt` can combine cells which are now identical but may have been missed due to e.g. the difference between `$add` and `$sub`.

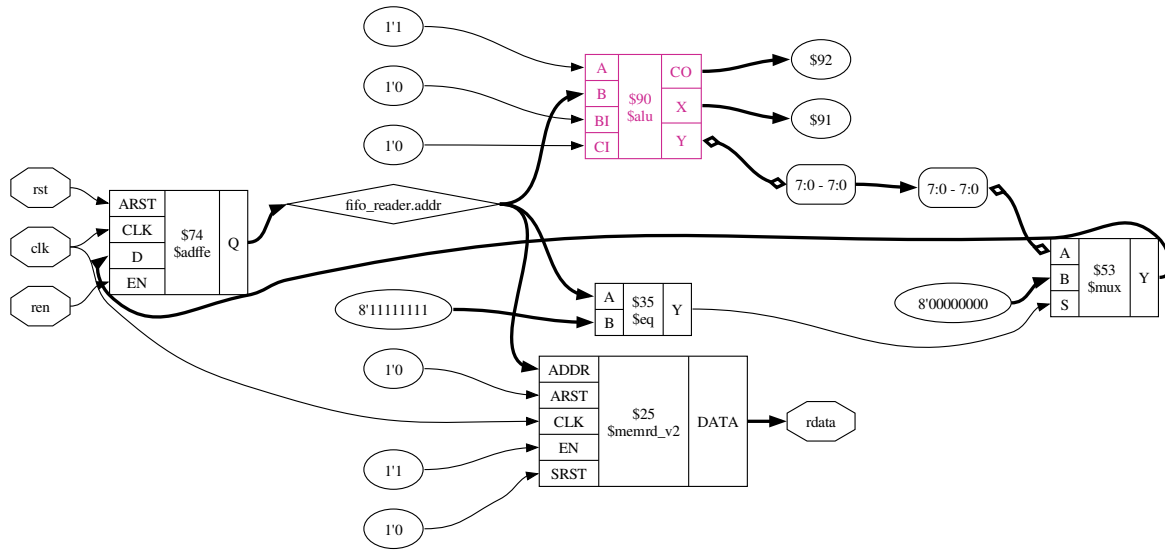
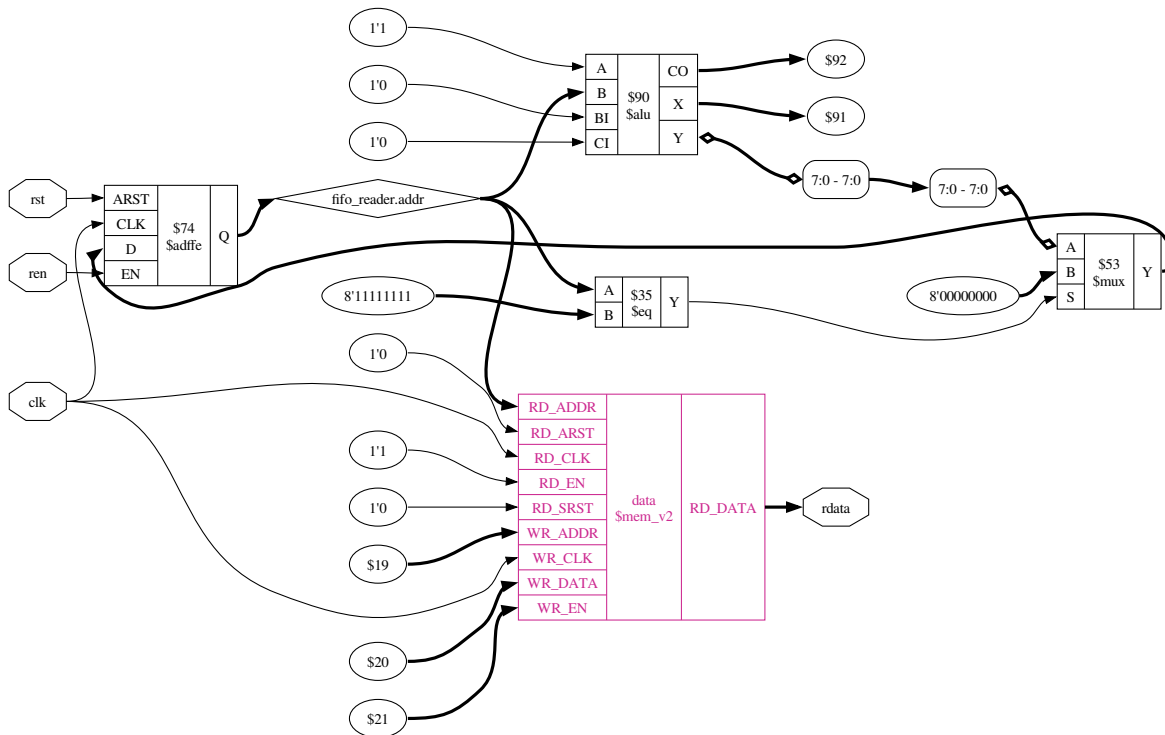
The other new command in this part is `memory` - *translate memories to basic cells*. `memory` is another macro command which we examine in more detail in *Memory handling*. For this document, let us focus just on the step most relevant to our example: `memory_collect`. Up until this point, our memory reads and our memory writes have been totally disjoint cells; operating on the same memory only in the abstract. `memory_collect` combines all of the reads and writes for a memory block into a single cell.

Looking at the schematic after running `memory_collect` we see that our `$memrd_v2` cell has been replaced with a `$mem_v2` cell named `data`, the same name that we used in `fifo.v`. Where before we had a single set of signals for address and enable, we now have one set for reading (`RD_*`) and one for writing (`WR_*`), as well as both `WR_DATA` input and `RD_DATA` output.

See also:

Advanced usage docs for

- *Optimization passes*
- *Memory handling*

Fig. 2.9: rdata output after *alu_macc*Fig. 2.10: rdata output after *memory_collect*

Final note

Having now reached the end of the the coarse-grain representation, we could also have gotten here by running `synth_ice40 -top fifo -run :map_ram` after loading the design. The `-run <from_label>:<to_label>` option with an empty `<from_label>` starts from the *begin section*, while the `<to_label>` runs up to but including the *map_ram section*.

2.2.6 Hardware mapping

The remaining sections each map a different type of hardware and are much more architecture dependent than the previous sections. As such we will only be looking at each section very briefly.

If you skipped calling `read_verilog -D ICE40_HX -lib -specify +/ice40/cells_sim.v` earlier, do it now.

Memory blocks

Mapping to hard memory blocks uses a combination of *memory_libmap* and *techmap*.

Listing 2.16: *map_ram* section

```
memory_libmap -lib +/ice40/brams.txt -lib +/ice40/spram.txt [-no-auto-huge] [-no-auto-
↪block]      (-no-auto-huge unless -spram, -no-auto-block if -nobram)
techmap -map +/ice40/brams_map.v -map +/ice40/spram_map.v
ice40_braminit
```

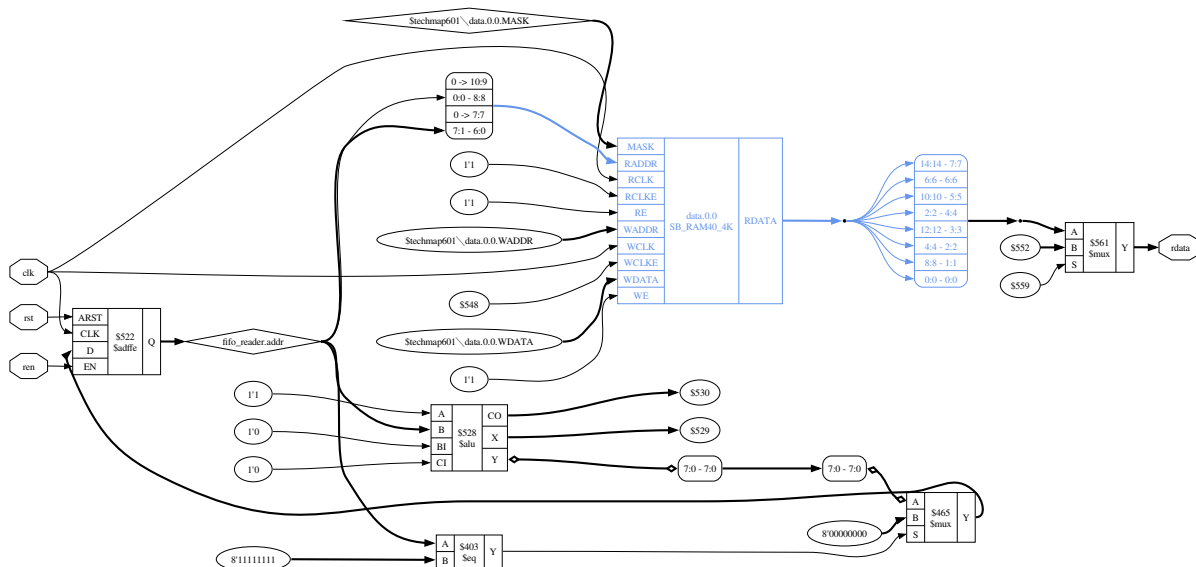


Fig. 2.11: *rdata* output after *map_ram* section

The *map_ram* section converts the generic *\$mem_v2* into the iCE40 SB_RAM40_4K (highlighted). We can also see the memory address has been remapped, and the data bits have been reordered (or swizzled). There is also now a *\$mux* cell controlling the value of *rdata*. In *fifo.v* we wrote our memory as read-before-write, however the SB_RAM40_4K has undefined behaviour when reading from and writing to the same address in

the same cycle. As a result, extra logic is added so that the generated circuit matches the behaviour of the verilog. *Synchronous SDP with undefined collision behavior* describes how we could change our verilog to match our hardware instead.

If we run `memory_libmap` under the `debug` command we can see candidates which were identified for mapping, along with the costs of each and what logic requires emulation.

```
yosys> debug memory_libmap -lib +/ice40/brams.txt -lib +/ice40/spram.txt -no-auto-huge
4. Executing MEMORY_LIBMAP pass (mapping memories to cells).
Memory fifo.data mapping candidates (post-geometry):
- logic fallback
  - cost: 2048.000000
- $__ICE40_RAM4K_:
  - option HAS_BE 0
  - emulation score: 7
  - replicates (for ports): 1
  - replicates (for data): 1
  - mux score: 0
  - demux score: 0
  - cost: 78.000000
  - abits 11 dbits 2 4 8 16
  - chosen base width 8
  - swizzle 0 1 2 3 4 5 6 7
  - emulate read-first behavior
  - write port 0: port group W
    - widths 2 4 8
  - read port 0: port group R
    - widths 2 4 8 16
    - emulate transparency with write port 0
- $__ICE40_RAM4K_:
  - option HAS_BE 1
  - emulation score: 7
  - replicates (for ports): 1
  - replicates (for data): 1
  - mux score: 0
  - demux score: 0
  - cost: 78.000000
  - abits 11 dbits 2 4 8 16
  - byte width 1
  - chosen base width 8
  - swizzle 0 1 2 3 4 5 6 7
  - emulate read-first behavior
  - write port 0: port group W
    - widths 16
  - read port 0: port group R
    - widths 2 4 8 16
    - emulate transparency with write port 0
Memory fifo.data mapping candidates (after post-geometry prune):
- logic fallback
  - cost: 2048.000000
- $__ICE40_RAM4K_:
  - option HAS_BE 0
  - emulation score: 7
  - replicates (for ports): 1
```

(continues on next page)

(continued from previous page)

```

- replicates (for data): 1
- mux score: 0
- demux score: 0
- cost: 78.000000
- abits 11 dbits 2 4 8 16
- chosen base width 8
- swizzle 0 1 2 3 4 5 6 7
- emulate read-first behavior
- write port 0: port group W
  - widths 2 4 8
- read port 0: port group R
  - widths 2 4 8 16
  - emulate transparency with write port 0
mapping memory fifo.data via $__ICE40_RAM4K_

```

The `$__ICE40_RAM4K_` cell is defined in the file `techlibs/ice40/brams.txt`, with the mapping to `SB_RAM40_4K` done by `techmap` using `techlibs/ice40/brams_map.v`. Any leftover memory cells are then converted into flip flops (the logic fallback) with `memory_map`.

Listing 2.17: map_ffram section

```

opt -fast -mux_undef -undriven -fine
memory_map
opt -undriven -fine

```

Note: The visual clutter on the `RDATA` output port (highlighted) is an unfortunate side effect of `opt_clean` on the swizzled data bits. In connecting the `$mux` input port directly to `RDATA` to reduce the number of wires, the `$techmap579\data.0.0.RDATA` wire becomes more visually complex.

See also:

Advanced usage docs for

- *Technology mapping*
- *Memory handling*

Arithmetic

Uses `techmap` to map basic arithmetic logic to hardware. This sees somewhat of an explosion in cells as multi-bit `$mux` and `$adffe` are replaced with single-bit `$MUX_` and `$DFFE_PPOP_` cells, while the `$alu` is replaced with primitive `$OR_` and `$NOT_` gates and a `$lut` cell.

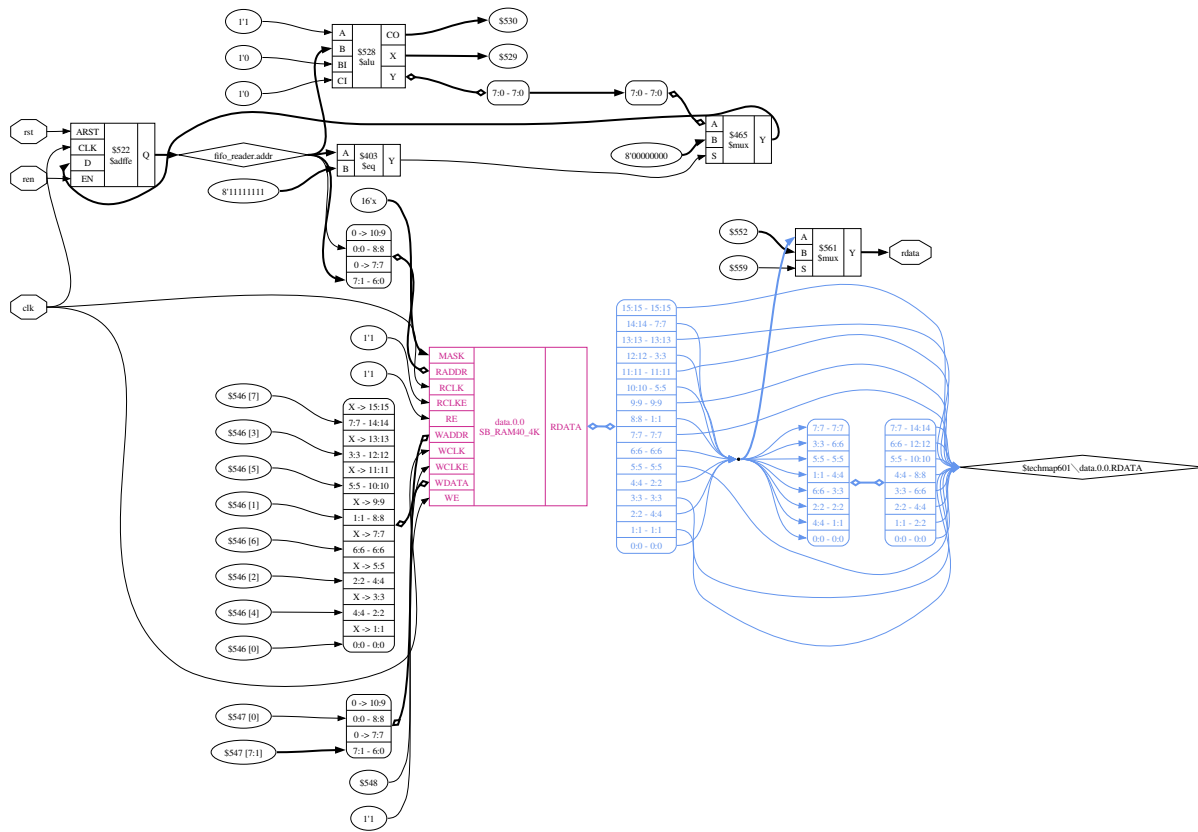
Fig. 2.12: rdata output after `map_ffram` section

Fig. 2.13: rdata output after *map_gates* section

Listing 2.18: map_gates section

```
ice40_wrapcarry
techmap -map +/techmap.v -map +/ice40/arith_map.v
opt -fast
abc -dff -D 1      (only if -retime)
ice40_opt
```

See also:

Advanced usage docs for *Technology mapping*

Flip-flops

Convert FFs to the types supported in hardware with *dfflegalize*, and then use *techmap* to map them. In our example, this converts the `$_DFFE_PPOP_` cells to `SB_DFFER`.

We also run *simplemap* here to convert any remaining cells which could not be mapped to hardware into gate-level primitives. This includes optimizing `$_MUX_` cells where one of the inputs is a constant 1'0, replacing it instead with an `$_AND_` cell.

Listing 2.19: map_ffs section

```
dfflegalize -cell $_DFF?_ 0 -cell $_DFFE?P_ 0 -cell $_DFF?P?_ 0 -cell $_DFFE?P?P_ 0 -
↪cell $_SDFF?P?_ 0 -cell $_SDFFCE?P?P_ 0 -cell $_DLATCH?_ x -mince -1
techmap -map +/ice40/ff_map.v
opt_expr -mux_undef
simplemap
ice40_opt -full
```

See also:

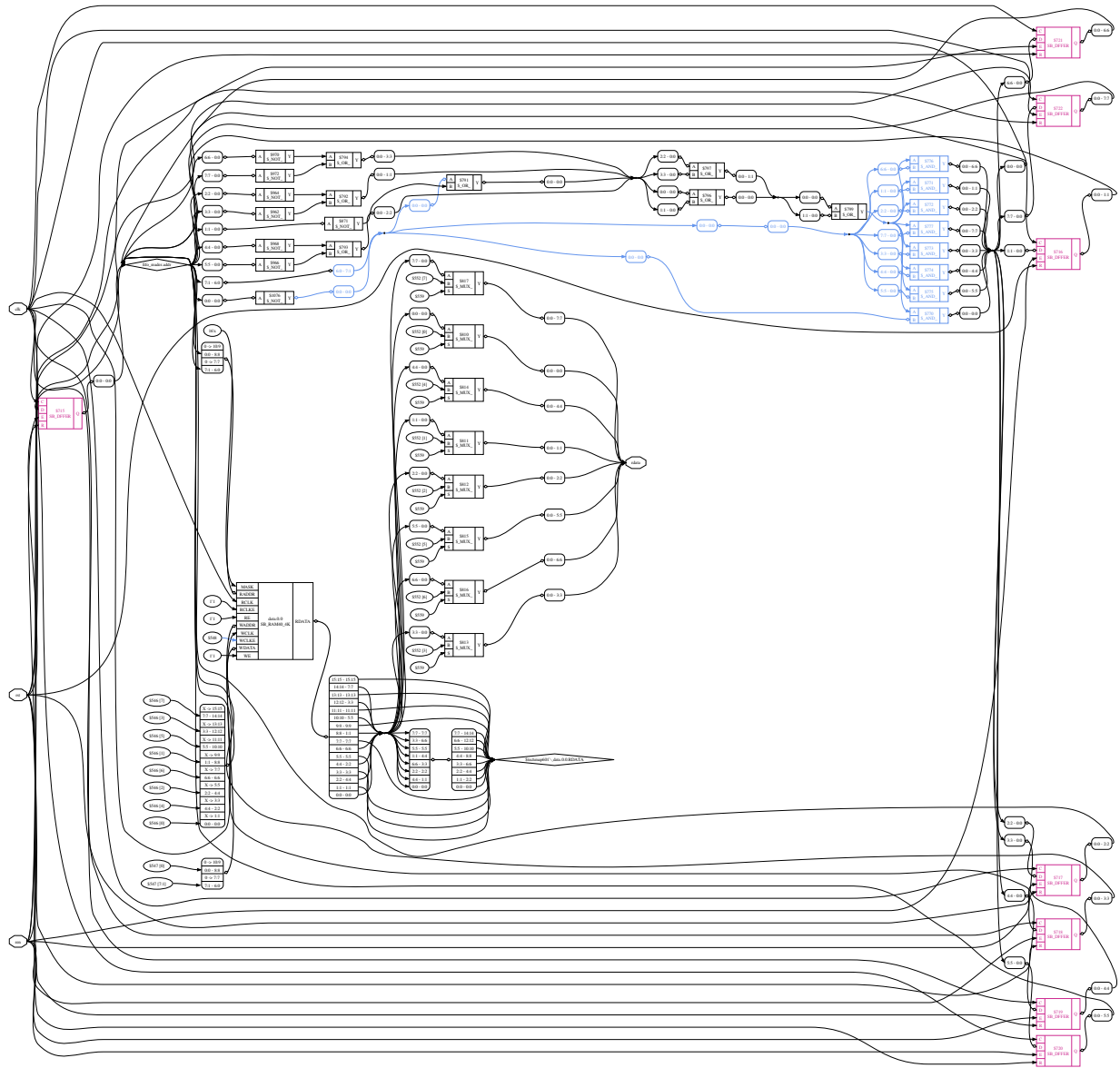
Advanced usage docs for *Technology mapping*

LUTs

abc and *techmap* are used to map LUTs; converting primitive cell types to use *\$lut* and `SB_CARRY` cells. Note that the iCE40 flow uses *abc9* rather than *abc*. For more on what these do, and what the difference between these two commands are, refer to *The ABC toolbox*.

Listing 2.20: map_luts section

```
abc          (only if -abc2)
ice40_opt    (only if -abc2)
techmap -map +/ice40/latches_map.v
simplemap                                (if -noabc or -flowmap)
techmap -map +/gate2lut.v -D LUT_WIDTH=4 (only if -noabc)
flowmap -maxlut 4 (only if -flowmap)
read_verilog -D ICE40_HX -icells -lib -specify +/ice40/abc9_model.v
abc9 -W 250
ice40_wrapcarry -unwrap
techmap -map +/ice40/ff_map.v
clean
opt_lut -tech ice40
```

Fig. 2.14: rdata output after *map_ffs* section

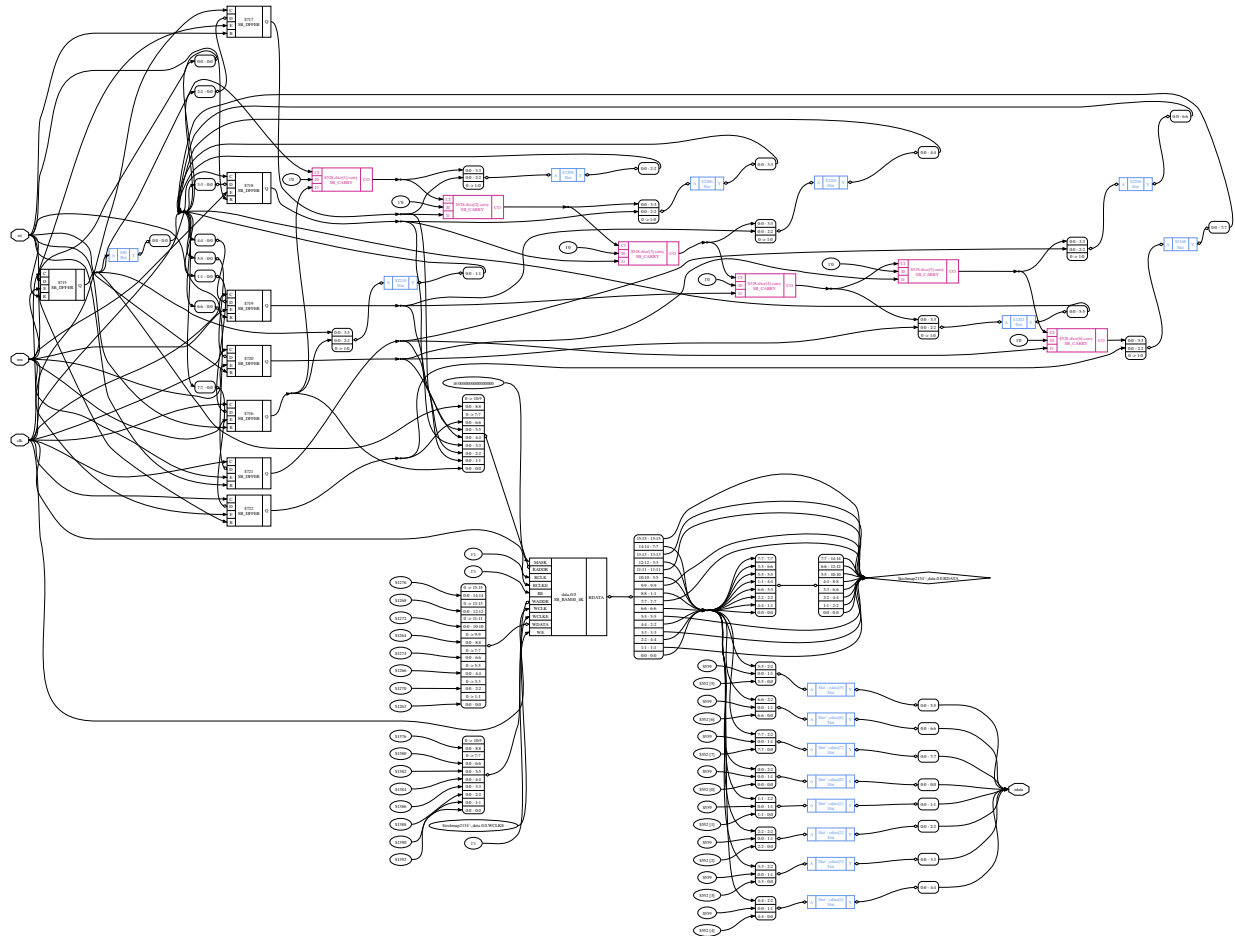


Fig. 2.15: rdata output after *map_luts* section

Finally we use *techmap* to map the generic *\$lut* cells to iCE40 SB_LUT4 cells.

Listing 2.21: *map_cells* section

```
techmap -map +/ice40/cells_map.v    (skip if -vpr)
clean
```

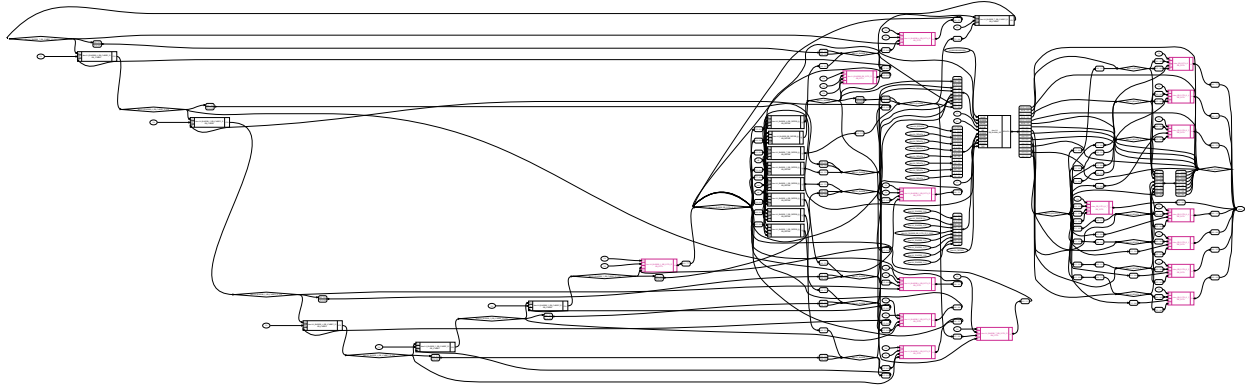


Fig. 2.16: *rdata* output after *map_cells* section

See also:

Advanced usage docs for

- *Technology mapping*
- *The ABC toolbox*

Other cells

The following commands may also be used for mapping other cells:

hilomap

Some architectures require special driver cells for driving a constant hi or lo value. This command replaces simple constants with instances of such driver cells.

iopadmap

Top-level input/outputs must usually be implemented using special I/O-pad cells. This command inserts such cells to the design.

These commands tend to either be in the *map_cells* section or after the *check* section depending on the flow.

2.2.7 Final steps

The next section of the iCE40 synth flow performs some sanity checking and final tidy up:

Listing 2.22: *check* section

```
autoname
hierarchy -check
stat
check -noinit
blackbox =A:whitebox
```

The new commands here are:

- *autoname* - automatically assign names to objects,
- *stat* - print some statistics, and
- *blackbox* - convert modules into blackbox modules.

The output from *stat* is useful for checking resource utilization; providing a list of cells used in the design and the number of each, as well as the number of other resources used such as wires and processes. For this design, the final call to *stat* should look something like the following:

```
yosys> stat -top fifo

17. Printing statistics.

=== fifo ===

Number of wires:          95
Number of wire bits:      262
Number of public wires:   95
Number of public wire bits: 262
Number of ports:          7
Number of port bits:      29
Number of memories:       0
Number of memory bits:    0
Number of processes:      0
Number of cells:          138
  $scopeinfo              2
  SB_CARRY                 26
  SB_DFF                   26
  SB_DFFER                 25
  SB_LUT4                  58
  SB_RAM40_4K              1
```

Note that the *-top fifo* here is optional. *stat* will automatically use the module with the *top* attribute set, which *fifo* was when we called *hierarchy*. If no module is marked *top*, then stats will be shown for each module selected.

The *stat* output is also useful as a kind of sanity-check: Since we have already run *proc*, we wouldn't expect there to be any processes. We also expect *data* to use hard memory; if instead of an *SB_RAM40_4K* saw a high number of flip-flops being used we might suspect something was wrong.

If we instead called *stat* immediately after *read_verilog fifo.v* we would see something very different:

```
yosys> stat

2. Printing statistics.

=== fifo ===

Number of wires:          28
Number of wire bits:      219
Number of public wires:   9
Number of public wire bits: 45
Number of ports:          7
```

(continues on next page)

(continued from previous page)

Number of port bits:	29
Number of memories:	1
Number of memory bits:	2048
Number of processes:	3
Number of cells:	9
\$add	1
\$logic_and	2
\$logic_not	2
\$memrd	1
\$sub	1
addr_gen	2
=== addr_gen ===	
Number of wires:	8
Number of wire bits:	60
Number of public wires:	4
Number of public wire bits:	11
Number of ports:	4
Number of port bits:	11
Number of memories:	0
Number of memory bits:	0
Number of processes:	2
Number of cells:	2
\$add	1
\$eq	1

Notice how `fifo` and `addr_gen` are listed separately, and the statistics for `fifo` show 2 `addr_gen` modules. Because this is before the memory has been mapped, we also see that there is 1 memory with 2048 memory bits; matching our 8-bit wide `data` memory with 256 values ($8 * 256 = 2048$).

Synthesis output

The iCE40 synthesis flow has the following output modes available:

- *write_blif* - write design to BLIF file,
- *write_edif* - write design to EDIF netlist file, and
- *write_json* - write design to a JSON file.

As an example, if we called `synth_ice40 -top fifo -json fifo.json`, our synthesized `fifo` design will be output as `fifo.json`. We can then read the design back into Yosys with *read_json*, but make sure you use `design -reset` or open a new interactive terminal first. The JSON output we get can also be loaded into *nextpnr* to do place and route; but that is beyond the scope of this documentation.

See also:

synth_ice40 - synthesis for iCE40 FPGAs

2.3 Scripting in Yosys

On the previous page we went through a synthesis script, running each command in the interactive Yosys shell. On this page, we will be introducing the script file format and how you can make your own synthesis scripts.

Yosys script files typically use the `.ys` extension and contain a set of commands for Yosys to run sequentially. These commands are the same ones we were using on the previous page like `read_verilog` and `hierarchy`.

2.3.1 Script parsing

As with the interactive shell, each command consists of the command name, and an optional whitespace separated list of arguments. Commands are terminated with the newline character, and anything after a hash sign `#` is a comment (i.e. it is ignored).

It is also possible to terminate commands with a semicolon `;`. This is particularly useful in conjunction with the `-p <command>` command line option, where `<command>` can be a string with multiple commands separated by semicolon. In-line comments can also be made with the colon `:`, where the end of the comment is a semicolon `;` or a new line.

Listing 2.23: Using the `-p` option

```
$ yosys -p "read_verilog fifo.v; :this is a comment; prep"
```

Warning: The space after the semicolon is required for correct parsing. `log a;log b;` for example will display `a;log b` instead of `a` and `b` as might be expected.

Another special character that can be used in Yosys scripts is the bang `!`. Anything after the bang will be executed as a shell command. This can only be terminated with a new line. Any semicolons, hashes, or other special characters will be passed to the shell. If an error code is returned from the shell it will be raised by Yosys. `exec` provides a much more flexible way of executing commands, allowing the output to be logged and more control over when to generate errors.

2.3.2 The synthesis starter script

All of the images and console output used in *Synthesis starter* were generated by Yosys, using Yosys script files found in `docs/source/code_examples/fifo`. If you haven't already, let's take a look at some of those script files now.

Listing 2.24: A section of `fifo.ys`, generating the images used for
The addr_gen module

```
10 echo on
11 hierarchy -top addr_gen
12 select -module addr_gen
13 select -list
14 select t:*
15 select -list
16 select -set new_cells %
17 select -clear
18 show -format dot -prefix addr_gen_show addr_gen
```

(continues on next page)

(continued from previous page)

```

19 show -format dot -prefix new_cells_show -notitle @new_cells
20 show -color maroon3 @new_cells -color cornflowerblue p:* -notitle -format dot -prefix addr_gen_hier
21
22 # =====
23 proc -noopt
24 select -set new_cells t:$mux t:*dff
25 show -color maroon3 @new_cells -notitle -format dot -prefix addr_gen_proc
26
27 # =====
28 opt_expr; clean
29 select -set new_cells t:$eq
30 show -color cornflowerblue @new_cells -notitle -format dot -prefix addr_gen_clean
31
32 # =====

```

The first command there, `echo on`, uses `echo` to enable command echoes on. This is how we generated the code listing for *hierarchy -top addr_gen output*. Turning command echoes on prints the `yosys> hierarchy -top addr_gen` line, making the output look the same as if it were an interactive terminal. `hierarchy -top addr_gen` is of course the command we were demonstrating, including the output text and an image of the design schematic after running it.

We briefly touched on `select` when it came up in *synth_ice40*, but let's look at it more now.

Selections intro

The `select` command is used to modify and view the list of selected objects:

```

yosys> select -module addr_gen

yosys [addr_gen]> select -list
addr_gen
addr_gen/$1\addr[7:0]
addr_gen/$add$fifo.v:19$3_Y
addr_gen/$eq$fifo.v:16$2_Y
addr_gen/$0\addr[7:0]
addr_gen/addr
addr_gen/rst
addr_gen/clk
addr_gen/en
addr_gen/$add$fifo.v:19$3
addr_gen/$eq$fifo.v:16$2
addr_gen/$proc$fifo.v:0$4
addr_gen/$proc$fifo.v:12$1

yosys [addr_gen]> select t:*

yosys [addr_gen]> select -list
addr_gen/$add$fifo.v:19$3
addr_gen/$eq$fifo.v:16$2

yosys [addr_gen]> select -set new_cells %

```

(continues on next page)

(continued from previous page)

```
yosys [addr_gen]> select -clear
```

When we call `select -module addr_gen` we are changing the currently active selection from the whole design, to just the `addr_gen` module. Notice how this changes the `yosys` at the start of each command to `yosys [addr_gen]`? This indicates that any commands we run at this point will *only* operate on the `addr_gen` module. When we then call `select -list` we get a list of all objects in the `addr_gen` module, including the module itself, as well as all of the wires, inputs, outputs, processes, and cells.

Next we perform another selection, `select t:*`. The `t:` part signifies we are matching on the *cell type*, and the `*` means to match anything. For this (very simple) selection, we are trying to find all of the cells, regardless of their type. The active selection is now shown as `[addr_gen]*`, indicating some sub-selection of the `addr_gen` module. This gives us the `$add` and `$eq` cells, which we want to highlight for the *addr_gen module after hierarchy* image.

We can assign a name to a selection with `select -set`. In our case we are using the name `new_cells`, and telling it to use the current selection, indicated by the `%` symbol. We can then use this named selection by referring to it as `@new_cells`, which we will see later. Then we clear the selection so that the following commands can operate on the full design. While we split that out for this document, we could have done the same thing in a single line by calling `select -set new_cells addr_gen/t:*`. If we know we only have the one module in our design, we can even skip the `addr_gen/` part. Looking further down *the fifo.ys code* we can see this with `select -set new_cells t:$mux t:*dff`. We can also see in that command that selections don't have to be limited to a single statement.

Many commands also support an optional `[selection]` argument which can be used to override the currently selected objects. We could, for example, call `clean addr_gen` to have *clean* operate on *just* the `addr_gen` module.

Detailed documentation of the select framework can be found under *Selections* or in the command reference at *select - modify and view the list of selected objects*.

Displaying schematics

While the *select* command is very useful, sometimes nothing beats being able to see a design for yourself. This is where *show* comes in. Note that this document is just an introduction to the *show* command, only covering the basics. For more information, including a guide on what the different symbols represent, see *A look at the show command* and the *Interactive design investigation* page.

Note: The *show* command requires a working installation of *GraphViz* and *xdot* for displaying the actual circuit diagrams.

This is the first *show* command we called in `fifo.ys`, *as we saw above*. If we look at the log output for this image we see the following:

```
yosys> show -format dot -prefix addr_gen_show addr_gen
```

```
4. Generating Graphviz representation of design.
Writing dot description to `addr_gen_show.dot'.
Dumping module addr_gen to page 1.
```

Calling *show* with `-format dot` tells it we want to output a `.dot` file rather than opening it for display. The `-prefix addr_gen_show` option indicates we want the file to be called `addr_gen_show.*`. Remember, we do this in `fifo.ys` because we need to store the image for displaying in the documentation you're reading.

But if you just want to display the images locally you can skip these two options. The `-format` option internally calls the `dot` command line program from GraphViz to convert to formats other than `.dot`. Check [GraphViz output docs](#) for more on available formats.

Note: If you are using a POSIX based version of Yosys (such as for Mac or Linux), `xdot` will be opened in the background and Yosys can continue to be used. If it is still open, future calls to `show` will use the same `xdot` instance.

The `addr_gen` at the end tells it we only want the `addr_gen` module, just like when we called `select -module addr_gen` in [Selections intro](#). That last parameter doesn't have to be a module name, it can be any valid selection string. Remember when we [assigned a name to a selection](#) and called it `new_cells`? We saw in the `select -list` output that it contained two cells, an `$add` and an `$eq`. We can call `show` on that selection just as easily:

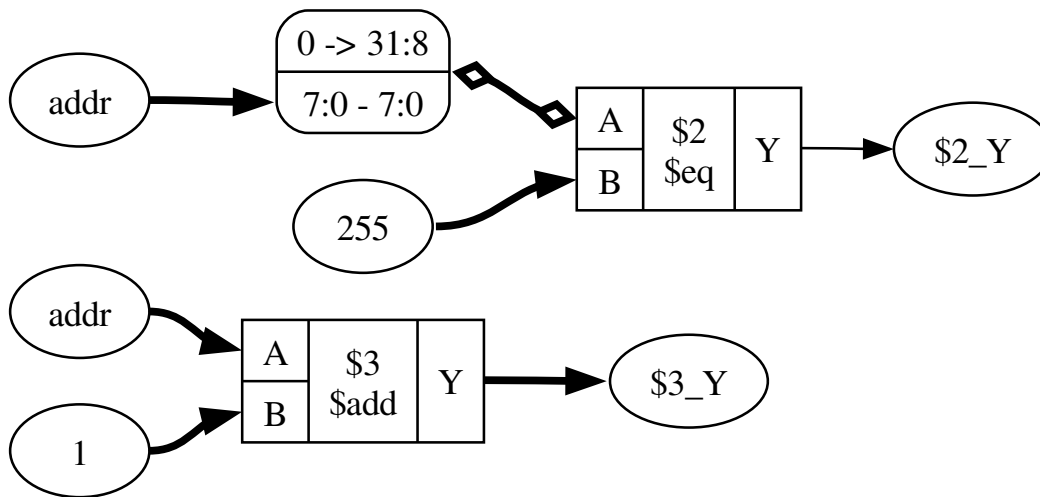


Fig. 2.18: Calling `show -notitle @new_cells`

We could have gotten the same output with `show -notitle t:$add t:$eq` if we didn't have the named selection. By adding the `-notitle` flag there we can also get rid of the `addr_gen` title that would have been automatically added. The last two images were both added for this introduction. The next image is the first one we saw in [Synthesis starter](#): showing the full `addr_gen` module while also highlighting `@new_cells` and the two PROC blocks. To achieve this highlight, we make use of the `-color` option:

As described in the [help](#) output for `show` (or by clicking on the `show` link), colors are specified as `-color <color> <object>`. Color names for the `<color>` portion can be found on the [GraphViz color docs](#). Unlike the final `show` parameter which can have be any selection string, the `<object>` part must be a single selection expression or named selection. That means while we can use `@new_cells`, we couldn't use `t:$eq` or `t:$add`. In general, if a command lists `[selection]` as its final parameter it can be any selection string. Any selections that are not the final parameter, such as those used in options, must be a single expression instead.

For all of the options available to `show`, check the command reference at [show - generate schematics using](#)

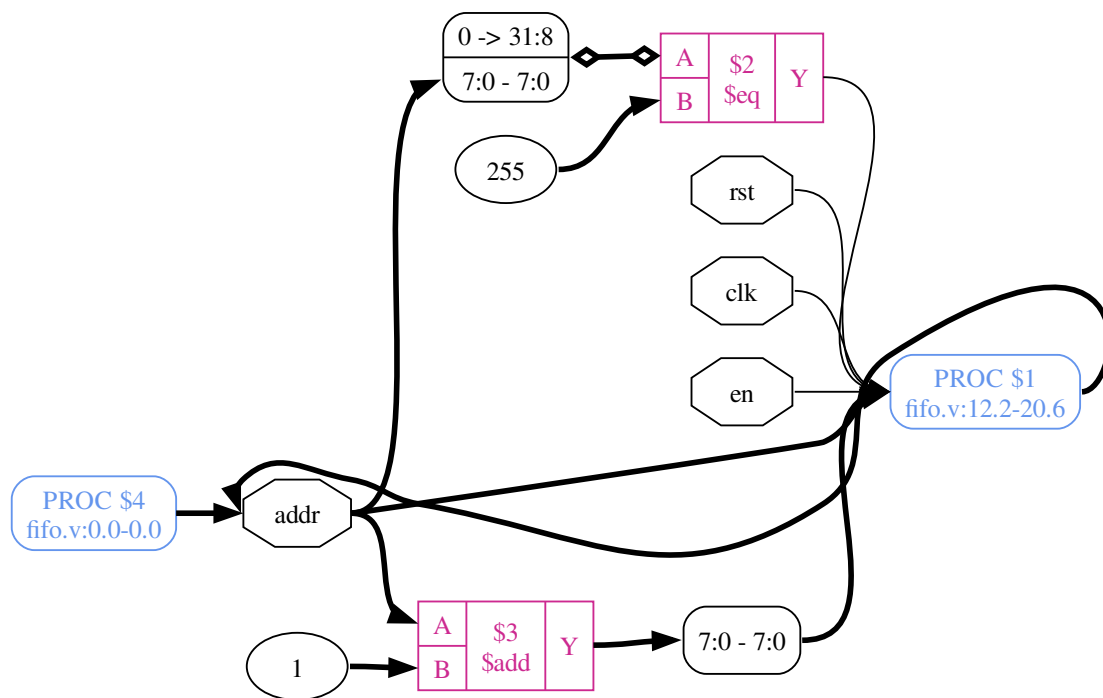


Fig. 2.19: Calling `show -color maroon3 @new_cells -color cornflowerblue p:* -notitle`

graphviz.

See also:

A look at the show command on the *Interactive design investigation* page.

USING YOSYS (ADVANCED)

While much of Yosys is focused around synthesis, there are also a number of other useful things that can be accomplished with Yosys scripts or in an interactive shell. As such this section is broken into two parts: *Synthesis in detail* expands on the *Synthesis starter* and goes into further detail on the major commands used in synthesis; *More scripting* covers the ways Yosys can interact with designs for a deeper investigation.

3.1 Synthesis in detail

Synthesis can generally be broken down into coarse-grain synthesis, and fine-grain synthesis. We saw this in *Synthesis starter* where a design was loaded and elaborated and then went through a series of coarse-grain optimizations before being mapped to hard blocks and fine-grain cells. Most commands in Yosys will target either coarse-grain representation or fine-grain representation, with only a select few compatible with both states.

Commands such as *proc*, *fsm*, and *memory* rely on the additional information in the coarse-grain representation, along with a number of optimizations such as *wreduce*, *share*, and *alumacc*. *opt* provides optimizations which are useful in both states, while *techmap* is used to convert coarse-grain cells to the corresponding fine-grain representation.

Single-bit cells (logic gates, FFs) as well as LUTs, half-adders, and full-adders make up the bulk of the fine-grain representation and are necessary for commands such as *abc/abc9*, *simplemap*, *dfflegalize*, and *memory_map*.

3.1.1 Synth commands

Todo: comment on common `synth_*` options, like `-run`

Packaged `synth_*` commands

The following is a list of all synth commands included in Yosys for different platforms. Each command runs a script of sub commands specific to the platform being targeted. Note that not all of these scripts are actively maintained and may not be up-to-date.

- *synth_achronix* - *synthesis for Achronix Speedster22i FPGAs*.
- *synth_anlogic* - *synthesis for Anlogic FPGAs*
- *synth_coolrunner2* - *synthesis for Xilinx Coolrunner-II CPLDs*
- *synth_easic* - *synthesis for eASIC platform*

- *synth_ecp5* - synthesis for ECP5 FPGAs
- *synth_efinix* - synthesis for Efinix FPGAs
- *synth_fabulous* - FABulous synthesis script
- *synth_gatamate* - synthesis for Cologne Chip GateMate FPGAs
- *synth_gowin* - synthesis for Gowin FPGAs
- *synth_greenpak4* - synthesis for GreenPAK4 FPGAs
- *synth_ice40* - synthesis for iCE40 FPGAs
- *synth_intel* - synthesis for Intel (Altera) FPGAs. (MAX10, Cyclone IV)
- *synth_intel_alm* - synthesis for ALM-based Intel (Altera) FPGAs. (Cyclone V, Arria V, Cyclone 10 GX)
- *synth_lattice* - synthesis for Lattice FPGAs
- *synth_nexus* - synthesis for Lattice Nexus FPGAs
- *synth_quicklogic* - Synthesis for QuickLogic FPGAs
- *synth_sf2* - synthesis for SmartFusion2 and IGLOO2 FPGAs
- *synth_xilinx* - synthesis for Xilinx FPGAs

General synthesis

In addition to the above hardware-specific synth commands, there is also *prep* - *generic synthesis script*. This command is limited to coarse-grain synthesis, without getting into any architecture-specific mappings or optimizations. Among other things, this is useful for design verification.

The following commands are executed by the *prep* command:

```
begin:
    hierarchy -check [-top <top> | -auto-top]

coarse:
    proc [-ifx]
    flatten      (if -flatten)
    future
    opt_expr -keepdc
    opt_clean
    check
    opt -noff -keepdc
    wreduce -keepdc [-memx]
    memory_dff   (if -rdff)
    memory_memx  (if -memx)
    opt_clean
    memory_collect
    opt -noff -keepdc -fast

check:
    stat
    check
```

Synthesis starter covers most of these commands and what they do.

3.1.2 Converting process blocks

The Verilog frontend converts `always`-blocks to RTL netlists for the expressions and “processes” for the control- and memory elements. The `proc` command then transforms these “processes” to netlists of RTL multiplexer and register cells. It also is a macro command that calls the other `proc_*` commands in a sensible order:

Listing 3.1: Passes called by `proc`

```
proc_clean # removes empty branches and processes
proc_rmdead # removes unreachable branches
proc_prune
proc_init # special handling of "initial" blocks
proc_arst # identifies modeling of async resets
proc_rom
proc_mux # converts decision trees to multiplexer networks
proc_dlatch
proc_dff # extracts registers from processes
proc_memwr
proc_clean # this should remove all the processes, provided all went fine
opt_expr -keepdc
```

After all the `proc_*` commands, `opt_expr` is called. This can be disabled by calling `proc -noopt`. For more information about `proc`, such as disabling certain sub commands, see `proc - translate processes to netlists`.

Many commands can not operate on modules with “processes” in them. Usually a call to `proc` is the first command in the actual synthesis procedure after design elaboration.

Example

Todo: describe proc images

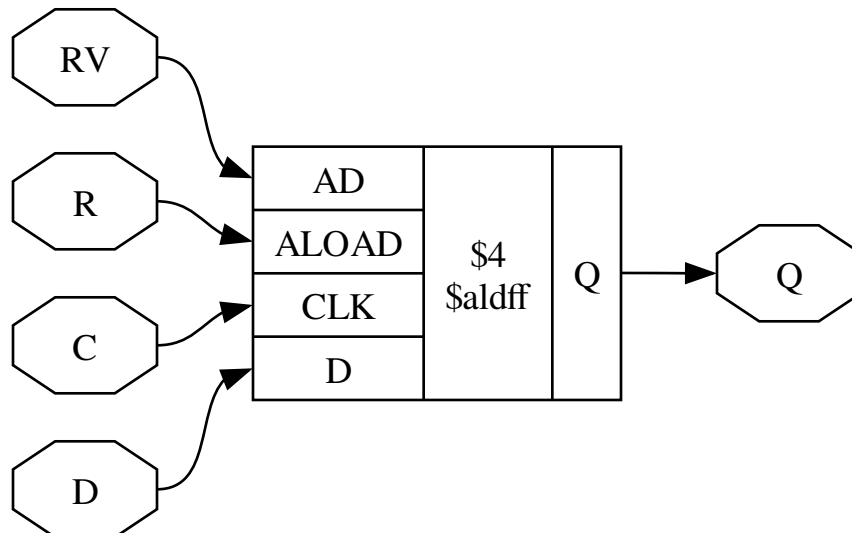
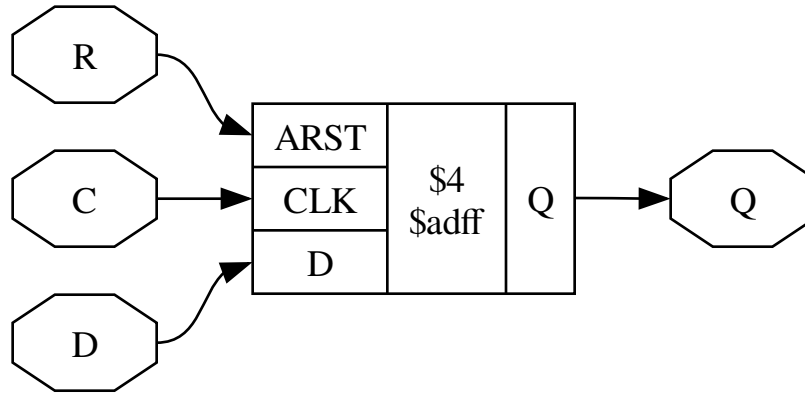
`docs/source/code_examples/synth_flow.`

Listing 3.2: `proc_01.v`

```
module test(input D, C, R, output reg Q);
    always @(posedge C, posedge R)
        if (R)
            Q <= 0;
        else
            Q <= D;
endmodule
```

Listing 3.3: `proc_01.y`

```
read_verilog proc_01.v
hierarchy -check -top test
proc;;
```



Listing 3.4: proc_02.v

```

module test(input D, C, R, RV,
            output reg Q);
    always @(posedge C, posedge R)
        if (R)
            Q <= RV;
        else
            Q <= D;
endmodule

```

Listing 3.5: proc_02.y

```

read_verilog proc_02.v
hierarchy -check -top test
proc;;

```



Listing 3.6: proc_03.y

```

read_verilog proc_03.v
hierarchy -check -top test
proc;;

```

Listing 3.7: proc_03.v

```

module test(input A, B, C, D, E,
            output reg Y);

```

(continues on next page)

(continued from previous page)

```
always @* begin
    Y <= A;
    if (B)
        Y <= C;
    if (D)
        Y <= E;
end
endmodule
```

3.1.3 FSM handling

The *fsm* command identifies, extracts, optimizes (re-encodes), and re-synthesizes finite state machines. It again is a macro that calls a series of other commands:

Listing 3.8: Passes called by *fsm*

```
# Identify and extract FSMs:
fsm_detect
fsm_extract

# Basic optimizations:
fsm_opt
opt_clean
fsm_opt

# Expanding to nearby gate-logic (if called with -expand):
fsm_expand
opt_clean
fsm_opt

# Re-code FSM states (unless called with -norecode):
fsm_recode

# Print information about FSMs:
fsm_info

# Export FSMs in KISS2 file format (if called with -export):
fsm_export

# Map FSMs to RTL cells (unless called with -nomap):
fsm_map
```

See also *fsm - extract and optimize finite state machines*.

The algorithms used for FSM detection and extraction are influenced by a more general reported technique [STGR10].

FSM detection

The *fsm_detect* pass identifies FSM state registers. It sets the `fsm_encoding = "auto"` attribute on any (multi-bit) wire that matches the following description:

- Does not already have the `fsm_encoding` attribute.
- Is not an output of the containing module.
- Is driven by single *\$dff* or *\$adff* cell.
- The D-Input of this *\$dff* or *\$adff* cell is driven by a multiplexer tree that only has constants or the old state value on its leaves.
- The state value is only used in the said multiplexer tree or by simple relational cells that compare the state value to a constant (usually *\$eq* cells).

This heuristic has proven to work very well. It is possible to overwrite it by setting `fsm_encoding = "auto"` on registers that should be considered FSM state registers and setting `fsm_encoding = "none"` on registers that match the above criteria but should not be considered FSM state registers.

Note however that marking state registers with `fsm_encoding` that are not suitable for FSM recoding can cause synthesis to fail or produce invalid results.

FSM extraction

The *fsm_extract* pass operates on all state signals marked with the (`fsm_encoding != "none"`) attribute. For each state signal the following information is determined:

- The state registers
- The asynchronous reset state if the state registers use asynchronous reset
- All states and the control input signals used in the state transition functions
- The control output signals calculated from the state signals and control inputs
- A table of all state transitions and corresponding control inputs- and outputs

The state registers (and asynchronous reset state, if applicable) is simply determined by identifying the driver for the state signal.

From there the *\$mux*-tree driving the state register inputs is recursively traversed. All select inputs are control signals and the leaves of the *\$mux*-tree are the states. The algorithm fails if a non-constant leaf that is not the state signal itself is found.

The list of control outputs is initialized with the bits from the state signal. It is then extended by adding all values that are calculated by cells that compare the state signal with a constant value.

In most cases this will cover all uses of the state register, thus rendering the state encoding arbitrary. If however a design uses e.g. a single bit of the state value to drive a control output directly, this bit of the state signal will be transformed to a control output of the same value.

Finally, a transition table for the FSM is generated. This is done by using the ConstEval C++ helper class (defined in `kernel/consteval.h`) that can be used to evaluate parts of the design. The ConstEval class can be asked to calculate a given set of result signals using a set of signal-value assignments. It can also be passed a list of stop-signals that abort the ConstEval algorithm if the value of a stop-signal is needed in order to calculate the result signals.

The *fsm_extract* pass uses the ConstEval class in the following way to create a transition table. For each state:

1. Create a ConstEval object for the module containing the FSM

2. Add all control inputs to the list of stop signals
3. Set the state signal to the current state
4. Try to evaluate the next state and control output
5. If step 4 was not successful:
 - Recursively goto step 4 with the offending stop-signal set to 0.
 - Recursively goto step 4 with the offending stop-signal set to 1.
6. If step 4 was successful: Emit transition

Finally a `$fsm` cell is created with the generated transition table and added to the module. This new cell is connected to the control signals and the old drivers for the control outputs are disconnected.

FSM optimization

The `fsm_opt` pass performs basic optimizations on `$fsm` cells (not including state recoding). The following optimizations are performed (in this order):

- Unused control outputs are removed from the `$fsm` cell. The attribute `unused_bits` (that is usually set by the `opt_clean` pass) is used to determine which control outputs are unused.
- Control inputs that are connected to the same driver are merged.
- When a control input is driven by a control output, the control input is removed and the transition table altered to give the same performance without the external feedback path.
- Entries in the transition table that yield the same output and only differ in the value of a single control input bit are merged and the different bit is removed from the sensitivity list (turned into a don't-care bit).
- Constant inputs are removed and the transition table is altered to give an unchanged behaviour.
- Unused inputs are removed.

FSM recoding

The `fsm_recode` pass assigns new bit pattern to the states. Usually this also implies a change in the width of the state signal. At the moment of this writing only one-hot encoding with all-zero for the reset state is supported.

The `fsm_recode` pass can also write a text file with the changes performed by it that can be used when verifying designs synthesized by Yosys using Synopsys Formality.

3.1.4 Memory handling

The `memory` command

In the RTL netlist, memory reads and writes are individual cells. This makes consolidating the number of ports for a memory easier. The `memory` pass transforms memories to an implementation. Per default that is logic for address decoders and registers. It also is a macro command that calls the other common `memory_*` passes in a sensible order:

Listing 3.9: Passes called by *memory*

```

opt_mem
opt_mem_priority
opt_mem_feedback
memory_bmux2rom
memory_dff
opt_clean
memory_share
opt_mem_widen
memory_memx                (when called with -memx)
opt_clean
memory_collect
memory_bram -rules <bram_rules>  (when called with -bram)
memory_map                  (skipped if called with -nomap)

```

Todo: Make `memory_*` notes less quick

Some quick notes:

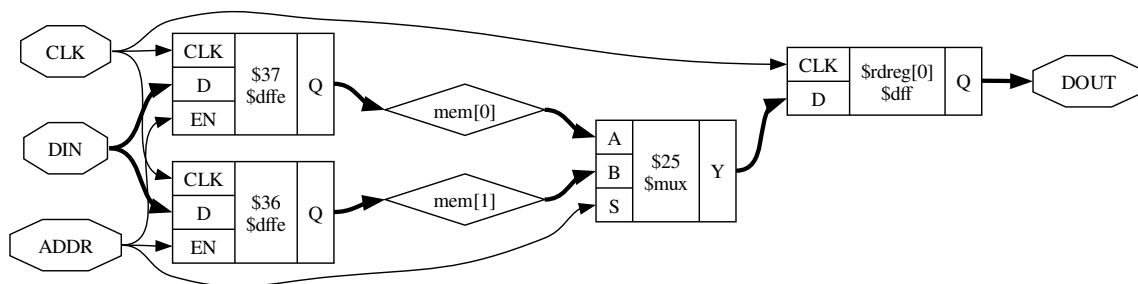
- *memory_dff* merges registers into the memory read- and write cells.
- *memory_collect* collects all read and write cells for a memory and transforms them into one multi-port memory cell.
- *memory_map* takes the multi-port memory cell and transforms it to address decoder logic and registers.

For more information about *memory*, such as disabling certain sub commands, see *memory - translate memories to basic cells*.

Example

Todo: describe memory images

`docs/source/code_examples/synth_flow.`



Listing 3.10: memory_01.ys

```

read_verilog memory_01.v
hierarchy -check -top test
proc;; memory; opt

```

Listing 3.11: memory_01.v

```

module test(input      CLK, ADDR,
            input      [7:0] DIN,
            output reg [7:0] DOUT);
  reg [7:0] mem [0:1];
  always @(posedge CLK) begin
    mem[ADDR] <= DIN;
    DOUT <= mem[ADDR];
  end
endmodule

```



Listing 3.12: memory_02.v

```

module test(
    input          WR1_CLK,  WR2_CLK,
    input          WR1_WEN,  WR2_WEN,
    input [7:0]    WR1_ADDR, WR2_ADDR,
    input [7:0]    WR1_DATA, WR2_DATA,
    input          RD1_CLK,  RD2_CLK,
    input [7:0]    RD1_ADDR, RD2_ADDR,
    output reg [7:0] RD1_DATA, RD2_DATA
);

reg [7:0] memory [0:255];

always @(posedge WR1_CLK)
    if (WR1_WEN)
        memory[WR1_ADDR] <= WR1_DATA;

always @(posedge WR2_CLK)
    if (WR2_WEN)
        memory[WR2_ADDR] <= WR2_DATA;

always @(posedge RD1_CLK)
    RD1_DATA <= memory[RD1_ADDR];

always @(posedge RD2_CLK)
    RD2_DATA <= memory[RD2_ADDR];

endmodule

```

Listing 3.13: memory_02.yo

```

read_verilog memory_02.v
hierarchy -check -top test
proc;; memory -nomap
opt -mux_undef -mux_bool

```

Memory mapping

Usually it is preferred to use architecture-specific RAM resources for memory. For example:

```

memory -nomap
memory_libmap -lib my_memory_map.txt
techmap -map my_memory_map.v
memory_map

```

memory_libmap attempts to convert memory cells (*\$mem_v2* etc) into hardware supported memory using a provided library (*my_memory_map.txt* in the example above). Where necessary, emulation logic is added to ensure functional equivalence before and after this conversion. *techmap -map my_memory_map.v* then uses *techmap* to map to hardware primitives. Any leftover memory cells unable to be converted are then picked up by *memory_map* and mapped to DFFs and address decoders.

Note: More information about what mapping options are available and associated costs of each can be found by enabling debug outputs. This can be done with the `debug` command, or by using the `-g` flag when calling Yosys to globally enable debug messages.

For more on the lib format for `memory_libmap`, see `passes/memory/memlib.md`

Supported memory patterns

Note that not all supported patterns are included in this document, of particular note is that combinations of multiple patterns should generally work. For example, `wbe` could be used in conjunction with any of the simple dual port (SDP) models. In general if a hardware memory definition does not support a given configuration, additional logic will be instantiated to guarantee behaviour is consistent with simulation.

Notes

Memory kind selection

The memory inference code will automatically pick target memory primitive based on memory geometry and features used. Depending on the target, there can be up to four memory primitive classes available for selection:

- FF RAM (aka logic): no hardware primitive used, memory lowered to a bunch of FFs and multiplexers
 - Can handle arbitrary number of write ports, as long as all write ports are in the same clock domain
 - Can handle arbitrary number and kind of read ports
- LUT RAM (aka distributed RAM): uses LUT storage as RAM
 - Supported on most FPGAs (with notable exception of ice40)
 - Usually has one synchronous write port, one or more asynchronous read ports
 - Small
 - Will never be used for ROMs (lowering to plain LUTs is always better)
- Block RAM: dedicated memory tiles
 - Supported on basically all FPGAs
 - Supports only synchronous reads
 - Two ports with separate clocks
 - Usually supports true dual port (with notable exception of ice40 that only supports SDP)
 - Usually supports asymmetric memories and per-byte write enables
 - Several kilobits in size
- Huge RAM:
 - Only supported on several targets:
 - * Some Xilinx UltraScale devices (UltraRAM)
 - Two ports, both with mutually exclusive synchronous read and write

- Single clock
- Initial data must be all-0
- * Some ice40 devices (SPRAM)
 - Single port with mutually exclusive synchronous read and write
 - Does not support initial data
- * Nexus (large RAM)
 - Two ports, both with mutually exclusive synchronous read and write
 - Single clock
- Will not be automatically selected by memory inference code, needs explicit opt-in via `ram_style` attribute

In general, you can expect the automatic selection process to work roughly like this:

- If any read port is asynchronous, only LUT RAM (or FF RAM) can be used.
- If there is more than one write port, only block RAM can be used, and this needs to be a hardware-supported true dual port pattern
 - ... unless all write ports are in the same clock domain, in which case FF RAM can also be used, but this is generally not what you want for anything but really small memories
- Otherwise, either FF RAM, LUT RAM, or block RAM will be used, depending on memory size

This process can be overridden by attaching a `ram_style` attribute to the memory:

- `(* ram_style = "logic" *)` selects FF RAM
- `(* ram_style = "distributed" *)` selects LUT RAM
- `(* ram_style = "block" *)` selects block RAM
- `(* ram_style = "huge" *)` selects huge RAM

It is an error if this override cannot be realized for the given target.

Many alternate spellings of the attribute are also accepted, for compatibility with other software.

Initial data

Most FPGA targets support initializing all kinds of memory to user-provided values. If explicit initialization is not used the initial memory value is undefined. Initial data can be provided by either initial statements writing memory cells one by one of `$readmemh` or `$readmemb` system tasks. For an example pattern, see [*sr_init*](#).

Write port with byte enables

- Byte enables can be used with any supported pattern
- To ensure that multiple writes will be merged into one port, they need to have disjoint bit ranges, have the same address, and the same clock
- Any write enable granularity will be accepted (down to per-bit write enables), but using smaller granularity than natively supported by the target is very likely to be inefficient (eg. using 4-bit bytes on ECP5 will result in either padding the bytes with 5 dummy bits to native 9-bit units or splitting the RAM into two block RAMs)

```
reg [31 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable[0])
        mem[write_addr][7:0] <= write_data[7:0];
    if (write_enable[1])
        mem[write_addr][15:8] <= write_data[15:8];
    if (write_enable[2])
        mem[write_addr][23:16] <= write_data[23:16];
    if (write_enable[3])
        mem[write_addr][31:24] <= write_data[31:24];
    if (read_enable)
        read_data <= mem[read_addr];
end
```

Simple dual port (SDP) memory patterns

Todo: assorted enables, e.g. cen, wen+ren

Asynchronous-read SDP

- This will result in LUT RAM on supported targets

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];
always @(posedge clk)
    if (write_enable)
        mem[write_addr] <= write_data;
assign read_data = mem[read_addr];
```

Synchronous SDP with clock domain crossing

- Will result in block RAM or LUT RAM depending on size
- No behavior guarantees in case of simultaneous read and write to the same address

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge write_clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
end

always @(posedge read_clk) begin
    if (read_enable)
        read_data <= mem[read_addr];
end
```


Synchronous SDP read first

- The read and write parts can be in the same or different processes.
- Will result in block RAM or LUT RAM depending on size
- As long as the same clock is used for both, yosys will ensure read-first behavior. This may require extra circuitry on some targets for block RAM. If this is not necessary, use one of the patterns below.

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
    if (read_enable)
        read_data <= mem[read_addr];
end
```

Synchronous SDP with undefined collision behavior

- Like above, but the read value is undefined when read and write ports target the same address in the same cycle

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;

    if (read_enable) begin
        read_data <= mem[read_addr];

        if (write_enable && read_addr == write_addr)
            // this if block
            read_data <= 'x;
    end
end
```

- Or below, using the no_rw_check attribute

```
(* no_rw_check *)
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;

    if (read_enable)
        read_data <= mem[read_addr];
end
```

Synchronous SDP with write-first behavior

- Will result in block RAM or LUT RAM depending on size
- May use additional circuitry for block RAM if write-first is not natively supported. Will always use additional circuitry for LUT RAM.

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;

    if (read_enable) begin
        read_data <= mem[read_addr];
        if (write_enable && read_addr == write_addr)
            read_data <= write_data;
    end
end
```

Synchronous SDP with write-first behavior (alternate pattern)

- This pattern is supported for compatibility, but is much less flexible than the above

```
reg [ADDR_WIDTH - 1 : 0] read_addr_reg;
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
    read_addr_reg <= read_addr;
end

assign read_data = mem[read_addr_reg];
```

Single-port RAM memory patterns

Asynchronous-read single-port RAM

- Will result in single-port LUT RAM on supported targets

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];
always @(posedge clk)
    if (write_enable)
        mem[addr] <= write_data;
assign read_data = mem[addr];
```

Synchronous single-port RAM with mutually exclusive read/write

- Will result in single-port block RAM or LUT RAM depending on size
- This is the correct pattern to infer ice40 SPRAM (with manual ram_style selection)
- On targets that don't support read/write block RAM ports (eg. ice40), will result in SDP block RAM instead
- For block RAM, will use "NO_CHANGE" mode if available

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[addr] <= write_data;
    else if (read_enable)
        read_data <= mem[addr];
end
```

Synchronous single-port RAM with read-first behavior

- Will only result in single-port block RAM when read-first behavior is natively supported; otherwise, SDP RAM with additional circuitry will be used
- Many targets (Xilinx, ECP5, ...) can only natively support read-first/write-first single-port RAM (or TDP RAM) where the write_enable signal implies the read_enable signal (ie. can never write without reading). The memory inference code will run a simple SAT solver on the control signals to determine if this is the case, and insert emulation circuitry if it cannot be easily proven.

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[addr] <= write_data;
    if (read_enable)
        read_data <= mem[addr];
end
```

Synchronous single-port RAM with write-first behavior

- Will result in single-port block RAM or LUT RAM when supported
- Block RAMs will require extra circuitry if write-first behavior not natively supported

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[addr] <= write_data;
    if (read_enable)
        if (write_enable)
            read_data <= write_data;
```

(continues on next page)

(continued from previous page)

```
        else
            read_data <= mem[addr];
    end
```

Synchronous read port with initial value

- Initial read port values can be combined with any other supported pattern
- If block RAM is used and initial read port values are not natively supported by the target, small emulation circuit will be inserted

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];
reg [DATA_WIDTH - 1 : 0] read_data;
initial read_data = 'h1234;

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
    if (read_enable)
        read_data <= mem[read_addr];
end
```

Read register reset patterns

Resets can be combined with any other supported pattern (except that synchronous reset and asynchronous reset cannot both be used on a single read port). If block RAM is used and the selected reset (synchronous or asynchronous) is used but not natively supported by the target, small emulation circuitry will be inserted.

Synchronous reset, reset priority over enable

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;

    if (read_reset)
        read_data <= 'h1234;
    else if (read_enable)
        read_data <= mem[read_addr];
end
```

Synchronous reset, enable priority over reset

```

reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
    if (read_enable)
        if (read_reset)
            read_data <= 'h1234;
        else
            read_data <= mem[read_addr];
end

```

Synchronous read port with asynchronous reset

```

reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
end

always @(posedge clk, posedge read_reset) begin
    if (read_reset)
        read_data <= 'h1234;
    else if (read_enable)
        read_data <= mem[read_addr];
end

```

Asymmetric memory patterns

To construct an asymmetric memory (memory with read/write ports of differing widths):

- Declare the memory with the width of the narrowest intended port
- Split all wide ports into multiple narrow ports
- To ensure the wide ports will be correctly merged:
 - For the address, use a concatenation of actual address in the high bits and a constant in the low bits
 - Ensure the actual address is identical for all ports belonging to the wide port
 - Ensure that clock is identical
 - For read ports, ensure that enable/reset signals are identical (for write ports, the enable signal may vary — this will result in using the byte enable functionality)

Asymmetric memory is supported on all targets, but may require emulation circuitry where not natively supported. Note that when the memory is larger than the underlying block RAM primitive, hardware asymmetric memory support is likely not to be used even if present as it is more expensive.

Wide synchronous read port

```
reg [7:0] mem [0:255];
wire [7:0] write_addr;
wire [5:0] read_addr;
wire [7:0] write_data;
reg [31:0] read_data;

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
    if (read_enable) begin
        read_data[7:0] <= mem[{read_addr, 2'b00}];
        read_data[15:8] <= mem[{read_addr, 2'b01}];
        read_data[23:16] <= mem[{read_addr, 2'b10}];
        read_data[31:24] <= mem[{read_addr, 2'b11}];
    end
end
```

Wide asynchronous read port

- Note: the only target natively supporting this pattern is Xilinx UltraScale

```
reg [7:0] mem [0:511];
wire [8:0] write_addr;
wire [5:0] read_addr;
wire [7:0] write_data;
wire [63:0] read_data;

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
end

assign read_data[7:0] = mem[{read_addr, 3'b000}];
assign read_data[15:8] = mem[{read_addr, 3'b001}];
assign read_data[23:16] = mem[{read_addr, 3'b010}];
assign read_data[31:24] = mem[{read_addr, 3'b011}];
assign read_data[39:32] = mem[{read_addr, 3'b100}];
assign read_data[47:40] = mem[{read_addr, 3'b101}];
assign read_data[55:48] = mem[{read_addr, 3'b110}];
assign read_data[63:56] = mem[{read_addr, 3'b111}];
```

Wide write port

```
reg [7:0] mem [0:255];
wire [5:0] write_addr;
wire [7:0] read_addr;
wire [31:0] write_data;
reg [7:0] read_data;

always @(posedge clk) begin
    if (write_enable[0])
        mem[{write_addr, 2'b00}] <= write_data[7:0];
    if (write_enable[1])
        mem[{write_addr, 2'b01}] <= write_data[15:8];
    if (write_enable[2])
        mem[{write_addr, 2'b10}] <= write_data[23:16];
    if (write_enable[3])
        mem[{write_addr, 2'b11}] <= write_data[31:24];
    if (read_enable)
        read_data <= mem[read_addr];
end
```

True dual port (TDP) patterns

- Many different variations of true dual port memory can be created by combining two single-port RAM patterns on the same memory
- When TDP memory is used, memory inference code has much less maneuver room to create requested semantics compared to individual single-port patterns (which can end up lowered to SDP memory where necessary) — supported patterns depend strongly on the target
- In particular, when both ports have the same clock, it's likely that “undefined collision” mode needs to be manually selected to enable TDP memory inference
- The examples below are non-exhaustive — many more combinations of port types are possible
- Note: if two write ports are in the same process, this defines a priority relation between them (if both ports are active in the same clock, the later one wins). On almost all targets, this will result in a bit of extra circuitry to ensure the priority semantics. If this is not what you want, put them in separate processes.
 - Priority is not supported when using the verilog front end and any priority semantics are ignored.

TDP with different clocks, exclusive read/write

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk_a) begin
    if (write_enable_a)
        mem[addr_a] <= write_data_a;
    else if (read_enable_a)
        read_data_a <= mem[addr_a];
end
```

(continues on next page)

(continued from previous page)

```
always @(posedge clk_b) begin
    if (write_enable_b)
        mem[addr_b] <= write_data_b;
    else if (read_enable_b)
        read_data_b <= mem[addr_b];
end
```

TDP with same clock, read-first behavior

- This requires hardware inter-port read-first behavior, and will only work on some targets (Xilinx, Nexus)

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable_a)
        mem[addr_a] <= write_data_a;
    if (read_enable_a)
        read_data_a <= mem[addr_a];
end

always @(posedge clk) begin
    if (write_enable_b)
        mem[addr_b] <= write_data_b;
    if (read_enable_b)
        read_data_b <= mem[addr_b];
end
```

TDP with multiple read ports

- The combination of a single write port with an arbitrary amount of read ports is supported on all targets — if a multi-read port primitive is available (like Xilinx RAM64M), it'll be used as appropriate. Otherwise, the memory will be automatically split into multiple primitives.

```
reg [31:0] mem [0:31];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
end

assign read_data_a = mem[read_addr_a];
assign read_data_b = mem[read_addr_b];
assign read_data_c = mem[read_addr_c];
```


Patterns only supported with Verific

The following patterns are only supported when the design is read in using the Verific front-end.

Synchronous SDP with write-first behavior via blocking assignments

- Use *sdp_wf* for compatibility with Yosys Verilog frontend.

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] = write_data;

    if (read_enable)
        read_data <= mem[read_addr];
end
```

Asymmetric memories via part selection

- Build wide ports out of narrow ports instead (see *wide_sr*) for compatibility with Yosys Verilog frontend.

```
reg [31:0] mem [2**ADDR_WIDTH - 1 : 0];

wire [1:0] byte_lane;
wire [7:0] write_data;

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr][byte_lane * 8 +: 8] <= write_data;

    if (read_enable)
        read_data <= mem[read_addr];
end
```

Undesired patterns

Asynchronous writes

- Not supported in modern FPGAs
- Not supported in yosys code anyhow

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @* begin
    if (write_enable)
        mem[write_addr] = write_data;
end
```

(continues on next page)

(continued from previous page)

```
assign read_data = mem[read_addr];
```

3.1.5 Optimization passes

Yosys employs a number of optimizations to generate better and cleaner results. This chapter outlines these optimizations.

Todo: “outlines these optimizations” or “outlines *some.*”?

The `opt` macro command

The Yosys pass `opt` runs a number of simple optimizations. This includes removing unused signals and cells and const folding. It is recommended to run this pass after each major step in the synthesis script. As listed in *opt - perform simple optimizations*, this macro command calls the following `opt_*` commands:

Listing 3.14: Passes called by `opt`

```
opt_expr
opt_merge -nomux

do
    opt_muxtree
    opt_reduce
    opt_merge
    opt_share (-full only)
    opt_dff (except when called with -noff)
    opt_clean
    opt_expr
while <changed design>
```

Constant folding and simple expression rewriting - `opt_expr`

Todo: unsure if this is too much detail and should be in *Yosys internals*

This pass performs constant folding on the internal combinational cell types described in *Internal cell library*. This means a cell with all constant inputs is replaced with the constant value this cell drives. In some cases this pass can also optimize cells with some constant inputs.

Table 3.1: Const folding rules for `$_AND_` cells as used in `opt_expr`.

A-Input	B-Input	Replacement
any	0	0
0	any	0
1	1	1
X/Z	X/Z	X
1	X/Z	X
X/Z	1	X
any	X/Z	0
X/Z	any	0
<i>a</i>	1	<i>a</i>
1	<i>b</i>	<i>b</i>

Table 3.1 shows the replacement rules used for optimizing an `$_AND_` gate. The first three rules implement the obvious const folding rules. Note that ‘any’ might include dynamic values calculated by other parts of the circuit. The following three lines propagate undef (X) states. These are the only three cases in which it is allowed to propagate an undef according to Sec. 5.1.10 of IEEE Std. 1364-2005 [A+06].

The next two lines assume the value 0 for undef states. These two rules are only used if no other substitutions are possible in the current module. If other substitutions are possible they are performed first, in the hope that the ‘any’ will change to an undef value or a 1 and therefore the output can be set to undef.

The last two lines simply replace an `$_AND_` gate with one constant-1 input with a buffer.

Besides this basic const folding the `opt_expr` pass can replace 1-bit wide `$eq` and `$ne` cells with buffers or not-gates if one input is constant. Equality checks may also be reduced in size if there are redundant bits in the arguments (i.e. bits which are constant on both inputs). This can, for example, result in a 32-bit wide constant like 255 being reduced to the 8-bit value of 8'11111111 if the signal being compared is only 8-bit as in *addr_gen module after opt_expr; clean* of *Synthesis starter*.

The `opt_expr` pass is very conservative regarding optimizing `$mux` cells, as these cells are often used to model decision-trees and breaking these trees can interfere with other optimizations.

Listing 3.15: example verilog for demonstrating `opt_expr`

```

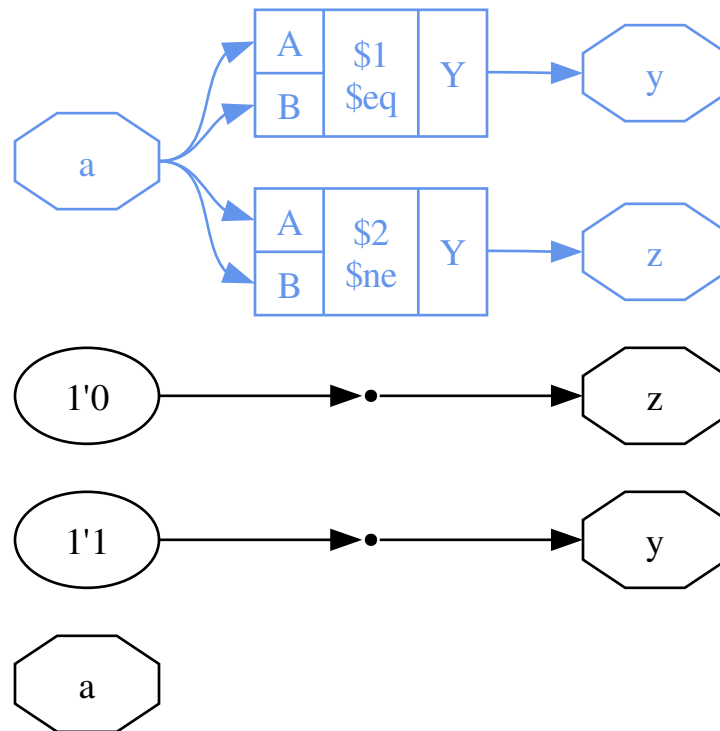
module uut(
    input a,
    output y, z
);
    assign y = a == a;
    assign z = a != a;
endmodule

```

Merging identical cells - `opt_merge`

This pass performs trivial resource sharing. This means that this pass identifies cells with identical inputs and replaces them with a single instance of the cell.

The option `-nomux` can be used to disable resource sharing for multiplexer cells (`$mux` and `$pmux`.) This can be useful as it prevents multiplexer trees to be merged, which might prevent `opt_muxtree` to identify possible optimizations.

Fig. 3.1: Before and after `opt_expr`

Listing 3.16: example verilog for demonstrating *opt_merge*

```

module uut(
  input  [3:0] a, b,
  output [3:0] y, z
);
  assign y = a + b;
  assign z = b + a;
endmodule

```

Fig. 3.2: Before and after *opt_merge*

Removing never-active branches from multiplexer tree - *opt_muxtree*

This pass optimizes trees of multiplexer cells by analyzing the select inputs. Consider the following simple example:

Listing 3.17: example verilog for demonstrating *opt_muxtree*

```

module uut(
  input a, b, c, d,
  output y
);
  assign y = a ? (a ? b : c) : d;
endmodule

```

The output can never be *c*, as this would require *a* to be 1 for the outer multiplexer and 0 for the inner

multiplexer. The *opt_muxtree* pass detects this contradiction and replaces the inner multiplexer with a constant 1, yielding the logic for $y = a ? b : d$.

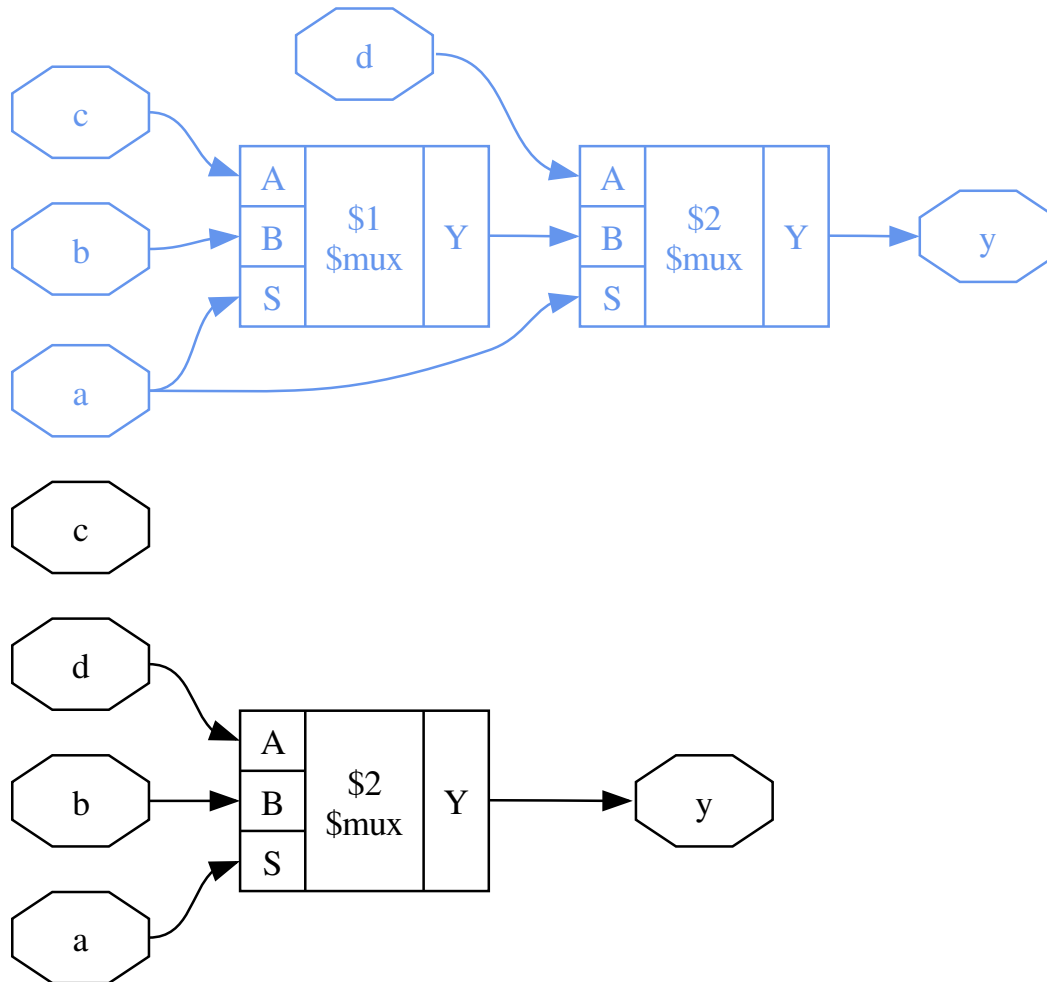


Fig. 3.3: Before and after *opt_muxtree*

Simplifying large MUXes and AND/OR gates - *opt_reduce*

This is a simple optimization pass that identifies and consolidates identical input bits to *\$reduce_and* and *\$reduce_or* cells. It also sorts the input bits to ease identification of shareable *\$reduce_and* and *\$reduce_or* cells in other passes.

This pass also identifies and consolidates identical inputs to multiplexer cells. In this case the new shared select bit is driven using a *\$reduce_or* cell that combines the original select bits.

Lastly this pass consolidates trees of *\$reduce_and* cells and trees of *\$reduce_or* cells to single large *\$reduce_and* or *\$reduce_or* cells.

These three simple optimizations are performed in a loop until a stable result is produced.

Merging mutually exclusive cells with shared inputs - *opt_share*

This pass identifies mutually exclusive cells of the same type that:

- share an input signal, and
- drive the same *\$mux*, *\$_MUX_*, or *\$pmux* multiplexing cell,

allowing the cell to be merged and the multiplexer to be moved from multiplexing its output to multiplexing the non-shared input signals.

Listing 3.18: example verilog for demonstrating *opt_share*

```
module uut(
  input  [15:0] a, b,
  input    sel,
  output [15:0] res,
);
  assign res = {sel ? a + b : a - b};
endmodule
```

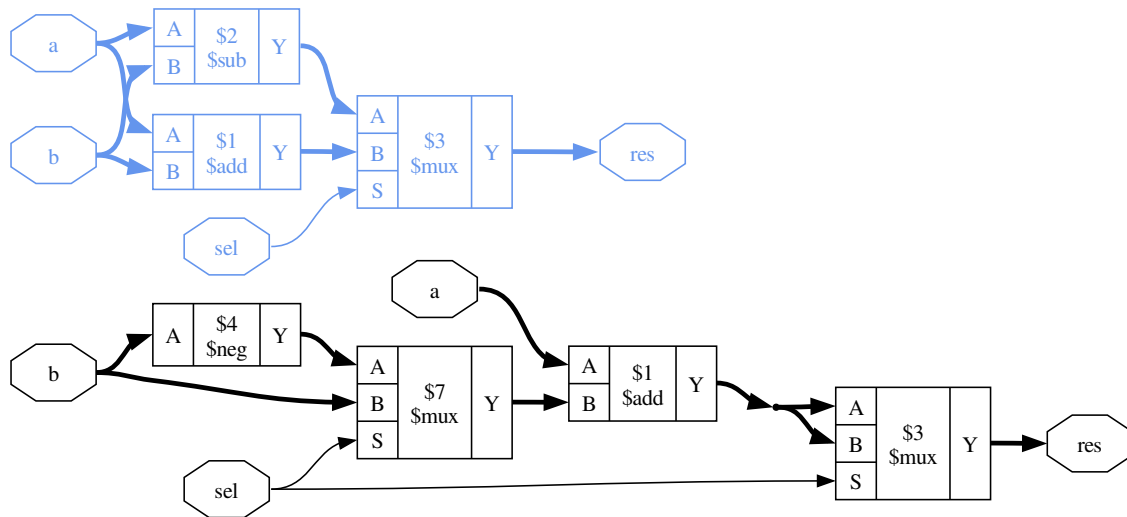


Fig. 3.4: Before and after *opt_share*

When running *opt* in full, the original *\$mux* (labeled \$3) is optimized away by *opt_expr*.

Performing DFF optimizations - `opt_dff`

Todo: `$_DFF_` isn't a valid cell

This pass identifies single-bit d-type flip-flops (`$_DFF_`, `$dff`, and `$adff` cells) with a constant data input and replaces them with a constant driver. It can also merge clock enables and synchronous reset multiplexers, removing unused control inputs.

Called with `-nodffe` and `-nosdff`, this pass is used to prepare a design for *FSM handling*.

Removing unused cells and wires - `opt_clean` pass

This pass identifies unused signals and cells and removes them from the design. It also creates an `unused_bits` attribute on wires with unused bits. This attribute can be used for debugging or by other optimization passes.

When to use `opt` or `clean`

Usually it does not hurt to call `opt` after each regular command in the synthesis script. But it increases the synthesis time, so it is favourable to only call `opt` when an improvement can be achieved.

It is generally a good idea to call `opt` before inherently expensive commands such as `sat` or `freduce`, as the possible gain is much higher in these cases as the possible loss.

The `clean` command, which is an alias for `opt_clean` with fewer outputs, on the other hand is very fast and many commands leave a mess (dangling signal wires, etc). For example, most commands do not remove any wires or cells. They just change the connections and depend on a later call to clean to get rid of the now unused objects. So the occasional `;;`, which itself is an alias for `clean`, is a good idea in every synthesis script, e.g:

```
hierarchy; proc; opt; memory; opt_expr;; fsm;;
```

Other optimizations

Todo: more on the other optimizations

- *wreduce* - reduce the word size of operations if possible
- *peepopt* - collection of peephole optimizers
- *share* - perform sat-based resource sharing
- *abc* and *abc9*, see also: *The ABC toolbox*.

3.1.6 Technology mapping

Todo: less academic, check text is coherent

Previous chapters outlined how HDL code is transformed into an RTL netlist. The RTL netlist is still based on abstract coarse-grain cell types like arbitrary width adders and even multipliers. This chapter covers how an RTL netlist is transformed into a functionally equivalent netlist utilizing the cell types available in the target architecture.

Technology mapping is often performed in two phases. In the first phase RTL cells are mapped to an internal library of single-bit cells (see *Gate-level cells*). In the second phase this netlist of internal gate types is transformed to a netlist of gates from the target technology library.

When the target architecture provides coarse-grain cells (such as block ram or ALUs), these must be mapped to directly form the RTL netlist, as information on the coarse-grain structure of the design is lost when it is mapped to bit-width gate types.

Cell substitution

The simplest form of technology mapping is cell substitution, as performed by the techmap pass. This pass, when provided with a Verilog file that implements the RTL cell types using simpler cells, simply replaces the RTL cells with the provided implementation.

When no map file is provided, techmap uses a built-in map file that maps the Yosys RTL cell types to the internal gate library used by Yosys. The curious reader may find this map file as `techlibs/common/techmap.v` in the Yosys source tree.

Additional features have been added to techmap to allow for conditional mapping of cells (see *techmap - generic technology mapper*). This can for example be useful if the target architecture supports hardware multipliers for certain bit-widths but not for others.

A usual synthesis flow would first use the techmap pass to directly map some RTL cells to coarse-grain cells provided by the target architecture (if any) and then use techmap with the built-in default file to map the remaining RTL cells to gate logic.

Subcircuit substitution

Sometimes the target architecture provides cells that are more powerful than the RTL cells used by Yosys. For example a cell in the target architecture that can calculate the absolute-difference of two numbers does not match any single RTL cell type but only combinations of cells.

For these cases Yosys provides the extract pass that can match a given set of modules against a design and identify the portions of the design that are identical (i.e. isomorphic subcircuits) to any of the given modules. These matched subcircuits are then replaced by instances of the given modules.

The extract pass also finds basic variations of the given modules, such as swapped inputs on commutative cell types.

In addition to this the extract pass also has limited support for frequent subcircuit mining, i.e. the process of finding recurring subcircuits in the design. This has a few applications, including the design of new coarse-grain architectures [GW13].

The hard algorithmic work done by the extract pass (solving the isomorphic subcircuit problem and frequent subcircuit mining) is performed using the SubCircuit library that can also be used stand-alone without Yosys (see *SubCircuit*).

Gate-level technology mapping

Todo: newer techmap libraries appear to be largely `.v` instead of `.lib`

On the gate-level the target architecture is usually described by a “Liberty file”. The Liberty file format is an industry standard format that can be used to describe the behaviour and other properties of standard library cells.

Mapping a design utilizing the Yosys internal gate library (e.g. as a result of mapping it to this representation using the techmap pass) is performed in two phases.

First the register cells must be mapped to the registers that are available on the target architectures. The target architecture might not provide all variations of d-type flip-flops with positive and negative clock edge, high-active and low-active asynchronous set and/or reset, etc. Therefore the process of mapping the registers might add additional inverters to the design and thus it is important to map the register cells first.

Mapping of the register cells may be performed by using the `dfflibmap` pass. This pass expects a Liberty file as argument (using the `-liberty` option) and only uses the register cells from the Liberty file.

Secondly the combinational logic must be mapped to the target architecture. This is done using the external program ABC via the `abc` pass by using the `-liberty` option to the pass. Note that in this case only the combinatorial cells are used from the cell library.

Occasionally Liberty files contain trade secrets (such as sensitive timing information) that cannot be shared freely. This complicates processes such as reporting bugs in the tools involved. When the information in the Liberty file used by Yosys and ABC are not part of the sensitive information, the additional tool `yosys-filterlib` (see [yosys-filterlib](#)) can be used to strip the sensitive information from the Liberty file.

3.1.7 The extract pass

- Like the `techmap` pass, the `extract` pass is called with a map file. It compares the circuits inside the modules of the map file with the design and looks for sub-circuits in the design that match any of the modules in the map file.
- If a match is found, the `extract` pass will replace the matching subcircuit with an instance of the module from the map file.
- In a way the `extract` pass is the inverse of the techmap pass.

Todo: add/expand supporting text, also mention custom pattern matching and pmgen

Example code can be found in `docs/source/code_examples/macc`.

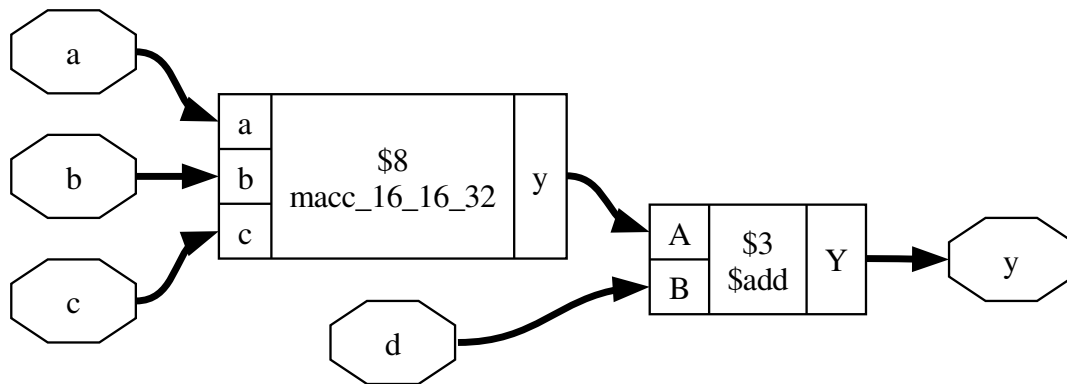
```
read_verilog macc_simple_test.v
hierarchy -check -top test;;
```

```
extract -constports -map macc_simple_xmap.v;;
```

Listing 3.19: `macc_simple_test.v`

```
module test(a, b, c, d, y);
input [15:0] a, b;
input [31:0] c, d;
```

(continues on next page)

Fig. 3.5: before *extract*Fig. 3.6: after *extract*

(continued from previous page)

```

output [31:0] y;
assign y = a * b + c + d;
endmodule

```

Listing 3.20: macc_simple_xmap.v

```

module macc_16_16_32(a, b, c, y);
input [15:0] a, b;
input [31:0] c;
output [31:0] y;
assign y = a*b + c;
endmodule

```

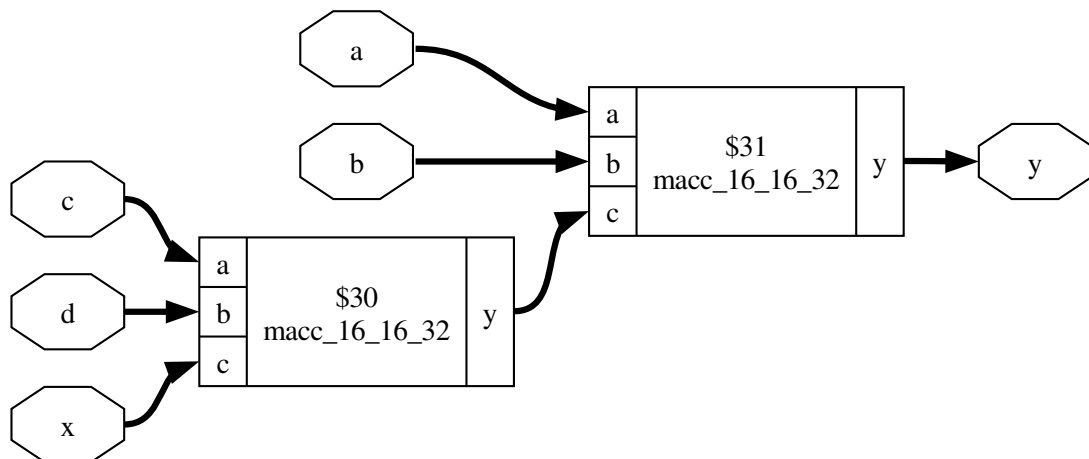
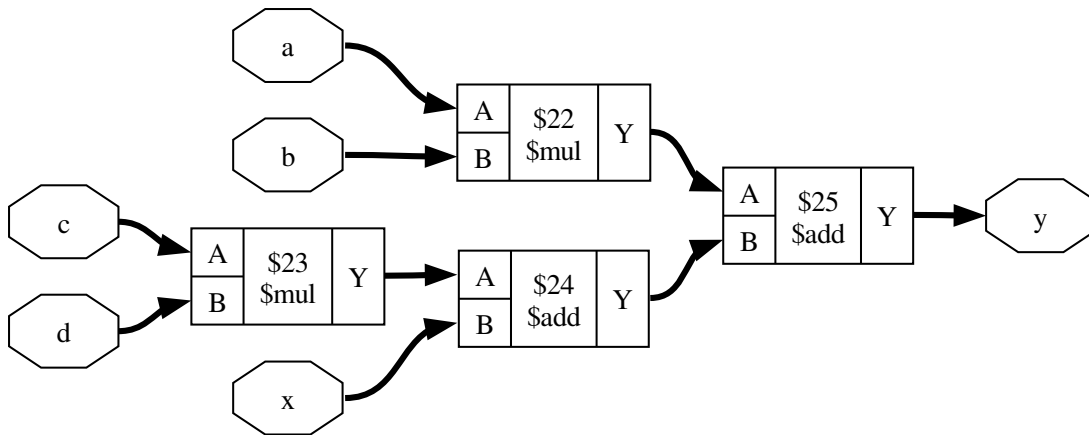
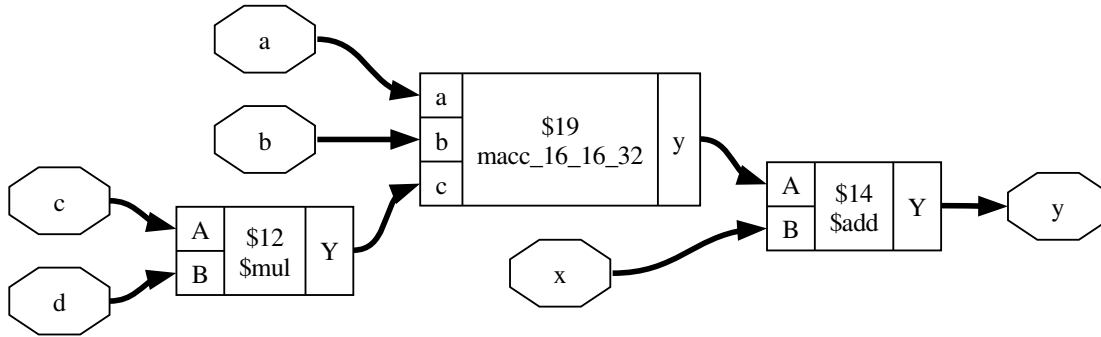
Listing 3.21: macc_simple_test_01.v

```

module test(a, b, c, d, x, y);
input [15:0] a, b, c, d;
input [31:0] x;
output [31:0] y;
assign y = a*b + c*d + x;
endmodule

```





Listing 3.22: `macc_simple_test_02.v`

```
module test(a, b, c, d, x, y);
input  [15:0] a, b, c, d;
input  [31:0] x;
output [31:0] y;
assign y = a*b + (c*d + x);
endmodule
```

The wrap-extract-unwrap method

Often a coarse-grain element has a constant bit-width, but can be used to implement operations with a smaller bit-width. For example, a 18x25-bit multiplier can also be used to implement 16x20-bit multiplication.

A way of mapping such elements in coarse grain synthesis is the wrap-extract-unwrap method:

wrap

Identify candidate-cells in the circuit and wrap them in a cell with a constant wider bit-width using *techmap*. The wrappers use the same parameters as the original cell, so the information about the original width of the ports is preserved. Then use the *connwrappers* command to connect up the bit-extended in- and outputs of the wrapper cells.

extract

Now all operations are encoded using the same bit-width as the coarse grain element. The *extract* command can be used to replace circuits with cells of the target architecture.

unwrap

The remaining wrapper cell can be unwrapped using *techmap*.

Example: DSP48_MACC

This section details an example that shows how to map MACC operations of arbitrary size to MACC cells with a 18x25-bit multiplier and a 48-bit adder (such as the Xilinx DSP48 cells).

Preconditioning: `macc_xilinx_swap_map.v`

Make sure **A** is the smaller port on all multipliers

Todo: add/expand supporting text

Listing 3.23: `macc_xilinx_swap_map.v`

```
(* techmap_celltype = "$mul" *)
module mul_swap_ports (A, B, Y);

parameter A_SIGNED = 0;
parameter B_SIGNED = 0;
parameter A_WIDTH  = 1;
parameter B_WIDTH  = 1;
parameter Y_WIDTH  = 1;

input [A_WIDTH-1:0] A;
```

(continues on next page)

(continued from previous page)

```

input [B_WIDTH-1:0] B;
output [Y_WIDTH-1:0] Y;

wire _TECHMAP_FAIL_ = A_WIDTH <= B_WIDTH;

\${mul} #(
    .A_SIGNED(B_SIGNED),
    .B_SIGNED(A_SIGNED),
    .A_WIDTH(B_WIDTH),
    .B_WIDTH(A_WIDTH),
    .Y_WIDTH(Y_WIDTH)
) _TECHMAP_REPLACE_ (
    .A(B),
    .B(A),
    .Y(Y)
);

endmodule

```

Wrapping multipliers: macc_xilinx_wrap_map.v

Listing 3.24: macc_xilinx_wrap_map.v

```

(* techmap_celltype = "${mul}" *)
module mul_wrap (A, B, Y);

parameter A_SIGNED = 0;
parameter B_SIGNED = 0;
parameter A_WIDTH = 1;
parameter B_WIDTH = 1;
parameter Y_WIDTH = 1;

input [A_WIDTH-1:0] A;
input [B_WIDTH-1:0] B;
output [Y_WIDTH-1:0] Y;

wire [17:0] A_18 = A;
wire [24:0] B_25 = B;
wire [47:0] Y_48;
assign Y = Y_48;

wire [1023:0] _TECHMAP_DO_ = "proc; clean";

reg _TECHMAP_FAIL_;
initial begin
    _TECHMAP_FAIL_ <= 0;
    if (A_SIGNED || B_SIGNED)
        _TECHMAP_FAIL_ <= 1;
    if (A_WIDTH < 4 || B_WIDTH < 4)
        _TECHMAP_FAIL_ <= 1;
    if (A_WIDTH > 18 || B_WIDTH > 25)
        _TECHMAP_FAIL_ <= 1;
end

```

(continues on next page)

(continued from previous page)

```

        if (A_WIDTH*B_WIDTH < 100)
            _TECHMAP_FAIL_ <= 1;
    end

    \$_mul_wrapper #(
        .A_SIGNED(A_SIGNED),
        .B_SIGNED(B_SIGNED),
        .A_WIDTH(A_WIDTH),
        .B_WIDTH(B_WIDTH),
        .Y_WIDTH(Y_WIDTH)
    ) _TECHMAP_REPLACE_ (
        .A(A_18),
        .B(B_25),
        .Y(Y_48)
    );

endmodule

```

Wrapping adders: `macc_xilinx_wrap_map.v`

Listing 3.25: `macc_xilinx_wrap_map.v`

```

(* techmap_celltype = "$add" *)
module add_wrap (A, B, Y);

    parameter A_SIGNED = 0;
    parameter B_SIGNED = 0;
    parameter A_WIDTH = 1;
    parameter B_WIDTH = 1;
    parameter Y_WIDTH = 1;

    input [A_WIDTH-1:0] A;
    input [B_WIDTH-1:0] B;
    output [Y_WIDTH-1:0] Y;

    wire [47:0] A_48 = A;
    wire [47:0] B_48 = B;
    wire [47:0] Y_48;
    assign Y = Y_48;

    wire [1023:0] _TECHMAP_DO_ = "proc; clean";

    reg _TECHMAP_FAIL_;
    initial begin
        _TECHMAP_FAIL_ <= 0;
        if (A_SIGNED || B_SIGNED)
            _TECHMAP_FAIL_ <= 1;
        if (A_WIDTH < 10 && B_WIDTH < 10)
            _TECHMAP_FAIL_ <= 1;
    end

    \$_add_wrapper #(

```

(continues on next page)

(continued from previous page)

```

        .A_SIGNED(A_SIGNED),
        .B_SIGNED(B_SIGNED),
        .A_WIDTH(A_WIDTH),
        .B_WIDTH(B_WIDTH),
        .Y_WIDTH(Y_WIDTH)
    ) _TECHMAP_REPLACE_ (
        .A(A_48),
        .B(B_48),
        .Y(Y_48)
    );
endmodule

```

Extract: macc_xilinx_xmap.v

Listing 3.26: macc_xilinx_xmap.v

```

module DSP48_MACC (a, b, c, y);

input [17:0] a;
input [24:0] b;
input [47:0] c;
output [47:0] y;

assign y = a*b + c;

endmodule

```

... simply use the same wrapping commands on this module as on the design to create a template for the *extract* command.

Unwrapping multipliers: macc_xilinx_unwrap_map.v

Listing 3.27: `$_mul_wrapper` module in
macc_xilinx_unwrap_map.v

```

module \$_mul_wrapper (A, B, Y);

parameter A_SIGNED = 0;
parameter B_SIGNED = 0;
parameter A_WIDTH = 1;
parameter B_WIDTH = 1;
parameter Y_WIDTH = 1;

input [17:0] A;
input [24:0] B;
output [47:0] Y;

wire [A_WIDTH-1:0] A_ORIG = A;
wire [B_WIDTH-1:0] B_ORIG = B;
wire [Y_WIDTH-1:0] Y_ORIG;
assign Y = Y_ORIG;

```

(continues on next page)

(continued from previous page)

```

\ $mul #(
    .A_SIGNED(A_SIGNED),
    .B_SIGNED(B_SIGNED),
    .A_WIDTH(A_WIDTH),
    .B_WIDTH(B_WIDTH),
    .Y_WIDTH(Y_WIDTH)
) _TECHMAP_REPLACE_ (
    .A(A_ORIG),
    .B(B_ORIG),
    .Y(Y_ORIG)
);

endmodule

```

Unwrapping adders: `macc_xilinx_unwrap_map.v`

Listing 3.28: `$_add_wrapper` module in `macc_xilinx_unwrap_map.v`

```

module \$_add_wrapper (A, B, Y);

parameter A_SIGNED = 0;
parameter B_SIGNED = 0;
parameter A_WIDTH = 1;
parameter B_WIDTH = 1;
parameter Y_WIDTH = 1;

input [47:0] A;
input [47:0] B;
output [47:0] Y;

wire [A_WIDTH-1:0] A_ORIG = A;
wire [B_WIDTH-1:0] B_ORIG = B;
wire [Y_WIDTH-1:0] Y_ORIG;
assign Y = Y_ORIG;

\ $add #(
    .A_SIGNED(A_SIGNED),
    .B_SIGNED(B_SIGNED),
    .A_WIDTH(A_WIDTH),
    .B_WIDTH(B_WIDTH),
    .Y_WIDTH(Y_WIDTH)
) _TECHMAP_REPLACE_ (
    .A(A_ORIG),
    .B(B_ORIG),
    .Y(Y_ORIG)
);

endmodule

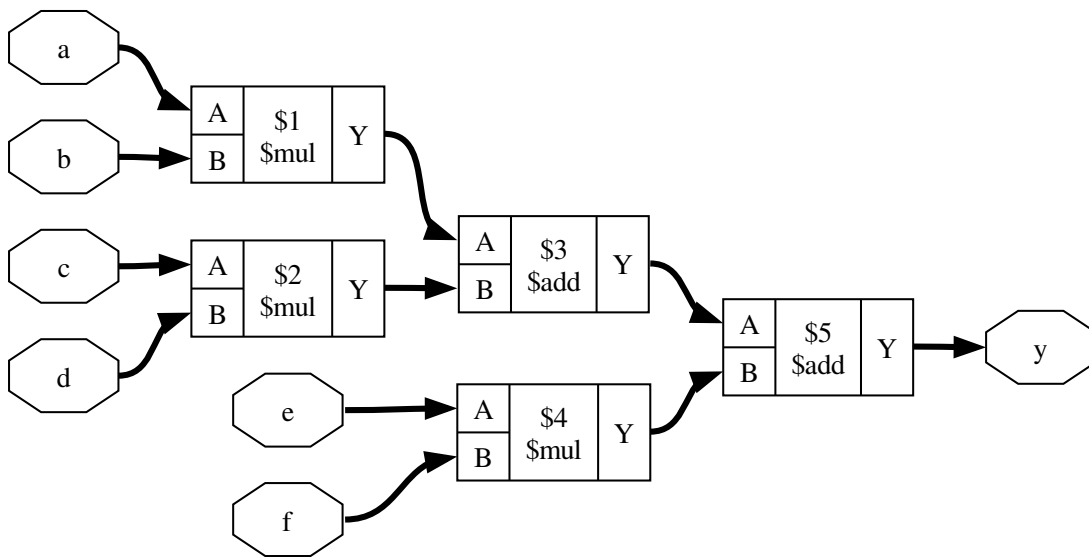
```

Listing 3.29: test1 of macc_xilinx_test.v

```

module test1(a, b, c, d, e, f, y);
  input [19:0] a, b, c;
  input [15:0] d, e, f;
  output [41:0] y;
  assign y = a*b + c*d + e*f;
endmodule

```



Listing 3.30: test2 of macc_xilinx_test.v

```

module test2(a, b, c, d, e, f, y);
  input [19:0] a, b, c;
  input [15:0] d, e, f;
  output [41:0] y;
  assign y = a*b + (c*d + e*f);
endmodule

```

Wrapping in test1:

```

techmap -map macc_xilinx_wrap_map.v

connwrappers -unsigned $__mul_wrapper Y Y_WIDTH \
              -unsigned $__add_wrapper Y Y_WIDTH;;

```

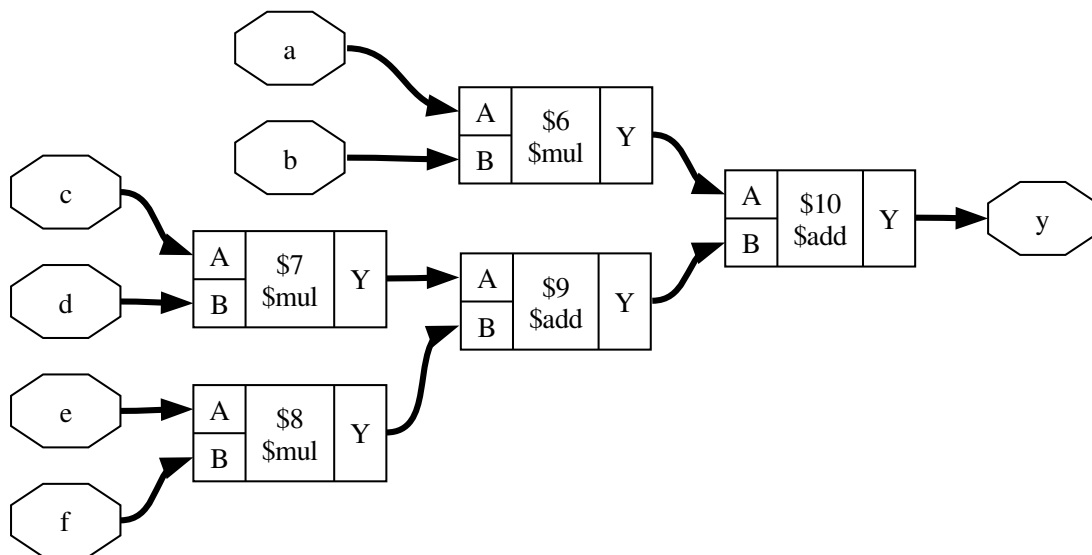
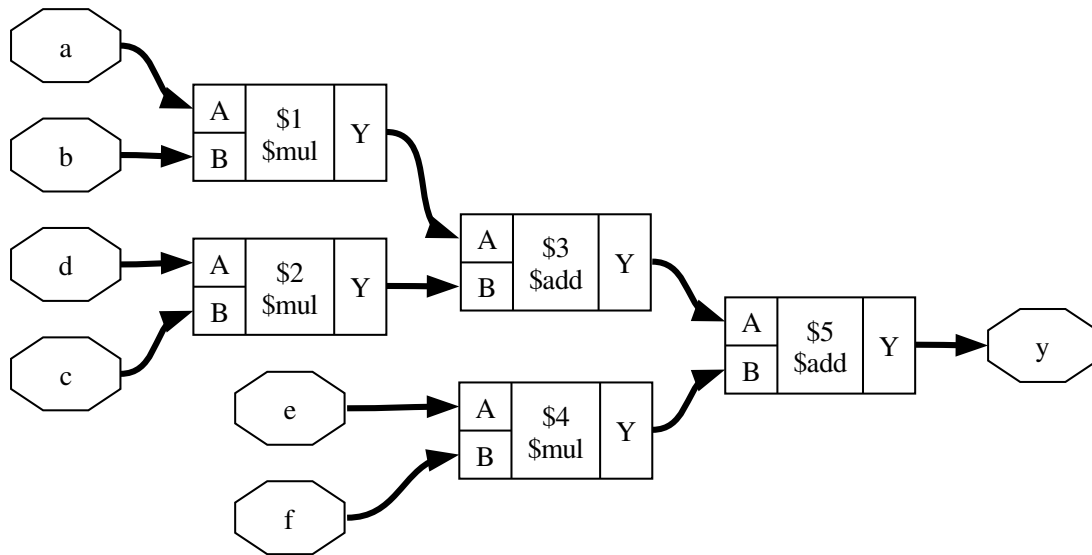
Wrapping in test2:

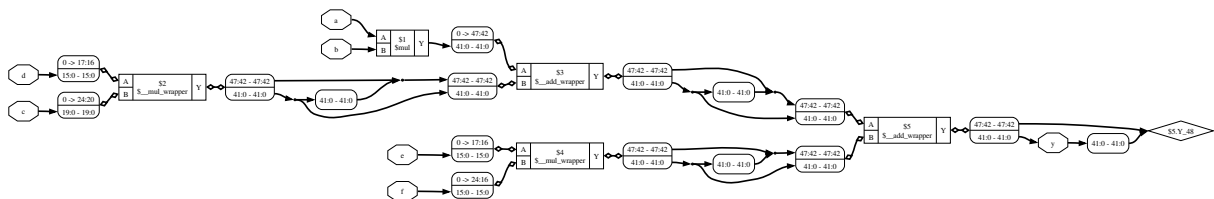
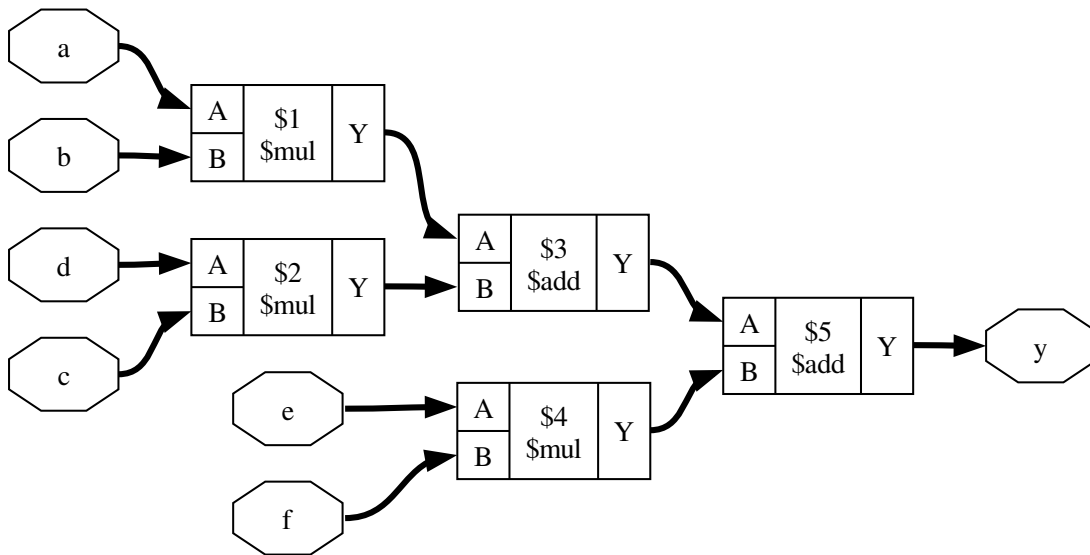
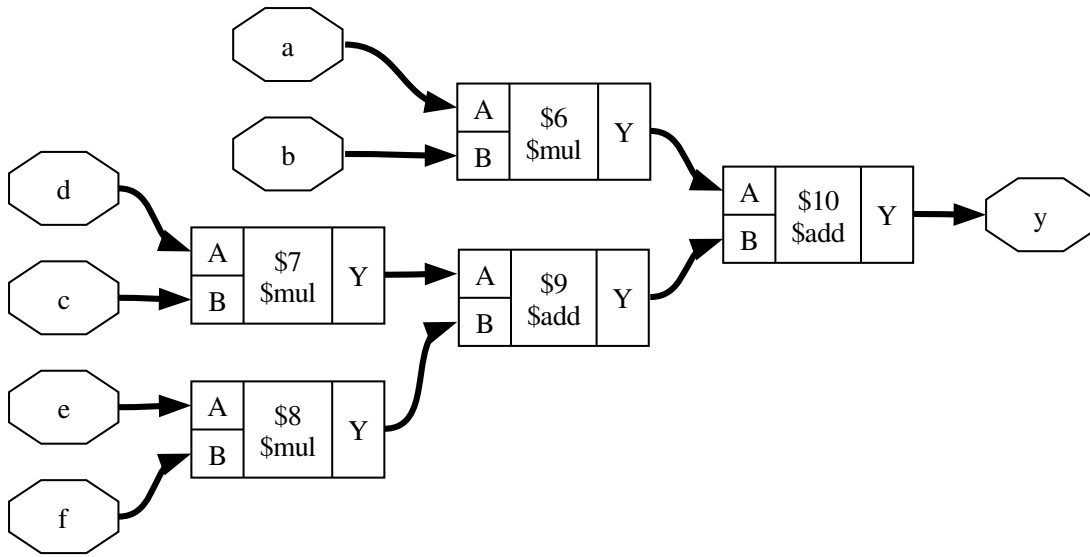
```

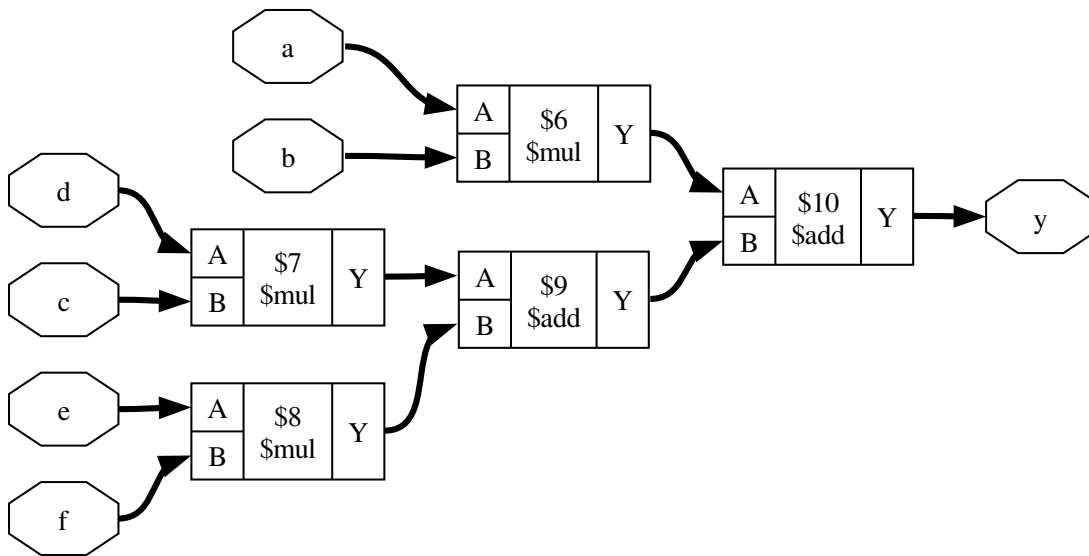
techmap -map macc_xilinx_wrap_map.v

```

(continues on next page)

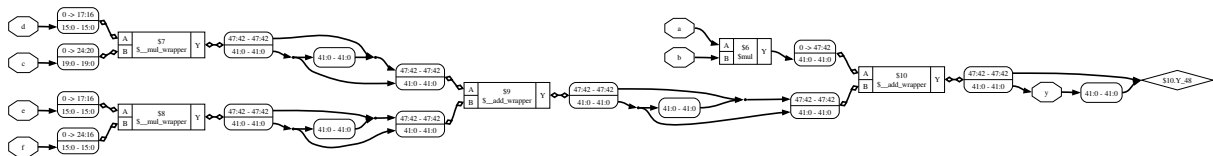




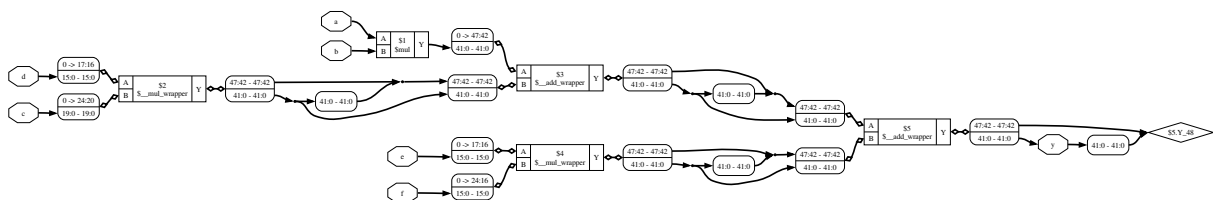


(continued from previous page)

```
connwrappers -unsigned $_mul_wrapper Y Y_WIDTH \
              -unsigned $_add_wrapper Y Y_WIDTH;;
```

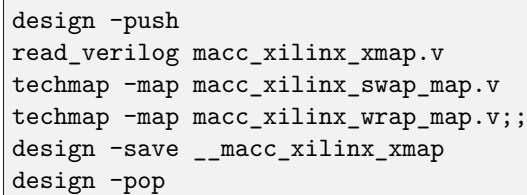
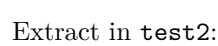


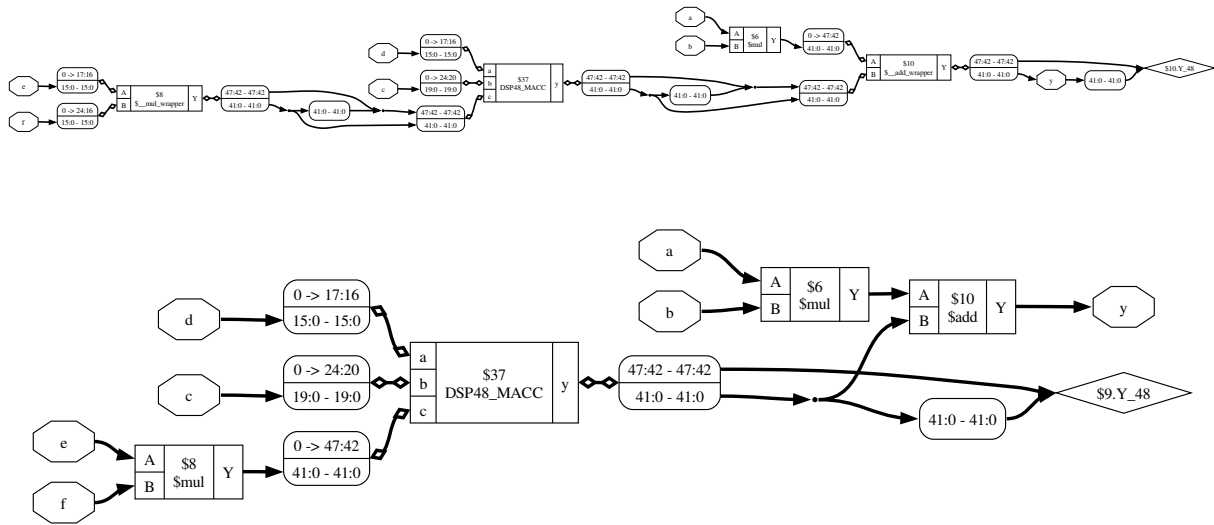
Extract in test1:



```
design -push
read_verilog macc_xilinx_xmap.v
techmap -map macc_xilinx_swap_map.v
techmap -map macc_xilinx_wrap_map.v;;
design -save __macc_xilinx_xmap
design -pop
```

(continues on next page)





3.1.8 The ABC toolbox

ABC, from the University of California, Berkeley, is a logic toolbox used for fine-grained optimisation and LUT mapping.

Yosys has two different commands, which both use this logic toolbox, but use it in different ways.

The `abc` pass can be used for both ASIC (e.g. `abc -liberty`) and FPGA (`abc -lut`) mapping, but this page will focus on FPGA mapping.

The `abc9` pass generally provides superior mapping quality due to being aware of combination boxes and DFF and LUT timings, giving it a more global view of the mapping problem.

ABC: the unit delay model, simple and efficient

The `abc` pass uses a highly simplified view of an FPGA:

- An FPGA is made up of a network of inputs that connect through LUTs to a network of outputs. These inputs may actually be I/O pins, D flip-flops, memory blocks or DSPs, but ABC is unaware of this.
- Each LUT has 1 unit of delay between an input and its output, and this applies for all inputs of a LUT, and for all sizes of LUT up to the maximum LUT size allowed; e.g. the delay between the input of a LUT2 and its output is the same as the delay between the input of a LUT6 and its output.
- A LUT may take up a variable number of area units. This is constant for each size of LUT; e.g. a LUT4 may take up 1 unit of area, but a LUT5 may take up 2 units of area, but this applies for all LUT4s and LUT5s.

This is known as the “unit delay model”, because each LUT uses one unit of delay.

From this view, the problem ABC has to solve is finding a mapping of the network to LUTs that has the lowest delay, and then optimising the mapping for size while maintaining this delay.

This approach has advantages:

- It is simple and easy to implement.
- Working with unit delays is fast to manipulate.

- It reflects *some* FPGA families, for example, the iCE40HX/LP fits the assumptions of the unit delay model quite well (almost all synchronous blocks, except for adders).

But this approach has drawbacks, too:

- The network of inputs and outputs with only LUTs means that a lot of combinational cells (multipliers and LUTRAM) are invisible to the unit delay model, meaning the critical path it optimises for is not necessarily the actual critical path.
- LUTs are implemented as multiplexer trees, so there is a delay caused by the result propagating through the remaining multiplexers. This means the assumption of delay being equal isn't true in physical hardware, and is proportionally larger for larger LUTs.
- Even synchronous blocks have arrival times (propagation delay between clock edge to output changing) and setup times (requirement for input to be stable before clock edge) which affect the delay of a path.

ABC9: the generalised delay model, realistic and flexible

ABC9 uses a more detailed and accurate model of an FPGA:

- An FPGA is made up of a network of inputs that connect through LUTs and combinational boxes to a network of outputs. These boxes have specified delays between inputs and outputs, and may have an associated network (“white boxes”) or not (“black boxes”), but must be treated as a whole.
- Each LUT has a specified delay between an input and its output in arbitrary delay units, and this varies for all inputs of a LUT and for all sizes of LUT, but each size of LUT has the same associated delay; e.g. the delay between input A and output is different between a LUT2 and a LUT6, but is constant for all LUT6s.
- A LUT may take up a variable number of area units. This is constant for each size of LUT; e.g. a LUT4 may take up 1 unit of area, but a LUT5 may take up 2 units of area, but this applies for all LUT4s and LUT5s.

This is known as the “generalised delay model”, because it has been generalised to arbitrary delay units. ABC9 doesn't actually care what units you use here, but the Yosys convention is picoseconds. Note the introduction of boxes as a concept. While the generalised delay model does not require boxes, they naturally fit into it to represent combinational delays. Even synchronous delays like arrival and setup can be emulated with combinational boxes that act as a delay. This is further extended to white boxes, where the mapper is able to see inside a box, and remove orphan boxes with no outputs, such as adders.

Again, ABC9 finds a mapping of the network to LUTs that has the lowest delay, and then minimises it to find the lowest area, but it has a lot more information to work with about the network.

The result here is that ABC9 can remove boxes (like adders) to reduce area, optimise better around those boxes, and also permute inputs to give the critical path the fastest inputs.

Todo: more about logic minimization & register balancing et al with ABC

Setting up a flow for ABC9

Much of the configuration comes from attributes and `specify` blocks in Verilog simulation models.

`specify` syntax

Since `specify` is a relatively obscure part of the Verilog standard, a quick guide to the syntax:

```
specify                                     // begins a specify block
  (A => B) = 123;                           // simple combinational path from A to B with a delay
  ↪ of 123.
  (A *> B) = 123;                           // simple combinational path from A to all bits of B
  ↪ with a delay of 123 for all.
  if (FOO) (A => B) = 123;                  // paths may apply under specific conditions.
  (posedge CLK => (Q : D)) = 123;           // combinational path triggered on the positive edge
  ↪ of CLK; used for clock-to-Q arrival paths.
  $setup(A, posedge CLK, 123);             // setup constraint for an input relative to a clock.
endspecify                                 // ends a specify block
```

By convention, all delays in `specify` blocks are in integer picoseconds. Files containing `specify` blocks should be read with the `-specify` option to `read_verilog` so that they aren't skipped.

LUTs

LUTs need to be annotated with an `(* abc9_lut=N *)` attribute, where `N` is the relative area of that LUT model. For example, if an architecture can combine LUTs to produce larger LUTs, then the combined LUTs would have increasingly larger `N`. Conversely, if an architecture can split larger LUTs into smaller LUTs, then the smaller LUTs would have smaller `N`.

LUTs are generally specified with simple combinational paths from the LUT inputs to the LUT output.

DFFs

DFFs should be annotated with an `(* abc9_flop *)` attribute, however ABC9 has some specific requirements for this to be valid: - the DFF must initialise to zero (consider using `dfflegalize` to ensure this). - the DFF cannot have any asynchronous resets/sets (see the simplification idiom and the Boxes section for what to do here).

It is worth noting that in pure `abc9` mode, only the setup and arrival times are passed to ABC9 (specifically, they are modelled as buffers with the given delay). In `abc9 -dff`, the flop itself is passed to ABC9, permitting sequential optimisations.

Some vendors have universal DFF models which include async sets/resets even when they're unused. Therefore the *simplification idiom* exists to handle this: by using a `techmap` file to discover flops which have a constant driver to those asynchronous controls, they can be mapped into an intermediate, simplified flop which qualifies as an `(* abc9_flop *)`, ran through `abc9`, and then mapped back to the original flop. This is used in `synth_intel_alm` and `synth_quicklogic` for the PolarPro3.

DFFs are usually specified to have setup constraints against the clock on the input signals, and an arrival time for the `Q` output.

Boxes

A “box” is a purely-combinational piece of hard logic. If the logic is exposed to ABC9, it’s a “whitebox”, otherwise it’s a “blackbox”. Carry chains would be best implemented as whiteboxes, but a DSP would be best implemented as a blackbox (multipliers are too complex to easily work with). LUT RAMs can be implemented as whiteboxes too.

Boxes are arguably the biggest advantage that ABC9 has over ABC: by being aware of carry chains and DSPs, it avoids optimising for a path that isn’t the actual critical path, while the generally-longer paths result in ABC9 being able to reduce design area by mapping other logic to larger-but-slower cells.

3.1.9 Mapping to cell libraries

While much of this documentation focuses on the use of Yosys with FPGAs, it is also possible to map to cell libraries which can be used in designing ASICs. This section will cover a brief [example project](#), available in the Yosys source code under `docs/source/code_examples/intro/`. The project contains a simple ASIC synthesis script (`counter.py`), a digital design written in Verilog (`counter.v`), and a simple CMOS cell library (`mycells.lib`). Many of the early steps here are already covered in more detail in the [Synthesis starter](#) document.

Note: The `counter.py` script includes the commands used to generate the images in this document. Code snippets in this document skip these commands; including line numbers to allow the reader to follow along with the source.

To learn more about these commands, check out [A look at the show command](#).

A simple counter

First, let’s quickly look at the design:

Listing 3.31: `counter.v`

```

1 module counter (clk, rst, en, count);
2
3     input clk, rst, en;
4     output reg [1:0] count;
5
6     always @(posedge clk)
7         if (rst)
8             count <= 2'd0;
9         else if (en)
10            count <= count + 2'd1;
11
12 endmodule

```

This is a simple counter with reset and enable. If the reset signal, `rst`, is high then the counter will reset to 0. Otherwise, if the enable signal, `en`, is high then the `count` register will increment by 1 each rising edge of the clock, `clk`.

Loading the design

Listing 3.32: counter.ys - read design

```
1 # read design
2 read_verilog counter.v
3 hierarchy -check -top counter
```

Our circuit now looks like this:

Fig. 3.7: counter after *hierarchy*

Coarse-grain representation

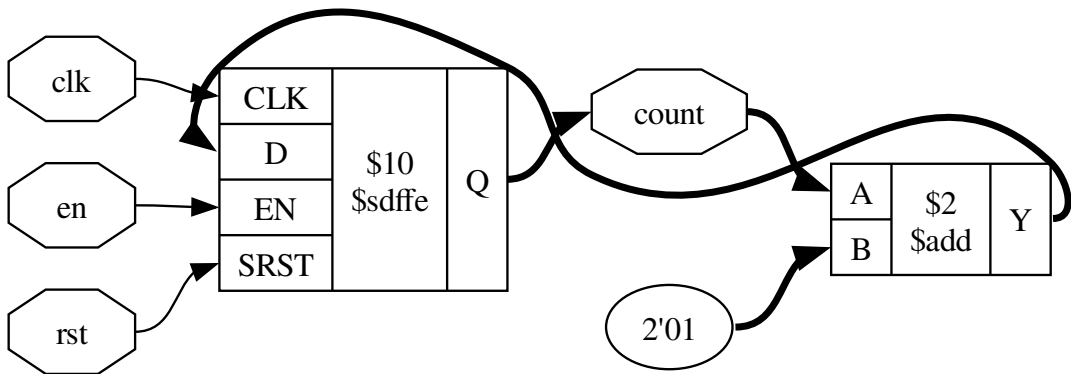


Fig. 3.8: Coarse-grain representation of the counter module

Listing 3.33: counter.y_s - the high-level stuff

```
7 # the high-level stuff
8 proc; opt
9 memory; opt
10 fsm; opt
```

Logic gate mapping

Listing 3.34: counter.y_s - mapping to internal cell library

```
14 # mapping to internal cell library
15 techmap; opt
```

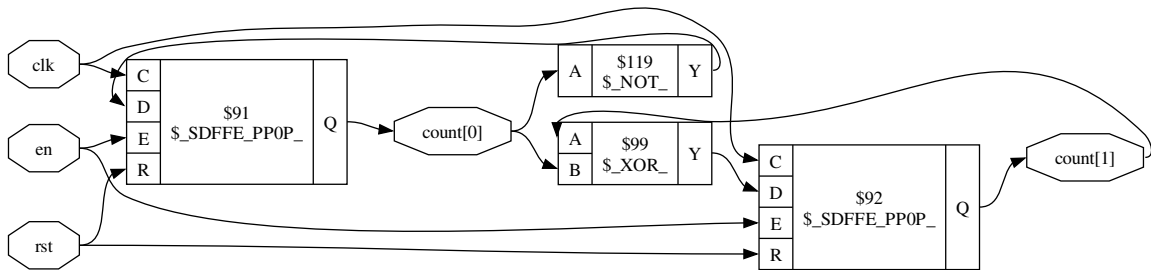


Fig. 3.9: counter after *techmap*

Mapping to hardware

For this example, we are using a Liberty file to describe a cell library which our internal cell library will be mapped to:

Todo: find a Liberty pygments style?

Listing 3.35: mycells.lib

```
1 library(demo) {
2   cell(BUF) {
3     area: 6;
4     pin(A) { direction: input; }
5     pin(Y) { direction: output;
6             function: "A"; }
7   }
8   cell(NOT) {
9     area: 3;
10    pin(A) { direction: input; }
11    pin(Y) { direction: output;
12            function: "A'"; }
13  }
14  cell(NAND) {
15    area: 4;
16    pin(A) { direction: input; }
17    pin(B) { direction: input; }
18    pin(Y) { direction: output;
19            function: "(A*B)"; }
20  }
21  cell(NOR) {
22    area: 4;
23    pin(A) { direction: input; }
24    pin(B) { direction: input; }
25    pin(Y) { direction: output;
26            function: "(A+B)"; }
27  }
28  cell(DFF) {
29    area: 18;
30    ff(IQ, IQN) { clocked_on: C;
31                 next_state: D; }
32    pin(C) { direction: input;
33            clock: true; }
34    pin(D) { direction: input; }
35    pin(Q) { direction: output;
36            function: "IQ"; }
37  }
38 }
```

Recall that the Yosys built-in logic gate types are `$_NOT_`, `$_AND_`, `$_OR_`, `$_XOR_`, and `$_MUX_` with an assortment of dff memory types. *mycells.lib* defines our target cells as BUF, NOT, NAND, NOR, and DFF. Mapping between these is performed with the commands *dfplibmap* and *abc* as follows:

Listing 3.36: counter.y - mapping to hardware

```
20 dfflibmap -liberty mycells.lib
21
22 # mapping logic to mycells.lib
23 abc -liberty mycells.lib
24
25 # cleanup
26 clean
```

The final version of our counter module looks like this:

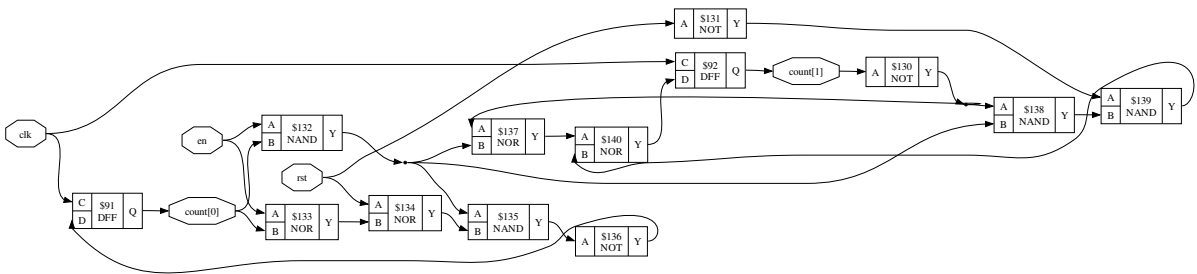


Fig. 3.10: counter after hardware cell mapping

Before finally being output as a verilog file with `write_verilog`, which can then be loaded into another tool:

Listing 3.37: counter.y - write synthesized design

```
30 # write synthesized design
31 write_verilog synth.v
```

3.2 More scripting

Todo: brief overview for the more scripting index

Todo: troubleshooting document(?)

3.2.1 Loading a design

keyword: Frontends

- *read_verilog* - read modules from Verilog file

Todo: include `read_verilog` <<EOF, also other methods of loading designs

```
read_verilog file1.v
read_verilog -I include_dir -D enable_foo -D WIDTH=12 file2.v
read_verilog -lib cell_library.v

verilog_defaults -add -I include_dir
read_verilog file3.v
read_verilog file4.v
verilog_defaults -clear

verilog_defaults -push
verilog_defaults -add -I include_dir
read_verilog file5.v
read_verilog file6.v
verilog_defaults -pop
```

Todo: more info on other `read_*` commands, also is this the first time we mention `verific`?

Note: The Verific frontend for Yosys, which provides the *verific* command, requires Yosys to be built with Verific. For full functionality, custom modifications to the Verific source code from YosysHQ are required, but limited useability can be achieved with some stock Verific builds. Check *Compiling with Verific library* for more.

Others:

- *read* - load HDL designs
- GHDL plugin for VHDL
- *read_rtlil* - read modules from RTLIL file (direct textual representation of Yosys internal state)
- *read_aiger* - read AIGER file
- *read_blif* - read BLIF file
- *read_json* - read JSON file
- *read_liberty* - read cells from liberty file

3.2.2 Selections

The selection framework

Todo: reduce overlap with *Scripting in Yosys* select section

The `select` command can be used to create a selection for subsequent commands. For example:

```
select foobar      # select the module foobar
delete            # delete selected objects
```

Normally the `select` command overwrites a previous selection. The commands `select -add` and `select -del` can be used to add or remove objects from the current selection.

The command `select -clear` can be used to reset the selection to the default, which is a complete selection of everything in the current module.

This selection framework can also be used directly in many other commands. Whenever a command has `[selection]` as last argument in its usage help, this means that it will use the engine behind the `select` command to evaluate additional arguments and use the resulting selection instead of the selection created by the last `select` command.

For example, the command `delete` will delete everything in the current selection; while `delete foobar` will only delete the module foobar. If no `select` command has been made, then the “current selection” will be the whole design.

Note: Many of the examples on this page make use of the `show` command to visually demonstrate the effect of selections. For a more detailed look at this command, refer to *A look at the show command*.

How to make a selection

Selection by object name

The easiest way to select objects is by object name. This is usually only done in synthesis scripts that are hand-tailored for a specific design.

```
select foobar      # select module foobar
select foo*        # select all modules whose names start with foo
select foo*/bar*   # select all objects matching bar* from modules matching foo*
select */clk       # select objects named clk from all modules
```

Module and design context

Commands can be executed in *module/* or *design/* context. Until now, all commands have been executed in design context. The `cd` command can be used to switch to module context.

In module context, all commands only effect the active module. Objects in the module are selected without the `<module_name>/` prefix. For example:

```
cd foo                # switch to module foo
delete bar            # delete object foo/bar

cd mycpu              # switch to module mycpu
dump reg_*            # print details on all objects whose names start with reg_

cd ..                 # switch back to design
```

Note: Most synthesis scripts never switch to module context. But it is a very powerful tool which we explore more in *Interactive design investigation*.

Selecting by object property or type

Special patterns can be used to select by object property or type. For example:

- select all wires whose names start with `reg_`: `select w:reg_*`
- select all objects with the attribute `foobar` set: `select a:foobar`
- select all objects with the attribute `foobar` set to 42: `select a:foobar=42`
- select all modules with the attribute `blabla` set: `select A:blabla`
- select all `$add` cells from the module `foo`: `select foo/t:$add`

A complete list of pattern expressions can be found in *select - modify and view the list of selected objects*.

Operations on selections

Combining selections

The `select` command is actually much more powerful than it might seem at first glance. When it is called with multiple arguments, each argument is evaluated and pushed separately on a stack. After all arguments have been processed it simply creates the union of all elements on the stack. So `select t:$add a:foo` will select all `$add` cells and all objects with the `foo` attribute set:

Listing 3.38: Test module for operations on selections

```
module foobaraddsub(a, b, c, d, fa, fs, ba, bs);
  input [7:0] a, b, c, d;
  output [7:0] fa, fs, ba, bs;
  assign fa = a + (* foo *) b;
  assign fs = a - (* foo *) b;
  assign ba = c + (* bar *) d;
  assign bs = c - (* bar *) d;
endmodule
```

Listing 3.39: Output for command `select t:$add a:foo -list`
on Listing 3.38

```
yosys> select t:$add a:foo -list
foobaraddsub/$add$foobaraddsub.v:6$3
foobaraddsub/$sub$foobaraddsub.v:5$2
foobaraddsub/$add$foobaraddsub.v:4$1
```

In many cases simply adding more and more stuff to the selection is an ineffective way of selecting the interesting part of the design. Special arguments can be used to combine the elements on the stack. For example the `%i` argument pops the last two elements from the stack, intersects them, and pushes the result back on the stack. So `select t:$add a:foo %i` will select all `$add` cells that have the `foo` attribute set:

Listing 3.40: Output for command `select t:$add a:foo %i`
`-list` on Listing 3.38

```
yosys> select t:$add a:foo %i -list
foobaraddsub/$add$foobaraddsub.v:4$1
```

Some of the special `%`-codes:

- `%u`: union of top two elements on stack – pop 2, push 1
- `%d`: difference of top two elements on stack – pop 2, push 1
- `%i`: intersection of top two elements on stack – pop 2, push 1
- `%n`: inverse of top element on stack – pop 1, push 1

See *select - modify and view the list of selected objects* for the full list.

Expanding selections

Listing 3.41 uses the Yosys non-standard `{... *}` syntax to set the attribute `sumstuff` on all cells generated by the first assign statement. (This works on arbitrary large blocks of Verilog code and can be used to mark portions of code for analysis.)

Listing 3.41: Another test module for operations on selections

```
module sumprod(a, b, c, sum, prod);

    input [7:0] a, b, c;
    output [7:0] sum, prod;

    {* sumstuff *}
    assign sum = a + b + c;
    {* *}

    assign prod = a * b * c;

endmodule
```

Selecting `a:sumstuff` in this module will yield the following circuit diagram:

As only the cells themselves are selected, but not the temporary wire `$1_Y`, the two adders are shown as two disjunct parts. This can be very useful for global signals like clock and reset signals: just unselect them



Fig. 3.11: Output of `show a:sumstuff` on [Listing 3.41](#)

using a command such as `select -del clk rst` and each cell using them will get its own net label.

In this case however we would like to see the cells connected properly. This can be achieved using the `%x` action, that broadens the selection, i.e. for each selected wire it selects all cells connected to the wire and vice versa. So `show a:sumstuff %x` yields the diagram shown in Fig. 3.12:

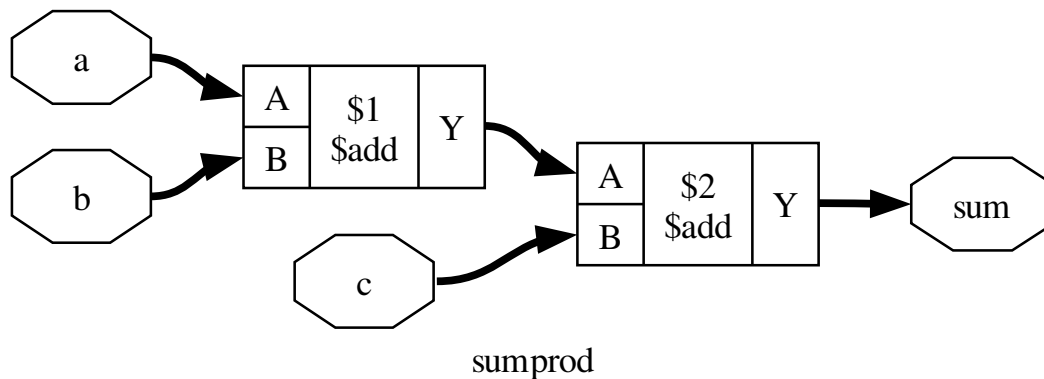


Fig. 3.12: Output of `show a:sumstuff %x` on Listing 3.41

Selecting logic cones

Fig. 3.12 shows what is called the **input cone** of `sum`, i.e. all cells and signals that are used to generate the signal `sum`. The `%ci` action can be used to select the input cones of all object in the top selection in the stack maintained by the `select` command.

As with the `%x` action, these commands broaden the selection by one “step”. But this time the operation only works against the direction of data flow. That means, wires only select cells via output ports and cells only select wires via input ports.

The following sequence of diagrams demonstrates this step-wise expansion:



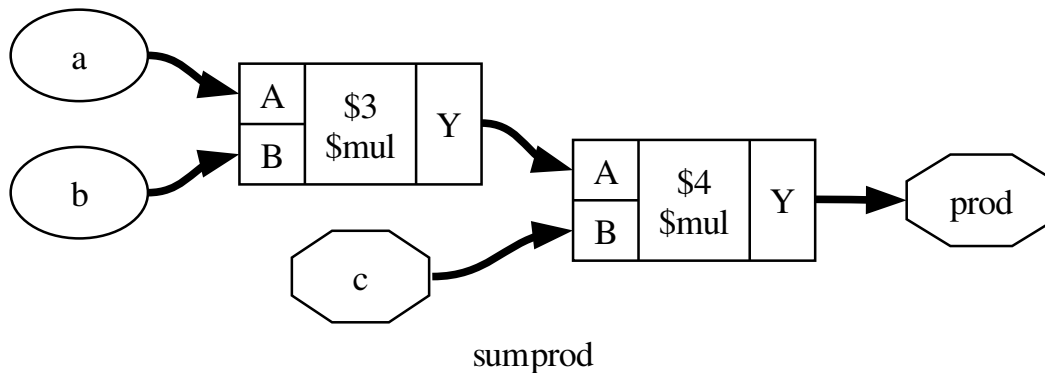
Fig. 3.13: Output of `show prod` on Listing 3.41



Fig. 3.14: Output of `show prod %ci` on Listing 3.41



Fig. 3.15: Output of `show prod %ci %ci` on Listing 3.41

Fig. 3.16: Output of `show prod %ci %ci %ci` on Listing 3.41

Notice the subtle difference between `show prod %ci` and `show prod %ci %ci`. Both images show the `$mul` cell driven by some inputs `$3_Y` and `c`. However it is not until the second image, having called `%ci` the second time, that `show` is able to distinguish between `$3_Y` being a wire and `c` being an input. We can see this better with the `dump` command instead:

Listing 3.42: Output of `dump prod %ci`

```

attribute \src "sumprod.v:4.21-4.25"
wire width 8 output 5 \prod

attribute \src "sumprod.v:10.17-10.26"
cell $mul $mul$sumprod.v:10$4
  parameter \A_SIGNED 0
  parameter \A_WIDTH 8
  parameter \B_SIGNED 0
  parameter \B_WIDTH 8
  parameter \Y_WIDTH 8
  connect \A $mul$sumprod.v:10$3_Y
  connect \B \c
  connect \Y \prod
end

```

Listing 3.43: Output of `dump prod %ci %ci`

```

attribute \src "sumprod.v:10.17-10.22"
wire width 8 $mul$sumprod.v:10$3_Y

attribute \src "sumprod.v:3.21-3.22"
wire width 8 input 3 \c

attribute \src "sumprod.v:4.21-4.25"
wire width 8 output 5 \prod

```

(continues on next page)

(continued from previous page)

```

attribute \src "sumprod.v:10.17-10.26"
cell $mul $mul$sumprod.v:10$4
  parameter \A_SIGNED 0
  parameter \A_WIDTH 8
  parameter \B_SIGNED 0
  parameter \B_WIDTH 8
  parameter \Y_WIDTH 8
  connect \A $mul$sumprod.v:10$3_Y
  connect \B \c
  connect \Y \prod
end

```

When selecting many levels of logic, repeating %ci over and over again can be a bit dull. So there is a shortcut for that: the number of iterations can be appended to the action. So for example the action %ci3 is identical to performing the %ci action three times.

The action %ci* performs the %ci action over and over again until it has no effect anymore.

Advanced logic cone selection

In most cases there are certain cell types and/or ports that should not be considered for the %ci action, or we only want to follow certain cell types and/or ports. This can be achieved using additional patterns that can be appended to the %ci action.

Lets consider Listing 3.44. It serves no purpose other than being a non-trivial circuit for demonstrating some of the advanced Yosys features. This code is available in docs/source/code_examples/selections of the Yosys source repository.

Listing 3.44: memdemo.v

```

module memdemo(clk, d, y);

input clk;
input [3:0] d;
output reg [3:0] y;

integer i;
reg [1:0] s1, s2;
reg [3:0] mem [0:3];

always @(posedge clk) begin
  for (i = 0; i < 4; i = i+1)
    mem[i] <= mem[(i+1) % 4] + mem[(i+2) % 4];
  { s2, s1 } = d ? { s1, s2 } ^ d : 4'b0;
  mem[s1] <= d;
  y <= mem[s2];
end

endmodule

```

The script memdemo.js is used to generate the images included here. Let's look at the first section:

Listing 3.45: Synthesizing *memdemo.v*

```
read_verilog memdemo.v
prep -top memdemo; memory; opt
```

This loads Listing 3.44 and synthesizes the included module. Note that this code can be copied and run directly in a Yosys command line session, provided *memdemo.v* is in the same directory. We can now change to the *memdemo* module with `cd memdemo`, and call *show* to see the diagram in Fig. 3.17.

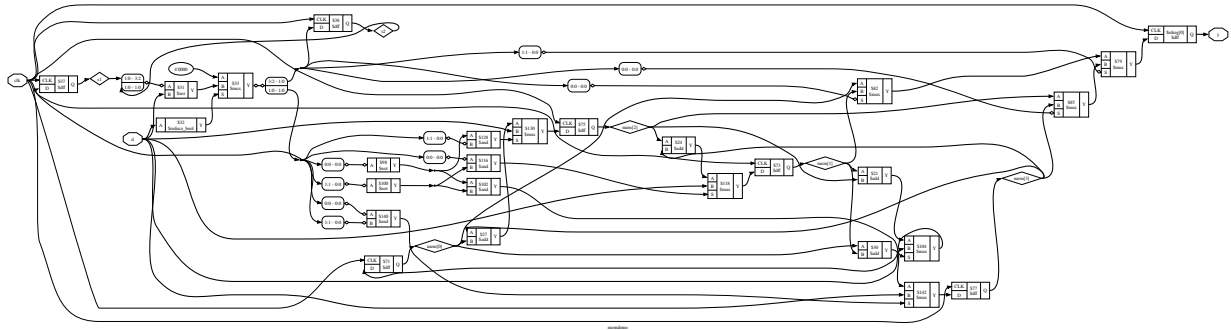


Fig. 3.17: Complete circuit diagram for the design shown in Listing 3.44

There's a lot going on there, but maybe we are only interested in the tree of multiplexers that select the output value. Let's start by just showing the output signal, *y*, and its immediate predecessors. Remember *Selecting logic cones* from above, we can use `show y %ci2`:

Fig. 3.18: Output of `show y %ci2`

From this we would learn that *y* is driven by a *\$dff* cell, that *y* is connected to the output port *Q*, that the *clk* signal goes into the *CLK* input port of the cell, and that the data comes from an auto-generated wire into the input *D* of the flip-flop cell (indicated by the *\$* at the start of the name). Let's go a bit further now and try `show y %ci5`:

That's starting to get a bit messy, so maybe we want to ignore the mux select inputs. To add a pattern we add a colon followed by the pattern to the `%ci` action. The pattern itself starts with *-* or *+*, indicating if it is an include or exclude pattern, followed by an optional comma separated list of cell types, followed by an

Fig. 3.19: Output of `show y %ci5`

optional comma separated list of port names in square brackets. In this case, we want to exclude the S port of the `$mux` cell type with `show y %ci5:-$mux[S]`:

We could use a command such as `show y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff` in which the first `%ci` jumps over the initial d-type flip-flop and the 2nd action selects the entire input cone without going over multiplexer select inputs and flip-flop cells:

Or we could use `show y %ci*:-[CLK,S]:+$dff:+$mux` instead, following the input cone all the way but only following `$dff` and `$mux` cells, and ignoring any ports named CLK or S:

Todo: pending discussion on whether rule ordering is a bug or a feature

Similar to `%ci` exists an action `%co` to select output cones that accepts the same syntax for pattern and repetition. The `%x` action mentioned previously also accepts this advanced syntax.

These actions for traversing the circuit graph, combined with the actions for boolean operations such as intersection (`%i`) and difference (`%d`) are powerful tools for extracting the relevant portions of the circuit under investigation.

Again, see *select - modify and view the list of selected objects* for full documentation of these expressions.

Incremental selection

Sometimes a selection can most easily be described by a series of add/delete operations. As mentioned previously, the commands `select -add` and `select -del` respectively add or remove objects from the current selection instead of overwriting it.

```
select -none          # start with an empty selection
select -add reg_*     # select a bunch of objects
select -del reg_42    # but not this one
select -add state %ci # and add more stuff
```

Within a select expression the token `%` can be used to push the previous selection on the stack.

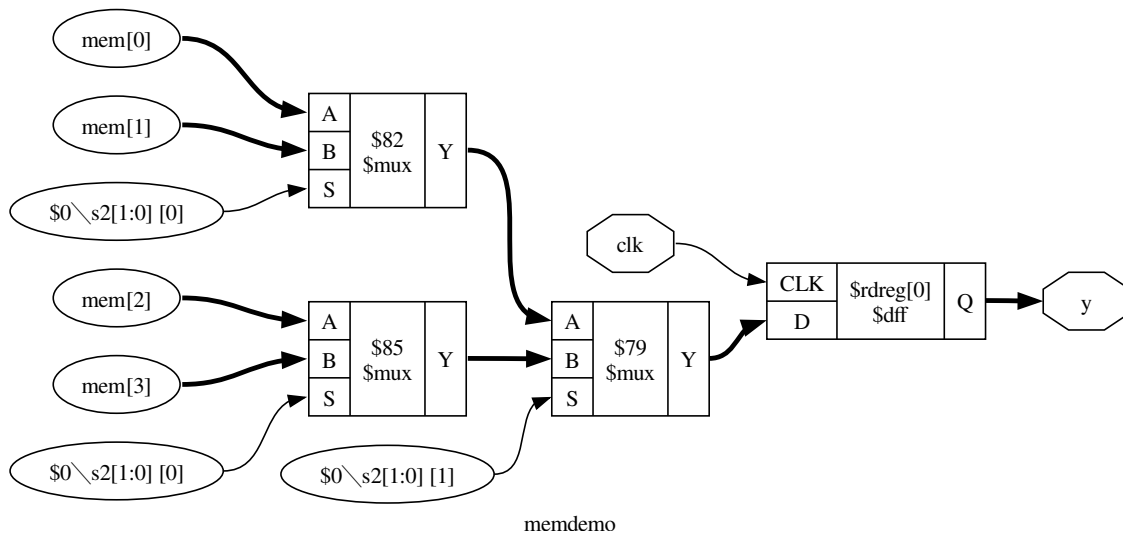


Fig. 3.20: Output of show y %ci5:-\$mux[S]

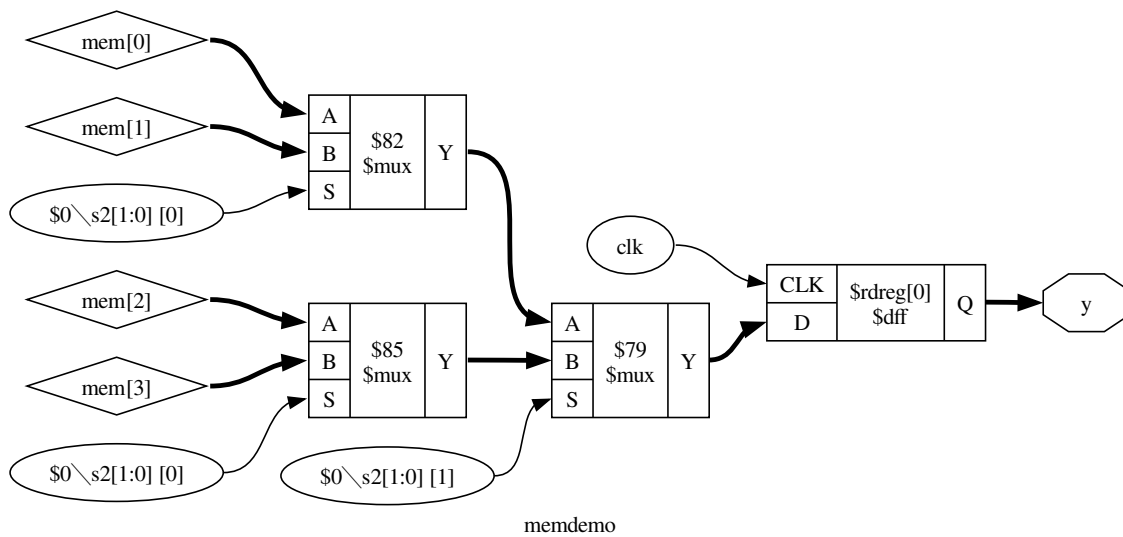
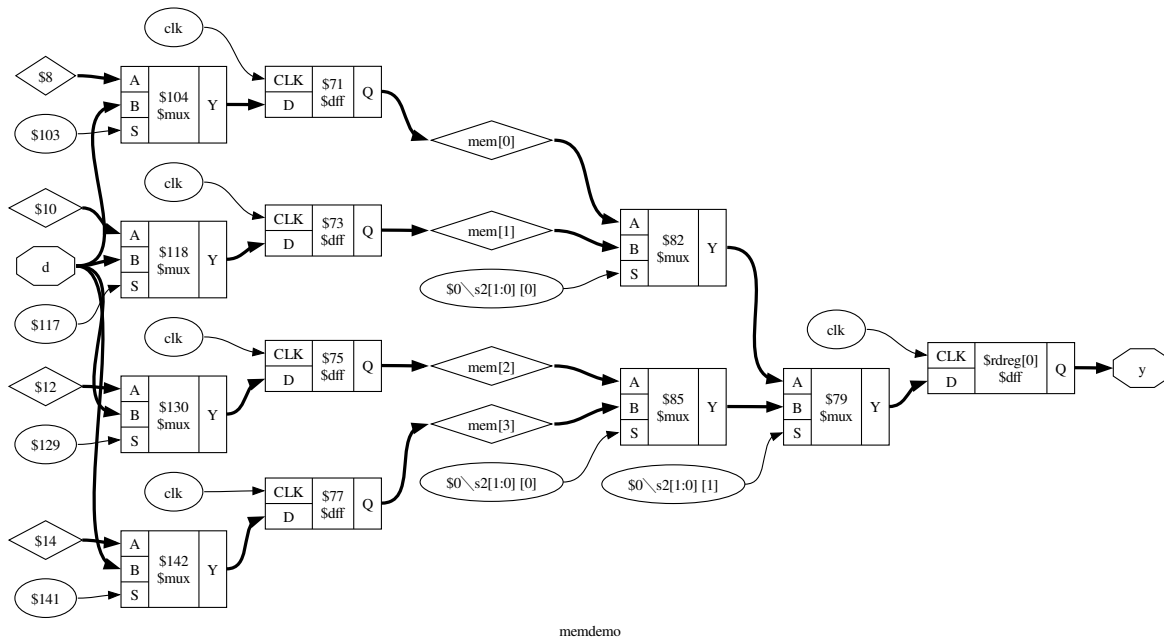


Fig. 3.21: Output of show y %ci2:+\$dff[Q,D] %ci*:-\$mux[S]:-\$dff

Fig. 3.22: Output of `show y %ci*:-[CLK,S]:+$dff,$mux`

```
select t:$add t:$sub      # select all $add and $sub cells
select % %ci % %d        # select only the input wires to those cells
```

Storing and recalling selections

Todo: reflow for not presentation

The current selection can be stored in memory with the command `select -set <name>`. It can later be recalled using `select @<name>`. In fact, the `@<name>` expression pushes the stored selection on the stack maintained by the `select` command. So for example `select @foo @bar %i` will select the intersection between the stored selections `foo` and `bar`.

In larger investigation efforts it is highly recommended to maintain a script that sets up relevant selections, so they can easily be recalled, for example when Yosys needs to be re-run after a design or source code change.

The `history` command can be used to list all recent interactive commands. This feature can be useful for creating such a script from the commands used in an interactive session.

Remember that select expressions can also be used directly as arguments to most commands. Some commands also accept a single select argument to some options. In those cases selection variables must be used to capture more complex selections.

Example code from [docs/source/code_examples/selections](#):

Listing 3.46: select.v

```

module test(clk, s, a, y);
    input clk, s;
    input [15:0] a;
    output [15:0] y;
    reg [15:0] b, c;

    always @(posedge clk) begin
        b <= a;
        c <= b;
    end

    wire [15:0] state_a = (a ^ b) + c;
    wire [15:0] state_b = (a ^ b) - c;
    assign y = !s ? state_a : state_b;
endmodule

```

Listing 3.47: select.yo

```

read_verilog select.v
prep -top test

cd test
select -set cone_a state_a %ci*:-$dff
select -set cone_b state_b %ci*:-$dff
select -set cone_ab @cone_a @cone_b %i
show -prefix select -format dot -notitle \
    -color red @cone_ab -color magenta @cone_a \
    -color blue @cone_b

```

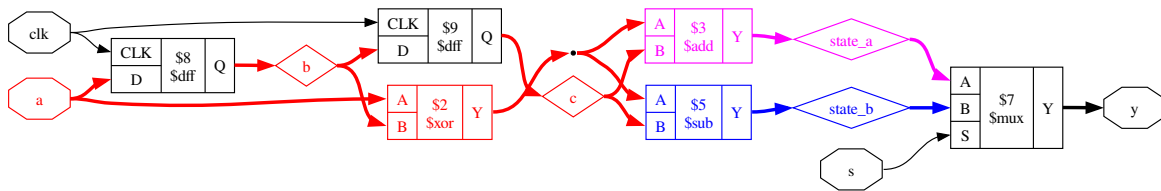


Fig. 3.23: Circuit diagram produced by Listing 3.47

3.2.3 Interactive design investigation

Todo: interactive design opening text

A look at the show command

Todo: merge into *Scripting in Yosys* show section

This section explores the *show* command and explains the symbols used in the circuit diagrams generated by it. The code used is included in the Yosys code base under `docs/source/code_examples/show`.

A simple circuit

example.v below provides the Verilog code for a simple circuit which we will use to demonstrate the usage of *show* in a simple setting.

Listing 3.48: `example.v`

```
module example(input clk, a, b, c,
               output reg [1:0] y);
    always @(posedge clk)
        if (c)
            y <= c ? a + b : 2'd0;
endmodule
```

The Yosys synthesis script we will be running is included as Listing 3.49. Note that *show* is called with the `-pause` option, that halts execution of the Yosys script until the user presses the Enter key. Using *show -pause* also allows the user to enter an interactive shell to further investigate the circuit before continuing synthesis.

Listing 3.49: `example_show.ys`

```
read_verilog example.v
show -pause # first
proc
show -pause # second
opt
show -pause # third
```

This script, when executed, will show the design after each of the three synthesis commands. We will now look at each of these diagrams and explain what is shown.

Note: The images used in this document are generated from the `example.ys` file, rather than `example_show.ys`. `example.ys` outputs the schematics as `.dot` files rather than displaying them directly. You can view these images yourself by running `yosys example.ys` and then `xdot example_first.dot` etc.

The first output shows the design directly after being read by the Verilog front-end. Input and output ports are displayed as octagonal shapes. Cells are displayed as rectangles with inputs on the left and outputs on

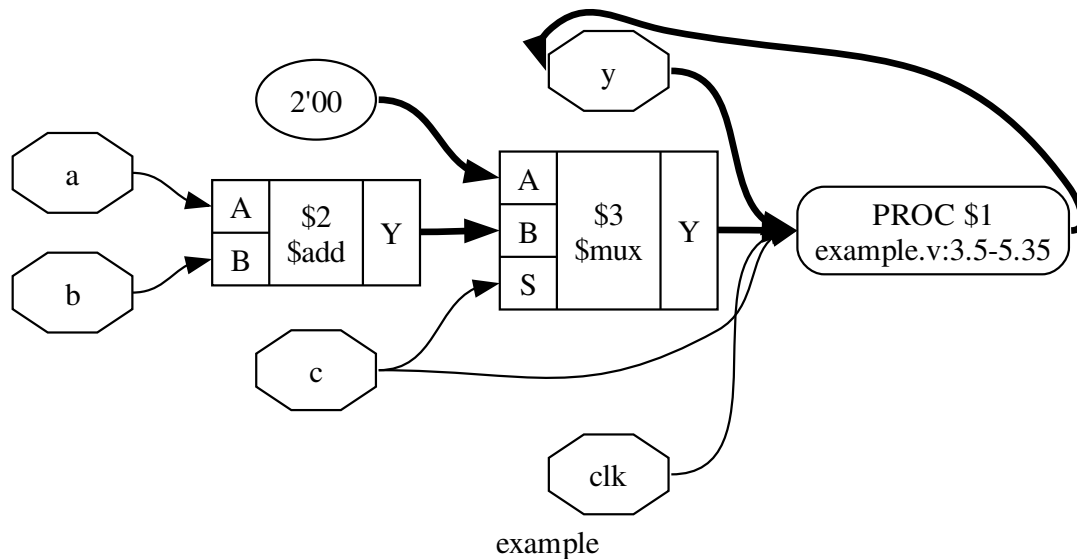


Fig. 3.24: Output of the first `show` command in Listing 3.49

the right side. The cell labels are two lines long: The first line contains a unique identifier for the cell and the second line contains the cell type. Internal cell types are prefixed with a dollar sign. For more details on the internal cell library, see *Internal cell library*.

Constants are shown as ellipses with the constant value as label. The syntax `<bit_width>'<bits>` is used for constants that are not 32-bit wide and/or contain bits that are not 0 or 1 (i.e. x or z). Ordinary 32-bit constants are written using decimal numbers.

Single-bit signals are shown as thin arrows pointing from the driver to the load. Signals that are multiple bits wide are shown as thick arrows.

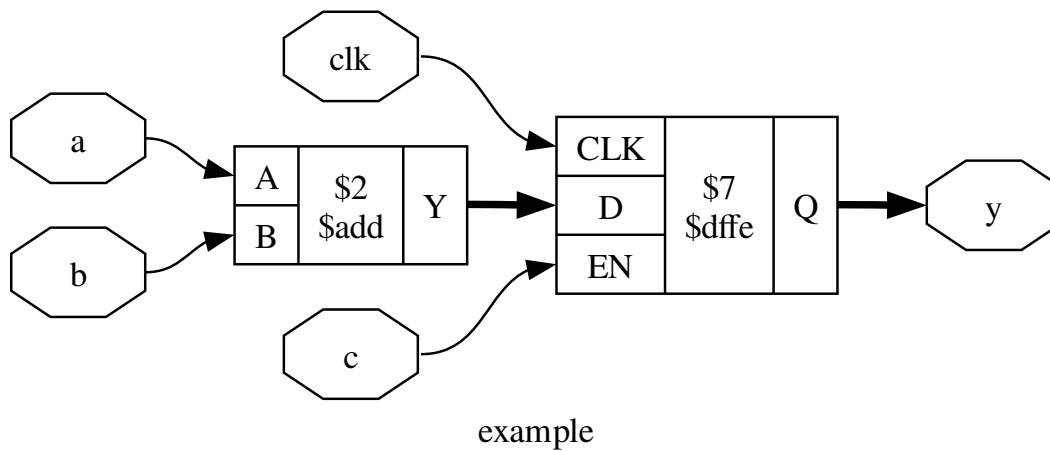
Finally *processes* are shown in boxes with round corners. Processes are Yosys' internal representation of the decision-trees and synchronization events modelled in a Verilog `always`-block. The label reads **PROC** followed by a unique identifier in the first line and contains the source code location of the original `always`-block in the second line. Note how the multiplexer from the `?:`-expression is represented as a `$mux` cell but the multiplexer from the `if`-statement is yet still hidden within the process.

The `proc` command transforms the process from the first diagram into a multiplexer and a d-type flip-flop, which brings us to the second diagram:

The Rhombus shape to the right is a dangling wire. (Wire nodes are only shown if they are dangling or have “public” names, for example names assigned from the Verilog input.) Also note that the design now contains two instances of a BUF-node. These are artefacts left behind by the `proc` command. It is quite usual to see such artefacts after calling commands that perform changes in the design, as most commands only care about doing the transformation in the least complicated way, not about cleaning up after them. The next call to `clean` (or `opt`, which includes `clean` as one of its operations) will clean up these artefacts. This operation is so common in Yosys scripts that it can simply be abbreviated with the `;;` token, which doubles as separator for commands. Unless one wants to specifically analyze this artefacts left behind some operations, it is therefore recommended to always call `clean` before calling `show`.

Fig. 3.25: Output of the second `show` command in Listing 3.49

In this script we directly call `opt` as the next step, which finally leads us to the third diagram:

Fig. 3.26: Output of the third `show` command in `example_show.y`

Here we see that the `opt` command not only has removed the artifacts left behind by `proc`, but also determined correctly that it can remove the first `$mux` cell without changing the behavior of the circuit.

Break-out boxes for signal vectors

The code listing below shows a simple circuit which uses a lot of spliced signal accesses.

Listing 3.50: splice.v

```
module splice_demo(a, b, c, d, e, f, x, y);

input [1:0] a, b, c, d, e, f;
output [1:0] x;
assign x = {a[0], a[1]};

output [11:0] y;
assign {y[11:4], y[1:0], y[3:2]} =
      {a, b, ~{c, d}, ~{e, f}};

endmodule
```

Notice how the output for this circuit from the `show` command (Fig. 3.27) appears quite complex. This is an unfortunate side effect of the way Yosys handles signal vectors (aka. multi-bit wires or buses) as native objects. While this provides great advantages when analyzing circuits that operate on wide integers, it also introduces some additional complexity when the individual bits of of a signal vector are accessed.

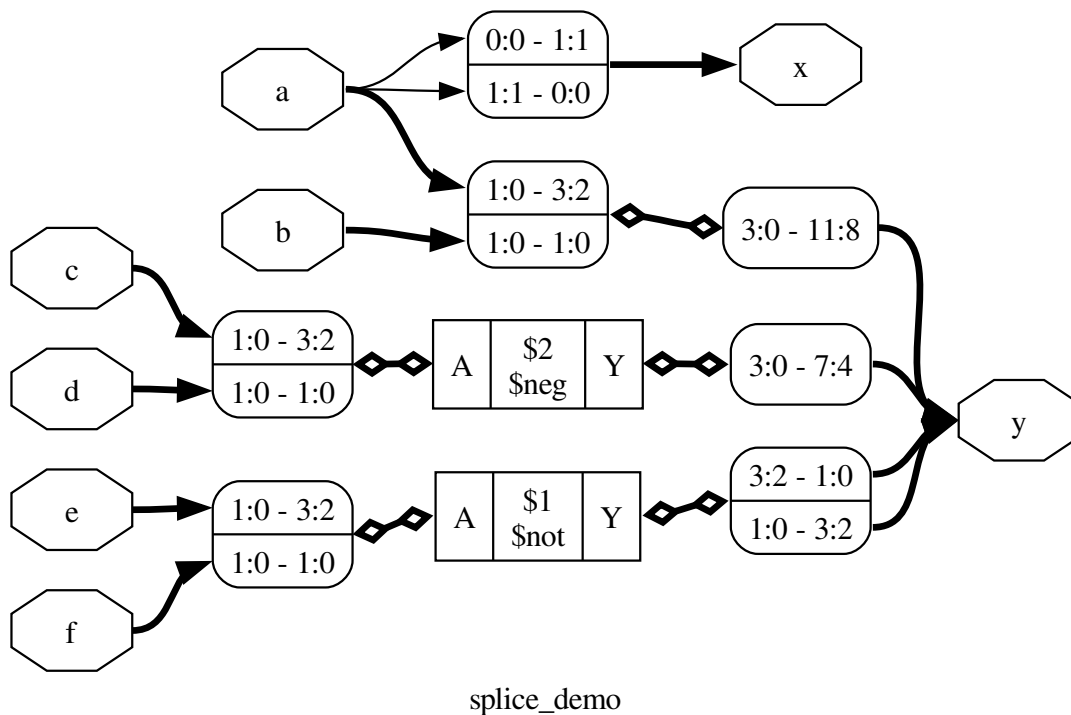


Fig. 3.27: Output of `yosys -p 'prep -top splice_demo; show' splice.v`

The key elements in understanding this circuit diagram are of course the boxes with round corners and rows labeled `<MSB_LEFT>:<LSB_LEFT>` - `<MSB_RIGHT>:<LSB_RIGHT>`. Each of these boxes have one signal per row on one side and a common signal for all rows on the other side. The `<MSB>:<LSB>` tuples specify which bits of the signals are broken out and connected. So the top row of the box connecting the signals `a` and `x` indicates that the bit 0 (i.e. the range 0:0) from signal `a` is connected to bit 1 (i.e. the range 1:1) of signal `x`.

Lines connecting such boxes together and lines connecting such boxes to cell ports have a slightly different look to emphasise that they are not actual signal wires but a necessity of the graphical representation. This distinction seems like a technicality, until one wants to debug a problem related to the way Yosys internally represents signal vectors, for example when writing custom Yosys commands.

Gate level netlists

Fig. 3.28 shows two common pitfalls when working with designs mapped to a cell library:

Listing 3.51: Generating Fig. 3.28

```
read_verilog cmos.v
prep -top cmos_demo
techmap
abc -liberty ../intro/mycells.lib;;
show -format dot -prefix cmos_00
```

First, Yosys did not have access to the cell library when this diagram was generated, resulting in all cell ports defaulting to being inputs. This is why all ports are drawn on the left side the cells are awkwardly arranged in a large column. Secondly the two-bit vector `y` requires breakout-boxes for its individual bits, resulting in an unnecessary complex diagram.

Listing 3.52: Generating Fig. 3.29

```
read_verilog cmos.v
prep -top cmos_demo
techmap
splitnets -ports
abc -liberty ../intro/mycells.lib;;
show -lib ../intro/mycells.v -format dot -prefix cmos_01
```

For Fig. 3.29, Yosys has been given a description of the cell library as Verilog file containing blackbox modules. There are two ways to load cell descriptions into Yosys: First the Verilog file for the cell library can be passed directly to the `show` command using the `-lib <filename>` option. Secondly it is possible to load cell libraries into the design with the `read_verilog -lib <filename>` command. The second method has the great advantage that the library only needs to be loaded once and can then be used in all subsequent calls to the `show` command.

In addition to that, Fig. 3.29 was generated after `splitnet -ports` was run on the design. This command splits all signal vectors into individual signal bits, which is often desirable when looking at gate-level circuits. The `-ports` option is required to also split module ports. Per default the command only operates on interior signals.

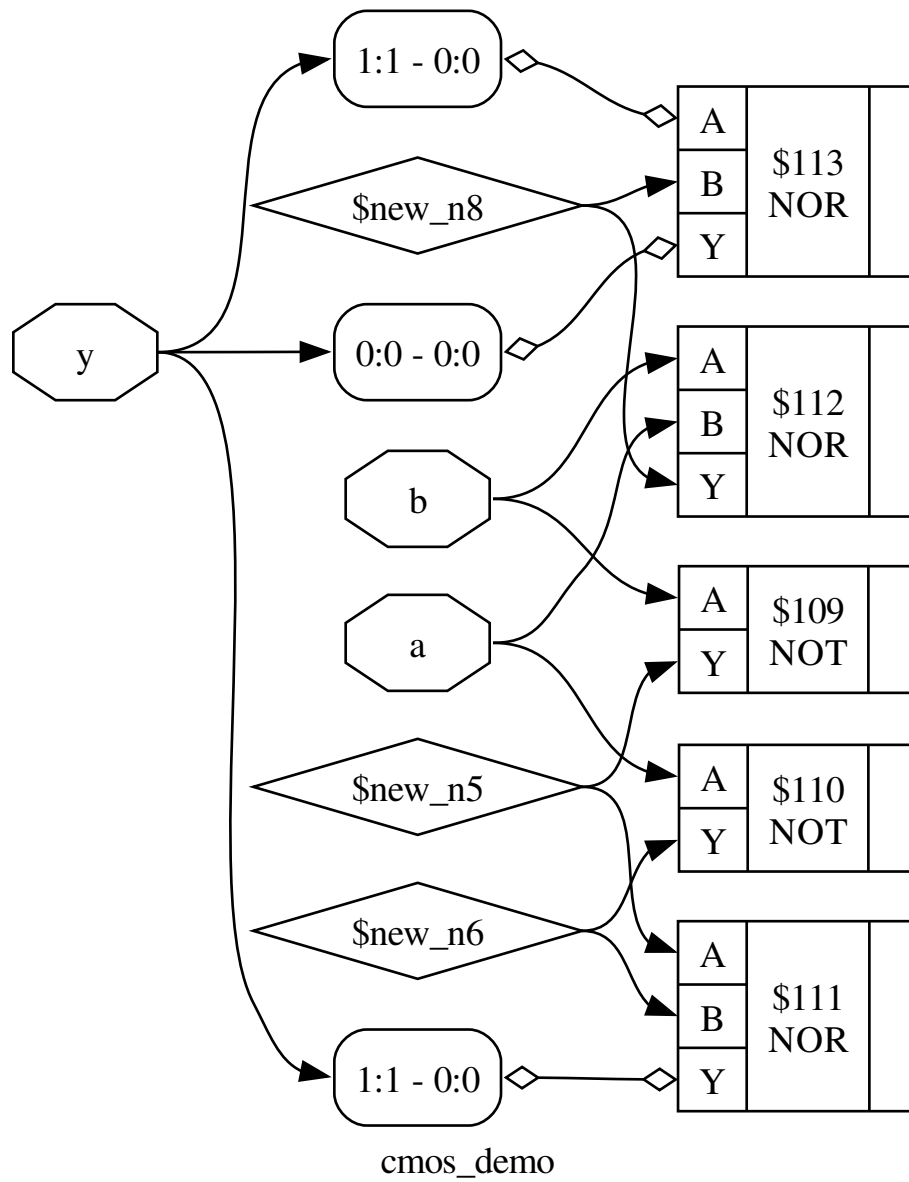


Fig. 3.28: A half-adder built from simple CMOS gates, demonstrating common pitfalls when using `show`

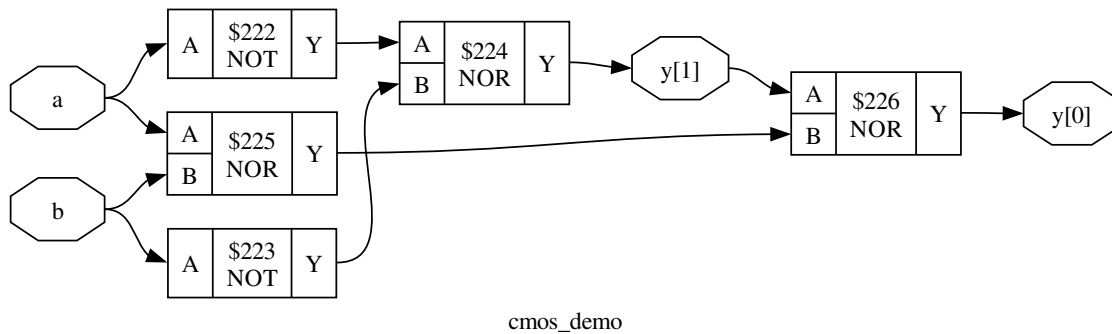


Fig. 3.29: Effects of `splitnets` command and of providing a cell library on design in Fig. 3.28

Miscellaneous notes

Per default the `show` command outputs a temporary dot file and launches `xdot` to display it. The options `-format`, `-viewer` and `-prefix` can be used to change format, viewer and filename prefix. Note that the `pdf` and `ps` format are the only formats that support plotting multiple modules in one run. The `dot` format can be used to output multiple modules, however `xdot` will raise an error when trying to read them.

In densely connected circuits it is sometimes hard to keep track of the individual signal wires. For these cases it can be useful to call `show` with the `-colors <integer>` argument, which randomly assigns colors to the nets. The integer (> 0) is used as seed value for the random color assignments. Sometimes it is necessary it try some values to find an assignment of colors that looks good.

The command `help show` prints a complete listing of all options supported by the `show` command.

Navigating the design

Plotting circuit diagrams for entire modules in the design brings us only helps in simple cases. For complex modules the generated circuit diagrams are just stupidly big and are no help at all. In such cases one first has to select the relevant portions of the circuit.

In addition to *what* to display one also needs to carefully decide *when* to display it, with respect to the synthesis flow. In general it is a good idea to troubleshoot a circuit in the earliest state in which a problem can be reproduced. So if, for example, the internal state before calling the `techmap` command already fails to verify, it is better to troubleshoot the coarse-grain version of the circuit before `techmap` than the gate-level circuit after `techmap`.

Note: It is generally recommended to verify the internal state of a design by writing it to a Verilog file using `write_verilog -noexpr` and using the simulation models from `simlib.v` and `simcells.v` from the Yosys data directory (as printed by `yosys-config --datdir`).

Interactive navigation

Once the right state within the synthesis flow for debugging the circuit has been identified, it is recommended to simply add the `shell` command to the matching place in the synthesis script. This command will stop the synthesis at the specified moment and go to shell mode, where the user can interactively enter commands.

For most cases, the shell will start with the whole design selected (i.e. when the synthesis script does not already narrow the selection). The command `ls` can now be used to create a list of all modules. The command `cd` can be used to switch to one of the modules (type `cd ..` to switch back). Now the `ls` command lists the objects within that module. This is demonstrated below using `example.v` from *A simple circuit*:

Listing 3.53: Output of `ls` and `cd` after running `yosys example.v`

```
yosys> ls

1 modules:
  example

yosys> cd example

yosys [example]> ls

8 wires:
  $0\y[1:0]
  $add$example.v:5$2_Y
  $ternary$example.v:5$3_Y
  a
  b
  c
  clk
  y

2 cells:
  $add$example.v:5$2
  $ternary$example.v:5$3

1 processes:
  $proc$example.v:3$1
```

When a module is selected using the `cd` command, all commands (with a few exceptions, such as the `read_` and `write_` commands) operate only on the selected module. This can also be useful for synthesis scripts where different synthesis strategies should be applied to different modules in the design.

We can see that the cell names from Fig. 3.26 are just abbreviations of the actual cell names, namely the part after the last dollar-sign. Most auto-generated names (the ones starting with a dollar sign) are rather long and contains some additional information on the origin of the named object. But in most cases those names can simply be abbreviated using the last part.

Usually all interactive work is done with one module selected using the `cd` command. But it is also possible to work from the design-context (`cd ..`). In this case all object names must be prefixed with `<module_name>/.`. For example `a*/b*` would refer to all objects whose names start with `b` from all modules whose names start with `a`.

The `dump` command can be used to print all information about an object. For example, calling `dump $2` after the `cd example` above:

Listing 3.54: Output of dump \$2 after Listing 3.53

```

attribute \src "example.v:5.22-5.27"
cell $add $add$example.v:5$2
  parameter \Y_WIDTH 2
  parameter \B_WIDTH 1
  parameter \A_WIDTH 1
  parameter \B_SIGNED 0
  parameter \A_SIGNED 0
  connect \Y $add$example.v:5$2_Y
  connect \B \b
  connect \A \a
end

```

This can for example be useful to determine the names of nets connected to cells, as the net-names are usually suppressed in the circuit diagram if they are auto-generated. Note that the output is in the RTLIL representation, described in *The RTL Intermediate Language (RTLIL)*.

Interactive Design Investigation

Yosys can also be used to investigate designs (or netlists created from other tools).

- The selection mechanism, especially patterns such as %ci and %co, can be used to figure out how parts of the design are connected.
- Commands such as *submod*, *expose*, and *splice* can be used to transform the design into an equivalent design that is easier to analyse.
- Commands such as *eval* and *sat* can be used to investigate the behavior of the circuit.
- *show* - generate schematics using graphviz.
- *dump* - print parts of the design in RTLIL format.
- *add* - add objects to the design and *delete* - delete objects in the design can be used to modify and reorganize a design dynamically.

The code used is included in the Yosys code base under docs/source/code_examples/scrambler.

Changing design hierarchy

Commands such as *flatten* and *submod* can be used to change the design hierarchy, i.e. flatten the hierarchy or moving parts of a module to a submodule. This has applications in synthesis scripts as well as in reverse engineering and analysis. An example using *submod* is shown below for reorganizing a module in Yosys and checking the resulting circuit.

Listing 3.55: scrambler.v

```

module scrambler(
    input clk, rst, in_bit,
    output reg out_bit
);
    reg [31:0] xs;
    always @(posedge clk) begin
        if (rst)

```

(continues on next page)



3.2. More scripting

(continued from previous page)

Eval result: \out = 67634689.

> sat -set out 632435482

Signal Name	Dec	Hex	Bin
\in	745495504	2c6f5bd0	00101100011011110101101111010000
\out	632435482	25b2331a	00100101101100100011001100011010

Behavioral changes

Commands such as *techmap* can be used to make behavioral changes to the design, for example changing asynchronous resets to synchronous resets. This has applications in design space exploration (evaluation of various architectures for one circuit).

The following techmap map file replaces all positive-edge async reset flip-flops with positive-edge sync reset flip-flops. The code is taken from the example Yosys script for ASIC synthesis of the Amber ARMv2 CPU.

Todo: replace inline code

```
(* techmap_celltype = "$adff" *)
module adff2dff (CLK, ARST, D, Q);

    parameter WIDTH = 1;
    parameter CLK_POLARITY = 1;
    parameter ARST_POLARITY = 1;
    parameter ARST_VALUE = 0;

    input CLK, ARST;
    input [WIDTH-1:0] D;
    output reg [WIDTH-1:0] Q;

    wire [1023:0] _TECHMAP_DO_ = "proc";

    wire _TECHMAP_FAIL_ = !CLK_POLARITY || !ARST_POLARITY;

    always @(posedge CLK)
        if (ARST)
            Q <= ARST_VALUE;
        else
            Q <= D;

endmodule
```

For more on the *techmap* command, see the page on *Techmap by example*.

Advanced investigation techniques

When working with very large modules, it is often not enough to just select the interesting part of the module. Instead it can be useful to extract the interesting part of the circuit into a separate module. This can for example be useful if one wants to run a series of synthesis commands on the critical part of the module and wants to carefully read all the debug output created by the commands in order to spot a problem. This kind of troubleshooting is much easier if the circuit under investigation is encapsulated in a separate module.

Recall the `memdemo` design from *Advanced logic cone selection*:



Fig. 3.30: `memdemo`

Because this produces a rather large circuit, it can be useful to split it into smaller parts for viewing and working with. Listing 3.57 does exactly that, utilising the `submod` command to split the circuit into three sections: `outstage`, `selstage`, and `scramble`.

Listing 3.57: Using `submod` to break up the circuit from `memdemo.v`

```
select -set outstage y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff
select -set selstage y %ci2:+$dff[Q,D] %ci*:-$dff @outstage %d
select -set scramble mem* %ci2 %ci*:-$dff mem* %d @selstage %d
submod -name scramble @scramble
submod -name outstage @outstage
submod -name selstage @selstage
```

The `-name` option is used to specify the name of the new module and also the name of the new cell in the current module. The resulting circuits are shown below.

Evaluation of combinatorial circuits

The `eval` command can be used to evaluate combinatorial circuits. As an example, we will use the `selstage` subnet of `memdemo` which we found above and is shown in Fig. 3.32.

Todo: replace inline code

```
yosys [selstage]> eval -set s2,s1 4'b1001 -set d 4'hc -show n2 -show n1
```

1. Executing EVAL pass (evaluate the circuit given an input).

Full command line: `eval -set s2,s1 4'b1001 -set d 4'hc -show n2 -show n1`

(continues on next page)

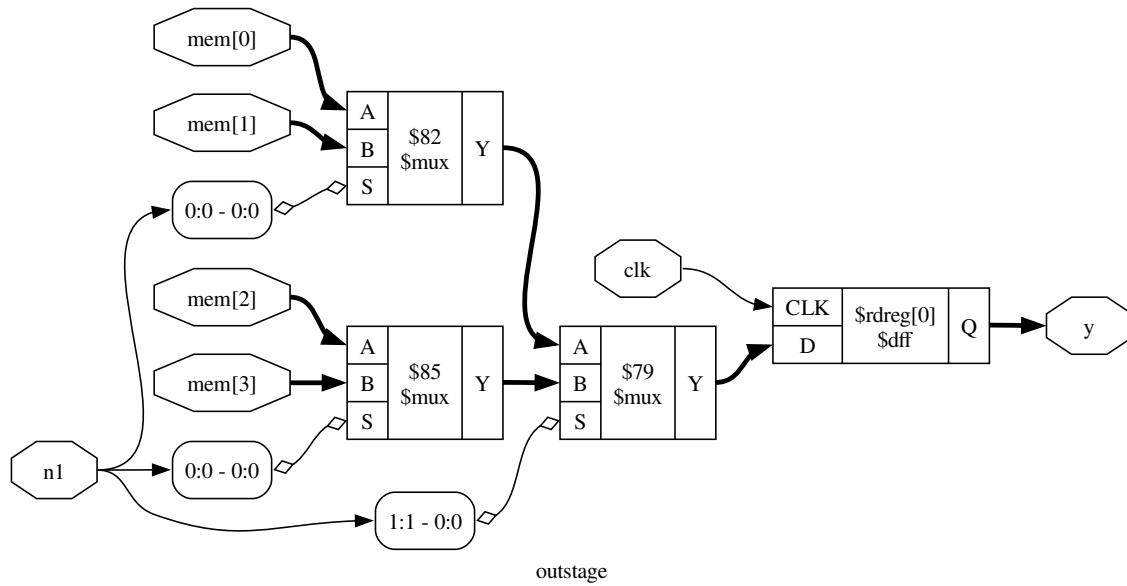


Fig. 3.31: outstage

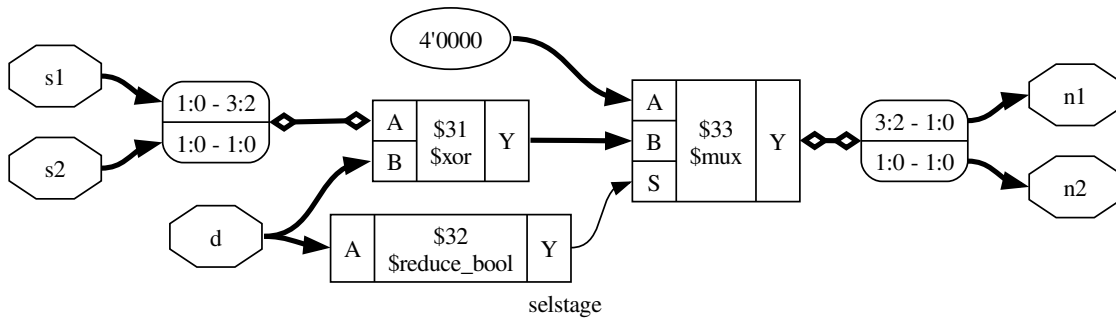


Fig. 3.32: selstage

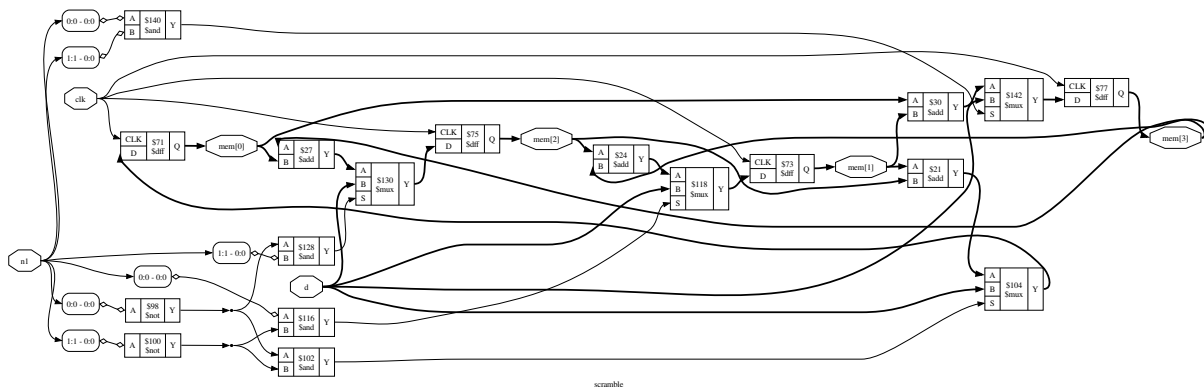


Fig. 3.33: scramble

(continued from previous page)

```
Eval result: \n2 = 2'10.
Eval result: \n1 = 2'10.
```

So the `-set` option is used to set input values and the `-show` option is used to specify the nets to evaluate. If no `-show` option is specified, all selected output ports are used per default.

If a necessary input value is not given, an error is produced. The option `-set-undef` can be used to instead set all unspecified input nets to undef (x).

The `-table` option can be used to create a truth table. For example:

```
yosys [selstage]> eval -set-undef -set d[3:1] 0 -table s1,d[0]
```

10. Executing EVAL pass (evaluate the circuit given an input).

Full command line: `eval -set-undef -set d[3:1] 0 -table s1,d[0]`

\s1	\d [0]	\n1	\n2
2'00	1'0	2'00	2'00
2'00	1'1	2'xx	2'00
2'01	1'0	2'00	2'00
2'01	1'1	2'xx	2'01
2'10	1'0	2'00	2'00
2'10	1'1	2'xx	2'10
2'11	1'0	2'00	2'00
2'11	1'1	2'xx	2'11

Assumed undef (x) value for the following signals: \s2

Note that the `eval` command (as well as the `sat` command discussed in the next sections) does only operate on flattened modules. It can not analyze signals that are passed through design hierarchy levels. So the `flatten` command must be used on modules that instantiate other modules before these commands can be applied.

Solving combinatorial SAT problems

Often the opposite of the `eval` command is needed, i.e. the circuits output is given and we want to find the matching input signals. For small circuits with only a few input bits this can be accomplished by trying all possible input combinations, as it is done by the `eval -table` command. For larger circuits however, Yosys provides the `sat` command that uses a SAT solver, `MiniSAT`, to solve this kind of problems.

Note: While it is possible to perform model checking directly in Yosys, it is highly recommended to use SBY or EQY for formal hardware verification.

The `sat` command works very similar to the `eval` command. The main difference is that it is now also possible to set output values and find the corresponding input values. For Example:

Todo: replace inline code

```
yosys [selstage]> sat -show s1,s2,d -set s1 s2 -set n2,n1 4'b1001
```

11. Executing SAT pass (solving SAT problems in the circuit).
Full command line: `sat -show s1,s2,d -set s1 s2 -set n2,n1 4'b1001`

Setting up SAT problem:
Import set-constraint: `\s1 = \s2`
Import set-constraint: `{ \n2 \n1 } = 4'b1001`
Final constraint equation: `{ \n2 \n1 \s1 } = { 4'b1001 \s2 }`
Imported 3 cells to SAT database.
Import show expression: `{ \s1 \s2 \d }`

Solving problem with 81 variables and 207 clauses..
SAT solving finished - model found:

Signal Name	Dec	Hex	Bin
<code>\d</code>	9	9	1001
<code>\s1</code>	0	0	00
<code>\s2</code>	0	0	00

Note that the `sat` command supports signal names in both arguments to the `-set` option. In the above example we used `-set s1 s2` to constraint `s1` and `s2` to be equal. When more complex constraints are needed, a wrapper circuit must be constructed that checks the constraints and signals if the constraint was met using an extra output port, which then can be forced to a value using the `-set` option. (Such a circuit that contains the circuit under test plus additional constraint checking circuitry is called a `miter` circuit.)

Listing 3.58: `primetest.v`, a simple miter circuit for testing if a number is prime. But it has a problem.

```
module primetest(p, a, b, ok);
input [15:0] p, a, b;
output ok = p != a*b || a == 1 || b == 1;
endmodule
```

Listing 3.58 shows a miter circuit that is supposed to be used as a prime number test. If `ok` is 1 for all input values `a` and `b` for a given `p`, then `p` is prime, or at least that is the idea.

Todo: replace inline code

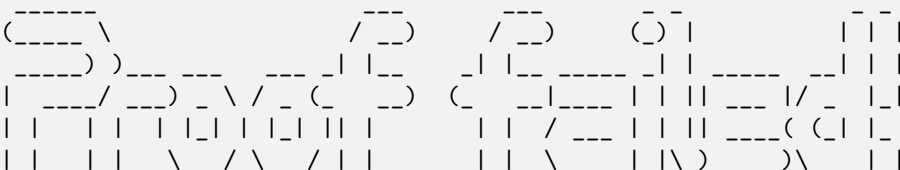
Listing 3.59: Experiments with the miter circuit from `primetest.v`.

```
yosys [primetest]> sat -prove ok 1 -set p 31
```

1. Executing SAT pass (solving SAT problems in the circuit).
Full command line: `sat -prove ok 1 -set p 31`

Setting up SAT problem:
Import set-constraint: `\p = 16'0000000000011111`
Final constraint equation: `\p = 16'0000000000011111`
Imported 6 cells to SAT database.
Import proof-constraint: `\ok = 1'1`
Final proof equation: `\ok = 1'1`

Solving problem with 2790 variables and 8241 clauses..
SAT proof finished - model found: FAIL!



Signal Name	Dec	Hex	Bin
\a	15029	3ab5	0011101010110101
\b	4099	1003	0001000000000011
\ok	0	0	0
\p	31	1f	0000000000011111

The Yosys shell session shown in Listing 3.59 demonstrates that SAT solvers can even find the unexpected solutions to a problem: Using integer overflow there actually is a way of “factorizing” 31. The clean solution would of course be to perform the test in 32 bits, for example by replacing `p != a*b` in the miter with `p != {16'd0,a}b`, or by using a temporary variable for the 32 bit product `a*b`. But as 31 fits well into 8 bits (and as the purpose of this document is to show off Yosys features) we can also simply force the upper 8 bits of `a` and `b` to zero for the `sat` call, as is done below.

Todo: replace inline code

Listing 3.60: Miter circuit from `primetest.v`, with the upper 8 bits of `a` and `b` constrained to prevent overflow.

```
yosys [primetest]> sat -prove ok 1 -set p 31 -set a[15:8],b[15:8] 0
```

1. Executing SAT pass (solving SAT problems in the circuit).

(continues on next page)

(continued from previous page)

```
Full command line: sat -prove ok 1 -set p 31 -set a[15:8],b[15:8] 0

Setting up SAT problem:
Import set-constraint: \p = 16'0000000000001111
Import set-constraint: { \a [15:8] \b [15:8] } = 16'0000000000000000
Final constraint equation: { \a [15:8] \b [15:8] \p } = { 16'0000000000000000 16
↪ '0000000000001111 }
Imported 6 cells to SAT database.
Import proof-constraint: \ok = 1'1
Final proof equation: \ok = 1'1

Solving problem with 2790 variables and 8257 clauses..
SAT proof finished - no model found: SUCCESS!
```

```

      /$$$$$      /$$$$$$      /$$$$$
    /$$__ $$    | $$_____/    | $$__ $$
   | $$ \ $$    | $$         | $$ \ $$
   | $$ | $$    | $$$$        | $$ | $$
   | $$ | $$    | $$__ /      | $$ | $$
   | $$/$$ $$   | $$         | $$ | $$
   | $$$$$$ /$$/ $$$$$$$ /$$/ $$$$$$//$$
   \___ $$$$|__/_/|_____/|__/_/|_____/|__/_/
       \__/_/

```

The `-prove` option used in [Listing 3.60](#) works similar to `-set`, but tries to find a case in which the two arguments are not equal. If such a case is not found, the property is proven to hold for all inputs that satisfy the other constraints.

It might be worth noting, that SAT solvers are not particularly efficient at factorizing large numbers. But if a small factorization problem occurs as part of a larger circuit problem, the Yosys SAT solver is perfectly capable of solving it.

Solving sequential SAT problems

The SAT solver functionality in Yosys can not only be used to solve combinatorial problems, but can also solve sequential problems. Let's consider the `memdemo` design from [Advanced logic cone selection](#) again, and suppose we want to know which sequence of input values for `d` will cause the output `y` to produce the sequence 1, 2, 3 from any initial state. Let's use the following command:

Todo: replace inline code?

```
sat -seq 6 -show y -show d -set-init-undef \
    -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3
```

The `-seq 6` option instructs the `sat` command to solve a sequential problem in 6 time steps. (Experiments with lower number of steps have show that at least 3 cycles are necessary to bring the circuit in a state from which the sequence 1, 2, 3 can be produced.)

The `-set-init-undef` option tells the `sat` command to initialize all registers to the undef (x) state. The way the x state is treated in Verilog will ensure that the solution will work for any initial state.

The `-max_undef` option instructs the `sat` command to find a solution with a maximum number of undefs. This way we can see clearly which inputs bits are relevant to the solution.

Finally the three `-set-at` options add constraints for the `y` signal to play the 1, 2, 3 sequence, starting with time step 4.

This produces the following output:

Todo: replace inline code

Listing 3.61: Solving a sequential SAT problem in the `memdemo` module.

```
yosys [memdemo]> sat -seq 6 -show y -show d -set-init-undef \
    -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3

1. Executing SAT pass (solving SAT problems in the circuit).
Full command line: sat -seq 6 -show y -show d -set-init-undef
    -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3

Setting up time step 1:
Final constraint equation: { } = { }
Imported 29 cells to SAT database.

Setting up time step 2:
Final constraint equation: { } = { }
Imported 29 cells to SAT database.

Setting up time step 3:
Final constraint equation: { } = { }
Imported 29 cells to SAT database.

Setting up time step 4:
Import set-constraint for timestep: \y = 4'0001
Final constraint equation: \y = 4'0001
Imported 29 cells to SAT database.

Setting up time step 5:
Import set-constraint for timestep: \y = 4'0010
Final constraint equation: \y = 4'0010
Imported 29 cells to SAT database.

Setting up time step 6:
Import set-constraint for timestep: \y = 4'0011
Final constraint equation: \y = 4'0011
Imported 29 cells to SAT database.

Setting up initial state:
Final constraint equation: { \y \s2 \s1 \mem[3] \mem[2] \mem[1]
    \mem[0] } = 24'xxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Import show expression: \y
Import show expression: \d
```

(continues on next page)

(continued from previous page)

Solving problem with 10322 variables and 27881 clauses..
 SAT model found. maximizing number of undefs.
 SAT solving finished - model found:

Time	Signal Name	Dec	Hex	Bin

init	\mem[0]	--	--	xxxx
init	\mem[1]	--	--	xxxx
init	\mem[2]	--	--	xxxx
init	\mem[3]	--	--	xxxx
init	\s1	--	--	xx
init	\s2	--	--	xx
init	\y	--	--	xxxx

1	\d	0	0	0000
1	\y	--	--	xxxx

2	\d	1	1	0001
2	\y	--	--	xxxx

3	\d	2	2	0010
3	\y	0	0	0000

4	\d	3	3	0011
4	\y	1	1	0001

5	\d	--	--	001x
5	\y	2	2	0010

6	\d	--	--	xxxx
6	\y	3	3	0011

It is not surprising that the solution sets `d = 0` in the first step, as this is the only way of setting the `s1` and `s2` registers to a known value. The input values for the other steps are a bit harder to work out manually, but the SAT solver finds the correct solution in an instant.

There is much more to write about the `sat` command. For example, there is a set of options that can be used to perform sequential proofs using temporal induction [EenSorensson03]. The command `help sat` can be used to print a list of all options with short descriptions of their functions.

3.2.4 Symbolic model checking

Todo: check text context

Note: While it is possible to perform model checking directly in Yosys, it is highly recommended to use SBY or EQY for formal hardware verification.

Symbolic Model Checking (SMC) is used to formally prove that a circuit has (or has not) a given property.

One application is Formal Equivalence Checking: Proving that two circuits are identical. For example this is a very useful feature when debugging custom passes in Yosys.

Other applications include checking if a module conforms to interface standards.

The `sat` command in Yosys can be used to perform Symbolic Model Checking.

Checking techmap

Todo: add/expand supporting text

Let's take a look at an example included in the Yosys code base under `docs/source/code_examples/synth_flow`:

Listing 3.62: `techmap_01_map.v`

```
module \${add} (A, B, Y);

parameter A_SIGNED = 0;
parameter B_SIGNED = 0;
parameter A_WIDTH = 1;
parameter B_WIDTH = 1;
parameter Y_WIDTH = 1;

input [A_WIDTH-1:0] A;
input [B_WIDTH-1:0] B;
output [Y_WIDTH-1:0] Y;

generate
  if ((A_WIDTH == 32) && (B_WIDTH == 32))
    begin
      wire [16:0] S1 = A[15:0] + B[15:0];
      wire [15:0] S2 = A[31:16] + B[31:16] + S1[16];
      assign Y = {S2[15:0], S1[15:0]};
    end
  else
    wire _TECHMAP_FAIL_ = 1;
endgenerate

endmodule
```

Listing 3.63: `techmap_01.v`

```
module test(input [31:0] a, b,
            output [31:0] y);
  assign y = a + b;
endmodule
```

Listing 3.64: `techmap_01.yo`

```
read_verilog techmap_01.v
hierarchy -check -top test
```

(continues on next page)

(continued from previous page)

```
techmap -map techmap_01_map.v;;
```

To see if it is correct we can use the following code:

Todo: replace inline code

```
# read test design
read_verilog techmap_01.v
hierarchy -top test

# create two version of the design: test_orig and test_mapped
copy test test_orig
rename test test_mapped

# apply the techmap only to test_mapped
techmap -map techmap_01_map.v test_mapped

# create a miter circuit to test equivalence
miter -equiv -make_assert -make_outputs test_orig test_mapped miter
flatten miter

# run equivalence check
sat -verify -prove-asserts -show-inputs -show-outputs miter
```

Result:

```
Solving problem with 945 variables and 2505 clauses..
SAT proof finished - no model found: SUCCESS!
```

AXI4 Stream Master

The code used in this section is included in the Yosys code base under [docs/source/code_examples/axis](#).

The following AXI4 Stream Master has a bug. But the bug is not exposed if the slave keeps `tready` asserted all the time. (Something a test bench might do.)

Symbolic Model Checking can be used to expose the bug and find a sequence of values for `tready` that yield the incorrect behavior.

Todo: add/expand supporting text

Listing 3.65: axis_master.v

```
module axis_master(aclk, aresetn, tvalid, tready, tdata);
    input aclk, aresetn, tready;
    output reg tvalid;
    output reg [7:0] tdata;

    reg [31:0] state;
```

(continues on next page)

(continued from previous page)

```

always @(posedge aclk) begin
    if (!aresetn) begin
        state <= 314159265;
        tvalid <= 0;
        tdata <= 'bx;
    end else begin
        if (tvalid && tready)
            tvalid <= 0;
        if (!tvalid || !tready) begin
            // ~ should not be inverted!
            state = state ^ state << 13;
            state = state ^ state >> 7;
            state = state ^ state << 17;
            if (state[9:8] == 0) begin
                tvalid <= 1;
                tdata <= state;
            end
        end
    end
end
endmodule

```

Listing 3.66: axis_test.v

```

module axis_test(aclk, tready);
    input aclk, tready;
    wire aresetn, tvalid;
    wire [7:0] tdata;

    integer counter = 0;
    reg aresetn = 0;

    axis_master uut (aclk, aresetn, tvalid, tready, tdata);

    always @(posedge aclk) begin
        if (aresetn && tready && tvalid) begin
            if (counter == 0) assert(tdata == 19);
            if (counter == 1) assert(tdata == 99);
            if (counter == 2) assert(tdata == 1);
            if (counter == 3) assert(tdata == 244);
            if (counter == 4) assert(tdata == 133);
            if (counter == 5) assert(tdata == 209);
            if (counter == 6) assert(tdata == 241);
            if (counter == 7) assert(tdata == 137);
            if (counter == 8) assert(tdata == 176);
            if (counter == 9) assert(tdata == 6);
            counter <= counter + 1;
        end
        aresetn <= 1;
    end
endmodule

```

Listing 3.67: test.yo

```
read_verilog -sv axis_master.v axis_test.v
hierarchy -top axis_test

proc; flatten;;
sat -seq 50 -prove-asserts
```

Result with unmodified axis_master.v:

Todo: replace inline code

```
Solving problem with 159344 variables and 442126 clauses..
SAT proof finished - model found: FAIL!
```

Result with fixed axis_master.v:

```
Solving problem with 159144 variables and 441626 clauses..
SAT proof finished - no model found: SUCCESS!
```

YOSYS INTERNALS

Todo: less academic

Yosys is an extensible open source hardware synthesis tool. It is aimed at designers who are looking for an easily accessible, universal, and vendor-independent synthesis tool, as well as scientists who do research in electronic design automation (EDA) and are looking for an open synthesis framework that can be used to test algorithms on complex real-world designs.

Yosys can synthesize a large subset of Verilog 2005 and has been tested with a wide range of real-world designs, including the [OpenRISC 1200 CPU](#), the [openMSP430 CPU](#), the [OpenCores I2C master](#), and the [k68 CPU](#).

Todo: add RISC-V core example

Yosys is written in C++, targeting C++17 at minimum. This chapter describes some of the fundamental Yosys data structures. For the sake of simplicity the C++ type names used in the Yosys implementation are used in this chapter, even though the chapter only explains the conceptual idea behind it and can be used as reference to implement a similar system in any language.

4.1 Internal flow

A (usually short) synthesis script controls Yosys.

These scripts contain three types of commands:

- **Frontends**, that read input files (usually Verilog);
- **Passes**, that perform transformations on the design in memory;
- **Backends**, that write the design in memory to a file (various formats are available: Verilog, BLIF, EDIF, SPICE, BTOR, ...).

4.1.1 Flow overview

Todo: less academic

Figure 4.1 shows the simplified data flow within Yosys. Rectangles in the figure represent program modules and ellipses internal data structures that are used to exchange design data between the program modules.

Design data is read in using one of the frontend modules. The high-level HDL frontends for Verilog and VHDL code generate an abstract syntax tree (AST) that is then passed to the AST frontend. Note that both HDL frontends use the same AST representation that is powerful enough to cover the Verilog HDL and VHDL language.

The AST Frontend then compiles the AST to Yosys's main internal data format, the RTL Intermediate Language (RTLIL). A more detailed description of this format is given in *The RTL Intermediate Language (RTLIL)*.

There is also a text representation of the RTLIL data structure that can be parsed using the RTLIL Frontend which is described in *RTLIL text representation*.

The design data may then be transformed using a series of passes that all operate on the RTLIL representation of the design.

Finally the design in RTLIL representation is converted back to text by one of the backends, namely the Verilog Backend for generating Verilog netlists and the RTLIL Backend for writing the RTLIL data in the same format that is understood by the RTLIL Frontend.

With the exception of the AST Frontend, which is called by the high-level HDL frontends and can't be called directly by the user, all program modules are called by the user (usually using a synthesis script that contains text commands for Yosys).

By combining passes in different ways and/or adding additional passes to Yosys it is possible to adapt Yosys to a wide range of applications. For this to be possible it is key that (1) all passes operate on the same data structure (RTLIL) and (2) that this data structure is powerful enough to represent the design in different stages of the synthesis.

4.1.2 Control and data flow

Todo: less academic

The data- and control-flow of a typical synthesis tool is very similar to the data- and control-flow of a typical compiler: different subsystems are called in a predetermined order, each consuming the data generated by the last subsystem and generating the data for the next subsystem (see Fig. 4.2).

The first subsystem to be called is usually called a frontend. It does not process the data generated by another subsystem but instead reads the user input—in the case of a HDL synthesis tool, the behavioural HDL code.

The subsystems that consume data from previous subsystems and produce data for the next subsystems (usually in the same or a similar format) are called passes.

The last subsystem that is executed transforms the data generated by the last pass into a suitable output format and writes it to a disk file. This subsystem is usually called the backend.

In Yosys all frontends, passes and backends are directly available as commands in the synthesis script. Thus the user can easily create a custom synthesis flow just by calling passes in the right order in a synthesis script.



Fig. 4.1: Yosys simplified data flow (ellipses: data structures, rectangles: program modules)



Fig. 4.2: General data- and control-flow of a synthesis tool

4.1.3 The Verilog and AST frontends

This chapter provides an overview of the implementation of the Yosys Verilog and AST frontends. The Verilog frontend reads Verilog-2005 code and creates an abstract syntax tree (AST) representation of the input. This AST representation is then passed to the AST frontend that converts it to RTLIL data, as illustrated in Fig. 4.3.



Fig. 4.3: Simplified Verilog to RTLIL data flow

Transforming Verilog to AST

The Verilog frontend converts the Verilog sources to an internal AST representation that closely resembles the structure of the original Verilog code. The Verilog frontend consists of three components, the Preprocessor, the Lexer and the Parser.

The source code to the Verilog frontend can be found in `frontends/verilog/` in the Yosys source tree.

The Verilog preprocessor

The Verilog preprocessor scans over the Verilog source code and interprets some of the Verilog compiler directives such as ``include`, ``define` and ``ifdef`.

It is implemented as a C++ function that is passed a file descriptor as input and returns the pre-processed Verilog code as a `std::string`.

The source code to the Verilog Preprocessor can be found in `frontends/verilog/preproc.cc` in the Yosys source tree.

The Verilog lexer

The Verilog Lexer is written using the lexer generator flex. Its source code can be found in `frontends/verilog/verilog_lexer.l` in the Yosys source tree. The lexer does little more than identifying all keywords and literals recognised by the Yosys Verilog frontend.

The lexer keeps track of the current location in the Verilog source code using some global variables. These variables are used by the constructor of AST nodes to annotate each node with the source code location it originated from.

Finally the lexer identifies and handles special comments such as “`// synopsys translate_off`” and “`// synopsys full_case`”. (It is recommended to use ``ifdef` constructs instead of the Synopsys `translate_on/off` comments and attributes such as `(* full_case *)` over “`// synopsys full_case`” whenever possible.)

The Verilog parser

The Verilog Parser is written using the parser generator bison. Its source code can be found in `frontends/verilog/verilog_parser.y` in the Yosys source tree.

It generates an AST using the `AST::AstNode` data structure defined in `frontends/ast/ast.h`. An `AST::AstNode` object has the following properties:

Table 4.1: AST node types with their corresponding Verilog constructs.

AST Node Type	Corresponding Verilog Construct
AST_NONE	This Node type should never be used.
AST_DESIGN	This node type is used for the top node of the AST tree. It has no corresponding Verilog construct.
AST_MODULE, AST_TASK, AST_FUNCTION	<code>module</code> , <code>task</code> and <code>function</code>
AST_WIRE	<code>input</code> , <code>output</code> , <code>wire</code> , <code>reg</code> and <code>integer</code>
AST_MEMORY	Verilog Arrays
AST_AUTOWIRE	Created by the simplifier when an undeclared signal name is used.
AST_PARAMETER, AST_LOCALPARAM	<code>parameter</code> and <code>localparam</code>
AST_PARASET	Parameter set in cell instantiation
AST_ARGUMENT	Port connection in cell instantiation
AST_RANGE	Bit-Index in a signal or element index in array
AST_CONSTANT	A literal value
AST_CELLTYPE	The type of cell in cell instantiation
AST_IDENTIFIER	An Identifier (signal name in expression or cell/task/etc. name in other contexts)
AST_PREFIX	Construct an identifier in the form <code><prefix>[<index>].<suffix></code> (used only in advanced generate constructs)
AST_FCALL, AST_TCALL	Call to function or task
AST_TO_SIGNED, AST_TO_UNSIGNED	The <code>\$signed()</code> and <code>\$unsigned()</code> functions
AST_CONCAT, AST_REPLICATE	The <code>{...}</code> and <code>{...{...}}</code> operators
AST_BIT_NOT, AST_BIT_AND, AST_BIT_OR, AST_BIT_XOR, AST_BIT_XNOR	The bitwise operators <code>~</code> , <code>&</code> , <code> </code> , <code>^</code> and <code>~^</code>
AST_REDUCE_AND, AST_REDUCE_OR, AST_REDUCE_XOR, AST_REDUCE_XNOR	The unary reduction operators <code>~</code> , <code>&</code> , <code> </code> , <code>^</code> and <code>~^</code>

continues on next page

Table 4.1 – continued from previous page

AST_REDUCE_BOOL	Conversion from multi-bit value to boolean value (equivalent to AST_REDUCE_OR)
AST_SHIFT_LEFT, AST_SHIFT_RIGHT, AST_SHIFT_SLEFT, AST_SHIFT_SRIGHT	The shift operators <<, >>, <<< and >>>
AST_LT, AST_LE, AST_EQ, AST_NE, AST_GE, AST_GT	The relational operators <, <=, ==, !=, >= and >
AST_ADD, AST_SUB, AST_MUL, AST_DIV, AST_MOD, AST_POW	The binary operators +, -, *, /, % and **
AST_POS, AST_NEG	The prefix operators + and -
AST_LOGIC_AND, AST_LOGIC_OR, AST_LOGIC_NOT	The logic operators &&, and !
AST_TERNARY	The ternary ?: -operator
AST_MEMRD AST_MEMWR	Read and write memories. These nodes are generated by the AST simplifier for writes/reads to/from Verilog arrays.
AST_ASSIGN	An assign statement
AST_CELL	A cell instantiation
AST_PRIMITIVE	A primitive cell (and , nand , or , etc.)
AST_ALWAYS, AST_INITIAL	Verilog always - and initial -blocks
AST_BLOCK	A begin-end -block
AST_ASSIGN_EQ, AST_ASSIGN_LE	Blocking (=) and nonblocking (<=) assignments within an always - or initial -block
AST_CASE, AST_COND, AST_DEFAULT	The case (if) statements, conditions within a case and the default case respectively
AST_FOR	A for -loop with an always - or initial -block
AST_GENVAR, AST_GENBLOCK, AST_GENFOR, AST_GENIF	The genvar and generate keywords and for and if within a generate block.
AST_POSEDGE, AST_NEGEDGE, AST_EDGE	Event conditions for always blocks.

- The node type
This enum (`AST::AstNodeType`) specifies the role of the node. [Table 4.1](#) contains a list of all node types.
- The child nodes
This is a list of pointers to all children in the abstract syntax tree.
- Attributes
As almost every AST node might have Verilog attributes assigned to it, the `AST::AstNode` has direct support for attributes. Note that the attribute values are again AST nodes.
- Node content
Each node might have additional content data. A series of member variables exist to hold such data. For example the member `std::string str` can hold a string value and is used e.g. in the `AST_IDENTIFIER` node type to store the identifier name.
- Source code location
Each `AST::AstNode` is automatically annotated with the current source code location by the `AST::AstNode` constructor. It is stored in the `std::string filename` and `int linenum` member variables.

The `AST::AstNode` constructor can be called with up to two child nodes that are automatically added to the list of child nodes for the new object. This simplifies the creation of AST nodes for simple expressions a bit. For example the bison code for parsing multiplications:

```

1 basic_expr '*' attr basic_expr {
2     $$ = new AstNode(AST_MUL, $1, $4);
3     append_attr($$, $3);
4 } |

```

The generated AST data structure is then passed directly to the AST frontend that performs the actual conversion to RTLIL.

Note that the Yosys command `read_verilog` provides the options `-yydebug` and `-dump_ast` that can be used to print the parse tree or abstract syntax tree respectively.

Transforming AST to RTLIL

The AST Frontend converts a set of modules in AST representation to modules in RTLIL representation and adds them to the current design. This is done in two steps: simplification and RTLIL generation.

The source code to the AST frontend can be found in `frontends/ast/` in the Yosys source tree.

AST simplification

A full-featured AST is too complex to be transformed into RTLIL directly. Therefore it must first be brought into a simpler form. This is done by calling the `AST::AstNode::simplify()` method of all `AST_MODULE` nodes in the AST. This initiates a recursive process that performs the following transformations on the AST data structure:

- Inline all task and function calls.
- Evaluate all `generate`-statements and unroll all `for`-loops.
- Perform const folding where it is necessary (e.g. in the value part of `AST_PARAMETER`, `AST_LOCALPARAM`, `AST_PARASET` and `AST_RANGE` nodes).
- Replace `AST_PRIMITIVE` nodes with appropriate `AST_ASSIGN` nodes.
- Replace dynamic bit ranges in the left-hand-side of assignments with `AST_CASE` nodes with `AST_COND` children for each possible case.
- Detect array access patterns that are too complicated for the `RTLIL::Memory` abstraction and replace them with a set of signals and cases for all reads and/or writes.
- Otherwise replace array accesses with `AST_MEMRD` and `AST_MEMWR` nodes.

In addition to these transformations, the simplifier also annotates the AST with additional information that is needed for the RTLIL generator, namely:

- All ranges (width of signals and bit selections) are not only const folded but (when a constant value is found) are also written to member variables in the `AST_RANGE` node.
- All identifiers are resolved and all `AST_IDENTIFIER` nodes are annotated with a pointer to the AST node that contains the declaration of the identifier. If no declaration has been found, an `AST_AUTOWIRE` node is created and used for the annotation.

This produces an AST that is fairly easy to convert to the RTLIL format.

Generating RTLIL

After AST simplification, the `AST::AstNode::genRTLIL()` method of each `AST_MODULE` node in the AST is called. This initiates a recursive process that generates equivalent RTLIL data for the AST data.

The `AST::AstNode::genRTLIL()` method returns an `RTLIL::SigSpec` structure. For nodes that represent expressions (operators, constants, signals, etc.), the cells needed to implement the calculation described by the expression are created and the resulting signal is returned. That way it is easy to generate the circuits for large expressions using depth-first recursion. For nodes that do not represent an expression (such as `AST_CELL`), the corresponding circuit is generated and an empty `RTLIL::SigSpec` is returned.

Synthesizing Verilog always blocks

For behavioural Verilog code (code utilizing `always`- and `initial`-blocks) it is necessary to also generate `RTLIL::Process` objects. This is done in the following way:

Whenever `AST::AstNode::genRTLIL()` encounters an `always`- or `initial`-block, it creates an instance of `AST_INTERNAL::ProcessGenerator`. This object then generates the `RTLIL::Process` object for the block. It also calls `AST::AstNode::genRTLIL()` for all right-hand-side expressions contained within the block.

First the `AST_INTERNAL::ProcessGenerator` creates a list of all signals assigned within the block. It then creates a set of temporary signals using the naming scheme `$ <number> \ <original_name>` for each of the assigned signals.

Then an `RTLIL::Process` is created that assigns all intermediate values for each left-hand-side signal to the temporary signal in its `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree.

Finally a `RTLIL::SyncRule` is created for the `RTLIL::Process` that assigns the temporary signals for the final values to the actual signals.

A process may also contain memory writes. A `RTLIL::MemWriteAction` is created for each of them.

Calls to `AST::AstNode::genRTLIL()` are generated for right hand sides as needed. When blocking assignments are used, `AST::AstNode::genRTLIL()` is configured using global variables to use the temporary signals that hold the correct intermediate values whenever one of the previously assigned signals is used in an expression.

Unfortunately the generation of a correct `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree for behavioural code is a non-trivial task. The AST frontend solves the problem using the approach described on the following pages. The following example illustrates what the algorithm is supposed to do. Consider the following Verilog code:

```
1  always @(posedge clock) begin
2      out1 = in1;
3      if (in2)
4          out1 = !out1;
5      out2 <= out1;
6      if (in3)
7          out2 <= out2;
8      if (in4)
9          if (in5)
10             out3 <= in6;
11             else
12                 out3 <= in7;
13      out1 = out1 ^ out2;
14  end
```

This is translated by the Verilog and AST frontends into the following RTLIL code (attributes, cell parameters and wire declarations not included):

```

1  cell $logic_not $logic_not$<input>:4$2
2      connect \A \in1
3      connect \Y $logic_not$<input>:4$2_Y
4  end
5  cell $xor $xor$<input>:13$3
6      connect \A $1\out1[0:0]
7      connect \B \out2
8      connect \Y $xor$<input>:13$3_Y
9  end
10 process $proc$<input>:1$1
11     assign $0\out3[0:0] \out3
12     assign $0\out2[0:0] $1\out1[0:0]
13     assign $0\out1[0:0] $xor$<input>:13$3_Y
14     switch \in2
15         case 1'1
16             assign $1\out1[0:0] $logic_not$<input>:4$2_Y
17         case
18             assign $1\out1[0:0] \in1
19     end
20     switch \in3
21         case 1'1
22             assign $0\out2[0:0] \out2
23         case
24     end
25     switch \in4
26         case 1'1
27             switch \in5
28                 case 1'1
29                     assign $0\out3[0:0] \in6
30                 case
31                     assign $0\out3[0:0] \in7
32             end
33         case
34     end
35     sync posedge \clock
36     update \out1 $0\out1[0:0]
37     update \out2 $0\out2[0:0]
38     update \out3 $0\out3[0:0]
39 end

```

Note that the two operators are translated into separate cells outside the generated process. The signal `out1` is assigned using blocking assignments and therefore `out1` has been replaced with a different signal in all expressions after the initial assignment. The signal `out2` is assigned using nonblocking assignments and therefore is not substituted on the right-hand-side expressions.

The `RTLIL::CaseRule/RTLIL::SwitchRule` tree must be interpreted the following way:

- On each case level (the body of the process is the root case), first the actions on this level are evaluated and then the switches within the case are evaluated. (Note that the last assignment on line 13 of the Verilog code has been moved to the beginning of the RTLIL process to line 13 of the RTLIL listing.)

I.e. the special cases deeper in the switch hierarchy override the defaults on the upper levels. The assignments in lines 12 and 22 of the RTLIL code serve as an example for this.

Note that in contrast to this, the order within the `RTLIL::SwitchRule` objects within a `RTLIL::CaseRule` is preserved with respect to the original AST and Verilog code.

- The whole `RTLIL::CaseRule/RTLIL::SwitchRule` tree describes an asynchronous circuit. I.e. the decision tree formed by the switches can be seen independently for each assigned signal. Whenever one assigned signal changes, all signals that depend on the changed signals are to be updated. For example the assignments in lines 16 and 18 in the RTLIL code in fact influence the assignment in line 12, even though they are in the “wrong order”.

The only synchronous part of the process is in the `RTLIL::SyncRule` object generated at line 35 in the RTLIL code. The sync rule is the only part of the process where the original signals are assigned. The synchronization event from the original Verilog code has been translated into the synchronization type (posedge) and signal (`\clock`) for the `RTLIL::SyncRule` object. In the case of this simple example the `RTLIL::SyncRule` object is later simply transformed into a set of d-type flip-flops and the `RTLIL::CaseRule/RTLIL::SwitchRule` tree to a decision tree using multiplexers.

In more complex examples (e.g. asynchronous resets) the part of the `RTLIL::CaseRule/RTLIL::SwitchRule` tree that describes the asynchronous reset must first be transformed to the correct `RTLIL::SyncRule` objects. This is done by the `proc_arst` pass.

The ProcessGenerator algorithm

The `AST_INTERNAL::ProcessGenerator` uses the following internal state variables:

- `subst_rvalue_from` and `subst_rvalue_to`
These two variables hold the replacement pattern that should be used by `AST::AstNode::genRTLIL()` for signals with blocking assignments. After initialization of `AST_INTERNAL::ProcessGenerator` these two variables are empty.
- `subst_lvalue_from` and `subst_lvalue_to`
These two variables contain the mapping from left-hand-side signals (`\ <name>`) to the current temporary signal for the same thing (initially `$0\ <name>`).
- `current_case`
A pointer to a `RTLIL::CaseRule` object. Initially this is the root case of the generated `RTLIL::Process`.

As the algorithm runs these variables are continuously modified as well as pushed to the stack and later restored to their earlier values by popping from the stack.

On startup the `ProcessGenerator` generates a new `RTLIL::Process` object with an empty root case and initializes its state variables as described above. Then the `RTLIL::SyncRule` objects are created using the synchronization events from the `AST_ALWAYS` node and the initial values of `subst_lvalue_from` and `subst_lvalue_to`. Then the AST for this process is evaluated recursively.

During this recursive evaluation, three different relevant types of AST nodes can be discovered: `AST_ASSIGN_LE` (nonblocking assignments), `AST_ASSIGN_EQ` (blocking assignments) and `AST_CASE` (if or case statement).

Handling of nonblocking assignments

When an `AST_ASSIGN_LE` node is discovered, the following actions are performed by the ProcessGenerator:

- The left-hand-side is evaluated using `AST::AstNode::genRTLIL()` and mapped to a temporary signal name using `subst_lvalue_from` and `subst_lvalue_to`.
- The right-hand-side is evaluated using `AST::AstNode::genRTLIL()`. For this call, the values of `subst_rvalue_from` and `subst_rvalue_to` are used to map blocking-assigned signals correctly.
- Remove all assignments to the same left-hand-side as this assignment from the `current_case` and all cases within it.
- Add the new assignment to the `current_case`.

Handling of blocking assignments

When an `AST_ASSIGN_EQ` node is discovered, the following actions are performed by the ProcessGenerator:

- Perform all the steps that would be performed for a nonblocking assignment (see above).
- Remove the found left-hand-side (before lvalue mapping) from `subst_rvalue_from` and also remove the respective bits from `subst_rvalue_to`.
- Append the found left-hand-side (before lvalue mapping) to `subst_rvalue_from` and append the found right-hand-side to `subst_rvalue_to`.

Handling of cases and if-statements

When an `AST_CASE` node is discovered, the following actions are performed by the ProcessGenerator:

- The values of `subst_rvalue_from`, `subst_rvalue_to`, `subst_lvalue_from` and `subst_lvalue_to` are pushed to the stack.
- A new `RTLIL::SwitchRule` object is generated, the selection expression is evaluated using `AST::AstNode::genRTLIL()` (with the use of `subst_rvalue_from` and `subst_rvalue_to`) and added to the `RTLIL::SwitchRule` object and the object is added to the `current_case`.
- All lvalues assigned to within the `AST_CASE` node using blocking assignments are collected and saved in the local variable `this_case_eq_lvalue`.
- New temporary signals are generated for all signals in `this_case_eq_lvalue` and stored in `this_case_eq_ltemp`.
- The signals in `this_case_eq_lvalue` are mapped using `subst_rvalue_from` and `subst_rvalue_to` and the resulting set of signals is stored in `this_case_eq_rvalue`.

Then the following steps are performed for each `AST_COND` node within the `AST_CASE` node:

- Set `subst_rvalue_from`, `subst_rvalue_to`, `subst_lvalue_from` and `subst_lvalue_to` to the values that have been pushed to the stack.
- Remove `this_case_eq_lvalue` from `subst_lvalue_from/subst_lvalue_to`.
- Append `this_case_eq_lvalue` to `subst_lvalue_from` and append `this_case_eq_ltemp` to `subst_lvalue_to`.
- Push the value of `current_case`.
- Create a new `RTLIL::CaseRule`. Set `current_case` to the new object and add the new object to the `RTLIL::SwitchRule` created above.

- Add an assignment from `this_case_eq_rvalue` to `this_case_eq_ltemp` to the new `current_case`.
- Evaluate the compare value for this case using `AST::AstNode::genRTLIL()` (with the use of `subst_rvalue_from` and `subst_rvalue_to`) modify the new `current_case` accordingly.
- Recursion into the children of the `AST_COND` node.
- Restore `current_case` by popping the old value from the stack.

Finally the following steps are performed:

- The values of `subst_rvalue_from`, `subst_rvalue_to`, `subst_lvalue_from` and `subst_lvalue_to` are popped from the stack.
- The signals from `this_case_eq_lvalue` are removed from the `subst_rvalue_from/subst_rvalue_to`-pair.
- The value of `this_case_eq_lvalue` is appended to `subst_rvalue_from` and the value of `this_case_eq_ltemp` is appended to `subst_rvalue_to`.
- Map the signals in `this_case_eq_lvalue` using `subst_lvalue_from/subst_lvalue_to`.
- Remove all assignments to signals in `this_case_eq_lvalue` in `current_case` and all cases within it.
- Add an assignment from `this_case_eq_ltemp` to `this_case_eq_lvalue` to `current_case`.

Further analysis of the algorithm for cases and if-statements

With respect to nonblocking assignments the algorithm is easy: later assignments invalidate earlier assignments. For each signal assigned using nonblocking assignments exactly one temporary variable is generated (with the `$0`-prefix) and this variable is used for all assignments of the variable.

Note how all the `_eq`-variables become empty when no blocking assignments are used and many of the steps in the algorithm can then be ignored as a result of this.

For a variable with blocking assignments the algorithm shows the following behaviour: First a new temporary variable is created. This new temporary variable is then registered as the assignment target for all assignments for this variable within the cases for this `AST_CASE` node. Then for each case the new temporary variable is first assigned the old temporary variable. This assignment is overwritten if the variable is actually assigned in this case and is kept as a default value otherwise.

This yields an `RTLIL::CaseRule` that assigns the new temporary variable in all branches. So when all cases have been processed a final assignment is added to the containing block that assigns the new temporary variable to the old one. Note how this step always overrides a previous assignment to the old temporary variable. Other than nonblocking assignments, the old assignment could still have an effect somewhere in the design, as there have been calls to `AST::AstNode::genRTLIL()` with a `subst_rvalue_from/subst_rvalue_to`-tuple that contained the right-hand-side of the old assignment.

The `proc` pass

The `ProcessGenerator` converts a behavioural model in AST representation to a behavioural model in `RTLIL::Process` representation. The actual conversion from a behavioural model to an RTL representation is performed by the `proc` pass and the passes it launches:

- `proc_clean` and `proc_rmdead`

These two passes just clean up the `RTLIL::Process` structure. The `proc_clean` pass removes empty parts (eg. empty assignments) from the process and `proc_rmdead` detects and removes unreachable branches from the process's decision trees.

- *proc_arst*
This pass detects processes that describe d-type flip-flops with asynchronous resets and rewrites the process to better reflect what they are modelling: Before this pass, an asynchronous reset has two edge-sensitive sync rules and one top-level RTLIL::SwitchRule for the reset path. After this pass the sync rule for the reset is level-sensitive and the top-level RTLIL::SwitchRule has been removed.
- *proc_mux*
This pass converts the RTLIL::CaseRule/RTLIL::SwitchRule-tree to a tree of multiplexers per written signal. After this, the RTLIL::Process structure only contains the RTLIL::SyncRule s that describe the output registers.
- *proc_dff*
This pass replaces the RTLIL::SyncRules to d-type flip-flops (with asynchronous resets if necessary).
- *proc_dff*
This pass replaces the RTLIL::MemWriteActions with *\$memwr* cells.
- *proc_clean*
A final call to *proc_clean* removes the now empty RTLIL::Process objects.

Performing these last processing steps in passes instead of in the Verilog frontend has two important benefits:

First it improves the transparency of the process. Everything that happens in a separate pass is easier to debug, as the RTLIL data structures can be easily investigated before and after each of the steps.

Second it improves flexibility. This scheme can easily be extended to support other types of storage-elements, such as sr-latches or d-latches, without having to extend the actual Verilog frontend.

Todo: Synthesizing Verilog arrays

Add some information on the generation of *\$memrd* and *\$memwr* cells and how they are processed in the memory pass.

Todo: Synthesizing parametric designs

Add some information on the RTLIL::Module::derive() method and how it is used to synthesize parametric modules via the hierarchy pass.

4.2 Internal formats

Yosys uses two different internal formats. The first is used to store an abstract syntax tree (AST) of a Verilog input file. This format is simply called AST and is generated by the Verilog Frontend. This data structure is consumed by a subsystem called AST Frontend¹. This AST Frontend then generates a design in Yosys' main internal format, the Register-Transfer-Level-Intermediate-Language (RTLIL) representation. It does that by first performing a number of simplifications within the AST representation and then generating RTLIL from the simplified AST data structure.

The RTLIL representation is used by all passes as input and outputs. This has the following advantages over using different representational formats between different passes:

- The passes can be rearranged in a different order and passes can be removed or inserted.

¹ In Yosys the term pass is only used to refer to commands that operate on the RTLIL data structure.

- Passes can simply pass-thru the parts of the design they don't change without the need to convert between formats. In fact Yosys passes output the same data structure they received as input and performs all changes in place.
- All passes use the same interface, thus reducing the effort required to understand a pass when reading the Yosys source code, e.g. when adding additional features.

The RTLIL representation is basically a netlist representation with the following additional features:

- An internal cell library with fixed-function cells to represent RTL datapath and register cells as well as logical gate-level cells (single-bit gates and registers).
- Support for multi-bit values that can use individual bits from wires as well as constant bits to represent coarse-grain netlists.
- Support for basic behavioural constructs (if-then-else structures and multi-case switches with a sensitivity list for updating the outputs).
- Support for multi-port memories.

The use of RTLIL also has the disadvantage of having a very powerful format between all passes, even when doing gate-level synthesis where the more advanced features are not needed. In order to reduce complexity for passes that operate on a low-level representation, these passes check the features used in the input RTLIL and fail to run when unsupported high-level constructs are used. In such cases a pass that transforms the higher-level constructs to lower-level constructs must be called from the synthesis script first.

4.2.1 The RTL Intermediate Language (RTLIL)

All frontends, passes and backends in Yosys operate on a design in RTLIL representation. The only exception are the high-level frontends that use the AST representation as an intermediate step before generating RTLIL data.

In order to avoid reinventing names for the RTLIL classes, they are simply referred to by their full C++ name, i.e. including the `RTLIL::` namespace prefix, in this document.

Figure 4.4 shows a simplified Entity-Relationship Diagram (ER Diagram) of RTLIL. In $1 : N$ relationships the arrow points from the N side to the 1. For example one `RTLIL::Design` contains N (zero to many) instances of `RTLIL::Module`. A two-pointed arrow indicates a $1 : 1$ relationship.

The `RTLIL::Design` is the root object of the RTLIL data structure. There is always one “current design” in memory which passes operate on, frontends add data to and backends convert to exportable formats. But in some cases passes internally generate additional `RTLIL::Design` objects. For example when a pass is reading an auxiliary Verilog file such as a cell library, it might create an additional `RTLIL::Design` object and call the Verilog frontend with this other object to parse the cell library.

There is only one active `RTLIL::Design` object that is used by all frontends, passes and backends called by the user, e.g. using a synthesis script. The `RTLIL::Design` then contains zero to many `RTLIL::Module` objects. This corresponds to modules in Verilog or entities in VHDL. Each module in turn contains objects from three different categories:

- `RTLIL::Cell` and `RTLIL::Wire` objects represent classical netlist data.
- `RTLIL::Process` objects represent the decision trees (if-then-else statements, etc.) and synchronization declarations (clock signals and sensitivity) from Verilog always and VHDL process blocks.
- `RTLIL::Memory` objects represent addressable memories (arrays).

Usually the output of the synthesis procedure is a netlist, i.e. all `RTLIL::Process` and `RTLIL::Memory` objects must be replaced by `RTLIL::Cell` and `RTLIL::Wire` objects by synthesis passes.

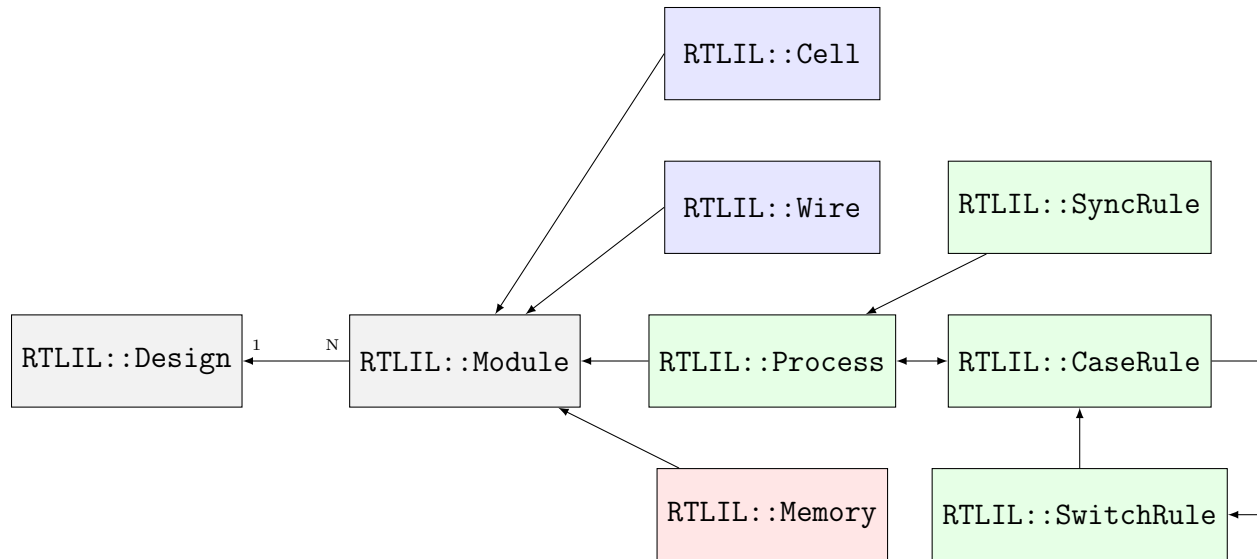


Fig. 4.4: Simplified RTLIL Entity-Relationship Diagram

All features of the HDL that cannot be mapped directly to these RTLIL classes must be transformed to an RTLIL-compatible representation by the HDL frontend. This includes Verilog-features such as generate-blocks, loops and parameters.

The following sections contain a more detailed description of the different parts of RTLIL and rationale behind some of the design decisions.

RTLIL identifiers

All identifiers in RTLIL (such as module names, port names, signal names, cell types, etc.) follow the following naming convention: they must either start with a backslash (\) or a dollar sign (\$).

Identifiers starting with a backslash are public visible identifiers. Usually they originate from one of the HDL input files. For example the signal name `\sig42` is most likely a signal that was declared using the name `sig42` in an HDL input file. On the other hand the signal name `$sig42` is an auto-generated signal name. The backends convert all identifiers that start with a dollar sign to identifiers that do not collide with identifiers that start with a backslash.

This has three advantages:

- First, it is impossible that an auto-generated identifier collides with an identifier that was provided by the user.
- Second, the information about which identifiers were originally provided by the user is always available which can help guide some optimizations. For example, `opt_clean` tries to preserve signals with a user-provided name but doesn't hesitate to delete signals that have auto-generated names when they just duplicate other signals. Note that this can be overridden with the `-purge` option to also delete internal nets with user-provided names.
- Third, the delicate job of finding suitable auto-generated public visible names is deferred to one central location. Internally auto-generated names that may hold important information for Yosys developers can be used without disturbing external tools. For example the Verilog backend assigns names in the form `_123_`.

Whitespace and control characters (any character with an ASCII code 32 or less) are not allowed in RTLIL identifiers; most frontends and backends cannot support these characters in identifiers.

In order to avoid programming errors, the RTLIL data structures check if all identifiers start with either a backslash or a dollar sign, and contain no whitespace or control characters. Violating these rules results in a runtime error.

All RTLIL identifiers are case sensitive.

Some transformations, such as flattening, may have to change identifiers provided by the user to avoid name collisions. When that happens, attribute `hdlname` is attached to the object with the changed identifier. This attribute contains one name (if emitted directly by the frontend, or is a result of disambiguation) or multiple names separated by spaces (if a result of flattening). All names specified in the `hdlname` attribute are public and do not include the leading `\`.

RTLIL::Design and RTLIL::Module

The `RTLIL::Design` object is basically just a container for `RTLIL::Module` objects. In addition to a list of `RTLIL::Module` objects the `RTLIL::Design` also keeps a list of selected objects, i.e. the objects that passes should operate on. In most cases the whole design is selected and therefore passes operate on the whole design. But this mechanism can be useful for more complex synthesis jobs in which only parts of the design should be affected by certain passes.

Besides the objects shown in the *ER diagram* above, an `RTLIL::Module` object contains the following additional properties:

- The module name
- A list of attributes
- A list of connections between wires
- An optional frontend callback used to derive parametrized variations of the module

The attributes can be Verilog attributes imported by the Verilog frontend or attributes assigned by passes. They can be used to store additional metadata about modules or just mark them to be used by certain part of the synthesis script but not by others.

Verilog and VHDL both support parametric modules (known as “generic entities” in VHDL). The RTLIL format does not support parametric modules itself. Instead each module contains a callback function into the AST frontend to generate a parametrized variation of the `RTLIL::Module` as needed. This callback then returns the auto-generated name of the parametrized variation of the module. (A hash over the parameters and the module name is used to prohibit the same parametrized variation from being generated twice. For modules with only a few parameters, a name directly containing all parameters is generated instead of a hash string.)

RTLIL::Cell and RTLIL::Wire

A module contains zero to many `RTLIL::Cell` and `RTLIL::Wire` objects. Objects of these types are used to model netlists. Usually the goal of all synthesis efforts is to convert all modules to a state where the functionality of the module is implemented only by cells from a given cell library and wires to connect these cells with each other. Note that module ports are just wires with a special property.

An `RTLIL::Wire` object has the following properties:

- The wire name
- A list of attributes
- A width (buses are just wires with a width more than 1)
- Bus direction (MSB to LSB or vice versa)

- Lowest valid bit index (LSB or MSB depending on bus direction)
- If the wire is a port: port number and direction (input/output/inout)

As with modules, the attributes can be Verilog attributes imported by the Verilog frontend or attributes assigned by passes.

In Yosys, busses (signal vectors) are represented using a single wire object with a width more than 1. So Yosys does not convert signal vectors to individual signals. This makes some aspects of RTLIL more complex but enables Yosys to be used for coarse grain synthesis where the cells of the target architecture operate on entire signal vectors instead of single bit wires.

In Verilog and VHDL, busses may have arbitrary bounds, and LSB can have either the lowest or the highest bit index. In RTLIL, bit 0 always corresponds to LSB; however, information from the HDL frontend is preserved so that the bus will be correctly indexed in error messages, backend output, constraint files, etc.

An `RTLIL::Cell` object has the following properties:

- The cell name and type
- A list of attributes
- A list of parameters (for parametric cells)
- Cell ports and the connections of ports to wires and constants

The connections of ports to wires are coded by assigning an `RTLIL::SigSpec` to each cell port. The `RTLIL::SigSpec` data type is described in the next section.

RTLIL::SigSpec

A “signal” is everything that can be applied to a cell port. I.e.

- Any constant value of arbitrary bit-width
1em For example: 1337, 16'b0000010100111001, 1'b1, 1'bx
- All bits of a wire or a selection of bits from a wire
1em For example: mywire, mywire[24], mywire[15:8]
- Concatenations of the above
1em For example: {16'd1337, mywire[15:8]}

The `RTLIL::SigSpec` data type is used to represent signals. The `RTLIL::Cell` object contains one `RTLIL::SigSpec` for each cell port.

In addition, connections between wires are represented using a pair of `RTLIL::SigSpec` objects. Such pairs are needed in different locations. Therefore the type name `RTLIL::SigSig` was defined for such a pair.

RTLIL::Process

When a high-level HDL frontend processes behavioural code it splits it up into data path logic (e.g. the expression `a + b` is replaced by the output of an adder that takes `a` and `b` as inputs) and an `RTLIL::Process` that models the control logic of the behavioural code. Let's consider a simple example:

```

1 module ff_with_en_and_async_reset(clock, reset, enable, d, q);
2 input clock, reset, enable, d;
3 output reg q;
4 always @(posedge clock, posedge reset)
5     if (reset)
```

(continues on next page)

(continued from previous page)

```

6         q <= 0;
7     else if (enable)
8         q <= d;
9 endmodule

```

In this example there is no data path and therefore the `RTLIL::Module` generated by the frontend only contains a few `RTLIL::Wire` objects and an `RTLIL::Process`. The `RTLIL::Process` in RTLIL syntax:

```

1 process $proc$ff_with_en_and_async_reset.v:4$1
2     assign $0\q[0:0] \q
3     switch \reset
4         case 1'1
5             assign $0\q[0:0] 1'0
6         case
7             switch \enable
8                 case 1'1
9                     assign $0\q[0:0] \d
10                case
11                end
12     end
13     sync posedge \clock
14         update \q $0\q[0:0]
15     sync posedge \reset
16         update \q $0\q[0:0]
17 end

```

This `RTLIL::Process` contains two `RTLIL::SyncRule` objects, two `RTLIL::SwitchRule` objects and five `RTLIL::CaseRule` objects. The wire `$0\q[0:0]` is an automatically created wire that holds the next value of `\q`. The lines 2..12 describe how `$0\q[0:0]` should be calculated. The lines 13..16 describe how the value of `$0\q[0:0]` is used to update `\q`.

An `RTLIL::Process` is a container for zero or more `RTLIL::SyncRule` objects and exactly one `RTLIL::CaseRule` object, which is called the root case.

An `RTLIL::SyncRule` object contains an (optional) synchronization condition (signal and edge-type), zero or more assignments (`RTLIL::SigSig`), and zero or more memory writes (`RTLIL::MemWriteAction`). The always synchronization condition is used to break combinatorial loops when a latch should be inferred instead.

An `RTLIL::CaseRule` is a container for zero or more assignments (`RTLIL::SigSig`) and zero or more `RTLIL::SwitchRule` objects. An `RTLIL::SwitchRule` objects is a container for zero or more `RTLIL::CaseRule` objects.

In the above example the lines 2..12 are the root case. Here `$0\q[0:0]` is first assigned the old value `\q` as default value (line 2). The root case also contains an `RTLIL::SwitchRule` object (lines 3..12). Such an object is very similar to the C switch statement as it uses a control signal (`\reset` in this case) to determine which of its cases should be active. The `RTLIL::SwitchRule` object then contains one `RTLIL::CaseRule` object per case. In this example there is a case¹ for `\reset == 1` that causes `$0\q[0:0]` to be set (lines 4 and 5) and a default case that in turn contains a switch that sets `$0\q[0:0]` to the value of `\d` if `\enable` is active (lines 6..11).

A case can specify zero or more compare values that will determine whether it matches. Each of the compare values must be the exact same width as the control signal. When more than one compare value is specified,

¹ The syntax `1'1` in the RTLIL code specifies a constant with a length of one bit (the first 1), and this bit is a one (the second 1).

the case matches if any of them matches the control signal; when zero compare values are specified, the case always matches (i.e. it is the default case).

A switch prioritizes cases from first to last: multiple cases can match, but only the first matched case becomes active. This normally synthesizes to a priority encoder. The `parallel_case` attribute allows passes to assume that no more than one case will match, and `full_case` attribute allows passes to assume that exactly one case will match; if these invariants are ever dynamically violated, the behavior is undefined. These attributes are useful when an invariant invisible to the synthesizer causes the control signal to never take certain bit patterns.

The lines 13..16 then cause `\q` to be updated whenever there is a positive clock edge on `\clock` or `\reset`.

In order to generate such a representation, the language frontend must be able to handle blocking and nonblocking assignments correctly. However, the language frontend does not need to identify the correct type of storage element for the output signal or generate multiplexers for the decision tree. This is done by passes that work on the RTLIL representation. Therefore it is relatively easy to substitute these steps with other algorithms that target different target architectures or perform optimizations or other transformations on the decision trees before further processing them.

One of the first actions performed on a design in RTLIL representation in most synthesis scripts is identifying asynchronous resets. This is usually done using the `proc_arst` pass. This pass transforms the above example to the following RTLIL::Process:

```

1 process $proc$ff_with_en_and_async_reset.v:4$1
2   assign $0\q[0:0] \q
3   switch \enable
4     case 1'1
5       assign $0\q[0:0] \d
6     case
7   end
8   sync posedge \clock
9     update \q $0\q[0:0]
10  sync high \reset
11    update \q 1'0
12 end

```

This pass has transformed the outer RTLIL::SwitchRule into a modified RTLIL::SyncRule object for the `\reset` signal. Further processing converts the RTLIL::Process into e.g. a d-type flip-flop with asynchronous reset and a multiplexer for the enable signal:

```

1 cell $adff $procdff$6
2   parameter \ARST_POLARITY 1'1
3   parameter \ARST_VALUE 1'0
4   parameter \CLK_POLARITY 1'1
5   parameter \WIDTH 1
6   connect \ARST \reset
7   connect \CLK \clock
8   connect \D $0\q[0:0]
9   connect \Q \q
10 end
11 cell $mux $procmux$3
12   parameter \WIDTH 1
13   connect \A \q
14   connect \B \d
15   connect \S \enable

```

(continues on next page)

(continued from previous page)

```
16     connect \Y $0\q[0:0]
17 end
```

Different combinations of passes may yield different results. Note that *\$adff* and *\$mux* are internal cell types that still need to be mapped to cell types from the target cell library.

Some passes refuse to operate on modules that still contain `RTLIL::Process` objects as the presence of these objects in a module increases the complexity. Therefore the passes to translate processes to a netlist of cells are usually called early in a synthesis script. The `proc` pass calls a series of other passes that together perform this conversion in a way that is suitable for most synthesis tasks.

RTLIL::Memory

For every array (memory) in the HDL code an `RTLIL::Memory` object is created. A memory object has the following properties:

- The memory name
- A list of attributes
- The width of an addressable word
- The size of the memory in number of words

All read accesses to the memory are transformed to *\$memrd* cells and all write accesses to *\$memwr* cells by the language frontend. These cells consist of independent read- and write-ports to the memory. Memory initialization is transformed to *\$meminit* cells by the language frontend. The `\MEMID` parameter on these cells is used to link them together and to the `RTLIL::Memory` object they belong to.

The rationale behind using separate cells for the individual ports versus creating a large multiport memory cell right in the language frontend is that the separate *\$memrd* and *\$memwr* cells can be consolidated using resource sharing. As resource sharing is a non-trivial optimization problem where different synthesis tasks can have different requirements it lends itself to do the optimisation in separate passes and merge the `RTLIL::Memory` objects and *\$memrd* and *\$memwr* cells to multiport memory blocks after resource sharing is completed.

The memory pass performs this conversion and can (depending on the options passed to it) transform the memories directly to d-type flip-flops and address logic or yield multiport memory blocks (represented using *\$mem* cells).

See *Memories* for details about the memory cell types.

4.3 Working with the Yosys codebase

This section goes into additional detail on the Yosys source code and git repository. This information is not needed for simply using Yosys, but may be of interest for developers looking to customise Yosys builds.

4.3.1 Writing extensions

Todo: check text is coherent

Todo: update to use `/code_examples/extensions/test*.log`

This chapter contains some bits and pieces of information about programming yosys extensions. Don't be afraid to ask questions on the YosysHQ Slack.

Todo: mention coding guide

Quick guide

Code examples from this section are included in the `docs/source/code_examples/extensions` directory of the Yosys source code.

Program components and data formats

See *The RTL Intermediate Language (RTLIL)* document for more information about the internal data storage format used in Yosys and the classes that it provides.

This document will focus on the much simpler version of RTLIL left after the commands `proc` and `memory` (or `memory -nomap`):

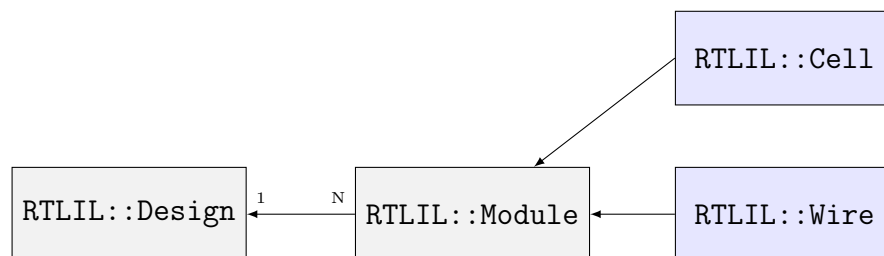


Fig. 4.5: Simplified RTLIL entity-relationship diagram without memories and processes

It is possible to only work on this simpler version:

Todo: consider replacing inline code

```

for (RTLIL::Module *module : design->selected_modules()) {
    if (module->has_memories_warn() || module->has_processes_warn())
        continue;
    ....
}
  
```

When trying to understand what a command does, creating a small test case to look at the output of `dump` and `show` before and after the command has been executed can be helpful. *Selections* has more information on using these commands.

Creating a command

Todo: add/expand supporting text

Let's create a very simple test command which prints the arguments we called it with, and lists off the current design's modules.

Listing 4.1: Example command `my_cmd` from `my_cmd.cc`

```
#include <kernel/yosys.h>
USING_YOSYS_NAMESPACE

struct MyPass : public Pass {
    MyPass() : Pass("my_cmd", "just a simple test") { }
    void execute(std::vector<std::string> args, RTLIL::Design *design) override
    {
        log("Arguments to my_cmd:\n");
        for (auto &arg : args)
            log("  %s\n", arg.c_str());

        log("Modules in current design:\n");
        for (auto mod : design->modules())
            log("  %s (%d wires, %d cells)\n", log_id(mod),
                GetSize(mod->wires()), GetSize(mod->cells()));
    }
} MyPass;
```

Note that we are making a global instance of a class derived from `Yosys::Pass`, which we get by including `kernel/yosys.h`.

Compiling to a plugin

Yosys can be extended by adding additional C++ code to the Yosys code base, or by loading plugins into Yosys. For maintainability it is generally recommended to create plugins.

The following command compiles our example `my_cmd` to a Yosys plugin:

Todo: replace inline code

```
yosys-config --exec --cxx --cxxflags --ldflags \
-o my_cmd.so -shared my_cmd.cc --ldlibs
```

Or shorter:

```
yosys-config --build my_cmd.so my_cmd.cc
```

Running Yosys with the `-m` option allows the plugin to be used. Here's a quick example that also uses the `-p` option to run `my_cmd foo bar`.

```
$ yosys -m ./my_cmd.so -p 'my_cmd foo bar'

-- Running command `my_cmd foo bar' --
Arguments to my_cmd:
  my_cmd
  foo
  bar
Modules in current design:
```

Creating modules from scratch

Let's create the following module using the RTLIL API:

Listing 4.2: absval_ref.v

```
module absval_ref(input signed [3:0] a, output [3:0] y);
    assign y = a[3] ? -a : a;
endmodule
```

We'll do the same as before and format it as a `Yosys::Pass`.

Listing 4.3: test1 - creating the absval module, from my_cmd.cc

```
struct Test1Pass : public Pass {
    Test1Pass() : Pass("test1", "creating the absval module") { }
    void execute(std::vector<std::string>, RTLIL::Design *design) override
    {
        if (design->has("\\absval") != 0)
            log_error("A module with the name absval already exists!\n");

        RTLIL::Module *module = design->addModule("\\absval");
        log("Name of this module: %s\n", log_id(module));

        RTLIL::Wire *a = module->addWire("\\a", 4);
        a->port_input = true;
        a->port_id = 1;

        RTLIL::Wire *y = module->addWire("\\y", 4);
        y->port_output = true;
        y->port_id = 2;

        RTLIL::Wire *a_inv = module->addWire(NEW_ID, 4);
        module->addNeg(NEW_ID, a, a_inv, true);
        module->addMux(NEW_ID, a, a_inv, RTLIL::SigSpec(a, 3), y);

        module->fixup_ports();
    }
} Test1Pass;
```

```
$ yosys -m ./my_cmd.so -p 'test1' -Q
```

(continues on next page)

(continued from previous page)

```
-- Running command `test1' --
Name of this module: absval
```

And if we look at the schematic for this new module we see the following:

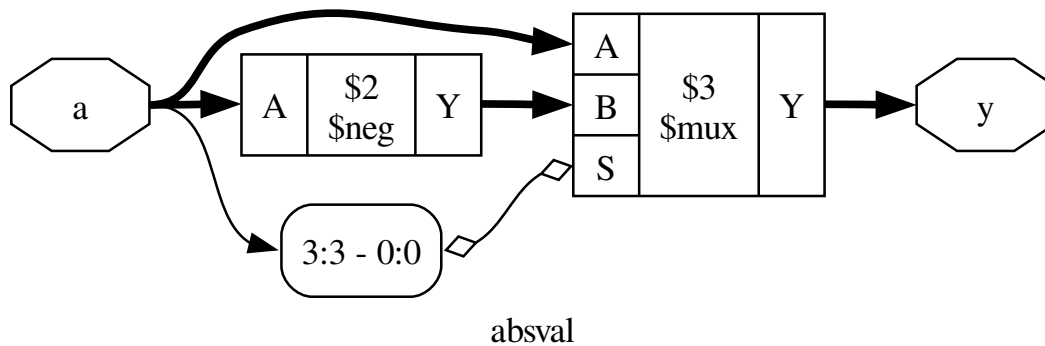


Fig. 4.6: Output of `yosys -m ./my_cmd.so -p 'test1; show'`

Modifying modules

Most commands modify existing modules, not create new ones.

When modifying existing modules, stick to the following DOs and DON'Ts:

- Do not remove wires. Simply disconnect them and let a successive *clean* command worry about removing it.
- Use `module->fixup_ports()` after changing the `port_*` properties of wires.
- You can safely remove cells or change the `connections` property of a cell, but be careful when changing the size of the `SigSpec` connected to a cell port.
- Use the `SigMap` helper class (see next section) when you need a unique handle for each signal bit.

Using the SigMap helper class

Consider the following module:

Listing 4.4: `sigmap_test.v`

```
module test(input a, output x, y);
    assign x = a, y = a;
endmodule
```

In this case `a`, `x`, and `y` are all different names for the same signal. However:

Todo: use my_cmd.cc literalincludes

```
RTLIL::SigSpec a(module->wire("\\a")), x(module->wire("\\x")),
                                     y(module->wire("\\y"));
log("%d %d %d\n", a == x, x == y, y == a); // will print "0 0 0"
```

The SigMap helper class can be used to map all such aliasing signals to a unique signal from the group (usually the wire that is directly driven by a cell or port).

```
SigMap sigmap(module);
log("%d %d %d\n", sigmap(a) == sigmap(x), sigmap(x) == sigmap(y),
    sigmap(y) == sigmap(a)); // will print "1 1 1"
```

Printing log messages

The log() function is a printf()-like function that can be used to create log messages.

Use log_signal() to create a C-string for a SigSpec object:

```
log("Mapped signal x: %s\n", log_signal(sigmap(x)));
```

The pointer returned by log_signal() is automatically freed by the log framework at a later time.

Use log_id() to create a C-string for an RTLIL::IdString:

```
log("Name of this module: %s\n", log_id(module->name));
```

Use log_header() and log_push()/log_pop() to structure log messages:

Todo: replace inline code

```
log_header(design, "Doing important stuff!\n");
log_push();
for (int i = 0; i < 10; i++)
    log("Log message #%d.\n", i);
log_pop();
```

Error handling

Use log_error() to report a non-recoverable error:

Todo: replace inline code

```
if (design->modules.count(module->name) != 0)
    log_error("A module with the name %s already exists!\n",
        RTLIL::id2cstr(module->name));
```

Use log_cmd_error() to report a recoverable error:

```
if (design->selection_stack.back().empty())
    log_cmd_error("This command can't operator on an empty selection!\n");
```

Use `log_assert()` and `log_abort()` instead of `assert()` and `abort()`.

The “stubnets” example module

The following is the complete code of the “stubnets” example module. It is included in the Yosys source distribution under `docs/source/code_examples/stubnets`.

Listing 4.5: `stubnets.cc`

```
1  // This is free and unencumbered software released into the public domain.
2  //
3  // Anyone is free to copy, modify, publish, use, compile, sell, or
4  // distribute this software, either in source code form or as a compiled
5  // binary, for any purpose, commercial or non-commercial, and by any
6  // means.
7
8  #include <kernel/yosys.h>
9  #include <kernel/sigtools.h>
10
11 #include <string>
12 #include <map>
13 #include <set>
14
15 USING_YOSYS_NAMESPACE
16 PRIVATE_NAMESPACE_BEGIN
17
18 // this function is called for each module in the design
19 static void find_stub_nets(RTLIL::Design *design, RTLIL::Module *module, bool report_
    ↪bits)
20 {
21     // use a SigMap to convert nets to a unique representation
22     SigMap sigmap(module);
23
24     // count how many times a single-bit signal is used
25     std::map<RTLIL::SigBit, int> bit_usage_count;
26
27     // count output lines for this module (needed only for summary output at the end)
28     int line_count = 0;
29
30     log("Looking for stub wires in module %s:\n", RTLIL::id2cstr(module->name));
31
32     // For all ports on all cells
33     for (auto &cell_iter : module->cells_)
34     for (auto &conn : cell_iter.second->connections())
35     {
36         // Get the signals on the port
37         // (use sigmap to get a unique signal name)
38         RTLIL::SigSpec sig = sigmap(conn.second);
39     }
```

(continues on next page)

(continued from previous page)

```

40         // add each bit to bit_usage_count, unless it is a constant
41         for (auto &bit : sig)
42             if (bit.wire != NULL)
43                 bit_usage_count[bit]++;
44     }
45
46     // for each wire in the module
47     for (auto &wire_iter : module->wires_)
48     {
49         RTLIL::Wire *wire = wire_iter.second;
50
51         // .. but only selected wires
52         if (!design->selected(module, wire))
53             continue;
54
55         // add +1 usage if this wire actually is a port
56         int usage_offset = wire->port_id > 0 ? 1 : 0;
57
58         // we will record which bits of the (possibly multi-bit) wire are stub
40→signals
59         std::set<int> stub_bits;
60
61         // get a signal description for this wire and split it into separate bits
62         RTLIL::SigSpec sig = sigmap(wire);
63
64         // for each bit (unless it is a constant):
65         // check if it is used at least two times and add to stub_bits otherwise
66         for (int i = 0; i < GetSize(sig); i++)
67             if (sig[i].wire != NULL && (bit_usage_count[sig[i]] + usage_
40→offset) < 2)
68                 stub_bits.insert(i);
69
70         // continue if no stub bits found
71         if (stub_bits.size() == 0)
72             continue;
73
74         // report stub bits and/or stub wires, don't report single bits
75         // if called with report_bits set to false.
76         if (GetSize(stub_bits) == GetSize(sig)) {
77             log(" found stub wire: %s\n", RTLIL::id2cstr(wire->name));
78         } else {
79             if (!report_bits)
80                 continue;
81             log(" found wire with stub bits: %s [", RTLIL::id2cstr(wire->
40→name));
82             for (int bit : stub_bits)
83                 log("%s%d", bit == *stub_bits.begin() ? "" : ", ", bit);
84             log("]\n");
85         }
86
87         // we have outputted a line, increment summary counter
88         line_count++;

```

(continues on next page)

(continued from previous page)

```

89     }
90
91     // report summary
92     if (report_bits)
93         log("  found %d stub wires or wires with stub bits.\n", line_count);
94     else
95         log("  found %d stub wires.\n", line_count);
96 }
97
98 // each pass contains a singleton object that is derived from Pass
99 struct StubnetsPass : public Pass {
100     StubnetsPass() : Pass("stubnets") { }
101     void execute(std::vector<std::string> args, RTLIL::Design *design) override
102     {
103         // variables to mirror information from passed options
104         bool report_bits = 0;
105
106         log_header(design, "Executing STUBNETS pass (find stub nets).\n");
107
108         // parse options
109         size_t argidx;
110         for (argidx = 1; argidx < args.size(); argidx++) {
111             std::string arg = args[argidx];
112             if (arg == "-report_bits") {
113                 report_bits = true;
114                 continue;
115             }
116             break;
117         }
118
119         // handle extra options (e.g. selection)
120         extra_args(args, argidx, design);
121
122         // call find_stub_nets() for each module that is either
123         // selected as a whole or contains selected objects.
124         for (auto &it : design->modules_)
125             if (design->selected_module(it.first))
126                 find_stub_nets(design, it.second, report_bits);
127     }
128 } StubnetsPass;
129
130 PRIVATE_NAMESPACE_END

```

Listing 4.6: Makefile

```

1 .PHONY: all dots examples
2 all: dots examples
3 dots:
4 examples:
5
6 .PHONY: test
7 test: stubnets.so

```

(continues on next page)

(continued from previous page)

```

8      yosys -ql test1.log -m ./stubnets.so test.v -p "stubnets"
9      yosys -ql test2.log -m ./stubnets.so test.v -p "opt; stubnets"
10     yosys -ql test3.log -m ./stubnets.so test.v -p "techmap; opt; stubnets -report_
↪bits"
11     tail test1.log test2.log test3.log
12
13 stubnets.so: stubnets.cc
14     yosys-config --exec --cxx --cxxflags -I../..../.. --ldflags -o $@ -shared $^ --
↪ldlibs
15
16 .PHONY: clean
17 clean:
18     rm -f test1.log test2.log test3.log
19     rm -f stubnets.so stubnets.d

```

Listing 4.7: test.v

```

1 module uut(in1, in2, in3, out1, out2);
2
3 input [8:0] in1, in2, in3;
4 output [8:0] out1, out2;
5
6 assign out1 = in1 + in2 + (in3 >> 4);
7
8 endmodule

```

4.3.2 Compiling with Verific library

The easiest way to get Yosys with Verific support is to [contact YosysHQ](#) for a [Tabby CAD Suite](#) evaluation license and download link. The TabbyCAD Suite includes additional patches and a custom extensions library in order to get the most out of the Verific parser when using Yosys.

If you already have a license for the Verific parser, in either source or binary form, you may be able to compile Yosys with partial Verific support yourself.

The Yosys-Verific patch

YosysHQ maintains and develops a patch for Verific in order to better integrate with Yosys and to provide features required by some of the formal verification front-end tools. To license this patch for your own Yosys builds, [contact YosysHQ](#).

Warning: While synthesis from RTL may be possible without this patch, YosysHQ provides no guarantees of correctness and is unable to provide support.

We recommend against using unpatched Yosys+Verific builds in conjunction with the formal verification front-end tools unless you are familiar with their inner workings. There are cases where the tools will appear to work, while producing incorrect results.

Note: Some of the formal verification front-end tools may not be fully supported without the full TabbyCAD

suite. If you want to use these tools, including SBY, make sure to ask us if the Yosys-Verific patch is right for you.

Compile options

To enable Verific support `ENABLE_VERIFIC` has to be set to 1 and `VERIFIC_DIR` needs to point to the location where the library is located.

Parameter	Default	Description
<code>ENABLE_VERIFIC</code>	0	Enable compilation with Verific
<code>VERIFIC_DIR</code>	<code>/usr/local/src/verific_lib</code>	Library and headers location

Since there are multiple Verific library builds and they can have different features, there are compile options to select them.

Parameter	Default	Description
<code>ENABLE_VERIFIC_SYSTEMVERILOG</code>	1	SystemVerilog support
<code>ENABLE_VERIFIC_VHDL</code>	1	VHDL support
<code>ENABLE_VERIFIC_HIER_TREE</code>	1	Hierarchy tree support
<code>ENABLE_VERIFIC_YOSYSHQ_EXTENSIONS</code>	0	YosysHQ specific extensions support
<code>ENABLE_VERIFIC_EDIF</code>	0	EDIF support
<code>ENABLE_VERIFIC_LIBERTY</code>	0	Liberty file support

To find the compile options used for a given Yosys build, call `yosys-config --cxxflags`. This documentation was built with the following compile options:

```
--cxxflags -O2 -flto=auto -ffat-lto-objects -fexceptions -g \  
-grecord-gcc-switches -pipe -Wall -Werror=format-security \  
-Wp,-U_FORTIFY_SOURCE,-D_FORTIFY_SOURCE=3 \  
-Wp,-D_GLIBCXX_ASSERTIONS \  
-specs=/usr/lib/rpm/redhat/redhat-hardened-cc1 \  
-fstack-protector-strong \  
-specs=/usr/lib/rpm/redhat/redhat-annobin-cc1 \  
-mbranch-protection=standard -fasynchronous-unwind-tables \  
-fstack-clash-protection -fno-omit-frame-pointer \  
-mno-omit-leaf-frame-pointer -Wall -Wextra -ggdb \  
-I"/usr/include/yosys" -MD -MP -D_YOSYS_ -fPIC \  
-I/usr/include -DYOSYS_VER=\"0.51\" -DYOSYS_MAJOR=0 \  
-DYOSYS_MINOR=51 -DYOSYS_COMMIT=0.51 -std=c++17 \  
-O3 -DYOSYS_ENABLE_READLINE -DYOSYS_ENABLE_PLUGINS \  
-DYOSYS_ENABLE_GLOB -DYOSYS_ENABLE_ZLIB -DWITH_GZFILEOP \  
-DYOSYS_ENABLE_TCL -DYOSYS_ENABLE_ABC -DYOSYS_ENABLE_COVER
```

Note: The YosysHQ specific extensions are only available with the TabbyCAD suite.

Required Verific features

The following features, along with their corresponding Yosys build parameters, are required for the Yosys-Verific patch:

- RTL elaboration with
 - SystemVerilog with `ENABLE_VERIFIC_SYSTEMVERILOG`, and/or
 - VHDL support with `ENABLE_VERIFIC_VHDL`.
- Hierarchy tree support and static elaboration with `ENABLE_VERIFIC_HIER_TREE`.

Please be aware that the following Verific configuration build parameter needs to be enabled in order to create the fully supported build:

```
database/DBCompileFlags.h:  
    DB_PRESERVE_INITIAL_VALUE
```

Note: Yosys+Verific builds may compile without these features, but we provide no guarantees and cannot offer support if they are disabled or the Yosys-Verific patch is not used.

Optional Verific features

The following Verific features are available with TabbyCAD and can be enabled in Yosys builds:

- EDIF support with `ENABLE_VERIFIC_EDIF`, and
- Liberty file support with `ENABLE_VERIFIC_LIBERTY`.

Partially supported builds

This section describes Yosys+Verific configurations which we have confirmed as working in the past, however they are not a part of our regular tests so we cannot guarantee they are still functional.

To be able to compile Yosys with Verific, the Verific library must have support for at least one HDL language with RTL elaboration enabled. The following table lists a series of build configurations which are possible, but only provide a limited subset of features. Please note that support is limited without YosysHQ specific extensions of Verific library.

Configuration values:

- `ENABLE_VERIFIC_SYSTEMVERILOG`
- `ENABLE_VERIFIC_VHDL`
- `ENABLE_VERIFIC_HIER_TREE`
- `ENABLE_VERIFIC_YOSYSHQ_EXTENSIONS`

Features	Configuration values			
	a	b	c	d
SystemVerilog + RTL elaboration	1	0	0	0
VHDL + RTL elaboration	0	1	0	0
SystemVerilog + VHDL + RTL elaboration	1	1	0	0
SystemVerilog + RTL elaboration + Static elaboration + Hier tree	1	0	1	0
VHDL + RTL elaboration + Static elaboration + Hier tree	0	1	1	0
SystemVerilog + VHDL + RTL elaboration + Static elaboration + Hier tree	1	1	1	0

Note: In case your Verific build has EDIF and/or Liberty support, you can enable those options. These are not mentioned above for simplification and since they are disabled by default.

4.3.3 Writing a new backend using FunctionalIR

What is FunctionalIR

To simplify the writing of backends for functional languages or similar targets, Yosys provides an alternative intermediate representation called FunctionalIR which maps more directly on those targets.

FunctionalIR represents the design as a function `(inputs, current_state) -> (outputs, next_state)`. This function is broken down into a series of assignments to variables. Each assignment is a simple operation, such as an addition. Complex operations are broken up into multiple steps. For example, an RTLIL addition will be translated into a sign/zero extension of the inputs, followed by an addition.

Like SSA form, each variable is assigned to exactly once. We can thus treat variables and assignments as equivalent and, since this is a graph-like representation, those variables are also called “nodes”. Unlike RTLIL’s cells and wires representation, this representation is strictly ordered (topologically sorted) with definitions preceding their use.

Every node has a “sort” (the FunctionalIR term for what might otherwise be called a “type”). The sorts available are

- `bit[n]` for an `n`-bit bitvector, and
- `memory[n,m]` for an immutable array of `2**n` values of sort `bit[m]`.

In terms of actual code, Yosys provides a class `Functional::IR` that represents a design in FunctionalIR. `Functional::IR::from_module` generates an instance from an RTLIL module. The entire design is stored as a whole in an internal data structure. To access the design, the `Functional::Node` class provides a reference to a particular node in the design. The `Functional::IR` class supports the syntax `for(auto node : ir)` to iterate over every node.

`Functional::IR` also keeps track of inputs, outputs and states. By a “state” we mean a pair of a “current state” input and a “next state” output. One such pair is created for every register and for every memory. Every input, output and state has a name (equal to their name in RTLIL), a sort and a kind. The kind field usually remains as the default value `$input`, `$output` or `$state`, however some RTLIL cells such as `$assert` or `$anyseq` generate auxiliary inputs/outputs/states that are given a different kind to distinguish them from ordinary RTLIL inputs/outputs/states.

- To access an individual input/output/state, use `ir.input(name, kind)`, `ir.output(name, kind)` or `ir.state(name, kind)`. `kind` defaults to the default kind.

- To iterate over all inputs/outputs/states of a certain kind, methods `ir.inputs`, `ir.outputs`, `ir.states` are provided. Their argument defaults to the default kinds mentioned.
- To iterate over inputs/outputs/states of any kind, use `ir.all_inputs`, `ir.all_outputs` and `ir.all_states`.
- Outputs have a node that indicate the value of the output, this can be retrieved via `output.value()`.
- States have a node that indicate the next value of the state, this can be retrieved via `state.next_value()`. They also have an initial value that is accessed as either `state.initial_value_signal()` or `state.initial_value_memory()`, depending on their sort.

Each node has a “function”, which defines its operation (for a complete list of functions and a specification of their operation, see `functional.h`). Functions are represented as an enum `Functional::Fn` and the function field can be accessed as `node.fn()`. Since the most common operation is a switch over the function that also accesses the arguments, the `Node` class provides a method `visit` that implements the visitor pattern. For example, for an addition node `node` with arguments `n1` and `n2`, `node.visit(visitor)` would call `visitor.add(node, n1, n2)`. Thus typically one would implement a class with a method for every function. Visitors should inherit from either `Functional::AbstractVisitor<ReturnType>` or `Functional::DefaultVisitor<ReturnType>`. The former will produce a compiler error if a case is unhandled, the latter will call `default_handler(node)` instead. Visitor methods should be marked as `override` to provide compiler errors if the arguments are wrong.

Utility classes

`functional.h` also provides utility classes that are independent of the main `FunctionalIR` representation but are likely to be useful for backends.

`Functional::Writer` provides a simple formatting class that wraps a `std::ostream` and provides the following methods:

- `writer << value` wraps `os << value`.
- `writer.print(fmt, value0, value1, value2, ...)` replaces `{0}`, `{1}`, `{2}`, etc in the string `fmt` with `value0`, `value1`, `value2`, resp. Each value is formatted using `os << value`. It is also possible to write `{}` to refer to one past the last index, i.e. `{1} {} {} {7} {}` is equivalent to `{1} {2} {3} {7} {8}`.
- `writer.print_with(fn, fmt, value0, value1, value2, ...)` functions much the same as `print` but it uses `os << fn(value)` to print each value and falls back to `os << value` if `fn(value)` is not legal.

`Functional::Scope` keeps track of variable names in a target language. It is used to translate between different sets of legal characters and to avoid accidentally re-defining identifiers. Users should derive a class from `Scope` and supply the following:

- `Scope<Id>` takes a template argument that specifies a type that’s used to uniquely distinguish variables. Typically this would be `int` (if variables are used for `Functional::IR` nodes) or `IdString`.
- The derived class should provide a constructor that calls `reserve` for every reserved word in the target language.
- A method `bool is_character_legal(char c, int index)` has to be provided that returns `true` iff `c` is legal in an identifier at position `index`.

Given an instance `scope` of the derived class, the following methods are then available:

- `scope.reserve(std::string name)` marks the given name as being in-use

- `scope.unique_name(IdString suggestion)` generates a previously unused name and attempts to make it similar to `suggestion`.
- `scope(Id id, IdString suggestion)` functions similar to `unique_name`, except that multiple calls with the same `id` are guaranteed to retrieve the same name (independent of `suggestion`).

`sexpr.h` provides classes that represent and pretty-print s-expressions. S-expressions can be constructed with `SEExpr::list`, for example `SEExpr expr = SEExpr::list("add", "x", SEExpr::list("mul", "y", "z"))` represents `(add x (mul y z))` (by adding using `SEExprUtil::list` to the top of the file, `list` can be used as shorthand for `SEExpr::list`). For prettyprinting, `SEExprWriter` wraps an `std::ostream` and provides the following methods:

- `writer << sexpr` writes the provided expression to the output, breaking long lines and adding appropriate indentation.
- `writer.open(sexpr)` is similar to `writer << sexpr` but will omit the last closing parenthesis. Further arguments can then be added separately with `<<` or `open`. This allows for printing large s-expressions without needing to construct the whole expression in memory first.
- `writer.open(sexpr, false)` is similar to `writer.open(sexpr)` but further arguments will not be indented. This is used to avoid unlimited indentation on structures with unlimited nesting.
- `writer.close(n = 1)` closes the last `n` open s-expressions.
- `writer.push()` and `writer.pop()` are used to automatically close s-expressions. `writer.pop()` closes all s-expressions opened since the last call to `writer.push()`.
- `writer.comment(string)` writes a comment on a separate-line. `writer.comment(string, true)` appends a comment to the last printed s-expression.
- `writer.flush()` flushes any buffering and should be called before any direct access to the underlying `std::ostream`. It does not close unclosed parentheses.
- The destructor calls `flush` but also closes all unclosed parentheses.

Example: A minimal functional backend

At its most basic, there are three steps we need to accomplish for a minimal functional backend. First, we need to convert our design into `FunctionalIR`. This is most easily done by calling the `Functional::IR::from_module()` static method with our top-level module, or iterating over and converting each of the modules in our design. Second, we need to handle each of the `Functional::Nodes` in our design. Iterating over the `Functional::IR` includes reading the module inputs and current state, but not writing the results. So our final step is to handle the outputs and next state.

In order to add an output command to Yosys, we implement the `Yosys::Backend` class and provide an instance of it:

Listing 4.8: Example source code for a minimal functional backend, `dummy.cc`

```
#include "kernel/functional.h"
#include "kernel/yosys.h"

USING_YOSYS_NAMESPACE
PRIVATE_NAMESPACE_BEGIN

struct FunctionalDummyBackend : public Backend {
    FunctionalDummyBackend() : Backend("functional_dummy", "dump generated_
```

(continues on next page)

(continued from previous page)

```

↪Functional IR") {}
    void execute(std::ostream *&f, std::string filename, std::vector<std::string>
↪args, RTLIL::Design *design) override
    {
        // backend pass boiler plate
        log_header(design, "Executing dummy functional backend.\n");

        size_t argidx = 1;
        extra_args(f, filename, args, argidx, design);

        for (auto module : design->selected_modules())
        {
            log("Processing module `%s`.\n", module->name.c_str());

            // convert module to FunctionalIR
            auto ir = Functional::IR::from_module(module);
            *f << "module " << module->name.c_str() << "\n";

            // write node functions
            for (auto node : ir)
                *f << "  assign " << id2cstr(node.name())
                    << " = " << node.to_string() << "\n";
            *f << "\n";

            // write outputs and next state
            for (auto output : ir.outputs())
                *f << " " << id2cstr(output->kind)
                    << " " << id2cstr(output->name)
                    << " = " << id2cstr(output->value().name()) << "\n";
            for (auto state : ir.states())
                *f << " " << id2cstr(state->kind)
                    << " " << id2cstr(state->name)
                    << " = " << id2cstr(state->next_value().name()) << "\n";
↪";
        }
    }
} FunctionalDummyBackend;

PRIVATE_NAMESPACE_END

```

Because we are using the Backend class, our "functional_dummy" is registered as the `write_functional_dummy` command. The `execute` method is the part that runs when the user calls the command, handling any options, preparing the output file for writing, and iterating over selected modules in the design. Since we don't have any options here, we set `argidx = 1` and call the `extra_args()` method. This method will read the command arguments, raising an error if there are any unexpected ones. It will also assign the pointer `f` to the output file, or `stdout` if none is given.

Note: For more on adding new commands to Yosys and how they work, refer to [Writing extensions](#).

For this minimal example all we are doing is printing out each node. The `node.name()` method returns an `RTLIL::IdString`, which we convert for printing with `id2cstr()`. Then, to print the function of the node, we use `node.to_string()` which gives us a string of the form `function(args)`. The function part is the

result of `Functional::IR::fn_to_string(node.fn())`; while `args` is the zero or more arguments passed to the function, most commonly the name of another node. Behind the scenes, the `node.to_string()` method actually wraps `node.visit(visitor)` with a private visitor whose return type is `std::string`.

Finally we iterate over the module's outputs and states, using `Functional::IROutput::value()` and `Functional::IRState::next_value()` respectively in order to get the results of the transfer function.

Example: Adapting SMT-LIB backend for Rosette

This section will introduce the SMT-LIB functional backend (`write_functional_smt2`) and what changes are needed to work with another s-expression target, `Rosette` (`write_functional_rosette`).

Overview

Rosette is a solver-aided programming language that extends `Racket` with language constructs for program synthesis, verification, and more. To verify or synthesize code, Rosette compiles it to logical constraints solved with off-the-shelf `SMT` solvers.

—<https://emina.github.io/rosette/>

Rosette, being backed by SMT solvers and written with s-expressions, uses code very similar to the `write_functional_smt2` output. As a result, the SMT-LIB functional backend can be used as a starting point for implementing a Rosette backend.

Full code listings for the initial SMT-LIB backend and the converted Rosette backend are included in the Yosys source repository under `backends/functional` as `smtlib.cc` and `smtlib_rosette.cc` respectively. Note that the Rosette language is an extension of the Racket language; this guide tends to refer to Racket when talking about the underlying semantics/syntax of the language.

The major changes from the SMT-LIB backend are as follows:

- all of the `Smt` prefixes in names are replaced with `Smtr` to mean `smtlib_rosette`;
- syntax is adjusted for Racket;
- data structures for input/output/state are changed from using `declare-datatype` with statically typed fields, to using `struct` with no static typing;
- the transfer function also loses its static typing;
- sign/zero extension in Rosette use the output width instead of the number of extra bits, gaining static typing;
- the single scope is traded for a global scope with local scope for each struct;
- initial state is provided as a constant value instead of a set of assertions;
- and the `-provides` option is introduced to more easily use generated code within Rosette based applications.

Scope

Our first addition to the *minimal backend* above is that for both SMT-LIB and Rosette backends, we are now targetting real languages which bring with them their own sets of constraints with what we can use as identifiers. This is where the `Functional::Scope` class described above comes in; by using this class we can safely rename our identifiers in the generated output without worrying about collisions or illegal names/characters.

In the SMT-LIB version, the `SmtScope` class implements `Scope<int>`; provides a constructor that iterates over a list of reserved keywords, calling `reserve` on each; and defines the `is_character_legal` method to reject any characters which are not allowed in SMT-LIB variable names to then be replaced with underscores in the output. To use this scope we create an instance of it, and call the `Scope::unique_name()` method to generate a unique and legal name for each of our identifiers.

In the Rosette version we update the list of legal ascii characters in the `is_character_legal` method to only those allowed in Racket variable names.

Listing 4.9: diff of `Scope` class

```
-struct SmtScope : public Functional::Scope<int> {
-    SmtScope() {
+struct SmtScope : public Functional::Scope<int> {
+    SmtScope() {
+        for(const char **p = reserved_keywords; *p != nullptr; p++)
+            reserve(*p);
+    }
+    bool is_character_legal(char c, int index) override {
-        return isascii(c) && (isalpha(c) || (isdigit(c) && index > 0) || strchr(
-            ↪ "~!@%~&*_-+=<>./\"", c));
+        return isascii(c) && (isalpha(c) || (isdigit(c) && index > 0) || strchr(
+            ↪ "@$%~&*_-+=.", c));
+    }
+};
```

For the reserved keywords we trade the SMT-LIB specification for Racket to prevent parts of our design from accidentally being treated as Racket code. We also no longer need to reserve `pair`, `first`, and `second`. In *write_functional_smt2* these are used for combining the `(inputs, current_state)` and `(outputs, next_state)` into a single variable. Racket provides this functionality natively with `cons`, which we will see later.

Listing 4.10: diff of `reserved_keywords` list

```
const char *reserved_keywords[] = {
-    // reserved keywords from the smtlib spec
-    ...
+    // reserved keywords from the racket spec
+    ...

    // reserved for our own purposes
-    "pair", "Pair", "first", "second",
-    "inputs", "state",
+    "inputs", "state", "name",
    nullptr
};
```

Note: We skip over the actual list of reserved keywords from both the smtlib and racket specifications to save on space in this document.

Sort

Next up in *write_functional_smt2* we see the `Sort` class. This is a wrapper for the `Functional::Sort` class, providing the additional functionality of mapping variable declarations to s-expressions with the `to_sexpr()` method. The main change from `SmtSort` to `SmtrSort` is a syntactical one with signals represented as `bitvectors`, and memories as `lists` of signals.

Listing 4.11: diff of `Sort` wrapper

```
SEExpr to_sexpr() const {
    if(sort.is_memory()) {
-         return list("Array", list("_", "BitVec", sort.addr_width()),
+         return list("list", list("bitvector", sort.addr_width()), list(
+         ↪ "bitvector", sort.data_width()));
-         ↪ list("_", "BitVec", sort.data_width()));
    } else if(sort.is_signal()) {
-         return list("_", "BitVec", sort.width());
+         return list("bitvector", sort.width());
    } else {
        log_error("unknown sort");
    }
}
```

Struct

As we saw in the *minimal backend* above, the `Functional::IR` class tracks the set of inputs, the set of outputs, and the set of “state” variables. The SMT-LIB backend maps each of these sets into its own `SmtStruct`, with each variable getting a corresponding field in the struct and a specified *Sort*. *write_functional_smt2* then defines each of these structs as a new `datatype`, with each element being strongly-typed.

In Rosette, rather than defining new datatypes for our structs, we use the native `struct`. We also only declare each field by name because Racket provides less static typing. For ease of use, we provide the expected type for each field as comments.

Listing 4.12: diff of `write_definition` method

```
void write_definition(SEExprWriter &w) {
-         w.open(list("declare-datatype", name));
-         w.open(list());
-         w.open(list(name));
-         for(const auto &field : fields)
-             w << list(field.accessor, field.sort.to_sexpr());
-         w.close(3);
+         vector<SEExpr> field_list;
+         for(const auto &field : fields) {
+             field_list.emplace_back(field.name);
+         }
}
```

(continues on next page)

(continued from previous page)

```

+         w.push();
+         w.open(list("struct", name, field_list, "#:transparent"));
+         if (field_names.size()) {
+             for (const auto &field : fields) {
+                 auto bv_type = field.sort.to_sexpr();
+                 w.comment(field.name + " " + bv_type.to_string());
+             }
+         }
+         w.pop();
+     }

```

Each field is added to the `SmtStruct` with the `insert` method, which also reserves a unique name (or accessor) within the `Scope`. These accessors combine the struct name and field name and are globally unique, being used in the `access` method for reading values from the input/current state.

Listing 4.13: `Struct::access()` method

```

SEExpr access(SEExpr record, IdString name) {
    size_t i = field_names.at(name);
    return list(fields[i].accessor, std::move(record));
}

```

In Rosette, struct fields are accessed as `<struct_name>-<field_name>` so including the struct name in the field name would be redundant. For `write_functional_rosette` we instead choose to make field names unique only within the struct, while accessors are unique across the whole module. We thus modify the class constructor and `insert` method to support this; providing one scope that is local to the struct (`local_scope`) and one which is shared across the whole module (`global_scope`), leaving the `access` method unchanged.

Listing 4.14: diff of struct constructor

```

-     SmtStruct(std::string name, SmtScope &scope) : scope(scope), name(name) {}
-     void insert(IdString field_name, SmtSort sort) {
+     SmtrStruct(std::string name, SmtrScope &scope) : global_scope(scope), local_
↪scope(), name(name) {}
+     void insert(IdString field_name, SmtrSort sort) {
+         field_names(field_name);
-         auto accessor = scope.unique_name("\\\" + name + \"_\" + RTLIL::unescape_
↪id(field_name));
-         fields.emplace_back(Field{sort, accessor});
+         auto base_name = local_scope.unique_name(field_name);
+         auto accessor = name + \"-\" + base_name;
+         global_scope.reserve(accessor);
+         fields.emplace_back(Field{sort, accessor, base_name});
+     }

```

Finally, `SmtStruct` also provides a `write_value` template method which calls a provided function on each element in the struct. This is used later for assigning values to the output/next state pair. The only change here is to remove the check for zero-argument constructors since this is not necessary with Rosette `structs`.

Listing 4.15: diff of `write_value` method

```

template<typename Fn> void write_value(SEExprWriter &w, Fn fn) {
-     if(field_names.empty()) {

```

(continues on next page)

(continued from previous page)

```

-           // Zero-argument constructors in SMTLIB must not be called as
- functions.
-           w << name;
-       } else {
-           w.open(list(name));
-           for(auto field_name : field_names) {
-               w << fn(field_name);
-               w.comment(RTLIL::unescape_id(field_name), true);
-           }
-           w.close();
+       w.open(list(name));
+       for(auto field_name : field_names) {
+           w << fn(field_name);
+           w.comment(RTLIL::unescape_id(field_name), true);
+       }
+       w.close();
+   }

```

PrintVisitor

Remember in the *minimal backend* we converted nodes into strings for writing using the `node.to_string()` method, which wrapped `node.visit()` with a private visitor. We now want a custom visitor which can convert nodes into s-expressions. This is where the `PrintVisitor` comes in, implementing the abstract `Functional::AbstractVisitor` class with a return type of `SExpr`. For most functions, the Rosette output is very similar to the corresponding SMT-LIB function with minor adjustments for syntax.

Listing 4.16: portion of `Functional::AbstractVisitor` implementation diff showing similarities

```

SExpr logical_shift_left(Node, Node a, Node b) override { return list("bvshl",
- n(a), extend(n(b), b.width(), a.width())); }
SExpr logical_shift_right(Node, Node a, Node b) override { return list("bvlshr",
- n(a), extend(n(b), b.width(), a.width())); }
SExpr arithmetic_shift_right(Node, Node a, Node b) override { return list(
- "bvashr", n(a), extend(n(b), b.width(), a.width())); }
- SExpr mux(Node, Node a, Node b, Node s) override { return list("ite", to_
- bool(n(s)), n(b), n(a)); }
- SExpr constant(Node, RTLIL::Const const &value) override { return smt_
- const(value); }
- SExpr memory_read(Node, Node mem, Node addr) override { return list("select",
- n(mem), n(addr)); }
- SExpr memory_write(Node, Node mem, Node addr, Node data) override { return list(
- "store", n(mem), n(addr), n(data)); }
+ SExpr mux(Node, Node a, Node b, Node s) override { return list("if", to_
+ bool(n(s)), n(b), n(a)); }
+ SExpr constant(Node, RTLIL::Const const& value) override { return list("bv",
+ smt_const(value), value.size()); }
+ SExpr memory_read(Node, Node mem, Node addr) override { return list("list-ref-bv
+ ", n(mem), n(addr)); }
+ SExpr memory_write(Node, Node mem, Node addr, Node data) override { return list(
+ "list-set-bv", n(mem), n(addr), n(data)); }

```

However there are some differences in the two formats with regards to how booleans are handled, with Rosette providing built-in functions for conversion.

Listing 4.17: portion of `Functional::AbstractVisitor` implementation diff showing differences

```

SExpr from_bool(SExpr &&arg) {
-     return list("ite", std::move(arg), "#b1", "#b0");
+     return list("bool->bitvector", std::move(arg));
}
SExpr to_bool(SExpr &&arg) {
-     return list("=", std::move(arg), "#b1");
+     return list("bitvector->bool", std::move(arg));
}

```

Of note here is the rare instance of the Rosette implementation *gaining* static typing rather than losing it. Where `SMT_LIB` calls zero/sign extension with the number of extra bits needed (given by `out_width - a.width()`), Rosette instead specifies the type of the output (given by `list("bitvector", out_width)`).

Listing 4.18: zero/sign extension implementation diff

```

-     SExpr zero_extend(Node, Node a, int out_width) override { return list(list("_",
↪ "zero_extend", out_width - a.width()), n(a)); }
-     SExpr sign_extend(Node, Node a, int out_width) override { return list(list("_",
↪ "sign_extend", out_width - a.width()), n(a)); }
+     SExpr zero_extend(Node, Node a, int out_width) override { return list("zero-
↪ extend", n(a), list("bitvector", out_width)); }
+     SExpr sign_extend(Node, Node a, int out_width) override { return list("sign-
↪ extend", n(a), list("bitvector", out_width)); }

```

Note: Be sure to check the source code for the full list of differences here.

Module

With most of the supporting classes out of the way, we now reach our three main steps from the *minimal backend*. These are all handled by the `SmtModule` class, with the mapping from RTLIL module to `FunctionalIR` happening in the constructor. Each of the three `SmtStructs`; inputs, outputs, and state; are also created in the constructor, with each value in the corresponding lists in the IR being inserted.

Listing 4.19: `SmtModule` constructor

```

SmtModule(Module *module)
: ir(Functional::IR::from_module(module))
, scope()
, name(scope.unique_name(module->name))
, input_struct(scope.unique_name(module->name.str() + "_Inputs"), scope)
, output_struct(scope.unique_name(module->name.str() + "_Outputs"),
↪ scope)
, state_struct(scope.unique_name(module->name.str() + "_State"), scope)
{
    scope.reserve(name + "-initial");
}

```

(continues on next page)

(continued from previous page)

```

    for (auto input : ir.inputs())
        input_struct.insert(input->name, input->sort);
    for (auto output : ir.outputs())
        output_struct.insert(output->name, output->sort);
    for (auto state : ir.states())
        state_struct.insert(state->name, state->sort);
}

```

Since Racket uses the `-` to access struct fields, the `SmtModule` instead uses an underscore for the name of the initial state.

Listing 4.20: diff of Module constructor

```

-         scope.reserve(name + "-initial");
+         scope.reserve(name + "_initial");

```

The `write` method is then responsible for writing the FunctionalIR to the output file, formatted for the corresponding backend. `SmtModule::write()` breaks the output file down into four parts: defining the three structs, declaring the `pair` datatype, defining the transfer function `(inputs, current_state) -> (outputs, next_state)` with `write_eval`, and declaring the initial state with `write_initial`. The only change for the `SmtModule` is that the `pair` declaration isn't needed.

Listing 4.21: diff of Module::write() method

```

void write(std::ostream &out)
{
    SExprWriter w(out);

    input_struct.write_definition(w);
    output_struct.write_definition(w);
    state_struct.write_definition(w);

-         w << list("declare-datatypes",
-                 list(list("Pair", 2)),
-                 list(list("par", list("X", "Y"), list(list("pair", list("first",
- ↪ "X"), list("second", "Y")))))));
-
    write_eval(w);
    write_initial(w);
}

```

The `write_eval` method is where the FunctionalIR nodes, outputs, and next state are handled. Just as with the *minimal backend*, we iterate over the nodes with `for(auto n : ir)`, and then use the `Struct::write_value()` method for the `output_struct` and `state_struct` to iterate over the outputs and next state respectively.

Listing 4.22: iterating over FunctionalIR nodes in SmtModule::write_eval()

```

for(auto n : ir)
    if(!inlined(n)) {
        w.open(list("let", list(list(node_to_sexpr(n), n.
- ↪ visit(visitor))))), false);

```

(continues on next page)

(continued from previous page)

```

                                w.comment(SmtSort(n.sort()).to_sexpr().to_string(),
↪true);
                                }

```

The main differences between our two backends here are syntactical. First we change the **define-fun** for the Racket style **define** which drops the explicitly typed inputs/outputs. And then we change the final result from a **pair** to the native **cons** which acts in much the same way, returning both the **outputs** and the **next_state** in a single variable.

Listing 4.23: diff of `Module::write_eval()` transfer function declaration

```

-      w.open(list("define-fun", name,
-                list(list("inputs", input_struct.name),
-                list("state", state_struct.name)),
-                list("Pair", output_struct.name, state_struct.name)));
+      w.open(list("define", list(name, "inputs", "state")));

```

Listing 4.24: diff of output/next state handling `Module::write_eval()`

```

-      w.open(list("pair"));
+      w.open(list("cons"));
      output_struct.write_value(w, [&](IdString name) { return node_to_
↪sexpr(ir.output(name).value()); });
      state_struct.write_value(w, [&](IdString name) { return node_to_
↪sexpr(ir.state(name).next_value()); });
      w.pop();

```

For the `write_initial` method, the SMT-LIB backend uses **declare-const** and **asserts** which must always hold true. For Rosette we instead define the initial state as any other variable that can be used by external code. This variable, `[name]_initial`, can then be used in the `[name]` function call; allowing the Rosette code to be used in the generation of the **next_state**, whereas the SMT-LIB code can only verify that a given **next_state** is correct.

Listing 4.25: diff of `Module::write_initial()` method

```

void write_initial(SExprWriter &w)
{
-      std::string initial = name + "-initial";
-      w << list("declare-const", initial, state_struct.name);
+      w.push();
+      auto initial = name + "_initial";
+      w.open(list("define", initial));
+      w.open(list(state_struct.name));
+      for (auto state : ir.states()) {
-          if(state->sort.is_signal())
-              w << list("assert", list("=", state_struct.
↪access(initial, state->name), smt_const(state->initial_value_signal())));
-          else if(state->sort.is_memory()) {
+              if (state->sort.is_signal())
+                  w << list("bv", smt_const(state->initial_value_

```

(continues on next page)

(continued from previous page)

```

↪signal(), state->sort.width());
+           else if (state->sort.is_memory()) {
+               const auto &contents = state->initial_value_memory();
+               w.open(list("list"));
+               for(int i = 0; i < 1<<state->sort.addr_width(); i++) {
-                   auto addr = smt_const(RTLIL::Const(i, state->
↪sort.addr_width()));
-                   w << list("assert", list("=", list("select",
↪state_struct.access(initial, state->name), addr), smt_const(contents[i])));
+                   w << list("bv", smt_const(contents[i]), state->
↪sort.data_width());
+               }
+               w.close();
+           }
+       }
+       w.pop();
+   }

```

Backend

The final part is the **Backend** itself, with much of the same boiler plate as the *minimal backend*. The main difference is that we use the *Module* to perform the actual processing.

Listing 4.26: The FunctionalSmtBackend

```

struct FunctionalSmtBackend : public Backend {
    FunctionalSmtBackend() : Backend("functional_smt2", "Generate SMT-LIB from
↪Functional IR") {}

    void help() override { log("\nFunctional SMT Backend.\n\n"); }

    void execute(std::ostream &f, std::string filename, std::vector<std::string>
↪args, RTLIL::Design *design) override
    {
        log_header(design, "Executing Functional SMT Backend.\n");

        size_t argidx = 1;
        extra_args(f, filename, args, argidx, design);

        for (auto module : design->selected_modules()) {
            log("Processing module `%s`.\n", module->name.c_str());
            SmtModule smt(module);
            smt.write(*f);
        }
    }
} FunctionalSmtBackend;

```

There are two additions here for Rosette. The first is that the output file needs to start with the `#lang` definition which tells the compiler/interpreter that we want to use the Rosette language module. The second is that the `write_functional_rosette` command takes an optional argument, `-provides`. If this argument is given, then the output file gets an additional line declaring that everything in the file should be exported

for use; allowing the file to be treated as a Racket package with structs and mapping function available for use externally.

Listing 4.27: relevant portion of diff of `Backend::execute()` method

```
+      *f << "#lang rosette/safe\n";
+      if (provides) {
+          *f << "(provide (all-defined-out))\n";
+      }
```

4.3.4 Contributing to Yosys

Note: For information on making a pull request on github, refer to our [CONTRIBUTING.md](#) file.

Coding Style

Formatting of code

- Yosys code is using tabs for indentation. Tabs are 8 characters.
- A continuation of a statement in the following line is indented by two additional tabs.
- Lines are as long as you want them to be. A good rule of thumb is to break lines at about column 150.
- Opening braces can be put on the same or next line as the statement opening the block (if, switch, for, while, do). Put the opening brace on its own line for larger blocks, especially blocks that contains blank lines.
- Otherwise stick to the [Linux Kernel Coding Style](#).

C++ Language

Yosys is written in C++17.

In general Yosys uses `int` instead of `size_t`. To avoid compiler warnings for implicit type casts, always use `GetSize(foobar)` instead of `foobar.size()`. (`GetSize()` is defined in `kernel/yosys.h`)

Use range-based for loops whenever applicable.

4.3.5 Testing Yosys

Todo: more about the included test suite and how to add tests

Automatic testing

For up to date information, including OS versions, refer to [the git actions page](#).

Todo: are unit tests currently working

4.4 Techmap by example

As a quick recap, the *techmap* command replaces cells in the design with implementations given as Verilog code (called “map files”). It can replace Yosys’ internal cell types (such as *\$or*) as well as user-defined cell types.

- Verilog parameters are used extensively to customize the internal cell types.
- Additional special parameters are used by techmap to communicate meta-data to the map files.
- Special wires are used to instruct techmap how to handle a module in the map file.
- Generate blocks and recursion are powerful tools for writing map files.

Code examples used in this document are included in the Yosys code base under [docs/source/code_examples/techmap](#).

4.4.1 Mapping OR3X1

Todo: add/expand supporting text

Note: This is a simple example for demonstration only. Techmap shouldn’t be used to implement basic logic optimization.

Listing 4.28: red_or3x1_map.v

```
module \${reduce_or} (A, Y);

    parameter A_SIGNED = 0;
    parameter A_WIDTH = 0;
    parameter Y_WIDTH = 0;

    input [A_WIDTH-1:0] A;
    output [Y_WIDTH-1:0] Y;

    function integer min;
        input integer a, b;
        begin
            if (a < b)
                min = a;
            else
                min = b;
        end
    endfunction

    Y = A;
```

(continues on next page)

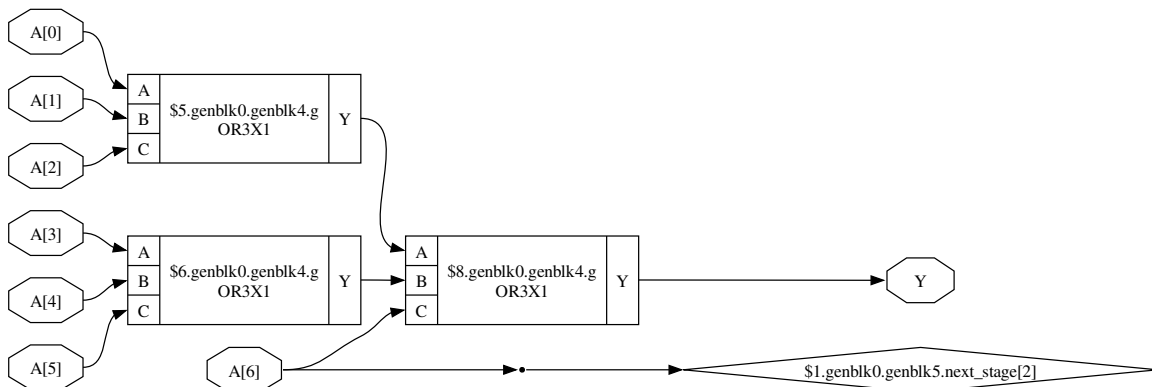
(continued from previous page)

```

    end
endfunction

genvar i;
generate begin
    if (A_WIDTH == 0) begin
        assign Y = 0;
    end
    if (A_WIDTH == 1) begin
        assign Y = A;
    end
    if (A_WIDTH == 2) begin
        wire ybuf;
        OR3X1 g (.A(A[0]), .B(A[1]), .C(1'b0), .Y(ybuf));
        assign Y = ybuf;
    end
    if (A_WIDTH == 3) begin
        wire ybuf;
        OR3X1 g (.A(A[0]), .B(A[1]), .C(A[2]), .Y(ybuf));
        assign Y = ybuf;
    end
    if (A_WIDTH > 3) begin
        localparam next_stage_sz = (A_WIDTH+2) / 3;
        wire [next_stage_sz-1:0] next_stage;
        for (i = 0; i < next_stage_sz; i = i+1) begin
            localparam bits = min(A_WIDTH - 3*i, 3);
            assign next_stage[i] = |A[3*i+: bits];
        end
        assign Y = |next_stage;
    end
end endgenerate
endmodule

```



Listing 4.29: red_or3x1_test.ys

```
read_verilog red_or3x1_test.v
hierarchy -check -top test

techmap -map red_or3x1_map.v;;

splitnets -ports
show -prefix red_or3x1 -format dot -notitle -lib red_or3x1_cells.v
```

Listing 4.30: red_or3x1_test.v

```
module test (A, Y);
    input [6:0] A;
    output Y;
    assign Y = |A;
endmodule
```

4.4.2 Conditional techmap

- In some cases only cells with certain properties should be substituted.
- The special wire `_TECHMAP_FAIL_` can be used to disable a module in the map file for a certain set of parameters.
- The wire `_TECHMAP_FAIL_` must be set to a constant value. If it is non-zero then the module is disabled for this set of parameters.
- Example use-cases:
 - coarse-grain cell types that only operate on certain bit widths
 - memory resources for different memory geometries (width, depth, ports, etc.)

Example:

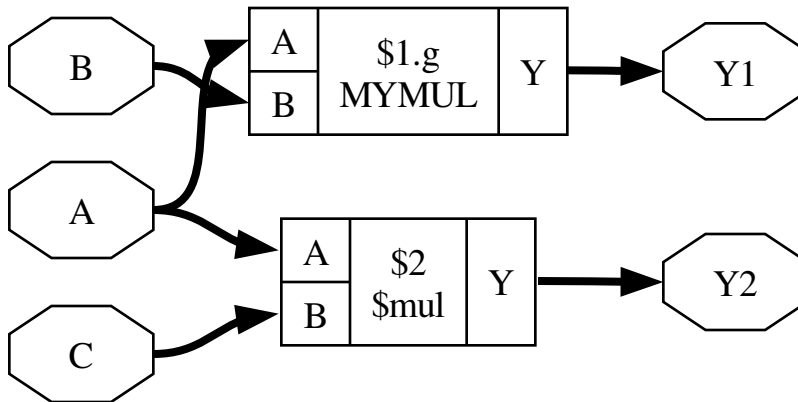
Listing 4.31: sym_mul_map.v

```
module \symmul (A, B, Y);
    parameter A_SIGNED = 0;
    parameter B_SIGNED = 0;
    parameter A_WIDTH = 1;
    parameter B_WIDTH = 1;
    parameter Y_WIDTH = 1;

    input [A_WIDTH-1:0] A;
    input [B_WIDTH-1:0] B;
    output [Y_WIDTH-1:0] Y;

    wire _TECHMAP_FAIL_ = A_WIDTH != B_WIDTH || B_WIDTH != Y_WIDTH;

    MYMUL #( .WIDTH(Y_WIDTH) ) g ( .A(A), .B(B), .Y(Y) );
endmodule
```



Listing 4.32: sym_mul_test.v

```

module test(A, B, C, Y1, Y2);
  input  [7:0] A, B, C;
  output [7:0] Y1 = A * B;
  output [15:0] Y2 = A * C;
endmodule

```

Listing 4.33: sym_mul_test.ys

```

read_verilog sym_mul_test.v
hierarchy -check -top test

techmap -map sym_mul_map.v;;

show -prefix sym_mul -format dot -notitle -lib sym_mul_cells.v

```

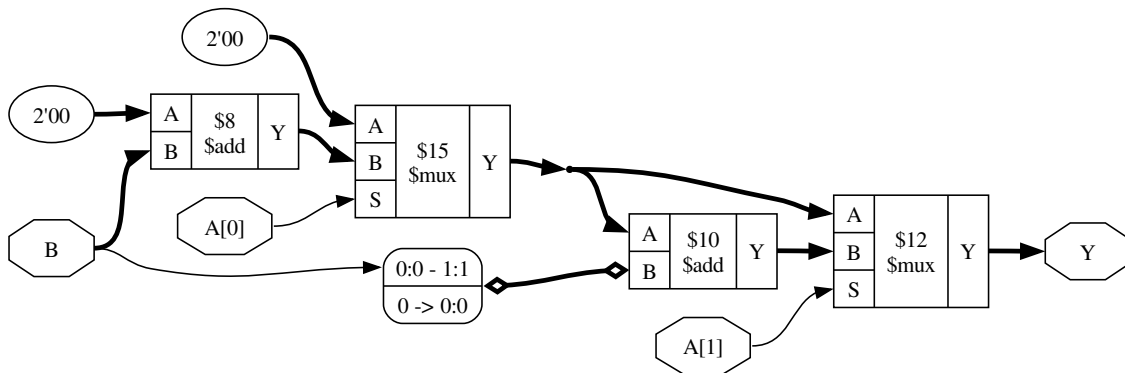
4.4.3 Scripting in map modules

- The special wires `_TECHMAP_DO_*` can be used to run Yosys scripts in the context of the replacement module.
- The wire that comes first in alphabetical order is interpreted as string (must be connected to constants) that is executed as script. Then the wire is removed. Repeat.
- You can even call `techmap` recursively!
- Example use-cases:
 - Using always blocks in map module: call `proc`
 - Perform expensive optimizations (such as `freduce`) on cells where this is known to work well.
 - Interacting with custom commands.

Note: PROTIP:

Commands such as `shell`, `show -pause`, and `dump` can be used in the `_TECHMAP_DO_*` scripts for debugging map modules.

Example:



Listing 4.34: mymul_map.v

```
module MYMUL(A, B, Y);
  parameter WIDTH = 1;
  input [WIDTH-1:0] A, B;
  output reg [WIDTH-1:0] Y;

  wire [1023:0] _TECHMAP_DO_ = "proc; clean";

  integer i;
  always @* begin
    Y = 0;
    for (i = 0; i < WIDTH; i=i+1)
      if (A[i])
        Y = Y + (B << i);
  end
endmodule
```

Listing 4.35: mymul_test.v

```
module test(A, B, Y);
  input [1:0] A, B;
  output [1:0] Y = A * B;
endmodule
```

Listing 4.36: mymul_test.js

```
read_verilog mymul_test.v
hierarchy -check -top test
```

(continues on next page)

(continued from previous page)

```

techmap -map sym_mul_map.v \
        -map mymul_map.v;;

rename test test_mapped
read_verilog mymul_test.v
miter -equiv test test_mapped miter
flatten miter

sat -verify -prove trigger 0 miter

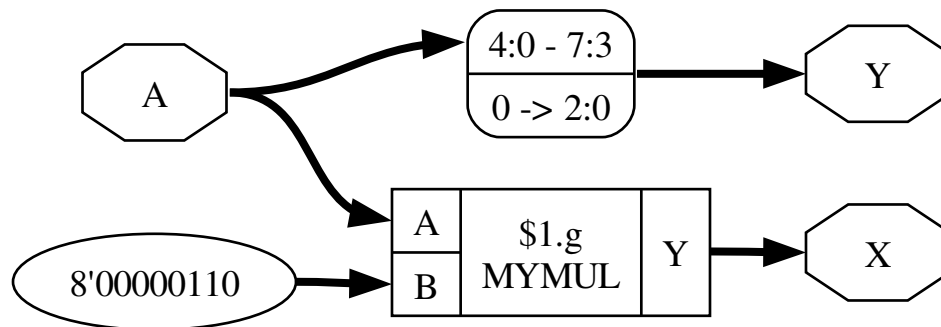
splitnets -ports test_mapped/A
show -prefix mymul -format dot -notitle test_mapped

```

4.4.4 Handling constant inputs

- The special parameters `_TECHMAP_CONSTMSK_<port-name>` and `_TECHMAP_CONSTVAL_<port-name>` can be used to handle constant input values to cells.
- The former contains 1-bits for all constant input bits on the port.
- The latter contains the constant bits or undef (x) for non-constant bits.
- Example use-cases:
 - Converting arithmetic (for example multiply to shift).
 - Identify constant addresses or enable bits in memory interfaces.

Example:



Listing 4.37: mulshift_map.v

```

module MYMUL(A, B, Y);
    parameter WIDTH = 1;
    input [WIDTH-1:0] A, B;

```

(continues on next page)

(continued from previous page)

```
output reg [WIDTH-1:0] Y;

parameter _TECHMAP_CONSTVAL_A_ = WIDTH'bx;
parameter _TECHMAP_CONSTVAL_B_ = WIDTH'bx;

reg _TECHMAP_FAIL_;
wire [1023:0] _TECHMAP_DO_ = "proc; clean";

integer i;
always @* begin
    _TECHMAP_FAIL_ <= 1;
    for (i = 0; i < WIDTH; i=i+1) begin
        if (_TECHMAP_CONSTVAL_A_ === WIDTH'd1 << i) begin
            _TECHMAP_FAIL_ <= 0;
            Y <= B << i;
        end
        if (_TECHMAP_CONSTVAL_B_ === WIDTH'd1 << i) begin
            _TECHMAP_FAIL_ <= 0;
            Y <= A << i;
        end
    end
end
endmodule
```

Listing 4.38: mulshift_test.v

```
module test (A, X, Y);
input [7:0] A;
output [7:0] X = A * 8'd 6;
output [7:0] Y = A * 8'd 8;
endmodule
```


Listing 4.40: addshift_map.v

```
module \${add} (A, B, Y);
  parameter A_SIGNED = 0;
  parameter B_SIGNED = 0;
  parameter A_WIDTH = 1;
  parameter B_WIDTH = 1;
  parameter Y_WIDTH = 1;

  input [A_WIDTH-1:0] A;
  input [B_WIDTH-1:0] B;
  output [Y_WIDTH-1:0] Y;

  parameter _TECHMAP_BITS_CONNMAP_ = 0;
  parameter _TECHMAP_CONNMAP_A_ = 0;
  parameter _TECHMAP_CONNMAP_B_ = 0;

  wire _TECHMAP_FAIL_ = A_WIDTH != B_WIDTH || B_WIDTH < Y_WIDTH ||
    _TECHMAP_CONNMAP_A_ != _TECHMAP_CONNMAP_B_;

  assign Y = A << 1;
endmodule
```

Listing 4.41: addshift_test.v

```
module test (A, B, X, Y);
  input [7:0] A, B;
  output [7:0] X = A + B;
  output [7:0] Y = A + A;
endmodule
```

Listing 4.42: addshift_test.js

```
read_verilog addshift_test.v
hierarchy -check -top test

techmap -map addshift_map.v;;

show -prefix addshift -format dot -notitle
```

4.4.6 Notes on using techmap

- Don't use positional cell parameters in map modules.
- You can use the `$_-` prefix for internal cell types to avoid collisions with the user-namespace. But always use two underscores or the internal consistency checker will trigger on these cells.
- Techmap has two major use cases:
 - Creating good logic-level representation of arithmetic functions. This also means using dedicated hardware resources such as half- and full-adder cells in ASICs or dedicated carry logic in FPGAs.
 - Mapping of coarse-grain resources such as block memory or DSP cells.

4.5 Notes on Verilog support in Yosys

Todo: how much of this is specific to the `read_verilog` and should be in *The Verilog and AST frontends*?

4.5.1 Unsupported Verilog-2005 Features

The following Verilog-2005 features are not supported by Yosys and there are currently no plans to add support for them:

- **Non-synthesizable language features as defined in**
IEC 62142(E):2005 / IEEE Std. 1364.1(E):2002
- The `tri`, `triand` and `trior` net types
- The `config` and `disable` keywords and library map files

4.5.2 Verilog Attributes and non-standard features

- The `full_case` attribute on case statements is supported (also the non-standard `// synopsys full_case` directive)
- The `parallel_case` attribute on case statements is supported (also the non-standard `// synopsys parallel_case` directive)
- The `// synopsys translate_off` and `// synopsys translate_on` directives are also supported (but the use of ``ifdef .. `endif`` is strongly recommended instead).
- The `nomem2reg` attribute on modules or arrays prohibits the automatic early conversion of arrays to separate registers. This is potentially dangerous. Usually the front-end has good reasons for converting an array to a list of registers. Prohibiting this step will likely result in incorrect synthesis results.
- The `mem2reg` attribute on modules or arrays forces the early conversion of arrays to separate registers.
- The `nomeminit` attribute on modules or arrays prohibits the creation of initialized memories. This effectively puts `mem2reg` on all memories that are written to in an `initial` block and are not ROMs.
- The `nolatches` attribute on modules or always-blocks prohibits the generation of logic-loops for latches. Instead all not explicitly assigned values default to x-bits. This does not affect clocked storage elements such as flip-flops.
- The `nosync` attribute on registers prohibits the generation of a storage element. The register itself will always have all bits set to 'x' (undefined). The variable may only be used as blocking assigned temporary variable within an always block. This is mostly used internally by Yosys to synthesize Verilog functions and access arrays.
- The `nowrshmsk` attribute on a register prohibits the generation of shift-and-mask type circuits for writing to bit slices of that register.
- The `onehot` attribute on wires mark them as one-hot state register. This is used for example for memory port sharing and set by the `fsm_map` pass.
- The `blackbox` attribute on modules is used to mark empty stub modules that have the same ports as the real thing but do not contain information on the internal configuration. This modules are only used by the synthesis passes to identify input and output ports of cells. The Verilog backend also does not output blackbox modules on default. `read_verilog`, unless called with `-noblackbox` will automatically set the blackbox attribute on any empty module it reads.

- The `noblackbox` attribute set on an empty module prevents `read_verilog` from automatically setting the blackbox attribute on the module.
- The `whitebox` attribute on modules triggers the same behavior as `blackbox`, but is for whitebox modules, i.e. library modules that contain a behavioral model of the cell type.
- The `lib_whitebox` attribute overwrites `whitebox` when `read_verilog` is run in `-lib` mode. Otherwise it's automatically removed.
- The `dynports` attribute is used by the Verilog front-end to mark modules that have ports with a width that depends on a parameter.
- The `hdlname` attribute is used by some passes to document the original (HDL) name of a module when renaming a module. It should contain a single name, or, when describing a hierarchical name in a flattened design, multiple names separated by a single space character.
- The `keep` attribute on cells and wires is used to mark objects that should never be removed by the optimizer. This is used for example for cells that have hidden connections that are not part of the netlist, such as IO pads. Setting the `keep` attribute on a module has the same effect as setting it on all instances of the module.
- The `keep_hierarchy` attribute on cells and modules keeps the `flatten` command from flattening the indicated cells and modules.
- The `gate_cost_equivalent` attribute on a module can be used to specify the estimated cost of the module as a number of basic gate instances. See the help message of command `keep_hierarchy` which interprets this attribute.
- The `init` attribute on wires is set by the frontend when a register is initialized “FPGA-style” with `reg foo = val`. It can be used during synthesis to add the necessary reset logic.
- The `top` attribute on a module marks this module as the top of the design hierarchy. The `hierarchy` command sets this attribute when called with `-top`. Other commands, such as `flatten` and various backends use this attribute to determine the top module.
- The `src` attribute is set on cells and wires created by the string `<hdl-file-name>:<line-number>` by the HDL front-end and is then carried through the synthesis. When entities are combined, a new `|`-separated string is created that contains all the strings from the original entities.
- The `defaultvalue` attribute is used to store default values for module inputs. The attribute is attached to the input wire by the HDL front-end when the input is declared with a default value.
- The `parameter` and `localparam` attributes are used to mark wires that represent module parameters or localparams (when the HDL front-end is run in `-pwires` mode).
- Wires marked with the `hierconn` attribute are connected to wires with the same name (format `cell_name.identifier`) when they are imported from sub-modules by `flatten`.
- The `clkbuf_driver` attribute can be set on an output port of a blackbox module to mark it as a clock buffer output, and thus prevent `clkbufmap` from inserting another clock buffer on a net driven by such output.
- The `clkbuf_sink` attribute can be set on an input port of a module to request clock buffer insertion by the `clkbufmap` pass.
- The `clkbuf_inv` attribute can be set on an output port of a module with the value set to the name of an input port of that module. When the `clkbufmap` would otherwise insert a clock buffer on this output, it will instead try inserting the clock buffer on the input port (this is used to implement clock inverter cells that clock buffer insertion will “see through”).
- The `clkbuf_inhibit` is the default attribute to set on a wire to prevent automatic clock buffer insertion by `clkbufmap`. This behaviour can be overridden by providing a custom selection to `clkbufmap`.

- The `invertible_pin` attribute can be set on a port to mark it as invertible via a cell parameter. The name of the inversion parameter is specified as the value of this attribute. The value of the inversion parameter must be of the same width as the port, with 1 indicating an inverted bit and 0 indicating a non-inverted bit.
- The `iopad_external_pin` attribute on a blackbox module's port marks it as the external-facing pin of an I/O pad, and prevents `iopadmap` from inserting another pad cell on it.
- The module attribute `abc9_lut` is an integer attribute indicating to `abc9` that this module describes a LUT with an area cost of this value, and propagation delays described using `specify` statements.
- The module attribute `abc9_box` is a boolean specifying a black/white-box definition, with propagation delays described using `specify` statements, for use by `abc9`.
- The port attribute `abc9_carry` marks the carry-in (if an input port) and carry-out (if output port) ports of a box. This information is necessary for `abc9` to preserve the integrity of carry-chains. Specifying this attribute onto a bus port will affect only its most significant bit.
- The module attribute `abc9_flop` is a boolean marking the module as a flip-flop. This allows `abc9` to analyse its contents in order to perform sequential synthesis.
- The frontend sets attributes `always_comb`, `always_latch` and `always_ff` on processes derived from SystemVerilog style always blocks according to the type of the always. These are checked for correctness in `proc_dlatch`.
- The cell attribute `wildcard_port_conns` represents wildcard port connections (SystemVerilog `.*`). These are resolved to concrete connections to matching wires in `hierarchy`.
- In addition to the `(* ... *)` attribute syntax, Yosys supports the non-standard `{* ... *}` attribute syntax to set default attributes for everything that comes after the `{* ... *}` statement. (Reset by adding an empty `{* *}` statement.)
- In module parameter and port declarations, and cell port and parameter lists, a trailing comma is ignored. This simplifies writing Verilog code generators a bit in some cases.
- Modules can be declared with `module mod_name(...);` (with three dots instead of a list of module ports). With this syntax it is sufficient to simply declare a module port as 'input' or 'output' in the module body.
- When defining a macro with `\`define`, all text between triple double quotes is interpreted as macro body, even if it contains unescaped newlines. The triple double quotes are removed from the macro body. For example:

```
`define MY_MACRO(a, b) """
    assign a = 23;
    assign b = 42;
    """
```

- The attribute `via_celltype` can be used to implement a Verilog task or function by instantiating the specified cell type. The value is the name of the cell type to use. For functions the name of the output port can be specified by appending it to the cell type separated by a whitespace. The body of the task or function is unused in this case and can be used to specify a behavioral model of the cell type for simulation. For example:

```
module my_add3(A, B, C, Y);
    parameter WIDTH = 8;
    input  [WIDTH-1:0] A, B, C;
    output [WIDTH-1:0] Y;
    ...
```

(continues on next page)

(continued from previous page)

```

endmodule

module top;
    ...
    (* via_celltype = "my_add3 Y" *)
    (* via_celltype_defparam_WIDTH = 32 *)
    function [31:0] add3;
        input [31:0] A, B, C;
        begin
            add3 = A + B + C;
        end
    endfunction
    ...
endmodule

```

- The `wiretype` attribute is added by the verilog parser for wires of a typedef'd type to indicate the type identifier.
- Various `enum_value_{value}` attributes are added to wires of an enumerated type to give a map of possible enum items to their values.
- The `enum_base_type` attribute is added to enum items to indicate which enum they belong to (enums – anonymous and otherwise – are automatically named with an auto-incrementing counter). Note that enums are currently not strongly typed.
- A limited subset of DPI-C functions is supported. The plugin mechanism (see `help plugin`) can be used to load .so files with implementations of DPI-C routines. As a non-standard extension it is possible to specify a plugin alias using the `<alias>`: syntax. For example:

```

module dpitest;
    import "DPI-C" function foo:round = real my_round (real);
    parameter real r = my_round(12.345);
endmodule

```

```
$ yosys -p 'plugin -a foo -i /lib/libm.so; read_verilog dpitest.v'
```

- Sized constants (the syntax `<size>'s?[bodh]<value>`) support constant expressions as `<size>`. If the expression is not a simple identifier, it must be put in parentheses. Examples: `WIDTH'd42`, `(4+2)'b101010`
- The system tasks `$finish`, `$stop` and `$display` are supported in initial blocks in an unconditional context (only if/case statements on expressions over parameters and constant values are allowed). The intended use for this is synthesis-time DRC.
- There is limited support for converting `specify .. endspecify` statements to special `$specify2`, `$specify3`, and `$specrule` cells, for use in blackboxes and whiteboxes. Use `read_verilog -specify` to enable this functionality. (By default these blocks are ignored.)
- The `reprocess_after` internal attribute is used by the Verilog frontend to mark cells with bindings which might depend on the specified instantiated module. Modules with such cells will be reprocessed during the *hierarchy* pass once the referenced module definition(s) become available.
- The `smtlib2_module` attribute can be set on a blackbox module to specify a formal model directly using SMT-LIB 2. For such a module, the `smtlib2_comb_expr` attribute can be used on output ports to define their value using an SMT-LIB 2 expression. For example:

```
(* blackbox *)
(* smtlib2_module *)
module submod(a, b);
  input [7:0] a;
  (* smtlib2_comb_expr = "(bvnot a)" *)
  output [7:0] b;
endmodule
```

4.5.3 Non-standard or SystemVerilog features for formal verification

- Support for `assert`, `assume`, `restrict`, and `cover` is enabled when `read_verilog` is called with `-formal`.
- The system task `$initstate` evaluates to 1 in the initial state and to 0 otherwise.
- The system function `$anyconst` evaluates to any constant value. This is equivalent to declaring a reg as `rand const`, but also works outside of checkers. (Yosys also supports `rand const` outside checkers.)
- The system function `$anyseq` evaluates to any value, possibly a different value in each cycle. This is equivalent to declaring a reg as `rand`, but also works outside of checkers. (Yosys also supports `rand` variables outside checkers.)
- The system functions `$allconst` and `$allseq` can be used to construct formal exist-forall problems. Assumptions only hold if the trace satisfies the assumption for all `$allconst/$allseq` values. For assertions and cover statements it is sufficient if just one `$allconst/$allseq` value triggers the property (similar to `$anyconst/$anyseq`).
- Wires/registers declared using the `anyconst/anyseq/allconst/allseq` attribute (for example `(* anyconst *) reg [7:0] foobar;`) will behave as if driven by a `$anyconst/$anyseq/$allconst/$allseq` function.
- The SystemVerilog tasks `$past`, `$stable`, `$rose` and `$fell` are supported in any clocked block.
- The syntax `@($global_clock)` can be used to create FFs that have no explicit clock input (`$ff` cells). The same can be achieved by using `@(posedge <netname>)` or `@(negedge <netname>)` when `<netname>` is marked with the `(* gclk *)` Verilog attribute.

4.5.4 Supported features from SystemVerilog

When `read_verilog` is called with `-sv`, it accepts some language features from SystemVerilog:

- The `assert` statement from SystemVerilog is supported in its most basic form. In module context: `assert property (<expression>);` and within an always block: `assert(<expression>;`. It is transformed to an `$assert` cell.
- The `assume`, `restrict`, and `cover` statements from SystemVerilog are also supported. The same limitations as with the `assert` statement apply.
- The keywords `always_comb`, `always_ff` and `always_latch`, `logic` and `bit` are supported.
- Declaring free variables with `rand` and `rand const` is supported.
- Checkers without a port list that do not need to be instantiated (but instead behave like a named block) are supported.
- SystemVerilog packages are supported. Once a SystemVerilog file is read into a design with `read_verilog`, all its packages are available to SystemVerilog files being read into the same design afterwards.

- **typedefs are supported (including inside packages)**
 - type casts are currently not supported
- **enums are supported (including inside packages)**
 - but are currently not strongly typed
- **packed structs and unions are supported**
 - arrays of packed structs/unions are currently not supported
 - structure literals are currently not supported
- **multidimensional arrays are supported**
 - array assignment of unpacked arrays is currently not supported
 - array literals are currently not supported
- SystemVerilog interfaces (SVIs) are supported. Modports for specifying whether ports are inputs or outputs are supported.
- Assignments within expressions are supported.

4.6 Hashing and associative data structures in Yosys

4.6.1 Container classes based on hashing

Yosys uses `dict<K, T>` and `pool<T>` as main container classes. `dict<K, T>` is essentially a replacement for `std::unordered_map<K, T>` and `pool<T>` is a replacement for `std::unordered_set<T>`. The main characteristics are:

- **`dict<K, T>` and `pool<T>` are about 2x faster than the std containers**
(though this claim hasn't been verified for over 10 years)
- **references to elements in a `dict<K, T>` or `pool<T>` are invalidated by**
insert and remove operations (similar to `std::vector<T>` on `push_back()`).
- **some iterators are invalidated by `erase()`. specifically, iterators**
that have not passed the erased element yet are invalidated. (`erase()` itself returns valid iterator to the next element.)
- **no iterators are invalidated by `insert()`. elements are inserted at**
`begin()`. i.e. only a new iterator that starts at `begin()` will see the inserted elements.
- **the method `.count(key, iterator)` is like `.count(key)` but only**
considers elements that can be reached via the iterator.
- **iterators can be compared. `it1 < it2` means that the position of `t2`**
can be reached via `t1` but not vice versa.
- **the method `.sort()` can be used to sort the elements in the container**
the container stays sorted until elements are added or removed.
- **`dict<K, T>` and `pool<T>` will have the same order of iteration across**
all compilers, standard libraries and architectures.

In addition to `dict<K, T>` and `pool<T>` there is also an `idict<K>` that creates a bijective map from `K` to the integers. For example:


```

idict<string, 42> si;
log("%d\n", si("hello"));      // will print 42
log("%d\n", si("world"));      // will print 43
log("%d\n", si.at("world"));   // will print 43
log("%d\n", si.at("dummy"));   // will throw exception
log("%s\n", si[42].c_str());   // will print hello
log("%s\n", si[43].c_str());   // will print world
log("%s\n", si[44].c_str());   // will throw exception

```

It is not possible to remove elements from an idict.

Finally `mfp<K>` implements a merge-find set data structure (aka. disjoint-set or union-find) over the type `K` (“mfp” = merge-find-promote).

4.6.2 The hash function

The hash function generally used in Yosys is the XOR version of DJB2:

```
state = ((state << 5) + state) ^ value
```

This is an old-school hash designed to hash ASCII characters. Yosys doesn’t hash a lot of ASCII text, but it still happens to be a local optimum due to factors described later.

Hash function quality is multi-faceted and highly dependent on what is being hashed. Yosys isn’t concerned by any cryptographic qualities, instead the goal is minimizing total hashing collision risk given the data patterns within Yosys. In general, a good hash function typically folds values into a state accumulator with a mathematical function that is fast to compute and has some beneficial properties. One of these is the avalanche property, which demands that a small change such as flipping a bit or incrementing by one in the input produces a large, unpredictable change in the output. Additionally, the bit independence criterion states that any pair of output bits should change independently when any single input bit is inverted. These properties are important for avoiding hash collision on data patterns like the hash of a sequence not colliding with its permutation, not losing from the state the information added by hashing preceding elements, etc.

DJB2 lacks these properties. Instead, since Yosys hashes large numbers of data structures composed of incrementing integer IDs, Yosys abuses the predictability of DJB2 to get lower hash collisions, with regular nature of the hashes surviving through the interaction with the “modulo prime” operations in the associative data structures. For example, some most common objects in Yosys are interned `IdStrings` of incrementing indices or `SigBits` with bit offsets into wire (represented by its unique `IdString` name) as the typical case. This is what makes DJB2 a local optimum. Additionally, the ADD version of DJB2 (like above but with addition instead of XOR) is used to this end for some types, abandoning the general pattern of folding values into a state value.

4.6.3 Making a type hashable

Let’s first take a look at the external interface on a simplified level. Generally, to get the hash for `T obj`, you would call the utility function `run_hash<T>(const T& obj)`, corresponding to `hash_ops<T>::hash(obj)`, the default implementation of which uses `hash_ops<T>::hash_into(Hasher(), obj)`. `Hasher` is the class actually implementing the hash function, hiding its initialized internal state, and passing it out on `hash_t yield()` with perhaps some finalization steps.

`hash_ops<T>` is the star of the show. By default it pulls the `Hasher h` through a `Hasher T::hash_into(Hasher h)` method. That’s the method you have to implement to make a record (class or struct) type easily hashable with Yosys hashlib associative data structures.

`hash_ops<T>` is specialized for built-in types like `int` or `bool` and treats pointers the same as integers, so it doesn't dereference pointers. Since many RTLIL data structures like `RTLIL::Wire` carry their own unique index `Hasher::hash_t hashidx_;`, there are specializations for `hash_ops<Wire*>` and others in `kernel/hashlib.h` that actually dereference the pointers and call `hash_into` on the instances pointed to.

`hash_ops<T>` is also specialized for simple compound types like `std::pair<U>` by calling `hash_into` in sequence on its members. For flexible size containers like `std::vector<U>` the size of the container is hashed first. That is also how implementing hashing for a custom record data type should be - unless there is strong reason to do otherwise, call `h.eat(m)` on the `Hasher h` you have received for each member in sequence and `return h;`

The `hash_ops<T>::hash(obj)` method is not intended to be called when context of implementing the hashing for a record or other compound type. When writing it, you should connect it to `hash_ops<T>::hash_into(Hasher h)` as shown below. If you have a strong reason to do so, and you have to create a special implementation for top-level hashing, look at how `hash_ops<RTLIL::SigBit>::hash(...)` is implemented in `kernel/rtlil.h`.

4.6.4 Porting plugins from the legacy interface

Previously, the interface to implement hashing on custom types was just `unsigned int T::hash() const`. This meant hashes for members were computed independently and then ad-hoc combined with the hash function with some xorshift operations thrown in to mix bits together somewhat. A plugin can stay compatible with both versions prior and after the break by implementing both interfaces based on the existence and value of `YS_HASHING_VERSION`.

Listing 4.43: Example hash compatibility wrapper

```
#ifndef YS_HASHING_VERSION
unsigned int T::hash() const {
    return mkhash(a, b);
}
#elif YS_HASHING_VERSION == 1
Hasher T::hash_into(Hasher h) const {
    h.eat(a);
    h.eat(b);
    return h;
}
Hasher T::hash() const {
    Hasher h;
    h.eat(*this);
    return h;
}
#else
#error "Unsupported hashing interface"
#endif
```

Feel free to contact Yosys maintainers with related issues.

A PRIMER ON DIGITAL CIRCUIT SYNTHESIS

This chapter contains a short introduction to the basic principles of digital circuit synthesis.

5.1 Levels of abstraction

Digital circuits can be represented at different levels of abstraction. During the design process a circuit is usually first specified using a higher level abstraction. Implementation can then be understood as finding a functionally equivalent representation at a lower abstraction level. When this is done automatically using software, the term synthesis is used.

So synthesis is the automatic conversion of a high-level representation of a circuit to a functionally equivalent low-level representation of a circuit. Figure 5.1 lists the different levels of abstraction and how they relate to different kinds of synthesis.

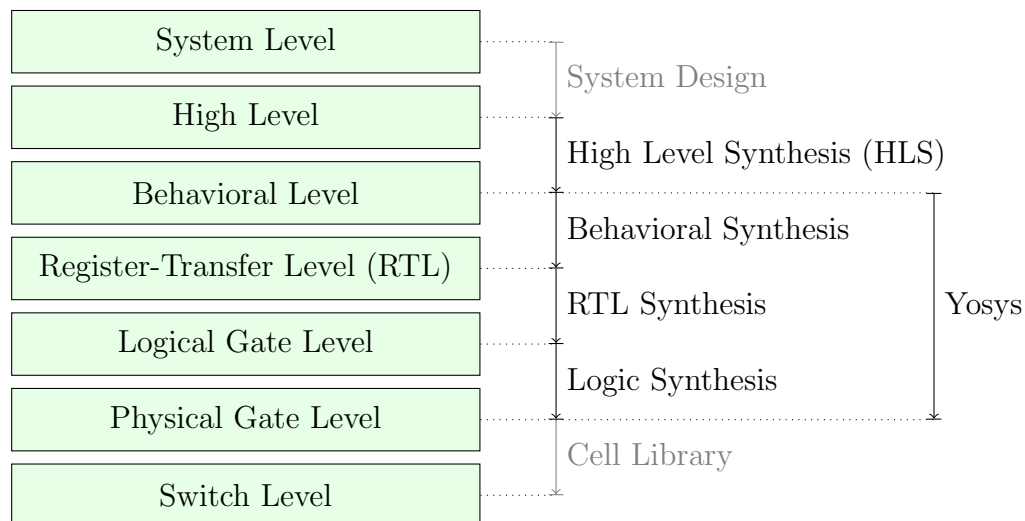


Fig. 5.1: Different levels of abstraction and synthesis.

Regardless of the way a lower level representation of a circuit is obtained (synthesis or manual design), the lower level representation is usually verified by comparing simulation results of the lower level and the higher level representation¹. Therefore even if no synthesis is used, there must still be a simulatable representation of the circuit in all levels to allow for verification of the design.

¹ In recent years formal equivalence checking also became an important verification method for validating RTL and lower abstraction representation of the design.

Note: The exact meaning of terminology such as “High-Level” is of course not fixed over time. For example the HDL “ABEL” was first introduced in 1985 as “A High-Level Design Language for Programmable Logic Devices” [LHBB85], but would not be considered a “High-Level Language” today.

5.1.1 System level

The System Level abstraction of a system only looks at its biggest building blocks like CPUs and computing cores. At this level the circuit is usually described using traditional programming languages like C/C++ or Matlab. Sometimes special software libraries are used that are aimed at simulation circuits on the system level, such as SystemC.

Usually no synthesis tools are used to automatically transform a system level representation of a circuit to a lower-level representation. But system level design tools exist that can be used to connect system level building blocks.

The IEEE 1685-2009 standard defines the IP-XACT file format that can be used to represent designs on the system level and building blocks that can be used in such system level designs. [A+10]

5.1.2 High level

The high-level abstraction of a system (sometimes referred to as algorithmic level) is also often represented using traditional programming languages, but with a reduced feature set. For example when representing a design at the high level abstraction in C, pointers can only be used to mimic concepts that can be found in hardware, such as memory interfaces. Full featured dynamic memory management is not allowed as it has no corresponding concept in digital circuits.

Tools exist to synthesize high level code (usually in the form of C/C++/SystemC code with additional metadata) to behavioural HDL code (usually in the form of Verilog or VHDL code). Aside from the many commercial tools for high level synthesis there are also a number of FOSS tools for high level synthesis .

5.1.3 Behavioural level

At the behavioural abstraction level a language aimed at hardware description such as Verilog or VHDL is used to describe the circuit, but so-called behavioural modelling is used in at least part of the circuit description. In behavioural modelling there must be a language feature that allows for imperative programming to be used to describe data paths and registers. This is the `always`-block in Verilog and the `process`-block in VHDL.

In behavioural modelling, code fragments are provided together with a sensitivity list; a list of signals and conditions. In simulation, the code fragment is executed whenever a signal in the sensitivity list changes its value or a condition in the sensitivity list is triggered. A synthesis tool must be able to transfer this representation into an appropriate datapath followed by the appropriate types of register.

For example consider the following Verilog code fragment:

```
1 always @(posedge clk)
2     y <= a + b;
```

In simulation the statement `y <= a + b` is executed whenever a positive edge on the signal `clk` is detected. The synthesis result however will contain an adder that calculates the sum `a + b` all the time, followed by a d-type flip-flop with the adder output on its D-input and the signal `y` on its Q-output.

Usually the imperative code fragments used in behavioural modelling can contain statements for conditional execution (`if`- and `case`-statements in Verilog) as well as loops, as long as those loops can be completely unrolled.

Interestingly there seems to be no other FOSS Tool that is capable of performing Verilog or VHDL behavioural syntheses besides Yosys.

5.1.4 Register-Transfer Level (RTL)

On the Register-Transfer Level the design is represented by combinatorial data paths and registers (usually d-type flip flops). The following Verilog code fragment is equivalent to the previous Verilog example, but is in RTL representation:

```

1 assign tmp = a + b;           // combinatorial data path
2
3 always @(posedge clk)        // register
4     y <= tmp;
```

A design in RTL representation is usually stored using HDLs like Verilog and VHDL. But only a very limited subset of features is used, namely minimalistic always-blocks (Verilog) or process-blocks (VHDL) that model the register type used and unconditional assignments for the datapath logic. The use of HDLs on this level simplifies simulation as no additional tools are required to simulate a design in RTL representation.

Many optimizations and analyses can be performed best at the RTL level. Examples include FSM detection and optimization, identification of memories or other larger building blocks and identification of shareable resources.

Note that RTL is the first abstraction level in which the circuit is represented as a graph of circuit elements (registers and combinatorial cells) and signals. Such a graph, when encoded as list of cells and connections, is called a netlist.

RTL synthesis is easy as each circuit node element in the netlist can simply be replaced with an equivalent gate-level circuit. However, usually the term RTL synthesis does not only refer to synthesizing an RTL netlist to a gate level netlist but also to performing a number of highly sophisticated optimizations within the RTL representation, such as the examples listed above.

A number of FOSS tools exist that can perform isolated tasks within the domain of RTL synthesis steps. But there seems to be no FOSS tool that covers a wide range of RTL synthesis operations.

5.1.5 Logical gate level

At the logical gate level the design is represented by a netlist that uses only cells from a small number of single-bit cells, such as basic logic gates (AND, OR, NOT, XOR, etc.) and registers (usually D-Type Flip-flops).

A number of netlist formats exists that can be used on this level, e.g. the Electronic Design Interchange Format (EDIF), but for ease of simulation often a HDL netlist is used. The latter is a HDL file (Verilog or VHDL) that only uses the most basic language constructs for instantiation and connecting of cells.

There are two challenges in logic synthesis: First finding opportunities for optimizations within the gate level netlist and second the optimal (or at least good) mapping of the logic gate netlist to an equivalent netlist of physically available gate types.

The simplest approach to logic synthesis is two-level logic synthesis, where a logic function is converted into a sum-of-products representation, e.g. using a Karnaugh map. This is a simple approach, but has exponential worst-case effort and cannot make efficient use of physical gates other than AND/NAND-, OR/NOR- and NOT-Gates.

Therefore modern logic synthesis tools utilize much more complicated multi-level logic synthesis algorithms [BHSV90]. Most of these algorithms convert the logic function to a Binary-Decision-Diagram (BDD) or

And-Inverter-Graph (AIG) and work from that representation. The former has the advantage that it has a unique normalized form. The latter has much better worst case performance and is therefore better suited for the synthesis of large logic functions.

Good FOSS tools exist for multi-level logic synthesis.

Yosys contains basic logic synthesis functionality but can also use ABC for the logic synthesis step. Using ABC is recommended.

5.1.6 Physical gate level

On the physical gate level only gates are used that are physically available on the target architecture. In some cases this may only be NAND, NOR and NOT gates as well as D-Type registers. In other cases this might include cells that are more complex than the cells used at the logical gate level (e.g. complete half-adders). In the case of an FPGA-based design the physical gate level representation is a netlist of LUTs with optional output registers, as these are the basic building blocks of FPGA logic cells.

For the synthesis tool chain this abstraction is usually the lowest level. In case of an ASIC-based design the cell library might contain further information on how the physical cells map to individual switches (transistors).

5.1.7 Switch level

A switch level representation of a circuit is a netlist utilizing single transistors as cells. Switch level modelling is possible in Verilog and VHDL, but is seldom used in modern designs, as in modern digital ASIC or FPGA flows the physical gates are considered the atomic build blocks of the logic circuit.

5.1.8 Yosys

Yosys is a Verilog HDL synthesis tool. This means that it takes a behavioural design description as input and generates an RTL, logical gate or physical gate level description of the design as output. Yosys' main strengths are behavioural and RTL synthesis. A wide range of commands (synthesis passes) exist within Yosys that can be used to perform a wide range of synthesis tasks within the domain of behavioural, rtl and logic synthesis. Yosys is designed to be extensible and therefore is a good basis for implementing custom synthesis tools for specialised tasks.

5.2 Features of synthesizable Verilog

The subset of Verilog [A+06] that is synthesizable is specified in a separate IEEE standards document, the IEEE standard 1364.1-2002 [A+02]. This standard also describes how certain language constructs are to be interpreted in the scope of synthesis.

This section provides a quick overview of the most important features of synthesizable Verilog, structured in order of increasing complexity.

5.2.1 Structural Verilog

Structural Verilog (also known as Verilog Netlists) is a Netlist in Verilog syntax. Only the following language constructs are used in this case:

- Constant values
- Wire and port declarations
- Static assignments of signals to other signals
- Cell instantiations

Many tools (especially at the back end of the synthesis chain) only support structural Verilog as input. ABC is an example of such a tool. Unfortunately there is no standard specifying what Structural Verilog actually is, leading to some confusion about what syntax constructs are supported in structural Verilog when it comes to features such as attributes or multi-bit signals.

5.2.2 Expressions in Verilog

In all situations where Verilog accepts a constant value or signal name, expressions using arithmetic operations such as `+`, `-` and `*`, boolean operations such as `&` (AND), `|` (OR) and `^` (XOR) and many others (comparison operations, unary operator, etc.) can also be used.

During synthesis these operators are replaced by cells that implement the respective function.

Many FOSS tools that claim to be able to process Verilog in fact only support basic structural Verilog and simple expressions. Yosys can be used to convert full featured synthesizable Verilog to this simpler subset, thus enabling such applications to be used with a richer set of Verilog features.

5.2.3 Behavioural modelling

Code that utilizes the Verilog `always` statement is using Behavioural Modelling. In behavioural modelling, a circuit is described by means of imperative program code that is executed on certain events, namely any change, a rising edge, or a falling edge of a signal. This is a very flexible construct during simulation but is only synthesizable when one of the following is modelled:

- **Asynchronous or latched logic**

In this case the sensitivity list must contain all expressions that are used within the `always` block.

The syntax `@*` can be used for these cases. Examples of this kind include:

```

1  // asynchronous
2  always @* begin
3      if (add_mode)
4          y <= a + b;
5      else
6          y <= a - b;
7  end
8
9  // latched
10 always @* begin
11     if (!hold)
12         y <= a + b;
13 end

```

Note that latched logic is often considered bad style and in many cases just the result of sloppy HDL design. Therefore many synthesis tools generate warnings whenever latched logic is generated.

- **Synchronous logic (with optional synchronous reset)**

This is logic with d-type flip-flops on the output. In this case the sensitivity list must only contain the respective clock edge. Example:

```
1 // counter with synchronous reset
2 always @(posedge clk) begin
3     if (reset)
4         y <= 0;
5     else
6         y <= y + 1;
7 end
```

- **Synchronous logic with asynchronous reset**

This is logic with d-type flip-flops with asynchronous resets on the output. In this case the sensitivity list must only contain the respective clock and reset edges. The values assigned in the reset branch must be constant. Example:

```
1 // counter with asynchronous reset
2 always @(posedge clk, posedge reset) begin
3     if (reset)
4         y <= 0;
5     else
6         y <= y + 1;
7 end
```

Many synthesis tools support a wider subset of flip-flops that can be modelled using always-statements (including Yosys). But only the ones listed above are covered by the Verilog synthesis standard and when writing new designs one should limit herself or himself to these cases.

In behavioural modelling, blocking assignments (=) and non-blocking assignments (<=) can be used. The concept of blocking vs. non-blocking assignment is one of the most misunderstood constructs in Verilog [CI00].

The blocking assignment behaves exactly like an assignment in any imperative programming language, while with the non-blocking assignment the right hand side of the assignment is evaluated immediately but the actual update of the left hand side register is delayed until the end of the time-step. For example the Verilog code `a <= b; b <= a;` exchanges the values of the two registers.

5.2.4 Functions and tasks

Verilog supports Functions and Tasks to bundle statements that are used in multiple places (similar to Procedures in imperative programming). Both constructs can be implemented easily by substituting the function/task-call with the body of the function or task.

5.2.5 Conditionals, loops and generate-statements

Verilog supports **if-else**-statements and **for**-loops inside **always**-statements.

It also supports both features in **generate**-statements on the module level. This can be used to selectively enable or disable parts of the module based on the module parameters (**if-else**) or to generate a set of similar subcircuits (**for**).

While the **if-else**-statement inside an **always**-block is part of behavioural modelling, the three other cases are (at least for a synthesis tool) part of a built-in macro processor. Therefore it must be possible for the synthesis tool to completely unroll all loops and evaluate the condition in all **if-else**-statement in **generate**-statements using const-folding..

5.2.6 Arrays and memories

Verilog supports arrays. This is in general a synthesizable language feature. In most cases arrays can be synthesized by generating addressable memories. However, when complex or asynchronous access patterns are used, it is not possible to model an array as memory. In these cases the array must be modelled using individual signals for each word and all accesses to the array must be implemented using large multiplexers.

In some cases it would be possible to model an array using memories, but it is not desired. Consider the following delay circuit:

```

1  module (clk, in_data, out_data);
2
3  parameter BITS = 8;
4  parameter STAGES = 4;
5
6  input clk;
7  input [BITS-1:0] in_data;
8  output [BITS-1:0] out_data;
9  reg [BITS-1:0] ffs [STAGES-1:0];
10
11 integer i;
12 always @(posedge clk) begin
13     ffs[0] <= in_data;
14     for (i = 1; i < STAGES; i = i+1)
15         ffs[i] <= ffs[i-1];
16 end
17
18 assign out_data = ffs[STAGES-1];
19
20 endmodule

```

This could be implemented using an addressable memory with STAGES input and output ports. A better implementation would be to use a simple chain of flip-flops (a so-called shift register). This better implementation can either be obtained by first creating a memory-based implementation and then optimizing it

based on the static address signals for all ports or directly identifying such situations in the language front end and converting all memory accesses to direct accesses to the correct signals.

5.3 Challenges in digital circuit synthesis

This section summarizes the most important challenges in digital circuit synthesis. Tools can be characterized by how well they address these topics.

5.3.1 Standards compliance

The most important challenge is compliance with the HDL standards in question (in case of Verilog the IEEE Standards 1364.1-2002 and 1364-2005). This can be broken down in two items:

- Completeness of implementation of the standard
- Correctness of implementation of the standard

Completeness is mostly important to guarantee compatibility with existing HDL code. Once a design has been verified and tested, HDL designers are very reluctant regarding changes to the design, even if it is only about a few minor changes to work around a missing feature in a new synthesis tool.

Correctness is crucial. In some areas this is obvious (such as correct synthesis of basic behavioural models). But it is also crucial for the areas that concern minor details of the standard, such as the exact rules for handling signed expressions, even when the HDL code does not target different synthesis tools. This is because (unlike software source code that is only processed by compilers), in most design flows HDL code is not only processed by the synthesis tool but also by one or more simulators and sometimes even a formal verification tool. It is key for this verification process that all these tools use the same interpretation for the HDL code.

5.3.2 Optimizations

Generally it is hard to give a one-dimensional description of how well a synthesis tool optimizes the design. First of all because not all optimizations are applicable to all designs and all synthesis tasks. Some optimizations work (best) on a coarse-grained level (with complex cells such as adders or multipliers) and others work (best) on a fine-grained level (single bit gates). Some optimizations target area and others target speed. Some work well on large designs while others don't scale well and can only be applied to small designs.

A good tool is capable of applying a wide range of optimizations at different levels of abstraction and gives the designer control over which optimizations are performed (or skipped) and what the optimization goals are.

5.3.3 Technology mapping

Technology mapping is the process of converting the design into a netlist of cells that are available in the target architecture. In an ASIC flow this might be the process-specific cell library provided by the fab. In an FPGA flow this might be LUT cells as well as special function units such as dedicated multipliers. In a coarse-grain flow this might even be more complex special function units.

An open and vendor independent tool is especially of interest if it supports a wide range of different types of target architectures.

5.4 Script-based synthesis flows

A digital design is usually started by implementing a high-level or system-level simulation of the desired function. This description is then manually transformed (or re-implemented) into a synthesizable lower-level description (usually at the behavioural level) and the equivalence of the two representations is verified by simulating both and comparing the simulation results.

Then the synthesizable description is transformed to lower-level representations using a series of tools and the results are again verified using simulation. This process is illustrated in Fig. 5.2.

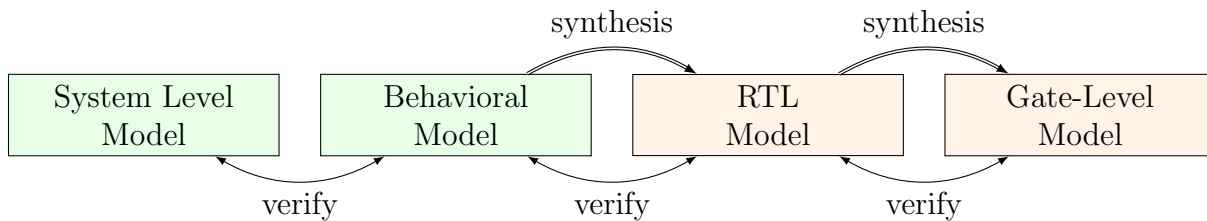


Fig. 5.2: Typical design flow. Green boxes represent manually created models. Orange boxes represent models generated by synthesis tools.

In this example the System Level Model and the Behavioural Model are both manually written design files. After the equivalence of system level model and behavioural model has been verified, the lower level representations of the design can be generated using synthesis tools. Finally the RTL Model and the Gate-Level Model are verified and the design process is finished.

However, in any real-world design effort there will be multiple iterations for this design process. The reason for this can be the late change of a design requirement or the fact that the analysis of a low-abstraction model (e.g. gate-level timing analysis) revealed that a design change is required in order to meet the design requirements (e.g. maximum possible clock speed).

Whenever the behavioural model or the system level model is changed their equivalence must be re-verified by re-running the simulations and comparing the results. Whenever the behavioural model is changed the synthesis must be re-run and the synthesis results must be re-verified.

In order to guarantee reproducibility it is important to be able to re-run all automatic steps in a design project with a fixed set of settings easily. Because of this, usually all programs used in a synthesis flow can be controlled using scripts. This means that all functions are available via text commands. When such a tool provides a GUI, this is complementary to, and not instead of, a command line interface.

Usually a synthesis flow in an UNIX/Linux environment would be controlled by a shell script that calls all required tools (synthesis and simulation/verification in this example) in the correct order. Each of these tools would be called with a script file containing commands for the respective tool. All settings required for the tool would be provided by these script files so that no manual interaction would be necessary. These script files are considered design sources and should be kept under version control just like the source code of the system level and the behavioural model.

5.5 Methods from compiler design

Some parts of synthesis tools involve problem domains that are traditionally known from compiler design. This section addresses some of these domains.

5.5.1 Lexing and parsing

The best known concepts from compiler design are probably lexing and parsing. These are two methods that together can be used to process complex computer languages easily. [ASU86]

A lexer consumes single characters from the input and generates a stream of lexical tokens that consist of a type and a value. For example the Verilog input `assign foo = bar + 42;` might be translated by the lexer to the list of lexical tokens given in Tab. 5.1.

Table 5.1: Exemplary token list for the statement `assign foo = bar + 42;`

Token-Type	Token-Value
TOK_ASSIGN	-
TOK_IDENTIFIER	"foo"
TOK_EQ	-
TOK_IDENTIFIER	"bar"
TOK_PLUS	-
TOK_NUMBER	42
TOK_SEMICOLON	-

The lexer is usually generated by a lexer generator (e.g. flex) from a description file that is using regular expressions to specify the text pattern that should match the individual tokens.

The lexer is also responsible for skipping ignored characters (such as whitespace outside string constants and comments in the case of Verilog) and converting the original text snippet to a token value.

Note that individual keywords use different token types (instead of a keyword type with different token values). This is because the parser usually can only use the Token-Type to make a decision on the grammatical role of a token.

The parser then transforms the list of tokens into a parse tree that closely resembles the productions from the computer languages grammar. As the lexer, the parser is also typically generated by a code generator (e.g. bison) from a grammar description in Backus-Naur Form (BNF).

Let's consider the following BNF (in Bison syntax):

```

1 assign_stmt: TOK_ASSIGN TOK_IDENTIFIER TOK_EQ expr TOK_SEMICOLON;
2 expr: TOK_IDENTIFIER | TOK_NUMBER | expr TOK_PLUS expr;
```

The parser converts the token list to the parse tree in Fig. 5.3. Note that the parse tree never actually exists as a whole as data structure in memory. Instead the parser calls user-specified code snippets (so-called reduce-functions) for all inner nodes of the parse tree in depth-first order.

In some very simple applications (e.g. code generation for stack machines) it is possible to perform the task at hand directly in the reduce functions. But usually the reduce functions are only used to build an in-memory data structure with the relevant information from the parse tree. This data structure is called an abstract syntax tree (AST).

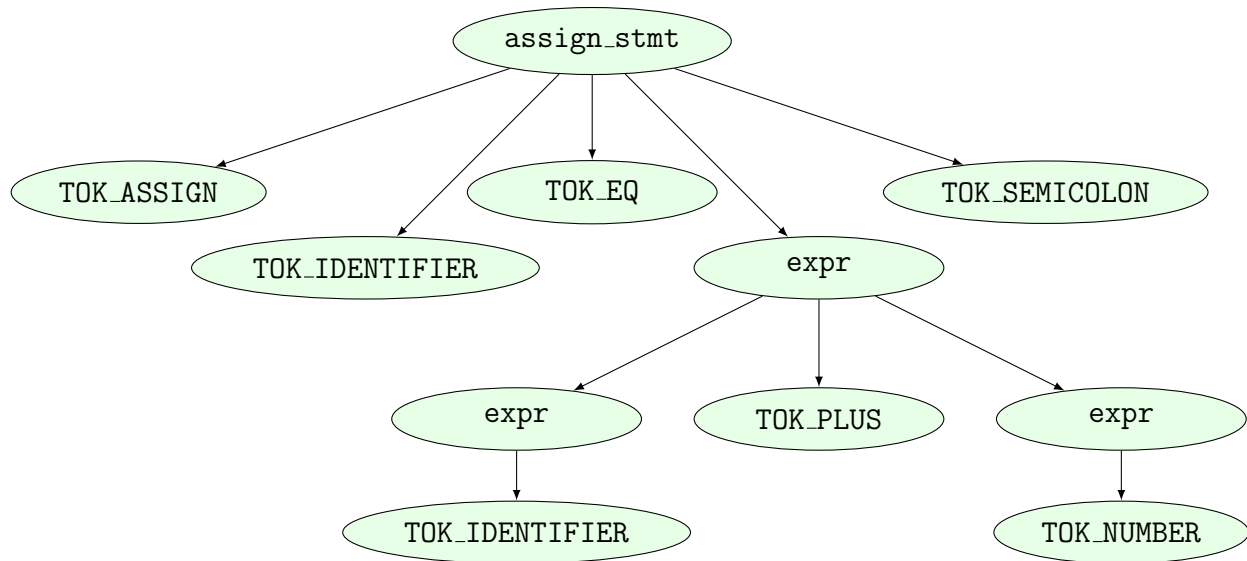


Fig. 5.3: Example parse tree for the Verilog expression `assign foo = bar + 42;`

The exact format for the abstract syntax tree is application specific (while the format of the parse tree and token list are mostly dictated by the grammar of the language at hand). Figure 5.4 illustrates what an AST for the parse tree in Fig. 5.3 could look like.

Usually the AST is then converted into yet another representation that is more suitable for further processing. In compilers this is often an assembler-like three-address-code intermediate representation. [ASU86]

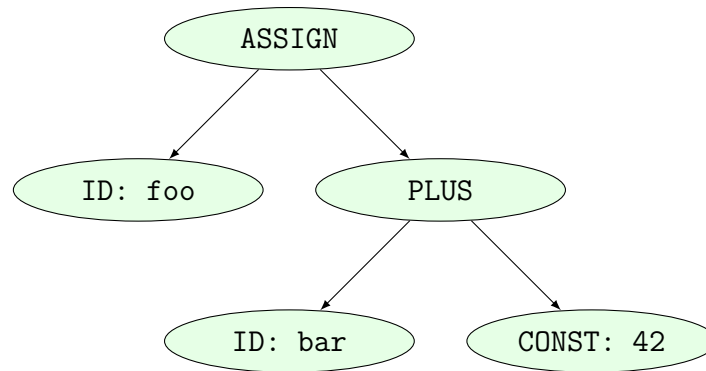


Fig. 5.4: Example abstract syntax tree for the Verilog expression `assign foo = bar + 42;`

5.5.2 Multi-pass compilation

Complex problems are often best solved when split up into smaller problems. This is certainly true for compilers as well as for synthesis tools. The components responsible for solving the smaller problems can be connected in two different ways: through Single-Pass Pipelining and by using Multiple Passes.

Traditionally a parser and lexer are connected using the pipelined approach: The lexer provides a function that is called by the parser. This function reads data from the input until a complete lexical token has been read. Then this token is returned to the parser. So the lexer does not first generate a complete list of lexical tokens and then pass it to the parser. Instead they run concurrently and the parser can consume tokens as the lexer produces them.

The single-pass pipelining approach has the advantage of lower memory footprint (at no time must the complete design be kept in memory) but has the disadvantage of tighter coupling between the interacting components.

Therefore single-pass pipelining should only be used when the lower memory footprint is required or the components are also conceptually tightly coupled. The latter certainly is the case for a parser and its lexer. But when data is passed between two conceptually loosely coupled components it is often beneficial to use a multi-pass approach.

In the multi-pass approach the first component processes all the data and the result is stored in a in-memory data structure. Then the second component is called with this data. This reduces complexity, as only one component is running at a time. It also improves flexibility as components can be exchanged easier.

Most modern compilers are multi-pass compilers.

5.5.3 Static Single Assignment (SSA) form

In imperative programming (and behavioural HDL design) it is possible to assign the same variable multiple times. This can either mean that the variable is independently used in two different contexts or that the final value of the variable depends on a condition.

The following examples show C code in which one variable is used independently in two different contexts:

```
1 void demo1()
2 {
3     int a = 1;
4     printf("%d\n", a);
5
6     a = 2;
7     printf("%d\n", a);
8 }
```

```
void demo1()
{
    int a = 1;
    printf("%d\n", a);

    int b = 2;
    printf("%d\n", b);
}
```

```
1 void demo2(bool foo)
2 {
3     int a;
4     if (foo) {
5         a = 23;
6         printf("%d\n", a);
7     } else {
8         a = 42;
9         printf("%d\n", a);
10    }
11 }
```

```

void demo2(bool foo)
{
    int a, b;
    if (foo) {
        a = 23;
        printf("%d\n", a);
    } else {
        b = 42;
        printf("%d\n", b);
    }
}

```

In both examples the left version (only variable `a`) and the right version (variables `a` and `b`) are equivalent. Therefore it is desired for further processing to bring the code in an equivalent form for both cases.

In the following example the variable is assigned twice but it cannot be easily replaced by two variables:

```

void demo3(bool foo)
{
    int a = 23
    if (foo)
        a = 42;
    printf("%d\n", a);
}

```

Static single assignment (SSA) form is a representation of imperative code that uses identical representations for the left and right version of demos 1 and 2, but can still represent demo 3. In SSA form each assignment assigns a new variable (usually written with an index). But it also introduces a special Φ -function to merge the different instances of a variable when needed. In C-pseudo-code the demo 3 would be written as follows using SSA form:

```

void demo3(bool foo)
{
    int a_1, a_2, a_3;
    a_1 = 23
    if (foo)
        a_2 = 42;
    a_3 = phi(a_1, a_2);
    printf("%d\n", a_3);
}

```

The Φ -function is usually interpreted as “these variables must be stored in the same memory location” during code generation. Most modern compilers for imperative languages such as C/C++ use SSA form for at least some of its passes as it is very easy to manipulate and analyse.

RTLIL TEXT REPRESENTATION

This appendix documents the text representation of RTLIL in extended Backus-Naur form (EBNF).

The grammar is not meant to represent semantic limitations. That is, the grammar is “permissive”, and later stages of processing perform more rigorous checks.

The grammar is also not meant to represent the exact grammar used in the RTLIL frontend, since that grammar is specific to processing by lex and yacc, is even more permissive, and is somewhat less understandable than simple EBNF notation.

Finally, note that all statements (rules ending in `-stmt`) terminate in an end-of-line. Because of this, a statement cannot be broken into multiple lines.

6.1 Lexical elements

6.1.1 Characters

An RTLIL file is a stream of bytes. Strictly speaking, a “character” in an RTLIL file is a single byte. The lexer treats multi-byte encoded characters as consecutive single-byte characters. While other encodings *may* work, UTF-8 is known to be safe to use. Byte order marks at the beginning of the file will cause an error.

ASCII spaces (32) and tabs (9) separate lexer tokens.

A `nonws` character, used in identifiers, is any character whose encoding consists solely of bytes above ASCII space (32).

An `eof` is one or more consecutive ASCII newlines (10) and carriage returns (13).

6.1.2 Identifiers

There are two types of identifiers in RTLIL:

- Publically visible identifiers
- Auto-generated identifiers

```
<id>          ::= <public-id> | <autogen-id>
<public-id>   ::= \ <nonws>+
<autogen-id>  ::= $ <nonws>+
```

6.1.3 Values

A *value* consists of a width in bits and a bit representation, most significant bit first. Bits may be any of:

- 0: A logic zero value
- 1: A logic one value
- x: An unknown logic value (or don't care in case patterns)
- z: A high-impedance value (or don't care in case patterns)
- m: A marked bit (internal use only)
- -: A don't care value

An *integer* is simply a signed integer value in decimal format. **Warning:** Integer constants are limited to 32 bits. That is, they may only be in the range $[-2147483648, 2147483648]$. Integers outside this range will result in an error.

```
<value>      ::= <decimal-digit>+ ' ' <binary-digit>*  
<decimal-digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  
<binary-digit> ::= 0 | 1 | x | z | m | -  
<integer>    ::= -? <decimal-digit>+
```

6.1.4 Strings

A string is a series of characters delimited by double-quote characters. Within a string, any character except ASCII NUL (0) may be used. In addition, certain escapes can be used:

- \n: A newline
- \t: A tab
- \ooo: A character specified as a one, two, or three digit octal value

All other characters may be escaped by a backslash, and become the following character. Thus:

- \\: A backslash
- \": A double-quote
- \r: An 'r' character

6.1.5 Comments

A comment starts with a # character and proceeds to the end of the line. All comments are ignored.

6.2 File

A file consists of an optional autoindex statement followed by zero or more modules.

```
<file> ::= <autoidx-stmt>? <module>*
```

6.2.1 Autoindex statements

The autoindex statement sets the global autoindex value used by Yosys when it needs to generate a unique name, e.g. `flattenN`. The N part is filled with the value of the global autoindex value, which is subsequently incremented. This global has to be dumped into RTLIL, otherwise e.g. dumping and running a pass would have different properties than just running a pass on a warm design.

```
<autoidx-stmt> ::= autoidx <integer> <eol>
```

6.2.2 Modules

Declares a module, with zero or more attributes, consisting of zero or more wires, memories, cells, processes, and connections.

```
<module>          ::= <attr-stmt>* <module-stmt> <module-body> <module-end-stmt>
<module-stmt>     ::= module <id> <eol>
<module-body>     ::= (<param-stmt>
                        | <wire>
                        | <memory>
                        | <cell>
                        | <process>)*
<param-stmt>      ::= parameter <id> <constant>? <eol>
<constant>        ::= <value> | <integer> | <string>
<module-end-stmt> ::= end <eol>
```

6.2.3 Attribute statements

Declares an attribute with the given identifier and value.

```
<attr-stmt> ::= attribute <id> <constant> <eol>
```

6.2.4 Signal specifications

A signal is anything that can be applied to a cell port, i.e. a constant value, all bits or a selection of bits from a wire, or concatenations of those.

Warning: When an integer constant is a sigspec, it is always 32 bits wide, 2's complement. For example, a constant of `-1` is the same as `32'11111111111111111111111111111111`, while a constant of `1` is the same as `32'1`.

See *RTLIL::SigSpec* for an overview of signal specifications.

```
<sigspec> ::= <constant>
           | <wire-id>
           | <sigspec> [ <integer> (:<integer>)? ]
           | { <sigspec>* }
```

6.2.5 Connections

Declares a connection between the given signals.

```
<conn-stmt> ::= connect <sigspec> <sigspec> <eol>
```

6.2.6 Wires

Declares a wire, with zero or more attributes, with the given identifier and options in the enclosing module.

See *RTLIL::Cell* and *RTLIL::Wire* for an overview of wires.

```
<wire>          ::= <attr-stmt>* <wire-stmt>
<wire-stmt>     ::= wire <wire-option>* <wire-id> <eol>
<wire-id>       ::= <id>
<wire-option>   ::= width <integer>
                  | offset <integer>
                  | input <integer>
                  | output <integer>
                  | inout <integer>
                  | upto
                  | signed
```

6.2.7 Memories

Declares a memory, with zero or more attributes, with the given identifier and options in the enclosing module.

See *RTLIL::Memory* for an overview of memory cells, and *Memories* for details about memory cell types.

```
<memory>        ::= <attr-stmt>* <memory-stmt>
<memory-stmt>   ::= memory <memory-option>* <id> <eol>
<memory-option> ::= width <integer>
                  | size <integer>
                  | offset <integer>
```

6.2.8 Cells

Declares a cell, with zero or more attributes, with the given identifier and type in the enclosing module.

Cells perform functions on input signals. See *Internal cell library* for a detailed list of cell types.

```
<cell>           ::= <attr-stmt>* <cell-stmt> <cell-body-stmt>* <cell-end-stmt>
<cell-stmt>      ::= cell <cell-type> <cell-id> <eol>
<cell-id>        ::= <id>
<cell-type>      ::= <id>
<cell-body-stmt> ::= parameter (signed | real)? <id> <constant> <eol>
                  | connect <id> <sigspec> <eol>
<cell-end-stmt>  ::= end <eol>
```

6.2.9 Processes

Declares a process, with zero or more attributes, with the given identifier in the enclosing module. The body of a process consists of zero or more assignments followed by zero or more switches and zero or more syncs.

See *RTLIL::Process* for an overview of processes.

```
<process>      ::= <attr-stmt>* <proc-stmt> <process-body> <proc-end-stmt>
<proc-stmt>    ::= process <id> <eol>
<process-body> ::= <assign-stmt>* <switch>* <sync>*
<assign-stmt>  ::= assign <dest-sigspec> <src-sigspec> <eol>
<dest-sigspec> ::= <sigspec>
<src-sigspec>  ::= <sigspec>
<proc-end-stmt> ::= end <eol>
```

6.2.10 Switches

Switches test a signal for equality against a list of cases. Each case specifies a comma-separated list of signals to check against. If there are no signals in the list, then the case is the default case. The body of a case consists of zero or more assignments followed by zero or more switches. Both switches and cases may have zero or more attributes.

```
<switch>      ::= <switch-stmt> <case>* <switch-end-stmt>
<switch-stmt> ::= <attr-stmt>* switch <sigspec> <eol>
<case>        ::= <attr-stmt>* <case-stmt> <case-body>
<case-stmt>   ::= case <compare>? <eol>
<compare>     ::= <sigspec> (, <sigspec>)*
<case-body>   ::= <assign-stmt>* <switch>*
<switch-end-stmt> ::= end <eol>
```

6.2.11 Syncs

Syncs update signals with other signals when an event happens. Such an event may be:

- An edge or level on a signal
- Global clock ticks
- Initialization
- Always

```
<sync>      ::= <sync-stmt> <update-stmt>*
<sync-stmt> ::= sync <sync-type> <sigspec> <eol>
              | sync global <eol>
              | sync init <eol>
              | sync always <eol>
<sync-type> ::= low | high | posedge | negedge | edge
<update-stmt> ::= update <dest-sigspec> <src-sigspec> <eol>
```


AUXILIARY LIBRARIES

The Yosys source distribution contains some auxiliary libraries that are compiled into Yosys and can be used in plugins.

7.1 BigInt

The files in `libs/bigint/` provide a library for performing arithmetic with arbitrary length integers. It is written by Matt McCutchen.

The BigInt library is used for evaluating constant expressions, e.g. using the `ConstEval` class provided in `kernel/consteval.h`.

See also: <http://mattmccutchen.net/bigint/>

7.2 dlfcn-win32

The `dlfcn` library enables runtime loading of plugins without requiring recompilation of Yosys. The files in `libs/dlfcn-win32` provide an implementation of `dlfcn` for Windows.

See also: <https://github.com/dlfcn-win32/dlfcn-win32>

7.3 ezSAT

The files in `libs/ezsat` provide a library for simplifying generating CNF formulas for SAT solvers. It also contains bindings of MiniSAT. The ezSAT library is written by C. Wolf. It is used by the `sat` pass (see `sat - solve a SAT problem in the circuit`).

7.4 fst

`libfst` files from `gtkwave` are included in `libs/fst` to support reading/writing signal traces from/to the GTKWave developed FST format. This is primarily used in the `sim` command.

7.5 json11

For reading/writing designs from/to JSON, *read_json* and *write_json* should be used. For everything else there is the *json11* library:

json11 is a tiny JSON library for C++11, providing JSON parsing and serialization.

This library is used for outputting machine-readable statistics (*stat* with *-json* flag), using the RPC frontend (*connect_rpc*), and the yosys-witness *yw* format.

7.6 MiniSAT

The files in *libs/minisat* provide a high-performance SAT solver, used by the *sat* command.

7.7 SHA1

The files in *libs/sha1/* provide a public domain SHA1 implementation written by Steve Reid, Bruce Guenter, and Volker Grabsch. It is used for generating unique names when specializing parameterized modules.

7.8 SubCircuit

The files in *libs/subcircuit* provide a library for solving the subcircuit isomorphism problem. It is written by C. Wolf and based on the Ullmann Subgraph Isomorphism Algorithm [Ull76]. It is used by the *extract* pass (see *extract - find subcircuits and replace them with cells*).

AUXILIARY PROGRAMS

Besides the main yosys executable, the Yosys distribution contains a set of additional helper programs.

8.1 yosys-config

The `yosys-config` tool (an auto-generated shell-script) can be used to query compiler options and other information needed for building loadable modules for Yosys. See [Writing extensions](#) for details.

```
Usage: ./yosys-config [--exec] [--prefix pf] args..  
      ./yosys-config --build modname.so cppsources..
```

Replacement args:

```
--cxx      g++  
--cxxflags -O2 -flto=auto -ffat-lto-objects -fexceptions -g \  
            -grecord-gcc-switches -pipe -Wall -Werror=format-security \  
            -Wp,-U_FORTIFY_SOURCE,-D_FORTIFY_SOURCE=3 \  
            -Wp,-D_GLIBCXX_ASSERTIONS \  
            -specs=/usr/lib/rpm/redhat/redhat-hardened-cc1 \  
            -fstack-protector-strong \  
            -specs=/usr/lib/rpm/redhat/redhat-annobin-cc1 \  
            -mbranch-protection=standard -fasynchronous-unwind-tables \  
            -fstack-clash-protection -fno-omit-frame-pointer \  
            -mno-omit-leaf-frame-pointer -Wall -Wextra -ggdb \  
            -I"/usr/include/yosys" -MD -MP -D_YOSYS_ -fPIC \  
            -I/usr/include -DYOSYS_VER=\"0.51\" -DYOSYS_MAJOR=0 \  
            -DYOSYS_MINOR=51 -DYOSYS_COMMIT=0.51 -std=c++17 \  
            -O3 -DYOSYS_ENABLE_READLINE -DYOSYS_ENABLE_PLUGINS \  
            -DYOSYS_ENABLE_GLOB -DYOSYS_ENABLE_ZLIB -DWITH_GZFILEOP \  
            -DYOSYS_ENABLE_TCL -DYOSYS_ENABLE_ABC -DYOSYS_ENABLE_COVER  
--linkflags -rdynamic  
--ldflags   (alias of --linkflags)  
--libs      -lstdc++ -lm -lrt -lreadline -L/usr/lib64/../../lib64 -lffi -ldl -lz -  
↳ libtcl8.6 -ltclstub8.6  
--ldlibs     (alias of --libs)  
--bindir     /usr/bin  
--datadir    /usr/share/yosys
```

All other args are passed through as they are.

Use `--exec` to call a command instead of generating output. Example usage:

(continues on next page)

(continued from previous page)

```
./yosys-config --exec --cxx --cxxflags --ldflags -o plugin.so -shared plugin.cc --libs
```

The above command can be abbreviated as:

```
./yosys-config --build plugin.so plugin.cc
```

Use `--prefix` to change the prefix for the special args from `'--'` to something else. Example:

```
./yosys-config --prefix @ bindir: @bindir
```

The args `--bindir` and `--datdir` can be directly followed by a slash and additional text. Example:

```
./yosys-config --datdir/simlib.v
```

8.2 yosys-filterlib

Todo: how does a filterlib rules-file work?

The `yosys-filterlib` tool is a small utility that can be used to strip or extract information from a Liberty file. This can be useful for removing sensitive or proprietary information such as timing or other trade secrets.

```
Usage: filterlib [rules-file [liberty-file]]
       or: filterlib -verilogsim [liberty-file]
```

8.3 yosys-abc

This is a fork of ABC with a small set of custom modifications that have not yet been accepted upstream. Not all versions of Yosys work with all versions of ABC. So Yosys comes with its own `yosys-abc` to avoid compatibility issues between the two.

8.4 yosys-smtbmc

The `yosys-smtbmc` tool is a utility used by SBY for interacting with smt solvers.

```
yosys-smtbmc [options] <yosys_smt2_output>

-h, --help
      show this message

-t <num_steps>
```

(continues on next page)

(continued from previous page)

```

-t <skip_steps>:<num_steps>
-t <skip_steps>:<step_size>:<num_steps>
    default: skip_steps=0, step_size=1, num_steps=20

-g
    generate an arbitrary trace that satisfies
    all assertions and assumptions.

-i
    instead of BMC run temporal induction

-c
    instead of regular BMC run cover analysis

-m <module_name>
    name of the top module

--smtc <constr_filename>
    read constraints file

--cex <cex_filename>
    read cex file as written by ABC's "write_cex -n"

--aig <prefix>
    read AIGER map file (as written by Yosys' "write_aiger -map")
    and AIGER witness file. The file names are <prefix>.aim for
    the map file and <prefix>.aiw for the witness file.

--aig <aim_filename>:<aiw_filename>
    like above, but for map files and witness files that do not
    share a filename prefix (or use different file extensions).

--aig-noheader
    the AIGER witness file does not include the status and
    properties lines.

--yw <yosys_witness_filename>
    read a Yosys witness.

--btorwit <btor_witness_filename>
    read a BTOR witness.

--noinfo
    only run the core proof, do not collect and print any
    additional information (e.g. which assert failed)

--presat
    check if the design with assumptions but without assertions
    is SAT before checking if assertions are UNSAT. This will
    detect if there are contradicting assumptions. In some cases
    this will also help to "warm up" the solver, potentially
    yielding a speedup.

```

(continues on next page)

(continued from previous page)

```
--final-only
    only check final constraints, assume base case

--assume-skipped <start_step>
    assume asserts in skipped steps in BMC.
    no assumptions are created for skipped steps
    before <start_step>.

--dump-vcd <vcd_filename>
    write trace to this VCD file
    (hint: use 'write_smt2 -wires' for maximum
    coverage of signals in generated VCD file)

--dump-yw <yw_filename>
    write trace as a Yosys witness trace

--dump-vlogtb <verilog_filename>
    write trace as Verilog test bench

--vlogtb-top <hierarchical_name>
    use the given entity as top module for the generated
    Verilog test bench. The <hierarchical_name> is relative
    to the design top module without the top module name.

--dump-smtc <constr_filename>
    write trace as constraints file

--smtc-init
    write just the last state as initial constraint to smtc file

--smtc-top <old>[:<new>]
    replace <old> with <new> in constraints dumped to smtc
    file and only dump object below <old> in design hierarchy.

--noinit
    do not assume initial conditions in state 0

--dump-all
    when using -g or -i, create a dump file for each
    step. The character '%' is replaced in all dump
    filenames with the step number.

--append <num_steps>
    add <num_steps> time steps at the end of the trace
    when creating a counter example (this additional time
    steps will still be constrained by assumptions)

--binary
    dump anyconst values as raw bit strings

--keep-going
```

(continues on next page)

(continued from previous page)

```

    continue BMC after the first failed assertion and report
    further failed assertions. To output multiple traces
    covering all found failed assertions, the character '%' is
    replaced in all dump filenames with an increasing number.
    In cover mode, don't stop when a cover trace contains a failed
    assertion.

--check-witness
    check that the used witness file contains sufficient
    constraints to force an assertion failure.

--detect-loops
    check if states are unique in temporal induction counter examples
    (this feature is experimental and incomplete)

--incremental
    run in incremental mode (experimental)

--track-assumes
    track individual assumptions and report a subset of used
    assumptions that are sufficient for the reported outcome. This
    can be used to debug PREUNSAT failures as well as to find a
    smaller set of sufficient assumptions.

--minimize-assumes
    when using --track-assumes, solve for a minimal set of sufficient assumptions.

-s <solver>
    set SMT solver: z3, yices, boolector, bitwuzla, cvc4, cvc5, mathsat, dummy
    default: yices

-S <opt>
    pass <opt> as command line argument to the solver

--timeout <value>
    set the solver timeout to the specified value (in seconds).

--logic <smt2_logic>
    use the specified SMT2 logic (e.g. QF_AUFBV)

--dummy <filename>
    if solver is "dummy", read solver output from that file
    otherwise: write solver output to that file

--smt2-option <option>=<value>
    enable an SMT-LIBv2 option.

-v
    enable debug output

--unroll
    unroll uninterpreted functions

```

(continues on next page)

(continued from previous page)

```
--noincr
    don't use incremental solving, instead restart solver for
    each (check-sat). This also avoids (push) and (pop).

--noprogess
    disable timer display during solving
    (this option is set implicitly on Windows)

--dump-smt2 <filename>
    write smt2 statements to file

--info <smt2-info-stmt>
    include the specified smt2 info statement in the smt2 output

--nocomments
    strip all comments from the generated smt2 code
```

8.5 yosys-witness

`yosys-witness` is a new tool to inspect and convert yosys witness traces. This is used in SBY and SCY for producing traces in a consistent format independent of the solver.

Usage: `yosys-witness [OPTIONS] COMMAND [ARGS]...`

Options:

`--help` Show this message and exit.

Commands:

<code>aiw2yw</code>	Convert an AIGER witness trace into a Yosys witness trace.
<code>display</code>	Display a Yosys witness trace in a human readable format.
<code>stats</code>	Display statistics of a Yosys witness trace.
<code>wit2yw</code>	Convert a BTOR witness trace into a Yosys witness trace.
<code>yw2aiw</code>	Convert a Yosys witness trace into an AIGER witness trace.
<code>yw2yw</code>	Transform a Yosys witness trace.

Note: `yosys-witness` requires [click](#) Python package for use.

Todo: see if we can get the two hanging appnotes as lit references

INTERNAL CELL LIBRARY

The intermediate language used by Yosys (RTLIL) represents logic and memory with a series of cells. This section provides details for those cells, breaking them down into two major categories: coarse-grain word-level cells; and fine-grain gate-level cells. An additional section contains a list of properties which may be shared across multiple cells.

9.1 Word-level cells

Most of the RTL cells closely resemble the operators available in HDLs such as Verilog or VHDL. Therefore Verilog operators are used in the following sections to define the behaviour of the RTL cells.

Note that all RTL cells have parameters indicating the size of inputs and outputs. When passes modify RTL cells they must always keep the values of these parameters in sync with the size of the signals connected to the inputs and outputs.

Simulation models for the RTL cells can be found in the file `techlibs/common/simlib.v` in the Yosys source tree.

9.1.1 Unary operators

All unary RTL cells have one input port **A** and one output port **Y**. They also have the following parameters:

A_SIGNED

Set to a non-zero value if the input **A** is signed and therefore should be sign-extended when needed.

A_WIDTH

The width of the input port **A**.

Y_WIDTH

The width of the output port **Y**.

Table 9.1: Cell types for unary operators with their corresponding Verilog expressions.

Verilog	Cell Type
<code>Y = ~A</code>	<code>\$not</code>
<code>Y = +A</code>	<code>\$pos</code>
<code>Y = -A</code>	<code>\$neg</code>
<code>Y = &A</code>	<code>\$reduce_and</code>
<code>Y = A</code>	<code>\$reduce_or</code>
<code>Y = ^A</code>	<code>\$reduce_xor</code>
<code>Y = ~^A</code>	<code>\$reduce_xnor</code>
<code>Y = A</code>	<code>\$reduce_bool</code>
<code>Y = !A</code>	<code>\$logic_not</code>

For the unary cells that output a logical value (`$reduce_and`, `$reduce_or`, `$reduce_xor`, `$reduce_xnor`, `$reduce_bool`, `$logic_not`), when the `Y_WIDTH` parameter is greater than 1, the output is zero-extended, and only the least significant bit varies.

Note that `$reduce_or` and `$reduce_bool` generally represent the same logic function. But the `read_verilog` frontend will generate them in different situations. A `$reduce_or` cell is generated when the prefix `|` operator is being used. A `$reduce_bool` cell is generated when a bit vector is used as a condition in an `if`-statement or `?:`-expression.

yosys> help \$buf

A simple coarse-grain buffer cell type for the experimental buffered-normalized mode. Note this cell doesn't get removed by 'opt_clean' and is not recommended for general use.

Properties

`is_evaluable`

Simulation model (verilog)

Listing 9.1: simlib.v

```

98 module \ $buf (A, Y);
99
100     parameter WIDTH = 0;
101
102     input [WIDTH-1:0] A;
103     output [WIDTH-1:0] Y;
104
105     assign Y = A;
106
107 endmodule

```

yosys> help \$logic_not

A logical inverter. This corresponds to the Verilog unary prefix '!' operator.

Properties

`is_evaluable`

Simulation model (verilog)

Listing 9.2: simlib.v

```

1399 module \$logic_not (A, Y);
1400
1401     parameter A_SIGNED = 0;
1402     parameter A_WIDTH = 0;
1403     parameter Y_WIDTH = 0;
1404
1405     input [A_WIDTH-1:0] A;
1406     output [Y_WIDTH-1:0] Y;
1407
1408     generate
1409         if (A_SIGNED) begin:BLOCK1
1410             assign Y = !$signed(A);
1411         end else begin:BLOCK2
1412             assign Y = !A;
1413         end
1414     endgenerate
1415
1416 endmodule

```

yosys> help \$neg

An arithmetic inverter. This corresponds to the Verilog unary prefix '-' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.3: simlib.v

```

118 module \$neg (A, Y);
119
120     parameter A_SIGNED = 0;
121     parameter A_WIDTH = 0;
122     parameter Y_WIDTH = 0;
123
124     input [A_WIDTH-1:0] A;
125     output [Y_WIDTH-1:0] Y;
126
127     generate
128         if (A_SIGNED) begin:BLOCK1
129             assign Y = -$signed(A);
130         end else begin:BLOCK2
131             assign Y = -A;
132         end
133     endgenerate
134
135 endmodule

```

yosys> help \$not

Bit-wise inverter

This corresponds to the Verilog unary prefix '~' operator.

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.4: simlib.v

```
40 module \$.not (A, Y);
41
42     parameter A_SIGNED = 0;
43     parameter A_WIDTH = 0;
44     parameter Y_WIDTH = 0;
45
46     input [A_WIDTH-1:0] A;
47     output [Y_WIDTH-1:0] Y;
48
49     generate
50         if (A_SIGNED) begin:BLOCK1
51             assign Y = ~$signed(A);
52         end else begin:BLOCK2
53             assign Y = ~A;
54         end
55     endgenerate
56
57 endmodule
```

yosys> help \$pos

A buffer. This corresponds to the Verilog unary prefix '+' operator.

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.5: simlib.v

```
68 module \$.pos (A, Y);
69
70     parameter A_SIGNED = 0;
71     parameter A_WIDTH = 0;
72     parameter Y_WIDTH = 0;
73
74     input [A_WIDTH-1:0] A;
75     output [Y_WIDTH-1:0] Y;
76
77     generate
78         if (A_SIGNED) begin:BLOCK1
79             assign Y = $signed(A);
80         end else begin:BLOCK2
81             assign Y = A;
82         end
83     endgenerate
84
85 endmodule
```

yosys> help \$reduce_and

An AND reduction. This corresponds to the Verilog unary prefix '&' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.6: simlib.v

```

270 module \ $reduce_and (A, Y);
271
272     parameter A_SIGNED = 0;
273     parameter A_WIDTH = 0;
274     parameter Y_WIDTH = 0;
275
276     input [A_WIDTH-1:0] A;
277     output [Y_WIDTH-1:0] Y;
278
279     generate
280         if (A_SIGNED) begin:BLOCK1
281             assign Y = &$signed(A);
282         end else begin:BLOCK2
283             assign Y = &A;
284         end
285     endgenerate
286
287 endmodule

```

yosys> help \$reduce_bool

An OR reduction. This cell type is used instead of \$reduce_or when a signal is implicitly converted to a boolean signal, e.g. for operands of '&&' and '||'.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.7: simlib.v

```

383 module \ $reduce_bool (A, Y);
384
385     parameter A_SIGNED = 0;
386     parameter A_WIDTH = 0;
387     parameter Y_WIDTH = 0;
388
389     input [A_WIDTH-1:0] A;
390     output [Y_WIDTH-1:0] Y;
391
392     generate
393         if (A_SIGNED) begin:BLOCK1
394             assign Y = !(~$signed(A));
395         end else begin:BLOCK2
396             assign Y = !(~A);
397         end
398     endgenerate
399
400 endmodule

```

(continues on next page)

(continued from previous page)

```

398     endgenerate
399
400 endmodule

```

yosys> help \$reduce_or

An OR reduction. This corresponds to the Verilog unary prefix ‘|’ operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.8: simlib.v

```

298 module \ $reduce_or (A, Y);
299
300     parameter A_SIGNED = 0;
301     parameter A_WIDTH = 0;
302     parameter Y_WIDTH = 0;
303
304     input [A_WIDTH-1:0] A;
305     output [Y_WIDTH-1:0] Y;
306
307     generate
308         if (A_SIGNED) begin:BLOCK1
309             assign Y = |$signed(A);
310         end else begin:BLOCK2
311             assign Y = |A;
312         end
313     endgenerate
314
315 endmodule

```

yosys> help \$reduce_xnor

A XNOR reduction. This corresponds to the Verilog unary prefix ‘~^’ operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.9: simlib.v

```

354 module \ $reduce_xnor (A, Y);
355
356     parameter A_SIGNED = 0;
357     parameter A_WIDTH = 0;
358     parameter Y_WIDTH = 0;
359
360     input [A_WIDTH-1:0] A;
361     output [Y_WIDTH-1:0] Y;
362
363     generate
364         if (A_SIGNED) begin:BLOCK1

```

(continues on next page)

(continued from previous page)

```

365         assign Y = ~$signed(A);
366     end else begin:BLOCK2
367         assign Y = ~A;
368     end
369 endgenerate
370
371 endmodule

```

yosys> help \$reduce_xor

A XOR reduction. This corresponds to the Verilog unary prefix ‘^’ operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.10: simlib.v

```

326 module \ $reduce_xor (A, Y);
327
328     parameter A_SIGNED = 0;
329     parameter A_WIDTH = 0;
330     parameter Y_WIDTH = 0;
331
332     input [A_WIDTH-1:0] A;
333     output [Y_WIDTH-1:0] Y;
334
335     generate
336         if (A_SIGNED) begin:BLOCK1
337             assign Y = $signed(A);
338         end else begin:BLOCK2
339             assign Y = ^A;
340         end
341     endgenerate
342
343 endmodule

```

9.1.2 Binary operators

All binary RTL cells have two input ports A and B and one output port Y. They also have the following parameters:

A_SIGNED

Set to a non-zero value if the input A is signed and therefore should be sign-extended when needed.

A_WIDTH

The width of the input port A.

B_SIGNED

Set to a non-zero value if the input B is signed and therefore should be sign-extended when needed.

B_WIDTH

The width of the input port B.

Y_WIDTH

The width of the output port Y.

Table 9.2: Cell types for binary operators with their corresponding Verilog expressions.

Verilog	Cell Type	Verilog	Cell Type
<code>Y = A & B</code>	<code>\$and</code>	<code>Y = A ** B</code>	<code>\$pow</code>
<code>Y = A B</code>	<code>\$or</code>	<code>Y = A < B</code>	<code>\$lt</code>
<code>Y = A ^ B</code>	<code>\$xor</code>	<code>Y = A <= B</code>	<code>\$le</code>
<code>Y = A ~^ B</code>	<code>\$xnor</code>	<code>Y = A == B</code>	<code>\$eq</code>
<code>Y = A << B</code>	<code>\$shl</code>	<code>Y = A != B</code>	<code>\$ne</code>
<code>Y = A >> B</code>	<code>\$shr</code>	<code>Y = A >= B</code>	<code>\$ge</code>
<code>Y = A <<< B</code>	<code>\$sshl</code>	<code>Y = A > B</code>	<code>\$gt</code>
<code>Y = A >>> B</code>	<code>\$sshr</code>	<code>Y = A + B</code>	<code>\$add</code>
<code>Y = A && B</code>	<code>\$logic_and</code>	<code>Y = A - B</code>	<code>\$sub</code>
<code>Y = A B</code>	<code>\$logic_or</code>	<code>Y = A * B</code>	<code>\$mul</code>
<code>Y = A === B</code>	<code>\$eqx</code>	<code>Y = A / B</code>	<code>\$div</code>
<code>Y = A !== B</code>	<code>\$nex</code>	<code>Y = A % B</code>	<code>\$mod</code>
N/A	<code>\$shift</code>	N/A	<code>\$divfloor</code>
N/A	<code>\$shiftx</code>	N/A	<code>\$modfloor</code>

The `$shl` and `$shr` cells implement logical shifts, whereas the `$sshl` and `$sshr` cells implement arithmetic shifts. The `$shl` and `$sshl` cells implement the same operation. All four of these cells interpret the second operand as unsigned, and require `B_SIGNED` to be zero.

Two additional shift operator cells are available that do not directly correspond to any operator in Verilog, `$shift` and `$shiftx`. The `$shift` cell performs a right logical shift if the second operand is positive (or unsigned), and a left logical shift if it is negative. The `$shiftx` cell performs the same operation as the `$shift` cell, but the vacated bit positions are filled with undef (x) bits, and corresponds to the Verilog indexed part-select expression.

For the binary cells that output a logical value (`$logic_and`, `$logic_or`, `$eqx`, `$nex`, `$lt`, `$le`, `$eq`, `$ne`, `$ge`, `$gt`), when the `Y_WIDTH` parameter is greater than 1, the output is zero-extended, and only the least significant bit varies.

Division and modulo cells are available in two rounding modes. The original `$div` and `$mod` cells are based on truncating division, and correspond to the semantics of the verilog `/` and `%` operators. The `$divfloor` and `$modfloor` cells represent flooring division and flooring modulo, the latter of which corresponds to the `%` operator in Python. See the following table for a side-by-side comparison between the different semantics.

Table 9.3: Comparison between different rounding modes for division and modulo cells.

Division	Result	Truncating		Flooring	
		<code>\$div</code>	<code>\$mod</code>	<code>\$divfloor</code>	<code>\$modfloor</code>
<code>-10 / 3</code>	<code>-3.3</code>	<code>-3</code>	<code>-1</code>	<code>-4</code>	<code>2</code>
<code>10 / -3</code>	<code>-3.3</code>	<code>-3</code>	<code>1</code>	<code>-4</code>	<code>-2</code>
<code>-10 / -3</code>	<code>3.3</code>	<code>3</code>	<code>-1</code>	<code>3</code>	<code>-1</code>
<code>10 / 3</code>	<code>3.3</code>	<code>3</code>	<code>1</code>	<code>3</code>	<code>1</code>

```
yosys> help $add
```

Addition of inputs ‘A’ and ‘B’. This corresponds to the Verilog ‘+’ operator.

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.11: simlib.v

```

973 module \${add} (A, B, Y);
974
975     parameter A_SIGNED = 0;
976     parameter B_SIGNED = 0;
977     parameter A_WIDTH = 0;
978     parameter B_WIDTH = 0;
979     parameter Y_WIDTH = 0;
980
981     input [A_WIDTH-1:0] A;
982     input [B_WIDTH-1:0] B;
983     output [Y_WIDTH-1:0] Y;
984
985     generate
986         if (A_SIGNED && B_SIGNED) begin:BLOCK1
987             assign Y = $signed(A) + $signed(B);
988         end else begin:BLOCK2
989             assign Y = A + B;
990         end
991     endgenerate
992
993 endmodule

```

yosys> help \$and

A bit-wise AND. This corresponds to the Verilog '&' operator.

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.12: simlib.v

```

146 module \${and} (A, B, Y);
147
148     parameter A_SIGNED = 0;
149     parameter B_SIGNED = 0;
150     parameter A_WIDTH = 0;
151     parameter B_WIDTH = 0;
152     parameter Y_WIDTH = 0;
153
154     input [A_WIDTH-1:0] A;
155     input [B_WIDTH-1:0] B;
156     output [Y_WIDTH-1:0] Y;
157
158     generate
159         if (A_SIGNED && B_SIGNED) begin:BLOCK1
160             assign Y = $signed(A) & $signed(B);
161         end else begin:BLOCK2

```

(continues on next page)

(continued from previous page)

```

162         assign Y = A & B;
163     end
164 endgenerate
165
166 endmodule

```

yosys> help \$bweqx

Bit-wise case equality

A bit-wise version of *\$eqx*.

Properties

- *is_evaluable*
- *x-aware*

Simulation model (verilog)

Listing 9.13: simlib.v

```

1891 module \bweqx (A, B, Y);
1892
1893     parameter WIDTH = 0;
1894
1895     input [WIDTH-1:0] A, B;
1896     output [WIDTH-1:0] Y;
1897
1898     genvar i;
1899     generate
1900         for (i = 0; i < WIDTH; i = i + 1) begin:slices
1901             assign Y[i] = A[i] == B[i];
1902         end
1903     endgenerate
1904
1905 endmodule

```

yosys> help \$div

Divider

This corresponds to the Verilog `/` operator, performing division and truncating the result (rounding towards 0).

Properties

- *is_evaluable*
- *x-output*

Simulation model (verilog)

Listing 9.14: simlib.v

```

1218 module \div (A, B, Y);
1219
1220     parameter A_SIGNED = 0;
1221     parameter B_SIGNED = 0;

```

(continues on next page)

(continued from previous page)

```

1222     parameter A_WIDTH = 0;
1223     parameter B_WIDTH = 0;
1224     parameter Y_WIDTH = 0;
1225
1226     input [A_WIDTH-1:0] A;
1227     input [B_WIDTH-1:0] B;
1228     output [Y_WIDTH-1:0] Y;
1229
1230     generate
1231         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1232             assign Y = $signed(A) / $signed(B);
1233         end else begin:BLOCK2
1234             assign Y = A / B;
1235         end
1236     endgenerate
1237
1238 endmodule

```

yosys> help \$divfloor

Division with floored result (rounded towards negative infinity).

Properties

is_evaluable

Simulation model (verilog)

Listing 9.15: simlib.v

```

1281 module \divfloor (A, B, Y);
1282
1283     parameter A_SIGNED = 0;
1284     parameter B_SIGNED = 0;
1285     parameter A_WIDTH = 0;
1286     parameter B_WIDTH = 0;
1287     parameter Y_WIDTH = 0;
1288
1289     input [A_WIDTH-1:0] A;
1290     input [B_WIDTH-1:0] B;
1291     output [Y_WIDTH-1:0] Y;
1292
1293     generate
1294         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1295             localparam WIDTH =
1296                 A_WIDTH >= B_WIDTH && A_WIDTH >= Y_WIDTH ? A_WIDTH :
1297                 B_WIDTH >= A_WIDTH && B_WIDTH >= Y_WIDTH ? B_WIDTH : Y_WIDTH;
1298             wire [WIDTH:0] A_buf, B_buf, N_buf;
1299             assign A_buf = $signed(A);
1300             assign B_buf = $signed(B);
1301             assign N_buf = (A[A_WIDTH-1] == B[B_WIDTH-1]) || A == 0 ? A_buf :
1302             ↪ $signed(A_buf - (B[B_WIDTH-1] ? B_buf+1 : B_buf-1));
1303             assign Y = $signed(N_buf) / $signed(B_buf);
1304         end else begin:BLOCK2
1305             assign Y = A / B;
1306         end
1307     endgenerate
1308 endmodule

```

(continues on next page)

(continued from previous page)

```

1305     end
1306   endgenerate
1307
1308 endmodule

```

yosys> help \$eq

An equality comparison between inputs 'A' and 'B'. This corresponds to the Verilog '==' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.16: simlib.v

```

782 module \seq (A, B, Y);
783
784   parameter A_SIGNED = 0;
785   parameter B_SIGNED = 0;
786   parameter A_WIDTH = 0;
787   parameter B_WIDTH = 0;
788   parameter Y_WIDTH = 0;
789
790   input [A_WIDTH-1:0] A;
791   input [B_WIDTH-1:0] B;
792   output [Y_WIDTH-1:0] Y;
793
794   generate
795     if (A_SIGNED && B_SIGNED) begin:BLOCK1
796       assign Y = $signed(A) == $signed(B);
797     end else begin:BLOCK2
798       assign Y = A == B;
799     end
800   endgenerate
801
802 endmodule

```

yosys> help \$eqx

Case equality

An exact equality comparison between inputs 'A' and 'B'. Also known as the case equality operator. This corresponds to the Verilog '===' operator. Unlike equality comparison that can give 'x' as output, an exact equality comparison will strictly give '0' or '1' as output, even if input includes 'x' or 'z' values.

Properties

- *is_evaluable*
- *x-aware*

Simulation model (verilog)

Listing 9.17: simlib.v

```

847 module \${eqx} (A, B, Y);
848
849     parameter A_SIGNED = 0;
850     parameter B_SIGNED = 0;
851     parameter A_WIDTH = 0;
852     parameter B_WIDTH = 0;
853     parameter Y_WIDTH = 0;
854
855     input [A_WIDTH-1:0] A;
856     input [B_WIDTH-1:0] B;
857     output [Y_WIDTH-1:0] Y;
858
859     generate
860         if (A_SIGNED && B_SIGNED) begin:BLOCK1
861             assign Y = $signed(A) == $signed(B);
862         end else begin:BLOCK2
863             assign Y = A == B;
864         end
865     endgenerate
866
867 endmodule

```

yosys> help \$ge

A greater-than-or-equal-to comparison between inputs 'A' and 'B'. This corresponds to the Verilog '>=' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.18: simlib.v

```

910 module \${ge} (A, B, Y);
911
912     parameter A_SIGNED = 0;
913     parameter B_SIGNED = 0;
914     parameter A_WIDTH = 0;
915     parameter B_WIDTH = 0;
916     parameter Y_WIDTH = 0;
917
918     input [A_WIDTH-1:0] A;
919     input [B_WIDTH-1:0] B;
920     output [Y_WIDTH-1:0] Y;
921
922     generate
923         if (A_SIGNED && B_SIGNED) begin:BLOCK1
924             assign Y = $signed(A) >= $signed(B);
925         end else begin:BLOCK2
926             assign Y = A >= B;
927         end
928     endgenerate

```

(continues on next page)

(continued from previous page)

```

929
930 endmodule

```

yosys> help \$gt

A greater-than comparison between inputs 'A' and 'B'. This corresponds to the Verilog '>' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.19: simlib.v

```

942 module \ $gt (A, B, Y);
943
944     parameter A_SIGNED = 0;
945     parameter B_SIGNED = 0;
946     parameter A_WIDTH = 0;
947     parameter B_WIDTH = 0;
948     parameter Y_WIDTH = 0;
949
950     input [A_WIDTH-1:0] A;
951     input [B_WIDTH-1:0] B;
952     output [Y_WIDTH-1:0] Y;
953
954     generate
955         if (A_SIGNED && B_SIGNED) begin:BLOCK1
956             assign Y = $signed(A) > $signed(B);
957         end else begin:BLOCK2
958             assign Y = A > B;
959         end
960     endgenerate
961
962 endmodule

```

yosys> help \$le

A less-than-or-equal-to comparison between inputs 'A' and 'B'. This corresponds to the Verilog '<=' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.20: simlib.v

```

750 module \ $le (A, B, Y);
751
752     parameter A_SIGNED = 0;
753     parameter B_SIGNED = 0;
754     parameter A_WIDTH = 0;
755     parameter B_WIDTH = 0;
756     parameter Y_WIDTH = 0;
757

```

(continues on next page)

(continued from previous page)

```

758   input [A_WIDTH-1:0] A;
759   input [B_WIDTH-1:0] B;
760   output [Y_WIDTH-1:0] Y;
761
762   generate
763       if (A_SIGNED && B_SIGNED) begin:BLOCK1
764           assign Y = $signed(A) <= $signed(B);
765       end else begin:BLOCK2
766           assign Y = A <= B;
767       end
768   endgenerate
769
770 endmodule

```

yosys> help \$logic_and

A logical AND. This corresponds to the Verilog '&&' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.21: simlib.v

```

1427 module \ $logic_and (A, B, Y);
1428
1429     parameter A_SIGNED = 0;
1430     parameter B_SIGNED = 0;
1431     parameter A_WIDTH = 0;
1432     parameter B_WIDTH = 0;
1433     parameter Y_WIDTH = 0;
1434
1435     input [A_WIDTH-1:0] A;
1436     input [B_WIDTH-1:0] B;
1437     output [Y_WIDTH-1:0] Y;
1438
1439     generate
1440         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1441             assign Y = $signed(A) && $signed(B);
1442         end else begin:BLOCK2
1443             assign Y = A && B;
1444         end
1445     endgenerate
1446
1447 endmodule

```

yosys> help \$logic_or

A logical OR. This corresponds to the Verilog '||' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.22: simlib.v

```

1458 module \ $logic_or (A, B, Y);
1459
1460     parameter A_SIGNED = 0;
1461     parameter B_SIGNED = 0;
1462     parameter A_WIDTH = 0;
1463     parameter B_WIDTH = 0;
1464     parameter Y_WIDTH = 0;
1465
1466     input [A_WIDTH-1:0] A;
1467     input [B_WIDTH-1:0] B;
1468     output [Y_WIDTH-1:0] Y;
1469
1470     generate
1471         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1472             assign Y = $signed(A) || $signed(B);
1473         end else begin:BLOCK2
1474             assign Y = A || B;
1475         end
1476     endgenerate
1477
1478 endmodule

```

yosys> help \$lt

A less-than comparison between inputs 'A' and 'B'. This corresponds to the Verilog '<' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.23: simlib.v

```

718 module \ $lt (A, B, Y);
719
720     parameter A_SIGNED = 0;
721     parameter B_SIGNED = 0;
722     parameter A_WIDTH = 0;
723     parameter B_WIDTH = 0;
724     parameter Y_WIDTH = 0;
725
726     input [A_WIDTH-1:0] A;
727     input [B_WIDTH-1:0] B;
728     output [Y_WIDTH-1:0] Y;
729
730     generate
731         if (A_SIGNED && B_SIGNED) begin:BLOCK1
732             assign Y = $signed(A) < $signed(B);
733         end else begin:BLOCK2
734             assign Y = A < B;
735         end
736     endgenerate
737

```

(continues on next page)

(continued from previous page)

738 `endmodule``yosys> help $mod`**Modulo**

This corresponds to the Verilog ‘%’ operator, giving the module (or remainder) of division and truncating the result (rounding towards 0).

Invariant: $\$div(A, B) * B + \$mod(A, B) == A$

Properties

- *is_evaluable*
- *x-output*

Simulation model (verilog)

Listing 9.24: simlib.v

```

1250 module \ $mod (A, B, Y);
1251
1252     parameter A_SIGNED = 0;
1253     parameter B_SIGNED = 0;
1254     parameter A_WIDTH = 0;
1255     parameter B_WIDTH = 0;
1256     parameter Y_WIDTH = 0;
1257
1258     input [A_WIDTH-1:0] A;
1259     input [B_WIDTH-1:0] B;
1260     output [Y_WIDTH-1:0] Y;
1261
1262     generate
1263         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1264             assign Y = $signed(A) % $signed(B);
1265         end else begin:BLOCK2
1266             assign Y = A % B;
1267         end
1268     endgenerate
1269
1270 endmodule

```

`yosys> help $modfloor`

Modulo/remainder of division with floored result (rounded towards negative infinity).

Invariant: $\$divfloor(A, B) * B + \$modfloor(A, B) == A$

Properties

- *is_evaluable*

Simulation model (verilog)

Listing 9.25: simlib.v

```

1321 module \ $modfloor (A, B, Y);
1322

```

(continues on next page)

(continued from previous page)

```

1323     parameter A_SIGNED = 0;
1324     parameter B_SIGNED = 0;
1325     parameter A_WIDTH = 0;
1326     parameter B_WIDTH = 0;
1327     parameter Y_WIDTH = 0;
1328
1329     input [A_WIDTH-1:0] A;
1330     input [B_WIDTH-1:0] B;
1331     output [Y_WIDTH-1:0] Y;
1332
1333     generate
1334         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1335             localparam WIDTH = B_WIDTH >= Y_WIDTH ? B_WIDTH : Y_WIDTH;
1336             wire [WIDTH-1:0] B_buf, Y_trunc;
1337             assign B_buf = $signed(B);
1338             assign Y_trunc = $signed(A) % $signed(B);
1339             // flooring mod is the same as truncating mod for positive division
1340             → results (A and B have
1341                 // the same sign), as well as when there's no remainder.
1342                 // For all other cases, they behave as `floor - trunc = B`
1343             assign Y = (A[A_WIDTH-1] == B[B_WIDTH-1]) || Y_trunc == 0 ? Y_trunc :
1344             → $signed(B_buf) + $signed(Y_trunc);
1345         end else begin:BLOCK2
1346             // no difference between truncating and flooring for unsigned
1347             assign Y = A % B;
1348         end
1349     endgenerate
1350 endmodule

```

yosys> help \$mul

Multiplication of inputs 'A' and 'B'. This corresponds to the Verilog '*' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.26: simlib.v

```

1037 module \ $mul (A, B, Y);
1038
1039     parameter A_SIGNED = 0;
1040     parameter B_SIGNED = 0;
1041     parameter A_WIDTH = 0;
1042     parameter B_WIDTH = 0;
1043     parameter Y_WIDTH = 0;
1044
1045     input [A_WIDTH-1:0] A;
1046     input [B_WIDTH-1:0] B;
1047     output [Y_WIDTH-1:0] Y;
1048
1049     generate

```

(continues on next page)

(continued from previous page)

```

1050     if (A_SIGNED && B_SIGNED) begin:BLOCK1
1051         assign Y = $signed(A) * $signed(B);
1052     end else begin:BLOCK2
1053         assign Y = A * B;
1054     end
1055 endgenerate
1056
1057 endmodule

```

yosys> help \$ne

An inequality comparison between inputs 'A' and 'B'. This corresponds to the Verilog '!=' operator.

Properties

is_evaluateable

Simulation model (verilog)

Listing 9.27: simlib.v

```

814 module \ $ne (A, B, Y);
815
816     parameter A_SIGNED = 0;
817     parameter B_SIGNED = 0;
818     parameter A_WIDTH = 0;
819     parameter B_WIDTH = 0;
820     parameter Y_WIDTH = 0;
821
822     input [A_WIDTH-1:0] A;
823     input [B_WIDTH-1:0] B;
824     output [Y_WIDTH-1:0] Y;
825
826     generate
827         if (A_SIGNED && B_SIGNED) begin:BLOCK1
828             assign Y = $signed(A) != $signed(B);
829         end else begin:BLOCK2
830             assign Y = A != B;
831         end
832     endgenerate
833
834 endmodule

```

yosys> help \$nex

Case inequality

This corresponds to the Verilog '!==' operator.

Refer to *\$eqx* for more details.

Properties

- *is_evaluateable*
- *x-aware*

Simulation model (verilog)

Listing 9.28: simlib.v

```

878 module \$_nex (A, B, Y);
879
880     parameter A_SIGNED = 0;
881     parameter B_SIGNED = 0;
882     parameter A_WIDTH = 0;
883     parameter B_WIDTH = 0;
884     parameter Y_WIDTH = 0;
885
886     input [A_WIDTH-1:0] A;
887     input [B_WIDTH-1:0] B;
888     output [Y_WIDTH-1:0] Y;
889
890     generate
891         if (A_SIGNED && B_SIGNED) begin:BLOCK1
892             assign Y = $signed(A) !== $signed(B);
893         end else begin:BLOCK2
894             assign Y = A !== B;
895         end
896     endgenerate
897
898 endmodule

```

yosys> help \$or

A bit-wise OR. This corresponds to the Verilog '|' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.29: simlib.v

```

177 module \$_or (A, B, Y);
178
179     parameter A_SIGNED = 0;
180     parameter B_SIGNED = 0;
181     parameter A_WIDTH = 0;
182     parameter B_WIDTH = 0;
183     parameter Y_WIDTH = 0;
184
185     input [A_WIDTH-1:0] A;
186     input [B_WIDTH-1:0] B;
187     output [Y_WIDTH-1:0] Y;
188
189     generate
190         if (A_SIGNED && B_SIGNED) begin:BLOCK1
191             assign Y = $signed(A) | $signed(B);
192         end else begin:BLOCK2
193             assign Y = A | B;
194         end
195     endgenerate
196

```

(continues on next page)

(continued from previous page)

```
197 endmodule
```

```
yosys> help $pow
```

Exponentiation of an input ($Y = A ** B$). This corresponds to the Verilog `**` operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.30: simlib.v

```
1363 module \ $pow (A, B, Y);
1364
1365     parameter A_SIGNED = 0;
1366     parameter B_SIGNED = 0;
1367     parameter A_WIDTH = 0;
1368     parameter B_WIDTH = 0;
1369     parameter Y_WIDTH = 0;
1370
1371     input [A_WIDTH-1:0] A;
1372     input [B_WIDTH-1:0] B;
1373     output [Y_WIDTH-1:0] Y;
1374
1375     generate
1376         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1377             assign Y = $signed(A) ** $signed(B);
1378         end else if (A_SIGNED) begin:BLOCK2
1379             assign Y = $signed(A) ** B;
1380         end else if (B_SIGNED) begin:BLOCK3
1381             assign Y = A ** $signed(B);
1382         end else begin:BLOCK4
1383             assign Y = A ** B;
1384         end
1385     endgenerate
1386
1387 endmodule
```

```
yosys> help $shift
```

Variable shifter

Performs a right logical shift if the second operand is positive (or unsigned), and a left logical shift if it is negative.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.31: simlib.v

```
535 module \ $shift (A, B, Y);
536
537     parameter A_SIGNED = 0;
```

(continues on next page)

(continued from previous page)

```

538 parameter B_SIGNED = 0;
539 parameter A_WIDTH = 0;
540 parameter B_WIDTH = 0;
541 parameter Y_WIDTH = 0;
542
543 input [A_WIDTH-1:0] A;
544 input [B_WIDTH-1:0] B;
545 output [Y_WIDTH-1:0] Y;
546
547 generate
548     if (A_SIGNED) begin:BLOCK1
549         if (B_SIGNED) begin:BLOCK2
550             assign Y = $signed(B) < 0 ? $signed(A) << -B : $signed(A) >> B;
551         end else begin:BLOCK3
552             assign Y = $signed(A) >> B;
553         end
554     end else begin:BLOCK4
555         if (B_SIGNED) begin:BLOCK5
556             assign Y = $signed(B) < 0 ? A << -B : A >> B;
557         end else begin:BLOCK6
558             assign Y = A >> B;
559         end
560     end
561 endgenerate
562
563 endmodule

```

yosys> help \$shiftx

Indexed part-select

Same as the *\$shift* cell, but fills with 'x'.

Properties

- *is_evaluable*
- *x-output*

Simulation model (verilog)

Listing 9.32: simlib.v

```

572 module \ $shiftx (A, B, Y);
573
574     parameter A_SIGNED = 0;
575     parameter B_SIGNED = 0;
576     parameter A_WIDTH = 0;
577     parameter B_WIDTH = 0;
578     parameter Y_WIDTH = 0;
579
580     input [A_WIDTH-1:0] A;
581     input [B_WIDTH-1:0] B;
582     output [Y_WIDTH-1:0] Y;
583

```

(continues on next page)

(continued from previous page)

```

584     generate
585         if (Y_WIDTH > 0)
586             if (B_SIGNED) begin:BLOCK1
587                 assign Y = A[$signed(B) +: Y_WIDTH];
588             end else begin:BLOCK2
589                 assign Y = A[B +: Y_WIDTH];
590             end
591         endgenerate
592
593     endmodule

```

yosys> help \$shl

A logical shift-left operation. This corresponds to the Verilog '<<' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.33: simlib.v

```

411 module \ $shl (A, B, Y);
412
413     parameter A_SIGNED = 0;
414     parameter B_SIGNED = 0;
415     parameter A_WIDTH = 0;
416     parameter B_WIDTH = 0;
417     parameter Y_WIDTH = 0;
418
419     input [A_WIDTH-1:0] A;
420     input [B_WIDTH-1:0] B;
421     output [Y_WIDTH-1:0] Y;
422
423     generate
424         if (A_SIGNED) begin:BLOCK1
425             assign Y = $signed(A) << B;
426         end else begin:BLOCK2
427             assign Y = A << B;
428         end
429     endgenerate
430
431 endmodule

```

yosys> help \$shr

A logical shift-right operation. This corresponds to the Verilog '>>' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.34: simlib.v

```

442 module \$$shr (A, B, Y);
443
444     parameter A_SIGNED = 0;
445     parameter B_SIGNED = 0;
446     parameter A_WIDTH = 0;
447     parameter B_WIDTH = 0;
448     parameter Y_WIDTH = 0;
449
450     input [A_WIDTH-1:0] A;
451     input [B_WIDTH-1:0] B;
452     output [Y_WIDTH-1:0] Y;
453
454     generate
455         if (A_SIGNED) begin:BLOCK1
456             assign Y = $signed(A) >> B;
457         end else begin:BLOCK2
458             assign Y = A >> B;
459         end
460     endgenerate
461
462 endmodule

```

yosys> help \$sshl

An arithmetic shift-left operation. This corresponds to the Verilog '<<<-' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.35: simlib.v

```

474 module \$$sshl (A, B, Y);
475
476     parameter A_SIGNED = 0;
477     parameter B_SIGNED = 0;
478     parameter A_WIDTH = 0;
479     parameter B_WIDTH = 0;
480     parameter Y_WIDTH = 0;
481
482     input [A_WIDTH-1:0] A;
483     input [B_WIDTH-1:0] B;
484     output [Y_WIDTH-1:0] Y;
485
486     generate
487         if (A_SIGNED) begin:BLOCK1
488             assign Y = $signed(A) <<< B;
489         end else begin:BLOCK2
490             assign Y = A <<< B;
491         end
492     endgenerate
493

```

(continues on next page)

(continued from previous page)

```
494 endmodule
```

```
yosys> help $sshr
```

An arithmetic shift-right operation. This corresponds to the Verilog '>>>' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.36: simlib.v

```
506 module \sshr (A, B, Y);
507
508     parameter A_SIGNED = 0;
509     parameter B_SIGNED = 0;
510     parameter A_WIDTH = 0;
511     parameter B_WIDTH = 0;
512     parameter Y_WIDTH = 0;
513
514     input [A_WIDTH-1:0] A;
515     input [B_WIDTH-1:0] B;
516     output [Y_WIDTH-1:0] Y;
517
518     generate
519         if (A_SIGNED) begin:BLOCK1
520             assign Y = $signed(A) >>> B;
521         end else begin:BLOCK2
522             assign Y = A >>> B;
523         end
524     endgenerate
525
526 endmodule
```

```
yosys> help $sub
```

Subtraction between inputs 'A' and 'B'. This corresponds to the Verilog '-' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.37: simlib.v

```
1005 module \sub (A, B, Y);
1006
1007     parameter A_SIGNED = 0;
1008     parameter B_SIGNED = 0;
1009     parameter A_WIDTH = 0;
1010     parameter B_WIDTH = 0;
1011     parameter Y_WIDTH = 0;
1012
1013     input [A_WIDTH-1:0] A;
1014     input [B_WIDTH-1:0] B;
```

(continues on next page)

(continued from previous page)

```

1015     output [Y_WIDTH-1:0] Y;
1016
1017     generate
1018         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1019             assign Y = $signed(A) - $signed(B);
1020         end else begin:BLOCK2
1021             assign Y = A - B;
1022         end
1023     endgenerate
1024
1025 endmodule

```

yosys> help \$xnor

A bit-wise XNOR. This corresponds to the Verilog ‘`~^`’ operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.38: simlib.v

```

239 module \ $xnor (A, B, Y);
240
241     parameter A_SIGNED = 0;
242     parameter B_SIGNED = 0;
243     parameter A_WIDTH = 0;
244     parameter B_WIDTH = 0;
245     parameter Y_WIDTH = 0;
246
247     input [A_WIDTH-1:0] A;
248     input [B_WIDTH-1:0] B;
249     output [Y_WIDTH-1:0] Y;
250
251     generate
252         if (A_SIGNED && B_SIGNED) begin:BLOCK1
253             assign Y = $signed(A) ~^ $signed(B);
254         end else begin:BLOCK2
255             assign Y = A ~^ B;
256         end
257     endgenerate
258
259 endmodule

```

yosys> help \$xor

A bit-wise XOR. This corresponds to the Verilog ‘`^`’ operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.39: simlib.v

```

208 module \${xor} (A, B, Y);
209
210     parameter A_SIGNED = 0;
211     parameter B_SIGNED = 0;
212     parameter A_WIDTH = 0;
213     parameter B_WIDTH = 0;
214     parameter Y_WIDTH = 0;
215
216     input [A_WIDTH-1:0] A;
217     input [B_WIDTH-1:0] B;
218     output [Y_WIDTH-1:0] Y;
219
220     generate
221         if (A_SIGNED && B_SIGNED) begin:BLOCK1
222             assign Y = $signed(A) ^ $signed(B);
223         end else begin:BLOCK2
224             assign Y = A ^ B;
225         end
226     endgenerate
227
228 endmodule

```

9.1.3 Multiplexers

Multiplexers are generated by the Verilog HDL frontend for `?:`-expressions. Multiplexers are also generated by the proc pass to map the decision trees from RTLIL::Process objects to logic.

The simplest multiplexer cell type is `$mux`. Cells of this type have a `WIDTH` parameter and data inputs `A` and `B` and a data output `Y`, all of the specified width. This cell also has a single bit control input `S`. If `S` is 0 the value from the input `A` is sent to the output, if it is 1 the value from the `B` input is sent to the output. So the `$mux` cell implements the function $Y = S ? B : A$.

The `$pmux` cell is used to multiplex between many inputs using a one-hot select signal. Cells of this type have a `WIDTH` and a `S_WIDTH` parameter and inputs `A`, `B`, and `S` and an output `Y`. The `S` input is `S_WIDTH` bits wide. The `A` input and the output are both `WIDTH` bits wide and the `B` input is `WIDTH*S_WIDTH` bits wide. When all bits of `S` are zero, the value from `A` input is sent to the output. If the n 'th bit from `S` is set, the value n 'th `WIDTH` bits wide slice of the `B` input is sent to the output. When more than one bit from `S` is set the output is undefined. Cells of this type are used to model “parallel cases” (defined by using the `parallel_case` attribute or detected by an optimization).

The `$tribuf` cell is used to implement tristate logic. Cells of this type have a `WIDTH` parameter and inputs `A` and `EN` and an output `Y`. The `A` input and `Y` output are `WIDTH` bits wide, and the `EN` input is one bit wide. When `EN` is 0, the output is not driven. When `EN` is 1, the value from `A` input is sent to the `Y` output. Therefore, the `$tribuf` cell implements the function $Y = EN ? A : 'bz$.

Behavioural code with cascaded if-then-else- and case-statements usually results in trees of multiplexer cells. Many passes (from various optimizations to FSM extraction) heavily depend on these multiplexer trees to understand dependencies between signals. Therefore optimizations should not break these multiplexer trees (e.g. by replacing a multiplexer between a calculated signal and a constant zero with an `$and` gate).

yosys> help \$bmux

Binary-encoded multiplexer

Selects between ‘slices’ of A where each value of S corresponds to a unique slice.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.40: simlib.v

```

1544 module \bmux (A, S, Y);
1545
1546     parameter WIDTH = 0;
1547     parameter S_WIDTH = 0;
1548
1549     input [(WIDTH << S_WIDTH)-1:0] A;
1550     input [S_WIDTH-1:0] S;
1551     output [WIDTH-1:0] Y;
1552
1553     wire [WIDTH-1:0] bm0_out, bm1_out;
1554
1555     generate
1556         if (S_WIDTH > 1) begin:muxlogic
1557             \bmux #(.WIDTH(WIDTH), .S_WIDTH(S_WIDTH-1)) bm0 (.A(A[(WIDTH << (S_
1558 ↪ WIDTH - 1))-1:0]), .S(S[S_WIDTH-2:0]), .Y(bm0_out));
1559             \bmux #(.WIDTH(WIDTH), .S_WIDTH(S_WIDTH-1)) bm1 (.A(A[(WIDTH << S_
1560 ↪ WIDTH)-1:WIDTH << (S_WIDTH - 1)]), .S(S[S_WIDTH-2:0]), .Y(bm1_out));
1561             assign Y = S[S_WIDTH-1] ? bm1_out : bm0_out;
1562         end else if (S_WIDTH == 1) begin:simple
1563             assign Y = S ? A[2*WIDTH-1:WIDTH] : A[WIDTH-1:0];
1564         end else begin:passthru
1565             assign Y = A;
1566         end
1567     endgenerate
1568 endmodule

```

yosys> help \$bwmux

Bit-wise multiplexer

Equivalent to a series of 1-bit wide *\$mux* cells.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.41: simlib.v

```

1913 module \bwmux (A, B, S, Y);
1914
1915     parameter WIDTH = 0;
1916
1917     input [WIDTH-1:0] A, B;

```

(continues on next page)

(continued from previous page)

```

1918     input [WIDTH-1:0] S;
1919     output [WIDTH-1:0] Y;
1920
1921     genvar i;
1922     generate
1923         for (i = 0; i < WIDTH; i = i + 1) begin:slices
1924             assign Y[i] = S[i] ? B[i] : A[i];
1925         end
1926     endgenerate
1927
1928 endmodule

```

yosys> help \$demux

Demultiplexer i.e routing single input to several outputs based on select signal. Unselected outputs are driven to zero.

Properties

is_evaluateable

Simulation model (verilog)

Listing 9.42: simlib.v

```

1619 module \demux (A, S, Y);
1620
1621     parameter WIDTH = 1;
1622     parameter S_WIDTH = 1;
1623
1624     input [WIDTH-1:0] A;
1625     input [S_WIDTH-1:0] S;
1626     output [(WIDTH << S_WIDTH)-1:0] Y;
1627
1628     genvar i;
1629     generate
1630         for (i = 0; i < (1 << S_WIDTH); i = i + 1) begin:slices
1631             assign Y[i*WIDTH+:WIDTH] = (S == i) ? A : 0;
1632         end
1633     endgenerate
1634
1635 endmodule

```

yosys> help \$mux

Multiplexer i.e selecting between two inputs based on select signal.

Properties

is_evaluateable

Simulation model (verilog)

Listing 9.43: simlib.v

```

1525 module \mux (A, B, S, Y);
1526
1527     parameter WIDTH = 0;

```

(continues on next page)

(continued from previous page)

```

1528
1529     input [WIDTH-1:0] A, B;
1530     input S;
1531     output [WIDTH-1:0] Y;
1532
1533     assign Y = S ? B : A;
1534
1535 endmodule

```

yosys> help \$pmux

Priority-encoded multiplexer

Selects between ‘slices’ of B where each slice corresponds to a single bit of S. Outputs A when all bits of S are low.

Properties

- *is_evaluable*
- *x-output*

Simulation model (verilog)

Listing 9.44: simlib.v

```

1577 module \ $pmux (A, B, S, Y);
1578
1579     parameter WIDTH = 0;
1580     parameter S_WIDTH = 0;
1581
1582     input [WIDTH-1:0] A;
1583     input [WIDTH*S_WIDTH-1:0] B;
1584     input [S_WIDTH-1:0] S;
1585     output reg [WIDTH-1:0] Y;
1586
1587     integer i;
1588     reg found_active_sel_bit;
1589
1590     always @* begin
1591         Y = A;
1592         found_active_sel_bit = 0;
1593         for (i = 0; i < S_WIDTH; i = i+1)
1594             case (S[i])
1595                 1'b1: begin
1596                     Y = found_active_sel_bit ? 'bx : B >> (WIDTH*i);
1597                     found_active_sel_bit = 1;
1598                 end
1599                 1'b0: ;
1600                 1'bx: begin
1601                     Y = 'bx;
1602                     found_active_sel_bit = 'bx;
1603                 end
1604             endcase
1605     end

```

(continues on next page)

(continued from previous page)

```

1606
1607 endmodule

```

yosys> help \$tribuf

A tri-state buffer. This buffer conditionally drives the output with the value of the input based on the enable signal.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.45: simlib.v

```

1694 module \ $tribuf (A, EN, Y);
1695
1696     parameter WIDTH = 0;
1697
1698     input [WIDTH-1:0] A;
1699     input EN;
1700     output [WIDTH-1:0] Y;
1701
1702     assign Y = EN ? A : 'bz;
1703
1704 endmodule

```

9.1.4 Registers

SR-type latches are represented by *\$sr* cells. These cells have input ports SET and CLR and an output port Q. They have the following parameters:

WIDTH

The width of inputs SET and CLR and output Q.

SET_POLARITY

The set input bits are active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

CLR_POLARITY

The reset input bits are active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

Both set and reset inputs have separate bits for every output bit. When both the set and reset inputs of an *\$sr* cell are active for a given bit index, the reset input takes precedence.

D-type flip-flops are represented by *\$dff* cells. These cells have a clock port CLK, an input port D and an output port Q. The following parameters are available for *\$dff* cells:

WIDTH

The width of input D and output Q.

CLK_POLARITY

Clock is active on the positive edge if this parameter has the value 1'b1 and on the negative edge if this parameter is 1'b0.

D-type flip-flops with asynchronous reset are represented by *\$adff* cells. As the *\$dff* cells they have CLK, D and Q ports. In addition they also have a single-bit ARST input port for the reset pin and the following additional two parameters:

ARST_POLARITY

The asynchronous reset is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

ARST_VALUE

The state of Q will be set to this value when the reset is active.

Usually these cells are generated by the *proc* pass using the information in the designs RTLIL::Process objects.

D-type flip-flops with synchronous reset are represented by *\$sdff* cells. As the *\$dff* cells they have CLK, D and Q ports. In addition they also have a single-bit SRST input port for the reset pin and the following additional two parameters:

SRST_POLARITY

The synchronous reset is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

SRST_VALUE

The state of Q will be set to this value when the reset is active.

Note that the *\$adff* and *\$sdff* cells can only be used when the reset value is constant.

D-type flip-flops with asynchronous load are represented by *\$aldff* cells. As the *\$dff* cells they have CLK, D and Q ports. In addition they also have a single-bit ALOAD input port for the async load enable pin, a AD input port with the same width as data for the async load data, and the following additional parameter:

ALOAD_POLARITY

The asynchronous load is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

D-type flip-flops with asynchronous set and reset are represented by *\$dffsr* cells. As the *\$dff* cells they have CLK, D and Q ports. In addition they also have multi-bit SET and CLR input ports and the corresponding polarity parameters, like *\$sr* cells.

D-type flip-flops with enable are represented by *\$dffe*, *\$adffe*, *\$aldffe*, *\$dffsre*, *\$sdffe*, and *\$sdffce* cells, which are enhanced variants of *\$dff*, *\$adff*, *\$aldff*, *\$dffsr*, *\$sdff* (with reset over enable) and *\$sdff* (with enable over reset) cells, respectively. They have the same ports and parameters as their base cell. In addition they also have a single-bit EN input port for the enable pin and the following parameter:

EN_POLARITY

The enable input is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

D-type latches are represented by *\$dlatch* cells. These cells have an enable port EN, an input port D, and an output port Q. The following parameters are available for *\$dlatch* cells:

WIDTH

The width of input D and output Q.

EN_POLARITY

The enable input is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

The latch is transparent when the EN input is active.

D-type latches with reset are represented by *\$adlatch* cells. In addition to *\$dlatch* ports and parameters, they also have a single-bit ARST input port for the reset pin and the following additional parameters:

ARST_POLARITY

The asynchronous reset is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

ARST_VALUE

The state of Q will be set to this value when the reset is active.

D-type latches with set and reset are represented by *\$dlatchsr* cells. In addition to *\$dlatch* ports and parameters, they also have multi-bit SET and CLR input ports and the corresponding polarity parameters, like *\$sr* cells.

```
yosys> help $adff
```

```
Simulation model (verilog)
```

Listing 9.46: simlib.v

```

2309 module \adff (CLK, ARST, D, Q);
2310
2311     parameter WIDTH = 0;
2312     parameter CLK_POLARITY = 1'b1;
2313     parameter ARST_POLARITY = 1'b1;
2314     parameter ARST_VALUE = 0;
2315
2316     input CLK, ARST;
2317     input [WIDTH-1:0] D;
2318     output reg [WIDTH-1:0] Q;
2319     wire pos_clk = CLK == CLK_POLARITY;
2320     wire pos_arst = ARST == ARST_POLARITY;
2321
2322     always @(posedge pos_clk, posedge pos_arst) begin
2323         if (pos_arst)
2324             Q <= ARST_VALUE;
2325         else
2326             Q <= D;
2327     end
2328
2329 endmodule

```

```
yosys> help $adffe
```

```
Simulation model (verilog)
```

Listing 9.47: simlib.v

```

2384 module \adffe (CLK, ARST, EN, D, Q);
2385
2386     parameter WIDTH = 0;
2387     parameter CLK_POLARITY = 1'b1;
2388     parameter EN_POLARITY = 1'b1;
2389     parameter ARST_POLARITY = 1'b1;
2390     parameter ARST_VALUE = 0;
2391
2392     input CLK, ARST, EN;

```

(continues on next page)

(continued from previous page)

```

2393     input [WIDTH-1:0] D;
2394     output reg [WIDTH-1:0] Q;
2395     wire pos_clk = CLK == CLK_POLARITY;
2396     wire pos_arst = ARST == ARST_POLARITY;
2397
2398     always @(posedge pos_clk, posedge pos_arst) begin
2399         if (pos_arst)
2400             Q <= ARST_VALUE;
2401         else if (EN == EN_POLARITY)
2402             Q <= D;
2403     end
2404
2405 endmodule

```

yosys> help \$adlatch

Simulation model (verilog)

Listing 9.48: simlib.v

```

2509 module \adlatch (EN, ARST, D, Q);
2510
2511     parameter WIDTH = 0;
2512     parameter EN_POLARITY = 1'b1;
2513     parameter ARST_POLARITY = 1'b1;
2514     parameter ARST_VALUE = 0;
2515
2516     input EN, ARST;
2517     input [WIDTH-1:0] D;
2518     output reg [WIDTH-1:0] Q;
2519
2520     always @* begin
2521         if (ARST == ARST_POLARITY)
2522             Q = ARST_VALUE;
2523         else if (EN == EN_POLARITY)
2524             Q = D;
2525     end
2526
2527 endmodule

```

yosys> help \$aldff

Simulation model (verilog)

Listing 9.49: simlib.v

```

2334 module \aldff (CLK, ALOAD, AD, D, Q);
2335
2336     parameter WIDTH = 0;
2337     parameter CLK_POLARITY = 1'b1;
2338     parameter ALOAD_POLARITY = 1'b1;
2339
2340     input CLK, ALOAD;

```

(continues on next page)

(continued from previous page)

```

2341     input [WIDTH-1:0] AD;
2342     input [WIDTH-1:0] D;
2343     output reg [WIDTH-1:0] Q;
2344     wire pos_clk = CLK == CLK_POLARITY;
2345     wire pos_aload = ALOAD == ALOAD_POLARITY;
2346
2347     always @(posedge pos_clk, posedge pos_aload) begin
2348         if (pos_aload)
2349             Q <= AD;
2350         else
2351             Q <= D;
2352     end
2353
2354 endmodule

```

yosys> help \$aldfife

Simulation model (verilog)

Listing 9.50: simlib.v

```

2410 module \ $aldfife (CLK, ALOAD, AD, EN, D, Q);
2411
2412     parameter WIDTH = 0;
2413     parameter CLK_POLARITY = 1'b1;
2414     parameter EN_POLARITY = 1'b1;
2415     parameter ALOAD_POLARITY = 1'b1;
2416
2417     input CLK, ALOAD, EN;
2418     input [WIDTH-1:0] D;
2419     input [WIDTH-1:0] AD;
2420     output reg [WIDTH-1:0] Q;
2421     wire pos_clk = CLK == CLK_POLARITY;
2422     wire pos_aload = ALOAD == ALOAD_POLARITY;
2423
2424     always @(posedge pos_clk, posedge pos_aload) begin
2425         if (pos_aload)
2426             Q <= AD;
2427         else if (EN == EN_POLARITY)
2428             Q <= D;
2429     end
2430
2431 endmodule

```

yosys> help \$dff

Simulation model (verilog)

Listing 9.51: simlib.v

```

2201 module \ $dff (CLK, D, Q);
2202
2203     parameter WIDTH = 0;

```

(continues on next page)

(continued from previous page)

```

2204     parameter CLK_POLARITY = 1'b1;
2205
2206     input CLK;
2207     input [WIDTH-1:0] D;
2208     output reg [WIDTH-1:0] Q;
2209     wire pos_clk = CLK == CLK_POLARITY;
2210
2211     always @(posedge pos_clk) begin
2212         Q <= D;
2213     end
2214
2215 endmodule

```

yosys> help \$dffe

Simulation model (verilog)

Listing 9.52: simlib.v

```

2220 module \dffe (CLK, EN, D, Q);
2221
2222     parameter WIDTH = 0;
2223     parameter CLK_POLARITY = 1'b1;
2224     parameter EN_POLARITY = 1'b1;
2225
2226     input CLK, EN;
2227     input [WIDTH-1:0] D;
2228     output reg [WIDTH-1:0] Q;
2229     wire pos_clk = CLK == CLK_POLARITY;
2230
2231     always @(posedge pos_clk) begin
2232         if (EN == EN_POLARITY) Q <= D;
2233     end
2234
2235 endmodule

```

yosys> help \$dffsr

Simulation model (verilog)

Listing 9.53: simlib.v

```

2241 module \dffsr (CLK, SET, CLR, D, Q);
2242
2243     parameter WIDTH = 0;
2244     parameter CLK_POLARITY = 1'b1;
2245     parameter SET_POLARITY = 1'b1;
2246     parameter CLR_POLARITY = 1'b1;
2247
2248     input CLK;
2249     input [WIDTH-1:0] SET, CLR, D;
2250     output reg [WIDTH-1:0] Q;
2251

```

(continues on next page)

(continued from previous page)

```

2252     wire pos_clk = CLK == CLK_POLARITY;
2253     wire [WIDTH-1:0] pos_set = SET_POLARITY ? SET : ~SET;
2254     wire [WIDTH-1:0] pos_clr = CLR_POLARITY ? CLR : ~CLR;
2255
2256     genvar i;
2257     generate
2258         for (i = 0; i < WIDTH; i = i+1) begin:bitslices
2259             always @(posedge pos_set[i], posedge pos_clr[i], posedge pos_clk)
2260                 if (pos_clr[i])
2261                     Q[i] <= 0;
2262                 else if (pos_set[i])
2263                     Q[i] <= 1;
2264                 else
2265                     Q[i] <= D[i];
2266         end
2267     endgenerate
2268
2269 endmodule

```

yosys> help \$dffsre

Simulation model (verilog)

Listing 9.54: simlib.v

```

2274 module \dffsre (CLK, SET, CLR, EN, D, Q);
2275
2276     parameter WIDTH = 0;
2277     parameter CLK_POLARITY = 1'b1;
2278     parameter SET_POLARITY = 1'b1;
2279     parameter CLR_POLARITY = 1'b1;
2280     parameter EN_POLARITY = 1'b1;
2281
2282     input CLK, EN;
2283     input [WIDTH-1:0] SET, CLR, D;
2284     output reg [WIDTH-1:0] Q;
2285
2286     wire pos_clk = CLK == CLK_POLARITY;
2287     wire [WIDTH-1:0] pos_set = SET_POLARITY ? SET : ~SET;
2288     wire [WIDTH-1:0] pos_clr = CLR_POLARITY ? CLR : ~CLR;
2289
2290     genvar i;
2291     generate
2292         for (i = 0; i < WIDTH; i = i+1) begin:bitslices
2293             always @(posedge pos_set[i], posedge pos_clr[i], posedge pos_clk)
2294                 if (pos_clr[i])
2295                     Q[i] <= 0;
2296                 else if (pos_set[i])
2297                     Q[i] <= 1;
2298                 else if (EN == EN_POLARITY)
2299                     Q[i] <= D[i];
2300         end

```

(continues on next page)

(continued from previous page)

```

2301     endgenerate
2302
2303 endmodule

```

```
yosys> help $dlatch
```

```
Simulation model (verilog)
```

Listing 9.55: simlib.v

```

2490 module \ $dlatch (EN, D, Q);
2491
2492     parameter WIDTH = 0;
2493     parameter EN_POLARITY = 1'b1;
2494
2495     input EN;
2496     input [WIDTH-1:0] D;
2497     output reg [WIDTH-1:0] Q;
2498
2499     always @* begin
2500         if (EN == EN_POLARITY)
2501             Q = D;
2502     end
2503
2504 endmodule

```

```
yosys> help $dlatchsr
```

```
Simulation model (verilog)
```

Listing 9.56: simlib.v

```

2533 module \ $dlatchsr (EN, SET, CLR, D, Q);
2534
2535     parameter WIDTH = 0;
2536     parameter EN_POLARITY = 1'b1;
2537     parameter SET_POLARITY = 1'b1;
2538     parameter CLR_POLARITY = 1'b1;
2539
2540     input EN;
2541     input [WIDTH-1:0] SET, CLR, D;
2542     output reg [WIDTH-1:0] Q;
2543
2544     wire pos_en = EN == EN_POLARITY;
2545     wire [WIDTH-1:0] pos_set = SET_POLARITY ? SET : ~SET;
2546     wire [WIDTH-1:0] pos_clr = CLR_POLARITY ? CLR : ~CLR;
2547
2548     genvar i;
2549     generate
2550         for (i = 0; i < WIDTH; i = i+1) begin:bitslices
2551             always @*
2552                 if (pos_clr[i])
2553                     Q[i] = 0;

```

(continues on next page)

(continued from previous page)

```

2554         else if (pos_set[i])
2555             Q[i] = 1;
2556         else if (pos_en)
2557             Q[i] = D[i];
2558     end
2559 endgenerate
2560
2561 endmodule

```

yosys> help \$sdff

Simulation model (verilog)

Listing 9.57: simlib.v

```

2359 module \ $sdff (CLK, SRST, D, Q);
2360
2361     parameter WIDTH = 0;
2362     parameter CLK_POLARITY = 1'b1;
2363     parameter SRST_POLARITY = 1'b1;
2364     parameter SRST_VALUE = 0;
2365
2366     input CLK, SRST;
2367     input [WIDTH-1:0] D;
2368     output reg [WIDTH-1:0] Q;
2369     wire pos_clk = CLK == CLK_POLARITY;
2370     wire pos_srst = SRST == SRST_POLARITY;
2371
2372     always @(posedge pos_clk) begin
2373         if (pos_srst)
2374             Q <= SRST_VALUE;
2375         else
2376             Q <= D;
2377     end
2378
2379 endmodule

```

yosys> help \$sdffce

Simulation model (verilog)

Listing 9.58: simlib.v

```

2462 module \ $sdffce (CLK, SRST, EN, D, Q);
2463
2464     parameter WIDTH = 0;
2465     parameter CLK_POLARITY = 1'b1;
2466     parameter EN_POLARITY = 1'b1;
2467     parameter SRST_POLARITY = 1'b1;
2468     parameter SRST_VALUE = 0;
2469
2470     input CLK, SRST, EN;
2471     input [WIDTH-1:0] D;

```

(continues on next page)

(continued from previous page)

```

2472     output reg [WIDTH-1:0] Q;
2473     wire pos_clk = CLK == CLK_POLARITY;
2474     wire pos_srst = SRST == SRST_POLARITY;
2475
2476     always @(posedge pos_clk) begin
2477         if (EN == EN_POLARITY) begin
2478             if (pos_srst)
2479                 Q <= SRST_VALUE;
2480             else
2481                 Q <= D;
2482         end
2483     end
2484
2485 endmodule

```

yosys> help \$sdffe

Simulation model (verilog)

Listing 9.59: simlib.v

```

2436 module \ $sdffe (CLK, SRST, EN, D, Q);
2437
2438     parameter WIDTH = 0;
2439     parameter CLK_POLARITY = 1'b1;
2440     parameter EN_POLARITY = 1'b1;
2441     parameter SRST_POLARITY = 1'b1;
2442     parameter SRST_VALUE = 0;
2443
2444     input CLK, SRST, EN;
2445     input [WIDTH-1:0] D;
2446     output reg [WIDTH-1:0] Q;
2447     wire pos_clk = CLK == CLK_POLARITY;
2448     wire pos_srst = SRST == SRST_POLARITY;
2449
2450     always @(posedge pos_clk) begin
2451         if (pos_srst)
2452             Q <= SRST_VALUE;
2453         else if (EN == EN_POLARITY)
2454             Q <= D;
2455     end
2456
2457 endmodule

```

yosys> help \$sr

Simulation model (verilog)

Listing 9.60: simlib.v

```

2151 module \ $sr (SET, CLR, Q);
2152
2153     parameter WIDTH = 0;

```

(continues on next page)

(continued from previous page)

```

2154     parameter SET_POLARITY = 1'b1;
2155     parameter CLR_POLARITY = 1'b1;
2156
2157     input [WIDTH-1:0] SET, CLR;
2158     output reg [WIDTH-1:0] Q;
2159
2160     wire [WIDTH-1:0] pos_set = SET_POLARITY ? SET : ~SET;
2161     wire [WIDTH-1:0] pos_clr = CLR_POLARITY ? CLR : ~CLR;
2162
2163     genvar i;
2164     generate
2165         for (i = 0; i < WIDTH; i = i+1) begin:bitslices
2166             always @*
2167                 if (pos_clr[i])
2168                     Q[i] <= 0;
2169                 else if (pos_set[i])
2170                     Q[i] <= 1;
2171         end
2172     endgenerate
2173
2174 endmodule

```

9.1.5 Memories

Memories are either represented using RTLIL::Memory objects, `$memrd_v2`, `$memwr_v2`, and `$meminit_v2` cells, or by `$mem_v2` cells alone.

In the first alternative the RTLIL::Memory objects hold the general metadata for the memory (bit width, size in number of words, etc.) and for each port a `$memrd_v2` (read port) or `$memwr_v2` (write port) cell is created. Having individual cells for read and write ports has the advantage that they can be consolidated using resource sharing passes. In some cases this drastically reduces the number of required ports on the memory cell. In this alternative, memory initialization data is represented by `$meminit_v2` cells, which allow delaying constant folding for initialization addresses and data until after the frontend finishes.

The `$memrd_v2` cells have a clock input CLK, an enable input EN, an address input ADDR, a data output DATA, an asynchronous reset input ARST, and a synchronous reset input SRST. They also have the following parameters:

MEMID

The name of the RTLIL::Memory object that is associated with this read port.

ABITS

The number of address bits (width of the ADDR input port).

WIDTH

The number of data bits (width of the DATA output port). Note that this may be a power-of-two multiple of the underlying memory's width – such ports are called wide ports and access an aligned group of cells at once. In this case, the corresponding low bits of ADDR must be tied to 0.

CLK_ENABLE

When this parameter is non-zero, the clock is used. Otherwise this read port is asynchronous and the CLK input is not used.

CLK_POLARITY

Clock is active on the positive edge if this parameter has the value 1'b1 and on the negative edge if

this parameter is 1'b0.

TRANSPARENCY_MASK

This parameter is a bitmask of write ports that this read port is transparent with. The bits of this parameter are indexed by the write port's **PORTID** parameter. Transparency can only be enabled between synchronous ports sharing a clock domain. When transparency is enabled for a given port pair, a read and write to the same address in the same cycle will return the new value. Otherwise the old value is returned.

COLLISION_X_MASK

This parameter is a bitmask of write ports that have undefined collision behavior with this port. The bits of this parameter are indexed by the write port's **PORTID** parameter. This behavior can only be enabled between synchronous ports sharing a clock domain. When undefined collision is enabled for a given port pair, a read and write to the same address in the same cycle will return the undefined (all-X) value. This option is exclusive (for a given port pair) with the transparency option.

ARST_VALUE

Whenever the **ARST** input is asserted, the data output will be reset to this value. Only used for synchronous ports.

SRST_VALUE

Whenever the **SRST** input is synchronously asserted, the data output will be reset to this value. Only used for synchronous ports.

INIT_VALUE

The initial value of the data output, for synchronous ports.

CE_OVER_SRST

If this parameter is non-zero, the **SRST** input is only recognized when **EN** is true. Otherwise, **SRST** is recognized regardless of **EN**.

The `$memwr_v2` cells have a clock input **CLK**, an enable input **EN** (one enable bit for each data bit), an address input **ADDR** and a data input **DATA**. They also have the following parameters:

MEMID

The name of the `RTLIL::Memory` object that is associated with this write port.

ABITS

The number of address bits (width of the **ADDR** input port).

WIDTH

The number of data bits (width of the **DATA** output port). Like with `$memrd_v2` cells, the width is allowed to be any power-of-two multiple of memory width, with the corresponding restriction on address.

CLK_ENABLE

When this parameter is non-zero, the clock is used. Otherwise this write port is asynchronous and the **CLK** input is not used.

CLK_POLARITY

Clock is active on positive edge if this parameter has the value 1'b1 and on the negative edge if this parameter is 1'b0.

PORTID

An identifier for this write port, used to index write port bit mask parameters.

PRIORITY_MASK

This parameter is a bitmask of write ports that this write port has priority over in case of writing to the same address. The bits of this parameter are indexed by the other write port's **PORTID** parameter. Write ports can only have priority over write ports with lower port ID. When two ports write to the

same address and neither has priority over the other, the result is undefined. Priority can only be set between two synchronous ports sharing the same clock domain.

The `$meminit_v2` cells have an address input `ADDR`, a data input `DATA`, with the width of the `DATA` port equal to `WIDTH` parameter times `WORDS` parameter, and a bit enable mask input `EN` with width equal to `WIDTH` parameter. All three of the inputs must resolve to a constant for synthesis to succeed.

MEMID

The name of the `RTLIL::Memory` object that is associated with this initialization cell.

ABITS

The number of address bits (width of the `ADDR` input port).

WIDTH

The number of data bits per memory location.

WORDS

The number of consecutive memory locations initialized by this cell.

PRIORITY

The cell with the higher integer value in this parameter wins an initialization conflict.

The HDL frontend models a memory using `RTLIL::Memory` objects and asynchronous `$memrd_v2` and `$memwr_v2` cells. The *memory* pass (i.e. its various sub-passes) migrates `$dff` cells into the `$memrd_v2` and `$memwr_v2` cells making them synchronous, then converts them to a single `$mem_v2` cell and (optionally) maps this cell type to `$dff` cells for the individual words and multiplexer-based address decoders for the read and write interfaces. When the last step is disabled or not possible, a `$mem_v2` cell is left in the design.

The `$mem_v2` cell provides the following parameters:

MEMID

The name of the original `RTLIL::Memory` object that became this `$mem_v2` cell.

SIZE

The number of words in the memory.

ABITS

The number of address bits.

WIDTH

The number of data bits per word.

INIT

The initial memory contents.

RD_PORTS

The number of read ports on this memory cell.

RD_WIDE_CONTINUATION

This parameter is `RD_PORTS` bits wide, containing a bitmask of “wide continuation” read ports. Such ports are used to represent the extra data bits of wide ports in the combined cell, and must have all control signals identical with the preceding port, except for address, which must have the proper sub-cell address encoded in the low bits.

RD_CLK_ENABLE

This parameter is `RD_PORTS` bits wide, containing a clock enable bit for each read port.

RD_CLK_POLARITY

This parameter is `RD_PORTS` bits wide, containing a clock polarity bit for each read port.

RD_TRANSPARENCY_MASK

This parameter is $\text{RD_PORTS} \times \text{WR_PORTS}$ bits wide, containing a concatenation of all **TRANSPARENCY_MASK** values of the original *\$memrd_v2* cells.

RD_COLLISION_X_MASK

This parameter is $\text{RD_PORTS} \times \text{WR_PORTS}$ bits wide, containing a concatenation of all **COLLISION_X_MASK** values of the original *\$memrd_v2* cells.

RD_CE_OVER_SRST

This parameter is RD_PORTS bits wide, determining relative synchronous reset and enable priority for each read port.

RD_INIT_VALUE

This parameter is $\text{RD_PORTS} \times \text{WIDTH}$ bits wide, containing the initial value for each synchronous read port.

RD_ARST_VALUE

This parameter is $\text{RD_PORTS} \times \text{WIDTH}$ bits wide, containing the asynchronous reset value for each synchronous read port.

RD_SRST_VALUE

This parameter is $\text{RD_PORTS} \times \text{WIDTH}$ bits wide, containing the synchronous reset value for each synchronous read port.

WR_PORTS

The number of write ports on this memory cell.

WR_WIDE_CONTINUATION

This parameter is WR_PORTS bits wide, containing a bitmask of “wide continuation” write ports.

WR_CLK_ENABLE

This parameter is WR_PORTS bits wide, containing a clock enable bit for each write port.

WR_CLK_POLARITY

This parameter is WR_PORTS bits wide, containing a clock polarity bit for each write port.

WR_PRIORITY_MASK

This parameter is $\text{WR_PORTS} \times \text{WR_PORTS}$ bits wide, containing a concatenation of all **PRIORITY_MASK** values of the original *\$memwr_v2* cells.

The *\$mem_v2* cell has the following ports:

RD_CLK

This input is RD_PORTS bits wide, containing all clock signals for the read ports.

RD_EN

This input is RD_PORTS bits wide, containing all enable signals for the read ports.

RD_ADDR

This input is $\text{RD_PORTS} \times \text{ABITS}$ bits wide, containing all address signals for the read ports.

RD_DATA

This output is $\text{RD_PORTS} \times \text{WIDTH}$ bits wide, containing all data signals for the read ports.

RD_ARST

This input is RD_PORTS bits wide, containing all asynchronous reset signals for the read ports.

RD_SRST

This input is RD_PORTS bits wide, containing all synchronous reset signals for the read ports.

WR_CLK

This input is WR_PORTS bits wide, containing all clock signals for the write ports.

WR_EN

This input is WR_PORTS*WIDTH bits wide, containing all enable signals for the write ports.

WR_ADDR

This input is WR_PORTS*ABITS bits wide, containing all address signals for the write ports.

WR_DATA

This input is WR_PORTS*WIDTH bits wide, containing all data signals for the write ports.

The *memory_collect* pass can be used to convert discrete *\$memrd_v2*, *\$memwr_v2*, and *\$meminit_v2* cells belonging to the same memory to a single *\$mem_v2* cell, whereas the *memory_unpack* pass performs the inverse operation. The *memory_dff* pass can combine asynchronous memory ports that are fed by or feeding registers into synchronous memory ports. The *memory_bram* pass can be used to recognize *\$mem_v2* cells that can be implemented with a block RAM resource on an FPGA. The *memory_map* pass can be used to implement *\$mem_v2* cells as basic logic: word-wide DFFs and address decoders.

yosys> help \$mem

Simulation model (verilog)

Listing 9.61: simlib.v

```

2820 module \ $mem (RD_CLK, RD_EN, RD_ADDR, RD_DATA, WR_CLK, WR_EN, WR_ADDR, WR_DATA);
2821
2822     parameter MEMID = "";
2823     parameter signed SIZE = 4;
2824     parameter signed OFFSET = 0;
2825     parameter signed ABITS = 2;
2826     parameter signed WIDTH = 8;
2827     parameter signed INIT = 1'bx;
2828
2829     parameter signed RD_PORTS = 1;
2830     parameter RD_CLK_ENABLE = 1'b1;
2831     parameter RD_CLK_POLARITY = 1'b1;
2832     parameter RD_TRANSPARENT = 1'b1;
2833
2834     parameter signed WR_PORTS = 1;
2835     parameter WR_CLK_ENABLE = 1'b1;
2836     parameter WR_CLK_POLARITY = 1'b1;
2837
2838     input [RD_PORTS-1:0] RD_CLK;
2839     input [RD_PORTS-1:0] RD_EN;
2840     input [RD_PORTS*ABITS-1:0] RD_ADDR;
2841     output reg [RD_PORTS*WIDTH-1:0] RD_DATA;
2842
2843     input [WR_PORTS-1:0] WR_CLK;
2844     input [WR_PORTS*WIDTH-1:0] WR_EN;
2845     input [WR_PORTS*ABITS-1:0] WR_ADDR;
2846     input [WR_PORTS*WIDTH-1:0] WR_DATA;
2847
2848     reg [WIDTH-1:0] memory [SIZE-1:0];
2849
2850     integer i, j;
2851     reg [WR_PORTS-1:0] LAST_WR_CLK;

```

(continues on next page)

(continued from previous page)

```

2852     reg [RD_PORTS-1:0] LAST_RD_CLK;
2853
2854     function port_active;
2855         input clk_enable;
2856         input clk_polarity;
2857         input last_clk;
2858         input this_clk;
2859         begin
2860             casez ({clk_enable, clk_polarity, last_clk, this_clk})
2861                 4'b0???: port_active = 1;
2862                 4'b1101: port_active = 1;
2863                 4'b1010: port_active = 1;
2864                 default: port_active = 0;
2865             endcase
2866         end
2867     endfunction
2868
2869     initial begin
2870         for (i = 0; i < SIZE; i = i+1)
2871             memory[i] = INIT >>> (i*WIDTH);
2872     end
2873
2874     always @(RD_CLK, RD_ADDR, RD_DATA, WR_CLK, WR_EN, WR_ADDR, WR_DATA) begin
2875         `ifdef SIMLIB_MEMDELAY
2876             #`SIMLIB_MEMDELAY;
2877         `endif
2878         for (i = 0; i < RD_PORTS; i = i+1) begin
2879             if (!RD_TRANSPARENT[i] && RD_CLK_ENABLE[i] && RD_EN[i] && port_
↪ active(RD_CLK_ENABLE[i], RD_CLK_POLARITY[i], LAST_RD_CLK[i], RD_CLK[i])) begin
2880                 // $display("Read from %s: addr=%b data=%b", MEMID, RD_ADDR[i*ABITS_
↪ +: ABITS], memory[RD_ADDR[i*ABITS +: ABITS] - OFFSET]);
2881                 RD_DATA[i*WIDTH +: WIDTH] <= memory[RD_ADDR[i*ABITS +: ABITS] -
↪ OFFSET];
2882             end
2883         end
2884
2885         for (i = 0; i < WR_PORTS; i = i+1) begin
2886             if (port_active(WR_CLK_ENABLE[i], WR_CLK_POLARITY[i], LAST_WR_CLK[i],
↪ WR_CLK[i]))
2887                 for (j = 0; j < WIDTH; j = j+1)
2888                     if (WR_EN[i*WIDTH+j]) begin
2889                         // $display("Write to %s: addr=%b data=%b", MEMID, WR_
↪ ADDR[i*ABITS +: ABITS], WR_DATA[i*WIDTH+j]);
2890                         memory[WR_ADDR[i*ABITS +: ABITS] - OFFSET][j] = WR_
↪ DATA[i*WIDTH+j];
2891                     end
2892             end
2893
2894         for (i = 0; i < RD_PORTS; i = i+1) begin
2895             if ((RD_TRANSPARENT[i] || !RD_CLK_ENABLE[i]) && port_active(RD_CLK_
↪ ENABLE[i], RD_CLK_POLARITY[i], LAST_RD_CLK[i], RD_CLK[i])) begin
2896                 // $display("Transparent read from %s: addr=%b data=%b", MEMID, RD_

```

(continues on next page)

(continued from previous page)

```

2897   ↪ADDR[i*ABITS +: ABITS], memory[RD_ADDR[i*ABITS +: ABITS] - OFFSET]);
      RD_DATA[i*WIDTH +: WIDTH] <= memory[RD_ADDR[i*ABITS +: ABITS] -
2898   ↪OFFSET];
2898       end
2899       end
2900
2901       LAST_RD_CLK <= RD_CLK;
2902       LAST_WR_CLK <= WR_CLK;
2903   end
2904
2905 endmodule

```

yosys> help \$mem_v2

Simulation model (verilog)

Listing 9.62: simlib.v

```

2909 module \ $mem_v2 (RD_CLK, RD_EN, RD_ARST, RD_SRST, RD_ADDR, RD_DATA, WR_CLK, WR_EN,
2910   ↪WR_ADDR, WR_DATA);
2911
2912     parameter MEMID = "";
2913     parameter signed SIZE = 4;
2914     parameter signed OFFSET = 0;
2915     parameter signed ABITS = 2;
2916     parameter signed WIDTH = 8;
2917     parameter signed INIT = 1'bx;
2918
2919     parameter signed RD_PORTS = 1;
2920     parameter RD_CLK_ENABLE = 1'b1;
2921     parameter RD_CLK_POLARITY = 1'b1;
2922     parameter RD_TRANSPARENCY_MASK = 1'b0;
2923     parameter RD_COLLISION_X_MASK = 1'b0;
2924     parameter RD_WIDE_CONTINUATION = 1'b0;
2925     parameter RD_CE_OVER_SRST = 1'b0;
2926     parameter RD_ARST_VALUE = 1'b0;
2927     parameter RD_SRST_VALUE = 1'b0;
2928     parameter RD_INIT_VALUE = 1'b0;
2929
2930     parameter signed WR_PORTS = 1;
2931     parameter WR_CLK_ENABLE = 1'b1;
2932     parameter WR_CLK_POLARITY = 1'b1;
2933     parameter WR_PRIORITY_MASK = 1'b0;
2934     parameter WR_WIDE_CONTINUATION = 1'b0;
2935
2936     input [RD_PORTS-1:0] RD_CLK;
2937     input [RD_PORTS-1:0] RD_EN;
2938     input [RD_PORTS-1:0] RD_ARST;
2939     input [RD_PORTS-1:0] RD_SRST;
2940     input [RD_PORTS*ABITS-1:0] RD_ADDR;
2941     output reg [RD_PORTS*WIDTH-1:0] RD_DATA;

```

(continues on next page)

(continued from previous page)

```

2942     input [WR_PORTS-1:0] WR_CLK;
2943     input [WR_PORTS*WIDTH-1:0] WR_EN;
2944     input [WR_PORTS*ABITS-1:0] WR_ADDR;
2945     input [WR_PORTS*WIDTH-1:0] WR_DATA;
2946
2947     reg [WIDTH-1:0] memory [SIZE-1:0];
2948
2949     integer i, j, k;
2950     reg [WR_PORTS-1:0] LAST_WR_CLK;
2951     reg [RD_PORTS-1:0] LAST_RD_CLK;
2952
2953     function port_active;
2954         input clk_enable;
2955         input clk_polarity;
2956         input last_clk;
2957         input this_clk;
2958         begin
2959             casez ({clk_enable, clk_polarity, last_clk, this_clk})
2960                 4'b0???: port_active = 1;
2961                 4'b1101: port_active = 1;
2962                 4'b1010: port_active = 1;
2963                 default: port_active = 0;
2964             endcase
2965         end
2966     endfunction
2967
2968     initial begin
2969         for (i = 0; i < SIZE; i = i+1)
2970             memory[i] = INIT >>> (i*WIDTH);
2971         RD_DATA = RD_INIT_VALUE;
2972     end
2973
2974     always @(RD_CLK, RD_ARST, RD_ADDR, RD_DATA, WR_CLK, WR_EN, WR_ADDR, WR_DATA)
2975     ↪begin
2976         `ifdef SIMLIB_MEMDELAY
2977             #`SIMLIB_MEMDELAY;
2978         `endif
2979         for (i = 0; i < RD_PORTS; i = i+1) begin
2980             if (RD_CLK_ENABLE[i] && RD_EN[i] && port_active(RD_CLK_ENABLE[i], RD_
2981             ↪CLK_POLARITY[i], LAST_RD_CLK[i], RD_CLK[i])) begin
2982                 // $display("Read from %s: addr=%b data=%b", MEMID, RD_ADDR[i*ABITS
2983             ↪+: ABITS], memory[RD_ADDR[i*ABITS +: ABITS] - OFFSET]);
2984                 RD_DATA[i*WIDTH +: WIDTH] <= memory[RD_ADDR[i*ABITS +: ABITS] -
2985             ↪OFFSET];
2986
2987                 for (j = 0; j < WR_PORTS; j = j+1) begin
2988                     if (RD_TRANSPARENCY_MASK[i*WR_PORTS + j] && port_active(WR_CLK_
2989             ↪ENABLE[j], WR_CLK_POLARITY[j], LAST_WR_CLK[j], WR_CLK[j]) && RD_ADDR[i*ABITS +:
2990             ↪ABITS] == WR_ADDR[j*ABITS +: ABITS])
2991                         for (k = 0; k < WIDTH; k = k+1)
2992                             if (WR_EN[j*WIDTH+k])
2993                                 RD_DATA[i*WIDTH+k] <= WR_DATA[j*WIDTH+k];

```

(continues on next page)

(continued from previous page)

```

2988         if (RD_COLLISION_X_MASK[i*WR_PORTS + j] && port_active(WR_CLK_
↳ENABLE[j], WR_CLK_POLARITY[j], LAST_WR_CLK[j], WR_CLK[j]) && RD_ADDR[i*ABITS +:
↳ABITS] == WR_ADDR[j*ABITS +: ABITS])
2989             for (k = 0; k < WIDTH; k = k+1)
2990                 if (WR_EN[j*WIDTH+k])
2991                     RD_DATA[i*WIDTH+k] <= 1'bx;
2992             end
2993         end
2994     end
2995
2996     for (i = 0; i < WR_PORTS; i = i+1) begin
2997         if (port_active(WR_CLK_ENABLE[i], WR_CLK_POLARITY[i], LAST_WR_CLK[i],
↳WR_CLK[i]))
2998             for (j = 0; j < WIDTH; j = j+1)
2999                 if (WR_EN[i*WIDTH+j]) begin
3000                     // $display("Write to %s: addr=%b data=%b", MEMID, WR_
↳ADDR[i*ABITS +: ABITS], WR_DATA[i*WIDTH+j]);
3001                     memory[WR_ADDR[i*ABITS +: ABITS] - OFFSET][j] = WR_
↳DATA[i*WIDTH+j];
3002                 end
3003             end
3004
3005             for (i = 0; i < RD_PORTS; i = i+1) begin
3006                 if (!RD_CLK_ENABLE[i]) begin
3007                     // $display("Combinatorial read from %s: addr=%b data=%b", MEMID,
↳RD_ADDR[i*ABITS +: ABITS], memory[RD_ADDR[i*ABITS +: ABITS] - OFFSET]);
3008                     RD_DATA[i*WIDTH +: WIDTH] <= memory[RD_ADDR[i*ABITS +: ABITS] -
↳OFFSET];
3009                 end
3010             end
3011
3012             for (i = 0; i < RD_PORTS; i = i+1) begin
3013                 if (RD_SRST[i] && port_active(RD_CLK_ENABLE[i], RD_CLK_POLARITY[i],
↳LAST_RD_CLK[i], RD_CLK[i]) && (RD_EN[i] || !RD_CE_OVER_SRST[i]))
3014                     RD_DATA[i*WIDTH +: WIDTH] <= RD_SRST_VALUE[i*WIDTH +: WIDTH];
3015                 if (RD_ARST[i])
3016                     RD_DATA[i*WIDTH +: WIDTH] <= RD_ARST_VALUE[i*WIDTH +: WIDTH];
3017             end
3018
3019             LAST_RD_CLK <= RD_CLK;
3020             LAST_WR_CLK <= WR_CLK;
3021         end
3022     endmodule
3023

```

yosys> help \$meminit

Simulation model (verilog)

Listing 9.63: simlib.v

```
2771 module \$(meminit (ADDR, DATA);
2772
2773     parameter MEMID = "";
2774     parameter ABITS = 8;
2775     parameter WIDTH = 8;
2776     parameter WORDS = 1;
2777
2778     parameter PRIORITY = 0;
2779
2780     input [ABITS-1:0] ADDR;
2781     input [WORDS*WIDTH-1:0] DATA;
2782
2783     initial begin
2784         if (MEMID != "") begin
2785             $display("ERROR: Found non-simulatable instance of $(meminit!");
2786             $finish;
2787         end
2788     end
2789
2790 endmodule
```

yosys> help \$(meminit_v2

Simulation model (verilog)

Listing 9.64: simlib.v

```
2795 module \$(meminit_v2 (ADDR, DATA, EN);
2796
2797     parameter MEMID = "";
2798     parameter ABITS = 8;
2799     parameter WIDTH = 8;
2800     parameter WORDS = 1;
2801
2802     parameter PRIORITY = 0;
2803
2804     input [ABITS-1:0] ADDR;
2805     input [WORDS*WIDTH-1:0] DATA;
2806     input [WIDTH-1:0] EN;
2807
2808     initial begin
2809         if (MEMID != "") begin
2810             $display("ERROR: Found non-simulatable instance of $(meminit_v2!");
2811             $finish;
2812         end
2813     end
2814
2815 endmodule
```

yosys> help \$(memrd

Simulation model (verilog)

Listing 9.65: simlib.v

```

2662 module \ $memrd (CLK, EN, ADDR, DATA);
2663
2664     parameter MEMID = "";
2665     parameter ABITS = 8;
2666     parameter WIDTH = 8;
2667
2668     parameter CLK_ENABLE = 0;
2669     parameter CLK_POLARITY = 0;
2670     parameter TRANSPARENT = 0;
2671
2672     input CLK, EN;
2673     input [ABITS-1:0] ADDR;
2674     output [WIDTH-1:0] DATA;
2675
2676     initial begin
2677         if (MEMID != "") begin
2678             $display("ERROR: Found non-simulatable instance of $memrd!");
2679             $finish;
2680         end
2681     end
2682
2683 endmodule

```

yosys> help \$memrd_v2

Simulation model (verilog)

Listing 9.66: simlib.v

```

2687 module \ $memrd_v2 (CLK, EN, ARST, SRST, ADDR, DATA);
2688
2689     parameter MEMID = "";
2690     parameter ABITS = 8;
2691     parameter WIDTH = 8;
2692
2693     parameter CLK_ENABLE = 0;
2694     parameter CLK_POLARITY = 0;
2695     parameter TRANSPARENCY_MASK = 0;
2696     parameter COLLISION_X_MASK = 0;
2697     parameter ARST_VALUE = 0;
2698     parameter SRST_VALUE = 0;
2699     parameter INIT_VALUE = 0;
2700     parameter CE_OVER_SRST = 0;
2701
2702     input CLK, EN, ARST, SRST;
2703     input [ABITS-1:0] ADDR;
2704     output [WIDTH-1:0] DATA;
2705
2706     initial begin
2707         if (MEMID != "") begin
2708             $display("ERROR: Found non-simulatable instance of $memrd_v2!");

```

(continues on next page)

(continued from previous page)

```

2709         $finish;
2710     end
2711 end
2712
2713 endmodule

```

```
yosys> help $memwr
```

```
Simulation model (verilog)
```

Listing 9.67: simlib.v

```

2718 module \ $memwr (CLK, EN, ADDR, DATA);
2719
2720     parameter MEMID = "";
2721     parameter ABITS = 8;
2722     parameter WIDTH = 8;
2723
2724     parameter CLK_ENABLE = 0;
2725     parameter CLK_POLARITY = 0;
2726     parameter PRIORITY = 0;
2727
2728     input CLK;
2729     input [WIDTH-1:0] EN;
2730     input [ABITS-1:0] ADDR;
2731     input [WIDTH-1:0] DATA;
2732
2733     initial begin
2734         if (MEMID != "") begin
2735             $display("ERROR: Found non-simulatable instance of $memwr!");
2736             $finish;
2737         end
2738     end
2739
2740 endmodule

```

```
yosys> help $memwr_v2
```

```
Simulation model (verilog)
```

Listing 9.68: simlib.v

```

2743 module \ $memwr_v2 (CLK, EN, ADDR, DATA);
2744
2745     parameter MEMID = "";
2746     parameter ABITS = 8;
2747     parameter WIDTH = 8;
2748
2749     parameter CLK_ENABLE = 0;
2750     parameter CLK_POLARITY = 0;
2751     parameter PORTID = 0;
2752     parameter PRIORITY_MASK = 0;
2753

```

(continues on next page)

(continued from previous page)

```

2754     input CLK;
2755     input [WIDTH-1:0] EN;
2756     input [ABITS-1:0] ADDR;
2757     input [WIDTH-1:0] DATA;
2758
2759     initial begin
2760         if (MEMID != "") begin
2761             $display("ERROR: Found non-simulatable instance of $memwr_v2!");
2762             $finish;
2763         end
2764     end
2765
2766 endmodule

```

9.1.6 Finite state machines

Todo: Describe `$fsm` cell

yosys> help \$fsm

Simulation model (verilog)

Listing 9.69: simlib.v

```

2567 module \fsm (CLK, ARST, CTRL_IN, CTRL_OUT);
2568
2569     parameter NAME = "";
2570
2571     parameter CLK_POLARITY = 1'b1;
2572     parameter ARST_POLARITY = 1'b1;
2573
2574     parameter CTRL_IN_WIDTH = 1;
2575     parameter CTRL_OUT_WIDTH = 1;
2576
2577     parameter STATE_BITS = 1;
2578     parameter STATE_NUM = 1;
2579     parameter STATE_NUM_LOG2 = 1;
2580     parameter STATE_RST = 0;
2581     parameter STATE_TABLE = 1'b0;
2582
2583     parameter TRANS_NUM = 1;
2584     parameter TRANS_TABLE = 4'b0x0x;
2585
2586     input CLK, ARST;
2587     input [CTRL_IN_WIDTH-1:0] CTRL_IN;
2588     output reg [CTRL_OUT_WIDTH-1:0] CTRL_OUT;
2589
2590     wire pos_clk = CLK == CLK_POLARITY;

```

(continues on next page)

(continued from previous page)

```

2591     wire pos_arst = ARST == ARST_POLARITY;
2592
2593     reg [STATE_BITS-1:0] state;
2594     reg [STATE_BITS-1:0] state_tmp;
2595     reg [STATE_BITS-1:0] next_state;
2596
2597     reg [STATE_BITS-1:0] tr_state_in;
2598     reg [STATE_BITS-1:0] tr_state_out;
2599     reg [CTRL_IN_WIDTH-1:0] tr_ctrl_in;
2600     reg [CTRL_OUT_WIDTH-1:0] tr_ctrl_out;
2601
2602     integer i;
2603
2604     task tr_fetch;
2605         input [31:0] tr_num;
2606         reg [31:0] tr_pos;
2607         reg [STATE_NUM_LOG2-1:0] state_num;
2608         begin
2609             tr_pos = (2*STATE_NUM_LOG2+CTRL_IN_WIDTH+CTRL_OUT_WIDTH)*tr_num;
2610             tr_ctrl_out = TRANS_TABLE >> tr_pos;
2611             tr_pos = tr_pos + CTRL_OUT_WIDTH;
2612             state_num = TRANS_TABLE >> tr_pos;
2613             tr_state_out = STATE_TABLE >> (STATE_BITS*state_num);
2614             tr_pos = tr_pos + STATE_NUM_LOG2;
2615             tr_ctrl_in = TRANS_TABLE >> tr_pos;
2616             tr_pos = tr_pos + CTRL_IN_WIDTH;
2617             state_num = TRANS_TABLE >> tr_pos;
2618             tr_state_in = STATE_TABLE >> (STATE_BITS*state_num);
2619             tr_pos = tr_pos + STATE_NUM_LOG2;
2620         end
2621     endtask
2622
2623     always @(posedge pos_clk, posedge pos_arst) begin
2624         if (pos_arst) begin
2625             state_tmp = STATE_TABLE[STATE_BITS*(STATE_RST+1)-1:STATE_BITS*STATE_
↵RST];
2626             for (i = 0; i < STATE_BITS; i = i+1)
2627                 if (state_tmp[i] === 1'bz)
2628                     state_tmp[i] = 0;
2629             state <= state_tmp;
2630         end else begin
2631             state_tmp = next_state;
2632             for (i = 0; i < STATE_BITS; i = i+1)
2633                 if (state_tmp[i] === 1'bz)
2634                     state_tmp[i] = 0;
2635             state <= state_tmp;
2636         end
2637     end
2638
2639     always @(state, CTRL_IN) begin
2640         next_state <= STATE_TABLE[STATE_BITS*(STATE_RST+1)-1:STATE_BITS*STATE_RST];
2641         CTRL_OUT <= 'bx;

```

(continues on next page)

(continued from previous page)

```

2642     // $display("---");
2643     // $display("Q: %b %b", state, CTRL_IN);
2644     for (i = 0; i < TRANS_NUM; i = i+1) begin
2645         tr_fetch(i);
2646         // $display("T: %b %b -> %b %b [%d]", tr_state_in, tr_ctrl_in, tr_state_
→out, tr_ctrl_out, i);
2647         casez ({state, CTRL_IN})
2648             {tr_state_in, tr_ctrl_in}: begin
2649                 // $display("-> %b %b <- MATCH", state, CTRL_IN);
2650                 {next_state, CTRL_OUT} <= {tr_state_out, tr_ctrl_out};
2651             end
2652         endcase
2653     end
2654 end
2655
2656 endmodule

```

9.1.7 Coarse arithmetics

Todo: Add information about *\$alu*, *\$fa*, and *\$lcu* cells.

The *\$macc* cell type represents a generalized multiply and accumulate operation. The cell is purely combinational. It outputs the result of summing up a sequence of products and other injected summands.

```

Y = 0 +- a0factor1 * a0factor2 +- a1factor1 * a1factor2 +- ...
    + B[0] + B[1] + ...

```

The A port consists of concatenated pairs of multiplier inputs (“factors”). A zero length factor2 acts as a constant 1, turning factor1 into a simple summand.

In this pseudocode, *u(foo)* means an unsigned int that’s foo bits long.

```

struct A {
    u(CONFIG.mul_info[0].factor1_len) a0factor1;
    u(CONFIG.mul_info[0].factor2_len) a0factor2;
    u(CONFIG.mul_info[1].factor1_len) a1factor1;
    u(CONFIG.mul_info[1].factor2_len) a1factor2;
    ...
};

```

The cell’s *CONFIG* parameter determines the layout of cell port A. The *CONFIG* parameter carries the following information:

```

struct CONFIG {
    u4 num_bits;
    struct mul_info {
        bool is_signed;
        bool is_subtract;
        u(num_bits) factor1_len;
        u(num_bits) factor2_len;
    };
};

```

(continues on next page)

(continued from previous page)

```

    }[num_ports];
};

```

B is an array of concatenated 1-bit-wide unsigned integers to also be summed up.

yosys> help \$alu

Arithmetic logic unit

A building block supporting both binary addition/subtraction operations, and indirectly, comparison operations. Typically created by the *alumacc* pass, which transforms: *\$add*, *\$sub*, *\$lt*, *\$le*, *\$ge*, *\$gt*, *\$eq*, *\$eqx*, *\$ne*, *\$nex* cells into this *\$alu* cell.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.70: simlib.v

```

657 module \alu (A, B, CI, BI, X, Y, CO);
658
659     parameter A_SIGNED = 0;
660     parameter B_SIGNED = 0;
661     parameter A_WIDTH = 1;
662     parameter B_WIDTH = 1;
663     parameter Y_WIDTH = 1;
664
665     input [A_WIDTH-1:0] A;      // Input operand
666     input [B_WIDTH-1:0] B;      // Input operand
667     output [Y_WIDTH-1:0] X;     // A xor B (sign-extended, optional B inversion,
668                                 //          used in combination with
669                                 //          reduction-AND for $eq/$ne ops)
670     output [Y_WIDTH-1:0] Y;     // Sum
671
672     input CI;                   // Carry-in (set for $sub)
673     input BI;                   // Invert-B (set for $sub)
674     output [Y_WIDTH-1:0] CO;    // Carry-out
675
676     wire [Y_WIDTH-1:0] AA, BB;
677
678     generate
679         if (A_SIGNED && B_SIGNED) begin:BLOCK1
680             assign AA = $signed(A), BB = BI ? ~$signed(B) : $signed(B);
681         end else begin:BLOCK2
682             assign AA = $unsigned(A), BB = BI ? ~$unsigned(B) : $unsigned(B);
683         end
684     endgenerate
685
686     // this is 'x' if Y and CO should be all 'x', and '0' otherwise
687     wire y_co_undef = ^{A, A, B, B, CI, CI, BI, BI};
688
689     assign X = AA ^ BB;

```

(continues on next page)

(continued from previous page)

```

690 // Full adder
691 assign Y = (AA + BB + CI) ^ {Y_WIDTH{y_co_undef}};
692
693 function get_carry;
694     input a, b, c;
695     get_carry = (a&b) | (a&c) | (b&c);
696 endfunction
697
698 genvar i;
699 generate
700     assign CO[0] = get_carry(AA[0], BB[0], CI) ^ y_co_undef;
701     for (i = 1; i < Y_WIDTH; i = i+1) begin:BLOCK3
702         assign CO[i] = get_carry(AA[i], BB[i], CO[i-1]) ^ y_co_undef;
703     end
704 endgenerate
705
706 endmodule

```

yosys> help \$fa

Properties

is_evaluable

Simulation model (verilog)

Listing 9.71: simlib.v

```

597 module \fa (A, B, C, X, Y);
598
599     parameter WIDTH = 1;
600
601     input [WIDTH-1:0] A, B, C;
602     output [WIDTH-1:0] X, Y;
603
604     wire [WIDTH-1:0] t1, t2, t3;
605
606     assign t1 = A ^ B, t2 = A & B, t3 = C & t1;
607     assign Y = t1 ^ C, X = (t2 | t3) ^ (Y ^ Y);
608
609 endmodule

```

yosys> help \$lcu

Lookahead carry unit A building block dedicated to fast computation of carry-bits used in binary arithmetic operations. By replacing the ripple carry structure used in full-adder blocks, the more significant bits of the sum can be expected to be computed more quickly. Typically created during *techmap* of \$alu cells (see the “_90_alu” rule in +/techmap.v).

Properties

is_evaluable

Simulation model (verilog)

Listing 9.72: simlib.v

```

625 module \l1cu (P, G, CI, CO);
626
627     parameter WIDTH = 1;
628
629     input [WIDTH-1:0] P;    // Propagate
630     input [WIDTH-1:0] G;    // Generate
631     input CI;               // Carry-in
632
633     output reg [WIDTH-1:0] CO; // Carry-out
634
635     integer i;
636     always @* begin
637         CO = 'bx;
638         if (~{P, G, CI} != 1'bx) begin
639             CO[0] = G[0] || (P[0] && CI);
640             for (i = 1; i < WIDTH; i = i+1)
641                 CO[i] = G[i] || (P[i] && CO[i-1]);
642         end
643     end
644
645 endmodule

```

yosys> help \$macc

Multiply and accumulate. A building block for summing any number of negated and unnegated signals and arithmetic products of pairs of signals. Cell port A concatenates pairs of signals to be multiplied together. When the second signal in a pair is zero length, a constant 1 is used instead as the second factor. Cell port B concatenates 1-bit-wide signals to also be summed, such as “carry in” in adders. Typically created by the *alumacc* pass, which transforms \$add and \$mul into \$macc cells.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.73: simlib.v

```

1073 module \macc (A, B, Y);
1074
1075     parameter A_WIDTH = 0;
1076     parameter B_WIDTH = 0;
1077     parameter Y_WIDTH = 0;
1078     // CONFIG determines the layout of A, as explained below
1079     parameter CONFIG = 4'b0000;
1080     parameter CONFIG_WIDTH = 4;
1081
1082     // In the terms used for this cell, there's mixed meanings for the term "port".
1083     ↪ To disambiguate:
1084     // A cell port is for example the A input (it is constructed in C++ as cell->
1085     ↪ setPort(ID::A, ...))
1086     // Multiplier ports are pairs of multiplier inputs ("factors").
1087     // If the second signal in such a pair is zero length, no multiplication is
1088     ↪ necessary, and the first signal is just added to the sum.

```

(continues on next page)

(continued from previous page)

```

1086     input [A_WIDTH-1:0] A; // Cell port A is the concatenation of all arithmetic
↳ports
1087     input [B_WIDTH-1:0] B; // Cell port B is the concatenation of single-bit
↳unsigned signals to be also added to the sum
1088     output reg [Y_WIDTH-1:0] Y; // Output sum
1089
1090     // Xilinx XSIM does not like $clog2() below..
1091     function integer my_clog2;
1092         input integer v;
1093         begin
1094             if (v > 0)
1095                 v = v - 1;
1096             my_clog2 = 0;
1097             while (v) begin
1098                 v = v >> 1;
1099                 my_clog2 = my_clog2 + 1;
1100             end
1101         end
1102     endfunction
1103
1104     // Bits that a factor's length field in CONFIG per factor in cell port A
1105     localparam integer num_bits = CONFIG[3:0] > 0 ? CONFIG[3:0] : 1;
1106     // Number of multiplier ports
1107     localparam integer num_ports = (CONFIG_WIDTH-4) / (2 + 2*num_bits);
1108     // Minium bit width of an induction variable to iterate over all bits of cell
↳port A
1109     localparam integer num_abits = my_clog2(A_WIDTH) > 0 ? my_clog2(A_WIDTH) : 1;
1110
1111     // In this pseudocode, u(foo) means an unsigned int that's foo bits long.
1112     // The CONFIG parameter carries the following information:
1113     //     struct CONFIG {
1114     //         u4 num_bits;
1115     //         struct port_field {
1116     //             bool is_signed;
1117     //             bool is_subtract;
1118     //             u(num_bits) factor1_len;
1119     //             u(num_bits) factor2_len;
1120     //         }[num_ports];
1121     //     };
1122
1123     // The A cell port carries the following information:
1124     //     struct A {
1125     //         u(CONFIG.port_field[0].factor1_len) port0factor1;
1126     //         u(CONFIG.port_field[0].factor2_len) port0factor2;
1127     //         u(CONFIG.port_field[1].factor1_len) port1factor1;
1128     //         u(CONFIG.port_field[1].factor2_len) port1factor2;
1129     //         ...
1130     //     };
1131     // and log(sizeof(A)) is num_abits.
1132     // No factor1 may have a zero length.
1133     // A factor2 having a zero length implies factor2 is replaced with a constant 1.
1134

```

(continues on next page)

(continued from previous page)

```

1135 // Additionally, B is an array of 1-bit-wide unsigned integers to also be
1136 // summed up.
1137 // Finally, we have:
1138 // Y = port0factor1 * port0factor2 + port1factor1 * port1factor2 + ...
1139 //      * B[0] + B[1] + ...
1140
1141 function [2*num_ports*num_abits-1:0] get_port_offsets;
1142     input [CONFIG_WIDTH-1:0] cfg;
1143     integer i, cursor;
1144     begin
1145         cursor = 0;
1146         get_port_offsets = 0;
1147         for (i = 0; i < num_ports; i = i+1) begin
1148             get_port_offsets[(2*i + 0)*num_abits +: num_abits] = cursor;
1149             cursor = cursor + cfg[4 + i*(2 + 2*num_bits) + 2 +: num_bits];
1150             get_port_offsets[(2*i + 1)*num_abits +: num_abits] = cursor;
1151             cursor = cursor + cfg[4 + i*(2 + 2*num_bits) + 2 + num_bits +: num_
1152 //bits];
1153         end
1154     end
1155 endfunction
1156
1157 localparam [2*num_ports*num_abits-1:0] port_offsets = get_port_offsets(CONFIG);
1158
1159 `define PORT_IS_SIGNED    (0 + CONFIG[4 + i*(2 + 2*num_bits)])
1160 `define PORT_DO_SUBTRACT (0 + CONFIG[4 + i*(2 + 2*num_bits) + 1])
1161 `define PORT_SIZE_A      (0 + CONFIG[4 + i*(2 + 2*num_bits) + 2 +: num_bits])
1162 `define PORT_SIZE_B      (0 + CONFIG[4 + i*(2 + 2*num_bits) + 2 + num_bits +:
1163 //num_bits])
1164 `define PORT_OFFSET_A    (0 + port_offsets[2*i*num_abits +: num_abits])
1165 `define PORT_OFFSET_B    (0 + port_offsets[2*i*num_abits + num_abits +: num_
1166 //abits])
1167
1168 integer i, j;
1169 reg [Y_WIDTH-1:0] tmp_a, tmp_b;
1170
1171 always @* begin
1172     Y = 0;
1173     for (i = 0; i < num_ports; i = i+1)
1174     begin
1175         tmp_a = 0;
1176         tmp_b = 0;
1177
1178         for (j = 0; j < `PORT_SIZE_A; j = j+1)
1179             tmp_a[j] = A[`PORT_OFFSET_A + j];
1180
1181         if (`PORT_IS_SIGNED && `PORT_SIZE_A > 0)
1182             for (j = `PORT_SIZE_A; j < Y_WIDTH; j = j+1)
1183                 tmp_a[j] = tmp_a[`PORT_SIZE_A-1];
1184
1185         for (j = 0; j < `PORT_SIZE_B; j = j+1)
1186             tmp_b[j] = A[`PORT_OFFSET_B + j];

```

(continues on next page)

(continued from previous page)

```

1183
1184     if (`PORT_IS_SIGNED && `PORT_SIZE_B > 0)
1185         for (j = `PORT_SIZE_B; j < Y_WIDTH; j = j+1)
1186             tmp_b[j] = tmp_b[`PORT_SIZE_B-1];
1187
1188     if (`PORT_SIZE_B > 0)
1189         tmp_a = tmp_a * tmp_b;
1190
1191     if (`PORT_DO_SUBTRACT)
1192         Y = Y - tmp_a;
1193     else
1194         Y = Y + tmp_a;
1195 end
1196 for (i = 0; i < B_WIDTH; i = i+1) begin
1197     Y = Y + B[i];
1198 end
1199 end
1200
1201 `undef PORT_IS_SIGNED
1202 `undef PORT_DO_SUBTRACT
1203 `undef PORT_SIZE_A
1204 `undef PORT_SIZE_B
1205 `undef PORT_OFFSET_A
1206 `undef PORT_OFFSET_B
1207
1208 endmodule

```

9.1.8 Arbitrary logic functions

The *\$lut* cell type implements a single-output LUT (lookup table). It implements an arbitrary logic function with its `\LUT` parameter to map input port `\A` to values of `\Y` output port values. In pseudocode: $Y = \text{LUT}[A]$. `\A` has width set by parameter `\WIDTH` and `\Y` has a width of 1. Every logic function with a single bit output has a unique *\$lut* representation.

The *\$sop* cell type implements a sum-of-products expression, also known as disjunctive normal form (DNF). It implements an arbitrary logic function. Its structure mimics a programmable logic array (PLA). Output port `\Y` is the sum of products of the bits of the input port `\A` as defined by parameter `\TABLE`. `\A` is `\WIDTH` bits wide. The number of products in the sum is set by parameter `\DEPTH`, and each product has two bits for each input bit - for the presence of the unnegated and negated version of said input bit in the product. Therefore the `\TABLE` parameter holds $2 * \text{\WIDTH} * \text{\DEPTH}$ bits.

For example:

Let `\WIDTH` be 3. We would like to represent $Y = \sim A[0] + A[1] \sim A[2]$. There are 2 products to be summed, so `\DEPTH` shall be 2.

```

~A[2]-----+
 A[2]-----+
~A[1]----+|
 A[1]---+||
~A[0]--+|||
 A[0]+||||

```

(continues on next page)

(continued from previous page)

```

||||| product formula
010000 ~\A[0]
001001 \A[1]~\A[2]

```

So the value of \TABLE will become 010000001001.

Any logic function with a single bit output can be represented with \$sop but may have variously minimized or ordered summands represented in the \TABLE values.

yosys> help \$lut

```

Properties
  is_evaluable

Simulation model (verilog)

```

Listing 9.74: simlib.v

```

1641 module \ $lut (A, Y);
1642
1643     parameter WIDTH = 0;
1644     parameter LUT = 0;
1645
1646     input [WIDTH-1:0] A;
1647     output Y;
1648
1649     \ $bmux #(.WIDTH(1), .S_WIDTH(WIDTH)) mux(.A(LUT[(1<<WIDTH)-1:0]), .S(A), .Y(Y));
1650
1651 endmodule

```

yosys> help \$sop

```

Properties
  is_evaluable

Simulation model (verilog)

```

Listing 9.75: simlib.v

```

1657 module \ $sop (A, Y);
1658
1659     parameter WIDTH = 0;
1660     parameter DEPTH = 0;
1661     parameter TABLE = 0;
1662
1663     input [WIDTH-1:0] A;
1664     output reg Y;
1665
1666     integer i, j;
1667     reg match;
1668
1669     always @* begin
1670         Y = 0;

```

(continues on next page)

(continued from previous page)

```

1671     for (i = 0; i < DEPTH; i=i+1) begin
1672         match = 1;
1673         for (j = 0; j < WIDTH; j=j+1) begin
1674             if (TABLE[2*WIDTH*i + 2*j + 0] && A[j]) match = 0;
1675             if (TABLE[2*WIDTH*i + 2*j + 1] && !A[j]) match = 0;
1676         end
1677         if (match) Y = 1;
1678     end
1679 end
1680
1681 endmodule

```

9.1.9 Specify rules

Todo: *\$specify2*, *\$specify3*, and *\$specrule* cells.

yosys> help \$specify2

Properties

is_evaluable

Simulation model (verilog)

Listing 9.76: simlib.v

```

1709 module \ $specify2 (EN, SRC, DST);
1710
1711     parameter FULL = 0;
1712     parameter SRC_WIDTH = 1;
1713     parameter DST_WIDTH = 1;
1714
1715     parameter SRC_DST_PEN = 0;
1716     parameter SRC_DST_POL = 0;
1717
1718     parameter T_RISE_MIN = 0;
1719     parameter T_RISE_TYP = 0;
1720     parameter T_RISE_MAX = 0;
1721
1722     parameter T_FALL_MIN = 0;
1723     parameter T_FALL_TYP = 0;
1724     parameter T_FALL_MAX = 0;
1725
1726     input EN;
1727     input [SRC_WIDTH-1:0] SRC;
1728     input [DST_WIDTH-1:0] DST;
1729
1730     localparam SD = SRC_DST_PEN ? (SRC_DST_POL ? 1 : 2) : 0;
1731

```

(continues on next page)

(continued from previous page)

```

1732 `ifdef SIMLIB_SPECIFY
1733     specify
1734         if (EN && SD==0 && !FULL) (SRC => DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1735         if (EN && SD==0 && FULL) (SRC *> DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1736         if (EN && SD==1 && !FULL) (SRC +=> DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1737         if (EN && SD==1 && FULL) (SRC **> DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1738         if (EN && SD==2 && !FULL) (SRC -=> DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1739         if (EN && SD==2 && FULL) (SRC -*> DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1740     endspecify
1741 `endif
1742
1743 endmodule

```

yosys> help \$specify3

Properties

is_evaluable

Simulation model (verilog)

Listing 9.77: simlib.v

```

1748 module \ $specify3 (EN, SRC, DST, DAT);
1749
1750     parameter FULL = 0;
1751     parameter SRC_WIDTH = 1;
1752     parameter DST_WIDTH = 1;
1753
1754     parameter EDGE_EN = 0;
1755     parameter EDGE_POL = 0;
1756
1757     parameter SRC_DST_PEN = 0;
1758     parameter SRC_DST_POL = 0;
1759
1760     parameter DAT_DST_PEN = 0;
1761     parameter DAT_DST_POL = 0;
1762
1763     parameter T_RISE_MIN = 0;
1764     parameter T_RISE_TYP = 0;
1765     parameter T_RISE_MAX = 0;
1766
1767     parameter T_FALL_MIN = 0;
1768     parameter T_FALL_TYP = 0;
1769     parameter T_FALL_MAX = 0;
1770
1771     input EN;
1772     input [SRC_WIDTH-1:0] SRC;

```

(continues on next page)

(continued from previous page)

```

1773     input [DST_WIDTH-1:0] DST, DAT;
1774
1775     localparam ED = EDGE_EN      ? (EDGE_POL      ? 1 : 2) : 0;
1776     localparam SD = SRC_DST_PEN ? (SRC_DST_POL ? 1 : 2) : 0;
1777     localparam DD = DAT_DST_PEN ? (DAT_DST_POL ? 1 : 2) : 0;
1778
1779     `ifdef SIMLIB_SPECIFY
1780     specify
1781         // DD=0
1782
1783         if (EN && DD==0 && SD==0 && ED==0 && !FULL) (      SRC => (DST : DAT))
1784     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1785         if (EN && DD==0 && SD==0 && ED==0 && FULL) (      SRC *> (DST : DAT))
1786     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1787         if (EN && DD==0 && SD==0 && ED==1 && !FULL) (posedge SRC => (DST : DAT))
1788     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1789         if (EN && DD==0 && SD==0 && ED==1 && FULL) (posedge SRC *> (DST : DAT))
1790     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1791         if (EN && DD==0 && SD==0 && ED==2 && !FULL) (negedge SRC => (DST : DAT))
1792     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1793         if (EN && DD==0 && SD==0 && ED==2 && FULL) (negedge SRC *> (DST : DAT))
1794     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1795
1796         if (EN && DD==0 && SD==1 && ED==0 && !FULL) (      SRC +> (DST : DAT))
1797     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1798         if (EN && DD==0 && SD==1 && ED==0 && FULL) (      SRC ++> (DST : DAT))
1799     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1800         if (EN && DD==0 && SD==1 && ED==1 && !FULL) (posedge SRC +> (DST : DAT))
1801     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1802         if (EN && DD==0 && SD==1 && ED==1 && FULL) (posedge SRC ++> (DST : DAT))
1803     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1804         if (EN && DD==0 && SD==1 && ED==2 && !FULL) (negedge SRC +> (DST : DAT))
1805     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1806         if (EN && DD==0 && SD==1 && ED==2 && FULL) (negedge SRC ++> (DST : DAT))
1807     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1808
1809         if (EN && DD==0 && SD==2 && ED==0 && !FULL) (      SRC --> (DST : DAT))
1810     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1811         if (EN && DD==0 && SD==2 && ED==0 && FULL) (      SRC -*> (DST : DAT))
1812     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1813         if (EN && DD==0 && SD==2 && ED==1 && !FULL) (posedge SRC --> (DST : DAT))
1814     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1815         if (EN && DD==0 && SD==2 && ED==1 && FULL) (posedge SRC -*> (DST : DAT))
1816     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1817         if (EN && DD==0 && SD==2 && ED==2 && !FULL) (negedge SRC --> (DST : DAT))
1818     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1819         if (EN && DD==0 && SD==2 && ED==2 && FULL) (negedge SRC -*> (DST : DAT))
1820     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1821
1822         // DD=1
1823
1824         if (EN && DD==1 && SD==0 && ED==0 && !FULL) (      SRC => (DST +: DAT))

```

(continues on next page)

(continued from previous page)

```

1807  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==0 && ED==0 && FULL) (      SRC *> (DST +: DAT))
1808  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==0 && ED==1 && !FULL) (posedge SRC => (DST +: DAT))
1809  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==0 && ED==1 && FULL) (posedge SRC *> (DST +: DAT))
1810  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==0 && ED==2 && !FULL) (negedge SRC => (DST +: DAT))
1811  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==0 && ED==2 && FULL) (negedge SRC *> (DST +: DAT))
1812  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1813      if (EN && DD==1 && SD==1 && ED==0 && !FULL) (      SRC +=> (DST +: DAT))
1814  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==1 && ED==0 && FULL) (      SRC +=> (DST +: DAT))
1815  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==1 && ED==1 && !FULL) (posedge SRC +=> (DST +: DAT))
1816  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==1 && ED==1 && FULL) (posedge SRC +=> (DST +: DAT))
1817  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==1 && ED==2 && !FULL) (negedge SRC +=> (DST +: DAT))
1818  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==1 && ED==2 && FULL) (negedge SRC +=> (DST +: DAT))
1819  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1820      if (EN && DD==1 && SD==2 && ED==0 && !FULL) (      SRC --> (DST +: DAT))
1821  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==2 && ED==0 && FULL) (      SRC -*> (DST +: DAT))
1822  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==2 && ED==1 && !FULL) (posedge SRC --> (DST +: DAT))
1823  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==2 && ED==1 && FULL) (posedge SRC -*> (DST +: DAT))
1824  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==2 && ED==2 && !FULL) (negedge SRC --> (DST +: DAT))
1825  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==1 && SD==2 && ED==2 && FULL) (negedge SRC -*> (DST +: DAT))
1826  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1827      // DD=2
1828
1829      if (EN && DD==2 && SD==0 && ED==0 && !FULL) (      SRC => (DST -: DAT))
1830  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==2 && SD==0 && ED==0 && FULL) (      SRC *> (DST -: DAT))
1831  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==2 && SD==0 && ED==1 && !FULL) (posedge SRC => (DST -: DAT))
1832  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==2 && SD==0 && ED==1 && FULL) (posedge SRC *> (DST -: DAT))
1833  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==2 && SD==0 && ED==2 && !FULL) (negedge SRC => (DST -: DAT))
1834  <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
      if (EN && DD==2 && SD==0 && ED==2 && FULL) (negedge SRC *> (DST -: DAT))
      <=> (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);

```

(continues on next page)

(continued from previous page)

```

1835
1836     if (EN && DD==2 && SD==1 && ED==0 && !FULL) (          SRC +=> (DST -: DAT))
1837   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1838     if (EN && DD==2 && SD==1 && ED==0 && FULL) (          SRC +=> (DST -: DAT))
1839   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1840     if (EN && DD==2 && SD==1 && ED==1 && !FULL) (posedge SRC +=> (DST -: DAT))
1841   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1842     if (EN && DD==2 && SD==1 && ED==1 && FULL) (posedge SRC +=> (DST -: DAT))
1843   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1844     if (EN && DD==2 && SD==1 && ED==2 && !FULL) (negedge SRC +=> (DST -: DAT))
1845   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1846     if (EN && DD==2 && SD==1 && ED==2 && FULL) (negedge SRC +=> (DST -: DAT))
1847   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1848
1849     if (EN && DD==2 && SD==2 && ED==0 && !FULL) (          SRC ==> (DST -: DAT))
1850   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1851     if (EN && DD==2 && SD==2 && ED==0 && FULL) (          SRC ==> (DST -: DAT))
1852   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1853     if (EN && DD==2 && SD==2 && ED==1 && !FULL) (posedge SRC ==> (DST -: DAT))
1854   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1855     if (EN && DD==2 && SD==2 && ED==1 && FULL) (posedge SRC ==> (DST -: DAT))
1856   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1857     if (EN && DD==2 && SD==2 && ED==2 && !FULL) (negedge SRC ==> (DST -: DAT))
1858   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1859     if (EN && DD==2 && SD==2 && ED==2 && FULL) (negedge SRC ==> (DST -: DAT))
1860   ⇨= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1861   endspecify
1862 `endif
1863 endmodule

```

yosys> help \$specrule

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.78: simlib.v

```

1857 module \specrule (EN_SRC, EN_DST, SRC, DST);
1858
1859     parameter TYPE = "";
1860     parameter T_LIMIT = 0;
1861     parameter T_LIMIT2 = 0;
1862
1863     parameter SRC_WIDTH = 1;
1864     parameter DST_WIDTH = 1;
1865
1866     parameter SRC_PEN = 0;
1867     parameter SRC_POL = 0;
1868
1869     parameter DST_PEN = 0;

```

(continues on next page)

(continued from previous page)

```
1870     parameter DST_POL = 0;
1871
1872     input EN_SRC, EN_DST;
1873     input [SRC_WIDTH-1:0] SRC;
1874     input [DST_WIDTH-1:0] DST;
1875
1876     `ifdef SIMLIB_SPECIFY
1877     specify
1878         // TBD
1879     endspecify
1880     `endif
1881
1882 endmodule
```

9.1.10 Formal verification cells

Note: Some front-ends may not support the generic *\$check* cell, in such cases calling `chformal -lower` will convert each *\$check* cell into it's equivalent. See *chformal* for more.

Todo: Describe formal cells

\$check, *\$assert*, *\$assume*, *\$live*, *\$fair*, *\$cover*, *\$equiv*, *\$initstate*, *\$anyconst*, *\$anyseq*, *\$anyinit*, *\$allconst*, and *\$allseq*.

Also *\$ff* and *\$_FF_* cells.

```
yosys> help $allconst
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.79: simlib.v

```
2061 module \ $allconst (Y);
2062
2063     parameter WIDTH = 0;
2064
2065     output [WIDTH-1:0] Y;
2066
2067     assign Y = 'bx;
2068
2069 endmodule
```

```
yosys> help $allseq
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.80: simlib.v

```

2074 module \${allseq} (Y);
2075
2076     parameter WIDTH = 0;
2077
2078     output [WIDTH-1:0] Y;
2079
2080     assign Y = 'bx;
2081
2082 endmodule

```

yosys> help \$anyconst

Properties

is_evaluable

Simulation model (verilog)

Listing 9.81: simlib.v

```

2014 module \${anyconst} (Y);
2015
2016     parameter WIDTH = 0;
2017
2018     output [WIDTH-1:0] Y;
2019
2020     assign Y = 'bx;
2021
2022 endmodule

```

yosys> help \$anyinit

Simulation model (verilog)

Listing 9.82: simlib.v

```

2043 module \${anyinit} (D, Q);
2044
2045     parameter WIDTH = 0;
2046
2047     input [WIDTH-1:0] D;
2048     output reg [WIDTH-1:0] Q;
2049
2050     initial Q <= 'bx;
2051
2052     always @(`SIMLIB_GLOBAL_CLOCK) begin
2053         Q <= D;
2054     end
2055
2056 endmodule

```

yosys> help \$anyseq

Properties
is_evaluable
Simulation model (verilog)

Listing 9.83: simlib.v

```
2027 module \${anyseq} (Y);  
2028  
2029     parameter WIDTH = 0;  
2030  
2031     output [WIDTH-1:0] Y;  
2032  
2033     assign Y = 'bx;  
2034  
2035 endmodule
```

yosys> help \$assert

Properties
is_evaluable
Simulation model (verilog)

Listing 9.84: simlib.v

```
1933 module \${assert} (A, EN);  
1934  
1935     input A, EN;  
1936  
1937     `ifndef SIMLIB_NOCHECKS  
1938     always @* begin  
1939         if (A !== 1'b1 && EN === 1'b1) begin  
1940             $display("Assertion %m failed!");  
1941             $stop;  
1942         end  
1943     end  
1944     `endif  
1945  
1946 endmodule
```

yosys> help \$assume

Properties
is_evaluable
Simulation model (verilog)

Listing 9.85: simlib.v

```
1951 module \${assume} (A, EN);  
1952  
1953     input A, EN;  
1954  
1955     `ifndef SIMLIB_NOCHECKS  
1956     always @* begin
```

(continues on next page)

(continued from previous page)

```

1957     if (A !== 1'b1 && EN === 1'b1) begin
1958         $display("Assumption %m failed!");
1959         $stop;
1960     end
1961 end
1962 `endif
1963
1964 endmodule

```

yosys> help \$cover

```

    Properties
        is_evaluable
    Simulation model (verilog)

```

Listing 9.86: simlib.v

```

1987 module \ $cover (A, EN);
1988
1989     input A, EN;
1990
1991 endmodule

```

yosys> help \$equiv

```

    Properties
        is_evaluable
    Simulation model (verilog)

```

Listing 9.87: simlib.v

```

2087 module \ $equiv (A, B, Y);
2088
2089     input A, B;
2090     output Y;
2091
2092     assign Y = (A !== 1'bx && A !== B) ? 1'bx : A;
2093
2094     `ifndef SIMLIB_NOCHECKS
2095     always @* begin
2096         if (A !== 1'bx && A !== B) begin
2097             $display("Equivalence failed!");
2098             $stop;
2099         end
2100     end
2101     `endif
2102
2103 endmodule

```

yosys> help \$fair

```

    Properties
        is_evaluable

```

Simulation model (verilog)

Listing 9.88: simlib.v

```
1978 module \ $fair (A, EN);
1979
1980     input A, EN;
1981
1982 endmodule
```

yosys> help \$ff

Simulation model (verilog)

Listing 9.89: simlib.v

```
2184 module \ $ff (D, Q);
2185
2186     parameter WIDTH = 0;
2187
2188     input [WIDTH-1:0] D;
2189     output reg [WIDTH-1:0] Q;
2190
2191     always @(`SIMLIB_GLOBAL_CLOCK) begin
2192         Q <= D;
2193     end
2194
2195 endmodule
```

yosys> help \$initstate

Properties

is_evaluable

Simulation model (verilog)

Listing 9.90: simlib.v

```
1996 module \ $initstate (Y);
1997
1998     output reg Y = 1;
1999     reg [3:0] cnt = 1;
2000     reg trig = 0;
2001
2002     initial trig <= 1;
2003
2004     always @(cnt, trig) begin
2005         Y <= |cnt;
2006         cnt <= cnt + |cnt;
2007     end
2008
2009 endmodule
```

yosys> help \$live

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.91: simlib.v

```

1969 module \live (A, EN);
1970
1971     input A, EN;
1972
1973 endmodule

```

Formal support cells

yosys> help \$future_ff

Simulation model (verilog)

Listing 9.92: simlib.v

```

3086 module \future_ff (A, Y);
3087
3088     parameter WIDTH = 0;
3089
3090     input [WIDTH-1:0] A;
3091     output [WIDTH-1:0] Y;
3092
3093     assign Y = A;
3094
3095 endmodule

```

yosys> help \$get_tag

Simulation model (verilog)

Listing 9.93: simlib.v

```

3045 module \get_tag (A, Y);
3046
3047     parameter TAG = "";
3048     parameter WIDTH = 0;
3049
3050     input [WIDTH-1:0] A;
3051     output [WIDTH-1:0] Y;
3052
3053     assign Y = A;
3054
3055 endmodule

```

yosys> help \$original_tag

Simulation model (verilog)

Listing 9.94: simlib.v

```
3072 module \${original_tag} (A, Y);
3073
3074     parameter TAG = "";
3075     parameter WIDTH = 0;
3076
3077     input [WIDTH-1:0] A;
3078     output [WIDTH-1:0] Y;
3079
3080     assign Y = A;
3081
3082 endmodule
```

yosys> help \$overwrite_tag

Simulation model (verilog)

Listing 9.95: simlib.v

```
3059 module \${overwrite_tag} (A, SET, CLR);
3060
3061     parameter TAG = "";
3062     parameter WIDTH = 0;
3063
3064     input [WIDTH-1:0] A;
3065     input [WIDTH-1:0] SET;
3066     input [WIDTH-1:0] CLR;
3067
3068 endmodule
```

yosys> help \$set_tag

Simulation model (verilog)

Listing 9.96: simlib.v

```
3029 module \${set_tag} (A, SET, CLR, Y);
3030
3031     parameter TAG = "";
3032     parameter WIDTH = 0;
3033
3034     input [WIDTH-1:0] A;
3035     input [WIDTH-1:0] SET;
3036     input [WIDTH-1:0] CLR;
3037     output [WIDTH-1:0] Y;
3038
3039     assign Y = A;
3040
3041 endmodule
```


9.1.11 Debugging cells

The *\$print* cell is used to log the values of signals, akin to (and translatable to) the *\$display* and *\$write* family of tasks in Verilog. It has the following parameters:

FORMAT

The internal format string. The syntax is described below.

ARGS_WIDTH

The width (in bits) of the signal on the *ARGS* port.

TRG_ENABLE

True if triggered on specific signals defined in *TRG*; false if triggered whenever *ARGS* or *EN* change and *EN* is 1.

If *TRG_ENABLE* is true, the following parameters also apply:

TRG_WIDTH

The number of bits in the *TRG* port.

TRG_POLARITY

For each bit in *TRG*, 1 if that signal is positive-edge triggered, 0 if negative-edge triggered.

PRIORITY

When multiple *\$print* or *\$check* cells fire on the same trigger, they execute in descending priority order.

Ports:

TRG

The signals that control when this *\$print* cell is triggered.

If the width of this port is zero and *TRG_ENABLE* is true, the cell is triggered during initial evaluation (time zero) only.

EN

Enable signal for the whole cell.

ARGS

The values to be displayed, in format string order.

```
yosys> help $check
```

```
Simulation model (verilog)
```

Listing 9.97: simlib.v

```

2128 module \ $check (A, EN, TRG, ARGS);
2129
2130     parameter FLAVOR = "";
2131     parameter PRIORITY = 0;
2132
2133     parameter FORMAT = "";
2134     parameter ARGS_WIDTH = 0;
2135
2136     parameter TRG_ENABLE = 1;
2137     parameter TRG_WIDTH = 0;
2138     parameter TRG_POLARITY = 0;
```

(continues on next page)

(continued from previous page)

```
2139
2140     input A;
2141     input EN;
2142     input [TRG_WIDTH-1:0] TRG;
2143     input [ARGS_WIDTH-1:0] ARGS;
2144
2145 endmodule
```

```
yosys> help $print
```

```
Simulation model (verilog)
```

Listing 9.98: simlib.v

```
2108 module \ $print (EN, TRG, ARGS);
2109
2110     parameter PRIORITY = 0;
2111
2112     parameter FORMAT = "";
2113     parameter ARGS_WIDTH = 0;
2114
2115     parameter TRG_ENABLE = 1;
2116     parameter TRG_WIDTH = 0;
2117     parameter TRG_POLARITY = 0;
2118
2119     input EN;
2120     input [TRG_WIDTH-1:0] TRG;
2121     input [ARGS_WIDTH-1:0] ARGS;
2122
2123 endmodule
```

```
yosys> help $scopeinfo
```

```
Simulation model (verilog)
```

Listing 9.99: simlib.v

```

3100 module \${scopeinfo} ();
3101
3102     parameter TYPE = "";
3103
3104 endmodule

```

Format string syntax

The format string syntax resembles Python f-strings. Regular text is passed through unchanged until a format specifier is reached, starting with a `{`.

Format specifiers have the following syntax. Unless noted, all items are required:

- `{`
Denotes the start of the format specifier.
- size**
Signal size in bits; this many bits are consumed from the **ARGS** port by this specifier.
- `:`
Separates the size from the remaining items.
- justify**
> for right-justified, < for left-justified.
- padding**
0 for zero-padding, or a space for space-padding.
- width?**
(optional) The number of characters wide to pad to.
- base**
 - **b** for base-2 integers (binary)
 - **o** for base-8 integers (octal)
 - **d** for base-10 integers (decimal)
 - **h** for base-16 integers (hexadecimal)
 - **c** for ASCII characters/strings
 - **t** and **r** for simulation time (corresponding to `$time` and `$realtime`)

For integers, this item may follow:

- +?**
(optional, decimals only) Include a leading plus for non-negative numbers. This can assist with symmetry with negatives in tabulated output.

- signedness**
u for unsigned, **s** for signed. This distinction is only respected when rendering decimals.

ASCII characters/strings have no special options, but the signal size must be divisible by 8.

For simulation time, the signal size must be zero.

Finally:

}

Denotes the end of the format specifier.

Some example format specifiers:

- {8:>02hu} - 8-bit unsigned integer rendered as hexadecimal, right-justified, zero-padded to 2 characters wide.
- {32:< 15d+s} - 32-bit signed integer rendered as decimal, left-justified, space-padded to 15 characters wide, positive values prefixed with +.
- {16:< 10hu} - 16-bit unsigned integer rendered as hexadecimal, left-justified, space-padded to 10 characters wide.
- {0:>010t} - simulation time, right-justified, zero-padded to 10 characters wide.

To include literal { and } characters in your format string, use {{ and }} respectively.

It is an error for a format string to consume more or less bits from `ARGS` than the port width.

Values are never truncated, regardless of the specified width.

Note that further restrictions on allowable combinations of options may apply depending on the backend used.

For example, Verilog does not have a format specifier that allows zero-padding a string (i.e. more than 1 ASCII character), though zero-padding a single character is permitted.

Thus, while the RTLIL format specifier {8:>02c} translates to %02c, {16:>02c} cannot be represented in Verilog and will fail to emit. In this case, {16:> 02c} must be used, which translates to %2s.

9.1.12 Wire cells

Todo: Add information about `$slice` and `$concat` cells.

yosys> help \$concat

Concatenation of inputs into a single output ($Y = \{B, A\}$).

Properties

is_evaluable

Simulation model (verilog)

Listing 9.100: simlib.v

```
1503 module \ $concat (A, B, Y);
1504
1505     parameter A_WIDTH = 0;
1506     parameter B_WIDTH = 0;
1507
1508     input [A_WIDTH-1:0] A;
1509     input [B_WIDTH-1:0] B;
1510     output [A_WIDTH+B_WIDTH-1:0] Y;
1511
1512     assign Y = {B, A};
```

(continues on next page)

(continued from previous page)

```

1513
1514 endmodule

```

```
yosys> help $slice
```

```

    Properties
        is_evaluable

    Simulation model (verilog)

```

Listing 9.101: simlib.v

```

1482 module \ $slice (A, Y);
1483
1484     parameter OFFSET = 0;
1485     parameter A_WIDTH = 0;
1486     parameter Y_WIDTH = 0;
1487
1488     input [A_WIDTH-1:0] A;
1489     output [Y_WIDTH-1:0] Y;
1490
1491     assign Y = A >> OFFSET;
1492
1493 endmodule

```

9.2 Gate-level cells

For gate level logic networks, fixed function single bit cells are used that do not provide any parameters.

Simulation models for these cells can be found in the file `techlibs/common/simcells.v` in the Yosys source tree.

In most cases gate level logic networks are created from RTL networks using the techmap pass. The flip-flop cells from the gate level logic network can be mapped to physical flip-flop cells from a Liberty file using the dfflibmap pass. The combinatorial logic cells can be mapped to physical cells from a Liberty file via ABC using the abc pass.

9.2.1 Combinatorial cells (simple)

Table 9.4: Cell types for gate level combinatorial cells (simple)

Verilog	Cell Type
$Y = A$	<code>\$_BUF_</code>
$Y = \sim A$	<code>\$_NOT_</code>
$Y = A \& B$	<code>\$_AND_</code>
$Y = \sim(A \& B)$	<code>\$_NAND_</code>
$Y = A B$	<code>\$_OR_</code>
$Y = \sim(A B)$	<code>\$_NOR_</code>
$Y = A \wedge B$	<code>\$_XOR_</code>
$Y = \sim(A \wedge B)$	<code>\$_XNOR_</code>
$Y = S ? B : A$	<code>\$_MUX_</code>

yosys> help \$_AND_

A 2-input AND gate.

Truth table:	A B Y
	-----+---
	0 0 0
	0 1 0
	1 0 0
	1 1 1

Properties

is_evaluable

Simulation model (verilog)

Listing 9.102: simcells.v

```
78 module \$_AND_ (A, B, Y);
79     input A, B;
80     output Y;
81     assign Y = A & B;
82 endmodule
```

yosys> help \$_BUF_

A buffer. This cell type is always optimized away by the opt_clean pass.

Truth table:	A Y
	---+---
	0 0
	1 1

Properties

is_evaluable

Simulation model (verilog)

Listing 9.103: simcells.v

```
40 module \$_BUF_ (A, Y);
41     input A;
42     output Y;
43     assign Y = A;
44 endmodule
```

yosys> help \$_MUX_

A 2-input MUX gate.

Truth table:	A B S Y
	-----+---
	a - 0 a
	- b 1 b

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.104: simcells.v

```

236 module \$_MUX_ (A, B, S, Y);
237     input A, B, S;
238     output Y;
239     assign Y = S ? B : A;
240 endmodule

```

yosys> help \$_NAND_

A 2-input NAND gate.

Truth table:	A B Y
	-----+---
	0 0 1
	0 1 1
	1 0 1
	1 1 0

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.105: simcells.v

```

98 module \$_NAND_ (A, B, Y);
99     input A, B;
100     output Y;
101     assign Y = ~(A & B);
102 endmodule

```

yosys> help \$_NOR_

A 2-input NOR gate.

Truth table:	A B Y
	-----+---
	0 0 1
	0 1 0
	1 0 0
	1 1 0

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.106: simcells.v

```
138 module \$_NOR_ (A, B, Y);
139     input A, B;
140     output Y;
141     assign Y = ~(A | B);
142 endmodule
```

yosys> help \\$_NOT_

An inverter gate.

Truth table:	A Y
	---+---
	0 1
	1 0

Properties

is_evaluable

Simulation model (verilog)

Listing 9.107: simcells.v

```
58 module \$_NOT_ (A, Y);
59     input A;
60     output Y;
61     assign Y = ~A;
62 endmodule
```

yosys> help \\$_OR_

A 2-input OR gate.

Truth table:	A B Y
	-----+---
	0 0 0
	0 1 1
	1 0 1
	1 1 1

Properties

is_evaluable

Simulation model (verilog)

Listing 9.108: simcells.v

```
118 module \$_OR_ (A, B, Y);
119     input A, B;
120     output Y;
121     assign Y = A | B;
122 endmodule
```


yosys> help \$_XNOR_

A 2-input XNOR gate.

Truth table:	A	B	Y
	0	0	1
	0	1	0
	1	0	0
	1	1	1

Properties

is_evaluable

Simulation model (verilog)

Listing 9.109: simcells.v

```

178 module \$_XNOR_ (A, B, Y);
179     input A, B;
180     output Y;
181     assign Y = ~(A ^ B);
182 endmodule

```

yosys> help \$_XOR_

A 2-input XOR gate.

Truth table:	A	B	Y
	0	0	0
	0	1	1
	1	0	1
	1	1	0

Properties

is_evaluable

Simulation model (verilog)

Listing 9.110: simcells.v

```

158 module \$_XOR_ (A, B, Y);
159     input A, B;
160     output Y;
161     assign Y = A ^ B;
162 endmodule

```

9.2.2 Combinatorial cells (combined)

These cells combine two or more combinatorial cells (simple) into a single cell.

Table 9.5: Cell types for gate level combinatorial cells (combined)

Verilog	Cell Type
$Y = A \& \sim B$	<code>\$_ANDNOT_</code>
$Y = A \mid \sim B$	<code>\$_ORNOT_</code>
$Y = \sim((A \& B) \mid C)$	<code>\$_AOI3_</code>
$Y = \sim((A \mid B) \& C)$	<code>\$_OAI3_</code>
$Y = \sim((A \& B) \mid (C \& D))$	<code>\$_AOI4_</code>
$Y = \sim((A \mid B) \& (C \mid D))$	<code>\$_OAI4_</code>
$Y = \sim(S ? B : A)$	<code>\$_NMUX_</code>
(see below)	<code>\$_MUX4_</code>
(see below)	<code>\$_MUX8_</code>
(see below)	<code>\$_MUX16_</code>

The `$_MUX4_`, `$_MUX8_` and `$_MUX16_` cells are used to model wide muxes, and correspond to the following Verilog code:

```

// $_MUX4_
assign Y = T ? (S ? D : C) :
           (S ? B : A);

// $_MUX8_
assign Y = U ? T ? (S ? H : G) :
           (S ? F : E) :
           T ? (S ? D : C) :
           (S ? B : A);

// $_MUX16_
assign Y = V ? U ? T ? (S ? P : O) :
           (S ? N : M) :
           T ? (S ? L : K) :
           (S ? J : I) :
           U ? T ? (S ? H : G) :
           (S ? F : E) :
           T ? (S ? D : C) :
           (S ? B : A);

```

```

yosys> help $_ANDNOT_
      A 2-input AND-NOT gate.

```

```

Truth table:   A B | Y
               -----+---
               0 0 | 0
               0 1 | 0
               1 0 | 1
               1 1 | 0

```

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.111: simcells.v

```

198 module \$_ANDNOT_ (A, B, Y);
199     input A, B;
200     output Y;
201     assign Y = A & (~B);
202 endmodule

```

yosys> help \$_AOI3_

A 3-input And-Or-Invert gate.

```

Truth table:   A B C | Y
               -----+---
               0 0 0 | 1
               0 0 1 | 0
               0 1 0 | 1
               0 1 1 | 0
               1 0 0 | 1
               1 0 1 | 0
               1 1 0 | 0
               1 1 1 | 0

```

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.112: simcells.v

```

367 module \$_AOI3_ (A, B, C, Y);
368     input A, B, C;
369     output Y;
370     assign Y = ~((A & B) | C);
371 endmodule

```

yosys> help \$_AOI4_

A 4-input And-Or-Invert gate.

```

Truth table:   A B C D | Y
               -----+---

```

(continues on next page)

(continued from previous page)

```

0 0 0 0 | 1
0 0 0 1 | 1
0 0 1 0 | 1
0 0 1 1 | 0
0 1 0 0 | 1
0 1 0 1 | 1
0 1 1 0 | 1
0 1 1 1 | 0
1 0 0 0 | 1
1 0 0 1 | 1
1 0 1 0 | 1
1 0 1 1 | 0
1 1 0 0 | 0
1 1 0 1 | 0
1 1 1 0 | 0
1 1 1 1 | 0

```

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.113: simcells.v

```

423 module \$_AOI4_ (A, B, C, D, Y);
424     input A, B, C, D;
425     output Y;
426     assign Y = ~((A & B) | (C & D));
427 endmodule

```

yosys> help \$_MUX16_

A 16-input MUX gate.

```

Truth table:  A B C D E F G H I J K L M N O P S T U V | Y
-----+-----
a - - - - - - - - - - - - - - 0 0 0 0 | a
- b - - - - - - - - - - - - - - 1 0 0 0 | b
- - c - - - - - - - - - - - - - - 0 1 0 0 | c
- - - d - - - - - - - - - - - - - - 1 1 0 0 | d
- - - - e - - - - - - - - - - - - - - 0 0 1 0 | e
- - - - - f - - - - - - - - - - - - - - 1 0 1 0 | f
- - - - - - g - - - - - - - - - - - - - - 0 1 1 0 | g
- - - - - - - h - - - - - - - - - - - - - - 1 1 1 0 | h
- - - - - - - - i - - - - - - - - - - - - - - 0 0 0 1 | i
- - - - - - - - - j - - - - - - - - - - - - - - 1 0 0 1 | j
- - - - - - - - - - k - - - - - - - - - - - - - - 0 1 0 1 | k
- - - - - - - - - - - l - - - - - - - - - - - - - - 1 1 0 1 | l
- - - - - - - - - - - - m - - - - - - - - - - - - - - 0 0 1 1 | m
- - - - - - - - - - - - - n - - - - - - - - - - - - - - 1 0 1 1 | n
- - - - - - - - - - - - - - o - 0 1 1 1 | o
- - - - - - - - - - - - - - p 1 1 1 1 | p

```

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.114: simcells.v

```

336 module \$_MUX16_ (A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, S, T, U, V, Y);
337     input A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, S, T, U, V;
338     output Y;
339     assign Y = V ? U ? T ? (S ? P : O) :
340                (S ? N : M) :
341                T ? (S ? L : K) :
342                (S ? J : I) :
343                U ? T ? (S ? H : G) :
344                (S ? F : E) :
345                T ? (S ? D : C) :
346                (S ? B : A);
347 endmodule

```

yosys> help \$_MUX4_

A 4-input MUX gate.

Truth table:	A	B	C	D	S	T		Y
	a	-	-	-	0	0		a
	-	b	-	-	1	0		b
	-	-	c	-	0	1		c
	-	-	-	d	1	1		d

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.115: simcells.v

```

276 module \$_MUX4_ (A, B, C, D, S, T, Y);
277     input A, B, C, D, S, T;
278     output Y;
279     assign Y = T ? (S ? D : C) :
280                (S ? B : A);
281 endmodule

```

yosys> help \$_MUX8_

An 8-input MUX gate.

Truth table:	A	B	C	D	E	F	G	H	S	T	U		Y
	a	-	-	-	-	-	-	-	0	0	0		a
	-	b	-	-	-	-	-	-	1	0	0		b
	-	-	c	-	-	-	-	-	0	1	0		c
	-	-	-	d	-	-	-	-	1	1	0		d

(continues on next page)

(continued from previous page)

```

- - - - e - - - 0 0 1 | e
- - - - - f - - 1 0 1 | f
- - - - - - g - 0 1 1 | g
- - - - - - - h 1 1 1 | h

```

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.116: simcells.v

```

301 module \$_MUX8_ (A, B, C, D, E, F, G, H, S, T, U, Y);
302     input A, B, C, D, E, F, G, H, S, T, U;
303     output Y;
304     assign Y = U ? T ? (S ? H : G) :
305                 (S ? F : E) :
306                 T ? (S ? D : C) :
307                 (S ? B : A);
308 endmodule

```

yosys> help \$_NMUX_

A 2-input inverting MUX gate.

```

Truth table:   A B S | Y
               -----+---
               0 - 0 | 1
               1 - 0 | 0
               - 0 1 | 1
               - 1 1 | 0

```

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.117: simcells.v

```

256 module \$_NMUX_ (A, B, S, Y);
257     input A, B, S;
258     output Y;
259     assign Y = S ? !B : !A;
260 endmodule

```

yosys> help \$_OAI3_

A 3-input Or-And-Invert gate.

```

Truth table:   A B C | Y
               -----+---
               0 0 0 | 1
               0 0 1 | 1

```

(continues on next page)

(continued from previous page)

0	1	0		1
0	1	1		0
1	0	0		1
1	0	1		0
1	1	0		1
1	1	1		0

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.118: simcells.v

```

391 module \$_OAI3_ (A, B, C, Y);
392     input A, B, C;
393     output Y;
394     assign Y = ~((A | B) & C);
395 endmodule

```

yosys> help \$_OAI4_

A 4-input Or-And-Invert gate.

Truth table:	A	B	C	D		Y
	0	0	0	0		1
	0	0	0	1		1
	0	0	1	0		1
	0	0	1	1		1
	0	1	0	0		1
	0	1	0	1		0
	0	1	1	0		0
	0	1	1	1		0
	1	0	0	0		1
	1	0	0	1		0
	1	0	1	0		0
	1	0	1	1		0
	1	1	0	0		1
	1	1	0	1		0
	1	1	1	0		0
	1	1	1	1		0

Properties*is_evaluable*

Simulation model (verilog)

Listing 9.119: simcells.v

```

455 module \$_OAI4_ (A, B, C, D, Y);
456     input A, B, C, D;

```

(continues on next page)

(continued from previous page)

```

457     output Y;
458     assign Y = ~((A | B) & (C | D));
459 endmodule

```

yosys> help \$_ORNOT_

A 2-input OR-NOT gate.

```

Truth table:   A B | Y
               -----+---
                   0 0 | 1
                   0 1 | 0
                   1 0 | 1
                   1 1 | 1

```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.120: simcells.v

```

218 module \$_ORNOT_ (A, B, Y);
219     input A, B;
220     output Y;
221     assign Y = A | (~B);
222 endmodule

```

9.2.3 Flip-flop cells

The cell types `$_DFF_N_` and `$_DFF_P_` represent d-type flip-flops.

Table 9.6: Cell types for basic flip-flops

Verilog	Cell Type
<code>always @(negedge C) Q <= D</code>	<code>\$_DFF_N_</code>
<code>always @(posedge C) Q <= D</code>	<code>\$_DFF_P_</code>

The cell types `$_DFFE_[NP][NP]_` implement d-type flip-flops with enable. The values in the table for these cell types relate to the following Verilog code template.

```

always @(CLK_EDGE C)
    if (EN == EN_LVL)
        Q <= D;

```


Table 9.7: Cell types for gate level logic networks (FFs with enable)

<i>ClkEdge</i>	<i>EnLvl</i>	Cell Type
negedge	0	<code>\$_DFFE_NN_</code>
negedge	1	<code>\$_DFFE_NP_</code>
posedge	0	<code>\$_DFFE_PN_</code>
posedge	1	<code>\$_DFFE_PP_</code>

The cell types `$_DFF_[NP][NP][01]_` implement d-type flip-flops with asynchronous reset. The values in the table for these cell types relate to the following Verilog code template, where `RST_EDGE` is `posedge` if `RST_LVL` if 1, and `negedge` otherwise.

```
always @(CLK_EDGE C, RST_EDGE R)
  if (R == RST_LVL)
    Q <= RST_VAL;
  else
    Q <= D;
```

The cell types `$_SDFF_[NP][NP][01]_` implement d-type flip-flops with synchronous reset. The values in the table for these cell types relate to the following Verilog code template:

```
always @(CLK_EDGE C)
  if (R == RST_LVL)
    Q <= RST_VAL;
  else
    Q <= D;
```

Table 9.8: Cell types for gate level logic networks (FFs with reset)

<i>ClkEdge</i>	<i>RstLvl</i>	<i>RstVal</i>	Cell Type
negedge	0	0	<code>\$_DFF_NNO_</code> , <code>\$_SDFF_NNO_</code>
negedge	0	1	<code>\$_DFF_NN1_</code> , <code>\$_SDFF_NN1_</code>
negedge	1	0	<code>\$_DFF_NPO_</code> , <code>\$_SDFF_NPO_</code>
negedge	1	1	<code>\$_DFF_NP1_</code> , <code>\$_SDFF_NP1_</code>
posedge	0	0	<code>\$_DFF_PNO_</code> , <code>\$_SDFF_PNO_</code>
posedge	0	1	<code>\$_DFF_PN1_</code> , <code>\$_SDFF_PN1_</code>
posedge	1	0	<code>\$_DFF_PPO_</code> , <code>\$_SDFF_PPO_</code>
posedge	1	1	<code>\$_DFF_PP1_</code> , <code>\$_SDFF_PP1_</code>

The cell types `$_DFFE_[NP][NP][01][NP]_` implement d-type flip-flops with asynchronous reset and enable. The values in the table for these cell types relate to the following Verilog code template, where `RST_EDGE` is `posedge` if `RST_LVL` if 1, and `negedge` otherwise.

```
always @(CLK_EDGE C, RST_EDGE R)
  if (R == RST_LVL)
    Q <= RST_VAL;
  else if (EN == EN_LVL)
    Q <= D;
```

The cell types `$_SDFFE_[NP][NP][01][NP]_` implement d-type flip-flops with synchronous reset and enable, with reset having priority over enable. The values in the table for these cell types relate to the following Verilog code template:

```

always @(CLK_EDGE C)
  if (R == RST_LVL)
    Q <= RST_VAL;
  else if (EN == EN_LVL)
    Q <= D;

```

The cell types `$_SDFFCE_[NP][NP][01][NP]_` implement d-type flip-flops with synchronous reset and enable, with enable having priority over reset. The values in the table for these cell types relate to the following Verilog code template:

```

always @(CLK_EDGE C)
  if (EN == EN_LVL)
    if (R == RST_LVL)
      Q <= RST_VAL;
    else
      Q <= D;

```

Table 9.9: Cell types for gate level logic networks (FFs with reset and enable)

<i>ClkEdge</i>	<i>RstLvl</i>	<i>RstVal</i>	<i>EnLvl</i>	Cell Type
negedge	0	0	0	<code>\$_DFFE_NNON_</code> , <code>\$_SDFFE_NNON_</code> , <code>\$_SDFFCE_NNON_</code>
negedge	0	0	1	<code>\$_DFFE_NNOP_</code> , <code>\$_SDFFE_NNOP_</code> , <code>\$_SDFFCE_NNOP_</code>
negedge	0	1	0	<code>\$_DFFE_NN1N_</code> , <code>\$_SDFFE_NN1N_</code> , <code>\$_SDFFCE_NN1N_</code>
negedge	0	1	1	<code>\$_DFFE_NN1P_</code> , <code>\$_SDFFE_NN1P_</code> , <code>\$_SDFFCE_NN1P_</code>
negedge	1	0	0	<code>\$_DFFE_NPON_</code> , <code>\$_SDFFE_NPON_</code> , <code>\$_SDFFCE_NPON_</code>
negedge	1	0	1	<code>\$_DFFE_NPOP_</code> , <code>\$_SDFFE_NPOP_</code> , <code>\$_SDFFCE_NPOP_</code>
negedge	1	1	0	<code>\$_DFFE_NP1N_</code> , <code>\$_SDFFE_NP1N_</code> , <code>\$_SDFFCE_NP1N_</code>
negedge	1	1	1	<code>\$_DFFE_NP1P_</code> , <code>\$_SDFFE_NP1P_</code> , <code>\$_SDFFCE_NP1P_</code>
posedge	0	0	0	<code>\$_DFFE_PNON_</code> , <code>\$_SDFFE_PNON_</code> , <code>\$_SDFFCE_PNON_</code>
posedge	0	0	1	<code>\$_DFFE_PNOP_</code> , <code>\$_SDFFE_PNOP_</code> , <code>\$_SDFFCE_PNOP_</code>
posedge	0	1	0	<code>\$_DFFE_PN1N_</code> , <code>\$_SDFFE_PN1N_</code> , <code>\$_SDFFCE_PN1N_</code>
posedge	0	1	1	<code>\$_DFFE_PN1P_</code> , <code>\$_SDFFE_PN1P_</code> , <code>\$_SDFFCE_PN1P_</code>
posedge	1	0	0	<code>\$_DFFE_PPON_</code> , <code>\$_SDFFE_PPON_</code> , <code>\$_SDFFCE_PPON_</code>
posedge	1	0	1	<code>\$_DFFE_PPOP_</code> , <code>\$_SDFFE_PPOP_</code> , <code>\$_SDFFCE_PPOP_</code>
posedge	1	1	0	<code>\$_DFFE_PP1N_</code> , <code>\$_SDFFE_PP1N_</code> , <code>\$_SDFFCE_PP1N_</code>
posedge	1	1	1	<code>\$_DFFE_PP1P_</code> , <code>\$_SDFFE_PP1P_</code> , <code>\$_SDFFCE_PP1P_</code>

The cell types `$_DFFSR_[NP][NP][NP]_` implement d-type flip-flops with asynchronous set and reset. The values in the table for these cell types relate to the following Verilog code template, where `RST_EDGE` is `posedge` if `RST_LVL` if 1, `negedge` otherwise, and `SET_EDGE` is `posedge` if `SET_LVL` if 1, `negedge` otherwise.

```

always @(CLK_EDGE C, RST_EDGE R, SET_EDGE S)
  if (R == RST_LVL)
    Q <= 0;
  else if (S == SET_LVL)
    Q <= 1;
  else
    Q <= D;

```

Table 9.10: Cell types for gate level logic networks (FFs with set and reset)

<i>ClkEdge</i>	<i>SetLvl</i>	<i>RstLvl</i>	Cell Type
negedge	0	0	<code>\$_DFFSR_NNN_</code>
negedge	0	1	<code>\$_DFFSR_NNP_</code>
negedge	1	0	<code>\$_DFFSR_NPN_</code>
negedge	1	1	<code>\$_DFFSR_NPP_</code>
posedge	0	0	<code>\$_DFFSR_PNN_</code>
posedge	0	1	<code>\$_DFFSR_PNP_</code>
posedge	1	0	<code>\$_DFFSR_PPN_</code>
posedge	1	1	<code>\$_DFFSR_PPP_</code>

The cell types `$_DFFSRE_[NP][NP][NP][NP]_` implement d-type flip-flops with asynchronous set and reset and enable. The values in the table for these cell types relate to the following Verilog code template, where `RST_EDGE` is `posedge` if `RST_LVL` if 1, `negedge` otherwise, and `SET_EDGE` is `posedge` if `SET_LVL` if 1, `negedge` otherwise.

```
always @(CLK_EDGE C, RST_EDGE R, SET_EDGE S)
  if (R == RST_LVL)
    Q <= 0;
  else if (S == SET_LVL)
    Q <= 1;
  else if (E == EN_LVL)
    Q <= D;
```

Table 9.11: Cell types for gate level logic networks (FFs with set and reset and enable)

<i>ClkEdge</i>	<i>SetLvl</i>	<i>RstLvl</i>	<i>EnLvl</i>	Cell Type
negedge	0	0	0	<code>\$_DFFSRE_NNNN_</code>
negedge	0	0	1	<code>\$_DFFSRE_NNNP_</code>
negedge	0	1	0	<code>\$_DFFSRE_NNPN_</code>
negedge	0	1	1	<code>\$_DFFSRE_NNPP_</code>
negedge	1	0	0	<code>\$_DFFSRE_NPNN_</code>
negedge	1	0	1	<code>\$_DFFSRE_NPNP_</code>
negedge	1	1	0	<code>\$_DFFSRE_NPPN_</code>
negedge	1	1	1	<code>\$_DFFSRE_NPPP_</code>
posedge	0	0	0	<code>\$_DFFSRE_PNNN_</code>
posedge	0	0	1	<code>\$_DFFSRE_PNNP_</code>
posedge	0	1	0	<code>\$_DFFSRE_PNPN_</code>
posedge	0	1	1	<code>\$_DFFSRE_PNPP_</code>
posedge	1	0	0	<code>\$_DFFSRE_PPNN_</code>
posedge	1	0	1	<code>\$_DFFSRE_PPNP_</code>
posedge	1	1	0	<code>\$_DFFSRE_PPPN_</code>
posedge	1	1	1	<code>\$_DFFSRE_PPPP_</code>

Todo: flip-flops with async load, `$_ALDFFE?[NP]{2,3}_`

yosys> help \$_ALDFFE_NNN_

A negative edge D-type flip-flop with negative polarity async load and negative polarity clock enable.

Truth table:	D	C	L	AD	E		Q
	-	-	0	a	-		a
	d	\	-	-	0		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.121: simcells.v

```

1420 module \$_ALDFFE_NNN_ (D, C, L, AD, E, Q);
1421     input D, C, L, AD, E;
1422     output reg Q;
1423     always @(negedge C or negedge L) begin
1424         if (L == 0)
1425             Q <= AD;
1426         else if (E == 0)
1427             Q <= D;
1428     end
1429 endmodule

```

yosys> help \$_ALDFFE_NNP_

A negative edge D-type flip-flop with negative polarity async load and positive polarity clock enable.

Truth table:	D	C	L	AD	E		Q
	-	-	0	a	-		a
	d	\	-	-	1		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.122: simcells.v

```

1445 module \$_ALDFFE_NNP_ (D, C, L, AD, E, Q);
1446     input D, C, L, AD, E;
1447     output reg Q;
1448     always @(negedge C or negedge L) begin
1449         if (L == 0)
1450             Q <= AD;
1451         else if (E == 1)
1452             Q <= D;
1453     end
1454 endmodule

```

yosys> help \$_ALDFFE_NPN_

A negative edge D-type flip-flop with positive polarity async load and negative polarity clock enable.

Truth table:	D	C	L	AD	E		Q
	-	-	-	-	-		q

(continues on next page)

(continued from previous page)

```

- - 1 a - | a
d \ - - 0 | d
- - - - - | q

```

Simulation model (verilog)

Listing 9.123: simcells.v

```

1470 module \$_ALDFFE_NPN_ (D, C, L, AD, E, Q);
1471     input D, C, L, AD, E;
1472     output reg Q;
1473     always @(negedge C or posedge L) begin
1474         if (L == 1)
1475             Q <= AD;
1476         else if (E == 0)
1477             Q <= D;
1478     end
1479 endmodule

```

yosys> help \$_ALDFFE_NPP_

A negative edge D-type flip-flop with positive polarity async load and positive polarity clock enable.

```

Truth table:   D C L AD E | Q
               +-----+
               - - 1 a - | a
               d \ - - 1 | d
               - - - - - | q

```

Simulation model (verilog)

Listing 9.124: simcells.v

```

1495 module \$_ALDFFE_NPP_ (D, C, L, AD, E, Q);
1496     input D, C, L, AD, E;
1497     output reg Q;
1498     always @(negedge C or posedge L) begin
1499         if (L == 1)
1500             Q <= AD;
1501         else if (E == 1)
1502             Q <= D;
1503     end
1504 endmodule

```

yosys> help \$_ALDFFE_PNN_

A positive edge D-type flip-flop with negative polarity async load and negative polarity clock enable.

```

Truth table:   D C L AD E | Q
               +-----+
               - - 0 a - | a
               d / - - 0 | d
               - - - - - | q

```

Simulation model (verilog)

Listing 9.125: simcells.v

```

1520 module \$_ALDFFE_PNN_ (D, C, L, AD, E, Q);
1521     input D, C, L, AD, E;
1522     output reg Q;
1523     always @(posedge C or negedge L) begin
1524         if (L == 0)
1525             Q <= AD;
1526         else if (E == 0)
1527             Q <= D;
1528     end
1529 endmodule

```

yosys> help \\$_ALDFFE_PNP_

A positive edge D-type flip-flop with negative polarity async load and positive polarity clock enable.

```

Truth table:   D C L AD E | Q
-----+-----
              - - 0 a - | a
              d / - - 1 | d
              - - - - - | q

```

Simulation model (verilog)

Listing 9.126: simcells.v

```

1545 module \$_ALDFFE_PNP_ (D, C, L, AD, E, Q);
1546     input D, C, L, AD, E;
1547     output reg Q;
1548     always @(posedge C or negedge L) begin
1549         if (L == 0)
1550             Q <= AD;
1551         else if (E == 1)
1552             Q <= D;
1553     end
1554 endmodule

```

yosys> help \\$_ALDFFE_PPN_

A positive edge D-type flip-flop with positive polarity async load and negative polarity clock enable.

```

Truth table:   D C L AD E | Q
-----+-----
              - - 1 a - | a
              d / - - 0 | d
              - - - - - | q

```

Simulation model (verilog)

Listing 9.127: simcells.v

```

1570 module \$_ALDFFE_PPN_ (D, C, L, AD, E, Q);
1571     input D, C, L, AD, E;
1572     output reg Q;

```

(continues on next page)

(continued from previous page)

```

1573     always @(posedge C or posedge L) begin
1574         if (L == 1)
1575             Q <= AD;
1576         else if (E == 0)
1577             Q <= D;
1578     end
1579 endmodule

```

yosys> help \$_ALDFFE_PPP_

A positive edge D-type flip-flop with positive polarity async load and positive polarity clock enable.

Truth table:	D	C	L	AD	E		Q
	-	-	1	a	-		a
	d	/	-	-	1		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.128: simcells.v

```

1595 module \$_ALDFFE_PPP_ (D, C, L, AD, E, Q);
1596     input D, C, L, AD, E;
1597     output reg Q;
1598     always @(posedge C or posedge L) begin
1599         if (L == 1)
1600             Q <= AD;
1601         else if (E == 1)
1602             Q <= D;
1603     end
1604 endmodule

```

yosys> help \$_ALDFF_NN_

A negative edge D-type flip-flop with negative polarity async load.

Truth table:	D	C	L	AD		Q
	-	-	0	a		a
	d	\	-	-		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.129: simcells.v

```

1323 module \$_ALDFF_NN_ (D, C, L, AD, Q);
1324     input D, C, L, AD;
1325     output reg Q;
1326     always @(negedge C or negedge L) begin
1327         if (L == 0)
1328             Q <= AD;
1329     else

```

(continues on next page)

(continued from previous page)

```

1330         Q <= D;
1331     end
1332 endmodule

```

yosys> help \$_ALDFF_NP_

A negative edge D-type flip-flop with positive polarity async load.

Truth table:	D	C	L	AD		Q
	-----+---					
	-	-	1	a		a
	d	\	-	-		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.130: simcells.v

```

1347 module \$_ALDFF_NP_ (D, C, L, AD, Q);
1348     input D, C, L, AD;
1349     output reg Q;
1350     always @(negedge C or posedge L) begin
1351         if (L == 1)
1352             Q <= AD;
1353         else
1354             Q <= D;
1355     end
1356 endmodule

```

yosys> help \$_ALDFF_PN_

A positive edge D-type flip-flop with negative polarity async load.

Truth table:	D	C	L	AD		Q
	-----+---					
	-	-	0	a		a
	d	/	-	-		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.131: simcells.v

```

1371 module \$_ALDFF_PN_ (D, C, L, AD, Q);
1372     input D, C, L, AD;
1373     output reg Q;
1374     always @(posedge C or negedge L) begin
1375         if (L == 0)
1376             Q <= AD;
1377         else
1378             Q <= D;
1379     end
1380 endmodule

```


yosys> help \$_ALDFF_PP_

A positive edge D-type flip-flop with positive polarity async load.

Truth table:	D	C	L	AD		Q
	-	-	1	a		a
	d	/	-	-		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.132: simcells.v

```

1395 module \$_ALDFF_PP_ (D, C, L, AD, Q);
1396     input D, C, L, AD;
1397     output reg Q;
1398     always @(posedge C or posedge L) begin
1399         if (L == 1)
1400             Q <= AD;
1401         else
1402             Q <= D;
1403     end
1404 endmodule

```

yosys> help \$_DFFE_NNON_

A negative edge D-type flip-flop with negative polarity reset and negative polarity clock enable.

Truth table:	D	C	R	E		Q
	-	-	0	-		0
	d	\	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.133: simcells.v

```

924 module \$_DFFE_NNON_ (D, C, R, E, Q);
925     input D, C, R, E;
926     output reg Q;
927     always @(negedge C or negedge R) begin
928         if (R == 0)
929             Q <= 0;
930         else if (E == 0)
931             Q <= D;
932     end
933 endmodule

```

yosys> help \$_DFFE_NNOP_

A negative edge D-type flip-flop with negative polarity reset and positive polarity clock enable.

Truth table:	D	C	R	E		Q
	-	-	-	-		q

(continues on next page)

(continued from previous page)

```

- - 0 - | 0
d \ - 1 | d
- - - - | q

```

Simulation model (verilog)

Listing 9.134: simcells.v

```

949 module \$_DFFE_NNOP_ (D, C, R, E, Q);
950     input D, C, R, E;
951     output reg Q;
952     always @(negedge C or negedge R) begin
953         if (R == 0)
954             Q <= 0;
955         else if (E == 1)
956             Q <= D;
957     end
958 endmodule

```

yosys> help \$_DFFE_NN1N_

A negative edge D-type flip-flop with negative polarity set and negative polarity clock enable.

```

Truth table:   D C R E | Q
               +-----+
               - - 0 - | 1
               d \ - 0 | d
               - - - - | q

```

Simulation model (verilog)

Listing 9.135: simcells.v

```

974 module \$_DFFE_NN1N_ (D, C, R, E, Q);
975     input D, C, R, E;
976     output reg Q;
977     always @(negedge C or negedge R) begin
978         if (R == 0)
979             Q <= 1;
980         else if (E == 0)
981             Q <= D;
982     end
983 endmodule

```

yosys> help \$_DFFE_NN1P_

A negative edge D-type flip-flop with negative polarity set and positive polarity clock enable.

```

Truth table:   D C R E | Q
               +-----+
               - - 0 - | 1
               d \ - 1 | d
               - - - - | q

```

Simulation model (verilog)

Listing 9.136: simcells.v

```

999 module \$_DFFE_NN1P_ (D, C, R, E, Q);
1000     input D, C, R, E;
1001     output reg Q;
1002     always @(negedge C or negedge R) begin
1003         if (R == 0)
1004             Q <= 1;
1005         else if (E == 1)
1006             Q <= D;
1007     end
1008 endmodule

```

yosys> help \$_DFFE_NN_

A negative edge D-type flip-flop with negative polarity enable.

Truth table:	D	C	E		Q
	-----+---				
	d	\	0		d
	-	-	-		q

Simulation model (verilog)

Listing 9.137: simcells.v

```

650 module \$_DFFE_NN_ (D, C, E, Q);
651     input D, C, E;
652     output reg Q;
653     always @(negedge C) begin
654         if (!E) Q <= D;
655     end
656 endmodule

```

yosys> help \$_DFFE_NPON_

A negative edge D-type flip-flop with positive polarity reset and negative polarity clock enable.

Truth table:	D	C	R	E		Q
	-----+---					
	-	-	1	-		0
	d	\	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.138: simcells.v

```

1024 module \$_DFFE_NPON_ (D, C, R, E, Q);
1025     input D, C, R, E;
1026     output reg Q;
1027     always @(negedge C or posedge R) begin
1028         if (R == 1)
1029             Q <= 0;
1030         else if (E == 0)

```

(continues on next page)

(continued from previous page)

```

1031         Q <= D;
1032     end
1033 endmodule

```

yosys> help \$_DFFE_NPOP_

A negative edge D-type flip-flop with positive polarity reset and positive polarity clock enable.

Truth table:	D	C	R	E	Q
	-	-	1	-	0
	d	\	-	1	d
	-	-	-	-	q

Simulation model (verilog)

Listing 9.139: simcells.v

```

1049 module \$_DFFE_NPOP_ (D, C, R, E, Q);
1050     input D, C, R, E;
1051     output reg Q;
1052     always @(negedge C or posedge R) begin
1053         if (R == 1)
1054             Q <= 0;
1055         else if (E == 1)
1056             Q <= D;
1057     end
1058 endmodule

```

yosys> help \$_DFFE_NP1N_

A negative edge D-type flip-flop with positive polarity set and negative polarity clock enable.

Truth table:	D	C	R	E	Q
	-	-	1	-	1
	d	\	-	0	d
	-	-	-	-	q

Simulation model (verilog)

Listing 9.140: simcells.v

```

1074 module \$_DFFE_NP1N_ (D, C, R, E, Q);
1075     input D, C, R, E;
1076     output reg Q;
1077     always @(negedge C or posedge R) begin
1078         if (R == 1)
1079             Q <= 1;
1080         else if (E == 0)
1081             Q <= D;
1082     end
1083 endmodule

```

yosys> help \$_DFFE_NP1P_

A negative edge D-type flip-flop with positive polarity set and positive polarity clock enable.

Truth table:	D	C	R	E	Q
	-	-	1	-	1
	d	\	-	1	d
	-	-	-	-	q

Simulation model (verilog)

Listing 9.141: simcells.v

```

1099 module \$_DFFE_NP1P_ (D, C, R, E, Q);
1100     input D, C, R, E;
1101     output reg Q;
1102     always @(negedge C or posedge R) begin
1103         if (R == 1)
1104             Q <= 1;
1105         else if (E == 1)
1106             Q <= D;
1107     end
1108 endmodule

```

yosys> help \$_DFFE_NP_

A negative edge D-type flip-flop with positive polarity enable.

Truth table:	D	C	E	Q
	d	\	1	d
	-	-	-	q

Simulation model (verilog)

Listing 9.142: simcells.v

```

670 module \$_DFFE_NP_ (D, C, E, Q);
671     input D, C, E;
672     output reg Q;
673     always @(negedge C) begin
674         if (E) Q <= D;
675     end
676 endmodule

```

yosys> help \$_DFFE_PN0N_

A positive edge D-type flip-flop with negative polarity reset and negative polarity clock enable.

Truth table:	D	C	R	E	Q
	-	-	0	-	0
	d	/	-	0	d
	-	-	-	-	q

Simulation model (verilog)

Listing 9.143: simcells.v

```

1124 module \$_DFFE_PNON_ (D, C, R, E, Q);
1125     input D, C, R, E;
1126     output reg Q;
1127     always @(posedge C or negedge R) begin
1128         if (R == 0)
1129             Q <= 0;
1130         else if (E == 0)
1131             Q <= D;
1132     end
1133 endmodule

```

yosys> help \$_DFFE_PNOP_

A positive edge D-type flip-flop with negative polarity reset and positive polarity clock enable.

Truth table:	D	C	R	E		Q
	-	-	0	-		0
	d	/	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.144: simcells.v

```

1149 module \$_DFFE_PNOP_ (D, C, R, E, Q);
1150     input D, C, R, E;
1151     output reg Q;
1152     always @(posedge C or negedge R) begin
1153         if (R == 0)
1154             Q <= 0;
1155         else if (E == 1)
1156             Q <= D;
1157     end
1158 endmodule

```

yosys> help \$_DFFE_PN1N_

A positive edge D-type flip-flop with negative polarity set and negative polarity clock enable.

Truth table:	D	C	R	E		Q
	-	-	0	-		1
	d	/	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.145: simcells.v

```

1174 module \$_DFFE_PN1N_ (D, C, R, E, Q);
1175     input D, C, R, E;
1176     output reg Q;

```

(continues on next page)

(continued from previous page)

```

1177     always @(posedge C or negedge R) begin
1178         if (R == 0)
1179             Q <= 1;
1180         else if (E == 0)
1181             Q <= D;
1182     end
1183 endmodule

```

yosys> help \$_DFFE_PN1P_

A positive edge D-type flip-flop with negative polarity set and positive polarity clock enable.

Truth table:	D	C	R	E		Q
	-	-	0	-		1
	d	/	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.146: simcells.v

```

1199 module \$_DFFE_PN1P_ (D, C, R, E, Q);
1200     input D, C, R, E;
1201     output reg Q;
1202     always @(posedge C or negedge R) begin
1203         if (R == 0)
1204             Q <= 1;
1205         else if (E == 1)
1206             Q <= D;
1207     end
1208 endmodule

```

yosys> help \$_DFFE_PN_

A positive edge D-type flip-flop with negative polarity enable.

Truth table:	D	C	E		Q
	d	/	0		d
	-	-	-		q

Simulation model (verilog)

Listing 9.147: simcells.v

```

690 module \$_DFFE_PN_ (D, C, E, Q);
691     input D, C, E;
692     output reg Q;
693     always @(posedge C) begin
694         if (!E) Q <= D;
695     end
696 endmodule

```

yosys> help \$_DFFE_PPON_

A positive edge D-type flip-flop with positive polarity reset and negative polarity clock enable.

Truth table:	D	C	R	E	Q
	-	-	1	-	0
	d	/	-	0	d
	-	-	-	-	q

Simulation model (verilog)

Listing 9.148: simcells.v

```

1224 module \$_DFFE_PPON_ (D, C, R, E, Q);
1225     input D, C, R, E;
1226     output reg Q;
1227     always @(posedge C or posedge R) begin
1228         if (R == 1)
1229             Q <= 0;
1230         else if (E == 0)
1231             Q <= D;
1232     end
1233 endmodule

```

yosys> help \$_DFFE_PPOP_

A positive edge D-type flip-flop with positive polarity reset and positive polarity clock enable.

Truth table:	D	C	R	E	Q
	-	-	1	-	0
	d	/	-	1	d
	-	-	-	-	q

Simulation model (verilog)

Listing 9.149: simcells.v

```

1249 module \$_DFFE_PPOP_ (D, C, R, E, Q);
1250     input D, C, R, E;
1251     output reg Q;
1252     always @(posedge C or posedge R) begin
1253         if (R == 1)
1254             Q <= 0;
1255         else if (E == 1)
1256             Q <= D;
1257     end
1258 endmodule

```

yosys> help \$_DFFE_PP1N_

A positive edge D-type flip-flop with positive polarity set and negative polarity clock enable.

Truth table:	D	C	R	E	Q
	-	-	-	-	q

(continues on next page)

(continued from previous page)

```

- - 1 - | 1
d / - 0 | d
- - - - | q

```

Simulation model (verilog)

Listing 9.150: simcells.v

```

1274 module \$_DFFE_PP1N_ (D, C, R, E, Q);
1275     input D, C, R, E;
1276     output reg Q;
1277     always @(posedge C or posedge R) begin
1278         if (R == 1)
1279             Q <= 1;
1280         else if (E == 0)
1281             Q <= D;
1282     end
1283 endmodule

```

yosys> help \$_DFFE_PP1P_

A positive edge D-type flip-flop with positive polarity set and positive polarity clock enable.

```

Truth table:   D C R E | Q
               +-----+
               - - 1 - | 1
               d / - 1 | d
               - - - - | q

```

Simulation model (verilog)

Listing 9.151: simcells.v

```

1299 module \$_DFFE_PP1P_ (D, C, R, E, Q);
1300     input D, C, R, E;
1301     output reg Q;
1302     always @(posedge C or posedge R) begin
1303         if (R == 1)
1304             Q <= 1;
1305         else if (E == 1)
1306             Q <= D;
1307     end
1308 endmodule

```

yosys> help \$_DFFE_PP_

A positive edge D-type flip-flop with positive polarity enable.

```

Truth table:   D C E | Q
               +-----+
               d / 1 | d
               - - - | q

```

Simulation model (verilog)

Listing 9.152: simcells.v

```

710 module \$_DFFE_PP_ (D, C, E, Q);
711     input D, C, E;
712     output reg Q;
713     always @(posedge C) begin
714         if (E) Q <= D;
715     end
716 endmodule

```

yosys> help \$_DFFSRE_NNNN_

A negative edge D-type flip-flop with negative polarity set, negative polarity reset and negative polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	0	-	-		0
	-	0	-	-	-		1
	\	-	-	0	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.153: simcells.v

```

1845 module \$_DFFSRE_NNNN_ (C, S, R, E, D, Q);
1846     input C, S, R, E, D;
1847     output reg Q;
1848     always @(negedge C, negedge S, negedge R) begin
1849         if (R == 0)
1850             Q <= 0;
1851         else if (S == 0)
1852             Q <= 1;
1853         else if (E == 0)
1854             Q <= D;
1855     end
1856 endmodule

```

yosys> help \$_DFFSRE_NNNP_

A negative edge D-type flip-flop with negative polarity set, negative polarity reset and positive polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	0	-	-		0
	-	0	-	-	-		1
	\	-	-	1	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.154: simcells.v

```

1873 module \$_DFFSRE_NNPN_ (C, S, R, E, D, Q);
1874     input C, S, R, E, D;
1875     output reg Q;
1876     always @(negedge C, negedge S, negedge R) begin
1877         if (R == 0)
1878             Q <= 0;
1879         else if (S == 0)
1880             Q <= 1;
1881         else if (E == 1)
1882             Q <= D;
1883     end
1884 endmodule

```

yosys> help \$_DFFSRE_NNPN_

A negative edge D-type flip-flop with negative polarity set, positive polarity reset and negative polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	1	-	-		0
	-	0	-	-	-		1
	\	-	-	0	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.155: simcells.v

```

1901 module \$_DFFSRE_NNPN_ (C, S, R, E, D, Q);
1902     input C, S, R, E, D;
1903     output reg Q;
1904     always @(negedge C, negedge S, posedge R) begin
1905         if (R == 1)
1906             Q <= 0;
1907         else if (S == 0)
1908             Q <= 1;
1909         else if (E == 0)
1910             Q <= D;
1911     end
1912 endmodule

```

yosys> help \$_DFFSRE_NNPP_

A negative edge D-type flip-flop with negative polarity set, positive polarity reset and positive polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	1	-	-		0
	-	0	-	-	-		1
	\	-	-	1	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.156: simcells.v

```

1929 module \$_DFFSRE_NNPP_ (C, S, R, E, D, Q);
1930     input C, S, R, E, D;
1931     output reg Q;
1932     always @(negedge C, negedge S, posedge R) begin
1933         if (R == 1)
1934             Q <= 0;
1935         else if (S == 0)
1936             Q <= 1;
1937         else if (E == 1)
1938             Q <= D;
1939     end
1940 endmodule

```

yosys> help \$_DFFSRE_NPNN_

A negative edge D-type flip-flop with positive polarity set, negative polarity reset and negative polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	0	-	-		0
	-	1	-	-	-		1
	\	-	-	0	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.157: simcells.v

```

1957 module \$_DFFSRE_NPNN_ (C, S, R, E, D, Q);
1958     input C, S, R, E, D;
1959     output reg Q;
1960     always @(negedge C, posedge S, negedge R) begin
1961         if (R == 0)
1962             Q <= 0;
1963         else if (S == 1)
1964             Q <= 1;
1965         else if (E == 0)
1966             Q <= D;
1967     end
1968 endmodule

```

yosys> help \$_DFFSRE_NPNP_

A negative edge D-type flip-flop with positive polarity set, negative polarity reset and positive polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	0	-	-		0
	-	1	-	-	-		1

(continues on next page)

(continued from previous page)

```

\ - - 1 d | d
- - - - - | q

```

Simulation model (verilog)

Listing 9.158: simcells.v

```

1985 module \$_DFFSRE_NPNP_ (C, S, R, E, D, Q);
1986     input C, S, R, E, D;
1987     output reg Q;
1988     always @(negedge C, posedge S, negedge R) begin
1989         if (R == 0)
1990             Q <= 0;
1991         else if (S == 1)
1992             Q <= 1;
1993         else if (E == 1)
1994             Q <= D;
1995     end
1996 endmodule

```

yosys> help \$_DFFSRE_NPPN_

A negative edge D-type flip-flop with positive polarity set, positive polarity reset and negative polarity clock enable.

```

Truth table:   C S R E D | Q
               +-----+
               - - 1 - - | 0
               - 1 - - - | 1
               \ - - 0 d | d
               - - - - - | q

```

Simulation model (verilog)

Listing 9.159: simcells.v

```

2013 module \$_DFFSRE_NPPN_ (C, S, R, E, D, Q);
2014     input C, S, R, E, D;
2015     output reg Q;
2016     always @(negedge C, posedge S, posedge R) begin
2017         if (R == 1)
2018             Q <= 0;
2019         else if (S == 1)
2020             Q <= 1;
2021         else if (E == 0)
2022             Q <= D;
2023     end
2024 endmodule

```

yosys> help \$_DFFSRE_NPPP_

A negative edge D-type flip-flop with positive polarity set, positive polarity reset and positive polarity clock enable.

Truth table:	C	S	R	E	D		Q
	---	---	---	---	---	+	---
	-	-	1	-	-		0
	-	1	-	-	-		1
	\	-	-	1	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.160: simcells.v

```

2041 module \$_DFFSRE_NPPP_ (C, S, R, E, D, Q);
2042     input C, S, R, E, D;
2043     output reg Q;
2044     always @(negedge C, posedge S, posedge R) begin
2045         if (R == 1)
2046             Q <= 0;
2047         else if (S == 1)
2048             Q <= 1;
2049         else if (E == 1)
2050             Q <= D;
2051     end
2052 endmodule

```

yosys> help \$_DFFSRE_PNNN_

A positive edge D-type flip-flop with negative polarity set, negative polarity reset and negative polarity clock enable.

Truth table:	C	S	R	E	D		Q
	---	---	---	---	---	+	---
	-	-	0	-	-		0
	-	0	-	-	-		1
	/	-	-	0	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.161: simcells.v

```

2069 module \$_DFFSRE_PNNN_ (C, S, R, E, D, Q);
2070     input C, S, R, E, D;
2071     output reg Q;
2072     always @(posedge C, negedge S, negedge R) begin
2073         if (R == 0)
2074             Q <= 0;
2075         else if (S == 0)
2076             Q <= 1;
2077         else if (E == 0)
2078             Q <= D;
2079     end
2080 endmodule

```

yosys> help \$_DFFSRE_PNNP_

A positive edge D-type flip-flop with negative polarity set, negative polarity reset and positive polarity

clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	0	-	-		0
	-	0	-	-	-		1
	/	-	-	1	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.162: simcells.v

```

2097 module \$_DFFSRE_PNNP_ (C, S, R, E, D, Q);
2098     input C, S, R, E, D;
2099     output reg Q;
2100     always @(posedge C, negedge S, negedge R) begin
2101         if (R == 0)
2102             Q <= 0;
2103         else if (S == 0)
2104             Q <= 1;
2105         else if (E == 1)
2106             Q <= D;
2107     end
2108 endmodule

```

yosys> help \$_DFFSRE_PNNP_

A positive edge D-type flip-flop with negative polarity set, positive polarity reset and negative polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	1	-	-		0
	-	0	-	-	-		1
	/	-	-	0	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.163: simcells.v

```

2125 module \$_DFFSRE_PNNP_ (C, S, R, E, D, Q);
2126     input C, S, R, E, D;
2127     output reg Q;
2128     always @(posedge C, negedge S, posedge R) begin
2129         if (R == 1)
2130             Q <= 0;
2131         else if (S == 0)
2132             Q <= 1;
2133         else if (E == 0)
2134             Q <= D;
2135     end
2136 endmodule

```

yosys> help \$_DFFSRE_PNPP_

A positive edge D-type flip-flop with negative polarity set, positive polarity reset and positive polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	1	-	-		0
	-	0	-	-	-		1
	/	-	-	1	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.164: simcells.v

```

2153 module \$_DFFSRE_PNPP_ (C, S, R, E, D, Q);
2154     input C, S, R, E, D;
2155     output reg Q;
2156     always @(posedge C, negedge S, posedge R) begin
2157         if (R == 1)
2158             Q <= 0;
2159         else if (S == 0)
2160             Q <= 1;
2161         else if (E == 1)
2162             Q <= D;
2163     end
2164 endmodule

```

yosys> help \$_DFFSRE_PPNN_

A positive edge D-type flip-flop with positive polarity set, negative polarity reset and negative polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	0	-	-		0
	-	1	-	-	-		1
	/	-	-	0	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.165: simcells.v

```

2181 module \$_DFFSRE_PPNN_ (C, S, R, E, D, Q);
2182     input C, S, R, E, D;
2183     output reg Q;
2184     always @(posedge C, posedge S, negedge R) begin
2185         if (R == 0)
2186             Q <= 0;
2187         else if (S == 1)
2188             Q <= 1;
2189         else if (E == 0)
2190             Q <= D;

```

(continues on next page)

(continued from previous page)

```

2191     end
2192 endmodule

```

```
yosys> help $_DFFSRE_PPNP_
```

A positive edge D-type flip-flop with positive polarity set, negative polarity reset and positive polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	0	-	-		0
	-	1	-	-	-		1
	/	-	-	1	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.166: simcells.v

```

2209 module \$_DFFSRE_PPNP_ (C, S, R, E, D, Q);
2210     input C, S, R, E, D;
2211     output reg Q;
2212     always @(posedge C, posedge S, negedge R) begin
2213         if (R == 0)
2214             Q <= 0;
2215         else if (S == 1)
2216             Q <= 1;
2217         else if (E == 1)
2218             Q <= D;
2219     end
2220 endmodule

```

```
yosys> help $_DFFSRE_PPPN_
```

A positive edge D-type flip-flop with positive polarity set, positive polarity reset and negative polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	1	-	-		0
	-	1	-	-	-		1
	/	-	-	0	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.167: simcells.v

```

2237 module \$_DFFSRE_PPPN_ (C, S, R, E, D, Q);
2238     input C, S, R, E, D;
2239     output reg Q;
2240     always @(posedge C, posedge S, posedge R) begin
2241         if (R == 1)
2242             Q <= 0;

```

(continues on next page)

(continued from previous page)

```

2243     else if (S == 1)
2244         Q <= 1;
2245     else if (E == 0)
2246         Q <= D;
2247     end
2248 endmodule

```

yosys> help \$_DFFSRE_PPPP_

A positive edge D-type flip-flop with positive polarity set, positive polarity reset and positive polarity clock enable.

Truth table:	C	S	R	E	D		Q
	-	-	1	-	-		0
	-	1	-	-	-		1
	/	-	-	1	d		d
	-	-	-	-	-		q

Simulation model (verilog)

Listing 9.168: simcells.v

```

2265 module \$_DFFSRE_PPPP_ (C, S, R, E, D, Q);
2266     input C, S, R, E, D;
2267     output reg Q;
2268     always @(posedge C, posedge S, posedge R) begin
2269         if (R == 1)
2270             Q <= 0;
2271         else if (S == 1)
2272             Q <= 1;
2273         else if (E == 1)
2274             Q <= D;
2275     end
2276 endmodule

```

yosys> help \$_DFFSR_NNN_

A negative edge D-type flip-flop with negative polarity set and negative polarity reset.

Truth table:	C	S	R	D		Q
	-	-	0	-		0
	-	0	-	-		1
	\	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.169: simcells.v

```

1621 module \$_DFFSR_NNN_ (C, S, R, D, Q);
1622     input C, S, R, D;
1623     output reg Q;

```

(continues on next page)

(continued from previous page)

```

1624     always @(negedge C, negedge S, negedge R) begin
1625         if (R == 0)
1626             Q <= 0;
1627         else if (S == 0)
1628             Q <= 1;
1629         else
1630             Q <= D;
1631     end
1632 endmodule

```

yosys> help \$_DFFSR_NNP_

A negative edge D-type flip-flop with negative polarity set and positive polarity reset.

Truth table:	C	S	R	D		Q
	-	-	1	-		0
	-	0	-	-		1
	\	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.170: simcells.v

```

1649 module \$_DFFSR_NNP_ (C, S, R, D, Q);
1650     input C, S, R, D;
1651     output reg Q;
1652     always @(negedge C, negedge S, posedge R) begin
1653         if (R == 1)
1654             Q <= 0;
1655         else if (S == 0)
1656             Q <= 1;
1657         else
1658             Q <= D;
1659     end
1660 endmodule

```

yosys> help \$_DFFSR_NPN_

A negative edge D-type flip-flop with positive polarity set and negative polarity reset.

Truth table:	C	S	R	D		Q
	-	-	0	-		0
	-	1	-	-		1
	\	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.171: simcells.v

```

1677 module \$_DFFSR_NPN_ (C, S, R, D, Q);
1678     input C, S, R, D;
1679     output reg Q;
1680     always @(negedge C, posedge S, negedge R) begin
1681         if (R == 0)
1682             Q <= 0;
1683         else if (S == 1)
1684             Q <= 1;
1685         else
1686             Q <= D;
1687     end
1688 endmodule

```

yosys> help \$_DFFSR_NPP_

A negative edge D-type flip-flop with positive polarity set and positive polarity reset.

Truth table:	C	S	R	D		Q
	-	-	1	-		0
	-	1	-	-		1
	\	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.172: simcells.v

```

1705 module \$_DFFSR_NPP_ (C, S, R, D, Q);
1706     input C, S, R, D;
1707     output reg Q;
1708     always @(negedge C, posedge S, posedge R) begin
1709         if (R == 1)
1710             Q <= 0;
1711         else if (S == 1)
1712             Q <= 1;
1713         else
1714             Q <= D;
1715     end
1716 endmodule

```

yosys> help \$_DFFSR_PNN_

A positive edge D-type flip-flop with negative polarity set and negative polarity reset.

Truth table:	C	S	R	D		Q
	-	-	0	-		0
	-	0	-	-		1
	/	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.173: simcells.v

```

1733 module \$_DFFSR_PNN_ (C, S, R, D, Q);
1734     input C, S, R, D;
1735     output reg Q;
1736     always @(posedge C, negedge S, negedge R) begin
1737         if (R == 0)
1738             Q <= 0;
1739         else if (S == 0)
1740             Q <= 1;
1741         else
1742             Q <= D;
1743     end
1744 endmodule

```

yosys> help \$_DFFSR_PNP_

A positive edge D-type flip-flop with negative polarity set and positive polarity reset.

Truth table:	C	S	R	D		Q
	-	-	1	-		0
	-	0	-	-		1
	/	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.174: simcells.v

```

1761 module \$_DFFSR_PNP_ (C, S, R, D, Q);
1762     input C, S, R, D;
1763     output reg Q;
1764     always @(posedge C, negedge S, posedge R) begin
1765         if (R == 1)
1766             Q <= 0;
1767         else if (S == 0)
1768             Q <= 1;
1769         else
1770             Q <= D;
1771     end
1772 endmodule

```

yosys> help \$_DFFSR_PPN_

A positive edge D-type flip-flop with positive polarity set and negative polarity reset.

Truth table:	C	S	R	D		Q
	-	-	0	-		0
	-	1	-	-		1
	/	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.175: simcells.v

```

1789 module \$_DFFSR_PPN_ (C, S, R, D, Q);
1790     input C, S, R, D;
1791     output reg Q;
1792     always @(posedge C, posedge S, negedge R) begin
1793         if (R == 0)
1794             Q <= 0;
1795         else if (S == 1)
1796             Q <= 1;
1797         else
1798             Q <= D;
1799     end
1800 endmodule

```

yosys> help \$_DFFSR_PPP_

A positive edge D-type flip-flop with positive polarity set and positive polarity reset.

Truth table:	C	S	R	D		Q
	-	-	1	-		0
	-	1	-	-		1
	/	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.176: simcells.v

```

1817 module \$_DFFSR_PPP_ (C, S, R, D, Q);
1818     input C, S, R, D;
1819     output reg Q;
1820     always @(posedge C, posedge S, posedge R) begin
1821         if (R == 1)
1822             Q <= 0;
1823         else if (S == 1)
1824             Q <= 1;
1825         else
1826             Q <= D;
1827     end
1828 endmodule

```

yosys> help \$_DFF_NN0_

A negative edge D-type flip-flop with negative polarity reset.

Truth table:	D	C	R		Q
	-	-	0		0
	d	\	-		d
	-	-	-		q

Simulation model (verilog)

Listing 9.177: simcells.v

```

731 module \$_DFF_NN0_ (D, C, R, Q);
732     input D, C, R;
733     output reg Q;
734     always @(negedge C or negedge R) begin
735         if (R == 0)
736             Q <= 0;
737         else
738             Q <= D;
739     end
740 endmodule

```

yosys> help \$_DFF_NN1_

A negative edge D-type flip-flop with negative polarity set.

Truth table:			
D	C	R	Q
-	-	0	1
d	\	-	d
-	-	-	q

Simulation model (verilog)

Listing 9.178: simcells.v

```

755 module \$_DFF_NN1_ (D, C, R, Q);
756     input D, C, R;
757     output reg Q;
758     always @(negedge C or negedge R) begin
759         if (R == 0)
760             Q <= 1;
761         else
762             Q <= D;
763     end
764 endmodule

```

yosys> help \$_DFF_NP0_

A negative edge D-type flip-flop with positive polarity reset.

Truth table:			
D	C	R	Q
-	-	1	0
d	\	-	d
-	-	-	q

Simulation model (verilog)

Listing 9.179: simcells.v

```

779 module \$_DFF_NP0_ (D, C, R, Q);
780     input D, C, R;
781     output reg Q;

```

(continues on next page)

(continued from previous page)

```

782     always @(negedge C or posedge R) begin
783         if (R == 1)
784             Q <= 0;
785         else
786             Q <= D;
787     end
788 endmodule

```

yosys> help \$_DFF_NP1_

A negative edge D-type flip-flop with positive polarity set.

```

Truth table:   D C R | Q
               -----+---
               - - 1 | 1
               d \ - | d
               - - - | q

```

Simulation model (verilog)

Listing 9.180: simcells.v

```

803 module \$_DFF_NP1_ (D, C, R, Q);
804     input D, C, R;
805     output reg Q;
806     always @(negedge C or posedge R) begin
807         if (R == 1)
808             Q <= 1;
809         else
810             Q <= D;
811     end
812 endmodule

```

yosys> help \$_DFF_N_

A negative edge D-type flip-flop.

```

Truth table:   D C | Q
               -----+---
               d \ | d
               - - | q

```

Simulation model (verilog)

Listing 9.181: simcells.v

```

610 module \$_DFF_N_ (D, C, Q);
611     input D, C;
612     output reg Q;
613     always @(negedge C) begin
614         Q <= D;
615     end
616 endmodule

```


yosys> help \$_DFF_PN0_

A positive edge D-type flip-flop with negative polarity reset.

Truth table:	D	C	R	Q
	-	-	0	0
	d	/	-	d
	-	-	-	q

Simulation model (verilog)

Listing 9.182: simcells.v

```

827 module \$_DFF_PN0_ (D, C, R, Q);
828     input D, C, R;
829     output reg Q;
830     always @(posedge C or negedge R) begin
831         if (R == 0)
832             Q <= 0;
833         else
834             Q <= D;
835     end
836 endmodule

```

yosys> help \$_DFF_PN1_

A positive edge D-type flip-flop with negative polarity set.

Truth table:	D	C	R	Q
	-	-	0	1
	d	/	-	d
	-	-	-	q

Simulation model (verilog)

Listing 9.183: simcells.v

```

851 module \$_DFF_PN1_ (D, C, R, Q);
852     input D, C, R;
853     output reg Q;
854     always @(posedge C or negedge R) begin
855         if (R == 0)
856             Q <= 1;
857         else
858             Q <= D;
859     end
860 endmodule

```

yosys> help \$_DFF_PP0_

A positive edge D-type flip-flop with positive polarity reset.

Truth table:	D	C	R	Q
	-	-	0	0
	d	/	-	d
	-	-	-	q

(continues on next page)

(continued from previous page)

```

- - 1 | 0
d / - | d
- - - | q

```

Simulation model (verilog)

Listing 9.184: simcells.v

```

875 module \$_DFF_PP0_ (D, C, R, Q);
876     input D, C, R;
877     output reg Q;
878     always @(posedge C or posedge R) begin
879         if (R == 1)
880             Q <= 0;
881         else
882             Q <= D;
883     end
884 endmodule

```

yosys> help \$_DFF_PP1_

A positive edge D-type flip-flop with positive polarity set.

```

Truth table:   D C R | Q
               -----+---
               - - 1 | 1
               d / - | d
               - - - | q

```

Simulation model (verilog)

Listing 9.185: simcells.v

```

899 module \$_DFF_PP1_ (D, C, R, Q);
900     input D, C, R;
901     output reg Q;
902     always @(posedge C or posedge R) begin
903         if (R == 1)
904             Q <= 1;
905         else
906             Q <= D;
907     end
908 endmodule

```

yosys> help \$_DFF_P_

A positive edge D-type flip-flop.

```

Truth table:   D C | Q
               -----+---
               d / | d
               - - | q

```

Simulation model (verilog)

Listing 9.186: simcells.v

```

630 module \$_DFF_P_ (D, C, Q);
631     input D, C;
632     output reg Q;
633     always @(posedge C) begin
634         Q <= D;
635     end
636 endmodule

```

yosys> help \$_FF_

A D-type flip-flop that is clocked from the implicit global clock. (This cell type is usually only used in netlists for formal verification.)

Simulation model (verilog)

Listing 9.187: simcells.v

```

589 module \$_FF_ (D, Q);
590     input D;
591     output reg Q;
592     always @($global_clock) begin
593         Q <= D;
594     end
595 endmodule

```

yosys> help \$_SDFFCE_NNON_

A negative edge D-type flip-flop with negative polarity synchronous reset and negative polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E	Q
	-	\	0	0	0
	d	\	-	0	d
	-	-	-	-	q

Simulation model (verilog)

Listing 9.188: simcells.v

```

2884 module \$_SDFFCE_NNON_ (D, C, R, E, Q);
2885     input D, C, R, E;
2886     output reg Q;
2887     always @(negedge C) begin
2888         if (E == 0) begin
2889             if (R == 0)
2890                 Q <= 0;
2891             else
2892                 Q <= D;
2893         end
2894     end
2895 endmodule

```

yosys> help \$_SDFFCE_NNOP_

A negative edge D-type flip-flop with negative polarity synchronous reset and positive polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-----	+	---			
	-	\	0	1		0
	d	\	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.189: simcells.v

```

2911 module \$_SDFFCE_NNOP_ (D, C, R, E, Q);
2912     input D, C, R, E;
2913     output reg Q;
2914     always @(negedge C) begin
2915         if (E == 1) begin
2916             if (R == 0)
2917                 Q <= 0;
2918             else
2919                 Q <= D;
2920         end
2921     end
2922 endmodule

```

yosys> help \\$_SDFFCE_NN1N_

A negative edge D-type flip-flop with negative polarity synchronous set and negative polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-----	+	---			
	-	\	0	0		1
	d	\	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.190: simcells.v

```

2938 module \$_SDFFCE_NN1N_ (D, C, R, E, Q);
2939     input D, C, R, E;
2940     output reg Q;
2941     always @(negedge C) begin
2942         if (E == 0) begin
2943             if (R == 0)
2944                 Q <= 1;
2945             else
2946                 Q <= D;
2947         end
2948     end
2949 endmodule

```

yosys> help \$_SDFFCE_NN1P_

A negative edge D-type flip-flop with negative polarity synchronous set and positive polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-	\	0	1		1
	d	\	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.191: simcells.v

```

2965 module \$_SDFFCE_NN1P_ (D, C, R, E, Q);
2966     input D, C, R, E;
2967     output reg Q;
2968     always @(negedge C) begin
2969         if (E == 1) begin
2970             if (R == 0)
2971                 Q <= 1;
2972             else
2973                 Q <= D;
2974         end
2975     end
2976 endmodule

```

yosys> help \$_SDFFCE_NPON_

A negative edge D-type flip-flop with positive polarity synchronous reset and negative polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-	\	1	0		0
	d	\	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.192: simcells.v

```

2992 module \$_SDFFCE_NPON_ (D, C, R, E, Q);
2993     input D, C, R, E;
2994     output reg Q;
2995     always @(negedge C) begin
2996         if (E == 0) begin
2997             if (R == 1)
2998                 Q <= 0;
2999             else
3000                 Q <= D;
3001         end
3002     end
3003 endmodule

```

yosys> help \$_SDFFCE_NPOP_

A negative edge D-type flip-flop with positive polarity synchronous reset and positive polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-----	+	---			
	-	\	1	1		0
	d	\	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.193: simcells.v

```
3019 module \$_SDFFCE_NPOP_ (D, C, R, E, Q);
3020     input D, C, R, E;
3021     output reg Q;
3022     always @(negedge C) begin
3023         if (E == 1) begin
3024             if (R == 1)
3025                 Q <= 0;
3026             else
3027                 Q <= D;
3028         end
3029     end
3030 endmodule
```

yosys> help \$_SDFFCE_NP1N_

A negative edge D-type flip-flop with positive polarity synchronous set and negative polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-----	+	---			
	-	\	1	0		1
	d	\	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.194: simcells.v

```
3046 module \$_SDFFCE_NP1N_ (D, C, R, E, Q);
3047     input D, C, R, E;
3048     output reg Q;
3049     always @(negedge C) begin
3050         if (E == 0) begin
3051             if (R == 1)
3052                 Q <= 1;
3053             else
3054                 Q <= D;
3055         end
3056     end
3057 endmodule
```

yosys> help \$_SDFFCE_NP1P_

A negative edge D-type flip-flop with positive polarity synchronous set and positive polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-	\	1	1		1
	d	\	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.195: simcells.v

```

3073 module \$_SDFFCE_NP1P_ (D, C, R, E, Q);
3074     input D, C, R, E;
3075     output reg Q;
3076     always @(negedge C) begin
3077         if (E == 1) begin
3078             if (R == 1)
3079                 Q <= 1;
3080             else
3081                 Q <= D;
3082         end
3083     end
3084 endmodule

```

yosys> help \$_SDFFCE_PNON_

A positive edge D-type flip-flop with negative polarity synchronous reset and negative polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-	/	0	0		0
	d	/	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.196: simcells.v

```

3100 module \$_SDFFCE_PNON_ (D, C, R, E, Q);
3101     input D, C, R, E;
3102     output reg Q;
3103     always @(posedge C) begin
3104         if (E == 0) begin
3105             if (R == 0)
3106                 Q <= 0;
3107             else
3108                 Q <= D;
3109         end
3110     end
3111 endmodule

```

yosys> help \$_SDFFCE_PNOP_

A positive edge D-type flip-flop with negative polarity synchronous reset and positive polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-	/	0	1		0
	d	/	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.197: simcells.v

```

3127 module \$_SDFFCE_PNOP_ (D, C, R, E, Q);
3128     input D, C, R, E;
3129     output reg Q;
3130     always @(posedge C) begin
3131         if (E == 1) begin
3132             if (R == 0)
3133                 Q <= 0;
3134             else
3135                 Q <= D;
3136         end
3137     end
3138 endmodule

```

yosys> help \$_SDFFCE_PN1N_

A positive edge D-type flip-flop with negative polarity synchronous set and negative polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-	/	0	0		1
	d	/	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.198: simcells.v

```

3154 module \$_SDFFCE_PN1N_ (D, C, R, E, Q);
3155     input D, C, R, E;
3156     output reg Q;
3157     always @(posedge C) begin
3158         if (E == 0) begin
3159             if (R == 0)
3160                 Q <= 1;
3161             else
3162                 Q <= D;
3163         end
3164     end
3165 endmodule

```


yosys> help \$_SDFFCE_PN1P_

A positive edge D-type flip-flop with negative polarity synchronous set and positive polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-	/	0	1		1
	d	/	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.199: simcells.v

```

3181 module \$_SDFFCE_PN1P_ (D, C, R, E, Q);
3182     input D, C, R, E;
3183     output reg Q;
3184     always @(posedge C) begin
3185         if (E == 1) begin
3186             if (R == 0)
3187                 Q <= 1;
3188             else
3189                 Q <= D;
3190         end
3191     end
3192 endmodule

```

yosys> help \$_SDFFCE_PPON_

A positive edge D-type flip-flop with positive polarity synchronous reset and negative polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-	/	1	0		0
	d	/	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.200: simcells.v

```

3208 module \$_SDFFCE_PPON_ (D, C, R, E, Q);
3209     input D, C, R, E;
3210     output reg Q;
3211     always @(posedge C) begin
3212         if (E == 0) begin
3213             if (R == 1)
3214                 Q <= 0;
3215             else
3216                 Q <= D;
3217         end
3218     end
3219 endmodule

```

yosys> help \$_SDFFCE_PPOP_

A positive edge D-type flip-flop with positive polarity synchronous reset and positive polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-	/	1	1		0
	d	/	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.201: simcells.v

```
3235 module \$_SDFFCE_PPOP_ (D, C, R, E, Q);
3236     input D, C, R, E;
3237     output reg Q;
3238     always @(posedge C) begin
3239         if (E == 1) begin
3240             if (R == 1)
3241                 Q <= 0;
3242             else
3243                 Q <= D;
3244         end
3245     end
3246 endmodule
```

yosys> help \$_SDFFCE_PP1N_

A positive edge D-type flip-flop with positive polarity synchronous set and negative polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-	/	1	0		1
	d	/	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.202: simcells.v

```
3262 module \$_SDFFCE_PP1N_ (D, C, R, E, Q);
3263     input D, C, R, E;
3264     output reg Q;
3265     always @(posedge C) begin
3266         if (E == 0) begin
3267             if (R == 1)
3268                 Q <= 1;
3269             else
3270                 Q <= D;
3271         end
3272     end
3273 endmodule
```

yosys> help \$_SDFFCE_PP1P_

A positive edge D-type flip-flop with positive polarity synchronous set and positive polarity clock enable (with clock enable having priority).

Truth table:	D	C	R	E		Q
	-	/	1	1		1
	d	/	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.203: simcells.v

```

3289 module \$_SDFFCE_PP1P_ (D, C, R, E, Q);
3290     input D, C, R, E;
3291     output reg Q;
3292     always @(posedge C) begin
3293         if (E == 1) begin
3294             if (R == 1)
3295                 Q <= 1;
3296             else
3297                 Q <= D;
3298         end
3299     end
3300 endmodule

```

yosys> help \$_SDFFE_NNON_

A negative edge D-type flip-flop with negative polarity synchronous reset and negative polarity clock enable (with reset having priority).

Truth table:	D	C	R	E		Q
	-	\	0	-		0
	d	\	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.204: simcells.v

```

2484 module \$_SDFFE_NNON_ (D, C, R, E, Q);
2485     input D, C, R, E;
2486     output reg Q;
2487     always @(negedge C) begin
2488         if (R == 0)
2489             Q <= 0;
2490         else if (E == 0)
2491             Q <= D;
2492     end
2493 endmodule

```

yosys> help \$_SDFFE_NNOP_

A negative edge D-type flip-flop with negative polarity synchronous reset and positive polarity clock

enable (with reset having priority).

Truth table:	D	C	R	E		Q
	-----	+	---			
	-	\	0	-		0
	d	\	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.205: simcells.v

```

2509 module \$_SDFFE_NNOP_ (D, C, R, E, Q);
2510     input D, C, R, E;
2511     output reg Q;
2512     always @(negedge C) begin
2513         if (R == 0)
2514             Q <= 0;
2515         else if (E == 1)
2516             Q <= D;
2517     end
2518 endmodule

```

yosys> help \$_SDFFE_NN1N_

A negative edge D-type flip-flop with negative polarity synchronous set and negative polarity clock enable (with set having priority).

Truth table:	D	C	R	E		Q
	-----	+	---			
	-	\	0	-		1
	d	\	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.206: simcells.v

```

2534 module \$_SDFFE_NN1N_ (D, C, R, E, Q);
2535     input D, C, R, E;
2536     output reg Q;
2537     always @(negedge C) begin
2538         if (R == 0)
2539             Q <= 1;
2540         else if (E == 0)
2541             Q <= D;
2542     end
2543 endmodule

```

yosys> help \$_SDFFE_NN1P_

A negative edge D-type flip-flop with negative polarity synchronous set and positive polarity clock enable (with set having priority).

Truth table:	D	C	R	E		Q
	-----	+	---			

(continues on next page)

(continued from previous page)

-	\	0	-		1
d	\	-	1		d
-	-	-	-		q

Simulation model (verilog)

Listing 9.207: simcells.v

```

2559 module \$_SDFFE_NN1P_ (D, C, R, E, Q);
2560     input D, C, R, E;
2561     output reg Q;
2562     always @(negedge C) begin
2563         if (R == 0)
2564             Q <= 1;
2565         else if (E == 1)
2566             Q <= D;
2567     end
2568 endmodule

```

yosys> help \$_SDFFE_NPON_

A negative edge D-type flip-flop with positive polarity synchronous reset and negative polarity clock enable (with reset having priority).

Truth table:	D	C	R	E		Q
	-	-	-	-		-
	-	\	1	-		0
	d	\	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.208: simcells.v

```

2584 module \$_SDFFE_NPON_ (D, C, R, E, Q);
2585     input D, C, R, E;
2586     output reg Q;
2587     always @(negedge C) begin
2588         if (R == 1)
2589             Q <= 0;
2590         else if (E == 0)
2591             Q <= D;
2592     end
2593 endmodule

```

yosys> help \$_SDFFE_NPOP_

A negative edge D-type flip-flop with positive polarity synchronous reset and positive polarity clock enable (with reset having priority).

Truth table:	D	C	R	E		Q
	-	-	-	-		-
	-	\	1	-		0
	d	\	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.209: simcells.v

```

2609 module \$_SDFFE_NPOP_ (D, C, R, E, Q);
2610     input D, C, R, E;
2611     output reg Q;
2612     always @(negedge C) begin
2613         if (R == 1)
2614             Q <= 0;
2615         else if (E == 1)
2616             Q <= D;
2617     end
2618 endmodule

```

yosys> help \$_SDFFE_NP1N_

A negative edge D-type flip-flop with positive polarity synchronous set and negative polarity clock enable (with set having priority).

Truth table:	D	C	R	E		Q
	-	-	-	-	+	-
	-	\	1	-		1
	d	\	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.210: simcells.v

```

2634 module \$_SDFFE_NP1N_ (D, C, R, E, Q);
2635     input D, C, R, E;
2636     output reg Q;
2637     always @(negedge C) begin
2638         if (R == 1)
2639             Q <= 1;
2640         else if (E == 0)
2641             Q <= D;
2642     end
2643 endmodule

```

yosys> help \$_SDFFE_NP1P_

A negative edge D-type flip-flop with positive polarity synchronous set and positive polarity clock enable (with set having priority).

Truth table:	D	C	R	E		Q
	-	-	-	-	+	-
	-	\	1	-		1
	d	\	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.211: simcells.v

```

2659 module \$_SDFFE_NP1P_ (D, C, R, E, Q);
2660     input D, C, R, E;
2661     output reg Q;
2662     always @(negedge C) begin
2663         if (R == 1)
2664             Q <= 1;
2665         else if (E == 1)
2666             Q <= D;
2667     end
2668 endmodule

```

yosys> help \$_SDFFE_PNON_

A positive edge D-type flip-flop with negative polarity synchronous reset and negative polarity clock enable (with reset having priority).

Truth table:	D	C	R	E		Q
	-	/	0	-		0
	d	/	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.212: simcells.v

```

2684 module \$_SDFFE_PNON_ (D, C, R, E, Q);
2685     input D, C, R, E;
2686     output reg Q;
2687     always @(posedge C) begin
2688         if (R == 0)
2689             Q <= 0;
2690         else if (E == 0)
2691             Q <= D;
2692     end
2693 endmodule

```

yosys> help \$_SDFFE_PNOP_

A positive edge D-type flip-flop with negative polarity synchronous reset and positive polarity clock enable (with reset having priority).

Truth table:	D	C	R	E		Q
	-	/	0	-		0
	d	/	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.213: simcells.v

```

2709 module \$_SDFFE_PNOP_ (D, C, R, E, Q);
2710     input D, C, R, E;
2711     output reg Q;
2712     always @(posedge C) begin
2713         if (R == 0)
2714             Q <= 0;
2715         else if (E == 1)
2716             Q <= D;
2717     end
2718 endmodule

```

yosys> help \$_SDFFE_PN1N_

A positive edge D-type flip-flop with negative polarity synchronous set and negative polarity clock enable (with set having priority).

Truth table:	D	C	R	E		Q
	-	-	-	-		q
	-	/	0	-		1
	d	/	-	0		d

Simulation model (verilog)

Listing 9.214: simcells.v

```

2734 module \$_SDFFE_PN1N_ (D, C, R, E, Q);
2735     input D, C, R, E;
2736     output reg Q;
2737     always @(posedge C) begin
2738         if (R == 0)
2739             Q <= 1;
2740         else if (E == 0)
2741             Q <= D;
2742     end
2743 endmodule

```

yosys> help \$_SDFFE_PN1P_

A positive edge D-type flip-flop with negative polarity synchronous set and positive polarity clock enable (with set having priority).

Truth table:	D	C	R	E		Q
	-	-	-	-		q
	-	/	0	-		1
	d	/	-	1		d

Simulation model (verilog)

Listing 9.215: simcells.v

```

2759 module \$_SDFFE_PN1P_ (D, C, R, E, Q);
2760     input D, C, R, E;
2761     output reg Q;
2762     always @(posedge C) begin
2763         if (R == 0)
2764             Q <= 1;
2765         else if (E == 1)
2766             Q <= D;
2767     end
2768 endmodule

```

yosys> help \$_SDFFE_PPON_

A positive edge D-type flip-flop with positive polarity synchronous reset and negative polarity clock enable (with reset having priority).

Truth table:	D	C	R	E		Q
	-----+---					
	-	/	1	-		0
	d	/	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.216: simcells.v

```

2784 module \$_SDFFE_PPON_ (D, C, R, E, Q);
2785     input D, C, R, E;
2786     output reg Q;
2787     always @(posedge C) begin
2788         if (R == 1)
2789             Q <= 0;
2790         else if (E == 0)
2791             Q <= D;
2792     end
2793 endmodule

```

yosys> help \$_SDFFE_PPOP_

A positive edge D-type flip-flop with positive polarity synchronous reset and positive polarity clock enable (with reset having priority).

Truth table:	D	C	R	E		Q
	-----+---					
	-	/	1	-		0
	d	/	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.217: simcells.v

```

2809 module \$_SDFFE_PPOP_ (D, C, R, E, Q);
2810     input D, C, R, E;
2811     output reg Q;
2812     always @(posedge C) begin
2813         if (R == 1)
2814             Q <= 0;
2815         else if (E == 1)
2816             Q <= D;
2817     end
2818 endmodule

```

yosys> help \$_SDFFE_PP1N_

A positive edge D-type flip-flop with positive polarity synchronous set and negative polarity clock enable (with set having priority).

Truth table:	D	C	R	E		Q
	-	/	1	-		1
	d	/	-	0		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.218: simcells.v

```

2834 module \$_SDFFE_PP1N_ (D, C, R, E, Q);
2835     input D, C, R, E;
2836     output reg Q;
2837     always @(posedge C) begin
2838         if (R == 1)
2839             Q <= 1;
2840         else if (E == 0)
2841             Q <= D;
2842     end
2843 endmodule

```

yosys> help \$_SDFFE_PP1P_

A positive edge D-type flip-flop with positive polarity synchronous set and positive polarity clock enable (with set having priority).

Truth table:	D	C	R	E		Q
	-	/	1	-		1
	d	/	-	1		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.219: simcells.v

```

2859 module \$_SDFFE_PP1P_ (D, C, R, E, Q);
2860     input D, C, R, E;
2861     output reg Q;
2862     always @(posedge C) begin
2863         if (R == 1)
2864             Q <= 1;
2865         else if (E == 1)
2866             Q <= D;
2867     end
2868 endmodule

```

yosys> help \$_SDFF_NN0_

A negative edge D-type flip-flop with negative polarity synchronous reset.

```

Truth table:   D C R | Q
              -----+---
              - \ 0 | 0
              d \ - | d
              - - - | q

```

Simulation model (verilog)

Listing 9.220: simcells.v

```

2291 module \$_SDFF_NN0_ (D, C, R, Q);
2292     input D, C, R;
2293     output reg Q;
2294     always @(negedge C) begin
2295         if (R == 0)
2296             Q <= 0;
2297         else
2298             Q <= D;
2299     end
2300 endmodule

```

yosys> help \$_SDFF_NN1_

A negative edge D-type flip-flop with negative polarity synchronous set.

```

Truth table:   D C R | Q
              -----+---
              - \ 0 | 1
              d \ - | d
              - - - | q

```

Simulation model (verilog)

Listing 9.221: simcells.v

```

2315 module \$_SDFF_NN1_ (D, C, R, Q);
2316     input D, C, R;
2317     output reg Q;

```

(continues on next page)

(continued from previous page)

```

2318     always @(negedge C) begin
2319         if (R == 0)
2320             Q <= 1;
2321         else
2322             Q <= D;
2323     end
2324 endmodule

```

yosys> help \$_SDFF_NP0_

A negative edge D-type flip-flop with positive polarity synchronous reset.

```

Truth table:   D C R | Q
               -----+---
               - \ 1 | 0
               d \ - | d
               - - - | q

```

Simulation model (verilog)

Listing 9.222: simcells.v

```

2339 module \$_SDFF_NP0_ (D, C, R, Q);
2340     input D, C, R;
2341     output reg Q;
2342     always @(negedge C) begin
2343         if (R == 1)
2344             Q <= 0;
2345         else
2346             Q <= D;
2347     end
2348 endmodule

```

yosys> help \$_SDFF_NP1_

A negative edge D-type flip-flop with positive polarity synchronous set.

```

Truth table:   D C R | Q
               -----+---
               - \ 1 | 1
               d \ - | d
               - - - | q

```

Simulation model (verilog)

Listing 9.223: simcells.v

```

2363 module \$_SDFF_NP1_ (D, C, R, Q);
2364     input D, C, R;
2365     output reg Q;
2366     always @(negedge C) begin
2367         if (R == 1)
2368             Q <= 1;
2369         else

```

(continues on next page)

(continued from previous page)

```

2370         Q <= D;
2371     end
2372 endmodule

```

yosys> help \$_SDFF_PNO_

A positive edge D-type flip-flop with negative polarity synchronous reset.

```

Truth table:   D C R | Q
               -----+---
               - / 0 | 0
               d / - | d
               - - - | q

```

Simulation model (verilog)

Listing 9.224: simcells.v

```

2387 module \$_SDFF_PNO_ (D, C, R, Q);
2388     input D, C, R;
2389     output reg Q;
2390     always @(posedge C) begin
2391         if (R == 0)
2392             Q <= 0;
2393         else
2394             Q <= D;
2395     end
2396 endmodule

```

yosys> help \$_SDFF_PN1_

A positive edge D-type flip-flop with negative polarity synchronous set.

```

Truth table:   D C R | Q
               -----+---
               - / 0 | 1
               d / - | d
               - - - | q

```

Simulation model (verilog)

Listing 9.225: simcells.v

```

2411 module \$_SDFF_PN1_ (D, C, R, Q);
2412     input D, C, R;
2413     output reg Q;
2414     always @(posedge C) begin
2415         if (R == 0)
2416             Q <= 1;
2417         else
2418             Q <= D;
2419     end
2420 endmodule

```

yosys> help \$_SDFF_PP0_

A positive edge D-type flip-flop with positive polarity synchronous reset.

Truth table:	D	C	R		Q
	-----+---				
	-	/	1		0
	d	/	-		d
	-	-	-		q

Simulation model (verilog)

Listing 9.226: simcells.v

```
2435 module \$_SDFF_PP0_ (D, C, R, Q);
2436     input D, C, R;
2437     output reg Q;
2438     always @(posedge C) begin
2439         if (R == 1)
2440             Q <= 0;
2441         else
2442             Q <= D;
2443     end
2444 endmodule
```

yosys> help \$_SDFF_PP1_

A positive edge D-type flip-flop with positive polarity synchronous set.

Truth table:	D	C	R		Q
	-----+---				
	-	/	1		1
	d	/	-		d
	-	-	-		q

Simulation model (verilog)

Listing 9.227: simcells.v

```

2459 module \$_SDFF_PP1_ (D, C, R, Q);
2460     input D, C, R;
2461     output reg Q;
2462     always @(posedge C) begin
2463         if (R == 1)
2464             Q <= 1;
2465         else
2466             Q <= D;
2467     end
2468 endmodule

```

9.2.4 Latch cells

The cell types `$_DLATCH_N_` and `$_DLATCH_P_` represent d-type latches.

Table 9.12: Cell types for basic latches

Verilog	Cell Type
<code>always @* if (!E) Q <= D</code>	<code>\$_DLATCH_N_</code>
<code>always @* if (E) Q <= D</code>	<code>\$_DLATCH_P_</code>

The cell types `$_DLATCH_[NP][NP][01]_` implement d-type latches with reset. The values in the table for these cell types relate to the following Verilog code template:

```

always @*
    if (R == RST_LVL)
        Q <= RST_VAL;
    else if (E == EN_LVL)
        Q <= D;

```

Table 9.13: Cell types for gate level logic networks (latches with reset)

<i>EnLvl</i>	<i>RstLvl</i>	<i>RstVal</i>	Cell Type
0	0	0	<code>\$_DLATCH_NNO_</code>
0	0	1	<code>\$_DLATCH>NN1_</code>
0	1	0	<code>\$_DLATCH_NPO_</code>
0	1	1	<code>\$_DLATCH_NP1_</code>
1	0	0	<code>\$_DLATCH_PNO_</code>
1	0	1	<code>\$_DLATCH_PN1_</code>
1	1	0	<code>\$_DLATCH_PPO_</code>
1	1	1	<code>\$_DLATCH_PP1_</code>

The cell types `$_DLATCHSR_[NP][NP][NP]_` implement d-type latches with set and reset. The values in the table for these cell types relate to the following Verilog code template:

```

always @*
    if (R == RST_LVL)

```

(continues on next page)

(continued from previous page)

```

    Q <= 0;
else if (S == SET_LVL)
    Q <= 1;
else if (E == EN_LVL)
    Q <= D;

```

Table 9.14: Cell types for gate level logic networks (latches with set and reset)

<i>EnLvl</i>	<i>SetLvl</i>	<i>RstLvl</i>	Cell Type
0	0	0	<code>\$_DLATCHSR_NNN_</code>
0	0	1	<code>\$_DLATCHSR_NNP_</code>
0	1	0	<code>\$_DLATCHSR_NPN_</code>
0	1	1	<code>\$_DLATCHSR_NPP_</code>
1	0	0	<code>\$_DLATCHSR_PNN_</code>
1	0	1	<code>\$_DLATCHSR_PNP_</code>
1	1	0	<code>\$_DLATCHSR_PPN_</code>
1	1	1	<code>\$_DLATCHSR_PPP_</code>

The cell types `$_SR_[NP][NP]_` implement sr-type latches. The values in the table for these cell types relate to the following Verilog code template:

```

always @*
  if (R == RST_LVL)
    Q <= 0;
  else if (S == SET_LVL)
    Q <= 1;

```

Table 9.15: Cell types for gate level logic networks (SR latches)

<i>SetLvl</i>	<i>RstLvl</i>	Cell Type
0	0	<code>\$_SR_NN_</code>
0	1	<code>\$_SR_NP_</code>
1	0	<code>\$_SR_PN_</code>
1	1	<code>\$_SR_PP_</code>

yosys> help `$_DLATCHSR_NNN_`

A negative enable D-type latch with negative polarity set and negative polarity reset.

```

Truth table:   E S R D | Q
               - - - - + -
               - - 0 - | 0
               - 0 - - | 1
               0 - - d | d
               - - - - | q

```

Simulation model (verilog)

Listing 9.228: simcells.v

```

3551 module \$_DLATCHSR_NNN_ (E, S, R, D, Q);
3552     input E, S, R, D;
3553     output reg Q;
3554     always @* begin
3555         if (R == 0)
3556             Q <= 0;
3557         else if (S == 0)
3558             Q <= 1;
3559         else if (E == 0)
3560             Q <= D;
3561     end
3562 endmodule

```

yosys> help \$_DLATCHSR_NNP_

A negative enable D-type latch with negative polarity set and positive polarity reset.

Truth table:	E	S	R	D		Q
	-	-	1	-		0
	-	0	-	-		1
	0	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.229: simcells.v

```

3579 module \$_DLATCHSR_NNP_ (E, S, R, D, Q);
3580     input E, S, R, D;
3581     output reg Q;
3582     always @* begin
3583         if (R == 1)
3584             Q <= 0;
3585         else if (S == 0)
3586             Q <= 1;
3587         else if (E == 0)
3588             Q <= D;
3589     end
3590 endmodule

```

yosys> help \$_DLATCHSR_NPN_

A negative enable D-type latch with positive polarity set and negative polarity reset.

Truth table:	E	S	R	D		Q
	-	-	0	-		0
	-	1	-	-		1
	0	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.230: simcells.v

```

3607 module \$_DLATCHSR_NPN_ (E, S, R, D, Q);
3608     input E, S, R, D;
3609     output reg Q;
3610     always @* begin
3611         if (R == 0)
3612             Q <= 0;
3613         else if (S == 1)
3614             Q <= 1;
3615         else if (E == 0)
3616             Q <= D;
3617     end
3618 endmodule

```

yosys> help \$_DLATCHSR_NPP_

A negative enable D-type latch with positive polarity set and positive polarity reset.

Truth table:	E	S	R	D		Q
	-	-	1	-		0
	-	1	-	-		1
	0	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.231: simcells.v

```

3635 module \$_DLATCHSR_NPP_ (E, S, R, D, Q);
3636     input E, S, R, D;
3637     output reg Q;
3638     always @* begin
3639         if (R == 1)
3640             Q <= 0;
3641         else if (S == 1)
3642             Q <= 1;
3643         else if (E == 0)
3644             Q <= D;
3645     end
3646 endmodule

```

yosys> help \$_DLATCHSR_PNN_

A positive enable D-type latch with negative polarity set and negative polarity reset.

Truth table:	E	S	R	D		Q
	-	-	0	-		0
	-	0	-	-		1
	1	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.232: simcells.v

```

3663 module \$_DLATCHSR_PNN_ (E, S, R, D, Q);
3664     input E, S, R, D;
3665     output reg Q;
3666     always @* begin
3667         if (R == 0)
3668             Q <= 0;
3669         else if (S == 0)
3670             Q <= 1;
3671         else if (E == 1)
3672             Q <= D;
3673     end
3674 endmodule

```

yosys> help \\$_DLATCHSR_PNP_

A positive enable D-type latch with negative polarity set and positive polarity reset.

Truth table:	E	S	R	D		Q
	-	-	1	-		0
	-	0	-	-		1
	1	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.233: simcells.v

```

3691 module \$_DLATCHSR_PNP_ (E, S, R, D, Q);
3692     input E, S, R, D;
3693     output reg Q;
3694     always @* begin
3695         if (R == 1)
3696             Q <= 0;
3697         else if (S == 0)
3698             Q <= 1;
3699         else if (E == 1)
3700             Q <= D;
3701     end
3702 endmodule

```

yosys> help \\$_DLATCHSR_PPN_

A positive enable D-type latch with positive polarity set and negative polarity reset.

Truth table:	E	S	R	D		Q
	-	-	0	-		0
	-	1	-	-		1
	1	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.234: simcells.v

```

3719 module \$_DLATCHSR_PPN_ (E, S, R, D, Q);
3720     input E, S, R, D;
3721     output reg Q;
3722     always @* begin
3723         if (R == 0)
3724             Q <= 0;
3725         else if (S == 1)
3726             Q <= 1;
3727         else if (E == 1)
3728             Q <= D;
3729     end
3730 endmodule

```

yosys> help \$_DLATCHSR_PPP_

A positive enable D-type latch with positive polarity set and positive polarity reset.

Truth table:	E	S	R	D		Q
	-	-	1	-		0
	-	1	-	-		1
	1	-	-	d		d
	-	-	-	-		q

Simulation model (verilog)

Listing 9.235: simcells.v

```

3747 module \$_DLATCHSR_PPP_ (E, S, R, D, Q);
3748     input E, S, R, D;
3749     output reg Q;
3750     always @* begin
3751         if (R == 1)
3752             Q <= 0;
3753         else if (S == 1)
3754             Q <= 1;
3755         else if (E == 1)
3756             Q <= D;
3757     end
3758 endmodule

```

yosys> help \$_DLATCH_NNO_

A negative enable D-type latch with negative polarity reset.

Truth table:	E	R	D		Q
	-	0	-		0
	0	-	d		d
	-	-	-		q

Simulation model (verilog)

Listing 9.236: simcells.v

```

3357 module \$_DLATCH_NNO_ (E, R, D, Q);
3358     input E, R, D;
3359     output reg Q;
3360     always @* begin
3361         if (R == 0)
3362             Q <= 0;
3363         else if (E == 0)
3364             Q <= D;
3365     end
3366 endmodule

```

yosys> help \$_DLATCH_NN1_

A negative enable D-type latch with negative polarity set.

Truth table:	E	R	D		Q
	-	0	-		1
	0	-	d		d
	-	-	-		q

Simulation model (verilog)

Listing 9.237: simcells.v

```

3381 module \$_DLATCH_NN1_ (E, R, D, Q);
3382     input E, R, D;
3383     output reg Q;
3384     always @* begin
3385         if (R == 0)
3386             Q <= 1;
3387         else if (E == 0)
3388             Q <= D;
3389     end
3390 endmodule

```

yosys> help \$_DLATCH_NPO_

A negative enable D-type latch with positive polarity reset.

Truth table:	E	R	D		Q
	-	1	-		0
	0	-	d		d
	-	-	-		q

Simulation model (verilog)

Listing 9.238: simcells.v

```

3405 module \$_DLATCH_NPO_ (E, R, D, Q);
3406     input E, R, D;
3407     output reg Q;

```

(continues on next page)

(continued from previous page)

```

3408     always @* begin
3409         if (R == 1)
3410             Q <= 0;
3411         else if (E == 0)
3412             Q <= D;
3413     end
3414 endmodule

```

yosys> help \$_DLATCH_NP1_

A negative enable D-type latch with positive polarity set.

```

Truth table:   E R D | Q
              - - - + -
              - 1 - | 1
              0 - d | d
              - - - | q

```

Simulation model (verilog)

Listing 9.239: simcells.v

```

3429 module \$_DLATCH_NP1_ (E, R, D, Q);
3430     input E, R, D;
3431     output reg Q;
3432     always @* begin
3433         if (R == 1)
3434             Q <= 1;
3435         else if (E == 0)
3436             Q <= D;
3437     end
3438 endmodule

```

yosys> help \$_DLATCH_N_

A negative enable D-type latch.

```

Truth table:   E D | Q
              - - - + -
              0 d | d
              - - | q

```

Simulation model (verilog)

Listing 9.240: simcells.v

```

3314 module \$_DLATCH_N_ (E, D, Q);
3315     input E, D;
3316     output reg Q;
3317     always @* begin
3318         if (E == 0)
3319             Q <= D;
3320     end
3321 endmodule

```

yosys> help \$_DLATCH_PNO_

A positive enable D-type latch with negative polarity reset.

Truth table:	E	R	D	Q
	-	0	-	0
	1	-	d	d
	-	-	-	q

Simulation model (verilog)

Listing 9.241: simcells.v

```

3453 module \$_DLATCH_PNO_ (E, R, D, Q);
3454     input E, R, D;
3455     output reg Q;
3456     always @* begin
3457         if (R == 0)
3458             Q <= 0;
3459         else if (E == 1)
3460             Q <= D;
3461     end
3462 endmodule

```

yosys> help \$_DLATCH_PN1_

A positive enable D-type latch with negative polarity set.

Truth table:	E	R	D	Q
	-	0	-	1
	1	-	d	d
	-	-	-	q

Simulation model (verilog)

Listing 9.242: simcells.v

```

3477 module \$_DLATCH_PN1_ (E, R, D, Q);
3478     input E, R, D;
3479     output reg Q;
3480     always @* begin
3481         if (R == 0)
3482             Q <= 1;
3483         else if (E == 1)
3484             Q <= D;
3485     end
3486 endmodule

```

yosys> help \$_DLATCH_PP0_

A positive enable D-type latch with positive polarity reset.

Truth table:	E	R	D	Q
	-	0	-	0
	1	-	d	d
	-	-	-	q

(continues on next page)

(continued from previous page)

-	1	-		0
1	-	d		d
-	-	-		q

Simulation model (verilog)

Listing 9.243: simcells.v

```

3501 module \$_DLATCH_PPO_ (E, R, D, Q);
3502     input E, R, D;
3503     output reg Q;
3504     always @* begin
3505         if (R == 1)
3506             Q <= 0;
3507         else if (E == 1)
3508             Q <= D;
3509     end
3510 endmodule

```

yosys> help \$_DLATCH_PP1_

A positive enable D-type latch with positive polarity set.

Truth table:

E	R	D		Q
-----+---				
-	1	-		1
1	-	d		d
-	-	-		q

Simulation model (verilog)

Listing 9.244: simcells.v

```

3525 module \$_DLATCH_PP1_ (E, R, D, Q);
3526     input E, R, D;
3527     output reg Q;
3528     always @* begin
3529         if (R == 1)
3530             Q <= 1;
3531         else if (E == 1)
3532             Q <= D;
3533     end
3534 endmodule

```

yosys> help \$_DLATCH_P_

A positive enable D-type latch.

Truth table:

E	D		Q
-----+---			
1	d		d
-	-		q

Simulation model (verilog)

Listing 9.245: simcells.v

```

3335 module \$_DLATCH_P_ (E, D, Q);
3336     input E, D;
3337     output reg Q;
3338     always @* begin
3339         if (E == 1)
3340             Q <= D;
3341     end
3342 endmodule

```

yosys> help \$_SR_NN_

A set-reset latch with negative polarity SET and negative polarity RESET.

Truth table:	S R Q
	-----+---
	- 0 0
	0 - 1
	- - q

Simulation model (verilog)

Listing 9.246: simcells.v

```

497 module \$_SR_NN_ (S, R, Q);
498     input S, R;
499     output reg Q;
500     always @* begin
501         if (R == 0)
502             Q <= 0;
503         else if (S == 0)
504             Q <= 1;
505     end
506 endmodule

```

yosys> help \$_SR_NP_

A set-reset latch with negative polarity SET and positive polarity RESET.

Truth table:	S R Q
	-----+---
	- 1 0
	0 - 1
	- - q

Simulation model (verilog)

Listing 9.247: simcells.v

```

521 module \$_SR_NP_ (S, R, Q);
522     input S, R;
523     output reg Q;
524     always @* begin
525         if (R == 1)

```

(continues on next page)

(continued from previous page)

```

526         Q <= 0;
527     else if (S == 0)
528         Q <= 1;
529     end
530 endmodule

```

yosys> help \$_SR_PN_

A set-reset latch with positive polarity SET and negative polarity RESET.

```

Truth table:   S R | Q
              -----+---
                - 0 | 0
                1 - | 1
                - - | q

```

Simulation model (verilog)

Listing 9.248: simcells.v

```

545 module \$_SR_PN_ (S, R, Q);
546     input S, R;
547     output reg Q;
548     always @* begin
549         if (R == 0)
550             Q <= 0;
551         else if (S == 1)
552             Q <= 1;
553     end
554 endmodule

```

yosys> help \$_SR_PP_

A set-reset latch with positive polarity SET and positive polarity RESET.

```

Truth table:   S R | Q
              -----+---
                - 1 | 0
                1 - | 1
                - - | q

```

Simulation model (verilog)

Listing 9.249: simcells.v

```

569 module \$_SR_PP_ (S, R, Q);
570     input S, R;
571     output reg Q;
572     always @* begin
573         if (R == 1)
574             Q <= 0;
575         else if (S == 1)
576             Q <= 1;
577     end
578 endmodule

```

9.2.5 Other gate-level cells

Other gate-level cells

```
yosys> help $_TBUF_
A tri-state buffer.
```

```
Truth table:   A E | Y
               -----+---
               a 1 | a
               - 0 | z
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.250: simcells.v

```
473 module \$_TBUF_ (A, E, Y);
474     input A, E;
475     output Y;
476     assign Y = E ? A : 1'bz;
477 endmodule
```

9.3 Cell properties

is_evaluable

These cells are able to be used in conjunction with the *eval* command. Some passes, such as *opt_expr*, may also be able to perform additional optimizations on cells which are evaluable.

x-aware

Some passes will treat these cells as the non ‘x’ aware cell. For example, during synthesis *\$eqx* will typically be treated as *\$eq*.

x-output

These cells can produce ‘x’ output even if all inputs are defined. For example, a *\$div* cell with divisor (B) equal to zero has undefined output.

Refer to the propindex for the list of cells with a given property.

COMMAND LINE REFERENCE

Usage:

```
./yosys [OPTION...] [<infile> [...]]
```

operation options:

```
-b, --backend <backend>      use <backend> for the output file specified on the
↳command line
-f, --frontend <frontend>    use <frontend> for the input files on the command line
-s, --scriptfile <scriptfile>
                              execute the commands in <scriptfile>
-c, --tcl-scriptfile <tcl_scriptfile>
                              execute the commands in the TCL <tcl_scriptfile> (see
↳'help tcl' for details)
-C, --tcl-interactive        enters TCL interactive shell mode
-p, --commands <commands>    execute <commands> (to chain commands, separate them
↳with semicolon + whitespace: 'cmd1; cmd2')
-r, --top <top>              elaborate the specified HDL <top> module
-m, --plugin <plugin>        load the specified <plugin> module
-D, --define <define>[=<value>]
                              set the specified Verilog define to <value> if supplied
↳via command "read -define"
-S, --synth                  shortcut for calling the "synth" command, a default
↳script for transforming the Verilog input to a gate-level netlist. For example: yosys -
↳o output.blif -S input.v For more complex synthesis jobs it is recommended to use the
↳read_* and write_* commands in a script file instead of specifying input and output
↳files on the command line.
-H                            print the command list
-h, --help [<command>]       print this help message. If given, print help for
↳<command>.
-V, --version                print version information and exit
```

logging options:

```
-Q                            suppress printing of banner (copyright, disclaimer,
↳version)
-T                            suppress printing of footer (log hash, version, timing
↳statistics)
-q, --quiet                  quiet operation. Only write warnings and error messages
↳to console. Use this option twice to also quiet warning messages
-v, --verbose <level>       print log headers up to <level> to the console. Implies -
↳q for everything except the 'End of script.' message.
-t, --timestamp              annotate all log messages with a time stamp
```

(continues on next page)

(continued from previous page)

```

-d, --detailed-timing      print more detailed timing stats at exit
-l, --logfile <logfile>    write log messages to <logfile>
-L, --line-buffered-logfile <logfile>
                             like -l but open <logfile> in line buffered mode
-o, --outfile <outfile>    write the design to <outfile> on exit
-P, --dump-design <header_id>[:<filename>]
                             dump the design when printing the specified log header
↳to a file. yosys_dump_<header_id>.il is used as filename if none is specified. Use 'ALL
↳' as <header_id> to dump at every header.
-W, --warning-as-warning <regex>
                             print a warning for all log messages matching <regex>
-w, --warning-as-message <regex>
                             if a warning message matches <regex>, it is printed as
↳regular message instead
-e, --warning-as-error <regex>
                             if a warning message matches <regex>, it is printed as
↳error message instead
-E, --deps-file <depsfile> write a Makefile dependencies file <depsfile> with input
↳and output file names

developer options:
-X, --trace                enable tracing of core data structure changes. for
↳debugging
-M, --randomize-pointers   will slightly randomize allocated pointer addresses. for
↳debugging
    --autoidx <idx>        start counting autoidx up from <seed>, similar effect to
↳--hash-seed
    --hash-seed <seed>     mix up hashing values with <seed>, for extreme
↳optimization and testing
-A, --abort                will call abort() at the end of the script. for debugging
-x, --experimental <feature> do not print warnings for the experimental <feature>
-g, --debug                globally enable debug log messages
    --perffile <perffile> write a JSON performance log to <perffile>

```

10.1 Yosys environment variables

HOME

Yosys command history is stored in `$HOME/.yosys_history`. Graphics (from `show` and `viz` commands) will output to this directory by default. This environment variable is also used in some cases for resolving filenames with `~`.

PATH

May be used in OpenBSD builds for finding the location of Yosys executable.

TMPDIR

Used for storing temporary files.

ABC

When compiling Yosys with out-of-tree ABC using `ABCEXTERNAL`, this variable can be used to override the external ABC executable.

YOSYS_NOVERIFIC

If Yosys was built with Verific, this environment variable can be used to temporarily disable Verific support.

YOSYS_COVER_DIR and YOSYS_COVER_FILE

When using code coverage, these environment variables control the output file name/location.

YOSYS_ABORT_ON_LOG_ERROR

Can be used for debugging Yosys internals. Setting it to 1 causes abort() to be called when Yosys terminates with an error message.

10.2 abc - use ABC for technology mapping

yosys> help abc

```
abc [options] [selection]
```

This pass uses the ABC tool [1] for technology mapping of yosys's internal gate library to a target architecture.

```
-exe <command>
```

use the specified command instead of "/usr/bin/abc" to execute ABC.
This can e.g. be used to call a specific version of ABC or a wrapper.

```
-script <file>
```

use the specified ABC script file instead of the default script.

if <file> starts with a plus sign (+), then the rest of the filename string is interpreted as the command string to be passed to ABC. The leading plus sign is removed and all commas (,) in the string are replaced with blanks before the string is passed to ABC.

if no -script parameter is given, the following scripts are used:

```
for -liberty/-genlib without -constr:
```

```
    strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
    &get -n; &dch -f; &nf {D}; &put
```

```
for -liberty/-genlib with -constr:
```

```
    strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
    &get -n; &dch -f; &nf {D}; &put; buffer; upsize {D};
    dnsiz {D}; stime -p
```

```
for -lut/-luts (only one LUT size):
```

```
    strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
    dch -f; if; mfs2; lutpack {S}
```

```
for -lut/-luts (different LUT sizes):
```

```
    strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
    dch -f; if; mfs2
```

```
for -sop:
```

(continues on next page)

(continued from previous page)

```

    strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
    dch -f; cover {I} {P}

otherwise:
    strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
    &get -n; &dch -f; &nf {D}; &put

-fast
use different default scripts that are slightly faster (at the cost
of output quality):

for -liberty/-genlib without -constr:
    strash; dretime; map {D}

for -liberty/-genlib with -constr:
    strash; dretime; map {D}; buffer; upsize {D}; dnsiz {D};
    stime -p

for -lut/-luts:
    strash; dretime; if

for -sop:
    strash; dretime; cover {I} {P}

otherwise:
    strash; dretime; map

-liberty <file>
generate netlists for the specified cell library (using the liberty
file format).

-dont_use <cell_name>
generate netlists for the specified cell library (using the liberty
file format).

-genlib <file>
generate netlists for the specified cell library (using the SIS Genlib
file format).

-constr <file>
pass this file with timing constraints to ABC.
use with -liberty/-genlib.

a constr file contains two lines:
    set_driving_cell <cell_name>
    set_load <floating_point_number>

the set_driving_cell statement defines which cell type is assumed to
drive the primary inputs and the set_load statement sets the load in
femtofarads for each primary output.

-D <picoseconds>

```

(continues on next page)

(continued from previous page)

```

set delay target. the string {D} in the default scripts above is
replaced by this option when used, and an empty string otherwise.
this also replaces 'dretime' with 'dretime; retime -o {D}' in the
default scripts above.

-I <num>
    maximum number of SOP inputs.
    (replaces {I} in the default scripts above)

-P <num>
    maximum number of SOP products.
    (replaces {P} in the default scripts above)

-S <num>
    maximum number of LUT inputs shared.
    (replaces {S} in the default scripts above, default: -S 1)

-lut <width>
    generate netlist using luts of (max) the specified width.

-lut <w1>:<w2>
    generate netlist using luts of (max) the specified width <w2>. All
    luts with width <= <w1> have constant cost. for luts larger than <w1>
    the area cost doubles with each additional input bit. the delay cost
    is still constant for all lut widths.

-luts <cost1>,<cost2>,<cost3>,<sizeN>:<cost4-N>,...
    generate netlist using luts. Use the specified costs for luts with 1,
    2, 3, .. inputs.

-sop
    map to sum-of-product cells and inverters

-g type1,type2,...
    Map to the specified list of gate types. Supported gates types are:
    AND, NAND, OR, NOR, XOR, XNOR, ANDNOT, ORNOT, MUX,
    NMUX, AOI3, OAI3, AOI4, OAI4.
    (The NOT gate is always added to this list automatically.)

The following aliases can be used to reference common sets of gate
types:
    simple: AND OR XOR MUX
    cmos2:  NAND NOR
    cmos3:  NAND NOR AOI3 OAI3
    cmos4:  NAND NOR AOI3 OAI3 AOI4 OAI4
    cmos:   NAND NOR AOI3 OAI3 AOI4 OAI4 NMUX MUX XOR XNOR
    gates:  AND NAND OR NOR XOR XNOR ANDNOT ORNOT
    aig:    AND NAND OR NOR ANDNOT ORNOT

The alias 'all' represent the full set of all gate types.

Prefix a gate type with a '-' to remove it from the list. For example

```

(continues on next page)

(continued from previous page)

the arguments 'AND,OR,XOR' and 'simple,-MUX' are equivalent.

The default is 'all,-NMUX,-AOI3,-OAI3,-AOI4,-OAI4'.

-dff

also pass `$_DFF_?` and `$_DFFE_??` cells through ABC. modules with many clock domains are automatically partitioned in clock domains and each domain is passed through ABC independently.

-clk [!]<clock-signal-name>[, [!]<enable-signal-name>]

use only the specified clock domain. this is like -dff, but only FF cells that belong to the specified clock domain are used.

-keepff

set the "keep" attribute on flip-flop output wires. (and thus preserve them, for example for equivalence checking.)

-nocleanup

when this option is used, the temporary files created by this pass are not removed. this is useful for debugging.

-showtmp

print the temp dir name in log. usually this is suppressed so that the command output is identical across runs.

-markgroups

set a 'abcgroup' attribute on all objects created by ABC. The value of this attribute is a unique integer for each ABC process started. This is useful for debugging the partitioning of clock domains.

-dress

run the 'dress' command after all other ABC commands. This aims to preserve naming by an equivalence check between the original and post-ABC netlists (experimental).

When no target cell library is specified the Yosys standard cell library is loaded into ABC before the ABC script is executed.

Note that this is a logic optimization pass within Yosys that is calling ABC internally. This is not going to "run ABC on your design". It will instead run ABC on logic snippets extracted from your design. You will not get any useful output when passing an ABC script that writes a file. Instead write your full design as BLIF file with `write_blif` and then load that into ABC externally if you want to use ABC to convert your design into another format.

[1] <http://www.eecs.berkeley.edu/~alanmi/abc/>

10.3 abc9 - use ABC9 for technology mapping

yosys> help abc9

```
abc9 [options] [selection]
```

This script pass performs a sequence of commands to facilitate the use of the ABC tool [1] for technology mapping of the current design to a target FPGA architecture. Only fully-selected modules are supported.

```
-run <from_label>:<to_label>
```

only run the commands between the labels (see below). an empty from label is synonymous to 'begin', and empty to label is synonymous to the end of the command list.

```
-exe <command>
```

use the specified command instead of "/usr/bin/abc" to execute ABC. This can e.g. be used to call a specific version of ABC or a wrapper.

```
-script <file>
```

use the specified ABC script file instead of the default script.

if <file> starts with a plus sign (+), then the rest of the filename string is interpreted as the command string to be passed to ABC. The leading plus sign is removed and all commas (,) in the string are replaced with blanks before the string is passed to ABC.

if no -script parameter is given, the following scripts are used:
&scorr; &sweep; &dc2; &dch -f; &ps; &if {C} {W} {D} {R} -v; &mfs

```
-fast
```

use different default scripts that are slightly faster (at the cost of output quality):
&if {C} {W} {D} {R} -v

```
-D <picoseconds>
```

set delay target. the string {D} in the default scripts above is replaced by this option when used, and an empty string otherwise (indicating best possible delay).

```
-lut <width>
```

generate netlist using luts of (max) the specified width.

```
-lut <w1>:<w2>
```

generate netlist using luts of (max) the specified width <w2>. All luts with width <= <w1> have constant cost. for luts larger than <w1> the area cost doubles with each additional input bit. the delay cost is still constant for all lut widths.

```
-lut <file>
```

pass this file with lut library to ABC.

(continues on next page)

(continued from previous page)

```

-luts <cost1>,<cost2>,<cost3>,<sizeN>:<cost4-N>,...
    generate netlist using luts. Use the specified costs for luts with 1,
    2, 3, .. inputs.

-maxlut <width>
    when auto-generating the lut library, discard all luts equal to or
    greater than this size (applicable when neither -lut nor -luts is
    specified).

-dff
    also pass $_DFF_[NP]_ cells through to ABC. modules with many clock
    domains are supported and automatically partitioned by ABC.

-nocleanup
    when this option is used, the temporary files created by this pass
    are not removed. this is useful for debugging.

-showtmp
    print the temp dir name in log. usually this is suppressed so that the
    command output is identical across runs.

-box <file>
    pass this file with box library to ABC.

```

Note that this is a logic optimization pass within Yosys that is calling ABC internally. This is not going to "run ABC on your design". It will instead run ABC on logic snippets extracted from your design. You will not get any useful output when passing an ABC script that writes a file. Instead write your full design as an XAIGER file with ``write_xaiger'` and then load that into ABC externally if you want to use ABC to convert your design into another format.

[1] <http://www.eecs.berkeley.edu/~alanmi/abc/>

```

check:
    abc9_ops -check [-dff]      (option if -dff)

map:
    abc9_ops -prep_hier [-dff]  (option if -dff)
    scc -specify -set_attr abc9_scc_id {}
    abc9_ops -prep_bypass [-prep_dff]  (option if -dff)
    design -stash $abc9
    design -load $abc9_map
    proc
    wbflip
    techmap -wb -map %$abc9 -map +/techmap.v A:abc9_flop
    opt -nodffe -nosdff
    abc9_ops -prep_dff_submod
↳ (only if -dff)
    setattr -set submod "$abc9_flop" t:$_DFF_?_ %ci* %co* t:$_DFF_?_ %d
↳ (only if -dff)
    submod

```

(continues on next page)

(continued from previous page)

```

↪(only if -dff)
    setattr -mod -set whitebox 1 -set abc9_flop 1 -set abc9_box 1 *$_abc9_flop
↪(only if -dff)
    foreach module in design
        rename <module-name>$_abc9_flop _TECHMAP_REPLACE_
↪(only if -dff)
    abc9_ops -prep_dff_unmap
↪(only if -dff)
    design -copy-to $abc9 =*$_abc9_flop
↪(only if -dff)
    delete =*$_abc9_flop
↪(only if -dff)
    design -stash $abc9_map
    design -load $abc9
    design -delete $abc9
    techmap -wb -max_iter 1 -map %$abc9_map -map +/abc9_map.v [-D DFF] (option if
↪-dff)
    design -delete $abc9_map

pre:
    read_verilog -icells -lib -specify +/abc9_model.v
    abc9_ops -break_scc -prep_delays -prep_xaiger [-dff] (option for -dff)
    abc9_ops -prep_lut <maxlut> (skip if -lut or -luts)
    abc9_ops -prep_box (skip if -box)
    design -stash $abc9
    design -load $abc9_holes
    techmap -wb -map %$abc9 -map +/techmap.v
    opt -purge
    aigmap
    design -stash $abc9_holes
    design -load $abc9
    design -delete $abc9

exe:
    aigmap
    foreach module in selection
        abc9_ops -write_lut <abc-temp-dir>/input.lut (skip if '-lut' or '-luts')
        abc9_ops -write_box <abc-temp-dir>/input.box (skip if '-box')
        write_xaiger -map <abc-temp-dir>/input.sym [-dff] <abc-temp-dir>/input.xaig
        abc9_exe [options] -cwd <abc-temp-dir> -lut [<abc-temp-dir>/input.lut] -box [
↪<abc-temp-dir>/input.box]
        read_aiger -xaiger -wideports -module_name <module-name>$abc9 -map <abc-temp-
↪dir>/input.sym <abc-temp-dir>/output.aig
        abc9_ops -reintegrate [-dff]

unmap:
    techmap -wb -map %$abc9_unmap -map +/abc9_unmap.v
    design -delete $abc9_unmap
    design -delete $abc9_holes
    delete =*$_abc9_byp
    setattr -mod -unset abc9_box_id

```

10.4 abc9_exe - use ABC9 for technology mapping

yosys> help abc9_exe

abc9_exe [options]

This pass uses the ABC tool [1] for technology mapping of the top module (according to the (* top *) attribute or if only one module is currently selected) to a target FPGA architecture.

-exe <command>

use the specified command instead of "/usr/bin/abc" to execute ABC.
This can e.g. be used to call a specific version of ABC or a wrapper.

-script <file>

use the specified ABC script file instead of the default script.

if <file> starts with a plus sign (+), then the rest of the filename string is interpreted as the command string to be passed to ABC. The leading plus sign is removed and all commas (,) in the string are replaced with blanks before the string is passed to ABC.

if no -script parameter is given, the following scripts are used:
&scorr; &sweep; &dc2; &dch -f; &ps; &if {C} {W} {D} {R} -v; &mfs

-fast

use different default scripts that are slightly faster (at the cost of output quality):
&if {C} {W} {D} {R} -v

-constr <file>

pass this file with timing constraints to ABC.
use with -liberty.

a constr file contains two lines:
set_driving_cell <cell_name>
set_load <floating_point_number>

the set_driving_cell statement defines which cell type is assumed to drive the primary inputs and the set_load statement sets the load in femtofarads for each primary output.

-liberty <file>

read the given Liberty file as a description of the target cell library.
this option can be used multiple times.

-dont_use <cell_name>

avoid usage of the technology cell <cell_name> when mapping the design.
this option can be used multiple times.

-D <picoseconds>

(continues on next page)

(continued from previous page)

```

set delay target. the string {D} in the default scripts above is
replaced by this option when used, and an empty string otherwise
(indicating best possible delay).

-lut <width>
    generate netlist using luts of (max) the specified width.

-lut <w1>:<w2>
    generate netlist using luts of (max) the specified width <w2>. All
    luts with width <= <w1> have constant cost. for luts larger than <w1>
    the area cost doubles with each additional input bit. the delay cost
    is still constant for all lut widths.

-lut <file>
    pass this file with lut library to ABC.

-luts <cost1>,<cost2>,<cost3>,<sizeN>:<cost4-N>,...
    generate netlist using luts. Use the specified costs for luts with 1,
    2, 3, .. inputs.

-showtmp
    print the temp dir name in log. usually this is suppressed so that the
    command output is identical across runs.

-box <file>
    pass this file with box library to ABC.

-cwd <dir>
    use this as the current working directory, inside which the 'input.xaig'
    file is expected. temporary files will be created in this directory, and
    the mapped result will be written to 'output.aig'.

```

Note that this is a logic optimization pass within Yosys that is calling ABC internally. This is not going to "run ABC on your design". It will instead run ABC on logic snippets extracted from your design. You will not get any useful output when passing an ABC script that writes a file. Instead write your full design as BLIF file with `write_blif` and then load that into ABC externally if you want to use ABC to convert your design into another format.

[1] <http://www.eecs.berkeley.edu/~alanmi/abc/>

10.5 abc9_ops - helper functions for ABC9

```
yosys> help abc9_ops
```

```
abc9_ops [options] [selection]
```

This pass contains a set of supporting operations for use during ABC technology mapping, and is expected to be called in conjunction with other operations from the ``abc9'` script pass. Only fully-selected modules are supported.

(continues on next page)

(continued from previous page)

```

-check
    check that the design is valid, e.g. (* abc9_box_id *) values are
    unique, (* abc9_carry *) is only given for one input/output port, etc.

-prep_hier
    derive all used (* abc9_box *) or (* abc9_flop *) (if -dff option)
    whitebox modules. with (* abc9_flop *) modules, only those containing
    $dff/$_DFF_[NP]_ cells with zero initial state -- due to an ABC
    limitation -- will be derived.

-prep_bypass
    create techmap rules in the '$abc9_map' and '$abc9_unmap' designs for
    bypassing sequential (* abc9_box *) modules using a combinatorial box
    (named *_abc9_byp). bypassing is necessary if sequential elements (e.g.
    $dff, $mem, etc.) are discovered inside so that any combinatorial paths
    will be correctly captured. this bypass box will only contain ports that
    are referenced by a simple path declaration ($specify2 cell) inside a
    specify block.

-prep_dff
    select all (* abc9_flop *) modules instantiated in the design and store
    in the named selection '$abc9_flops'.

-prep_dff_submod
    within (* abc9_flop *) modules, rewrite all edge-sensitive path
    declarations and $setup() timing checks ($specify3 and $specrule cells)
    that share a 'DST' port with the $_DFF_[NP]_.Q port from this 'Q' port
    to the DFF's 'D' port. this is to prepare such specify cells to be moved
    into the flop box.

-prep_dff_unmap
    populate the '$abc9_unmap' design with techmap rules for mapping
    *_abc9_flop cells back into their derived cell types (where the rules
    created by -prep_hier will then map back to the original cell with
    parameters).

-prep_delays
    insert `$_ABC9_DELAY' blackbox cells into the design to account for
    certain required times.

-break_scc
    for an arbitrarily chosen cell in each unique SCC of each selected
    module (tagged with an (* abc9_scc_id = <int> *) attribute) interrupt
    all wires driven by this cell's outputs with a temporary
    $_ABC9_SCC_BREAKER cell to break the SCC.

-prep_xaiger
    prepare the design for XAIGER output. this includes computing the
    topological ordering of ABC9 boxes, as well as preparing the
    '$abc9_holes' design that contains the logic behaviour of ABC9
    whiteboxes.

```

(continues on next page)

(continued from previous page)

```

-dff
    consider flop cells (those instantiating modules marked with
    (* abc9_flop *)) during -prep_{delays,xaiger,box}.

-prep_lut <maxlut>
    pre-compute the lut library by analysing all modules marked with
    (* abc9_lut=<area> *).

-write_lut <dst>
    write the pre-computed lut library to <dst>.

-prep_box
    pre-compute the box library by analysing all modules marked with
    (* abc9_box *).

-write_box <dst>
    write the pre-computed box library to <dst>.

-reintegrate
    for each selected module, re-intergrate the module '<module-name>$abc9'
    by first recovering ABC9 boxes, and then stitching in the remaining
    primary inputs and outputs.

```

10.6 abc_new - (experimental) use ABC for SC technology mapping (new)

yosys> help abc_new

```
abc_new [options] [selection]
```

This command uses the ABC tool [1] to optimize the current design and map it to the target standard cell library.

```

-run <from_label>:<to_label>
    only run the commands between the labels (see below). an empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-exe <command>
-script <file>
-D <picoseconds>
-constr <file>
-dont_use <cell_name>
-liberty <file>
    these options are passed on to the 'abc9_exe' command which invokes
    the ABC tool on individual modules of the design. please see
    'help abc9_exe' for more details

```

(continues on next page)

(continued from previous page)

```
[1] http://www.eecs.berkeley.edu/~alanmi/abc/

check:
    abc9_ops -check

prep_boxes:
    box_derive
    abc9_ops -prep_box

map:
    foreach module in selection
        abc9_ops -write_box <abc-temp-dir>/input.box
        write_xaiger2 -mapping_prep -map2 <abc-temp-dir>/input.map2 <abc-temp-dir>/
↪input.xaig
        abc9_exe [options] -cwd <abc-temp-dir> -box <abc-temp-dir>/input.box
        read_xaiger2 -sc_mapping -module_name <module> -map2 <abc-temp-dir>/input.map2
↪<abc-temp-dir>/output.aig

WARNING: THE 'abc_new' COMMAND IS EXPERIMENTAL.
```

10.7 abstract - replace signals with abstract values during formal verification

yosys> help abstract

```
abstract [mode] [options] [selection]
```

Perform abstraction of signals within the design. Abstraction replaces a signal with an unconstrained abstract value that can take an arbitrary concrete value during formal verification. The mode and options control when a signal should be abstracted and how it should affect FFs present in the design.

Modes:

-state

The selected FFs will be modified to load a new abstract value on every active clock edge, async reset or async load. This is independent of any clock enable that may be present on the FF cell. Conditional abstraction is supported with the -enable/-enabeln options. If present, the condition is sampled at the same time as the FF would sample the corresponding data or async-data input whose value will be replaced with an abstract value.

The selection can be used to specify which state bits to abstract. For each selected wire, any state bits that the wire is driven by will be abstracted. For a selected FF cell, all of its state is abstracted. Individual bits of a single wire can be abstracted using the -slice and -rtlilslice options.

(continues on next page)

(continued from previous page)

-init

The selected FFs will be modified to have an abstract initial value. The `-enable/-enablen` options are not supported in this mode.

The selection is used in the same way as it is for the `-state` mode.

-value

The drivers of the selected signals will be replaced with an abstract value. In this mode, the abstract value can change at any time and is not synchronized to any clock or other signal. Conditional abstraction is supported with the `-enable/-enablen` options. The condition will combinationally select between the original driver and the abstract value.

The selection can be used to specify which output bits of which drivers to abstract. For a selected cell, all its output bits will be abstracted. For a selected wire, every output bit that is driving the wire will be abstracted. Individual bits of a single wire can be abstracted using the `-slice` and `-rtlilslice` options.

-enable <wire-name>**-enablen <wire-name>**

Perform conditional abstraction with a named single bit wire as condition. For `-enable` the wire is used as an active-high condition and for `-enablen` as an active-low condition. See the description of the `-state` and `-value` modes for details on how the condition affects the abstractions performed by either mode. This option is not supported in the `-init` mode.

-slice <lhs>:<rhs>**-slice <index>****-rtlilslice <lhs>:<rhs>****-rtlilslice <index>**

Limit the abstraction to a slice of a single selected wire. The targeted bits of the wire can be given as an inclusive range of indices or as a single index. When using the `-slice` option, the indices are interpreted following the source level declaration of the wire. This means the `-slice` option will respect declarations with a non-zero-based index range or with reversed bitorder. The `-rtlilslice` options will always use zero-based indexing where index 0 corresponds to the least significant bit of the wire.

10.8 add - add objects to the design

yosys> help add

```
add <command> [selection]
```

This command adds objects to the design. It operates on all fully selected modules. So e.g. 'add -wire foo' will add a wire foo to all selected modules.

```
add {-wire|-input|-inout|-output} <name> <width> [selection]
```

Add a wire (input, inout, output port) with the given name and width. The command will fail if the object exists already and has different properties than the object to be created.

```
add -global_input <name> <width> [selection]
```

Like 'add -input', but also connect the signal between instances of the selected modules.

```
add {-assert|-assume|-live|-fair|-cover} <name1> [-if <name2>]
```

Add an \$assert, \$assume, etc. cell connected to a wire named name1, with its enable signal optionally connected to a wire named name2 (default: 1'b1).

```
add -mod <name[s]>
```

Add module[s] with the specified name[s].

10.9 aigmap - map logic to and-inverter-graph circuit

yosys> help aigmap

```
aigmap [options] [selection]
```

Replace all logic cells with circuits made of only \$_AND_ and \$_NOT_ cells.

```
-nand
```

Enable creation of \$_NAND_ cells

```
-select
```

Overwrite replaced cells in the current selection with new \$_AND_, \$_NOT_, and \$_NAND_ cells

10.10 alumacc - extract ALU and MACC cells

```
yosys> help alumacc
```

```
alumacc [selection]
```

This pass translates arithmetic operations like \$add, \$mul, \$lt, etc. to \$alu and \$macc cells.

10.11 anlogic_eqn - Anlogic: Calculate equations for luts

```
yosys> help anlogic_eqn
```

```
anlogic_eqn [selection]
```

Calculate equations for luts since bitstream generator depends on it.

10.12 anlogic_fixcarry - Anlogic: fix carry chain

```
yosys> help anlogic_fixcarry
```

```
anlogic_fixcarry [options] [selection]
```

Add Anlogic adders to fix carry chain if needed.

10.13 assertpmux - adds asserts for parallel muxes

```
yosys> help assertpmux
```

```
assertpmux [options] [selection]
```

This command adds asserts to the design that assert that all parallel muxes (\$pmux cells) have a maximum of one of their inputs enable at any time.

```
-noinit
```

do not enforce the pmux condition during the init state

```
-always
```

usually the \$pmux condition is only checked when the \$pmux output is used by the mux tree it drives. this option will deactivate this additional constraint and check the \$pmux condition always.

10.14 async2sync - convert async FF inputs to sync circuits

```
yosys> help async2sync
```

```
async2sync [options] [selection]
```

This command replaces async FF inputs with sync circuits emulating the same behavior for when the async signals are actually synchronized to the clock.

This pass assumes negative hold time for the async FF inputs. For example when a reset deasserts with the clock edge, then the FF output will still drive the reset value in the next cycle regardless of the data-in value at the time of the clock edge.

```
-nolower
```

Do not automatically run 'chformal -lower' to lower \$check cells.

10.15 attrmap - renaming attributes

```
yosys> help attrmap
```

```
attrmap [options] [selection]
```

This command renames attributes and/or maps key/value pairs to other key/value pairs.

```
-tocase <name>
```

Match attribute names case-insensitively and set it to the specified name.

```
-rename <old_name> <new_name>
```

Rename attributes as specified

```
-map <old_name>=<old_value> <new_name>=<new_value>
```

Map key/value pairs as indicated.

```
-imap <old_name>=<old_value> <new_name>=<new_value>
```

Like -map, but use case-insensitive match for <old_value> when it is a string value.

```
-remove <name>=<value>
```

Remove attributes matching this pattern.

```
-modattr
```

Operate on module attributes instead of attributes on wires and cells.

For example, mapping Xilinx-style "keep" attributes to Yosys-style:

```
attrmap -tocase keep -imap keep="true" keep=1 \  
-imap keep="false" keep=0 -remove keep=0
```

10.16 attrmvcp - move or copy attributes from wires to driving cells

yosys> help attrmvcp

```
attrmvcp [options] [selection]
```

Move or copy attributes on wires to the cells driving them.

-copy

By default, attributes are moved. This will only add the attribute to the cell, without removing it from the wire.

-purge

If no selected cell consumes the attribute, then it is left on the wire by default. This option will cause the attribute to be removed from the wire, even if no selected cell takes it.

-driven

By default, attributes are moved to the cell driving the wire. With this option set it will be moved to the cell driven by the wire instead.

-attr <attrname>

Move or copy this attribute. This option can be used multiple times.

10.17 autaname - automatically assign names to objects

yosys> help autaname

```
autaname [selection]
```

Assign auto-generated public names to objects with private names (the ones with \$-prefix).

10.18 blackbox - convert modules into blackbox modules

yosys> help blackbox

```
blackbox [options] [selection]
```

Convert modules into blackbox modules (remove contents and set the blackbox module attribute).

10.19 bmuxmap - transform \$bmux cells to trees of \$mux cells

```
yosys> help bmuxmap
```

```
bmuxmap [selection]
```

This pass transforms \$bmux cells to trees of \$mux cells.

```
-pmux
    transform to $pmux instead of $mux cells.
```

10.20 booth - map \$mul cells to Booth multipliers

```
yosys> help booth
```

```
booth [selection]
```

This pass replaces multiplier cells with a radix-4 Booth-encoded implementation. It operates on \$mul cells whose width of operands is at least 4x4 and whose width of result is at least 8.

```
-lowpower
    use an alternative low-power architecture for the generated multiplier
    (signed multipliers only)
```

10.21 box_derive - derive box modules

```
yosys> help box_derive
```

```
box_derive [-base <base_module>] [-naming_attr <attr>] [selection]
```

As part of the assembly of the design hierarchy done by the 'hierarchy' command, specializations of parametric modules are derived on demand: for each choice of parameter values appearing in the design, a copy of the parametric module is derived which is specialized to that choice.

This derivation process ignores blackboxes and whiteboxes (boxes). To supplement, this 'box_derive' command can be used to request the derivation of modules based on box instances appearing in the design, which is desirable in certain use cases. Only the selected cells are considered as the instances that steer the derivation process.

```
-base <base_module>
    instead of deriving the module that directly corresponds to each box
    instance, derive a specialization of <base_module> (this option applies
    to all selected box cells)

-naming_attr <attr>
```

(continues on next page)

(continued from previous page)

once a specialization is derived, use the value of the module attribute `<attr>` for a name which should be used for the derived module (this replaces the internal Yosys naming scheme in which the names of derived modules start with '\$paramod\$')

10.22 bufnorm - (experimental) convert design into buffered-normalized form

```
yosys> help bufnorm
```

```
bufnorm [options] [selection]
```

Insert buffer cells into the design as needed, to make sure that each wire has exactly one driving cell port, and aliasing wires are buffered using buffer cells, than can be chained in a canonical order.

Running 'bufnorm' on the whole design enters 'buffered-normalized mode'.

```
-buf
```

Create \$buf cells for all buffers. The default is to use \$_BUF_ cells for sigle-bit buffers and \$buf cells only for multi-bit buffers.

```
-chain
```

Chain all alias wires. By default only wires with positive-valued 'chain' or 'keep' attribute on them are chained.

```
-output
```

Enable chaining of ouput ports wires.

```
-public
```

Enable chaining of wires wth public names.

```
-nochain
```

Disable chaining of wires with 'chain' attribute.

```
-nokeep
```

Disable chaining of wires with 'keep' attribute.

```
-flat
```

Alias for -nokeep and -nochain.

```
-nosticky
```

Disable 'sticky' behavior of output ports already driving whole wires, and always enforce canonical sort order instead.

```
-alphasort
```

Strictly use alphanumeric sort for chain-order. (Default is to chain 'keep' wires first, then ports in declaration order, and then the other wires in alphanumeric sort order.)

(continues on next page)

(continued from previous page)

Run 'bufnorm' with -pos, -bits, or -conn on the whole design to remove all \$buf buffer cells and exit 'buffered-normalized mode' again.

```
-pos
    Create (multi- and single-bit) $pos cells instead $buf and $_BUF_.

-bits
    Create arrays of $_BUF_ cells instead of multi-bit $buf cells.

-conn
    Create 'direct connections' instead of buffer cells.

-nomode
    Do not automatically enter or leave 'buffered-normalized mode'.
```

The 'bufnorm' command can also be used to just switch in and out of 'buffered-normalized mode' and run the low-level re-normalizer.

```
-update
    Enter 'buffered-normalized mode' and (re-)normalize.

-reset
    Leave 'buffered-normalized mode' without changing the netlist.
```

WARNING: THE 'bufnorm' COMMAND IS EXPERIMENTAL.

10.23 bugpoint - minimize testcases

yosys> help bugpoint

```
bugpoint [options] [-script <filename> | -command "<command>"]
```

This command minimizes the current design that is known to crash Yosys with the given script into a smaller testcase. It does this by removing an arbitrary part of the design and recursively invokes a new Yosys process with this modified design and the same script, repeating these steps while it can find a smaller design that still causes a crash. Once this command finishes, it replaces the current design with the smallest testcase it was able to produce.

In order to save the reduced testcase you must write this out to a file with another command after `bugpoint` like `write_rtlil` or `write_verilog`.

```
-script <filename> | -command "<command>"
    use this script file or command to crash Yosys. required.

-yosys <filename>
    use this Yosys binary. if not specified, `yosys` is used.

-grep "<string>"
```

(continues on next page)

(continued from previous page)

only consider crashes that place this string in the log file.

-fast

run ``proc_clean; clean -purge`` after each minimization step. converges faster, but produces larger testcases, and may fail to produce any testcase at all if the crash is related to dangling wires.

-clean

run ``proc_clean; clean -purge`` before checking testcase and after finishing. produces smaller and more useful testcases, but may fail to produce any testcase at all if the crash is related to dangling wires.

It is possible to constrain which parts of the design will be considered for removal. Unless one or more of the following options are specified, all parts will be considered.

-modules

try to remove modules. modules with a `(* bugpoint_keep *)` attribute will be skipped.

-ports

try to remove module ports. ports with a `(* bugpoint_keep *)` attribute will be skipped (useful for clocks, resets, etc.)

-cells

try to remove cells. cells with a `(* bugpoint_keep *)` attribute will be skipped.

-connections

try to reconnect ports to 'x'.

-processes

try to remove processes. processes with a `(* bugpoint_keep *)` attribute will be skipped.

-assigns

try to remove process assigns from cases.

-updates

try to remove process updates from syncs.

-runner "<prefix>"

child process wrapping command, e.g., `"timeout 30"`, or `valgrind`.

10.24 bwmuxmap - replace \$bwmux cells with equivalent logic

```
yosys> help bwmuxmap
```

```
bwmuxmap [options] [selection]
```

This pass replaces \$bwmux cells with equivalent logic

10.25 cd - a shortcut for 'select -module <name>'

```
yosys> help cd
```

```
cd <modname>
```

This is just a shortcut for 'select -module <modname>'.

```
cd <cellname>
```

When no module with the specified name is found, but there is a cell with the specified name in the current module, then this is equivalent to 'cd <celltype>'.

```
cd ..
```

Remove trailing substrings that start with '.' in current module name until the name of a module in the current design is generated, then switch to that module. Otherwise clear the current selection.

```
cd
```

This is just a shortcut for 'select -clear'.

10.26 cellmatch - match cells to their targets in cell library

```
yosys> help cellmatch
```

```
cellmatch -lib <design> [module selection]
```

This pass identifies functionally equivalent counterparts between each of the selected modules and a module from the secondary design <design>. For every such correspondence found, a techmap rule is generated for mapping instances of the former to instances of the latter. This techmap rule is saved in yet another design called '\$cellmatch', which is created if non-existent.

This pass restricts itself to combinational modules. Modules are functionally

(continues on next page)

(continued from previous page)

equivalent as long as their truth tables are identical upto a permutation of inputs and outputs. The supported number of inputs is limited to 6.

```
cellmatch -derive_luts [module selection]
```

For every port in each selected module, characterize its combinational function with a 'lut' attribute if possible.

10.27 check - check for obvious problems in the design

```
yosys> help check
```

```
check [options] [selection]
```

This pass identifies the following problems in the current design:

- combinatorial loops
- two or more conflicting drivers for one wire
- used wires that do not have a driver

Options:

```
-noinit
```

also check for wires which have the 'init' attribute set

```
-initdrv
```

also check for wires that have the 'init' attribute set and are not driven by an FF cell type

```
-mapped
```

also check for internal cells that have not been mapped to cells of the target architecture

```
-allow-tbuf
```

modify the -mapped behavior to still allow \$_TBUF_ cells

```
-assert
```

produce a runtime error if any problems are found in the current design

```
-force-detailed-loop-check
```

for the detection of combinatorial loops, use a detailed connectivity model for all internal cells for which it is available. This disables falling back to a simpler overapproximating model for those cells for which the detailed model is expected costly.

10.28 chformal - change formal constraints of the design

```
yosys> help chformal
```

```
chformal [types] [mode] [options] [selection]
```

Make changes to the formal constraints of the design. The [types] options the type of constraint to operate on. If none of the following options are given, the command will operate on all constraint types:

```
-assert      $assert cells, representing assert(...) constraints
-assume      $assume cells, representing assume(...) constraints
-live        $live cells, representing assert(s_eventually ...)
-fair        $fair cells, representing assume(s_eventually ...)
-cover       $cover cells, representing cover() statements
```

Additionally chformal will operate on \$check cells corresponding to the selected constraint types.

Exactly one of the following modes must be specified:

```
-remove
    remove the cells and thus constraints from the design

-early
    bypass FFs that only delay the activation of a constraint. When inputs
    of the bypassed FFs do not remain stable between clock edges, this may
    result in unexpected behavior.

-delay <N>
    delay activation of the constraint by <N> clock cycles

-skip <N>
    ignore activation of the constraint in the first <N> clock cycles

-coverenable
    add cover statements for the enable signals of the constraints

-assert2assume
-assume2assert
-live2fair
-fair2live
    change the roles of cells as indicated. these options can be combined

-lower
    convert each $check cell into an $assert, $assume, $live, $fair or
    $cover cell. If the $check cell contains a message, also produce a
    $print cell.
```

10.29 chparam - re-evaluate modules with new parameters

```
yosys> help chparam
```

```
chparam [ -set name value ]... [selection]
```

Re-evaluate the selected modules with new parameters. String values must be passed in double quotes (").

```
chparam -list [selection]
```

List the available parameters of the selected modules.

10.30 chtype - change type of cells in the design

```
yosys> help chtype
```

```
chtype [options] [selection]
```

Change the types of cells in the design.

```
-set <type>
```

set the cell type to the given type

```
-map <old_type> <new_type>
```

change cells types that match <old_type> to <new_type>

10.31 clean - remove unused cells and wires

```
yosys> help clean
```

```
clean [options] [selection]
```

This is identical to 'opt_clean', but less verbose.

When commands are separated using the ';' token, this command will be executed between the commands.

When commands are separated using the ';;;' token, this command will be executed in -purge mode between the commands.

10.32 clean_zerowidth - clean zero-width connections from the design

```
yosys> help clean_zerowidth
```

```
clean_zerowidth [selection]
```

Fixes the selected cells and processes to contain no zero-width connections. Depending on the cell type, this may be implemented by removing the connection, widening it to 1-bit, or removing the cell altogether.

10.33 clk2fflogic - convert clocked FFs to generic \$ff cells

```
yosys> help clk2fflogic
```

```
clk2fflogic [options] [selection]
```

This command replaces clocked flip-flops with generic \$ff cells that use the implicit global clock. This is useful for formal verification of designs with multiple clocks.

This pass assumes negative hold time for the async FF inputs. For example when a reset deasserts with the clock edge, then the FF output will still drive the reset value in the next cycle regardless of the data-in value at the time of the clock edge.

```
-nolower
```

Do not automatically run 'chformal -lower' to lower \$check cells.

```
-nopeepopt
```

Do not automatically run 'peepopt -formalclk' to rewrite clock patterns to more formal friendly forms.

10.34 clkbufmap - insert clock buffers on clock networks

```
yosys> help clkbufmap
```

```
clkbufmap [options] [selection]
```

Inserts clock buffers between nets connected to clock inputs and their drivers.

In the absence of any selection, all wires without the 'clkbuf_inhibit' attribute will be considered for clock buffer insertion.

Alternatively, to consider all wires without the 'buffer_type' attribute set to 'none' or 'bufr' one would specify:

```
'w:* a:buffer_type=none a:buffer_type=bufr %u %d'
```

as the selection.

```
-buf <celltype> <portname_out>:<portname_in>
```

(continues on next page)

(continued from previous page)

Specifies the cell type to use for the clock buffers and its port names. The first port will be connected to the clock network sinks, and the second will be connected to the actual clock source.

`-inpad <celltype> <portname_out>:<portname_in>`

If specified, a PAD cell of the given type is inserted on clock nets that are also top module's inputs (in addition to the clock buffer, if any).

At least one of `-buf` or `-inpad` should be specified.

10.35 clockgate - extract clock gating out of flip flops

yosys> help clockgate

`clockgate [options] [selection]`

This pass transforms each set of FFs sharing the same clock and enable signal into a clock-gating cell and a set of enable-less FFs. Primarily a power-saving transformation on ASIC designs.

`-pos <celltype> <ce>:<clk>:<gclk>`

If specified, rising-edge FFs will have CE inputs removed and a gated clock will be created by the user-specified `<celltype>` ICG (integrated clock gating) cell with ports named `<ce>`, `<clk>`, `<gclk>`. The ICG's clock enable pin must be active high.

`-neg <celltype> <ce>:<clk>:<gclk>`

If specified, falling-edge FFs will have CE inputs removed and a gated clock will be created by the user-specified `<celltype>` ICG (integrated clock gating) cell with ports named `<ce>`, `<clk>`, `<gclk>`. The ICG's clock enable pin must be active high.

`-liberty <filename>`

If specified, ICGs will be selected from the liberty files if available. Priority is given to cells with fewer `tie_lo` inputs and smaller size. This removes the need to manually specify `-pos` or `-neg` and `-tie_lo`.

`-dont_use <celltype>`

Cells `<celltype>` won't be considered when searching for ICGs in the liberty file specified by `-liberty`.

`-tie_lo <port_name>`

Port `<port_name>` of the ICG will be tied to zero. Intended for DFT scan-enable pins.

`-min_net_size <n>`

Only transform sets of at least `<n>` eligible FFs.

10.36 connect - create or remove connections

yosys> help connect

```
connect [-nomap] [-nounset] -set <lhs-expr> <rhs-expr> [selection]
```

Create a connection. This is equivalent to adding the statement 'assign <lhs-expr> = <rhs-expr>;' to the Verilog input. Per default, all existing drivers for <lhs-expr> are unconnected. This can be overwritten by using the -nounset option.

```
connect [-nomap] -unset <expr> [selection]
```

Unconnect all existing drivers for the specified expression.

```
connect [-nomap] [-assert] -port <cell> <port> <expr> [selection]
```

Connect the specified cell port to the specified cell port.

Per default signal alias names are resolved and all signal names are mapped the the signal name of the primary driver. Using the -nomap option deactivates this behavior.

The connect command operates in one module only. Either only one module must be selected or an active module must be set using the 'cd' command.

The -assert option verifies that the connection already exists, instead of making it.

This command does not operate on module with processes.

10.37 connect_rpc - connect to RPC frontend

yosys> help connect_rpc

```
connect_rpc -exec <command> [args...]  
connect_rpc -path <path>
```

Load modules using an out-of-process frontend.

```
-exec <command> [args...]  
    run <command> with arguments [args...]. send requests on stdin, read  
    responses from stdout.
```

```
-path <path>  
    connect to Unix domain socket at <path>. (Unix)  
    connect to bidirectional byte-type named pipe at <path>. (Windows)
```

(continues on next page)

(continued from previous page)

A simple JSON-based, newline-delimited protocol is used for communicating with the frontend. Yosys requests data from the frontend by sending exactly 1 line of JSON. Frontend responds with data or error message by replying with exactly 1 line of JSON as well.

```
-> {"method": "modules"}
<- {"modules": ["<module-name>", ...]}
<- {"error": "<error-message>"}
    request for the list of modules that can be derived by this frontend.
    the 'hierarchy' command will call back into this frontend if a cell
    with type <module-name> is instantiated in the design.

-> {"method": "derive", "module": "<module-name>", "parameters": {
    "<param-name>": {"type": "[unsigned|signed|string|real]",
                    "value": "<param-value>"}, ...}}
<- {"frontend": "[rtlil|verilog|...]", "source": "<source>"}
<- {"error": "<error-message>"}
    request for the module <module-name> to be derived for a specific set of
    parameters. <param-name> starts with \ for named parameters, and with $
    for unnamed parameters, which are numbered starting at 1. <param-value>
    for integer parameters is always specified as a binary string of
    unlimited precision. the <source> returned by the frontend is
    hygienically parsed by a built-in Yosys <frontend>, allowing the RPC
    frontend to return any convenient representation of the module. the
    derived module is cached, so the response should be the same whenever the
    same set of parameters is provided.
```

10.38 connwrappers - match width of input-output port pairs

yosys> help connwrappers

```
connwrappers [options] [selection]
```

Wrappers are used in coarse-grain synthesis to wrap cells with smaller ports in wrapper cells with a (larger) constant port size. I.e. the upper bits of the wrapper output are signed/unsigned bit extended. This command uses this knowledge to rewire the inputs of the driven cells to match the output of the driving cell.

```
-signed <cell_type> <port_name> <width_param>
-unsigned <cell_type> <port_name> <width_param>
    consider the specified signed/unsigned wrapper output

-port <cell_type> <port_name> <width_param> <sign_param>
    use the specified parameter to decide if signed or unsigned
```

The options -signed, -unsigned, and -port can be specified multiple times.

10.39 coolrunner2_fixup - insert necessary buffer cells for CoolRunner-II architecture

```
yosys> help coolrunner2_fixup
```

```
coolrunner2_fixup [options] [selection]
```

Insert necessary buffer cells for CoolRunner-II architecture.

10.40 coolrunner2_sop - break \$sop cells into ANDTERM/ORTERM cells

```
yosys> help coolrunner2_sop
```

```
coolrunner2_sop [options] [selection]
```

Break \$sop cells into ANDTERM/ORTERM cells.

10.41 copy - copy modules in the design

```
yosys> help copy
```

```
copy old_name new_name
```

Copy the specified module. Note that selection patterns are not supported by this command.

10.42 cover - print code coverage counters

```
yosys> help cover
```

```
cover [options] [pattern]
```

Print the code coverage counters collected using the cover() macro in the Yosys C++ code. This is useful to figure out what parts of Yosys are utilized by a test bench.

```
-q
    Do not print output to the normal destination (console and/or log file)

-o file
    Write output to this file, truncate if exists.

-a file
```

(continues on next page)

(continued from previous page)

Write output to this file, append if exists.

`-d dir`

Write output to a newly created file in the specified directory.

When one or more pattern (shell wildcards) are specified, then only counters matching at least one pattern are printed.

It is also possible to instruct Yosys to print the coverage counters on program exit to a file using environment variables:

```
YOSYS_COVER_DIR="{dir-name}" yosys {args}
```

This will create a file (with an auto-generated name) in this directory and write the coverage counters to it.

```
YOSYS_COVER_FILE="{file-name}" yosys {args}
```

This will append the coverage counters to the specified file.

Hint: Use the following AWK command to consolidate Yosys coverage files:

```
gawk '{ p[$3] = $1; c[$3] += $2; } END { for (i in p)
  printf "%-60s %10d %s\n", p[i], c[i], i; }' {files} | sort -k3
```

Coverage counters are only available in Yosys for Linux.

10.43 cutpoint - adds formal cut points to the design

yosys> help cutpoint

```
cutpoint [options] [selection]
```

This command adds formal cut points to the design.

`-undef`

set cutpoint nets to undef (x). the default behavior is to create a \$anyseq cell and drive the cutpoint net from that

10.44 debug - run command with debug log messages enabled

```
yosys> help debug
```

```
debug cmd
```

Execute the specified command with debug log messages enabled

10.45 delete - delete objects in the design

```
yosys> help delete
```

```
delete [selection]
```

Deletes the selected objects. This will also remove entire modules, if the whole module is selected.

```
delete {-input|-output|-port} [selection]
```

Does not delete any object but removes the input and/or output flag on the selected wires, thus 'deleting' module ports.

10.46 deminout - demote inout ports to input or output

```
yosys> help deminout
```

```
deminout [options] [selection]
```

"Demote" inout ports to input or output ports, if possible.

10.47 demuxmap - transform \$demux cells to \$eq + \$mux cells

```
yosys> help demuxmap
```

```
demuxmap [selection]
```

This pass transforms \$demux cells to a bunch of equality comparisons.

10.48 design - save, restore and reset current design

yosys> help design

```
design -reset
```

Clear the current design.

```
design -save <name>
```

Save the current design under the given name.

```
design -stash <name>
```

Save the current design under the given name and then clear the current design.

```
design -push
```

Push the current design to the stack and then clear the current design.

```
design -push-copy
```

Push the current design to the stack without clearing the current design.

```
design -pop
```

Reset the current design and pop the last design from the stack.

```
design -load <name>
```

Reset the current design and load the design previously saved under the given name.

```
design -copy-from <name> [-as <new_mod_name>] <selection>
```

Copy modules from the specified design into the current one. The selection is evaluated in the other design.

```
design -copy-to <name> [-as <new_mod_name>] [selection]
```

Copy modules from the current design into the specified one.

```
design -import <name> [-as <new_top_name>] [selection]
```

(continues on next page)

(continued from previous page)

Import the specified design into the current design. The source design must either have a selected top module or the selection must contain exactly one module that is then used as top module for this command.

```
design -reset-vlog
```

The Verilog front-end remembers defined macros and top-level declarations between calls to 'read_verilog'. This command resets this memory.

```
design -delete <name>
```

Delete the design previously saved under the given name.

10.49 dffinit - set INIT param on FF cells

```
yosys> help dffinit
```

```
dffinit [options] [selection]
```

This pass sets an FF cell parameter to the the initial value of the net it drives. (This is primarily used in FPGA flows.)

```
-ff <cell_name> <output_port> <init_param>
```

operate on the specified cell type. this option can be used multiple times.

```
-highlow
```

use the string values "high" and "low" to represent a single-bit initial value of 1 or 0. (multi-bit values are not supported in this mode.)

```
-strinit <string for high> <string for low>
```

use string values in the command line to represent a single-bit initial value of 1 or 0. (multi-bit values are not supported in this mode.)

```
-noreinit
```

fail if the FF cell has already a defined initial value set in other passes and the initial value of the net it drives is not equal to the already defined initial value.

10.50 dfflegalize - convert FFs to types supported by the target

yosys> help dfflegalize

```
dfflegalize [options] [selection]
```

Converts FFs to types supported by the target.

-cell <cell_type_pattern> <init_values>

specifies a supported group of FF cells. <cell_type_pattern> is a yosys internal fine cell name, where ? characters can be as a wildcard matching any character. <init_values> specifies which initialization values these FF cells can support, and can be one of:

- x (no init value supported)
- 0
- 1
- r (init value has to match reset value, only for some FF types)
- 01 (both 0 and 1 supported).

-mince <num>

specifies a minimum number of FFs that should be using any given clock enable signal. If a clock enable signal doesn't meet this threshold, it is unmapped into soft logic.

-minsrst <num>

specifies a minimum number of FFs that should be using any given sync set/reset signal. If a sync set/reset signal doesn't meet this threshold, it is unmapped into soft logic.

The following cells are supported by this pass (ie. will be ingested, and can be specified as allowed targets):

- \$_DFF_[NP]_
- \$_DFFE_[NP][NP]_
- \$_DFF_[NP][NP][01]_
- \$_DFFE_[NP][NP][01][NP]_
- \$_ALDFF_[NP][NP]_
- \$_ALDFFE_[NP][NP][NP]_
- \$_DFFSR_[NP][NP][NP]_
- \$_DFFSRE_[NP][NP][NP][NP]_
- \$_SDFF_[NP][NP][01]_
- \$_SDFFE_[NP][NP][01][NP]_
- \$_SDFFCE_[NP][NP][01][NP]_
- \$_SR_[NP][NP]_
- \$_DLATCH_[NP]_
- \$_DLATCH_[NP][NP][01]_
- \$_DLATCHSR_[NP][NP][NP]_

The following transformations are performed by this pass:

(continues on next page)

(continued from previous page)

- upconversion from a less capable cell to a more capable cell, if the less capable cell is not supported (eg. dff -> dffe, or adff -> dffsr)
- unmapping FFs with clock enable (due to unsupported cell type or -mince)
- unmapping FFs with sync reset (due to unsupported cell type or -minrst)
- adding inverters on the control pins (due to unsupported polarity)
- adding inverters on the D and Q pins and inverting the init/reset values (due to unsupported init or reset value)
- converting sr into adlatch (by tying D to 1 and using E as set input)
- emulating unsupported dffsr cell by adff + adff + sr + mux
- emulating unsupported dlatchsr cell by adlatch + adlatch + sr + mux
- emulating adff when the (reset, init) value combination is unsupported by dff + adff + dlatch + mux
- emulating adlatch when the (reset, init) value combination is unsupported by dlatch + adlatch + dlatch + mux

If the pass is unable to realize a given cell type (eg. adff when only plain dff is available), an error is raised.

10.51 dfflibmap - technology mapping of flip-flops

```
yosys> help dfflibmap
```

```
    dfflibmap [-prepare] [-map-only] [-info] [-dont_use <cell_name>] -liberty <file> ↵
    ↵[selection]
```

Map internal flip-flop cells to the flip-flop cells in the technology library specified in the given liberty files.

This pass may add inverters as needed. Therefore it is recommended to first run this pass and then map the logic paths to the target technology.

When called with -prepare, this command will convert the internal FF cells to the internal cell types that best match the cells found in the given liberty file, but won't actually map them to the target cells.

When called with -map-only, this command will only map internal cell types that are already of exactly the right type to match the target cells, leaving remaining internal cells untouched.

When called with -info, this command will only print the target cell list, along with their associated internal cell types, and the arguments that would be passed to the dfflegalize pass. The design will not be changed.

When called with -dont_use, this command will not map to the specified cell name as an alternative to setting the dont_use property in the Liberty file. This argument can be called multiple times with different cell names. This argument also supports simple glob patterns in the cell name.

10.52 dffunmap - unmap clock enable and synchronous reset from FFs

yosys> help dffunmap

```
dffunmap [options] [selection]
```

This pass transforms FF types with clock enable and/or synchronous reset into their base type (with neither clock enable nor sync reset) by emulating the clock enable and synchronous reset with multiplexers on the cell input.

-ce-only

unmap only clock enables, leave synchronous resets alone.

-srst-only

unmap only synchronous resets, leave clock enables alone.

10.53 dft_tag - create tagging logic for data flow tracking

yosys> help dft_tag

```
dft_tag [options] [selection]
```

This pass... TODO

-overwrite-only

Only process \$overwrite_tag and \$original_tag cells.

-tag-public

For each public wire that may carry tagged data, create a new public wire (named <wirename>:<tagname>) that carries the tag bits. Note that without this, tagging logic will only be emitted as required for uses of \$get_tag.

10.54 dump - print parts of the design in RTLIL format

yosys> help dump

```
dump [options] [selection]
```

Write the selected parts of the design to the console or specified file in RTLIL format.

-m

also dump the module headers, even if only parts of a single module is selected

-n

only dump the module headers if the entire module is selected

(continues on next page)

(continued from previous page)

```
-o <filename>
    write to the specified file.

-a <filename>
    like -outfile but append instead of overwrite
```

10.55 echo - turning echoing back of commands on and off

```
yosys> help echo
```

```
    echo on

Print all commands to log before executing them.

    echo off

Do not print all commands to log before executing them. (default)
```

10.56 edgetypes - list all types of edges in selection

```
yosys> help edgetypes
```

```
    edgetypes [options] [selection]

This command lists all unique types of 'edges' found in the selection. An 'edge'
is a 4-tuple of source and sink cell type and port name.
```

10.57 efinix_fixcarry - Efinix: fix carry chain

```
yosys> help efinix_fixcarry
```

```
    efinix_fixcarry [options] [selection]

Add Efinix adders to fix carry chain if needed.
```

10.58 equiv_add - add a \$equiv cell

yosys> help equiv_add

```
equiv_add [-try] gold_sig gate_sig
```

This command adds an \$equiv cell for the specified signals.

```
equiv_add [-try] -cell gold_cell gate_cell
```

This command adds \$equiv cells for the ports of the specified cells.

10.59 equiv_induct - proving \$equiv cells using temporal induction

yosys> help equiv_induct

```
equiv_induct [options] [selection]
```

Uses a version of temporal induction to prove \$equiv cells.

Only selected \$equiv cells are proven and only selected cells are used to perform the proof.

```
-undef
    enable modelling of undef states
```

```
-seq <N>
    the max. number of time steps to be considered (default = 4)
```

This command is very effective in proving complex sequential circuits, when the internal state of the circuit quickly propagates to \$equiv cells.

However, this command uses a weak definition of 'equivalence': This command proves that the two circuits will not diverge after they produce equal outputs (observable points via \$equiv) for at least <N> cycles (the <N> specified via -seq).

Combined with simulation this is very powerful because simulation can give you confidence that the circuits start out synced for at least <N> cycles after reset.

10.60 equiv_make - prepare a circuit for equivalence checking

```
yosys> help equiv_make
```

```
equiv_make [options] gold_module gate_module equiv_module
```

This creates a module annotated with \$equiv cells from two presumably equivalent modules. Use commands such as 'equiv_simple' and 'equiv_status' to work with the created equivalent checking module.

-inames

Also match cells and wires with \$... names.

-blacklist <file>

Do not match cells or signals that match the names in the file.

-encfile <file>

Match FSM encodings using the description from the file.

See 'help fsm_recode' for details.

-make_assert

Check equivalence with \$assert cells instead of \$equiv.

\$eqx (==) is used to compare signals.

Note: The circuit created by this command is not a miter (with something like a trigger output), but instead uses \$equiv cells to encode the equivalence checking problem. Use 'miter -equiv' if you want to create a miter circuit.

10.61 equiv_mark - mark equivalence checking regions

```
yosys> help equiv_mark
```

```
equiv_mark [options] [selection]
```

This command marks the regions in an equivalence checking module. Region 0 is the proven part of the circuit. Regions with higher numbers are connected unproven subcircuits. The integer attribute 'equiv_region' is set on all wires and cells.

10.62 equiv_miter - extract miter from equiv circuit

```
yosys> help equiv_miter
```

```
equiv_miter [options] miter_module [selection]
```

This creates a miter module for further analysis of the selected \$equiv cells.

-trigger

Create a trigger output

(continues on next page)

(continued from previous page)

```
-cmp
    Create cmp_* outputs for individual unproven $equiv cells

-assert
    Create a $assert cell for each unproven $equiv cell

-undef
    Create compare logic that handles undefs correctly
```

10.63 equiv_opt - prove equivalence for optimized circuit

```
yosys> help equiv_opt
```

```
equiv_opt [options] [command]
```

This command uses temporal induction to check circuit equivalence before and after an optimization pass.

```
-run <from_label>:<to_label>
    only run the commands between the labels (see below). an empty
    from label is synonymous to the start of the command list, and empty to
    label is synonymous to the end of the command list.

-map <filename>
    expand the modules in this file before proving equivalence. this is
    useful for handling architecture-specific primitives.

-blacklist <file>
    Do not match cells or signals that match the names in the file
    (passed to equiv_make).

-assert
    produce an error if the circuits are not equivalent.

-multiclock
    run clk2fflogic before equivalence checking.

-async2sync
    run async2sync before equivalence checking.

-undef
    enable modelling of undef states during equiv_induct.

-nocheck
    disable running check before and after the command under test.
```

The following commands are executed by this verification command:

```
run_pass:
```

(continues on next page)

(continued from previous page)

```
hierarchy -auto-top
design -save preopt
check -assert      (unless -nocheck)
[command]
check -assert      (unless -nocheck)
design -stash postopt

prepare:
  design -copy-from preopt  -as gold A:top
  design -copy-from postopt -as gate A:top

techmap:      (only with -map)
  techmap -wb -D EQUIV -autoproc -map <filename> ...

prove:
  clk2fflogic      (only with -multiclock)
  async2sync       (only with -async2sync)
  equiv_make -blacklist <filename> ... gold gate equiv
  equiv_induct [-undef] equiv
  equiv_status [-assert] equiv

restore:
  design -load preopt
```

10.64 equiv_purge - purge equivalence checking module

```
yosys> help equiv_purge
```

```
equiv_purge [options] [selection]
```

This command removes the proven part of an equivalence checking module, leaving only the unproven segments in the design. This will also remove and add module ports as needed.

10.65 equiv_remove - remove \$equiv cells

```
yosys> help equiv_remove
```

```
equiv_remove [options] [selection]
```

This command removes the selected \$equiv cells. If neither -gold nor -gate is used then only proven cells are removed.

```
-gold
  keep gold circuit
```

```
-gate
  keep gate circuit
```


10.66 equiv_simple - try proving simple \$equiv instances

yosys> help equiv_simple

```
equiv_simple [options] [selection]
```

This command tries to prove \$equiv cells using a simple direct SAT approach.

```
-v
    verbose output

-undef
    enable modelling of undef states

-short
    create shorter input cones that stop at shared nodes. This yields
    simpler SAT problems but sometimes fails to prove equivalence.

-nogroup
    disabling grouping of $equiv cells by output wire

-seq <N>
    the max. number of time steps to be considered (default = 1)
```

10.67 equiv_status - print status of equivalent checking module

yosys> help equiv_status

```
equiv_status [options] [selection]
```

This command prints status information for all selected \$equiv cells.

```
-assert
    produce an error if any unproven $equiv cell is found
```

10.68 equiv_struct - structural equivalence checking

yosys> help equiv_struct

```
equiv_struct [options] [selection]
```

This command adds additional \$equiv cells based on the assumption that the gold and gate circuit are structurally equivalent. Note that this can introduce bad \$equiv cells in cases where the netlists are not structurally equivalent, for example when analyzing circuits with cells with commutative inputs. This command will also de-duplicate gates.

```
-fwd
```

(continues on next page)

(continued from previous page)

by default this command performs forward sweeps until nothing can be merged by forward sweeps, then backward sweeps until forward sweeps are effective again. with this option set only forward sweeps are performed.

`-fwnonly <cell_type>`

add the specified cell type to the list of cell types that are only merged in forward sweeps and never in backward sweeps. \$equiv is in this list automatically.

`-icells`

by default, the internal RTL and gate cell types are ignored. add this option to also process those cell types with this command.

`-maxiter <N>`

maximum number of iterations to run before aborting

10.69 eval - evaluate the circuit given an input

yosys> help eval

```
eval [options] [selection]
```

This command evaluates the value of a signal given the value of all required inputs.

`-set <signal> <value>`

set the specified signal to the specified value.

`-set-undef`

set all unspecified source signals to undef (x)

`-table <signal>`

create a truth table using the specified input signals

`-show <signal>`

show the value for the specified signal. if no -show option is passed then all output ports of the current module are used.

10.70 example_dt - drivertools example

yosys> help example_dt

TODO: add help message

10.71 exec - execute commands in the operating system shell

yosys> help exec

```
exec [options] -- [command]
```

Execute a command in the operating system shell. All supplied arguments are concatenated and passed as a command to `popen(3)`. Whitespace is not guaranteed to be preserved, even if quoted. `stdin` and `stderr` are not connected, while `stdout` is logged unless the `"-q"` option is specified.

```
-q
  Suppress stdout and stderr from subprocess

-expect-return <int>
  Generate an error if popen() does not return specified value.
  May only be specified once; the final specified value is controlling
  if specified multiple times.

-expect-stdout <regex>
  Generate an error if the specified regex does not match any line
  in subprocess's stdout. May be specified multiple times.

-not-expect-stdout <regex>
  Generate an error if the specified regex matches any line
  in subprocess's stdout. May be specified multiple times.
```

Example: `exec -q -expect-return 0 -- echo "bananapie" | grep "nana"`

10.72 expose - convert internal signals to module ports

yosys> help expose

```
expose [options] [selection]
```

This command exposes all selected internal signals of a module as additional outputs.

```
-dff
  only consider wires that are directly driven by register cell.

-cut
  when exposing a wire, create an input/output pair and cut the internal
  signal path at that wire.

-input
  when exposing a wire, create an input port and disconnect the internal
  driver.
```

(continues on next page)

(continued from previous page)

```
-shared
    only expose those signals that are shared among the selected modules.
    this is useful for preparing modules for equivalence checking.

-evert
    also turn connections to instances of other modules to additional
    inputs and outputs and remove the module instances.

-evert-dff
    turn flip-flops to sets of inputs and outputs.

-sep <separator>
    when creating new wire/port names, the original object name is suffixed
    with this separator (default: '.') and the port name or a type
    designator for the exposed signal.
```

10.73 extract - find subcircuits and replace them with cells

```
yosys> help extract
```

```
extract -map <map_file> [options] [selection]
extract -mine <out_file> [options] [selection]
```

This pass looks for subcircuits that are isomorphic to any of the modules in the given map file and replaces them with instances of this modules. The map file can be a Verilog source file (*.v) or an RTLIL source file (*.il).

```
-map <map_file>
    use the modules in this file as reference. This option can be used
    multiple times.

-map %<design-name>
    use the modules in this in-memory design as reference. This option can
    be used multiple times.

-verbose
    print debug output while analyzing

-constports
    also find instances with constant drivers. this may be much
    slower than the normal operation.

-nodefaultswaps
    normally builtin port swapping rules for internal cells are used per
    default. This turns that off, so e.g. 'a^b' does not match 'b^a'
    when this option is used.

-compatible <needle_type> <haystack_type>
    Per default, the cells in the map file (needle) must have the
```

(continues on next page)

(continued from previous page)

type as the cells in the active design (haystack). This option can be used to register additional pairs of types that should match. This option can be used multiple times.

`-swap <needle_type> <port1>,<port2>[,...]`

Register a set of swappable ports for a needle cell type. This option can be used multiple times.

`-perm <needle_type> <port1>,<port2>[,...] <portA>,<portB>[,...]`

Register a valid permutation of swappable ports for a needle cell type. This option can be used multiple times.

`-cell_attr <attribute_name>`

Attributes on cells with the given name must match.

`-wire_attr <attribute_name>`

Attributes on wires with the given name must match.

`-ignore_parameters`

Do not use parameters when matching cells.

`-ignore_param <cell_type> <parameter_name>`

Do not use this parameter when matching cells.

This pass does not operate on modules with unprocessed processes in it. (I.e. the 'proc' pass should be used first to convert processes to netlists.)

This pass can also be used for mining for frequent subcircuits. In this mode the following options are to be used instead of the `-map` option.

`-mine <out_file>`

mine for frequent subcircuits and write them to the given RTLIL file

`-mine_cells_span <min> <max>`

only mine for subcircuits with the specified number of cells
default value: 3 5

`-mine_min_freq <num>`

only mine for subcircuits with at least the specified number of matches
default value: 10

`-mine_limit_matches_per_module <num>`

when calculating the number of matches for a subcircuit, don't count more than the specified number of matches per module

`-mine_max_fanout <num>`

don't consider internal signals with more than <num> connections

The modules in the map file may have the attribute 'extract_order' set to an integer value. Then this value is used to determine the order in which the pass tries to map the modules to the design (ascending, default value is 0).

(continues on next page)

(continued from previous page)

See 'help techmap' for a pass that does the opposite thing.

10.74 extract_counter - Extract GreenPak4 counter cells

yosys> help extract_counter

```
extract_counter [options] [selection]
```

This pass converts non-resettable or async resettable counters to counter cells. Use a target-specific 'techmap' map file to convert those cells to the actual target cells.

-maxwidth N

Only extract counters up to N bits wide (default 64)

-minwidth N

Only extract counters at least N bits wide (default 2)

-allow_arst yes|no

Allow counters to have async reset (default yes)

-dir up|down|both

Look for up-counters, down-counters, or both (default down)

-pout X,Y,...

Only allow parallel output from the counter to the listed cell types (if not specified, parallel outputs are not restricted)

10.75 extract_fa - find and extract full/half adders

yosys> help extract_fa

```
extract_fa [options] [selection]
```

This pass extracts full/half adders from a gate-level design.

-fa, -ha

Enable cell types (fa=full adder, ha=half adder)

All types are enabled if none of this options is used

-d <int>

Set maximum depth for extracted logic cones (default=20)

-b <int>

Set maximum breadth for extracted logic cones (default=6)

-v

Verbose output

10.76 extract_reduce - converts gate chains into \$reduce_* cells

```
yosys> help extract_reduce
```

```
extract_reduce [options] [selection]
```

converts gate chains into \$reduce_* cells

This command finds chains of \$AND_, \$OR_, and \$XOR_ cells and replaces them with their corresponding \$reduce_* cells. Because this command only operates on these cell types, it is recommended to map the design to only these cell types using the `abc -g` command. Note that, in some cases, it may be more effective to map the design to only \$AND_ cells, run extract_reduce, map the remaining parts of the design to AND/OR/XOR cells, and run extract_reduce a second time.

```
-allow-off-chain
```

Allows matching of cells that have loads outside the chain. These cells will be replicated and folded into the \$reduce_* cell, but the original cell will remain, driving its original loads.

10.77 extractinv - extract explicit inverter cells for invertible cell pins

```
yosys> help extractinv
```

```
extractinv [options] [selection]
```

Searches the design for all cells with invertible pins controlled by a cell parameter (eg. IS_CLK_INVERTED on many Xilinx cells) and removes the parameter. If the parameter was set to 1, inserts an explicit inverter cell in front of the pin instead. Normally used for output to ISE, which does not support the inversion parameters.

To mark a cell port as invertible, use (* invertible_pin = "param_name" *) on the wire in the blackbox module. The parameter value should have the same width as the port, and will be effectively XORed with it.

```
-inv <celltype> <portname_out>:<portname_in>
```

Specifies the cell type to use for the inverters and its port names. This option is required.

10.78 flatten - flatten design

```
yosys> help flatten
```

```
flatten [options] [selection]
```

This pass flattens the design by replacing cells by their implementation. This pass is very similar to the 'techmap' pass. The only difference is that this

(continues on next page)

(continued from previous page)

pass is using the current design as mapping library.

Cells and/or modules with the 'keep_hierarchy' attribute set will not be flattened by this command.

- wb
Ignore the 'whitebox' attribute on cell implementations.
- noscopeinfo
Do not create '\$scopeinfo' cells that preserve attributes of cells and modules that were removed during flattening. With this option, the 'src' attribute of a given cell is merged into all objects replacing that cell, with multiple distinct 'src' locations separated by '|'. Without this option these 'src' locations can be found via the cell_src' and 'module_src' attribute of '\$scopeinfo' cells.
- scopename
Create 'scopename' attributes for objects with a private name. This attribute records the 'hdlname' of the enclosing scope. For objects with a public name the enclosing scope can be found via their 'hdlname' attribute.
- separator <char>
Use this separator char instead of '.' when concatenating design levels.

10.79 flowmap - pack LUTs with FlowMap

yosys> help flowmap

```
flowmap [options] [selection]
```

This pass uses the FlowMap technology mapping algorithm to pack logic gates into k-LUTs with optimal depth. It allows mapping any circuit elements that can be evaluated with the `eval` pass, including cells with multiple output ports and multi-bit input and output ports.

- maxlut k
perform technology mapping for a k-LUT architecture. if not specified, defaults to 3.
- minlut n
only produce n-input or larger LUTs. if not specified, defaults to 1.
- cells <cell>[,<cell>,...]
map specified cells. if not specified, maps \$_NOT_, \$_AND_, \$_OR_, \$_XOR_ and \$_MUX_, which are the outputs of the `simplemap` pass.
- relax
perform depth relaxation and area minimization.

(continues on next page)

(continued from previous page)

```

-r-alpha n, -r-beta n, -r-gamma n
    parameters of depth relaxation heuristic potential function.
    if not specified, alpha=8, beta=2, gamma=1.

-optarea n
    optimize for area by trading off at most n logic levels for fewer LUTs.
    n may be zero, to optimize for area without increasing depth.
    implies -relax.

-debug
    dump intermediate graphs.

-debug-relax
    explain decisions performed during depth relaxation.

```

10.80 fmcombine - combine two instances of a cell into one

yosys> help fmcombine

```
fmcombine [options] module_name gold_cell gate_cell
```

This pass takes two cells, which are instances of the same module, and replaces them with one instance of a special 'combined' module, that effectively contains two copies of the original module, plus some formal properties.

This is useful for formal test benches that check what differences in behavior a slight difference in input causes in a module.

```

-initeq
    Insert assumptions that initially all FFs in both circuits have the
    same initial values.

-anyeq
    Do not duplicate $anyseq/$anyconst cells.

-fwd
    Insert forward hint assumptions into the combined module.

-bwd
    Insert backward hint assumptions into the combined module.
    (Backward hints are logically equivalent to forward hits, but
    some solvers are faster with bwd hints, or even both -bwd and -fwd.)

-nop
    Don't insert hint assumptions into the combined module.
    (This should not provide any speedup over the original design, but
    strangely sometimes it does.)

```

If none of -fwd, -bwd, and -nop is given, then -fwd is used as default.

10.81 fminit - set init values/sequences for formal

```
yosys> help fminit
```

```
fminit [options] <selection>
```

This pass creates init constraints (for example for reset sequences) in a formal model.

```
-seq <signal> <sequence>
```

Set sequence using comma-separated list of values, use 'z for unconstrained bits. The last value is used for the remainder of the trace.

```
-set <signal> <value>
```

Add constant value constraint

```
-posedge <signal>
```

```
-negedge <signal>
```

Set clock for init sequences

10.82 formalff - prepare FFs for formal

```
yosys> help formalff
```

```
formalff [options] [selection]
```

This pass transforms clocked flip-flops to prepare a design for formal verification. If a design contains latches and/or multiple different clocks run the `async2sync` or `clk2fflogic` passes before using this pass.

```
-clk2ff
```

Replace all clocked flip-flops with \$ff cells that use the implicit global clock. This assumes, without checking, that the design uses a single global clock. If that is not the case, the `clk2fflogic` pass should be used instead.

```
-ff2anyinit
```

Replace uninitialized bits of \$ff cells with \$anyinit cells. An \$anyinit cell behaves exactly like an \$ff cell with an undefined initialization value. The difference is that \$anyinit inhibits don't-care optimizations and is used to track solver-provided values in witness traces.

If combined with `-clk2ff` this also affects newly created \$ff cells.

```
-anyinit2ff
```

Replaces \$anyinit cells with uninitialized \$ff cells. This performs the reverse of `-ff2anyinit` and can be used, before running a backend pass (or similar) that is not yet aware of \$anyinit cells.

(continues on next page)

(continued from previous page)

Note that after running `-anyinit2ff`, in general, performing don't-care optimizations is not sound in a formal verification setting.

-fine

Emit fine-grained `$_FF_` cells instead of coarse-grained `$ff` cells for `-anyinit2ff`. Cannot be combined with `-clk2ff` or `-ff2anyinit`.

-setundef

Find FFs with undefined initialization values for which changing the initialization does not change the observable behavior and initialize them. For `-ff2anyinit`, this reduces the number of generated `$anyinit` cells that drive wires with private names.

-hierarchy

Propagates the `'replaced_by_gclk'` attribute set by `clk2ff` upwards through the design hierarchy towards the toplevel inputs. This option works on the whole design and ignores the selection.

-assume

Add assumptions that constrain wires with the `'replaced_by_gclk'` attribute to the value they would have before an active clock edge.

10.83 `freduce` - perform functional reduction

yosys> help `freduce`

```
freduce [options] [selection]
```

This pass performs functional reduction in the circuit. I.e. if two nodes are equivalent, they are merged to one node and one of the redundant drivers is disconnected. A subsequent call to `'clean'` will remove the redundant drivers.

-v, -vv

enable verbose or very verbose output

-inv

enable explicit handling of inverted signals

-stop <n>

stop after <n> reduction operations. this is mostly used for debugging the `freduce` command itself.

-dump <prefix>

dump the design to `<prefix>_<module>_<num>.il` after each reduction operation. this is mostly used for debugging the `freduce` command.

This pass is undef-aware, i.e. it considers don't-care values for detecting equivalent nodes.

(continues on next page)

(continued from previous page)

All selected wires are considered for rewiring. The selected cells cover the circuit that is analyzed.

10.84 fsm - extract and optimize finite state machines

yosys> help fsm

```
fsm [options] [selection]
```

This pass calls all the other fsm_* passes in a useful order. This performs FSM extraction and optimization. It also calls opt_clean as needed:

```
fsm_detect          unless got option -nodetect
fsm_extract
```

```
fsm_opt
opt_clean
fsm_opt
```

```
fsm_expand          if got option -expand
opt_clean            if got option -expand
fsm_opt              if got option -expand
```

```
fsm_recode          unless got option -norecode
```

```
fsm_info
```

```
fsm_export          if got option -export
fsm_map              unless got option -nomap
```

Options:

```
-expand, -norecode, -export, -nomap
    enable or disable passes as indicated above
```

```
-fullexpand
    call expand with -full option
```

```
-encoding type
-fm_set_fsm_file file
-encfile file
    passed through to fsm_recode pass
```

This pass uses a subset of FF types to detect FSMs. Run 'opt -nosdff -nodffe' before this pass to prepare the design.

10.85 fsm_detect - finding FSMs in design

```
yosys> help fsm_detect
```

```
fsm_detect [options] [selection]
```

This pass detects finite state machines by identifying the state signal. The state signal is then marked by setting the attribute 'fsm_encoding' on the state signal to "auto".

```
-ignore-self-reset
    Mark FSMs even if they are self-resetting
```

Existing 'fsm_encoding' attributes are not changed by this pass.

Signals can be protected from being detected by this pass by setting the 'fsm_encoding' attribute to "none".

This pass uses a subset of FF types to detect FSMs. Run 'opt -nosdff -nodffe' before this pass to prepare the design for fsm_detect.

10.86 fsm_expand - expand FSM cells by merging logic into it

```
yosys> help fsm_expand
```

```
fsm_expand [-full] [selection]
```

The fsm_extract pass is conservative about the cells that belong to a finite state machine. This pass can be used to merge additional auxiliary gates into the finite state machine.

By default, fsm_expand is still a bit conservative regarding merging larger word-wide cells. Call with -full to consider all cells for merging.

10.87 fsm_export - exporting FSMs to KISS2 files

```
yosys> help fsm_export
```

```
fsm_export [-noauto] [-o filename] [-origenc] [selection]
```

This pass creates a KISS2 file for every selected FSM. For FSMs with the 'fsm_export' attribute set, the attribute value is used as filename, otherwise the module and cell name is used as filename. If the parameter '-o' is given, the first exported FSM is written to the specified filename. This overwrites the setting as specified with the 'fsm_export' attribute. All other FSMs are exported to the default name as mentioned above.

```
-noauto
```

(continues on next page)

(continued from previous page)

```
    only export FSMs that have the 'fsm_export' attribute set

-o filename
    filename of the first exported FSM

-origenc
    use binary state encoding as state names instead of s0, s1, ...
```

10.88 fsm_extract - extracting FSMs in design

```
yosys> help fsm_extract
```

```
fsm_extract [selection]
```

This pass operates on all signals marked as FSM state signals using the 'fsm_encoding' attribute. It consumes the logic that creates the state signal and uses the state signal to generate control signal and replaces it with an FSM cell.

The generated FSM cell still generates the original state signal with its original encoding. The 'fsm_opt' pass can be used in combination with the 'opt_clean' pass to eliminate this signal.

10.89 fsm_info - print information on finite state machines

```
yosys> help fsm_info
```

```
fsm_info [selection]
```

This pass dumps all internal information on FSM cells. It can be useful for analyzing the synthesis process and is called automatically by the 'fsm' pass so that this information is included in the synthesis log file.

10.90 fsm_map - mapping FSMs to basic logic

```
yosys> help fsm_map
```

```
fsm_map [selection]
```

This pass translates FSM cells to flip-flops and logic.

10.91 fsm_opt - optimize finite state machines

yosys> help fsm_opt

```
fsm_opt [selection]
```

This pass optimizes FSM cells. It detects which output signals are actually not used and removes them from the FSM. This pass is usually used in combination with the 'opt_clean' pass (see also 'help fsm').

10.92 fsm_recode - recoding finite state machines

yosys> help fsm_recode

```
fsm_recode [options] [selection]
```

This pass reassign the state encodings for FSM cells. At the moment only one-hot encoding and binary encoding is supported.

```
-encoding <type>
```

specify the encoding scheme used for FSMs without the 'fsm_encoding' attribute or with the attribute set to 'auto'.

```
-fm_set_fsm_file <file>
```

generate a file containing the mapping from old to new FSM encoding in form of Synopsys Formality set_fsm_* commands.

```
-encfile <file>
```

write the mappings from old to new FSM encoding to a file in the following format:

```
.fsm <module_name> <state_signal>
.map <old_bitpattern> <new_bitpattern>
```

10.93 fst2tb - generate testbench out of fst file

yosys> help fst2tb

```
fst2tb [options] [top-level]
```

This command generates testbench for the circuit using the given top-level module and stimulus signal from FST file

```
-tb <name>
```

generated testbench name.
files <name>.v and <name>.txt are created as result.

```
-r <filename>
```

read simulation FST file

(continues on next page)

(continued from previous page)

```
-clock <portname>
    name of top-level clock input

-clockn <portname>
    name of top-level clock input (inverse polarity)

-scope <name>
    scope of simulation top model

-start <time>
    start co-simulation in arbitrary time (default 0)

-stop <time>
    stop co-simulation in arbitrary time (default END)

-n <integer>
    number of clock cycles to simulate (default: 20)
```

10.94 future - resolve future sampled value functions

```
yosys> help future
```

```
future [options] [selection]
```

10.95 gatemate_foldinv - fold inverters into Gatemate LUT trees

```
yosys> help gatemate_foldinv
```

```
gatemate_foldinv [selection]
```

This pass searches for `$_CC_NOT` cells and folds them into `CC_LUT2`, `CC_L2T4` and `CC_L2T5` cells as created by LUT tree mapping.

10.96 glift - create GLIFT models and optimization problems

```
yosys> help glift
```

```
glift <command> [options] [selection]
```

Augments the current or specified module with gate-level information flow tracking (GLIFT) logic using the "constructive mapping" approach. Also can set up QBF-SAT optimization problems in order to optimize GLIFT models or trade off precision and complexity.

(continues on next page)

(continued from previous page)

Commands:

`-create-precise-model`

Replaces the current or specified module with one that has corresponding "taint" inputs, outputs, and internal nets along with precise taint tracking logic. For example, precise taint tracking logic for an AND gate is:

```
y_t = a & b_t | b & a_t | a_t & b_t
```

`-create-imprecise-model`

Replaces the current or specified module with one that has corresponding "taint" inputs, outputs, and internal nets along with imprecise "All OR" taint tracking logic:

```
y_t = a_t | b_t
```

`-create-instrumented-model`

Replaces the current or specified module with one that has corresponding "taint" inputs, outputs, and internal nets along with 4 varying-precision versions of taint tracking logic. Which version of taint tracking logic is used for a given gate is determined by a MUX selected by an \$anyconst cell.

By default, unless the ``-no-cost-model`` option is provided, an additional wire named ``__glift_weight`` with the ``keep`` and ``minimize`` attributes is added to the module along with pmuxes and adders to calculate a rough estimate of the number of logic gates in the GLIFT model given an assignment for the \$anyconst cells. The four versions of taint tracking logic for an AND gate are:

```
y_t = a & b_t | b & a_t | a_t & b_t      (like `-create-precise-model`)
y_t = a_t | a & b_t
y_t = b_t | b & a_t
y_t = a_t | b_t                          (like `-create-imprecise-model`)
```

Options:

`-taint-constants`

Constant values in the design are labeled as tainted.
(default: label constants as un-tainted)

`-keep-outputs`

Do not remove module outputs. Taint tracking outputs will appear in the module ports alongside the original outputs.
(default: original module outputs are removed)

`-simple-cost-model`

Do not model logic area. Instead model the number of non-zero assignments to \$anyconsts. Taint tracking logic versions vary in their size, but all

(continues on next page)

(continued from previous page)

reduced-precision versions are significantly smaller than the fully-precise version. A non-zero \$anyconst assignment means that reduced-precision taint tracking logic was chosen for some gate. Only applicable in combination with ``-create-instrumented-model``. (default: use a complex model and give that wire the "keep" and "minimize" attributes)

-no-cost-model

Do not model taint tracking logic area and do not create a ``__glift_weight`` wire. Only applicable in combination with ``-create-instrumented-model``. (default: model area and give that wire the "keep" and "minimize" attributes)

-instrument-more

Allow choice from more versions of (even simpler) taint tracking logic. A total of 8 versions of taint tracking logic will be added per gate, including the 4 versions from ``-create-instrumented-model`` and these additional versions:

```
y_t = a_t
y_t = b_t
y_t = 1
y_t = 0
```

Only applicable in combination with ``-create-instrumented-model``. (default: do not add more versions of taint tracking logic.)

10.97 greenpak4_dffinv - merge greenpak4 inverters and DFF/latches

```
yosys> help greenpak4_dffinv
```

```
greenpak4_dffinv [options] [selection]
```

Merge GP_INV cells with GP_DFF* and GP_DLATCH* cells.

10.98 help - display help messages

```
yosys> help help
```

```
help ..... list all commands
help <command> ..... print help message for given command
help -all ..... print complete command reference

help -cells ..... list all cell types
help <celltype> ..... print help message for given cell type
help <celltype>+ .... print verilog code for given cell type
```

10.99 hierarchy - check, expand and clean up design hierarchy

```
yosys> help hierarchy
```

```
hierarchy [-check] [-top <module>]
hierarchy -generate <cell-types> <port-decls>
```

In parametric designs, a module might exist in several variations with different parameter values. This pass looks at all modules in the current design and re-runs the language frontends for the parametric modules as needed. It also resolves assignments to wired logic data types (wand/wor), resolves positional module parameters, unrolls array instances, and more.

```
-check
    also check the design hierarchy. this generates an error when
    an unknown module is used as cell type.

-simcheck
    like -check, but also throw an error if blackbox modules are
    instantiated, and throw an error if the design has no top module.

-smtcheck
    like -simcheck, but allow smtlib2_module modules.

-purge_lib
    by default the hierarchy command will not remove library (blackbox)
    modules. use this option to also remove unused blackbox modules.

-libdir <directory>
    search for files named <module_name>.v in the specified directory
    for unknown modules and automatically run read_verilog for each
    unknown module.

-keep_positionals
    per default this pass also converts positional arguments in cells
    to arguments using port names. This option disables this behavior.

-keep_portwidths
    per default this pass adjusts the port width on cells that are
    module instances when the width does not match the module port. This
    option disables this behavior.

-nodefualts
    do not resolve input port default values

-nokeep_prints
    per default this pass sets the "keep" attribute on all modules
    that directly or indirectly display text on the terminal.
    This option disables this behavior.

-nokeep_asserts
    per default this pass sets the "keep" attribute on all modules
```

(continues on next page)

(continued from previous page)

that directly or indirectly contain one or more formal properties.
This option disables this behavior.

`-top <module>`

use the specified top module to build the design hierarchy. Modules outside this tree (unused modules) are removed.

when the `-top` option is used, the 'top' attribute will be set on the specified top module. otherwise a module with the 'top' attribute set will implicitly be used as top module, if such a module exists.

`-auto-top`

automatically determine the top of the design hierarchy and mark it.

`-chparam name value`

elaborate the top module using this parameter value. Modules on which this parameter does not exist may cause a warning message to be output. This option can be specified multiple times to override multiple parameters. String values must be passed in double quotes (").

In `-generate` mode this pass generates blackbox modules for the given cell types (wildcards supported). For this the design is searched for cells that match the given types and then the given port declarations are used to determine the direction of the ports. The syntax for a port declaration is:

```
{i|o|io}[<num>]:<portname>
```

Input ports are specified with the 'i' prefix, output ports with the 'o' prefix and inout ports with the 'io' prefix. The optional `<num>` specifies the position of the port in the parameter list (needed when instantiated using positional arguments). When `<num>` is not specified, the `<portname>` can also contain wildcard characters.

This pass ignores the current selection and always operates on all modules in the current design.

10.100 hilomap - technology mapping of constant hi- and/or lo-drivers

```
yosys> help hilomap
```

```
hilomap [options] [selection]
```

Map constants to 'tielo' and 'tiehi' driver cells.

`-hicell <celltype> <portname>`

Replace constant hi bits with this cell.

`-locell <celltype> <portname>`

Replace constant lo bits with this cell.

(continues on next page)

(continued from previous page)

```
-singleton
  Create only one hi/lo cell and connect all constant bits
  to that cell. Per default a separate cell is created for
  each constant bit.
```

10.101 history - show last interactive commands

```
yosys> help history
```

```
history
```

This command prints all commands in the shell history buffer. This are all commands executed in an interactive session, but not the commands from executed scripts.

10.102 ice40_braminit - iCE40: perform SB_RAM40_4K initialization from file

```
yosys> help ice40_braminit
```

```
ice40_braminit
```

This command processes all SB_RAM40_4K blocks with a non-empty INIT_FILE parameter and converts it into the required INIT_x attributes

10.103 ice40_dsp - iCE40: map multipliers

```
yosys> help ice40_dsp
```

```
ice40_dsp [options] [selection]
```

Map multipliers (\$mul/SB_MAC16) and multiply-accumulate (\$mul/SB_MAC16 + \$add) cells into iCE40 DSP resources.

Currently, only the 16x16 multiply mode is supported and not the 2 x 8x8 mode.

Pack input registers (A, B, {C,D}; with optional hold), pipeline registers ({F,J,K,G}, H), output registers (O -- full 32-bits or lower 16-bits only; with optional hold), and post-adder into the SB_MAC16 resource.

Multiply-accumulate operations using the post-adder with feedback on the {C,D} input will be folded into the DSP. In this scenario only, resetting the the accumulator to an arbitrary value can be inferred to use the {C,D} input.

10.104 ice40_opt - iCE40: perform simple optimizations

```
yosys> help ice40_opt
```

```
ice40_opt [options] [selection]
```

This command executes the following script:

```
do
    <ice40 specific optimizations>
    opt_expr -mux_undef -undriven [-full]
    opt_merge
    opt_dff
    opt_clean
while <changed design>
```

10.105 ice40_wrapcarry - iCE40: wrap carries

```
yosys> help ice40_wrapcarry
```

```
ice40_wrapcarry [selection]
```

Wrap manually instantiated SB_CARRY cells, along with their associated SB_LUT4s, into an internal `$_ICE40_CARRY_WRAPPER` cell for preservation across technology mapping.

Attributes on both cells will have their names prefixed with 'SB_CARRY.' or 'SB_LUT4.' and attached to the wrapping cell.

A (* keep *) attribute on either cell will be logically OR-ed together.

```
-unwrap
    unwrap $_ICE40_CARRY_WRAPPER cells back into SB_CARRYs and SB_LUT4s,
    including restoring their attributes.
```

10.106 insbuf - insert buffer cells for connected wires

```
yosys> help insbuf
```

```
insbuf [options] [selection]
```

Insert buffer cells into the design for directly connected wires.

```
-buf <celltype> <in-portname> <out-portname>
    Use the given cell type instead of $_BUF_. (Notice that the next
    call to "clean" will remove all $_BUF_ in the design.)
```

```
-chain
    Chain buffer cells
```

10.107 internal_stats - print internal statistics

```
yosys> help internal_stats
```

Print internal statistics for developers (experimental)

10.108 iopadmap - technology mapping of i/o pads (or buffers)

```
yosys> help iopadmap
```

```
iopadmap [options] [selection]
```

Map module inputs/outputs to PAD cells from a library. This pass can only map to very simple PAD cells. Use 'techmap' to further map the resulting cells to more sophisticated PAD cells.

```
-inpad <celltype> <in_port>[:<ext_port>]
```

Map module input ports to the given cell type with the given output port name. if a 2nd portname is given, the signal is passed through the pad cell, using the 2nd portname as the port facing the module port.

```
-outpad <celltype> <out_port>[:<ext_port>]
```

```
-inoutpad <celltype> <io_port>[:<ext_port>]
```

Similar to -inpad, but for output and inout ports.

```
-toutpad <celltype> <oe_port>:<out_port>[:<ext_port>]
```

Merges \$_TBUF_ cells into the output pad cell. This takes precedence over the other -outpad cell. The first portname is the enable input of the tristate driver, which can be prefixed with `~` for negative polarity enable.

```
-tinoutpad <celltype> <oe_port>:<in_port>:<out_port>[:<ext_port>]
```

Merges \$_TBUF_ cells into the inout pad cell. This takes precedence over the other -inoutpad cell. The first portname is the enable input of the tristate driver and the 2nd portname is the internal output buffering the external signal. Like with `~`-toutpad, the enable can be marked as negative polarity by prefixing the name with `~`.

```
-ignore <celltype> <portname>[:<portname>]*
```

Skips mapping inputs/outputs that are already connected to given ports of the given cell. Can be used multiple times. This is in addition to the cells specified as mapping targets.

```
-widthparam <param_name>
```

Use the specified parameter name to set the port width.

```
-nameparam <param_name>
```

Use the specified parameter to set the port name.

(continues on next page)

(continued from previous page)

```
-bits
    create individual bit-wide buffers even for ports that
    are wider. (the default behavior is to create word-wide
    buffers using -widthparam to set the word size on the cell.)
```

Tristate PADS (-toutpad, -tinoutpad) always operate in -bits mode.

10.109 jny - write design and metadata

```
yosys> help jny
```

```
jny [options] [selection]
```

Write JSON netlist metadata for the current design

```
-o <filename>
    write to the specified file.

-no-connections
    Don't include connection information in the netlist output.

-no-attributes
    Don't include attributed information in the netlist output.

-no-properties
    Don't include property information in the netlist output.
```

See 'help write_jny' for a description of the JSON format used.

10.110 json - write design in JSON format

```
yosys> help json
```

```
json [options] [selection]
```

Write a JSON netlist of all selected objects.

```
-o <filename>
    write to the specified file.

-aig
    also include AIG models for the different gate types

-compat-int
    emit 32-bit or smaller fully-defined parameter values directly
    as JSON numbers (for compatibility with old parsers)
```

(continues on next page)

(continued from previous page)

```
-noscopeinfo
    don't include $scopeinfo cells in the output
```

See 'help write_json' for a description of the JSON format used.

10.111 keep_hierarchy - selectively add the keep_hierarchy attribute

```
yosys> help keep_hierarchy
```

```
keep_hierarchy [options] [selection]
```

Add the keep_hierarchy attribute.

```
-min_cost <min_cost>
    only add the attribute to modules estimated to have more than <min_cost>
    gates after simple techmapping. Intended for tuning trade-offs between
    quality and yosys runtime.
```

When evaluating a module's cost, gates which are within a submodule which is marked with the keep_hierarchy attribute are not counted towards the upper module's cost. This applies to both when the attribute was added by this command or was pre-existing.

10.112 lattice_gsr - Lattice: handle GSR

```
yosys> help lattice_gsr
```

```
lattice_gsr [options] [selection]
```

Trim active low async resets connected to GSR and resolve GSR parameter, if a GSR or SGSR primitive is used in the design.

If any cell has the GSR parameter set to "AUTO", this will be resolved to "ENABLED" if a GSR primitive is present and the (* nogsr *) attribute is not set, otherwise it will be resolved to "DISABLED".

10.113 license - print license terms

```
yosys> help license
```

```
license
```

This command produces the following notice.

```
/-----\
```

(continues on next page)

(continued from previous page)

```

| yosys -- Yosys Open SYnthesis Suite |
| Copyright (C) 2012 - 2025  Claire Xenia Wolf <claire@yosyshq.com> |
| Permission to use, copy, modify, and/or distribute this software for any |
| purpose with or without fee is hereby granted, provided that the above |
| copyright notice and this permission notice appear in all copies. |
| THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES |
| WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF |
| MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR |
| ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES |
| WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN |
| ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF |
| OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE. |
| -----/

```

10.114 log - print text and log files

yosys> help log

```

log [options] string
log [ -push | -pop ]

```

Print the given string to the screen and/or the log file. This is useful for TCL scripts, because the TCL command "puts" only goes to stdout but not to logfiles.

-stdout

Print the output to stdout too. This is useful when all Yosys is executed with a script and the -q (quiet operation) argument to notify the user.

-stderr

Print the output to stderr too.

-nolog

Don't use the internal log() command. Use either -stdout or -stderr, otherwise no output will be generated at all.

-n

do not append a newline

-header

log a pass header

-push

push a new level on the pass counter

(continues on next page)

(continued from previous page)

```
-pop
    pop from the pass counter
```

10.115 logger - set logger properties

```
yosys> help logger
```

```
logger [options]
```

This command sets global logger properties, also available using command line options.

```
-[no]time
    enable/disable display of timestamp in log output.

-[no]stderr
    enable/disable logging errors to stderr.

-warn regex
    print a warning for all log messages matching the regex.

-nowarn regex
    if a warning message matches the regex, it is printed as regular
    message instead.

-werror regex
    if a warning message matches the regex, it is printed as error
    message instead and the tool terminates with a nonzero return code.

-[no]debug
    globally enable/disable debug log messages.

-experimental <feature>
    do not print warnings for the specified experimental feature

-expect <type> <regex> <expected_count>
    expect log, warning or error to appear. matched errors will terminate
    with exit code 0.

-expect-no-warnings
    gives error in case there is at least one warning that is not expected.

-check-expected
    verifies that the patterns previously set up by -expect have actually
    been met, then clears the expected log list. If this is not called
    manually, the check will happen at yosys exist time instead.
```

10.116 ls - list modules or objects in modules

```
yosys> help ls
```

```
ls [selection]
```

When no active module is selected, this prints a list of modules.

When an active module is selected, this prints a list of objects in the module.

10.117 ltp - print longest topological path

```
yosys> help ltp
```

```
ltp [options] [selection]
```

This command prints the longest topological path in the design. (Only considers paths within a single module, so the design must be flattened.)

```
-noff  
    automatically exclude FF cell types
```

10.118 lut2mux - convert \$lut to \$_MUX_

```
yosys> help lut2mux
```

```
lut2mux [options] [selection]
```

This pass converts \$lut cells to \$_MUX_ gates.

10.119 maccmap - mapping macc cells

```
yosys> help maccmap
```

```
maccmap [-unmap] [selection]
```

This pass maps \$macc cells to yosys \$fa and \$alu cells. When the -unmap option is used then the \$macc cell is mapped to \$add, \$sub, etc. cells instead.

10.120 memory - translate memories to basic cells

yosys> help memory

```
memory [-norom] [-nomap] [-nordff] [-nowiden] [-nosat] [-memx] [-no-rw-check] [-bram
↪<bram_rules>] [selection]
```

This pass calls all the other memory_* passes in a useful order:

```
opt_mem
opt_mem_priority
opt_mem_feedback
memory_bmux2rom                (skipped if called with -norom)
memory_dff [-no-rw-check]      (skipped if called with -nordff or -memx)
opt_clean
memory_share [-nowiden] [-nosat]
opt_mem_widen
memory_memx                    (when called with -memx)
opt_clean
memory_collect
memory_bram -rules <bram_rules> (when called with -bram)
memory_map                    (skipped if called with -nomap)
```

This converts memories to word-wide DFFs and address decoders or multiport memory blocks if called with the -nomap option.

10.121 memory_bmux2rom - convert muxes to ROMs

yosys> help memory_bmux2rom

```
memory_bmux2rom [options] [selection]
```

This pass converts \$bmux cells with constant A input to ROMs.

10.122 memory_bram - map memories to block rams

yosys> help memory_bram

```
memory_bram -rules <rule_file> [selection]
```

This pass converts the multi-port \$mem memory cells into block ram instances. The given rules file describes the available resources and how they should be used.

The rules file contains configuration options, a set of block ram description and a sequence of match rules.

The option 'attr_icase' configures how attribute values are matched. The value 0

(continues on next page)

(continued from previous page)

means case-sensitive, 1 means case-insensitive.

A block ram description looks like this:

```

bram RAMB1024X32      # name of BRAM cell
  init 1              # set to '1' if BRAM can be initialized
  abits 10            # number of address bits
  dbits 32            # number of data bits
  groups 2            # number of port groups
  ports 1 1           # number of ports in each group
  wrmode 1 0          # set to '1' if this groups is write ports
  enable 4 1          # number of enable bits
  transp 0 2          # transparent (for read ports)
  clocks 1 2          # clock configuration
  clkpol 2 2          # clock polarity configuration
endbram

```

For the option 'transp' the value 0 means non-transparent, 1 means transparent and a value greater than 1 means configurable. All groups with the same value greater than 1 share the same configuration bit.

For the option 'clocks' the value 0 means non-clocked, and a value greater than 0 means clocked. All groups with the same value share the same clock signal.

For the option 'clkpol' the value 0 means negative edge, 1 means positive edge and a value greater than 1 means configurable. All groups with the same value greater than 1 share the same configuration bit.

Using the same bram name in different bram blocks will create different variants of the bram. Verilog configuration parameters for the bram are created as needed.

It is also possible to create variants by repeating statements in the bram block and appending '@<label>' to the individual statements.

A match rule looks like this:

```

match RAMB1024X32
  max waste 16384      # only use this bram if <= 16k ram bits are unused
  min efficiency 80    # only use this bram if efficiency is at least 80%
endmatch

```

It is possible to match against the following values with min/max rules:

```

words ..... number of words in memory in design
abits ..... number of address bits on memory in design
dbits ..... number of data bits on memory in design
wports ..... number of write ports on memory in design
rports ..... number of read ports on memory in design
ports ..... number of ports on memory in design
bits ..... number of bits in memory in design

```

(continues on next page)

(continued from previous page)

```

dups ..... number of duplications for more read ports

awaste ..... number of unused address slots for this match
dwaste ..... number of unused data bits for this match
bwaste ..... number of unused bram bits for this match
waste ..... total number of unused bram bits (bwaste*dups)
efficiency ... total percentage of used and non-duplicated bits

acells ..... number of cells in 'address-direction'
dcells ..... number of cells in 'data-direction'
cells ..... total number of cells (acells*dcells*dups)

```

A match containing the command 'attribute' followed by a list of space separated 'name[=string_value]' values requires that the memory contains any one of the given attribute name and string values (where specified), or name and integer 1 value (if no string_value given, since Verilog will interpret '(* attr *)' as '(* attr=1 *)').

A name prefixed with '!' indicates that the attribute must not exist.

The interface for the created bram instances is derived from the bram description. Use 'techmap' to convert the created bram instances into instances of the actual bram cells of your target architecture.

A match containing the command 'or_next_if_better' is only used if it has a higher efficiency than the next match (and the one after that if the next also has 'or_next_if_better' set, and so forth).

A match containing the command 'make_transp' will add external circuitry to simulate 'transparent read', if necessary.

A match containing the command 'make_outreg' will add external flip-flops to implement synchronous read ports, if necessary.

A match containing the command 'shuffle_enable A' will re-organize the data bits to accommodate the enable pattern of port A.

10.123 memory_collect - creating multi-port memory cells

```
yosys> help memory_collect
```

```
memory_collect [selection]
```

This pass collects memories and memory ports and creates generic multiport memory cells.

10.124 memory_dff - merge input/output DFFs into memory read ports

```
yosys> help memory_dff
```

```
memory_dff [-no-rw-check] [selection]
```

This pass detects DFFs at memory read ports and merges them into the memory port. I.e. it consumes an asynchronous memory port and the flip-flops at its interface and yields a synchronous memory port.

-no-rw-check

marks all recognized read ports as "return don't-care value on read/write collision" (same result as setting the `no_rw_check` attribute on all memories).

10.125 memory_libmap - map memories to cells

```
yosys> help memory_libmap
```

```
memory_libmap -lib <library_file> [-D <condition>] [selection]
```

This pass takes a description of available RAM cell types and maps all selected memories to one of them, or leaves them to be mapped to FFs.

-lib <library_file>

Selects a library file containing RAM cell definitions. This option can be passed more than once to select multiple libraries. See `passes/memory/memlib.md` for description of the library format.

-D <condition>

Enables a condition that can be checked within the library file to eg. select between slightly different hardware variants. This option can be passed any number of times.

-logic-cost-rom <num>

-logic-cost-ram <num>

Sets the cost of a single bit for memory lowered to soft logic.

-no-auto-distributed

-no-auto-block

-no-auto-huge

Disables automatic mapping of given kind of RAMs. Manual mapping (using `ram_style` or other attributes) is still supported.

10.126 memory_map - translate multiport memories to basic cells

yosys> help memory_map

```
memory_map [options] [selection]
```

This pass converts multiport memory cells as generated by the memory_collect pass to word-wide DFFs and address decoders.

```
-attr !<name>
    do not map memories that have attribute <name> set.

-attr <name>[=<value>]
    for memories that have attribute <name> set, only map them if its value
    is a string <value> (if specified), or an integer 1 (otherwise). if this
    option is specified multiple times, map the memory if the attribute is
    to any of the values.

-iattr
    for -attr, suppress case sensitivity in matching of <value>.

-rom-only
    only perform conversion for ROMs (memories with no write ports).

-keepdc
    when mapping ROMs, keep x-bits shared across read ports.

-formal
    map memories for a global clock based formal verification flow.
    This implies -keepdc, uses $ff cells for ROMs and sets hdlname
    attributes. It also has limited support for async write ports
    as generated by clk2fflogic.
```

10.127 memory_memx - emulate vlog sim behavior for mem ports

yosys> help memory_memx

```
memory_memx [selection]
```

This pass adds additional circuitry that emulates the Verilog simulation behavior for out-of-bounds memory reads and writes.

10.128 memory_narrow - split up wide memory ports

```
yosys> help memory_narrow
```

```
memory_narrow [options] [selection]
```

This pass splits up wide memory ports into several narrow ports.

10.129 memory_nordff - extract read port FFs from memories

```
yosys> help memory_nordff
```

```
memory_nordff [options] [selection]
```

This pass extracts FFs from memory read ports. This results in a netlist similar to what one would get from not calling memory_dff.

10.130 memory_share - consolidate memory ports

```
yosys> help memory_share
```

```
memory_share [-nosat] [-nowiden] [selection]
```

This pass merges share-able memory ports into single memory ports.

The following methods are used to consolidate the number of memory ports:

- When multiple write ports access the same address then this is converted to a single write port with a more complex data and/or enable logic path.
- When multiple read or write ports access adjacent aligned addresses, they are merged to a single wide read or write port. This transformation can be disabled with the "-nowiden" option.
- When multiple write ports are never accessed at the same time (a SAT solver is used to determine this), then the ports are merged into a single write port. This transformation can be disabled with the "-nosat" option.

Note that in addition to the algorithms implemented in this pass, the \$memrd and \$memwr cells are also subject to generic resource sharing passes (and other optimizations) such as "share" and "opt_merge".

10.131 memory_unpack - unpack multi-port memory cells

yosys> help memory_unpack

```
memory_unpack [selection]
```

This pass converts the multi-port \$mem memory cells into individual \$memrd and \$memwr cells. It is the counterpart to the memory_collect pass.

10.132 microchip_dffopt - MICROCHIP: optimize FF control signal usage

yosys> help microchip_dffopt

```
microchip_dffopt [options] [selection]
```

Converts hardware clock enable and set/reset signals on FFs to emulation using LUTs, if doing so would improve area. Operates on post-techmap LUT, DFF cells.

10.133 microchip_dsp - MICROCHIP: pack resources into DSPs

yosys> help microchip_dsp

```
microchip_dsp [options] [selection]
```

Pack input registers 'A', 'B', 'C', and 'D' (with optional enable/reset), output register 'P' (with optional enable/reset), pre-adder and/or post-adder into Microchip DSP resources.

Multiply-accumulate operations using the post-adder with feedback on the 'C' input will be folded into the DSP. In this scenario only, the 'C' input can be used to override the current accumulation result with a new value. This will be added to the multiplier result to form the next accumulation result.

Use of the dedicated 'PCOUT' -> 'PCIN' cascade path is detected for 'P' -> 'C' connections (optionally, where 'P' is right-shifted by 17-bits and used as an input to the post-adder. This pattern is common for summing partial products to implement wide multipliers). Cascade chains are limited to a maximum length of 24 cells, corresponding to PolarFire (pf) devices.

This pass is a no-op if the scratchpad variable 'microchip_dsp.multonly' is set to 1.

```
-family {polarfire}
  select the family to target
  default: polarfire
```

10.134 miter - automatically create a miter circuit

yosys> help miter

```
miter -equiv [options] gold_name gate_name miter_name
```

Creates a miter circuit for equivalence checking. The gold- and gate- modules must have the same interfaces. The miter circuit will have all inputs of the two source modules, prefixed with 'in_'. The miter circuit has a 'trigger' output that goes high if an output mismatch between the two source modules is detected.

-ignore_gold_x

a undef (x) bit in the gold module output will match any value in the gate module output.

-make_outputs

also route the gold- and gate-outputs to 'gold_*' and 'gate_*' outputs on the miter circuit.

-make_outcmp

also create a cmp_* output for each gold/gate output pair.

-make_assert

also create an 'assert' cell that checks if trigger is always low.

-make_cover

also create a 'cover' cell for each gold/gate output pair.

-flatten

call 'flatten -wb; opt_expr -keepdc -undriven;;' on the miter circuit.

-cross

allow output ports on the gold module to match input ports on the gate module. This is useful when the gold module contains additional logic to drive some of the gate module inputs.

```
miter -assert [options] module [miter_name]
```

Creates a miter circuit for property checking. All input ports are kept, output ports are discarded. An additional output 'trigger' is created that goes high when an assert is violated. Without a miter_name, the existing module is modified.

-make_outputs

keep module output ports.

-flatten

call 'flatten -wb; opt_expr -keepdc -undriven;;' on the miter circuit.

10.135 mutate - generate or apply design mutations

yosys> help mutate

```
mutate -list N [options] [selection]
```

Create a list of N mutations using an even sampling.

```
-o filename
```

Write list to this file instead of console output

```
-s filename
```

Write a list of all src tags found in the design to the specified file

```
-seed N
```

RNG seed for selecting mutations

```
-none
```

Include a "none" mutation in the output

```
-ctrl name width value
```

Add -ctrl options to the output. Use 'value' for first mutation, then simply count up from there.

```
-mode name
```

```
-module name
```

```
-cell name
```

```
-port name
```

```
-portbit int
```

```
-ctrlbit int
```

```
-wire name
```

```
-wirebit int
```

```
-src string
```

Filter list of mutation candidates to those matching the given parameters.

```
-cfg option int
```

Set a configuration option. Options available:

```
weight_pq_w weight_pq_b weight_pq_c weight_pq_s
```

```
weight_pq_mw weight_pq_mb weight_pq_mc weight_pq_ms
```

```
weight_cover pick_cover_prct
```

```
mutate -mode MODE [options]
```

Apply the given mutation.

```
-ctrl name width value
```

Add a control signal with the given name and width. The mutation is activated if the control signal equals the given value.

```
-module name
```

(continues on next page)

(continued from previous page)

```
-cell name
-port name
-portbit int
-ctrlbit int
    Mutation parameters, as generated by 'mutate -list N'.

-wire name
-wirebit int
-src string
    Ignored. (They are generated by -list for documentation purposes.)
```

10.136 muxcover - cover trees of MUX cells with wider MUXes

```
yosys> help muxcover
```

```
    muxcover [options] [selection]

Cover trees of $_MUX_ cells with $_MUX{4,8,16}_ cells

-mux4[=cost], -mux8[=cost], -mux16[=cost]
    Cover $_MUX_ trees using the specified types of MUXes (with optional
    integer costs). If none of these options are given, the effect is the
    same as if all of them are.
    Default costs: $_MUX4_ = 220, $_MUX8_ = 460,
                  $_MUX16_ = 940

-mux2=cost
    Use the specified cost for $_MUX_ cells when making covering decisions.
    Default cost: $_MUX_ = 100

-dmux=cost
    Use the specified cost for $_MUX_ cells used in decoders.
    Default cost: 90

-nocode
    Do not insert decoder logic. This reduces the number of possible
    substitutions, but guarantees that the resulting circuit is not
    less efficient than the original circuit.

-nopartial
    Do not consider mappings that use $_MUX<N>_ to select from less
    than <N> different signals.
```

10.137 muxpack - \$mux/\$pmux cascades to \$pmux

yosys> help muxpack

```
muxpack [selection]
```

This pass converts cascaded chains of \$pmux cells (e.g. those created from case constructs) and \$mux cells (e.g. those created by if-else constructs) into \$pmux cells.

This optimisation is conservative --- it will only pack \$mux or \$pmux cells whose select lines are driven by '\$eq' cells with other such cells if it can be certain that their select inputs are mutually exclusive.

10.138 nlutmap - map to LUTs of different sizes

yosys> help nlutmap

```
nlutmap [options] [selection]
```

This pass uses successive calls to 'abc' to map to an architecture. That provides a small number of differently sized LUTs.

```
-luts N_1,N_2,N_3,...
```

The number of LUTs with 1, 2, 3, ... inputs that are available in the target architecture.

```
-assert
```

Create an error if not all logic can be mapped

Excess logic that does not fit into the specified LUTs is mapped back to generic logic gates (\$_AND_, etc.).

10.139 nx_carry - NanoXplore: create carry cells

yosys> help nx_carry

```
nx_carry [options] [selection]
```

Fixes carry chain if needed, break it on 24 elements and group by 4 into NX_CY.

10.140 onehot - optimize \$eq cells for onehot signals

```
yosys> help onehot
```

```
onehot [options] [selection]
```

This pass optimizes \$eq cells that compare one-hot signals against constants

```
-v, -vv  
    verbose output
```

10.141 opt - perform simple optimizations

```
yosys> help opt
```

```
opt [options] [selection]
```

This pass calls all the other opt_* passes in a useful order. This performs a series of trivial optimizations and cleanups. This pass executes the other passes in the following order:

```
opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-keepdc]  
opt_merge [-share_all] -nomux
```

```
do
```

```
    opt_muxtree  
    opt_reduce [-fine] [-full]  
    opt_merge [-share_all]  
    opt_share (-full only)  
    opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat] (except when called with -noff)  
    opt_clean [-purge]  
    opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-
```

```
↪keepdc]
```

```
    while <changed design>
```

When called with -fast the following script is used instead:

```
do
```

```
    opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-
```

```
↪keepdc]
```

```
    opt_merge [-share_all]  
    opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat] (except when called with -noff)  
    opt_clean [-purge]
```

```
    while <changed design in opt_dff>
```

Note: Options in square brackets (such as [-keepdc]) are passed through to the opt_* commands when given to 'opt'.

10.142 opt_clean - remove unused cells and wires

```
yosys> help opt_clean
```

```
opt_clean [options] [selection]
```

This pass identifies wires and cells that are unused and removes them. Other passes often remove cells but leave the wires in the design or reconnect the wires but leave the old cells in the design. This pass can be used to clean up after the passes that do the actual work.

This pass only operates on completely selected modules without processes.

```
-purge
```

```
    also remove internal nets if they have a public name
```

10.143 opt_demorgan - Optimize reductions with DeMorgan equivalents

```
yosys> help opt_demorgan
```

```
opt_demorgan [selection]
```

This pass pushes inverters through \$reduce_* cells if this will reduce the overall gate count of the circuit

10.144 opt_dff - perform DFF optimizations

```
yosys> help opt_dff
```

```
opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat] [selection]
```

This pass converts flip-flops to a more suitable type by merging clock enables and synchronous reset multiplexers, removing unused control inputs, or potentially removes the flip-flop altogether, converting it to a constant driver.

```
-nodffe
```

```
    disables dff -> dffe conversion, and other transforms recognizing clock enable
```

```
-nosdff
```

```
    disables dff -> sdff conversion, and other transforms recognizing sync resets
```

```
-simple-dffe
```

```
    only enables clock enable recognition transform for obvious cases
```

(continues on next page)

(continued from previous page)

```
-sat
    additionally invoke SAT solver to detect and remove flip-flops (with
    non-constant inputs) that can also be replaced with a constant driver

-keepdc
    some optimizations change the behavior of the circuit with respect to
    don't-care bits. for example in 'a+0' a single x-bit in 'a' will cause
    all result bits to be set to x. this behavior changes when 'a+0' is
    replaced by 'a'. the -keepdc option disables all such optimizations.
```

10.145 `opt_expr` - perform const folding and simple expression rewriting

```
yosys> help opt_expr
```

```
opt_expr [options] [selection]
```

This pass performs const folding on internal cell types with constant inputs.
It also performs some simple expression rewriting.

```
-mux_undef
    remove 'undef' inputs from $mux, $pmux and $_MUX_ cells

-mux_bool
    replace $mux cells with inverters or buffers when possible

-undriven
    replace undriven nets with undef (x) constants

-noclkinv
    do not optimize clock inverters by changing FF types

-fine
    perform fine-grain optimizations

-full
    alias for -mux_undef -mux_bool -undriven -fine

-keepdc
    some optimizations change the behavior of the circuit with respect to
    don't-care bits. for example in 'a+0' a single x-bit in 'a' will cause
    all result bits to be set to x. this behavior changes when 'a+0' is
    replaced by 'a'. the -keepdc option disables all such optimizations.
```

10.146 opt_ffinv - push inverters through FFs

```
yosys> help opt_ffinv
```

```
opt_ffinv [selection]
```

This pass pushes inverters to the other side of a FF when they can be merged into LUTs on the other side.

10.147 opt_lut - optimize LUT cells

```
yosys> help opt_lut
```

```
opt_lut [options] [selection]
```

This pass combines cascaded \$lut cells with unused inputs.

```
-tech ice40
    treat the design as a LUT-mapped circuit for the iCE40 architecture
    and preserve connections to SB_CARRY as appropriate

-limit N
    only perform the first N combines, then stop. useful for debugging.
```

10.148 opt_lut_ins - discard unused LUT inputs

```
yosys> help opt_lut_ins
```

```
opt_lut_ins [options] [selection]
```

This pass removes unused inputs from LUT cells (that is, inputs that can not influence the output signal given this LUT's value). While such LUTs cannot be directly emitted by ABC, they can be a result of various post-ABC transformations, such as mapping wide LUTs (not all sub-LUTs will use the full set of inputs) or optimizations such as xilinx_dffopt.

```
-tech <technology>
    Instead of generic $lut cells, operate on LUT cells specific
    to the given technology. Valid values are: xilinx, lattice, gowin.
```

10.149 `opt_mem` - optimize memories

```
yosys> help opt_mem
```

```
opt_mem [options] [selection]
```

This pass performs various optimizations on memories in the design.

10.150 `opt_mem_feedback` - convert memory read-to-write port feedback paths to write enables

```
yosys> help opt_mem_feedback
```

```
opt_mem_feedback [selection]
```

This pass detects cases where an asynchronous read port is only connected via a mux tree to a write port with the same address. When such a connection is found, it is replaced with a new condition on an enable signal, allowing for removal of the read port.

10.151 `opt_mem_priority` - remove priority relations between write ports that can never collide

```
yosys> help opt_mem_priority
```

```
opt_mem_priority [selection]
```

This pass detects cases where one memory write port has priority over another even though they can never collide with each other -- ie. there can never be a situation where a given memory bit is written by both ports at the same time, for example because of always-different addresses, or mutually exclusive enable signals. In such cases, the priority relation is removed.

10.152 `opt_mem_widen` - optimize memories where all ports are wide

```
yosys> help opt_mem_widen
```

```
opt_mem_widen [options] [selection]
```

This pass looks for memories where all ports are wide and adjusts the base memory width up until that stops being the case.

10.153 opt_merge - consolidate identical cells

yosys> help opt_merge

```
opt_merge [options] [selection]
```

This pass identifies cells with identical type and input signals. Such cells are then merged to one cell.

```
-nomux
```

Do not merge MUX cells.

```
-share_all
```

Operate on all cell types, not just built-in types.

```
-keepdc
```

Do not merge flipflops with don't-care bits in their initial value.

10.154 opt_muxtree - eliminate dead trees in multiplexer trees

yosys> help opt_muxtree

```
opt_muxtree [selection]
```

This pass analyzes the control signals for the multiplexer trees in the design and identifies inputs that can never be active. It then removes this dead branches from the multiplexer trees.

This pass only operates on completely selected modules without processes.

10.155 opt_reduce - simplify large MUXes and AND/OR gates

yosys> help opt_reduce

```
opt_reduce [options] [selection]
```

This pass performs two interlinked optimizations:

1. it consolidates trees of large AND gates or OR gates and eliminates duplicated inputs.
2. it identifies duplicated inputs to MUXes and replaces them with a single input with the original control signals OR'ed together.

```
-fine
```

perform fine-grain optimizations

(continues on next page)

(continued from previous page)

```
-full  
  alias for -fine
```

10.156 `opt_share` - merge mutually exclusive cells of the same type that share an input signal

```
yosys> help opt_share
```

```
opt_share [selection]
```

This pass identifies mutually exclusive cells of the same type that:

- (a) share an input signal,
- (b) drive the same `$mux`, `$_MUX_`, or `$pmux` multiplexing cell,

allowing the cell to be merged and the multiplexer to be moved from multiplexing its output to multiplexing the non-shared input signals.

10.157 `paramap` - renaming cell parameters

```
yosys> help paramap
```

```
paramap [options] [selection]
```

This command renames cell parameters and/or maps key/value pairs to other key/value pairs.

```
-tocase <name>  
  Match attribute names case-insensitively and set it to the specified  
  name.  
  
-rename <old_name> <new_name>  
  Rename attributes as specified  
  
-map <old_name>=<old_value> <new_name>=<new_value>  
  Map key/value pairs as indicated.  
  
-imap <old_name>=<old_value> <new_name>=<new_value>  
  Like -map, but use case-insensitive match for <old_value> when  
  it is a string value.  
  
-remove <name>=<value>  
  Remove attributes matching this pattern.
```

For example, mapping Diamond-style ECP5 "init" attributes to Yosys-style:

```
paramap -tocase INIT t:LUT4
```

10.158 peepopt - collection of peephole optimizers

yosys> help peepopt

```
peepopt [options] [selection]
```

This pass applies a collection of peephole optimizers to the current design.

This pass employs the following rules by default:

- * muldiv - Replace $(A*B)/B$ with A

- * shiftmul - Replace $A \gg (B*C)$ with $A' \gg (B \ll K)$ where C and K are constants and A' is derived from A by appropriately inserting padding into the signal. (right variant)

Analogously, replace $A \ll (B*C)$ with appropriate selection of output bits from $A \ll (B \ll K)$. (left variant)

- * shiftadd - Replace $A \gg (B+D)$ with $(A' \gg D) \gg (B)$ where D is constant and A' is derived from A by padding or cutting inaccessible bits. Scratchpad: 'peepopt.shiftadd.max_data_multiple' (default: 2) limits the amount of padding to a multiple of the data, to avoid high resource usage from large temporary MUX trees.

If -formalclk is specified it instead employs the following rules:

- * clockgateff - Replace latch based clock gating patterns with a flip-flop based pattern to prevent combinational paths from the output to the enable input after running clk2fflogic.

10.159 plugin - load and list loaded plugins

yosys> help plugin

```
plugin [options]
```

Load and list loaded plugins.

- i <plugin_filename>

Load (install) the specified plugin.

- a <alias_name>

Register the specified alias name for the loaded plugin

- l

List loaded plugins

10.160 pmux2shiftx - transform \$pmux cells to \$shiftx cells

```
yosys> help pmux2shiftx
```

```
pmux2shiftx [options] [selection]
```

This pass transforms \$pmux cells to \$shiftx cells.

```
-v, -vv
```

verbose output

```
-min_density <percentage>
```

specifies the minimum density for the shifter

default: 50

```
-min_choices <int>
```

specified the minimum number of choices for a control signal

default: 3

```
-onehot ignore|pmux|shiftx
```

select strategy for one-hot encoded control signals

default: pmux

```
-norange
```

disable \$sub inference for "range decoders"

10.161 pmuxtree - transform \$pmux cells to trees of \$mux cells

```
yosys> help pmuxtree
```

```
pmuxtree [selection]
```

This pass transforms \$pmux cells to trees of \$mux cells.

10.162 portarcs - derive port arcs for propagation delay

```
yosys> help portarcs
```

```
portarcs [options] [selection]
```

This command characterizes the combinational content of selected modules and derives timing arcs going from module inputs to module outputs representing the propagation delay of the module.

```
-draw
```

plot the computed delay table to the terminal

```
-icells
```

(continues on next page)

(continued from previous page)

```

    assign unit delay to gates from the internal Yosys cell library

-write
    write the computed arcs back into the module as $specify2 instances

```

10.163 portlist - list (top-level) ports

yosys> help portlist

```
portlist [options] [selection]
```

This command lists all module ports found in the selected modules.

If no selection is provided then it lists the ports on the top module.

```

-m
    print verilog blackbox module definitions instead of port lists

```

10.164 prep - generic synthesis script

yosys> help prep

```
prep [options]
```

This command runs a conservative RTL synthesis. A typical application for this is the preparation stage of a verification flow. This command does not operate on partly selected designs.

```

-top <module>
    use the specified module as top module (default='top')

-auto-top
    automatically determine the top of the design hierarchy

-flatten
    flatten the design before synthesis. this will pass '-auto-top' to
    'hierarchy' if no top module is specified.

-ifx
    passed to 'proc'. uses verilog simulation behavior for verilog if/case
    undef handling. this also prevents 'wreduce' from being run.

-memx
    simulate verilog simulation behavior for out-of-bounds memory accesses
    using the 'memory_memx' pass.

-nomem

```

(continues on next page)

(continued from previous page)

```
do not run any of the memory_* passes

-rdff
    call 'memory_dff'. This enables merging of FFs into
    memory read ports.

-nokeepdc
    do not call opt_* with -keepdc

-run <from_label>[:<to_label>]
    only run the commands between the labels (see below). an empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.
```

The following commands are executed by this synthesis command:

```
begin:
    hierarchy -check [-top <top> | -auto-top]

coarse:
    proc [-ifx]
    flatten      (if -flatten)
    future
    opt_expr -keepdc
    opt_clean
    check
    opt -noff -keepdc
    wreduce -keepdc [-memx]
    memory_dff   (if -rdff)
    memory_memx  (if -memx)
    opt_clean
    memory_collect
    opt -noff -keepdc -fast

check:
    stat
    check
```

10.165 printattrs - print attributes of selected objects

yosys> help printattrs

```
printattrs [selection]
```

Print all attributes of the selected objects.

10.166 proc - translate processes to netlists

yosys> help proc

```
proc [options] [selection]
```

This pass calls all the other proc_* passes in the most common order.

```
proc_clean
proc_rmdead
proc_prune
proc_init
proc_arst
proc_rom
proc_mux
proc_dlatch
proc_dff
proc_memwr
proc_clean
opt_expr -keepdc
```

This replaces the processes in the design with multiplexers, flip-flops and latches.

The following options are supported:

```
-nomux
    Will omit the proc_mux pass.

-norom
    Will omit the proc_rom pass.

-global_arst [!]<netname>
    This option is passed through to proc_arst.

-ifx
    This option is passed through to proc_mux. proc_rmdead is not
    executed in -ifx mode.

-noopt
    Will omit the opt_expr pass.
```

10.167 proc_arst - detect asynchronous resets

yosys> help proc_arst

```
proc_arst [-global_arst [!]<netname>] [selection]
```

This pass identifies asynchronous resets in the processes and converts them to a different internal representation that is suitable for generating

(continues on next page)

(continued from previous page)

flip-flop cells with asynchronous resets.

```
-global_arst [!]<netname>
```

In modules that have a net with the given name, use this net as async reset for registers that have been assign initial values in their declaration ('reg foobar = constant_value;'). Use the '!' modifier for active low reset signals. Note: the frontend stores the default value in the 'init' attribute on the net.

10.168 `proc_clean` - remove empty parts of processes

```
yosys> help proc_clean
```

```
proc_clean [options] [selection]
```

```
-quiet
```

do not print any messages.

This pass removes empty parts of processes and ultimately removes a process if it contains only empty structures.

10.169 `proc_dff` - extract flip-flops from processes

```
yosys> help proc_dff
```

```
proc_dff [selection]
```

This pass identifies flip-flops in the processes and converts them to d-type flip-flop cells.

10.170 `proc_dlatch` - extract latches from processes

```
yosys> help proc_dlatch
```

```
proc_dlatch [selection]
```

This pass identifies latches in the processes and converts them to d-type latches.

10.171 proc_init - convert initial block to init attributes

```
yosys> help proc_init
```

```
proc_init [selection]
```

This pass extracts the 'init' actions from processes (generated from Verilog 'initial' blocks) and sets the initial value to the 'init' attribute on the respective wire.

10.172 proc_memwr - extract memory writes from processes

```
yosys> help proc_memwr
```

```
proc_memwr [selection]
```

This pass converts memory writes in processes into \$memwr cells.

10.173 proc_mux - convert decision trees to multiplexers

```
yosys> help proc_mux
```

```
proc_mux [options] [selection]
```

This pass converts the decision trees in processes (originating from if-else and case statements) to trees of multiplexer cells.

```
-ifx
```

Use Verilog simulation behavior with respect to undef values in 'case' expressions and 'if' conditions.

10.174 proc_prune - remove redundant assignments

```
yosys> help proc_prune
```

```
proc_prune [selection]
```

This pass identifies assignments in processes that are always overwritten by a later assignment to the same signal and removes them.

10.175 `proc_rmdead` - eliminate dead trees in decision trees

```
yosys> help proc_rmdead
```

```
proc_rmdead [selection]
```

This pass identifies unreachable branches in decision trees and removes them.

10.176 `proc_rom` - convert switches to ROMs

```
yosys> help proc_rom
```

```
proc_rom [selection]
```

This pass converts switches into read-only memories when appropriate.

10.177 `qbfsat` - solve a 2QBF-SAT problem in the circuit

```
yosys> help qbfsat
```

```
qbfsat [options] [selection]
```

This command solves an "exists-forall" 2QBF-SAT problem defined over the currently selected module. Existentially-quantified variables are declared by assigning a wire "\$anyconst". Universally-quantified variables may be explicitly declared by assigning a wire "\$allconst", but module inputs will be treated as universally-quantified variables by default.

`-nocleanup`

Do not delete temporary files and directories. Useful for debugging.

`-dump-final-smt2 <file>`

Pass the `--dump-smt2` option to `yosys-smtbmc`.

`-assume-outputs`

Add an "\$assume" cell for the conjunction of all one-bit module output wires.

`-assume-negative-polarity`

When adding \$assume cells for one-bit module output wires, assume they are negative polarity signals and should always be low, for example like the miterers created with the ``miter`` command.

`-nooptimize`

Ignore "\minimize" and "\maximize" attributes, do not emit "(maximize)" or "(minimize)" in the SMT-LIBv2, and generally make no attempt to optimize anything.

(continues on next page)

(continued from previous page)

-nobisection

If a wire is marked with the "\minimize" or "\maximize" attribute, do not attempt to optimize that value with the default iterated solving and threshold bisection approach. Instead, have yosys-smtbmc emit a "(minimize)" or "(maximize)" command in the SMT-LIBv2 output and hope that the solver supports optimizing quantified bitvector problems.

-solver <solver>

Use a particular solver. Choose one of: "z3", "yices", "cvc4" and "cvc5". (default: yices)

-solver-option <name> <value>

Set the specified solver option in the SMT-LIBv2 problem file.

-timeout <value>

Set the per-iteration timeout in seconds.
(default: no timeout)

-00, -01, -02

Control the use of ABC to simplify the QBF-SAT problem before solving.

-sat

Generate an error if the solver does not return "sat".

-unsat

Generate an error if the solver does not return "unsat".

-show-smtbmc

Print the output from yosys-smtbmc.

-specialize

If the problem is satisfiable, replace each "\$anyconst" cell with its corresponding constant value from the model produced by the solver.

-specialize-from-file <solution file>

Do not run the solver, but instead only attempt to replace each "\$anyconst" cell in the current module with a constant value provided by the specified file.

-write-solution <solution file>

If the problem is satisfiable, write the corresponding constant value for each "\$anyconst" cell from the model produced by the solver to the specified file.

10.178 ql_bram_merge - Infers QuickLogic k6n10f BRAM pairs that can operate independently

```
yosys> help ql_bram_merge
```

```
ql_bram_merge [selection]
```

This pass identifies k6n10f 18K BRAM cells and packs pairs of them together into a TDP36K cell operating in split mode

10.179 ql_bram_types - Change TDP36K type to subtypes

```
yosys> help ql_bram_types
```

```
ql_bram_types [selection]
```

This pass changes the type of TDP36K cells to different types based on the configuration of the cell.

10.180 ql_dsp_io_regs - change types of QL_DSP2 depending on configuration

```
yosys> help ql_dsp_io_regs
```

```
ql_dsp_io_regs [options] [selection]
```

This pass looks for QL_DSP2 cells and changes their cell type depending on their configuration.

10.181 ql_dsp_macc - infer QuickLogic multiplier-accumulator DSP cells

```
yosys> help ql_dsp_macc
```

```
ql_dsp_macc [selection]
```

This pass looks for a multiply-accumulate pattern based on which it infers a QuickLogic DSP cell.

10.182 ql_dsp_simd - merge QuickLogic K6N10f DSP pairs to operate in SIMD mode

```
yosys> help ql_dsp_simd
```

```
ql_dsp_simd [selection]
```

This pass identifies K6N10f DSP cells with identical configuration and pack pairs of them together into other DSP cells that can perform SIMD operation.

10.183 ql_ioff - Infer I/O FFs for qlf_k6n10f architecture

```
yosys> help ql_ioff
```

```
ql_ioff [selection]
```

This pass promotes qlf_k6n10f registers directly connected to a top-level I/O port to I/O FFs.

10.184 read - load HDL designs

```
yosys> help read
```

```
read {-vlog95|-vlog2k|-sv2005|-sv2009|-sv2012|-sv|-formal} <verilog-file>..
```

Load the specified Verilog/SystemVerilog files. (Full SystemVerilog support is only available via Verific.)

Additional -D<macro>[=<value>] options may be added after the option indicating the language version (and before file names) to set additional verilog defines.

```
read {-liberty} <liberty-file>..
```

Load the specified Liberty files.

```
-lib
    only create empty blackbox modules
```

```
read {-f|-F} <command-file>
```

Load and execute the specified command file. (Requires Verific.)
Check verific command for more information about supported commands in file.

```
read -define <macro>[=<value>]..
```

(continues on next page)

(continued from previous page)

Set global Verilog/SystemVerilog defines.

```
read -undef <macro>..
```

Unset global Verilog/SystemVerilog defines.

```
read -incdir <directory>
```

Add directory to global Verilog/SystemVerilog include directories.

```
read -verific
read -noverific
```

Subsequent calls to 'read' will either use or not use Verific. Calling 'read' with -verific will result in an error on Yosys binaries that are built without Verific support. The default is to use Verific if it is available.

10.185 read_aiger - read AIGER file

yosys> help read_aiger

```
read_aiger [options] [filename]
```

Load module from an AIGER file into the current design.

```
-module_name <module_name>
    name of module to be created (default: <filename>)

-cclk_name <wire_name>
    if specified, AIGER latches to be transformed into $_DFF_P_ cells
    clocked by wire of this name. otherwise, $_FF_ cells will be used

-map <filename>
    read file with port and latch symbols

-wideports
    merge ports that match the pattern 'name[int]' into a single
    multi-bit port 'name'

-xaiger
    read XAIGER extensions
```

10.186 read_blif - read BLIF file

yosys> help read_blif

```
read_blif [options] [filename]
```

Load modules from a BLIF file into the current design.

-sop

Create \$sop cells instead of \$lut cells

-wideports

Merge ports that match the pattern 'name[int]' into a single multi-bit port 'name'.

10.187 read_json - read JSON file

yosys> help read_json

```
read_json [filename]
```

Load modules from a JSON file into the current design See "help write_json" for a description of the file format.

10.188 read_liberty - read cells from liberty file

yosys> help read_liberty

```
read_liberty [filename]
```

Read cells from liberty file as modules into current design.

-lib

only create empty blackbox modules

-wb

mark imported cells as whiteboxes

-nooverwrite

ignore re-definitions of modules. (the default behavior is to create an error message if the existing module is not a blackbox module, and overwrite the existing module if it is a blackbox module.)

-overwrite

overwrite existing modules with the same name

-ignore_miss_func

ignore cells with missing function specification of outputs

(continues on next page)

(continued from previous page)

```
-ignore_miss_dir
    ignore cells with a missing or invalid direction
    specification on a pin

-ignore_miss_data_latch
    ignore latches with missing data and/or enable pins

-ignore_buses
    ignore cells with bus interfaces (wide ports)

-setattr <attribute_name>
    set the specified attribute (to the value 1) on all loaded modules

-unit_delay
    import combinational timing arcs under the unit delay model
```

10.189 read_rtlil - read modules from RTLIL file

```
yosys> help read_rtlil
```

```
read_rtlil [filename]
```

Load modules from an RTLIL file to the current design. (RTLIL is a text representation of a design in yosys's internal format.)

```
-nooverwrite
    ignore re-definitions of modules. (the default behavior is to
    create an error message if the existing module is not a blackbox
    module, and overwrite the existing module if it is a blackbox module.)

-overwrite
    overwrite existing modules with the same name

-lib
    only create empty blackbox modules
```

10.190 read_verilog - read modules from Verilog file

```
yosys> help read_verilog
```

```
read_verilog [options] [filename]
```

Load modules from a Verilog file to the current design. A large subset of Verilog-2005 is supported.

```
-sv
```

(continues on next page)

(continued from previous page)

```

    enable support for SystemVerilog features. (only a small subset
    of SystemVerilog is supported)

-formal
    enable support for SystemVerilog assertions and some Yosys extensions
    replace the implicit -D SYNTHESIS with -D FORMAL

-nosynthesis
    don't add implicit -D SYNTHESIS

-noassert
    ignore assert() statements

-noassume
    ignore assume() statements

-norestrict
    ignore restrict() statements

-assume-asserts
    treat all assert() statements like assume() statements

-assert-assumes
    treat all assume() statements like assert() statements

-nodisplay
    suppress output from display system tasks ($display et. al).
    This does not affect the output from a later 'sim' command.

-debug
    alias for -dump_ast1 -dump_ast2 -dump_vlog1 -dump_vlog2 -yydebug

-dump_ast1
    dump abstract syntax tree (before simplification)

-dump_ast2
    dump abstract syntax tree (after simplification)

-no_dump_ptr
    do not include hex memory addresses in dump (easier to diff dumps)

-dump_vlog1
    dump ast as Verilog code (before simplification)

-dump_vlog2
    dump ast as Verilog code (after simplification)

-dump_rtlil
    dump generated RTLIL netlist

-yydebug
    enable parser debug output

```

(continues on next page)

(continued from previous page)

-nolatches

usually latches are synthesized into logic loops
this option prohibits this and sets the output to 'x'
in what would be the latches hold condition

this behavior can also be achieved by setting the
'nolatches' attribute on the respective module or
always block.

-nomem2reg

under certain conditions memories are converted to registers
early during simplification to ensure correct handling of
complex corner cases. this option disables this behavior.

this can also be achieved by setting the 'nomem2reg'
attribute on the respective module or register.

This is potentially dangerous. Usually the front-end has good
reasons for converting an array to a list of registers.
Prohibiting this step will likely result in incorrect synthesis
results.

-mem2reg

always convert memories to registers. this can also be
achieved by setting the 'mem2reg' attribute on the respective
module or register.

-nomeminit

do not infer \$meminit cells and instead convert initialized
memories to registers directly in the front-end.

-ppdump

dump Verilog code after pre-processor

-nopp

do not run the pre-processor

-nodpi

disable DPI-C support

-noblackbox

do not automatically add a (* blackbox *) attribute to an
empty module.

-lib

only create empty blackbox modules. This implies -DBLACKBOX.
modules with the (* whitebox *) attribute will be preserved.
(* lib_whitebox *) will be treated like (* whitebox *).

-nowb

delete (* whitebox *) and (* lib_whitebox *) attributes from

(continues on next page)

(continued from previous page)

```

    all modules.

-specify
    parse and import specify blocks

-noopt
    don't perform basic optimizations (such as const folding) in the
    high-level front-end.

-icells
    interpret cell types starting with '$' as internal cell types

-pwires
    add a wire for each module parameter

-nooverwrite
    ignore re-definitions of modules. (the default behavior is to
    create an error message if the existing module is not a black box
    module, and overwrite the existing module otherwise.)

-overwrite
    overwrite existing modules with the same name

-defer
    only read the abstract syntax tree and defer actual compilation
    to a later 'hierarchy' command. Useful in cases where the default
    parameters of modules yield invalid or not synthesizable code.

-noautowire
    make the default of `default_nettype be "none" instead of "wire".

-setattr <attribute_name>
    set the specified attribute (to the value 1) on all loaded modules

-Dname[=definition]
    define the preprocessor symbol 'name' and set its optional value
    'definition'

-Idir
    add 'dir' to the directories which are used when searching include
    files

```

The command 'verilog_defaults' can be used to register default options for subsequent calls to 'read_verilog'.

Note that the Verilog frontend does a pretty good job of processing valid verilog input, but has not very good error reporting. It generally is recommended to use a simulator (for example Icarus Verilog) for checking the syntax of the code, rather than to rely on read_verilog for that.

Depending on if read_verilog is run in -formal mode, either the macro SYNTHESIS or FORMAL is defined automatically, unless -nosynthesis is used.

(continues on next page)

(continued from previous page)

In addition, `read_verilog` always defines the macro `YOSYS`.

See the Yosys README file for a list of non-standard Verilog features supported by the Yosys Verilog front-end.

10.191 `read_xaiger2` - (experimental) read XAIGER file

```
yosys> help read_xaiger2
```

```
read_xaiger2 -sc_mapping [options] <filename>
```

Read a standard cell mapping from a XAIGER file into an existing module.

```
-module_name <name>
```

name of the target module

```
-map2 <filename>
```

read file with symbol information

WARNING: THE 'read_xaiger2' COMMAND IS EXPERIMENTAL.

10.192 `recover_names` - Execute a lossy mapping command and recover original netnames

```
yosys> help recover_names
```

```
recover_names [command]
```

This pass executes a lossy mapping command and uses a combination of simulation to find candidate equivalences and SAT to recover exact original net names.

10.193 `rename` - rename object in the design

```
yosys> help rename
```

```
rename old_name new_name
```

Rename the specified object. Note that selection patterns are not supported by this command.

```
rename -output old_name new_name
```

(continues on next page)

(continued from previous page)

Like above, but also make the wire an output. This will fail if the object is not a wire.

```
rename -src [selection]
```

Assign names auto-generated from the src attribute to all selected wires and cells with private names.

```
rename -wire [selection] [-suffix <suffix>]
```

Assign auto-generated names based on the wires they drive to all selected cells with private names. Ignores cells driving privately named wires. By default, the cell is named after the wire with the cell type as suffix. The -suffix option can be used to set the suffix to the given string instead.

```
rename -enumerate [-pattern <pattern>] [selection]
```

Assign short auto-generated names to all selected wires and cells with private names. The -pattern option can be used to set the pattern for the new names. The character % in the pattern is replaced with a integer number. The default pattern is '%_-'.

```
rename -witness
```

Assigns auto-generated names to all \$any*/\$all* output wires and containing cells that do not have a public name. This ensures that, during formal verification, a solver-found trace can be fully specified using a public hierarchical names.

```
rename -hide [selection]
```

Assign private names (the ones with \$-prefix) to all selected wires and cells with public names. This ignores all selected ports.

```
rename -top new_name
```

Rename top module.

```
rename -scramble-name [-seed <seed>] [selection]
```

Assign randomly-generated names to all selected wires and cells. The seed option can be used to change the random number generator seed from the default, but it must be non-zero.

10.194 `rmports` - remove module ports with no connections

```
yosys> help rmports
```

```
rmports [selection]
```

This pass identifies ports in the selected modules which are not used or driven and removes them.

10.195 `sat` - solve a SAT problem in the circuit

```
yosys> help sat
```

```
sat [options] [selection]
```

This command solves a SAT problem defined over the currently selected circuit and additional constraints passed as parameters.

`-all`

show all solutions to the problem (this can grow exponentially, use `-max <N>` instead to get `<N>` solutions)

`-max <N>`

like `-all`, but limit number of solutions to `<N>`

`-enable_undef`

enable modeling of undef value (aka 'x-bits')
this option is implied by `-set-def`, `-set-undef` et. cetera

`-max_undef`

maximize the number of undef bits in solutions, giving a better picture of which input bits are actually vital to the solution.

`-set <signal> <value>`

set the specified signal to the specified value.

`-set-def <signal>`

add a constraint that all bits of the given signal must be defined

`-set-any-undef <signal>`

add a constraint that at least one bit of the given signal is undefined

`-set-all-undef <signal>`

add a constraint that all bits of the given signal are undefined

`-set-def-inputs`

add `-set-def` constraints for all module inputs

`-set-def-formal`

add `-set-def` constraints for formal `$anyinit`, `$anyconst`, `$anyseq` cells

(continues on next page)

(continued from previous page)

```

-show <signal>
    show the model for the specified signal. if no -show option is
    passed then a set of signals to be shown is automatically selected.

-show-inputs, -show-outputs, -show-ports
    add all module (input/output) ports to the list of shown signals

-show-regs, -show-public, -show-all
    show all registers, show signals with 'public' names, show all signals

-ignore_div_by_zero
    ignore all solutions that involve a division by zero

-ignore_unknown_cells
    ignore all cells that can not be matched to a SAT model

```

The following options can be used to set up a sequential problem:

```

-seq <N>
    set up a sequential problem with <N> time steps. The steps will
    be numbered from 1 to N.

    note: for large <N> it can be significantly faster to use
    -tempinduct-baseonly -maxsteps <N> instead of -seq <N>.

-set-at <N> <signal> <value>
-unset-at <N> <signal>
    set or unset the specified signal to the specified value in the
    given timestep. this has priority over a -set for the same signal.

-set-assumes
    set all assumptions provided via $assume cells

-set-def-at <N> <signal>
-set-any-undef-at <N> <signal>
-set-all-undef-at <N> <signal>
    add undef constraints in the given timestep.

-set-init <signal> <value>
    set the initial value for the register driving the signal to the value

-set-init-undef
    set all initial states (not set using -set-init) to undef

-set-init-def
    do not force a value for the initial state but do not allow undef

-set-init-zero
    set all initial states (not set using -set-init) to zero

-dump_vcd <vcd-file-name>

```

(continues on next page)

(continued from previous page)

```

    dump SAT model (counter example in proof) to VCD file

-dump_json <json-file-name>
    dump SAT model (counter example in proof) to a WaveJSON file.

-dump_cnf <cnf-file-name>
    dump CNF of SAT problem (in DIMACS format). in temporal induction
    proofs this is the CNF of the first induction step.

```

The following additional options can be used to set up a proof. If also `-seq` is passed, a temporal induction proof is performed.

```

-tempinduct
    Perform a temporal induction proof. In a temporal induction proof it is
    proven that the condition holds forever after the number of time steps
    specified using -seq.

-tempinduct-def
    Perform a temporal induction proof. Assume an initial state with all
    registers set to defined values for the induction step.

-tempinduct-baseonly
    Run only the basecase half of temporal induction (requires -maxsteps)

-tempinduct-inductonly
    Run only the induction half of temporal induction

-tempinduct-skip <N>
    Skip the first <N> steps of the induction proof.

    note: this will assume that the base case holds for <N> steps.
    this must be proven independently with "-tempinduct-baseonly
    -maxsteps <N>". Use -initsteps if you just want to set a
    minimal induction length.

-prove <signal> <value>
    Attempt to proof that <signal> is always <value>.

-prove-x <signal> <value>
    Like -prove, but an undef (x) bit in the lhs matches any value on
    the right hand side. Useful for equivalence checking.

-prove-asserts
    Prove that all asserts in the design hold.

-prove-skip <N>
    Do not enforce the prove-condition for the first <N> time steps.

-maxsteps <N>
    Set a maximum length for the induction.

-initsteps <N>

```

(continues on next page)

(continued from previous page)

```

    Set initial length for the induction.
    This will speed up the search of the right induction length
    for deep induction proofs.

    -stepsize <N>
        Increase the size of the induction proof in steps of <N>.
        This will speed up the search of the right induction length
        for deep induction proofs.

    -timeout <N>
        Maximum number of seconds a single SAT instance may take.

    -verify
        Return an error and stop the synthesis script if the proof fails.

    -verify-no-timeout
        Like -verify but do not return an error for timeouts.

    -falsify
        Return an error and stop the synthesis script if the proof succeeds.

    -falsify-no-timeout
        Like -falsify but do not return an error for timeouts.

```

10.196 scatter - add additional intermediate nets

```
yosys> help scatter
```

```
scatter [selection]
```

This command adds additional intermediate nets on all cell ports. This is used for testing the correct use of the SigMap helper in passes. If you don't know what this means: don't worry -- you only need this pass when testing your own extensions to Yosys.

Use the `opt_clean` command to get rid of the additional nets.

10.197 scc - detect strongly connected components (logic loops)

```
yosys> help scc
```

```
scc [options] [selection]
```

This command identifies strongly connected components (aka logic loops) in the design.

```
-expect <num>
```

(continues on next page)

(continued from previous page)

```
expect to find exactly <num> SCCs. A different number of SCCs will
produce an error.

-max_depth <num>
    limit to loops not longer than the specified number of cells. This
    can e.g. be useful in identifying small local loops in a module that
    implements one large SCC.

-nofeedback
    do not count cells that have their output fed back into one of their
    inputs as single-cell scc.

-all_cell_types
    Usually this command only considers internal non-memory cells. With
    this option set, all cells are considered. For unknown cells all ports
    are assumed to be bidirectional 'inout' ports.

-set_attr <name> <value>
    set the specified attribute on all cells that are part of a logic
    loop. the special token {} in the value is replaced with a unique
    identifier for the logic loop.

-select
    replace the current selection with a selection of all cells and wires
    that are part of a found logic loop

-specify
    examine specify rules to detect logic loops in whitebox/blackbox cells
```

10.198 scratchpad - get/set values in the scratchpad

```
yosys> help scratchpad
```

```
scratchpad [options]
```

This pass allows to read and modify values from the scratchpad of the current design. Options:

```
-get <identifier>
    print the value saved in the scratchpad under the given identifier.

-set <identifier> <value>
    save the given value in the scratchpad under the given identifier.

-unset <identifier>
    remove the entry for the given identifier from the scratchpad.

-copy <identifier_from> <identifier_to>
    copy the value of the first identifier to the second identifier.
```

(continues on next page)

(continued from previous page)

```
-assert <identifier> <value>
    assert that the entry for the given identifier is set to the given
    value.

-assert-set <identifier>
    assert that the entry for the given identifier exists.

-assert-unset <identifier>
    assert that the entry for the given identifier does not exist.
```

The identifier may not contain whitespace. By convention, it is usually prefixed by the name of the pass that uses it, e.g. 'opt.did_something'. If the value contains whitespace, it must be enclosed in double quotes.

10.199 script - execute commands from file or wire

yosys> help script

```
script <filename> [<from_label>:<to_label>]
script -scriptwire [selection]
```

This command executes the yosys commands in the specified file (default behaviour), or commands embedded in the constant text value connected to the selected wires.

In the default (file) case, the 2nd argument can be used to only execute the section of the file between the specified labels. An empty from label is synonymous with the beginning of the file and an empty to label is synonymous with the end of the file.

If only one label is specified (without ':') then only the block marked with that label (until the next label) is executed.

In "-scriptwire" mode, the commands on the selected wire(s) will be executed in the scope of (and thus, relative to) the wires' owning module(s). This '-module' mode can be exited by using the 'cd' command.

10.200 select - modify and view the list of selected objects

yosys> help select

```
select [ -add | -del | -set <name> ] {-read <filename> | <selection>}
select [ -unset <name> ]
select [ <assert_option> ] {-read <filename> | <selection>}
select [ -list | -list-mod | -write <filename> | -count | -clear ]
select -module <modname>
```

(continues on next page)

(continued from previous page)

Most commands use the list of currently selected objects to determine which part of the design to operate on. This command can be used to modify and view this list of selected objects.

Note that many commands support an optional [selection] argument that can be used to override the global selection for the command. The syntax of this optional argument is identical to the syntax of the <selection> argument described here.

```
-add, -del
    add or remove the given objects to the current selection.
    without this options the current selection is replaced.

-set <name>
    do not modify the current selection. instead save the new selection
    under the given name (see @<name> below). to save the current selection,
    use "select -set <name> %"

-unset <name>
    do not modify the current selection. instead remove a previously saved
    selection under the given name (see @<name> below).

-assert-none
    do not modify the current selection. instead assert that the given
    selection is empty. i.e. produce an error if any object or module
    matching the selection is found.

-assert-any
    do not modify the current selection. instead assert that the given
    selection is non-empty. i.e. produce an error if no object or module
    matching the selection is found.

-assert-mod-count N
    do not modify the current selection. instead assert that the given
    selection contains exactly N modules (partially or fully selected).

-assert-count N
    do not modify the current selection. instead assert that the given
    selection contains exactly N objects.

-assert-max N
    do not modify the current selection. instead assert that the given
    selection contains less than or exactly N objects.

-assert-min N
    do not modify the current selection. instead assert that the given
    selection contains at least N objects.

-list
    list all objects in the current selection

-write <filename>
```

(continues on next page)

(continued from previous page)

```

    like -list but write the output to the specified file

-read <filename>
    read the specified file (written by -write)

-count
    count all objects in the current selection

-clear
    clear the current selection. this effectively selects the whole
    design. it also resets the selected module (see -module). use the
    command 'select *' to select everything but stay in the current module.

-none
    create an empty selection. the current module is unchanged.

-module <modname>
    limit the current scope to the specified module.
    the difference between this and simply selecting the module
    is that all object names are interpreted relative to this
    module after this command until the selection is cleared again.

```

When this command is called without an argument, the current selection is displayed in a compact form (i.e. only the module name when a whole module is selected).

The <selection> argument itself is a series of commands for a simple stack machine. Each element on the stack represents a set of selected objects. After this commands have been executed, the union of all remaining sets on the stack is computed and used as selection for the command.

Pushing (selecting) object when not in -module mode:

```

<mod_pattern>
    select the specified module(s)

<mod_pattern>/<obj_pattern>
    select the specified object(s) from the module(s)

```

Pushing (selecting) object when in -module mode:

```

<obj_pattern>
    select the specified object(s) from the current module

```

By default, patterns will not match black/white-box modules or their contents. To include such objects, prefix the pattern with '='.

A <mod_pattern> can be a module name, wildcard expression (*, ?, [...]) matching module names, or one of the following:

```

A:<pattern>, A:<pattern>=<pattern>
    all modules with an attribute matching the given pattern

```

(continues on next page)

(continued from previous page)

in addition to = also <, <=, >=, and > are supported

N:<pattern>
all modules with a name matching the given pattern
(i.e. 'N:' is optional as it is the default matching rule)

An <obj_pattern> can be an object name, wildcard expression, or one of the following:

w:<pattern>
all wires with a name matching the given wildcard pattern

i:<pattern>, o:<pattern>, x:<pattern>
all inputs (i:), outputs (o:) or any ports (x:) with matching names

s:<size>, s:<min>:<max>
all wires with a matching width

m:<pattern>
all memories with a name matching the given pattern

c:<pattern>
all cells with a name matching the given pattern

t:<pattern>
all cells with a type matching the given pattern

t:@<name>
all cells with a type matching a module in the saved selection <name>

p:<pattern>
all processes with a name matching the given pattern

a:<pattern>
all objects with an attribute name matching the given pattern

a:<pattern>=<pattern>
all objects with a matching attribute name-value-pair.
in addition to = also <, <=, >=, and > are supported

r:<pattern>, r:<pattern>=<pattern>
cells with matching parameters. also with <, <=, >= and >.

n:<pattern>
all objects with a name matching the given pattern
(i.e. 'n:' is optional as it is the default matching rule)

@<name>
push the selection saved prior with 'select -set <name> ...'

The following actions can be performed on the top sets on the stack:

(continues on next page)

(continued from previous page)

```

%
    push a copy of the current selection to the stack

%%
    replace the stack with a union of all elements on it

%n
    replace top set with its invert

%u
    replace the two top sets on the stack with their union

%i
    replace the two top sets on the stack with their intersection

%d
    pop the top set from the stack and subtract it from the new top

%D
    like %d but swap the roles of two top sets on the stack

%c
    create a copy of the top set from the stack and push it

%x[<num1>|*][.<num2>][:<rule>[:<rule>..]]
    expand top set <num1> num times according to the specified rules.
    (i.e. select all cells connected to selected wires and select all
    wires connected to selected cells) The rules specify which cell
    ports to use for this. the syntax for a rule is a '-' for exclusion
    and a '+' for inclusion, followed by an optional comma separated
    list of cell types followed by an optional comma separated list of
    cell ports in square brackets. a rule can also be just a cell or wire
    name that limits the expansion (is included but does not go beyond).
    select at most <num2> objects. a warning message is printed when this
    limit is reached. When '*' is used instead of <num1> then the process
    is repeated until no further object are selected.

%ci[<num1>|*][.<num2>][:<rule>[:<rule>..]]
%co[<num1>|*][.<num2>][:<rule>[:<rule>..]]
    similar to %x, but only select input (%ci) or output cones (%co)

%xe[...] %cie[...] %coe
    like %x, %ci, and %co but only consider combinatorial cells

%a
    expand top set by selecting all wires that are (at least in part)
    aliases for selected wires.

%s
    expand top set by adding all modules that implement cells in selected
    modules

```

(continues on next page)

(continued from previous page)

```
%m
    expand top set by selecting all modules that contain selected objects

%M
    select modules that implement selected cells

%C
    select cells that implement selected modules

%R[<num>]
    select <num> random objects from top selection (default 1)
```

Example: the following command selects all wires that are connected to a 'GATE' input of a 'SWITCH' cell:

```
select */t:SWITCH %x:+[GATE] */t:SWITCH %d
```

10.201 setattr - set/unset attributes on objects

```
yosys> help setattr
```

```
setattr [ -mod ] [ -set name value | -unset name ]... [selection]
```

Set/unset the given attributes on the selected objects. String values must be passed in double quotes (").

When called with -mod, this command will set and unset attributes on modules instead of objects within modules.

10.202 setenv - set an environment variable

```
yosys> help setenv
```

```
setenv name value
```

Set the given environment variable on the current process. Values containing whitespace must be passed in double quotes (").

10.203 setparam - set/unset parameters on objects

yosys> help setparam

```
setparam [ -type cell_type ] [ -set name value | -unset name ]... [selection]
```

Set/unset the given parameters on the selected cells. String values must be passed in double quotes (").

The -type option can be used to change the cell type of the selected cells.

10.204 setundef - replace undef values with defined constants

yosys> help setundef

```
setundef [options] [selection]
```

This command replaces undef (x) constants with defined (0/1) constants.

-undriven

also set undriven nets to constant values

-expose

also expose undriven nets as inputs (use with -undriven)

-zero

replace with bits cleared (0)

-one

replace with bits set (1)

-undef

replace with undef (x) bits, may be used with -undriven

-anyseq

replace with \$anyseq drivers (for formal)

-anyconst

replace with \$anyconst drivers (for formal)

-random <seed>

replace with random bits using the specified integer as seed value for the random number generator.

-init

also create/update init values for flip-flops

-params

replace undef in cell parameters

10.205 share - perform sat-based resource sharing

yosys> help share

```
share [options] [selection]
```

This pass merges shareable resources into a single resource. A SAT solver is used to determine if two resources are share-able.

-force

Per default the selection of cells that is considered for sharing is narrowed using a list of cell types. With this option all selected cells are considered for resource sharing.

IMPORTANT NOTE: If the -all option is used then no cells with internal state must be selected!

-aggressive

Per default some heuristics are used to reduce the number of cells considered for resource sharing to only large resources. This options turns this heuristics off, resulting in much more cells being considered for resource sharing.

-fast

Only consider the simple part of the control logic in SAT solving, resulting in much easier SAT problems at the cost of maybe missing some opportunities for resource sharing.

-limit N

Only perform the first N merges, then stop. This is useful for debugging.

10.206 shell - enter interactive command mode

yosys> help shell

```
shell
```

This command enters the interactive command mode. This can be useful in a script to interrupt the script at a certain point and allow for interactive inspection or manual synthesis of the design at this point.

The command prompt of the interactive shell indicates the current selection (see 'help select'):

```
yosys>
  the entire design is selected
```

```
yosys*>
  only part of the design is selected
```

(continues on next page)

(continued from previous page)

```
yosys [modname]>
    the entire module 'modname' is selected using 'select -module modname'

yosys [modname]*>
    only part of current module 'modname' is selected
```

When in interactive shell, some errors (e.g. invalid command arguments) do not terminate yosys but return to the command prompt.

This command is the default action if nothing else has been specified on the command line.

Press Ctrl-D or type 'exit' to leave the interactive shell.

10.207 show - generate schematics using graphviz

```
yosys> help show
```

```
show [options] [selection]
```

Create a graphviz DOT file for the selected part of the design and compile it to a graphics file (usually SVG or PostScript).

```
-viewer <viewer>
```

Run the specified command with the graphics file as parameter.
On Windows, this pauses yosys until the viewer exits.
Use "-viewer none" to not run any command.

```
-format <format>
```

Generate a graphics file in the specified format. Use 'dot' to just generate a .dot file, or other <format> strings such as 'svg' or 'ps' to generate files in other formats (this calls the 'dot' command).

```
-lib <verilog_or_rtlil_file>
```

Use the specified library file for determining whether cell ports are inputs or outputs. This option can be used multiple times to specify more than one library.

note: in most cases it is better to load the library before calling show with 'read_verilog -lib <filename>'. it is also possible to load liberty files with 'read_liberty -lib <filename>'.
load liberty files with 'read_liberty -lib <filename>'.

```
-prefix <prefix>
```

generate <prefix>.* instead of ~/.yosys_show.*

```
-color <color> <object>
```

assign the specified color to the specified object. The object can be a single selection wildcard expressions or a saved set of objects in the @<name> syntax (see "help select" for details).

(continues on next page)

(continued from previous page)

`-label <text> <object>`
assign the specified label text to the specified object. The object can be a single selection wildcard expressions or a saved set of objects in the `@<name>` syntax (see "help select" for details).

`-colors <seed>`
Randomly assign colors to the wires. The integer argument is the seed for the random number generator. Change the seed value if the colored graph still is ambiguous. A seed of zero deactivates the coloring.

`-colorattr <attribute_name>`
Use the specified attribute to assign colors. A unique color is assigned to each unique value of this attribute.

`-width`
annotate buses with a label indicating the width of the bus.

`-signed`
mark ports (A, B) that are declared as signed (using the `[AB]_SIGNED` cell parameter) with an asterisk next to the port name.

`-stretch`
stretch the graph so all inputs are on the left side and all outputs (including inout ports) are on the right side.

`-pause`
wait for the user to press enter to before returning

`-enum`
enumerate objects with internal (`$`-prefixed) names

`-long`
do not abbreviate objects with internal (`$`-prefixed) names

`-notitle`
do not add the module name as graph title to the dot file

`-nobg`
don't run viewer in the background, IE wait for the viewer tool to exit before returning

`-href`
adds href attribute to all items representing cells and wires, using src attribute of origin

When no `<format>` is specified, 'dot' is used. When no `<format>` and `<viewer>` is specified, 'xdot' is used to display the schematic (POSIX systems only).

The generated output files are `'~/yosys_show.dot'` and `'~/yosys_show.<format>'`, unless another prefix is specified using `-prefix <prefix>`.

Yosys on Windows and YosysJS use different defaults: The output is written

(continues on next page)

(continued from previous page)

to 'show.dot' in the current directory and new viewer is launched each time the 'show' command is executed.

10.208 shregmap - map shift registers

yosys> help shregmap

```
shregmap [options] [selection]
```

This pass converts chains of `$_DFF_[NP]_` gates to target specific shift register primitives. The generated shift register will be of type `$_SHREG_DFF_[NP]_` and will use the same interface as the original `$_DFF_*_` cells. The cell parameter 'DEPTH' will contain the depth of the shift register. Use a target-specific 'techmap' map file to convert those cells to the actual target cells.

```
-minlen N
    minimum length of shift register (default = 2)
    (this is the length after -keep_before and -keep_after)

-maxlen N
    maximum length of shift register (default = no limit)
    larger chains will be mapped to multiple shift register instances

-keep_before N
    number of DFFs to keep before the shift register (default = 0)

-keep_after N
    number of DFFs to keep after the shift register (default = 0)

-clkpol pos|neg|any
    limit match to only positive or negative edge clocks. (default = any)

-enpol pos|neg|none|any_or_none|any
    limit match to FFs with the specified enable polarity. (default = none)

-match <cell_type>[:<d_port_name>:<q_port_name>]
    match the specified cells instead of $_DFF_N_ and $_DFF_P_. If
    '[:<d_port_name>:<q_port_name>]' is omitted then 'D' and 'Q' is used
    by default. E.g. the option '-clkpol pos' is just an alias for
    '-match $_DFF_P_', which is an alias for '-match $_DFF_P_:D:Q'.

-params
    instead of encoding the clock and enable polarity in the cell name by
    deriving from the original cell name, simply name all generated cells
    $_SHREG_ and use CLKPOL and ENPOL parameters. An ENPOL value of 2 is
    used to denote cells without enable input. The ENPOL parameter is
    omitted when '-enpol none' (or no -enpol option) is passed.

-zinit
    assume the shift register is automatically zero-initialized, so it
```

(continues on next page)

(continued from previous page)

```
becomes legal to merge zero initialized FFs into the shift register.

-init
  map initialized registers to the shift reg, add an INIT parameter to
  generated cells with the initialization value. (first bit to shift out
  in LSB position)

-tech greenpak4
  map to greenpak4 shift registers.
```

10.209 sim - simulate the circuit

```
yosys> help sim
```

```
sim [options] [top-level]
```

This command simulates the circuit using the given top-level module.

```
-vcd <filename>
  write the simulation results to the given VCD file

-fst <filename>
  write the simulation results to the given FST file

-aiw <filename>
  write the simulation results to an AIGER witness file
  (requires a *.aim file via -map)

-hdlname
  use the hdlname attribute when writing simulation results
  (preserves hierarchy in a flattened design)

-x
  ignore constant x outputs in simulation file.

-date
  include date and full version info in output.

-clock <portname>
  name of top-level clock input

-clockn <portname>
  name of top-level clock input (inverse polarity)

-multiclock
  mark that witness file is multiclock.

-reset <portname>
  name of top-level reset input (active high)
```

(continues on next page)

(continued from previous page)

```
-resethn <portname>
    name of top-level inverted reset input (active low)

-rstlen <integer>
    number of cycles reset should stay active (default: 1)

-zinit
    zero-initialize all uninitialized regs and memories

-timescale <string>
    include the specified timescale declaration in the vcd

-n <integer>
    number of clock cycles to simulate (default: 20)

-noinitstate
    do not activate $initstate cells during the first cycle

-a
    use all nets in VCD/FST operations, not just those with public names

-w
    writeback mode: use final simulation state as new init state

-r <filename>
    read simulation or formal results file
    File formats supported: FST, VCD, AIW, WIT and .yw
    VCD support requires vcd2fst external tool to be present

-append <integer>
    number of extra clock cycles to simulate for a Yosys witness input

-summary <filename>
    write a JSON summary to the given file

-map <filename>
    read file with port and latch symbols, needed for AIGER witness input

-scope <name>
    scope of simulation top model

-at <time>
    sets start and stop time

-start <time>
    start co-simulation in arbitrary time (default 0)

-stop <time>
    stop co-simulation in arbitrary time (default END)

-sim
    simulation with stimulus from FST (default)
```

(continues on next page)

(continued from previous page)

```
-sim-cmp
    co-simulation expect exact match

-sim-gold
    co-simulation, x in simulation can match any value in FST

-sim-gate
    co-simulation, x in FST can match any value in simulation

-assert
    fail the simulation command if, in the course of simulating,
    any of the asserts in the design fail

-fst-noinit
    do not initialize latches and memories from an input FST or VCD file
    (use the initial defined by the design instead)

-q
    disable per-cycle/sample log message

-d
    enable debug output
```

10.210 simplemap - mapping simple coarse-grain cells

```
yosys> help simplemap
```

```
simplemap [selection]
```

This pass maps a small selection of simple coarse-grain cells to yosys gate primitives. The following internal cell types are mapped by this pass:

```
$not, $pos, $and, $or, $xor, $xnor
$reduce_and, $reduce_or, $reduce_xor, $reduce_xnor, $reduce_bool
$logics_not, $logics_and, $logics_or, $mux, $tribuf
$sr, $ff, $dff, $dffe, $dffsr, $dffsre, $adff, $adffe, $aldff, $aldffe, $sdff,
$sdffe, $sdffce, $dlatch, $adlatch, $dlatchsr
```

10.211 splice - create explicit splicing cells

```
yosys> help splice
```

```
splice [options] [selection]
```

This command adds \$slice and \$concat cells to the design to make the splicing of multi-bit signals explicit. This for example is useful for coarse grain

(continues on next page)

(continued from previous page)

synthesis, where dedicated hardware is needed to splice signals.

```
-sel_by_cell
    only select the cell ports to rewire by the cell. if the selection
    contains a cell, than all cell inputs are rewired, if necessary.

-sel_by_wire
    only select the cell ports to rewire by the wire. if the selection
    contains a wire, than all cell ports driven by this wire are wired,
    if necessary.

-sel_any_bit
    it is sufficient if the driver of any bit of a cell port is selected.
    by default all bits must be selected.

-wires
    also add $slice and $concat cells to drive otherwise unused wires.

-no_outputs
    do not rewire selected module outputs.

-port <name>
    only rewire cell ports with the specified name. can be used multiple
    times. implies -no_output.

-no_port <name>
    do not rewire cell ports with the specified name. can be used multiple
    times. can not be combined with -port <name>.
```

By default selected output wires and all cell ports of selected cells driven by selected wires are rewired.

10.212 splitcells - split up multi-bit cells

yosys> help splitcells

```
splitcells [options] [selection]
```

This command splits multi-bit cells into smaller chunks, based on usage of the cell output bits.

This command operates only in cells such as \$or, \$and, and \$mux, that are easily cut into bit-slices.

```
-format char1[char2[char3]]
    the first char is inserted between the cell name and the bit index, the
    second char is appended to the cell name. e.g. -format () creates cell
    names like 'mycell(42)'. the 3rd character is the range separation
    character when creating multi-bit cells. the default is '[]:'.
```

10.213 splitnets - split up multi-bit nets

yosys> help splitnets

```
splitnets [options] [selection]
```

This command splits multi-bit nets into single-bit nets.

```
-format char1[char2[char3]]
```

the first char is inserted between the net name and the bit index, the second char is appended to the netname. e.g. -format () creates net names like 'mysignal(42)'. the 3rd character is the range separation character when creating multi-bit wires. the default is '[]:'.

```
-ports
```

also split module ports. per default only internal signals are split.

```
-driver
```

don't blindly split nets in individual bits. instead look at the driver and split nets so that no driver drives only part of a net.

10.214 sta - perform static timing analysis

yosys> help sta

```
sta [options] [selection]
```

This command performs static timing analysis on the design. (Only considers paths within a single module, so the design must be flattened.)

10.215 stat - print some statistics

yosys> help stat

```
stat [options] [selection]
```

Print some statistics (number of objects) on the selected portion of the design.

```
-top <module>
```

print design hierarchy with this module as top. if the design is fully selected and a module has the 'top' attribute set, this module is used default value for this option.

```
-liberty <liberty_file>
```

use cell area information from the provided liberty file

```
-tech <technology>
```

(continues on next page)

(continued from previous page)

```

print area estimate for the specified technology. Currently supported
values for <technology>: xilinx, cmos

-width
    annotate internal cell types with their word width.
    e.g. $add_8 for an 8 bit wide $add cell.

-json
    output the statistics in a machine-readable JSON format.
    this is output to the console; use "tee" to output to a file.

```

10.216 submod - moving part of a module to a new submodule

```
yosys> help submod
```

```
submod [options] [selection]
```

This pass identifies all cells with the 'submod' attribute and moves them to a newly created module. The value of the attribute is used as name for the cell that replaces the group of cells with the same attribute value.

This pass can be used to create a design hierarchy in flat design. This can be useful for analyzing or reverse-engineering a design.

This pass only operates on completely selected modules with no processes or memories.

```

-copy
    by default the cells are 'moved' from the source module and the source
    module will use an instance of the new module after this command is
    finished. call with -copy to not modify the source module.

-name <name>
    don't use the 'submod' attribute but instead use the selection. only
    objects from one module might be selected. the value of the -name option
    is used as the value of the 'submod' attribute instead.

-hidden
    instead of creating submodule ports with public names, create ports with
    private names so that a subsequent 'flatten; clean' call will restore
    the original module with original public names.

```

10.217 supercover - add hi/lo cover cells for each wire bit

yosys> help supercover

```
supercover [options] [selection]
```

This command adds two cover cells for each bit of each selected wire, one checking for a hi signal level and one checking for lo level.

10.218 synth - generic synthesis script

yosys> help synth

```
synth [options]
```

This command runs the default synthesis script. This command does not operate on partly selected designs.

-top <module>

use the specified module as top module (default='top')

-auto-top

automatically determine the top of the design hierarchy

-flatten

flatten the design before synthesis. this will pass '-auto-top' to 'hierarchy' if no top module is specified.

-encfile <file>

passed to 'fsm_recode' via 'fsm'

-lut <k>

perform synthesis for a k-LUT architecture.

-nofsm

do not run FSM optimization

-noabc

do not run abc (as if yosys was compiled without ABC support)

-booth

run the booth pass to map \$mul to Booth encoded multipliers

-noalumacc

do not run 'alumacc' pass. i.e. keep arithmetic operators in their direct form (\$add, \$sub, etc.).

-nordff

passed to 'memory'. prohibits merging of FFs into memory read ports

(continues on next page)

(continued from previous page)

```

-noshare
    do not run SAT-based resource sharing

-run <from_label>[:<to_label>]
    only run the commands between the labels (see below). an empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-abc9
    use new ABC9 flow (EXPERIMENTAL)

-flowmap
    use FlowMap LUT techmapping instead of ABC

-no-rw-check
    marks all recognized read ports as "return don't-care value on
    read/write collision" (same result as setting the no_rw_check
    attribute on all memories).

-extra-map filename
    source extra rules from the given file to complement the default
    mapping library in the `techmap` step. this option can be
    repeated.

```

The following commands are executed by this synthesis command:

```

begin:
    hierarchy -check [-top <top> | -auto-top]

coarse:
    proc
    flatten      (if -flatten)
    opt_expr
    opt_clean
    check
    opt -nodffe -nosdff
    fsm          (unless -nofsm)
    opt
    wreduce
    peepopt
    opt_clean
    techmap -map +/cmp2lut.v -map +/cmp2lcu.v      (if -lut)
    booth        (if -booth)
    alumacc       (unless -noalumacc)
    share         (unless -noshare)
    opt
    memory -nomap
    opt_clean

fine:
    opt -fast -full
    memory_map

```

(continues on next page)

(continued from previous page)

```

opt -full
techmap                (unless -extra-map)
techmap -map +/techmap.v -map <inject>      (if -extra-map)
techmap -map +/gate2lut.v (if -noabc and -lut)
clean; opt_lut          (if -noabc and -lut)
flowmap -maxlut K       (if -flowmap and -lut)
opt -fast
abc -fast               (unless -noabc, unless -lut)
abc -fast -lut k        (unless -noabc, if -lut)
opt -fast               (unless -noabc)

check:
  hierarchy -check
  stat
  check

```

10.219 synth_achronix - synthesis for Achronix Speedster22i FPGAs.

yosys> help synth_achronix

```
synth_achronix [options]
```

This command runs synthesis for Achronix Speedster eFPGAs. This work is still experimental.

```

-top <module>
  use the specified module as top module (default='top')

-vout <file>
  write the design to the specified Verilog netlist file. writing of an
  output file is omitted if this parameter is not specified.

-run <from_label>:<to_label>
  only run the commands between the labels (see below). an empty
  from label is synonymous to 'begin', and empty to label is
  synonymous to the end of the command list.

-noflatten
  do not flatten design before synthesis

-rtetime
  run 'abc' with '-dff -D 1' options

```

The following commands are executed by this synthesis command:

```

begin:
  read_verilog -sv -lib +/achronix/speedster22i/cells_sim.v
  hierarchy -check -top <top>

```

(continues on next page)

(continued from previous page)

```

flatten:      (unless -noflatten)
    proc
    flatten
    tribuf -logic
    deminout

coarse:
    synth -run coarse

fine:
    opt -fast -mux_undef -undriven -fine -full
    memory_map
    opt -undriven -fine
    opt -fine
    techmap -map +/techmap.v
    opt -full
    clean -purge
    setundef -undriven -zero
    dfflegalize -cell $_DFF_P_ x
    abc -markgroups -dff -D 1      (only if -retime)

map_luts:
    abc -lut 4
    clean

map_cells:
    iopadmap -bits -outpad $__outpad I:0 -inpad $__inpad 0:I
    techmap -map +/achronix/speedster22i/cells_map.v
    clean -purge

check:
    hierarchy -check
    stat
    check -noinit
    blackbox =A:whitebox

vout:
    write_verilog -nodec -attr2comment -defparam -renameprefix syn_ <file-name>

```

10.220 synth_anlogic - synthesis for Anlogic FPGAs

yosys> help synth_anlogic

```
synth_anlogic [options]
```

This command runs synthesis for Anlogic FPGAs.

```

-top <module>
    use the specified module as top module

```

(continues on next page)

(continued from previous page)

```

-edif <file>
    write the design to the specified EDIF file. writing of an output file
    is omitted if this parameter is not specified.

-json <file>
    write the design to the specified JSON file. writing of an output file
    is omitted if this parameter is not specified.

-run <from_label>:<to_label>
    only run the commands between the labels (see below). an empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-noflatten
    do not flatten design before synthesis

-retain
    run 'abc' with '-dff -D 1' options

-nolutram
    do not use EG_LOGIC_DRAM16X4 cells in output netlist

-nobram
    do not use EG_PHY_BRAM or EG_PHY_BRAM32K cells in output netlist

```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -lib +/anlogic/cells_sim.v +/anlogic/eagle_bb.v
    hierarchy -check -top <top>

flatten:    (unless -noflatten)
    proc
    flatten
    tribuf -logic
    deminout

coarse:
    synth -run coarse

map_ram:
    memory_libmap -lib +/anlogic/lutrams.txt -lib +/anlogic/brams.txt [-no-auto-
↪block] [-no-auto-distributed]    (-no-auto-block if -nobram, -no-auto-distributed if -
↪nolutram)
    techmap -map +/anlogic/lutrams_map.v -map +/anlogic/brams_map.v

map_ffram:
    opt -fast -mux_undef -undriven -fine
    memory_map
    opt -undriven -fine

```

(continues on next page)

(continued from previous page)

```

map_gates:
    techmap -map +/techmap.v -map +/anlogic/arith_map.v
    opt -fast
    abc -dff -D 1      (only if -retime)

map_ffs:
    dfflegalize -cell $_DFFE_P??P_r -cell $_SDFFE_P??P_r -cell $_DLATCH_N??_r
    techmap -D NO_LUT -map +/anlogic/cells_map.v
    opt_expr -mux_undef
    simplemap

map_luts:
    abc -lut 4:6
    clean

map_cells:
    techmap -map +/anlogic/cells_map.v
    clean

map_anlogic:
    anlogic_fixcarry
    anlogic_eqn

check:
    hierarchy -check
    stat
    check -noinit
    blackbox =A:whitebox

edif:
    write_edif <file-name>

json:
    write_json <file-name>

```

10.221 synth_coolrunner2 - synthesis for Xilinx Coolrunner-II CPLDs

yosys> help synth_coolrunner2

```
synth_coolrunner2 [options]
```

This command runs synthesis for Coolrunner-II CPLDs. This work is experimental. It is intended to be used with <https://github.com/azonenberg/openfpga> as the place-and-route.

```

-top <module>
    use the specified module as top module (default='top')

-json <file>
    write the design to the specified JSON file. writing of an output file

```

(continues on next page)

(continued from previous page)

```

    is omitted if this parameter is not specified.

-run <from_label>:<to_label>
    only run the commands between the labels (see below). an empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-noflatten
    do not flatten design before synthesis

-retime
    run 'abc' with '-dff -D 1' options

```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -lib +/coolrunner2/cells_sim.v
    hierarchy -check -top <top>

flatten:    (unless -noflatten)
    proc
    flatten
    tribuf -logic

coarse:
    synth -run coarse

fine:
    extract_counter -dir up -allow_arst no
    techmap -map +/coolrunner2/cells_counter_map.v
    clean
    opt -fast -full
    techmap -map +/techmap.v -map +/coolrunner2/cells_latch.v
    opt -fast
    dfflibmap -prepare -liberty +/coolrunner2/xc2_dff.lib

map_tff:
    abc -g AND,XOR
    clean
    extract -map +/coolrunner2/tff_extract.v

map_pla:
    abc -sop -I 40 -P 56
    clean

map_cells:
    dfflibmap -liberty +/coolrunner2/xc2_dff.lib
    dffinit -ff FDCP Q INIT
    dffinit -ff FDCP_N Q INIT
    dffinit -ff FTCP Q INIT
    dffinit -ff FTCP_N Q INIT

```

(continues on next page)

(continued from previous page)

```

dffinit -ff LDCP Q INIT
dffinit -ff LDCP_N Q INIT
coolrunner2_sop
clean
iopadmap -bits -inpad IBUF 0:I -outpad IOBUFE I:IO -inoutpad IOBUFE 0:IO -
↪toutpad IOBUFE E:I:IO -tinoutpad IOBUFE E:0:I:IO
attrmvp -attr src -attr LOC t:IOBUFE n:*
attrmvp -attr src -attr LOC -driven t:IBUF n:*
coolrunner2_fixup
splitnets
clean

check:
  hierarchy -check
  stat
  check -noinit
  blackbox =A:whitebox

json:
  write_json <file-name>

```

10.222 synth_easic - synthesis for eASIC platform

yosys> help synth_easic

```

synth_easic [options]

This command runs synthesis for eASIC platform.

-top <module>
  use the specified module as top module

-vlog <file>
  write the design to the specified structural Verilog file. writing of
  an output file is omitted if this parameter is not specified.

-etools <path>
  set path to the eTools installation. (default=/opt/eTools)

-run <from_label>:<to_label>
  only run the commands between the labels (see below). an empty
  from label is synonymous to 'begin', and empty to label is
  synonymous to the end of the command list.

-noflatten
  do not flatten design before synthesis

-retime
  run 'abc' with '-dff -D 1' options

```

(continues on next page)

(continued from previous page)

The following commands are executed by this synthesis command:

```
begin:
  read_liberty -lib <etools_phys_clk_lib>
  read_liberty -lib <etools_logic_lut_lib>
  hierarchy -check -top <top>

flatten:      (unless -noflatten)
  proc
  flatten

coarse:
  synth -run coarse

fine:
  opt -fast -mux_undef -undriven -fine
  memory_map
  opt -undriven -fine
  techmap
  opt -fast
  abc -dff -D 1      (only if -retime)
  opt_clean         (only if -retime)

map:
  dfflibmap -liberty <etools_phys_clk_lib>
  abc -liberty <etools_logic_lut_lib>
  opt_clean

check:
  hierarchy -check
  stat
  check -noinit
  blackbox =A:whitebox

vlog:
  write_verilog -noexpr -attr2comment <file-name>
```

10.223 synth_ecp5 - synthesis for ECP5 FPGAs

yosys> help synth_ecp5

```
synth_ecp5 [options]
```

This command runs synthesis for ECP5 FPGAs.

```
-top <module>
  use the specified module as top module

-blif <file>
```

(continues on next page)

(continued from previous page)

```

write the design to the specified BLIF file. writing of an output file
is omitted if this parameter is not specified.

-edif <file>
    write the design to the specified EDIF file. writing of an output file
    is omitted if this parameter is not specified.

-json <file>
    write the design to the specified JSON file. writing of an output file
    is omitted if this parameter is not specified.

-run <from_label>:<to_label>
    only run the commands between the labels (see below). an empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-noflatten
    do not flatten design before synthesis

-dff
    run 'abc'/'abc9' with -dff option

-retain
    run 'abc' with '-dff -D 1' options

-noccu2
    do not use CCU2 cells in output netlist

-nodffe
    do not use flipflops with CE in output netlist

-nobram
    do not use block RAM cells in output netlist

-nolutram
    do not use LUT RAM cells in output netlist

-nowidelut
    do not use PFU muxes to implement LUTs larger than LUT4s

-asyncprld
    use async PRLD mode to implement ALDFF (EXPERIMENTAL)

-abc2
    run two passes of 'abc' for slightly improved logic density

-noabc9
    disable use of new ABC9 flow

-vpr
    generate an output netlist (and BLIF file) suitable for VPR
    (this feature is experimental and incomplete)

```

(continues on next page)

(continued from previous page)

```

-iopad
    insert IO buffers

-nodsp
    do not map multipliers to MULT18X18D

-no-rw-check
    marks all recognized read ports as "return don't-care value on
    read/write collision" (same result as setting the no_rw_check
    attribute on all memories).

```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -lib -specify +/ecp5/cells_sim.v +/ecp5/cells_bb.v
    hierarchy -check -top <top>

coarse:
    proc
    flatten
    tribuf -logic
    deminout
    opt_expr
    opt_clean
    check
    opt -nodffe -nosdff
    fsm
    opt
    wreduce
    peepopt
    opt_clean
    share
    techmap -map +/cmp2lut.v -D LUT_WIDTH=4
    opt_expr
    opt_clean
    techmap -map +/mul2dsp.v -map +/ecp5/dsp_map.v -D DSP_A_MAXWIDTH=18 -D DSP_B_
↳ MAXWIDTH=18 -D DSP_A_MINWIDTH=2 -D DSP_B_MINWIDTH=2 -D DSP_NAME=$_MUL18X18  ↳
↳ (unless -nodsp)
    chtype -set $mul t:$_soft_mul      (unless -nodsp)
    alumacc
    opt
    memory -nomap [-no-rw-check]
    opt_clean

map_ram:
    memory_libmap -lib +/ecp5/lutrams.txt -lib +/ecp5/brams.txt [-no-auto-block] [-
↳ no-auto-distributed]      (-no-auto-block if -nobram, -no-auto-distributed if -nolutram)
    techmap -map +/ecp5/lutrams_map.v -map +/ecp5/brams_map.v

map_ffram:

```

(continues on next page)

(continued from previous page)

```

    opt -fast -mux_undef -undriven -fine
    memory_map
    opt -undriven -fine

map_gates:
    techmap -map +/techmap.v -map +/ecp5/arith_map.v
    iopadmap -bits -outpad OB I:0 -inpad IB 0:I -toutpad OBZ ~T:I:0 -tinoutpad BB ~
↪T:0:I:B A:top    (only if '-iopad')
    attrmvcp -attr src -attr LOC t:OB %x:+[0] t:OBZ %x:+[0] t:BB %x:+[B]
    attrmvcp -attr src -attr LOC -driven t:IB %x:+[I]
    opt -fast
    abc -dff -D 1    (only if -retime)

map_ffs:
    opt_clean
    dfflegalize -cell $_DFF_?_ 01 -cell $_DFF_?P?_ r -cell $_SDFF_?P?_ r [-cell $_
↪DFFE_??_ 01 -cell $_DFFE_?P??_ r -cell $_SDFFE_?P??_ r] [-cell $_ALDFF_?P_ x -cell $_
↪ALDFFE_?P?_ x] [-cell $_DLATCH_?_ x]    ($_ALDFF_*_ only if -asyncprld, $_DLATCH_*_
↪only if not -asyncprld, $_*DFFE_* only if not -nodffe)
    zinit -all w:* t:$_DFF_?_ t:$_DFFE_??_ t:$_SDFF*    (only if -abc9 and -dff)
    techmap -D NO_LUT -map +/ecp5/cells_map.v
    opt_expr -undriven -mux_undef
    simplemap
    lattice_gsr
    attrmvcp -copy -attr syn_useioff
    opt_clean

map_luts:
    abc    (only if -abc2)
    techmap -map +/ecp5/latches_map.v    (skip if -asyncprld)
    abc9 -W 300
    clean

map_cells:
    techmap -map +/ecp5/cells_map.v    (skip if -vpr)
    opt_lut_ins -tech lattice
    clean

check:
    autoname
    hierarchy -check
    stat
    check -noinit
    blackbox =A:whitebox

blif:
    opt_clean -purge    (vpr mode)
    write_blif -attr -cname -conn -param <file-name>    (vpr mode)
    write_blif -gates -attr -param <file-name>    (non-vpr mode)

edif:
    write_edif <file-name>

```

(continues on next page)

(continued from previous page)

```
json:
  write_json <file-name>
```

10.224 synth_efinix - synthesis for Efinix FPGAs

```
yosys> help synth_efinix
```

```
synth_efinix [options]
```

This command runs synthesis for Efinix FPGAs.

```
-top <module>
  use the specified module as top module

-edif <file>
  write the design to the specified EDIF file. writing of an output file
  is omitted if this parameter is not specified.

-json <file>
  write the design to the specified JSON file. writing of an output file
  is omitted if this parameter is not specified.

-run <from_label>:<to_label>
  only run the commands between the labels (see below). an empty
  from label is synonymous to 'begin', and empty to label is
  synonymous to the end of the command list.

-noflatten
  do not flatten design before synthesis

-retain
  run 'abc' with '-dff -D 1' options

-nobram
  do not use EFX_RAM_5K cells in output netlist
```

The following commands are executed by this synthesis command:

```
begin:
  read_verilog -lib +/efinix/cells_sim.v
  hierarchy -check -top <top>

flatten:      (unless -noflatten)
  proc
  flatten
  tribuf -logic
  deminout
```

(continues on next page)

(continued from previous page)

```

coarse:
    synth -run coarse

map_ram:
    memory_libmap -lib +/efinix/brams.txt [-no-auto-block]    (-no-auto-block if -
↪ nobram)
    techmap -map +/efinix/brams_map.v

map_ffram:
    opt -fast -mux_undef -undriven -fine
    memory_map
    opt -undriven -fine

map_gates:
    techmap -map +/techmap.v -map +/efinix/arith_map.v
    opt -fast
    abc -dff -D 1      (only if -retime)

map_ffs:
    dfflegalize -cell $_DFFE_????_ 0 -cell $_SDFFE_????_ 0 -cell $_SDFFCE_????_ 0 -
↪ cell $_DLATCH_?_ x
    techmap -D NO_LUT -map +/efinix/cells_map.v
    opt_expr -mux_undef
    simplemap

map_luts:
    abc -lut 4
    clean

map_cells:
    techmap -map +/efinix/cells_map.v
    clean

map_gbuf:
    clkbufmap -buf $__EFX_GBUF 0:I
    techmap -map +/efinix/gbuf_map.v
    efinix_fixcarry
    clean

check:
    hierarchy -check
    stat
    check -noinit
    blackbox =A:whitebox

edif:
    write_edif <file-name>

json:
    write_json <file-name>

```

10.225 synth_fabulous - FABulous synthesis script

yosys> help synth_fabulous

```
synth_fabulous [options]
```

This command runs synthesis for FPGA fabrics generated with FABulous. This command does ↪not operate on partly selected designs.

```
-top <module>
```

use the specified module as top module (default='top')

```
-auto-top
```

automatically determine the top of the design hierarchy

```
-blif <file>
```

write the design to the specified BLIF file. writing of an output file is omitted if this parameter is not specified.

```
-edif <file>
```

write the design to the specified EDIF file. writing of an output file is omitted if this parameter is not specified.

```
-json <file>
```

write the design to the specified JSON file. writing of an output file is omitted if this parameter is not specified.

```
-lut <k>
```

perform synthesis for a k-LUT architecture (default 4).

```
-vpr
```

perform synthesis for the FABulous VPR flow (using slightly different ↪techmapping).

```
-plib <primitive_library.v>
```

use the specified Verilog file as a primitive library.

```
-extra-plib <primitive_library.v>
```

use the specified Verilog file for extra primitives (can be specified multiple times).

```
-extra-map <techamp.v>
```

use the specified Verilog file for extra techmap rules (can be specified multiple times).

```
-encfile <file>
```

passed to 'fsm_recode' via 'fsm'

```
-nofsm
```

do not run FSM optimization

(continues on next page)

(continued from previous page)

```

-noalumacc
    do not run 'alumacc' pass. i.e. keep arithmetic operators in
    their direct form ($add, $sub, etc.).

-carry <none|ha>
    carry mapping style (none, half-adders, ...) default=none

-noregfile
    do not map register files

-iopad
    enable automatic insertion of IO buffers (otherwise a wrapper
    with manually inserted and constrained IO should be used.)

-complex-dff
    enable support for FFs with enable and synchronous SR (must also be
    supported by the target fabric.)

-noflatten
    do not flatten design after elaboration

-nordff
    passed to 'memory'. prohibits merging of FFs into memory read ports

-noshare
    do not run SAT-based resource sharing

-run <from_label>[:<to_label>]
    only run the commands between the labels (see below). an empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-no-rw-check
    marks all recognized read ports as "return don't-care value on
    read/write collision" (same result as setting the no_rw_check
    attribute on all memories).

```

The following commands are executed by this synthesis command:

```

    read_verilog -lib +/fabulous/prims.v
    read_verilog -lib <extra_plib.v>      (for each -extra-plib)

begin:
    hierarchy -check
    proc

flatten:      (unless -noflatten)
    flatten
    tribuf -logic
    deminout

coarse:

```

(continues on next page)

(continued from previous page)

```

tribuf -logic
deminout
opt_expr
opt_clean
check
opt -nodffe -nosdff
fsm          (unless -nofsm)
opt
wreduce
peepopt
opt_clean
techmap -map +/cmp2lut.v -map +/cmp2lcu.v      (if -lut)
alumacc      (unless -noalumacc)
share        (unless -noshare)
opt
memory -nomap
opt_clean

map_ram:      (unless -noregfile)
memory_libmap -lib +/fabulous/ram_regfile.txt
techmap -map +/fabulous/regfile_map.v

map_ffram:
opt -fast -mux_undef -undriven -fine
memory_map
opt -undriven -fine

map_gates:
opt -full
techmap -map +/techmap.v -map +/fabulous/arith_map.v -D ARITH_<carry>
opt -fast

map_iopad:    (if -iopad)
opt -full
iopadmap -bits -outpad $_FABULOUS_OBUF I:PAD -inpad $_FABULOUS_IBUF O:PAD -
→toutpad IO_1_bidirectional_frame_config_pass ~T:I:PAD -tinoutpad IO_1_bidirectional_
→frame_config_pass ~T:O:I:PAD A:top      (skip if '-noiopad')
techmap -map +/fabulous/io_map.v

map_ffs:
dfflegalize -cell $_DFF_P_0 -cell $_DLATCH_?_ x      without -complex-dff
techmap -map +/fabulous/latches_map.v
techmap -map +/fabulous/ff_map.v
techmap -map <extra_map.v>...      (for each -extra-map)
clean

map_luts:
abc -lut 4 -dress
clean

map_cells:
techmap -D LUT_K=4 -map +/fabulous/cells_map.v

```

(continues on next page)

(continued from previous page)

```

    clean

check:
    hierarchy -check
    stat

blif:
    opt_clean -purge
    write_blif -attr -cname -conn -param <file-name>

json:
    write_json <file-name>

```

10.226 synth_gatemate - synthesis for Cologne Chip GateMate FPGAs

yosys> help synth_gatemate

```
synth_gatemate [options]
```

This command runs synthesis for Cologne Chip AG GateMate FPGAs.

```

-top <module>
    use the specified module as top module.

-vlog <file>
    write the design to the specified verilog file. Writing of an output
    file is omitted if this parameter is not specified.

-json <file>
    write the design to the specified JSON file. Writing of an output file
    is omitted if this parameter is not specified.

-run <from_label>:<to_label>
    only run the commands between the labels (see below). An empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-noflatten
    do not flatten design before synthesis.

-nobram
    do not use CC_BRAM_20K or CC_BRAM_40K cells in output netlist.

-noaddf
    do not use CC_ADDF full adder cells in output netlist.

-nomult
    do not use CC_MULT multiplier cells in output netlist.

-nomx8, -nomx4

```

(continues on next page)

(continued from previous page)

```
do not use CC_MX{8,4} multiplexer cells in output netlist.

-luttree
    use new LUT tree mapping approach (EXPERIMENTAL).

-dff
    run 'abc' with -dff option

-rttime
    run 'abc' with '-dff -D 1' options

-noiopad
    disable I/O buffer insertion (useful for hierarchical or
    out-of-context flows).

-noclkbuf
    disable automatic clock buffer insertion.
```

The following commands are executed by this synthesis command:

```
begin:
    read_verilog -lib -specify +/gatemate/cells_sim.v +/gatemate/cells_bb.v
    hierarchy -check -top <top>

prepare:
    proc
    flatten
    tribuf -logic
    deminout
    opt_expr
    opt_clean
    check
    opt -nodffe -nosdff
    fsm
    opt
    wreduce
    peepopt
    opt_clean
    muxpack
    share
    techmap -map +/cmp2lut.v -D LUT_WIDTH=4
    opt_expr
    opt_clean

map_mult:    (skip if '-nomult')
    techmap -map +/gatemate/mul_map.v

coarse:
    alumacc
    opt
    memory -nomap
    opt_clean
```

(continues on next page)

(continued from previous page)

```

map_bram:      (skip if '-nobram')
    memory_libmap -lib +/gatemate/brams.txt
    techmap -map +/gatemate/brams_map.v

map_ffram:
    opt -fast -mux_undef -undriven -fine
    memory_map
    opt -undriven -fine

map_gates:
    techmap -map +/techmap.v -map +/gatemate/arith_map.v
    opt -fast

map_io:      (skip if '-noiopad')
    iopadmap -bits -inpad CC_IBUF Y:I -outpad CC_OBUF A:0 -toutpad CC_TOBUF ~T:A:0 -
    ↪tinoutpad CC_IOBUF ~T:Y:A:IO
    clean

map_regs:
    opt_clean
    dfflegalize -cell $_DFFE_????_ 01 -cell $_DLATCH_???_ 01
    techmap -map +/gatemate/reg_map.v
    opt_expr -mux_undef
    simplemap
    opt_clean

map_muxs:
    muxcover -mux4 -mux8
    opt -full
    simplemap
    techmap -map +/gatemate/mux_map.v

map_luts:
    abc -genlib +/gatemate/lut_tree_cells.genlib (with -luttree)
    techmap -map +/gatemate/lut_tree_map.v (with -luttree)
    gatemate_foldinv (with -luttree)
    techmap -map +/gatemate/inv_map.v (with -luttree)
    abc -dress -lut 4 (without -luttree)
    clean

map_cells:
    techmap -map +/gatemate/lut_map.v
    clean

map_bufg:      (skip if '-noclkbuf')
    clkbufmap -buf CC_BUFG 0:I
    clean

check:
    hierarchy -check
    stat -width

```

(continues on next page)

(continued from previous page)

```
check -noinit
blackbox =A:whitebox

vlog:
  opt_clean -purge
  write_verilog -noattr <file-name>

json:
  write_json <file-name>
```

10.227 synth_gowin - synthesis for Gowin FPGAs

yosys> help synth_gowin

```
synth_gowin [options]
```

This command runs synthesis for Gowin FPGAs. This work is experimental.

```
-top <module>
  use the specified module as top module (default='top')

-vout <file>
  write the design to the specified Verilog netlist file. writing of an
  output file is omitted if this parameter is not specified.

-json <file>
  write the design to the specified JSON netlist file. writing of an
  output file is omitted if this parameter is not specified.
  This disables features not yet supported by nexprn-gowin.

-run <from_label>:<to_label>
  only run the commands between the labels (see below). an empty
  from label is synonymous to 'begin', and empty to label is
  synonymous to the end of the command list.

-nodffe
  do not use flipflops with CE in output netlist

-nobram
  do not use BRAM cells in output netlist

-nolutram
  do not use distributed RAM cells in output netlist

-noflatten
  do not flatten design before synthesis

-retime
  run 'abc' with '-dff -D 1' options
```

(continues on next page)

(continued from previous page)

```

-nowidelut
    do not use muxes to implement LUTs larger than LUT4s

-noiopads
    do not emit IOB at top level ports

-noalu
    do not use ALU cells

-noabc9
    disable use of new ABC9 flow

-no-rw-check
    marks all recognized read ports as "return don't-care value on
    read/write collision" (same result as setting the no_rw_check
    attribute on all memories).

-family <family>
    sets the gowin family to the specified value. The default is 'gwin'.
    The following families are supported:
    'gwin', 'gw2a', 'gw5a'.

```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -specify -lib +/gowin/cells_sim.v
    read_verilog -specify -lib +/gowin/cells_extra_<family>.v
    hierarchy -check -top <top>

flatten:    (unless -noflatten)
    proc
    flatten
    tribuf -logic
    deminout

coarse:
    synth -run coarse [-no-rw-check]

map_ram:
    memory_libmap -lib +/gowin/lutrams.txt -lib +/gowin/brams.txt [-no-auto-block] [-
↪no-auto-distributed]    (-no-auto-block if -nobram, -no-auto-distributed if -nolutram)
    techmap -map +/gowin/lutrams_map.v -map +/gowin/brams_map.v

map_ffram:
    opt -fast -mux_undef -undriven -fine
    memory_map
    opt -undriven -fine

map_gates:
    techmap -map +/techmap.v -map +/gowin/arith_map.v
    opt -fast
    abc -dff -D 1    (only if -retime)

```

(continues on next page)

(continued from previous page)

```

        iopadmap -bits -inpad IBUF 0:I -outpad OBUF I:0 -toutpad TBUF ~OEN:I:0 -
↪tinoutpad IOBUF ~OEN:0:I:IO      (unless -noiopads)

map_ffs:
    opt_clean
    dfflegalize -cell $_DFF?_ 0 -cell $_DFFE?P_ 0 -cell $_SDFF?P?_ r -cell $_
↪SDFFE?P?P_ r -cell $_DFF?P?_ r -cell $_DFFE?P?P_ r
    techmap -map +/gowin/cells_map.v
    opt_expr -mux_undef
    simplemap

map_luts:
    read_verilog -icells -lib -specify +/abc9_model.v
    abc9 -maxlut 8 -W 500
    clean

map_cells:
    techmap -map +/gowin/cells_map.v
    opt_lut_ins -tech gowin
    setundef -undriven -params -zero
    hilomap -singleton -hicell VCC V -locell GND G
    splitnets -ports      (only if -vout used)
    clean
    autoname

check:
    hierarchy -check
    stat
    check -noinit
    blackbox =A:whitebox

vout:
    write_verilog -simple-lhs -decimal -attr2comment -defparam -renameprefix gen
↪<file-name>
    write_json <file-name>

```

10.228 synth_greenpak4 - synthesis for GreenPAK4 FPGAs

yosys> help synth_greenpak4

```
synth_greenpak4 [options]
```

This command runs synthesis for GreenPAK4 FPGAs. This work is experimental. It is intended to be used with <https://github.com/azonenberg/openfpga> as the place-and-route.

```

-top <module>
    use the specified module as top module (default='top')

-part <part>

```

(continues on next page)

(continued from previous page)

```

synthesize for the specified part. Valid values are SLG46140V,
SLG46620V, and SLG46621V (default).

-json <file>
    write the design to the specified JSON file. writing of an output file
    is omitted if this parameter is not specified.

-run <from_label>:<to_label>
    only run the commands between the labels (see below). an empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-noflatten
    do not flatten design before synthesis

-retime
    run 'abc' with '-dff -D 1' options

```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -lib +/greenpak4/cells_sim.v
    hierarchy -check -top <top>

flatten:    (unless -noflatten)
    proc
    flatten
    tribuf -logic

coarse:
    synth -run coarse

fine:
    extract_counter -pout GP_DCMP,GP_DAC -maxwidth 14
    clean
    opt -fast -mux_undef -undriven -fine
    memory_map
    opt -undriven -fine
    techmap -map +/techmap.v -map +/greenpak4/cells_latch.v
    dfflibmap -prepare -liberty +/greenpak4/gp_dff.lib
    opt -fast -noclkinv -noff
    abc -dff -D 1    (only if -retime)

map_luts:
    nlutmap -assert -luts 0,6,8,2    (for -part SLG46140V)
    nlutmap -assert -luts 2,8,16,2   (for -part SLG46620V)
    nlutmap -assert -luts 2,8,16,2   (for -part SLG46621V)
    clean

map_cells:
    shregmap -tech greenpak4

```

(continues on next page)

(continued from previous page)

```

dfflibmap -liberty +/greenpak4/gp_dff.lib
dffinit -ff GP_DFF Q INIT
dffinit -ff GP_DFFR Q INIT
dffinit -ff GP_DFFS Q INIT
dffinit -ff GP_DFFSR Q INIT
iopadmap -bits -inpad GP_IBUF OUT:IN -outpad GP_OBUF IN:OUT -inoutpad GP_OBUF
→OUT:IN -toutpad GP_OBUFT OE:IN:OUT -tinoutpad GP_IOBUF OE:OUT:IN:IO
attrmvp -attr src -attr LOC t:GP_OBUF t:GP_OBUFT t:GP_IOBUF n:*
attrmvp -attr src -attr LOC -driven t:GP_IBUF n:*
techmap -map +/greenpak4/cells_map.v
greenpak4_dffinv
clean

check:
  hierarchy -check
  stat
  check -noinit
  blackbox =A:whitebox

json:
  write_json <file-name>

```

10.229 synth_ice40 - synthesis for iCE40 FPGAs

yosys> help synth_ice40

```
synth_ice40 [options]
```

This command runs synthesis for iCE40 FPGAs.

```

-device < hx | lp | u >
  relevant only for '-abc9' flow, optimise timing for the specified
  device. default: hx

-top <module>
  use the specified module as top module

-blif <file>
  write the design to the specified BLIF file. writing of an output file
  is omitted if this parameter is not specified.

-edif <file>
  write the design to the specified EDIF file. writing of an output file
  is omitted if this parameter is not specified.

-json <file>
  write the design to the specified JSON file. writing of an output file
  is omitted if this parameter is not specified.

-run <from_label>:<to_label>

```

(continues on next page)

(continued from previous page)

```

only run the commands between the labels (see below). an empty
from label is synonymous to 'begin', and empty to label is
synonymous to the end of the command list.

-noflatten
    do not flatten design before synthesis

-dff
    run 'abc'/'abc9' with -dff option

-retime
    run 'abc' with '-dff -D 1' options

-nocarry
    do not use SB_CARRY cells in output netlist

-nodffe
    do not use SB_DFFE* cells in output netlist

-dffe_min_ce_use <min_ce_use>
    do not use SB_DFFE* cells if the resulting CE line would go to less
    than min_ce_use SB_DFFE* in output netlist

-nobram
    do not use SB_RAM40_4K* cells in output netlist

-spram
    enable automatic inference of SB_SPRAM256KA

-dsp
    use iCE40 UltraPlus DSP cells for large arithmetic

-noabc
    use built-in Yosys LUT techmapping instead of abc

-abc2
    run two passes of 'abc' for slightly improved logic density

-vpr
    generate an output netlist (and BLIF file) suitable for VPR
    (this feature is experimental and incomplete)

-noabc9
    disable use of new ABC9 flow

-flowmap
    use FlowMap LUT techmapping instead of abc (EXPERIMENTAL)

-no-rw-check
    marks all recognized read ports as "return don't-care value on
    read/write collision" (same result as setting the no_rw_check
    attribute on all memories).

```

(continues on next page)

(continued from previous page)

The following commands are executed by this synthesis command:

```
begin:
  read_verilog -D ICE40_HX -lib -specify +/ice40/cells_sim.v
  hierarchy -check -top <top>
  proc

flatten:      (unless -noflatten)
  flatten
  tribuf -logic
  deminout

coarse:
  opt_expr
  opt_clean
  check
  opt -nodffe -nosdff
  fsm
  opt
  wreduce
  peepopt
  opt_clean
  share
  techmap -map +/cmp2lut.v -D LUT_WIDTH=4
  opt_expr
  opt_clean
  memory_dff [-no-rw-check]
  wreduce t:$mul
  techmap -map +/mul2dsp.v -map +/ice40/dsp_map.v -D DSP_A_MAXWIDTH=16 -D DSP_B_
↳ MAXWIDTH=16 -D DSP_A_MINWIDTH=2 -D DSP_B_MINWIDTH=2 -D DSP_Y_MINWIDTH=11 -D DSP_NAME=$_
↳ _MUL16X16      (if -dsp)
    select a:mul2dsp                (if -dsp)
    setattr -unset mul2dsp          (if -dsp)
    opt_expr -fine                   (if -dsp)
    wreduce                         (if -dsp)
    select -clear                    (if -dsp)
    ice40_dsp                       (if -dsp)
    chtype -set $mul t:$__soft_mul  (if -dsp)
    alumacc
    opt
    memory -nomap [-no-rw-check]
    opt_clean

map_ram:
  memory_libmap -lib +/ice40/brams.txt -lib +/ice40/spram.txt [-no-auto-huge] [-no-
↳ auto-block]      (-no-auto-huge unless -spram, -no-auto-block if -nobram)
  techmap -map +/ice40/brams_map.v -map +/ice40/spram_map.v
  ice40_braminit

map_ffram:
```

(continues on next page)

(continued from previous page)

```

    opt -fast -mux_undef -undriven -fine
    memory_map
    opt -undriven -fine

map_gates:
    ice40_wrapcarry
    techmap -map +/techmap.v -map +/ice40/arith_map.v
    opt -fast
    abc -dff -D 1      (only if -retime)
    ice40_opt

map_ffs:
    dfflegalize -cell $_DFF_?_ 0 -cell $_DFFE_?P_ 0 -cell $_DFF_?P?_ 0 -cell $_DFFE_?
    ↪P?P_ 0 -cell $_SDFF_?P?_ 0 -cell $_SDFFCE_?P?P_ 0 -cell $_DLATCH_?_ x -mince -1
    techmap -map +/ice40/ff_map.v
    opt_expr -mux_undef
    simplemap
    ice40_opt -full

map_luts:
    abc      (only if -abc2)
    ice40_opt (only if -abc2)
    techmap -map +/ice40/latches_map.v
    simplemap                                (if -noabc or -flowmap)
    techmap -map +/gate2lut.v -D LUT_WIDTH=4 (only if -noabc)
    flowmap -maxlut 4      (only if -flowmap)
    read_verilog -D ICE40_HX -icells -lib -specify +/ice40/abc9_model.v
    abc9 -W 250
    ice40_wrapcarry -unwrap
    techmap -map +/ice40/ff_map.v
    clean
    opt_lut -tech ice40

map_cells:
    techmap -map +/ice40/cells_map.v (skip if -vpr)
    clean

check:
    autoname
    hierarchy -check
    stat
    check -noinit
    blackbox =A:whitebox

blif:
    opt_clean -purge                                (vpr mode)
    write_blif -attr -cname -conn -param <file-name> (vpr mode)
    write_blif -gates -attr -param <file-name>      (non-vpr mode)

edif:
    write_edif <file-name>

```

(continues on next page)

(continued from previous page)

```
json:
  write_json <file-name>
```

10.230 synth_intel - synthesis for Intel (Altera) FPGAs.

```
yosys> help synth_intel
```

```
synth_intel [options]
```

This command runs synthesis for Intel FPGAs.

```
-family <max10 | cyclone10lp | cycloneiv | cycloneive>
  generate the synthesis netlist for the specified family.
  MAX10 is the default target if no family argument specified.
  For Cyclone IV GX devices, use cycloneiv argument; for Cyclone IV E, use
  cycloneive. For Cyclone V and Cyclone 10 GX, use the synth_intel_alm
  backend instead.

-top <module>
  use the specified module as top module (default='top')

-vqm <file>
  write the design to the specified Verilog Quartus Mapping File. Writing
  of an output file is omitted if this parameter is not specified.
  Note that this backend has not been tested and is likely incompatible
  with recent versions of Quartus.

-vpr <file>
  write BLIF files for VPR flow experiments. The synthesized BLIF output
  file is not compatible with the Quartus flow. Writing of an
  output file is omitted if this parameter is not specified.

-run <from_label>:<to_label>
  only run the commands between the labels (see below). an empty
  from label is synonymous to 'begin', and empty to label is
  synonymous to the end of the command list.

-dff
  pass DFFs to ABC to perform sequential logic optimisations
  (EXPERIMENTAL)

-iopads
  use IO pad cells in output netlist

-nobram
  do not use block RAM cells in output netlist

-nodsp
  do not map multipliers to MUL18/MUL9 cells
```

(continues on next page)

(continued from previous page)

```

-noflatten
    do not flatten design before synthesis

-retime
    run 'abc' with '-dff -D 1' options

```

The following commands are executed by this synthesis command:

```

begin:

family:
    read_verilog -sv -lib +/intel/max10/cells_sim.v
    read_verilog -sv -lib +/intel/common/m9k_bb.v
    read_verilog -sv -lib +/intel/common/altp11_bb.v
    hierarchy -check -top <top>

coarse:
    proc
    flatten      (skip if -noflatten)
    tribuf -logic
    deminout
    opt_expr
    opt_clean
    check
    opt -nodffe -nosdff
    fsm
    opt
    wreduce
    peepopt
    opt_clean
    techmap -map +/cmp2lut.v -D LUT_WIDTH=4
    opt_expr
    opt_clean
    techmap -map +mul2dsp.v [...]      (unless -nodsp)
    opt
    memory -nomap
    opt_clean

map_bram:      (skip if -nobram)
    memory_bram -rules +/intel/common/brams_m9k.txt      (if applicable for family)
    techmap -map +/intel/common/brams_map_m9k.v          (if applicable for family)

map_ffram:
    opt -fast -mux_undef -undriven -fine -full
    memory_map
    opt -undriven -fine
    techmap -map +/techmap.v
    opt -full
    clean -purge
    setundef -undriven -zero
    abc -markgroups -dff -D 1      (only if -retime)

```

(continues on next page)

(continued from previous page)

```

map_ffs:
    dfflegalize -cell $_DFFE_PNOP_ 01
    techmap -map +/intel/common/ff_map.v

map_luts:
    abc9 -lut 4 -W 300
    clean
    opt -fast
    autoname
    clean

map_cells:
    iopadmap -bits -outpad $__outpad I:0 -inpad $__inpad 0:I    (if -iopads)
    techmap -map +/intel/max10/cells_map.v
    clean -purge

check:
    hierarchy -check
    stat
    check -noinit
    blackbox =A:whitebox

vqm:
    write_verilog -attr2comment -defparam -nohex -decimal -renameprefix syn_ <file-
↪name>

vpr:
    opt_clean -purge
    write_blif <file-name>

```

WARNING: THE 'synth_intel' COMMAND IS EXPERIMENTAL.

10.231 synth_intel_alm - synthesis for ALM-based Intel (Altera) FPGAs.

yosys> help synth_intel_alm

```

synth_intel_alm [options]

This command runs synthesis for ALM-based Intel FPGAs.

-top <module>
    use the specified module as top module

-family <family>
    target one of:
    "cyclonev"    - Cyclone V (default)

```

(continues on next page)

(continued from previous page)

```

-noflatten
    do not flatten design before synthesis; useful for per-module area
    statistics

-dff
    pass DFFs to ABC to perform sequential logic optimisations
    (EXPERIMENTAL)

-run <from_label>:<to_label>
    only run the commands between the labels (see below). an empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-nolutram
    do not use LUT RAM cells in output netlist

-nobram
    do not use block RAM cells in output netlist

-nodsp
    do not map multipliers to MISTRAL_MUL cells

-noiopad
    do not instantiate IO buffers

-noclkbuf
    do not insert global clock buffers

```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -specify -lib -D <family> +/intel_alm/common/alm_sim.v
    read_verilog -specify -lib -D <family> +/intel_alm/common/dff_sim.v
    read_verilog -specify -lib -D <family> +/intel_alm/common/dsp_sim.v
    read_verilog -specify -lib -D <family> +/intel_alm/common/mem_sim.v
    read_verilog -specify -lib -D <family> +/intel_alm/common/misc_sim.v
    read_verilog -specify -lib -D <family> -icells +/intel_alm/common/abc9_model.v
    read_verilog -lib +/intel/common/altp11_bb.v
    read_verilog -lib +/intel_alm/common/megafunction_bb.v
    hierarchy -check -top <top>

coarse:
    proc
    flatten      (skip if -noflatten)
    tribuf -logic
    deminout
    opt_expr
    opt_clean
    check
    opt -nodffe -nosdff
    fsm
    opt

```

(continues on next page)

(continued from previous page)

```

wreduce
peepopt
opt_clean
share
techmap -map +/cmp2lut.v -D LUT_WIDTH=6
opt_expr
opt_clean
techmap -map +/mul2dsp.v [...]      (unless -nodsp)
alumacc
iopadmap -bits -outpad MISTRAL_OB I:PAD -inpad MISTRAL_IB O:PAD -toutpad MISTRAL_
↪IO OE:O:PAD -tinoutpad MISTRAL_IO OE:O:I:PAD A:top      (unless -noiopad)
techmap -map +/intel_alm/common/arith_alm_map.v -map +/intel_alm/common/dsp_map.v
opt
memory -nomap
opt_clean

map_bram:      (skip if -nobram)
memory_bram -rules +/intel_alm/common/bram_<bram_type>.txt
techmap -map +/intel_alm/common/bram_<bram_type>_map.v

map_lutram:    (skip if -nolutram)
memory_bram -rules +/intel_alm/common/lutram_mlab.txt      (for Cyclone V)

map_ffram:
memory_map
opt -full

map_ffs:
techmap
dfflegalize -cell $_DFFE_PNOP_0 -cell $_SDFFCE_PPOP_0
techmap -map +/intel_alm/common/dff_map.v
opt -full -undriven -mux_undef
clean -purge
clkbufmap -buf MISTRAL_CLKBUF Q:A      (unless -noclkbuf)

map_luts:
techmap -map +/intel_alm/common/abc9_map.v
abc9 [-dff] -maxlut 6 -W 600
techmap -map +/intel_alm/common/abc9_unmap.v
techmap -map +/intel_alm/common/alm_map.v
opt -fast
autoname
clean

check:
hierarchy -check
stat
check
blackbox =A:whitebox

```


10.232 synth_lattice - synthesis for Lattice FPGAs

yosys> help synth_lattice

```
synth_lattice [options]
```

This command runs synthesis for Lattice FPGAs (excluding iCE40 and Nexus).

```
-top <module>
```

use the specified module as top module

```
-family <family>
```

run synthesis for the specified Lattice architecture
generate the synthesis netlist for the specified family.
supported values:

- ecp5: ECP5
- xo2: MachX02
- xo3: MachX03L/LF
- xo3d: MachX03D

```
-edif <file>
```

write the design to the specified EDIF file. writing of an output file
is omitted if this parameter is not specified.

```
-json <file>
```

write the design to the specified JSON file. writing of an output file
is omitted if this parameter is not specified.

```
-run <from_label>:<to_label>
```

only run the commands between the labels (see below). an empty
from label is synonymous to 'begin', and empty to label is
synonymous to the end of the command list.

```
-noflatten
```

do not flatten design before synthesis

```
-dff
```

run 'abc'/'abc9' with -dff option

```
-retime
```

run 'abc' with '-dff -D 1' options

```
-noccu2
```

do not use CCU2 cells in output netlist

```
-nodffe
```

do not use flipflops with CE in output netlist

```
-nobram
```

do not use block RAM cells in output netlist

```
-nolutram
```

(continues on next page)

(continued from previous page)

```

do not use LUT RAM cells in output netlist

-nowidelut
    do not use PFU muxes to implement LUTs larger than LUT4s
    (by default enabled on MachX02/X03/X03D)

-widelut
    force use of PFU muxes to implement LUTs larger than LUT4s

-asyncprld
    use async PRLD mode to implement ALDFF (EXPERIMENTAL)

-abc2
    run two passes of 'abc' for slightly improved logic density

-abc9
    use new ABC9 flow (EXPERIMENTAL)

-iopad
    insert IO buffers

-nodsp
    do not map multipliers to MULT18X18D

-no-rw-check
    marks all recognized read ports as "return don't-care value on
    read/write collision" (same result as setting the no_rw_check
    attribute on all memories).

-cmp2softlogic
    implement constant comparisons in soft logic, do not involve
    hard carry chains

```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -lib -specify +/lattice/cells_sim.v +/lattice/cells_bb.v
    hierarchy -check -top <top>

coarse:
    proc
    flatten
    tribuf -logic
    deminout
    opt_expr
    opt_clean
    check
    opt -nodffe -nosdff
    fsm
    opt
    wreduce

```

(continues on next page)

(continued from previous page)

```

peepopt
opt_clean
share
techmap -map +/cmp2lut.v -D LUT_WIDTH=4
opt_expr
opt_clean
booth      (only if '-family xo3')
alumacc
opt
memory -nomap [-no-rw-check]
opt_clean

map_ram:
memory_libmap -lib +/lattice/lutrams.txt -lib +/lattice/brams.txt [-no-auto-
↪block] [-no-auto-distributed]      (-no-auto-block if -nobram, -no-auto-distributed if -
↪nolutram)
techmap -map +/lattice/lutrams_map.v -map +/lattice/brams_map.v

map_ffram:
opt -fast -mux_undef -undriven -fine
memory_map
opt -undriven -fine -mux_undef

map_gates:
techmap -map +/techmap.v -map +/lattice/arith_map.v
iopadmap -bits -outpad OB I:0 -inpad IB 0:I -toutpad OBZ ~T:I:0 -tinoutpad BB ~
↪T:0:I:B A:top      (only if '-iopad')
attrmvp -attr src -attr LOC t:OB %x:+[0] t:OBZ %x:+[0] t:BB %x:+[B]
attrmvp -attr src -attr LOC -driven t:IB %x:+[I]
opt -fast
abc -dff -D 1      (only if -retime)

map_ffs:
opt_clean
dfflegalize -cell $_DFF?_ 01 -cell $_DFF?P?_ r -cell $_SDFF?P?_ r [-cell $_
↪DFFE??_ 01 -cell $_DFFE?P??_ r -cell $_SDFFE?P??_ r] [-cell $_ALDFF?P_ x -cell $_
↪ALDFFE?P?_ x] [-cell $_DLATCH?_ x]      ($_ALDFF*_ only if -asyncprld, $_DLATCH*_
↪only if not -asyncprld, $_*DFFE*_ only if not -nodffe)
opt_merge
zinit -all w:* t:$_DFF?_ t:$_DFFE??_ t:$_SDFF*      (only if -abc9 and -dff)
techmap -D NO_LUT -map +/lattice/cells_map.v
opt_expr -undriven -mux_undef
simplemap
lattice_gsr
attrmvp -copy -attr syn_useioff
opt_clean

map_luts:
abc      (only if -abc2)
techmap -map +/lattice/latches_map.v      (skip if -asyncprld)
abc -dress -lut 4:7
clean

```

(continues on next page)

(continued from previous page)

```
map_cells:
    techmap -map +/lattice/cells_map.v
    opt_lut_ins -tech lattice
    clean

check:
    autoname
    hierarchy -check
    stat
    check -noinit
    blackbox =A:whitebox

edif:
    write_edif <file-name>

json:
    write_json <file-name>
```

10.233 synth_microchip - synthesis for Microchip FPGAs

yosys> help synth_microchip

```
synth_microchip [options]
```

This command runs synthesis for Microchip FPGAs. This command creates netlists that are compatible with Microchip PolarFire devices.

```
-top <module>
    use the specified module as the top module

-family <family>
    Run synthesis for the specified Microchip architecture.
    Generate the synthesis netlist for the specified family.
    supported values:
    - polarfire: PolarFire

-edif <file>
    Write the design to the specified edif file. Writing of an output file
    is omitted if this parameter is not specified.

-blif <file>
    Write the design to the specified BLIF file. Writing of an output file
    is omitted if this parameter is not specified.

-vlog <file>
    write the design to the specified Verilog file. writing of an output
    file is omitted if this parameter is not specified.

-nobram
    Do not use block RAM cells in output netlist
```

(continues on next page)

(continued from previous page)

```

-nocarry
    Do not use ARI1 cells in output netlist

-nodsp
    Do not use MATH blocks to implement multipliers and associated logic

-noiopad
    Disable I/O buffer insertion (useful for hierarchical or
    out-of-context flows)

-noclkbuf
    Disable automatic clock buffer insertion

-run <from_label>:<to_label>
    Only run the commands between the labels (see below). an empty
    'from_label' is synonymous to 'begin', and empty 'to_label' is
    synonymous to the end of the command list.

-noflatten
    do not flatten design before synthesis

-dff
    Run 'abc'/'abc9' with -dff option

-retime
    Run 'abc' with '-D 1' option to enable flip-flop retiming.
    implies -dff.

-noabc9
    Use classic ABC flow instead of ABC9

-discard-ffinit
    discard FF init value instead of emitting an error

```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -lib -specify +/microchip/cells_sim.v
    hierarchy -check -auto-top

prepare:
    proc
    flatten    (with '-flatten')
    tribuf -logic
    deminout
    opt_expr
    opt_clean
    check
    opt -nodffe -nosdff
    fsm

```

(continues on next page)

(continued from previous page)

```

    opt
    wreduce
    peepopt
    opt_clean

map_dsp:    (skip if '-nodsp')
    memory_dff
    techmap -map +/mul2dsp.v -map +/microchip/{family}_dsp_map.v {options}
    select a:mul2dsp
    setattr -unset mul2dsp
    opt_expr -fine
    wreduce
    select -clear
    microchip_dsp -family <family>
    chtype -set $mul t:$__soft_mul

coarse:
    techmap -map +/cmp2lut.v -map +/cmp2lcu.v -D LUT_WIDTH=[4]
    alumacc
    share
    opt
    memory -nomap
    opt_clean
    attrmap -remove init      (only if -discard-ffinit)

map_memory:
    memory_libmap [...]
    techmap -map +/microchip/LSRAM_map.v
    techmap -map +/microchip/uSRAM_map.v

map_ffram:
    opt -fast -full
    memory_map

fine:
    opt -full
    simplemap t:$mux
    simplemap t:$xor
    extract_reduce
    muxcover -nodecode -mux4=220
    techmap -map +/microchip/arith_map.v
    techmap -map +/techmap.v
    opt -fast

map_cells:
    iopadmap -bits -inpad INBUF Y:PAD -outpad OUTBUF D:PAD -toutpad TRIBUFF E:D:PAD -
    ↪tinoutpad BIBUF E:Y:D:PAD      (unless -noiobs)
    techmap -map +/techmap.v -map +/microchip/cells_map.v
    clean

map_ffs:
    dfflegalize -cell $_DFFE_PN?P_ x -cell $_SDFFCE_PN?P_ x -cell $_DLATCH_PN?_ x

```

(continues on next page)

(continued from previous page)

```

↪(Converts FFs to supported types)
    zinit -all w:* t:$_SDFICE_*      ('-dff' only)
    techmap -D NO_LUT -map +/microchip/cells_map.v      ('-abc9' only)

    map_luts:
        opt_expr -mux_undef -noclkinv
        abc -luts 2:2,3,6:5[,10,20] [-dff] [-D 1]      (option for '-nowidelut', '-dff', '-
↪retime')
        clean
        techmap -D NO_LUT -map +/microchip/cells_map.v      (only if not '-abc9')
        techmap -map +/microchip/cells_map.v -D FINAL_MAP -D LUT_WIDTH=[4]
        microchip_dffopt
        clkbufmap -buf CLKINT Y:A -inpad CLKBUF Y:PAD
        clean -purge

    check:
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    edif:
        write_edif -pvector bra

    blif:
        write_blif

    vlog:
        write_verilog <file-name>

```

10.234 synth_nanoxplore - synthesis for NanoXplore FPGAs

yosys> help synth_nanoxplore

```
synth_nanoxplore [options]
```

This command runs synthesis for NanoXplore FPGAs.

```

-top <module>
    use the specified module as top module

-family <family>
    run synthesis for the specified NanoXplore architecture
    generate the synthesis netlist for the specified family.
    supported values:
    - medium: NG-Medium
    - large: NG-Large
    - ultra: NG-Ultra

-json <file>

```

(continues on next page)

(continued from previous page)

```

write the design to the specified JSON file. writing of an output file
is omitted if this parameter is not specified.

-run <from_label>:<to_label>
    only run the commands between the labels (see below). an empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-noflatten
    do not flatten design before synthesis

-abc9
    use new ABC9 flow (EXPERIMENTAL)

-nocy
    do not map adders to CY cells

-nodffe
    do not use flipflops with L in output netlist

-min_ce_use <min_ce_use>
    do not use flip-flops with load signal if the resulting count is less
    than min_ce_use in output netlist

-min_srst_use <min_srst_use>
    do not use flip-flops with async reset signal if the resulting count is less
    than min_srst_use in output netlist

-norfram
    do not use Register File RAM cells in output netlist

-nobram
    do not use block NX_RAM cells in output netlist

-noiopad
    do not insert IO buffers

-no-rw-check
    marks all recognized read ports as "return don't-care value on
    read/write collision" (same result as setting the no_rw_check
    attribute on all memories).
```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -lib -specify +/nanoxplore/cells_sim.v +/nanoxplore/cells_sim.v +/
    ↪nanoxplore/cells_bb.v +/nanoxplore/cells_bb.v
    techmap -map +/nanoxplore/cells_wrap.v
    techmap -map +/nanoxplore/cells_wrap.v
    hierarchy -check -top <top>
```

(continues on next page)

(continued from previous page)

```

coarse:
    proc
    flatten      (skip if -noflatten)
    tribuf -logic
    deminout
    opt_expr
    opt_clean
    check
    opt -nodffe -nosdff
    fsm
    opt
    wreduce
    peepopt
    opt_clean
    share
    techmap -map +/cmp2lut.v -D LUT_WIDTH=4
    opt_expr
    opt_clean
    alumacc
    opt
    memory -nomap [-no-rw-check]
    opt_clean

map_ram:
    memory_libmap -lib +/nanoxplore/rf_rams.txt -lib +/nanoxplore/brams.txt [-no-
↳auto-block] [-no-auto-distributed]      (-no-auto-block if -nobram, -no-auto-distributed_
↳if -norfram)
    techmap -map +/nanoxplore/rf_rams_map.v -map +/nanoxplore/brams_map.v
    techmap -map +/nanoxplore/cells_wrap.v t:NX_RAM*
    techmap -map +/nanoxplore/cells_wrap.v t:NX_XRFB* t:NX_RFB*

map_ffram:
    opt -fast -mux_undef -undriven -fine
    memory_map
    opt -undriven -fine -mux_undef

map_gates:
    techmap -map +/techmap.v -map +/nanoxplore/arith_map.v
    nx_carry
    iopadmap -bits -outpad $__BEYOND_OBUF I:PAD -toutpad $__BEYOND_TOBUF C:I:PAD -
↳inpad $__BEYOND_IBUF O:PAD -tinoutpad $__BEYOND_IOBUF C:O:I:PAD A:top      (skip if '-
↳noiopad')
    techmap -map +/nanoxplore/io_map.v
    opt -fast

map_ffs:
    dfflegalize -cell $_DFF_?_ 01 -cell $_DFF_?P?_ r -cell $_SDFF_?P?_ r [-cell $_
↳DFFE_?P_ 01 -cell $_DFFE_?P?P_ r -cell $_SDFFE_?P?P_ r] -cell $_DLATCH_?_ x -mince 8 -
↳minsrst 8      ($_DFFE_* only if not -nodffe)
    opt_merge
    techmap -map +/nanoxplore/latches_map.v
    techmap -map +/nanoxplore/cells_map.v

```

(continues on next page)

(continued from previous page)

```
opt_expr -undriven -mux_undef
clean -purge

map_luts:
  abc -dress -lut 4
  techmap -map +/nanoxplore/cells_map.v t:$lut
  opt -fast
  clean

check:
  autname
  hierarchy -check
  stat
  check -noinit
  blackbox =A:whitebox
  setundef -zero -undriven

json:
  write_json <file-name>
```

10.235 synth_nexus - synthesis for Lattice Nexus FPGAs

yosys> help synth_nexus

```
synth_nexus [options]
```

This command runs synthesis for Lattice Nexus FPGAs.

```
-top <module>
  use the specified module as top module

-family <device>
  run synthesis for the specified Nexus device
  supported values: lifcl, lfd2nx

-json <file>
  write the design to the specified JSON file. writing of an output file
  is omitted if this parameter is not specified.

-vm <file>
  write the design to the specified structural Verilog file. writing of
  an output file is omitted if this parameter is not specified.

-run <from_label>:<to_label>
  only run the commands between the labels (see below). an empty
  from label is synonymous to 'begin', and empty to label is
  synonymous to the end of the command list.

-noflatten
  do not flatten design before synthesis
```

(continues on next page)

(continued from previous page)

```

-dff
    run 'abc'/'abc9' with -dff option

-retain
    run 'abc' with '-dff -D 1' options

-noccu2
    do not use CCU2 cells in output netlist

-nodffe
    do not use flipflops with CE in output netlist

-nolram
    do not use large RAM cells in output netlist
    note that large RAM must be explicitly requested with a (* lram *)
    attribute on the memory.

-nobram
    do not use block RAM cells in output netlist

-nolutram
    do not use LUT RAM cells in output netlist

-nowidelut
    do not use PFU muxes to implement LUTs larger than LUT4s

-noiopad
    do not insert IO buffers

-nodsp
    do not infer DSP multipliers

-abc9
    use new ABC9 flow (EXPERIMENTAL)

```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -lib -specify +/nexus/cells_sim.v +/nexus/cells_xtra.v
    hierarchy -check -top <top>

coarse:
    proc
    flatten
    tribuf -logic
    deminout
    opt_expr
    opt_clean
    check
    opt -nodffe -nosdff
    fsm

```

(continues on next page)

(continued from previous page)

```

    opt
    wreduce
    peepopt
    opt_clean
    share
    techmap -map +/cmp2lut.v -D LUT_WIDTH=4
    opt_expr
    opt_clean
    techmap -map +/mul2dsp.v [...]      (unless -nodsp)
    techmap -map +/nexus/dsp_map.v      (unless -nodsp)
    alumacc
    opt
    memory -nomap
    opt_clean

    map_ram:
        memory_libmap -lib +/nexus/lutrams.txt -lib +/nexus/brams.txt -lib +/nexus/lrams.
↪txt -no-auto-huge [-no-auto-block] [-no-auto-distributed]      (-no-auto-block if -
↪nobram, -no-auto-distributed if -nolutram)
        techmap -map +/nexus/lutrams_map.v -map +/nexus/brams_map.v -map +/nexus/lrams_
↪map.v

    map_ffram:
        opt -fast -mux_undef -undriven -fine
        memory_map
        opt -undriven -fine

    map_gates:
        techmap -map +/techmap.v -map +/nexus/arith_map.v
        iopadmap -bits -outpad OB I:0 -inpad IB 0:I -toutpad OBZ ~T:I:0 -tinoutpad BB ~
↪T:0:I:B A:top      (skip if '-noiopad')
        opt -fast
        abc -dff -D 1      (only if -retime)

    map_ffs:
        opt_clean
        dfflegalize -cell $_DFF_P_ 01 -cell $_DFF_PP?_ r -cell $_SDFF_PP?_ r -cell $_
↪DLATCH_?_ x [-cell $_DFFE_PP_ 01 -cell $_DFFE_PP?P_ r -cell $_SDFFE_PP?P_ r]      ($_
↪*DFFE_* only if not -nodffe)
        zinit -all w:* t:$_DFF_?_ t:$_DFFE_??_ t:$_SDFF*      (only if -abc9 and -dff
        techmap -D NO_LUT -map +/nexus/cells_map.v
        opt_expr -undriven -mux_undef
        simplemap
        attrmvp -copy -attr syn_useioff
        opt_clean

    map_luts:
        techmap -map +/nexus/latches_map.v
        abc -dress -lut 4:5
        clean

    map_cells:

```

(continues on next page)

(continued from previous page)

```

techmap -map +/nexus/cells_map.v
setundef -zero
hilomap -singleton -hicell VHI Z -locell VLO Z
clean

check:
    autoname
    hierarchy -check
    stat
    check -noinit
    blackbox =A:whitebox

json:
    write_json <file-name>

vm:
    write_verilog <file-name>

```

10.236 synth_quicklogic - Synthesis for QuickLogic FPGAs

yosys> help synth_quicklogic

```

synth_quicklogic [options]
This command runs synthesis for QuickLogic FPGAs

-top <module>
    use the specified module as top module

-family <family>
    run synthesis for the specified QuickLogic architecture
    generate the synthesis netlist for the specified family.
    supported values:
    - pp3: PolarPro 3
    - qlf_k6n10f: K6N10f

-nodsp
    do not use dsp_t1_* to implement multipliers and associated logic
    (qlf_k6n10f only).

-nocarry
    do not use adder_carry cells in output netlist.

-nobram
    do not use block RAM cells in output netlist.

-bramtypes
    Emit specialized BRAM cells for particular address and data width
    configurations.

-blif <file>

```

(continues on next page)

(continued from previous page)

write the design to the specified BLIF file. writing of an output file is omitted if this parameter is not specified.

`-verilog <file>`

write the design to the specified verilog file. writing of an output file is omitted if this parameter is not specified.

`-abc`

use old ABC flow, which has generally worse mapping results but is less likely to have bugs.

The following commands are executed by this synthesis command:

```
begin:
  read_verilog -lib -specify +/quicklogic/common/cells_sim.v +/quicklogic/<family>/
↳ cells_sim.v
  hierarchy -check -top <top>

prepare:
  proc
  flatten
  tribuf -logic                      (for pp3)
  deminout
  opt_expr
  opt_clean
  check
  opt -nodffe -nosdff
  fsm
  opt
  wreduce
  peepopt
  opt_clean
  share

map_dsp:    (for qlf_k6n10f, skip if -nodsp)
  wreduce t:$mul
  ql_dsp_macc
  techmap -map +/mul2dsp.v -D DSP_A_MAXWIDTH=20 -D DSP_B_MAXWIDTH=18 -D DSP_A_
↳ MINWIDTH=11 -D DSP_B_MINWIDTH=10 -D DSP_NAME=$__QL_MUL20X18
  techmap -map +/mul2dsp.v -D DSP_A_MAXWIDTH=10 -D DSP_B_MAXWIDTH=9 -D DSP_A_
↳ MINWIDTH=4 -D DSP_B_MINWIDTH=4 -D DSP_NAME=$__QL_MUL10X9
  chtype -set $mul t:$__soft_mul
  techmap -map +/quicklogic/<family>/dsp_map.v -D USE_DSP_CFG_PARAMS=0
  ql_dsp_simd
  techmap -map +/quicklogic/<family>/dsp_final_map.v
  ql_dsp_io_regs

coarse:
  techmap -map +/cmp2lut.v -D LUT_WIDTH=4
  opt_expr
  opt_clean
  alumacc
```

(continues on next page)

(continued from previous page)

```

pmuxtree
opt
memory -nomap
opt_clean

map_bram:    (for qlf_k6n10f, skip if -no_bram)
memory_libmap -lib +/quicklogic/<family>/libmap_brams.txt
ql_bram_merge
techmap -map +/quicklogic/<family>/libmap_brams_map.v
techmap -autoproc -map +/quicklogic/<family>/brams_map.v
ql_bram_types    (if -bramtypes)

map_ffram:
opt -fast -mux_undef -undriven -fine
memory_map -iattr -attr !ram_block -attr !rom_block -attr logic_block -attr syn_
↪ramstyle=auto -attr syn_ramstyle=registers -attr syn_romstyle=auto -attr syn_
↪romstyle=logic
opt -undriven -fine

map_gates:
techmap
opt -fast
muxcover -mux8 -mux4    (for pp3)

map_ffs:
opt_expr
shregmap -minlen <min> -maxlen <max>    (for qlf_k6n10f)
dfflegalize -cell <supported FF types>
techmap -map +/quicklogic/<family>/cells_map.v    (for pp3)
techmap -map +/quicklogic/<family>/ffs_map.v    (for ql_k6n10f)
opt

map_luts:    (for pp3)
techmap -map +/quicklogic/<family>/latches_map.v
read_verilog -lib -specify -icells +/quicklogic/<family>/abc9_model.v
techmap -map +/quicklogic/<family>/abc9_map.v
abc9 -maxlut 4 -dff
techmap -map +/quicklogic/<family>/abc9_unmap.v
clean

map_luts:    (for qlf_k6n10f)
abc9 -maxlut 6
clean
opt_lut

map_cells:    (for pp3)
techmap -map +/quicklogic/<family>/lut_map.v
clean
opt_lut

iomap:    (for qlf_k6n10f, skip if -noioff)
ql_ioff

```

(continues on next page)

(continued from previous page)

```

    opt_clean

check:
    autaname
    hierarchy -check
    stat
    check -noinit

iomap:    (for pp3)
    clkbufmap -inpad ckpad Q:P
    iopadmap -bits -outpad outpad A:P -inpad inpad Q:P -tinoutpad bipad EN:Q:A:P
↪A:top

finalize:
    setundef -zero -params -undriven    (for pp3)
    opt_clean -purge
    check
    blackbox =A:whitebox

blif:    (if -blif)
    write_blif -attr -param -auto-top

verilog:    (if -verilog)
    write_verilog -noattr -nohex <file-name>

```

10.237 synth_sf2 - synthesis for SmartFusion2 and IGLOO2 FPGAs

yosys> help synth_sf2

```
synth_sf2 [options]
```

This command runs synthesis for SmartFusion2 and IGLOO2 FPGAs.

```

-top <module>
    use the specified module as top module

-edif <file>
    write the design to the specified EDIF file. writing of an output file
    is omitted if this parameter is not specified.

-vlog <file>
    write the design to the specified Verilog file. writing of an output
    file is omitted if this parameter is not specified.

-json <file>
    write the design to the specified JSON file. writing of an output file
    is omitted if this parameter is not specified.

-run <from_label>:<to_label>
    only run the commands between the labels (see below). an empty

```

(continues on next page)

(continued from previous page)

```

    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-noflatten
    do not flatten design before synthesis

-noiobs
    run synthesis in "block mode", i.e. do not insert IO buffers

-clkbuf
    insert direct PAD->global_net buffers

-discard-ffinit
    discard FF init value instead of emitting an error

-retime
    run 'abc' with '-dff -D 1' options

```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -lib +/sf2/cells_sim.v
    hierarchy -check -top <top>

flatten:      (unless -noflatten)
    proc
    flatten
    tribuf -logic
    deminout

coarse:
    attrmap -remove init      (only if -discard-ffinit)
    synth -run coarse

fine:
    opt -fast -mux_undef -undriven -fine
    memory_map
    opt -undriven -fine
    techmap -map +/techmap.v -map +/sf2/arith_map.v
    opt -fast
    abc -dff -D 1      (only if -retime)

map_ffs:
    dfflegalize -cell $_DFFE_PN?P_ x -cell $_SDFFCE_PN?P_ x -cell $_DLATCH_PN?_ x
    techmap -D NO_LUT -map +/sf2/cells_map.v
    opt_expr -mux_undef
    simplemap

map_luts:
    abc -lut 4
    clean

```

(continues on next page)

(continued from previous page)

```

map_cells:
    techmap -map +/sf2/cells_map.v
    clean

map_iobs:
    clkbufmap -buf CLKINT Y:A [-inpad CLKBUF Y:PAD]    (unless -noiobs, -inpad only,
↳passed if -clkbuf)
    iopadmap -bits -inpad INBUF Y:PAD -outpad OUTBUF D:PAD -toutpad TRIBUFF E:D:PAD -
↳tinoutpad BIBUF E:Y:D:PAD    (unless -noiobs)
    clean -purge

check:
    hierarchy -check
    stat
    check -noinit
    blackbox =A:whitebox

edif:
    write_edif -gndvccy <file-name>

vlog:
    write_verilog <file-name>

json:
    write_json <file-name>

```

10.238 synth_xilinx - synthesis for Xilinx FPGAs

yosys> help synth_xilinx

```
synth_xilinx [options]
```

This command runs synthesis for Xilinx FPGAs. This command does not operate on partly selected designs. At the moment this command creates netlists that are compatible with 7-Series Xilinx devices.

```

-top <module>
    use the specified module as top module

-family <family>
    run synthesis for the specified Xilinx architecture
    generate the synthesis netlist for the specified family.
    supported values:
    - xcup: Ultrascale Plus
    - xcu: Ultrascale
    - xc7: Series 7 (default)
    - xc6s: Spartan 6
    - xc6v: Virtex 6
    - xc5v: Virtex 5 (EXPERIMENTAL)

```

(continues on next page)

(continued from previous page)

```

- xc4v: Virtex 4 (EXPERIMENTAL)
- xc3sda: Spartan 3A DSP (EXPERIMENTAL)
- xc3sa: Spartan 3A (EXPERIMENTAL)
- xc3se: Spartan 3E (EXPERIMENTAL)
- xc3s: Spartan 3 (EXPERIMENTAL)
- xc2vp: Virtex 2 Pro (EXPERIMENTAL)
- xc2v: Virtex 2 (EXPERIMENTAL)
- xcve: Virtex E, Spartan 2E (EXPERIMENTAL)
- xcv: Virtex, Spartan 2 (EXPERIMENTAL)

-edif <file>
    write the design to the specified edif file. writing of an output file
    is omitted if this parameter is not specified.

-blif <file>
    write the design to the specified BLIF file. writing of an output file
    is omitted if this parameter is not specified.

-ise
    generate an output netlist suitable for ISE

-nobram
    do not use block RAM cells in output netlist

-nolutram
    do not use distributed RAM cells in output netlist

-nosrl
    do not use distributed SRL cells in output netlist

-nocarry
    do not use XORCY/MUXCY/CARRY4 cells in output netlist

-nowidelut
    do not use MUXF[5-9] resources to implement LUTs larger than native for
    the target

-nodsp
    do not use DSP48*s to implement multipliers and associated logic

-noiopad
    disable I/O buffer insertion (useful for hierarchical or
    out-of-context flows)

-noclkbuf
    disable automatic clock buffer insertion

-uram
    infer URAM288s for large memories (xcup only)

-widemux <int>
    enable inference of hard multiplexer resources (MUXF[78]) for muxes at

```

(continues on next page)

(continued from previous page)

```

    or above this number of inputs (minimum value 2, recommended value >= 5)
    default: 0 (no inference)

-json <file>
    write the design to the specified JSON file. writing of an output file
    is omitted if this parameter is not specified.

-run <from_label>:<to_label>
    only run the commands between the labels (see below). an empty
    from label is synonymous to 'begin', and empty to label is
    synonymous to the end of the command list.

-flatten
    flatten design before synthesis

-dff
    run 'abc'/'abc9' with -dff option

-retain
    run 'abc' with '-D 1' option to enable flip-flop retiming.
    implies -dff.

-abc9
    use new ABC9 flow (EXPERIMENTAL)

```

The following commands are executed by this synthesis command:

```

begin:
    read_verilog -lib -specify +/xilinx/cells_sim.v
    read_verilog -lib +/xilinx/cells_xtra.v
    hierarchy -check -auto-top

prepare:
    proc
    flatten      (with '-flatten')
    tribuf -logic
    deminout
    opt_expr
    opt_clean
    check
    opt -nodffe -nosdff
    fsm
    opt
    wreduce [-keepdc]      (option for '-widemux')
    peepopt
    opt_clean
    muxpack          ('-widemux' only)
    pmux2shiftx      (skip if '-nosrl' and '-widemux=0')
    clean            (skip if '-nosrl' and '-widemux=0')

map_dsp:      (skip if '-nodsp')

```

(continues on next page)

(continued from previous page)

```

memory_dff
techmap -map +/mul2dsp.v -map +/xilinx/{family}_dsp_map.v {options}
select a:mul2dsp
setattr -unset mul2dsp
opt_expr -fine
wreduce
select -clear
xilinx_dsp -family <family>
chtype -set $mul t:$__soft_mul

coarse:
techmap -map +/cmp2lut.v -map +/cmp2lcu.v -D LUT_WIDTH=[46]
alumacc
share
opt
memory -nomap
opt_clean

map_memory:
memory_libmap [...]
techmap -map +/xilinx/lutrams_<family>_map.v
techmap -map +/xilinx/brams_<family>_map.v

map_ffram:
opt -fast -full
memory_map

fine:
simplemap t:$mux    ('-widemux' only)
muxcover <internal options>    ('-widemux' only)
opt -full
xilinx_srl -variable -minlen 3    (skip if '-nosrl')
techmap -map +/techmap.v -D LUT_SIZE=[46] [-map +/xilinx/mux_map.v] -map +/
↪xilinx/arith_map.v
opt -fast

map_cells:
iopadmap -bits -outpad OBUF I:0 -inpad IBUF O:I -toutpad OBUFT ~T:I:0 -tinoutpad ↪
↪IOBUF ~T:O:I:IO A:top    (skip if '-noiopad')
techmap -map +/techmap.v -map +/xilinx/cells_map.v
clean

map_ffs:
dfflegalize -cell $_DFFE_?P?P_ 01 -cell $_SDFFE_?P?P_ 01 -cell $_DLATCH_?P?_ 01 ↪
↪ (for xc6v, xc7, xcu, xcup)
zinit -all w:* t:$_SDFFE_*    ('-dff' only)
techmap -map +/xilinx/ff_map.v    ('-abc9' only)

map_luts:
opt_expr -mux_undef -noclkinv
abc -luts 2:2,3,6:5[,10,20] [-dff] [-D 1]    (option for '-nowidelut', '-dff', '-
↪retime')

```

(continues on next page)

(continued from previous page)

```

clean
techmap -map +/xilinx/ff_map.v      (only if not '-abc9')
xilinx_srl -fixed -minlen 3        (skip if '-nosrl')
techmap -map +/xilinx/lut_map.v -map +/xilinx/cells_map.v -D LUT_WIDTH=[46]
xilinx_dffopt [-lut4]
opt_lut_ins -tech xilinx

finalize:
  clkbufmap -buf BUFG 0:I          (skip if '-noclkbuf')
  extractinv -inv INV 0:I          (only if '-ise')
  clean

check:
  hierarchy -check
  stat -tech xilinx
  check -noinit
  blackbox =A:whitebox

edif:
  write_edif -pvector bra

blif:
  write_blif

json:
  write_json <file-name>

```

10.239 synthprop - synthesize SVA properties

yosys> help synthprop

```
synthprop [options]
```

This creates synthesizable properties for the selected module.

```

-name <portname>
    name of the output port for assertions (default: assertions).

-map <filename>
    write the port mapping for synthesizable properties into the given file.

-or_outputs
    Or all outputs together to create a single output that goes high when
    any property is violated, instead of generating individual output bits.

-reset <portname>
    name of the top-level reset input. Latch a high state on the generated
    outputs until an asynchronous top-level reset input is activated.

```

(continues on next page)

(continued from previous page)

```
-resetn <portname>
    like above but with inverse polarity
```

10.240 tcl - execute a TCL script file

```
yosys> help tcl
```

```
tcl <filename> [args]
```

This command executes the tcl commands in the specified file.
Use 'yosys cmd' to run the yosys command 'cmd' from tcl.

The tcl command 'yosys -import' can be used to import all yosys commands directly as tcl commands to the tcl shell. Yosys commands 'proc' and 'rename' are wrapped to tcl commands 'procs' and 'renames' in order to avoid a name collision with the built in commands.

If any arguments are specified, these arguments are provided to the script via the standard \$argc and \$argv variables.

Note, tcl will not receive the output of any yosys command. If the output of the tcl commands are needed, use the yosys command 'tee -s result.string' to redirect yosys's output to the 'result.string' scratchpad value. The 'result.string' value is then used as the tcl output value of the command.

10.241 techmap - generic technology mapper

```
yosys> help techmap
```

```
techmap [-map filename] [selection]
```

This pass implements a very simple technology mapper that replaces cells in the design with implementations given in form of a Verilog or RTLIL source file.

```
-map filename
    the library of cell implementations to be used.
    without this parameter a builtin library is used that
    transforms the internal RTL cells to the internal gate
    library.
```

```
-map %<design-name>
    like -map above, but with an in-memory design instead of a file.
```

```
-extern
    load the cell implementations as separate modules into the design
    instead of inlining them.
```

(continues on next page)

(continued from previous page)

```

-max_iter <number>
    only run the specified number of iterations on each module.
    default: unlimited

-recursive
    instead of the iterative breadth-first algorithm use a recursive
    depth-first algorithm. both methods should yield equivalent results,
    but may differ in performance.

-autoproc
    Automatically call "proc" on implementations that contain processes.

-wb
    Ignore the 'whitebox' attribute on cell implementations.

-assert
    this option will cause techmap to exit with an error if it can't map
    a selected cell. only cell types that end on an underscore are accepted
    as final cell types by this mode.

-D <define>, -I <incdir>
    this options are passed as-is to the Verilog frontend for loading the
    map file. Note that the Verilog frontend is also called with the
    '-nooverwrite' option set.

-dont_map <celltype>
    leave the given cell type unmapped by ignoring any mapping rules for it

```

When a module in the map file has the 'techmap_celltype' attribute set, it will match cells with a type that match the text value of this attribute. Otherwise the module name will be used to match the cell. Multiple space-separated cell types can be listed, and wildcards using [] will be expanded (ie. "\$_DFF_[PN]_" is the same as "\$_DFF_P_ \$_DFF_N_").

When a module in the map file has the 'techmap_simplemap' attribute set, techmap will use 'simplemap' (see 'help simplemap') to map cells matching the module.

When a module in the map file has the 'techmap_maccmap' attribute set, techmap will use 'maccmap' (see 'help maccmap') to map cells matching the module.

When a module in the map file has the 'techmap_wrap' attribute set, techmap will create a wrapper for the cell and then run the command string that the attribute is set to on the wrapper module.

When a port on a module in the map file has the 'techmap_autopurge' attribute set, and that port is not connected in the instantiation that is mapped, then a cell port connected only to such wires will be omitted in the mapped version of the circuit.

All wires in the modules from the map file matching the pattern _TECHMAP_* or *._TECHMAP_* are special wires that are used to pass instructions from

(continues on next page)

(continued from previous page)

the mapping module to the techmap command. At the moment the following special wires are supported:

`_TECHMAP_FAIL_`

When this wire is set to a non-zero constant value, techmap will not use this module and instead try the next module with a matching 'techmap_celltype' attribute.

When such a wire exists but does not have a constant value after all `_TECHMAP_DO_*` commands have been executed, an error is generated.

`_TECHMAP_DO_*`

This wires are evaluated in alphabetical order. The constant text value of this wire is a yosys command (or sequence of commands) that is run by techmap on the module. A common use case is to run 'proc' on modules that are written using always-statements.

When such a wire has a non-constant value at the time it is to be evaluated, an error is produced. That means it is possible for such a wire to start out as non-constant and evaluate to a constant value during processing of other `_TECHMAP_DO_*` commands.

A `_TECHMAP_DO_*` command may start with the special token 'CONSTMAP; '. In this case techmap will create a copy for each distinct configuration of constant inputs and shorted inputs at this point and import the constant and connected bits into the map module. All further commands are executed in this copy. This is a very convenient way of creating optimized specializations of techmap modules without using the special parameters described below.

A `_TECHMAP_DO_*` command may start with the special token 'RECURSION; '. Then techmap will recursively replace the cells in the module with their implementation. This is not affected by the `-max_iter` option.

It is possible to combine both prefixes to 'RECURSION; CONSTMAP; '.

`_TECHMAP_REMOVEINIT_<port-name>_`

When this wire is set to a constant value, the init attribute of the wire(s) connected to this port will be consumed. This wire must have the same width as the given port, and for every bit that is set to 1 in the value, the corresponding init attribute bit will be changed to 1'bx. If all bits of an init attribute are left as x, it will be removed.

In addition to this special wires, techmap also supports special parameters in modules in the map file:

`_TECHMAP_CELLTYPE_`

When a parameter with this name exists, it will be set to the type name of the cell that matches the module.

`_TECHMAP_CELLNAME_`

When a parameter with this name exists, it will be set to the name

(continues on next page)

(continued from previous page)

of the cell that matches the module.

```
_TECHMAP_CONSTMSK_<port-name>_
_TECHMAP_CONSTVAL_<port-name>_
```

When this pair of parameters is available in a module for a port, then former has a 1-bit for each constant input bit and the latter has the value for this bit. The unused bits of the latter are set to undef (x).

```
_TECHMAP_WIREINIT_<port-name>_
```

When a parameter with this name exists, it will be set to the initial value of the wire(s) connected to the given port, as specified by the init attribute. If the attribute doesn't exist, x will be filled for the missing bits. To remove the init attribute bits used, use the `_TECHMAP_REMOVEINIT_*_ wires`.

```
_TECHMAP_BITS_CONNMAP_
_TECHMAP_CONNMAP_<port-name>_
```

For an N-bit port, the `_TECHMAP_CONNMAP_<port-name>_` parameter, if it exists, will be set to an `N*_TECHMAP_BITS_CONNMAP_` bit vector containing N words (of `_TECHMAP_BITS_CONNMAP_` bits each) that assign each single bit driver a unique id. The values 0-3 are reserved for 0, 1, x, and z. This can be used to detect shorted inputs.

When a module in the map file has a parameter where the according cell in the design has a port, the module from the map file is only used if the port in the design is connected to a constant value. The parameter is then set to the constant value.

A cell with the name `_TECHMAP_REPLACE_` in the map file will inherit the name and attributes of the cell that is being replaced.

A cell with a name of the form ``_TECHMAP_REPLACE_.<suffix>`` in the map file will be named thus but with the ``_TECHMAP_REPLACE_'` prefix substituted with the name of the cell being replaced.

Similarly, a wire named in the form ``_TECHMAP_REPLACE_.<suffix>`` will cause a new wire alias to be created and named as above but with the ``_TECHMAP_REPLACE_'` prefix also substituted.

A cell with the type `_TECHMAP_PLACEHOLDER_` in the map file will have its type changed to the content of the `techmap_chtype` attribute. This allows for choosing the cell type dynamically.

See 'help extract' for a pass that does the opposite thing.

See 'help flatten' for a pass that does flatten the design (which is essentially techmap but using the design itself as map library).

10.242 tee - redirect command output to file

yosys> help tee

```
tee [-q] [-o logfile|-a logfile] cmd
```

Execute the specified command, optionally writing the commands output to the specified logfile(s).

```
-q
    Do not print output to the normal destination (console and/or log file).

-o logfile
    Write output to this file, truncate if exists.

-a logfile
    Write output to this file, append if exists.

-s scratchpad
    Write output to this scratchpad value, truncate if it exists.

+INT, -INT
    Add/subtract INT from the -v setting for this command.
```

10.243 test_abcloop - automatically test handling of loops in abc command

yosys> help test_abcloop

```
test_abcloop [options]
```

Test handling of logic loops in ABC.

```
-n {integer}
    create this number of circuits and test them (default = 100).

-s {positive_integer}
    use this value as rng seed value (default = unix time).
```

10.244 test_autotb - generate simple test benches

yosys> help test_autotb

```
test_autotb [options] [filename]
```

Automatically create primitive Verilog test benches for all modules in the design. The generated testbenches toggle the input pins of the module in

(continues on next page)

(continued from previous page)

a semi-random manner and dumps the resulting output signals.

This can be used to check the synthesis results for simple circuits by comparing the testbench output for the input files and the synthesis results.

The backend automatically detects clock signals. Additionally a signal can be forced to be interpreted as clock signal by setting the attribute 'gentb_clock' on the signal.

The attribute 'gentb_constant' can be used to force a signal to a constant value after initialization. This can e.g. be used to force a reset signal low in order to explore more inner states in a state machine.

The attribute 'gentb_skip' can be attached to modules to suppress testbench generation.

```
-n <int>
    number of iterations the test bench should run (default = 1000)

-seed <int>
    seed used for pseudo-random number generation (default = 0).
    a value of 0 will cause an arbitrary seed to be chosen, based on
    the current system time.
```

10.245 test_cell - automatically test the implementation of a cell type

```
yosys> help test_cell
```

```
test_cell [options] {cell-types}
```

Tests the internal implementation of the given cell type (for example '\$add') by comparing SAT solver, EVAL and TECHMAP implementations of the cell types..

Run with 'all' instead of a cell type to run the test on all supported cell types. Use for example 'all /\$add' for all cell types except \$add.

```
-n {integer}
    create this number of cell instances and test them (default = 100).

-s {positive_integer}
    use this value as rng seed value (default = unix time).

-f {rtlil_file}
    don't generate circuits. instead load the specified RTLIL file.

-w {filename_prefix}
    don't test anything. just generate the circuits and write them
    to RTLIL files with the specified prefix

-map {filename}
```

(continues on next page)

(continued from previous page)

```

    pass this option to techmap.

-simlib
    use "techmap -D SIMLIB_NOCHECKS -map +/simlib.v -max_iter 2 -autoproc"

-aigmap
    instead of calling "techmap", call "aigmap"

-muxdiv
    when creating test benches with dividers, create an additional mux
    to mask out the division-by-zero case

-script {script_file}
    instead of calling "techmap", call "script {script_file}".

-const
    set some input bits to random constant values

-nosat
    do not check SAT model or run SAT equivalence checking

-noeval
    do not check const-eval models

-noopt
    do not opt techmapped design

-edges
    test cell edges db creator against sat-based implementation

-v
    print additional debug information to the console

-vlog {filename}
    create a Verilog test bench to test simlib and write_verilog

-bloat {factor}
    increase cell size limits b{factor} times where possible

-check_cost
    check if the estimated cell cost is a valid upper bound for
    the techmapped cell count

```

10.246 test_generic - test the generic compute graph

```
yosys> help test_generic
```

TODO: add help message

10.247 test_pmgen - test pass for pmgen

yosys> help test_pmgen

```
test_pmgen -reduce_chain [options] [selection]
```

Demo for recursive pmgen patterns. Map chains of AND/OR/XOR to \$reduce_*.

```
test_pmgen -reduce_tree [options] [selection]
```

Demo for recursive pmgen patterns. Map trees of AND/OR/XOR to \$reduce_*.

```
test_pmgen -eqpmux [options] [selection]
```

Demo for recursive pmgen patterns. Optimize EQ/NE/PMUX circuits.

```
test_pmgen -generate [options] <pattern_name>
```

Create modules that match the specified pattern.

10.248 torder - print cells in topological order

yosys> help torder

```
torder [options] [selection]
```

This command prints the selected cells in topological order.

```
-stop <cell_type> <cell_port>
```

do not use the specified cell port in topological sorting

```
-noautostop
```

by default Q outputs of internal FF cells and memory read port outputs are not used in topological sorting. this option deactivates that.

10.249 trace - redirect command output to file

yosys> help trace

```
trace cmd
```

Execute the specified command, logging all changes the command performs on the design in real time.

10.250 tribuf - infer tri-state buffers

yosys> help tribuf

```
tribuf [options] [selection]
```

This pass transforms \$mux cells with 'z' inputs to tristate buffers.

```
-merge
```

merge multiple tri-state buffers driving the same net into a single buffer.

```
-logic
```

convert tri-state buffers that do not drive output ports to non-tristate logic. this option implies -merge.

```
-formal
```

convert all tri-state buffers to non-tristate logic and add a formal assertion that no two buffers are driving the same net simultaneously. this option implies -merge.

10.251 uniquify - create unique copies of modules

yosys> help uniquify

```
uniquify [selection]
```

By default, a module that is instantiated by several other modules is only kept once in the design. This preserves the original modularity of the design and reduces the overall size of the design in memory. But it prevents certain optimizations and other operations on the design. This pass creates unique modules for all selected cells. The created modules are marked with the 'unique' attribute.

This commands only operates on modules that by themselves have the 'unique' attribute set (the 'top' module is unique implicitly).

10.252 verifc - load Verilog and VHDL designs using Verific

yosys> help verifc

```
verifc [-work <libname>] {-sv|-vhdl|...} <hdl-file>
```

Load the specified Verilog/SystemVerilog/VHDL file into the specified library.
(default library when -work is not present: "work")

```
verifc [-L <libname>] {-sv|-vhdl|...} <hdl-file>
```

Look up external definitions in the specified library.
(-L may be used more than once)

```
verifc -set-error <msg_id>..  
verifc -set-warning <msg_id>..  
verifc -set-info <msg_id>..  
verifc -set-ignore <msg_id>..
```

Set message severity. <msg_id> is the string in square brackets when a message is printed, such as VERI-1209.
Also errors, warnings, infos and comments could be used to set new severity for all messages of certain type.

```
verifc -import [options] <top>..
```

Elaborate the design for the specified top modules or configurations, import to Yosys and reset the internal state of Verific.

Import options:

```
-all  
  Elaborate all modules, not just the hierarchy below the given top  
  modules. With this option the list of modules to import is optional.  
  
-gates  
  Create a gate-level netlist.  
  
-flatten  
  Flatten the design in Verific before importing.  
  
-extnets  
  Resolve references to external nets by adding module ports as needed.  
  
-no-split-complex-ports  
  Complex ports (structs or arrays) are not split and remain packed as a single port.  
  
-fullinit  
  Keep all register initializations, even those for non-FF registers.
```

(continues on next page)

(continued from previous page)

```
-cells
  Import all cell definitions from Verific loaded libraries even if they are
  unused in design. Useful with "-edif" and "-liberty" option.

-chparam name value
  Elaborate the specified top modules (all modules when -all given) using
  this parameter value. Modules on which this parameter does not exist will
  cause Verific to produce a VERI-1928 or VHDL-1676 message. This option
  can be specified multiple times to override multiple parameters.
  String values must be passed in double quotes ("").

-v, -vv
  Verbose log messages. (-vv is even more verbose than -v.)

-pp <filename>
  Pretty print design after elaboration to specified file.
```

The following additional import options are useful for debugging the Verific bindings (for Yosys and/or Verific developers):

```
-k
  Keep going after an unsupported verific primitive is found. The
  unsupported primitive is added as blockbox module to the design.
  This will also add all SVA related cells to the design parallel to
  the checker logic inferred by it.

-V
  Import Verific netlist as-is without translating to Yosys cell types.

-n
  Keep all Verific names on instances and nets. By default only
  user-declared names are preserved.

-d <dump_file>
  Dump the Verific netlist as a verilog file.

  verific [-work <libname>] -pp [options] <filename> [<module>]..
```

Pretty print design (or just module) to the specified file from the specified library. (default library when -work is not present: "work")

Pretty print options:

```
-verilog
  Save output for Verilog/SystemVerilog design modules (default).

-vhdl
  Save output for VHDL design units.
```

(continues on next page)

(continued from previous page)

```
verific -cfg [<name> [<value>]]
```

Get/set Verific runtime flags.

Use YosysHQ Tabby CAD Suite if you need Yosys+Verific.
<https://www.yosyshq.com/>

Contact office@yosyshq.com for free evaluation
binaries of YosysHQ Tabby CAD Suite.

10.253 verilog_defaults - set default options for read_verilog

yosys> help verilog_defaults

```
verilog_defaults -add [options]
```

Add the specified options to the list of default options to read_verilog.

```
verilog_defaults -clear
```

Clear the list of Verilog default options.

```
verilog_defaults -push  
verilog_defaults -pop
```

Push or pop the list of default options to a stack. Note that -push does
not imply -clear.

10.254 verilog_defines - define and undefine verilog defines

yosys> help verilog_defines

```
verilog_defines [options]
```

Define and undefine verilog preprocessor macros.

```
-Dname[=definition]  
    define the preprocessor symbol 'name' and set its optional value  
    'definition'
```

```
-Uname[=definition]  
    undefine the preprocessor symbol 'name'
```

```
-reset
```

(continues on next page)

(continued from previous page)

```

clear list of defined preprocessor symbols

-list
    list currently defined preprocessor symbols

```

10.255 viz - visualize data flow graph

yosys> help viz

```
viz [options] [selection]
```

Create a graphviz DOT file for the selected part of the design, showing the relationships between the selected wires, and compile it to a graphics file (usually SVG or PostScript).

```

-viewer <viewer>
    Run the specified command with the graphics file as parameter.
    On Windows, this pauses yosys until the viewer exits.

-format <format>
    Generate a graphics file in the specified format. Use 'dot' to just
    generate a .dot file, or other <format> strings such as 'svg' or 'ps'
    to generate files in other formats (this calls the 'dot' command).

-prefix <prefix>
    generate <prefix>.* instead of ~/.yosys_viz.*

-pause
    wait for the user to press enter to before returning

-nobg
    don't run viewer in the background, IE wait for the viewer tool to
    exit before returning

-set-vg-attr
    set their group index as 'vg' attribute on cells and wires

-g <selection>
    manually define a group of terminal signals. this group is not being
    merged with other terminal groups.

-u <selection>
    manually define a unique group for each wire in the selection.

-x <selection>
    manually exclude wires from being considered. (usually this is
    used for global signals, such as reset.)

-s <selection>
    like -g, but mark group as 'special', changing the algorithm to

```

(continues on next page)

(continued from previous page)

```

    preserve as much info about this groups connectivity as possible.

-G <selection_expr> .
-U <selection_expr> .
-X <selection_expr> .
-S <selection_expr> .
    like -u, -g, -x, and -s, but parse all arguments up to a terminating .
    as a single select expression. (see 'help select' for details)

-0, -1, -2, -3, -4, -5, -6, -7, -8, -9
    select effort level. each level corresponds to an incresingly more
    aggressive sequence of strategies for merging nodes of the data flow
    graph. (default: 9)

```

When no <format> is specified, 'dot' is used. When no <format> and <viewer> is specified, 'xdot' is used to display the schematic (POSIX systems only).

The generated output files are '~/yosys_viz.dot' and '~/yosys_viz.<format>', unless another prefix is specified using -prefix <prefix>.

Yosys on Windows and YosysJS use different defaults: The output is written to 'show.dot' in the current directory and new viewer is launched each time the 'show' command is executed.

10.256 wbflip - flip the whitebox attribute

```
yosys> help wbflip
```

```
wbflip [selection]
```

Flip the whitebox attribute on selected cells. I.e. if it's set, unset it, and vice-versa. Blackbox cells are not effected by this command.

10.257 wrapcell - wrap individual cells into new modules

```
yosys> help wrapcell
```

```
wrapcell -name <format> [selection]
```

This command wraps the selected cells individually into modules. The name for each wrapper module is derived from the template <format> by substituting parameter values as specified in curly brackets. If the named module already exists, it is reused.

If the template contains the special string '{%unused}', the command tracks unused output ports -- specialized wrapper modules will be generated per every distinct set of unused port bits as appearing on any selected cell.

(continues on next page)

(continued from previous page)

```
-setattr <attribute-name>
    set the given boolean attribute on each created wrapper module

-formatattr <attribute-name> <format>
    set a string attribute on the created wrapper module by substituting
    parameter values into <format>
```

Currently this command only supports wrapping internal cell types.

10.258 wreduce - reduce the word size of operations if possible

yosys> help wreduce

```
wreduce [options] [selection]
```

This command reduces the word size of operations. For example it will replace the 32 bit adders in the following code with adders of more appropriate widths:

```
module test(input [3:0] a, b, c, output [7:0] y);
    assign y = a + b + c + 1;
endmodule
```

Options:

```
-memx
    Do not change the width of memory address ports. Use this options in
    flows that use the 'memory_memx' pass.

-mux_undef
    remove 'undef' inputs from $mux, $pmux and $_MUX_ cells

-keepdc
    Do not optimize explicit don't-care values.
```

10.259 write_aiger - write design to AIGER file

yosys> help write_aiger

```
write_aiger [options] [filename]
```

Write the current design to an AIGER file. The design must be flattened and must not contain any cell types except \$_AND_, \$_NOT_, simple FF types, \$assert and \$assume cells, and \$initstate cells.

\$assert and \$assume cells are converted to AIGER bad state properties and invariant constraints.

(continues on next page)

(continued from previous page)

```

-ascii
    write ASCII version of AIGER format

-zinit
    convert FFs to zero-initialized FFs, adding additional inputs for
    uninitialized FFs.

-miter
    design outputs are AIGER bad state properties

-symbols
    include a symbol table in the generated AIGER file

-map <filename>
    write an extra file with port and latch symbols

-vmap <filename>
    like -map, but more verbose

-no-startoffset
    make indexes zero based, enable using map files with smt solvers.

-ywmap <filename>
    write a map file for conversion to and from yosys witness traces.

-I, -O, -B, -L
    If the design contains no input/output/assert/flip-flop then create one
    dummy input/output/bad_state-pin or latch to make the tools reading the
    AIGER file happy.

```

10.260 write_aiger2 - (experimental) write design to AIGER file

```
yosys> help write_aiger2
```

```
write_aiger2 [options] [filename]
```

Write the selected module to an AIGER file.

```

-strash
    perform structural hashing while writing

-flatten
    allow descending into submodules and write a flattened view of the design
    hierarchy starting at the selected top

```

This command is able to ingest all combinational cells except for:

```

$alu, $lcu, $demux, $bweqx, $macc, $concat, $pow, $modfloor, $divfloor,
$mod, $div, $mul, $sub, $add, $nex, $eqx, $shiftx, $shift, $sshr,

```

(continues on next page)

(continued from previous page)

```
$sshl, $shr, $shl, $sop, $lut, $slice, $neg,
```

And all combinational gates except for:

```
$_TBUF_, $_MUX16_, $_MUX8_, $_MUX4_,
```

WARNING: THE 'write_aiger2' COMMAND IS EXPERIMENTAL.

10.261 write_blif - write design to BLIF file

```
yosys> help write_blif
```

```
write_blif [options] [filename]
```

Write the current design to an BLIF file.

```
-top top_module
```

set the specified module as design top module

```
-buf <cell-type> <in-port> <out-port>
```

use cells of type <cell-type> with the specified port names for buffers

```
-unbuf <cell-type> <in-port> <out-port>
```

replace buffer cells with the specified name and port names with
a .names statement that models a buffer

```
-true <cell-type> <out-port>
```

```
-false <cell-type> <out-port>
```

```
-undef <cell-type> <out-port>
```

use the specified cell types to drive nets that are constant 1, 0, or undefined. when '-' is used as <cell-type>, then <out-port> specifies the wire name to be used for the constant signal and no cell driving that wire is generated. when '+' is used as <cell-type>, then <out-port> specifies the wire name to be used for the constant signal and a .names statement is generated to drive the wire.

```
-noalias
```

if a net name is aliasing another net name, then by default a net without fanout is created that is driven by the other net. This option suppresses the generation of this nets without fanout.

The following options can be useful when the generated file is not going to be read by a BLIF parser but a custom tool. It is recommended not to name the output file *.blif when any of these options are used.

```
-icells
```

do not translate Yosys's internal gates to generic BLIF logic functions. Instead create .subckt or .gate lines for all cells.

```
-gates
```

(continues on next page)

(continued from previous page)

```
print .gate instead of .subckt lines for all cells that are not
instantiations of other modules from this design.

-conn
    do not generate buffers for connected wires. instead use the
    non-standard .conn statement.

-attr
    use the non-standard .attr statement to write cell attributes

-param
    use the non-standard .param statement to write cell parameters

-cname
    use the non-standard .cname statement to write cell names

-iname, -iattr
    enable -cname and -attr functionality for .names statements
    (the .cname and .attr statements will be included in the BLIF
    output after the truth table for the .names statement)

-blackbox
    write blackbox cells with .blackbox statement.

-impltf
    do not write definitions for the $true, $false and $undef wires.
```

10.262 write_btor - write design to BTOR file

```
yosys> help write_btor
```

```
write_btor [options] [filename]

Write a BTOR description of the current design.

-v
    Add comments and indentation to BTOR output file

-s
    Output only a single bad property for all asserts

-c
    Output cover properties using 'bad' statements instead of asserts

-i <filename>
    Create additional info file with auxiliary information

-x
    Output symbols for internal netnames (starting with '$')
```

(continues on next page)

(continued from previous page)

```
-ywmap <filename>
  Create a map file for conversion to and from Yosys witness traces
```

10.263 write_cxxrtl - convert design to C++ RTL simulation

```
yosys> help write_cxxrtl
```

```
write_cxxrtl [options] [filename]
```

Write C++ code that simulates the design. The generated code requires a driver that instantiates the design, toggles its clock, and interacts with its ports.

The following driver may be used as an example for a design with a single clock driving rising edge triggered flip-flops:

```
#include "top.cc"

int main() {
    cxxrtl_design::p_top top;
    top.step();
    while (1) {
        /* user logic */
        top.p_clk.set(false);
        top.step();
        top.p_clk.set(true);
        top.step();
    }
}
```

Note that CXXRTL simulations, just like the hardware they are simulating, are subject to race conditions. If, in the example above, the user logic would run simultaneously with the rising edge of the clock, the design would malfunction.

This backend supports replacing parts of the design with black boxes implemented in C++. If a module marked as a CXXRTL black box, its implementation is ignored, and the generated code consists only of an interface and a factory function. The driver must implement the factory function that creates an implementation of the black box, taking into account the parameters it is instantiated with.

For example, the following Verilog code defines a CXXRTL black box interface for a synchronous debug sink:

```
(* cxxrtl_blackbox *)
module debug(...);
    (* cxxrtl_edge = "p" *) input clk;
    input en;
    input [7:0] i_data;
    (* cxxrtl_sync *) output [7:0] o_data;
endmodule
```

(continues on next page)

(continued from previous page)

For this HDL interface, this backend will generate the following C++ interface:

```
struct bb_p_debug : public module {
    value<1> p_clk;
    bool posedge_p_clk() const { /* ... */ }
    value<1> p_en;
    value<8> p_i_data;
    wire<8> p_o_data;

    bool eval(performer *performer) override;
    virtual bool commit(observer &observer);
    bool commit() override;

    static std::unique_ptr<bb_p_debug>
        create(std::string name, metadata_map parameters, metadata_map attributes);
};
```

The `'create'` function must be implemented by the driver. For example, it could always provide an implementation logging the values to standard error stream:

```
namespace cxxrtl_design {

struct stderr_debug : public bb_p_debug {
    bool eval(performer *performer) override {
        if (posedge_p_clk() && p_en)
            fprintf(stderr, "debug: %02x\n", p_i_data.data[0]);
        p_o_data.next = p_i_data;
        return bb_p_debug::eval(performer);
    }
};

std::unique_ptr<bb_p_debug>
bb_p_debug::create(std::string name, cxxrtl::metadata_map parameters,
                  cxxrtl::metadata_map attributes) {
    return std::make_unique<stderr_debug>();
}

}
```

For complex applications of black boxes, it is possible to parameterize their port widths. For example, the following Verilog code defines a CXXRTL black box interface for a configurable width debug sink:

```
(* cxxrtl_blackbox, cxxrtl_template = "WIDTH" *)
module debug(...);
    parameter WIDTH = 8;
    (* cxxrtl_edge = "p" *) input clk;
    input en;
    (* cxxrtl_width = "WIDTH" *) input [WIDTH - 1:0] i_data;
    (* cxxrtl_width = "WIDTH" *) output [WIDTH - 1:0] o_data;
endmodule
```

(continues on next page)

(continued from previous page)

For this parametric HDL interface, this backend will generate the following C++ interface (only the differences are shown):

```
template<size_t WIDTH>
struct bb_p_debug : public module {
    // ...
    value<WIDTH> p_i_data;
    wire<WIDTH> p_o_data;
    // ...
    static std::unique_ptr<bb_p_debug<WIDTH>>
    create(std::string name, metadata_map parameters, metadata_map attributes);
};
```

The ``create'` function must be implemented by the driver, specialized for every possible combination of template parameters. (Specialization is necessary to enable separate compilation of generated code and black box implementations.)

```
template<size_t SIZE>
struct stderr_debug : public bb_p_debug<SIZE> {
    // ...
};

template<>
std::unique_ptr<bb_p_debug<8>>
bb_p_debug<8>::create(std::string name, cxxrtl::metadata_map parameters,
                     cxxrtl::metadata_map attributes) {
    return std::make_unique<stderr_debug<8>>();
}
```

The following attributes are recognized by this backend:

`cxxrtl_blackbox`

only valid on modules. if specified, the module contents are ignored, and the generated code includes only the module interface and a factory function, which will be called to instantiate the module.

`cxxrtl_edge`

only valid on inputs of black boxes. must be one of "p", "n", "a". if specified on signal ``clk'`, the generated code includes edge detectors ``posedge_p_clk()'` (if "p"), `negedge_p_clk()'` (if "n"), or both (if "a"), simplifying implementation of clocked black boxes.`

`cxxrtl_template`

only valid on black boxes. must contain a space separated sequence of identifiers that have a corresponding black box parameters. for each of them, the generated code includes a ``size_t'` template parameter.

`cxxrtl_width`

only valid on ports of black boxes. must be a constant expression, which is directly inserted into generated code.

`cxxrtl_comb, cxxrtl_sync`

(continues on next page)

(continued from previous page)

only valid on outputs of black boxes. if specified, indicates that every bit of the output port is driven, correspondingly, by combinatorial or synchronous logic. this knowledge is used for scheduling optimizations. if neither is specified, the output will be pessimistically treated as driven by both combinatorial and synchronous logic.

The following options are supported by this backend:

- print-wire-types, -print-debug-wire-types
enable additional debug logging, for pass developers.
- header
generate separate interface (.h) and implementation (.cc) files.
if specified, the backend must be called with a filename, and filename of the interface is derived from filename of the implementation.
otherwise, interface and implementation are generated together.
- namespace <ns-name>
place the generated code into namespace <ns-name>. if not specified, "cxxrtl_design" is used.
- print-output <stream>
\$print cells in the generated code direct their output to <stream>.
must be one of "std::cout", "std::cerr". if not specified, "std::cout" is used. explicitly provided performer overrides this.
- nohierarchy
use design hierarchy as-is. in most designs, a top module should be present as it is exposed through the C API and has unbuffered outputs for improved performance; it will be determined automatically if absent.
- noflatten
don't flatten the design. fully flattened designs can evaluate within one delta cycle if they have no combinatorial feedback.
note that the debug interface and waveform dumps use full hierarchical names for all wires even in flattened designs.
- noprocs
don't convert processes to netlists. in most designs, converting processes significantly improves evaluation performance at the cost of slight increase in compilation time.
- O <level>
set the optimization level. the default is -O6. higher optimization levels dramatically decrease compile and run time, and highest level possible for a design should be used.
- O0
no optimization.
- O1
unbuffer internal wires if possible.

(continues on next page)

(continued from previous page)

```

-02
    like -01, and localize internal wires if possible.

-03
    like -02, and inline internal wires if possible.

-04
    like -03, and unbuffer public wires not marked (*keep*) if possible.

-05
    like -04, and localize public wires not marked (*keep*) if possible.

-06
    like -05, and inline public wires not marked (*keep*) if possible.

-g <level>
    set the debug level. the default is -g4. higher debug levels provide
    more visibility and generate more code, but do not pessimize evaluation.

-g0
    no debug information. the C API is disabled.

-g1
    include bare minimum of debug information necessary to access all design
    state. the C API is enabled.

-g2
    like -g1, but include debug information for all public wires that are
    directly accessible through the C++ interface.

-g3
    like -g2, and include debug information for public wires that are tied
    to a constant or another public wire.

-g4
    like -g3, and compute debug information on demand for all public wires
    that were optimized out.

```

10.264 write_edif - write design to EDIF netlist file

```
yosys> help write_edif
```

```
write_edif [options] [filename]
```

Write the current design to an EDIF netlist file.

```

-top top_module
    set the specified module as design top module

```

(continues on next page)

(continued from previous page)

```
-nogndvcc
    do not create "GND" and "VCC" cells. (this will produce an error
    if the design contains constant nets. use "hilomap" to map to custom
    constant drivers first)

-gndvccy
    create "GND" and "VCC" cells with "Y" outputs. (the default is
    "G" for "GND" and "P" for "VCC".)

-attrprop
    create EDIF properties for cell attributes

-keep
    create extra KEEP nets by allowing a cell to drive multiple nets.

-pvector {par|bra|ang}
    sets the delimiting character for module port rename clauses to
    parentheses, square brackets, or angle brackets.

-lsbidx
    use index 0 for the LSB bit of a net or port instead of MSB.
```

Unfortunately there are different "flavors" of the EDIF file format. This command generates EDIF files for the Xilinx place&route tools. It might be necessary to make small modifications to this command when a different tool is targeted.

10.265 write_file - write a text to a file

```
yosys> help write_file
```

```
write_file [options] output_file [input_file]
```

Write the text from the input file to the output file.

```
-a
    Append to output file (instead of overwriting)
```

Inside a script the input file can also can a here-document:

```
write_file hello.txt <<EOT
Hello World!
EOT
```

10.266 write_firrtl - write design to a FIRRTL file

```
yosys> help write_firrtl
```

```
write_firrtl [options] [filename]
```

Write a FIRRTL netlist of the current design.

The following commands are executed by this command:

```
pmuxtree
bmuxmap
demuxmap
bwmuxmap
```

10.267 write_functional_cxx - convert design to C++ using the functional backend

```
yosys> help write_functional_cxx
```

TODO: add help message

10.268 write_functional_rosette - Generate Rosette compatible Racket from Functional IR

```
yosys> help write_functional_rosette
```

```
write_functional_rosette [options] [filename]
```

Functional Rosette Backend.

```
-provides
  include 'provide' statement(s) for loading output as a module
```

10.269 write_functional_smt2 - Generate SMT-LIB from Functional IR

```
yosys> help write_functional_smt2
```

Functional SMT Backend.

10.270 write_intersynth - write design to InterSynth netlist file

```
yosys> help write_intersynth
```

```
write_intersynth [options] [filename]
```

Write the current design to an 'intersynth' netlist file. InterSynth is a tool for Coarse-Grain Example-Driven Interconnect Synthesis.

-notypes

do not generate celltypes and conntypes commands. i.e. just output the netlists. this is used for postsilicon synthesis.

-lib <verilog_or_rtlil_file>

Use the specified library file for determining whether cell ports are inputs or outputs. This option can be used multiple times to specify more than one library.

-selected

only write selected modules. modules must be selected entirely or not at all.

<http://bygone.clairexen.net/intersynth/>

10.271 write_jny - generate design metadata

```
yosys> help write_jny
```

```
jny [options] [selection]
```

Write JSON netlist metadata for the current design

-no-connections

Don't include connection information in the netlist output.

-no-attributes

Don't include attributed information in the netlist output.

-no-properties

Don't include property information in the netlist output.

The JSON schema for JNY output files is located in the "jny.schema.json" file which is located at "<https://raw.githubusercontent.com/YosysHQ/yosys/main/misc/jny.schema.json>"

10.272 write_json - write design to a JSON file

yosys> help write_json

```
write_json [options] [filename]
```

Write a JSON netlist of the current design.

-aig

include AIG models for the different gate types

-compat-int

emit 32-bit or smaller fully-defined parameter values directly
as JSON numbers (for compatibility with old parsers)

-selected

output only select module

-noscopeinfo

don't include \$scopeinfo cells in the output

The general syntax of the JSON output created by this command is as follows:

```
{
  "creator": "Yosys <version info>",
  "modules": {
    <module_name>: {
      "attributes": {
        <attribute_name>: <attribute_value>,
        ...
      },
      "parameter_default_values": {
        <parameter_name>: <parameter_value>,
        ...
      },
      "ports": {
        <port_name>: <port_details>,
        ...
      },
      "cells": {
        <cell_name>: <cell_details>,
        ...
      },
      "memories": {
        <memory_name>: <memory_details>,
        ...
      },
      "netnames": {
        <net_name>: <net_details>,
        ...
      }
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    }
  },
  "models": {
    ...
  },
}

```

Where <port_details> is:

```

{
  "direction": <"input" | "output" | "inout">,
  "bits": <bit_vector>
  "offset": <the lowest bit index in use, if non-0>
  "upto": <1 if the port bit indexing is MSB-first>
  "signed": <1 if the port is signed>
}

```

The "offset" and "upto" fields are skipped if their value would be 0. They don't affect connection semantics, and are only used to preserve original HDL bit indexing.

And <cell_details> is:

```

{
  "hide_name": <1 | 0>,
  "type": <cell_type>,
  "model": <AIG model name, if -aig option used>,
  "parameters": {
    <parameter_name>: <parameter_value>,
    ...
  },
  "attributes": {
    <attribute_name>: <attribute_value>,
    ...
  },
  "port_directions": {
    <port_name>: <"input" | "output" | "inout">,
    ...
  },
  "connections": {
    <port_name>: <bit_vector>,
    ...
  },
}

```

And <memory_details> is:

```

{
  "hide_name": <1 | 0>,
  "attributes": {
    <attribute_name>: <attribute_value>,
    ...
  },
}

```

(continues on next page)

(continued from previous page)

```

    "width": <memory width>
    "start_offset": <the lowest valid memory address>
    "size": <memory size>
}

```

And <net_details> is:

```

{
  "hide_name": <1 | 0>,
  "bits": <bit_vector>
  "offset": <the lowest bit index in use, if non-0>
  "upto": <1 if the port bit indexing is MSB-first>
  "signed": <1 if the port is signed>
}

```

The "hide_name" fields are set to 1 when the name of this cell or net is automatically created and is likely not of interest for a regular user.

The "port_directions" section is only included for cells for which the interface is known.

Module and cell ports and nets can be single bit wide or vectors of multiple bits. Each individual signal bit is assigned a unique integer. The <bit_vector> values referenced above are vectors of this integers. Signal bits that are connected to a constant driver are denoted as string "0", "1", "x", or "z" instead of a number.

Bit vectors (including integers) are written as string holding the binary representation of the value. Strings are written as strings, with an appended blank in cases of strings of the form /[01xz]* */.

For example the following Verilog code:

```

module test(input x, y);
  (* keep *) foo #(.P(42), .Q(1337))
    foo_inst (.A({x, y}), .B({y, x}), .C({4'd10, {4{x}}}));
endmodule

```

Translates to the following JSON output:

```

{
  "creator": "Yosys 0.9+2406 (git sha1 fb1168d8, clang 9.0.1 -fPIC -Os)",
  "modules": {
    "test": {
      "attributes": {
        "cells_not_processed": "00000000000000000000000000000001",
        "src": "test.v:1.1-4.10"
      },
      "ports": {
        "x": {
          "direction": "input",
          "bits": [ 2 ]
        }
      }
    }
  }
}

```

(continues on next page)

(continued from previous page)

```
,  
    "y": {  
      "direction": "input",  
      "bits": [ 3 ]  
    }  
  },  
  "cells": {  
    "foo_inst": {  
      "hide_name": 0,  
      "type": "foo",  
      "parameters": {  
        "P": "00000000000000000000000000000101010",  
        "Q": "000000000000000000000000010100111001"  
      },  
      "attributes": {  
        "keep": "0000000000000000000000000000000001",  
        "module_not_derived": "0000000000000000000000000000000001",  
        "src": "test.v:3.1-3.55"  
      },  
      "connections": {  
        "A": [ 3, 2 ],  
        "B": [ 2, 3 ],  
        "C": [ 2, 2, 2, 2, "0", "1", "0", "1" ]  
      }  
    }  
  },  
  "netnames": {  
    "x": {  
      "hide_name": 0,  
      "bits": [ 2 ],  
      "attributes": {  
        "src": "test.v:1.19-1.20"  
      }  
    },  
    "y": {  
      "hide_name": 0,  
      "bits": [ 3 ],  
      "attributes": {  
        "src": "test.v:1.22-1.23"  
      }  
    }  
  }  
}  
  
}
```

The models are given as And-Inverter-Graphs (AIGs) in the following form:

```
"models": {
  <model_name>: [
    /* 0 */ [ <node-spec> ],
    /* 1 */ [ <node-spec> ],
```

(continues on next page)

(continued from previous page)

```

    /* 2 */ [ <node-spec> ],
    ...
  ],
  ...
},

```

The following node-types may be used:

```

[ "port", <portname>, <bitindex>, <out-list> ]
- the value of the specified input port bit

[ "nport", <portname>, <bitindex>, <out-list> ]
- the inverted value of the specified input port bit

[ "and", <node-index>, <node-index>, <out-list> ]
- the ANDed value of the specified nodes

[ "nand", <node-index>, <node-index>, <out-list> ]
- the inverted ANDed value of the specified nodes

[ "true", <out-list> ]
- the constant value 1

[ "false", <out-list> ]
- the constant value 0

```

All nodes appear in topological order. I.e. only nodes with smaller indices are referenced by "and" and "nand" nodes.

The optional <out-list> at the end of a node specification is a list of output portname and bitindex pairs, specifying the outputs driven by this node.

For example, the following is the model for a 3-input 3-output \$reduce_and cell inferred by the following code:

```

module test(input [2:0] in, output [2:0] out);
  assign in = &out;
endmodule

"$reduce_and:3U:3": [
  /* 0 */ [ "port", "A", 0 ],
  /* 1 */ [ "port", "A", 1 ],
  /* 2 */ [ "and", 0, 1 ],
  /* 3 */ [ "port", "A", 2 ],
  /* 4 */ [ "and", 2, 3, "Y", 0 ],
  /* 5 */ [ "false", "Y", 1, "Y", 2 ]
]

```

Future version of Yosys might add support for additional fields in the JSON format. A program processing this format must ignore all unknown fields.

10.273 write_rtlil - write design to RTLIL file

yosys> help write_rtlil

```
write_rtlil [filename]
```

Write the current design to an RTLIL file. (RTLIL is a text representation of a design in yosys's internal format.)

```
-selected  
    only write selected parts of the design.
```

10.274 write_simplec - convert design to simple C code

yosys> help write_simplec

```
write_simplec [options] [filename]
```

Write simple C code for simulating the design. The C code written can be used to simulate the design in a C environment, but the purpose of this command is to generate code that works well with C-based formal verification.

```
-verbose  
    this will print the recursive walk used to export the modules.  
  
-i8, -i16, -i32, -i64  
    set the maximum integer bit width to use in the generated code.
```

THIS COMMAND IS UNDER CONSTRUCTION

10.275 write_smt2 - write design to SMT-LIBv2 file

yosys> help write_smt2

```
write_smt2 [options] [filename]
```

Write a SMT-LIBv2 [1] description of the current design. For a module with name '<mod>' this will declare the sort '<mod>_s' (state of the module) and will define and declare functions operating on that state.

The following SMT2 functions are generated for a module with name '<mod>'. Some declarations/definitions are printed with a special comment. A prover using the SMT2 files can use those comments to collect all relevant metadata about the design.

```
; yosys-smt2-module <mod>  
(declare-sort |<mod>_s| 0)  
    The sort representing a state of module <mod>.
```

(continues on next page)

(continued from previous page)

```

(define-fun |<mod>_h| ((state |<mod>_s|)) Bool (...))
  This function must be asserted for each state to establish the
  design hierarchy.

; yosys-smt2-input <wirename> <width>
; yosys-smt2-output <wirename> <width>
; yosys-smt2-register <wirename> <width>
; yosys-smt2-wire <wirename> <width>
(define-fun |<mod>_n <wirename>| (|<mod>_s|) (_ BitVec <width>))
(define-fun |<mod>_n <wirename>| (|<mod>_s|) Bool)
  For each port, register, and wire with the 'keep' attribute set an
  accessor function is generated. Single-bit wires are returned as Bool,
  multi-bit wires as BitVec.

; yosys-smt2-cell <submod> <instancename>
(declare-fun |<mod>_h <instancename>| (|<mod>_s|) |<submod>_s|)
  There is a function like that for each hierarchical instance. It
  returns the sort that represents the state of the sub-module that
  implements the instance.

(declare-fun |<mod>_is| (|<mod>_s|) Bool)
  This function must be asserted 'true' for initial states, and 'false'
  otherwise.

(define-fun |<mod>_i| ((state |<mod>_s|)) Bool (...))
  This function must be asserted 'true' for initial states. For
  non-initial states it must be left unconstrained.

(define-fun |<mod>_t| ((state |<mod>_s|) (next_state |<mod>_s|)) Bool (...))
  This function evaluates to 'true' if the states 'state' and
  'next_state' form a valid state transition.

(define-fun |<mod>_a| ((state |<mod>_s|)) Bool (...))
  This function evaluates to 'true' if all assertions hold in the state.

(define-fun |<mod>_u| ((state |<mod>_s|)) Bool (...))
  This function evaluates to 'true' if all assumptions hold in the state.

; yosys-smt2-assert <id> <filename:linenum>
(define-fun |<mod>_a <id>| ((state |<mod>_s|)) Bool (...))
  Each $assert cell is converted into one of this functions. The function
  evaluates to 'true' if the assert statement holds in the state.

; yosys-smt2-assume <id> <filename:linenum>
(define-fun |<mod>_u <id>| ((state |<mod>_s|)) Bool (...))
  Each $assume cell is converted into one of this functions. The function
  evaluates to 'true' if the assume statement holds in the state.

; yosys-smt2-cover <id> <filename:linenum>
(define-fun |<mod>_c <id>| ((state |<mod>_s|)) Bool (...))
  Each $cover cell is converted into one of this functions. The function

```

(continues on next page)

(continued from previous page)

evaluates to 'true' if the cover statement is activated in the state.

Options:

-verbose

this will print the recursive walk used to export the modules.

-stbv

Use a BitVec sort to represent a state instead of an uninterpreted sort. As a side-effect this will prevent use of arrays to model memories.

-stdt

Use SMT-LIB 2.6 style datatypes to represent a state instead of an uninterpreted sort.

-nobv

disable support for BitVec (FixedSizeBitVectors theory). without this option multi-bit wires are represented using the BitVec sort and support for coarse grain cells (incl. arithmetic) is enabled.

-nomem

disable support for memories (via ArraysEx theory). this option is implied by -nobv. only \$mem cells without merged registers in read ports are supported. call "memory" with -nordff to make sure that no registers are merged into \$mem read ports. '<mod>_m' functions will be generated for accessing the arrays that are used to represent memories.

-wires

create '<mod>_n' functions for all public wires. by default only ports, registers, and wires with the 'keep' attribute are exported.

-tpl <template_file>

use the given template file. the line containing only the token '%%' is replaced with the regular output of this command.

-solver-option <option> <value>

emit a `; yosys-smt2-solver-option` directive for yosys-smtbmc to write the given option as a `(set-option ...)` command in the SMT-LIBv2.

[1] For more information on SMT-LIBv2 visit <http://smt-lib.org/> or read David R. Cok's tutorial: <https://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf>

Example:

Consider the following module (test.v). We want to prove that the output can never transition from a non-zero value to a zero value.

```
module test(input clk, output reg [3:0] y);
```

(continues on next page)

(continued from previous page)

```

    always @(posedge clk)
        y <= (y << 1) | ^y;
endmodule

```

For this proof we create the following template (test.tpl).

```

; we need QF_UFBV for this proof
(set-logic QF_UFBV)

; insert the auto-generated code here
%%

; declare two state variables s1 and s2
(declare-fun s1 () test_s)
(declare-fun s2 () test_s)

; state s2 is the successor of state s1
(assert (test_t s1 s2))

; we are looking for a model with y non-zero in s1
(assert (distinct (|test_n y| s1) #b0000))

; we are looking for a model with y zero in s2
(assert (= (|test_n y| s2) #b0000))

; is there such a model?
(check-sat)

```

The following yosys script will create a 'test.smt2' file for our proof:

```

read_verilog test.v
hierarchy -check; proc; opt; check -assert
write_smt2 -bv -tpl test.tpl test.smt2

```

Running 'cvc4 test.smt2' will print 'unsat' because y can never transition from non-zero to zero in the test design.

10.276 write_smv - write design to SMV file

yosys> help write_smv

```
write_smv [options] [filename]
```

Write an SMV description of the current design.

```
-verbose
```

this will print the recursive walk used to export the modules.

```
-tpl <template_file>
```

use the given template file. the line containing only the token '%%'

(continues on next page)

(continued from previous page)

is replaced with the regular output of this command.

THIS COMMAND IS UNDER CONSTRUCTION

10.277 write_spice - write design to SPICE netlist file

yosys> help write_spice

```
write_spice [options] [filename]
```

Write the current design to an SPICE netlist file.

```
-big_endian
    generate multi-bit ports in MSB first order
    (default is LSB first)

-neg net_name
    set the net name for constant 0 (default: Vss)

-pos net_name
    set the net name for constant 1 (default: Vdd)

-buf DC|subckt_name
    set the name for jumper element (default: DC)
    (used to connect different nets)

-nc_prefix
    prefix for not-connected nets (default: _NC)

-inames
    include names of internal ($-prefixed) nets in outputs
    (default is to use net numbers instead)

-top top_module
    set the specified module as design top module
```

10.278 write_table - write design as connectivity table

yosys> help write_table

```
write_table [options] [filename]
```

Write the current design as connectivity table. The output is a tab-separated ASCII table with the following columns:

```
module name
cell name
```

(continues on next page)

(continued from previous page)

```

cell type
cell port
direction
signal

```

module inputs and outputs are output using cell type and port '-' and with 'pi' (primary input) or 'po' (primary output) or 'pio' as direction.

10.279 write_verilog - write design to Verilog file

```
yosys> help write_verilog
```

```
write_verilog [options] [filename]
```

Write the current design to a Verilog file.

```
-sv
```

with this option, SystemVerilog constructs like always_comb are used

```
-norename
```

without this option all internal object names (the ones with a dollar instead of a backslash prefix) are changed to short names in the format '_<number>_'.

```
-renameprefix <prefix>
```

insert this prefix in front of auto-generated instance names

```
-noattr
```

with this option no attributes are included in the output

```
-attr2comment
```

with this option attributes are included as comments in the output

```
-noexpr
```

without this option all internal cells are converted to Verilog expressions.

```
-noparallelcase
```

With this option no parallel_case attributes are used. Instead, a case statement that assigns don't-care values for priority dependent inputs is generated.

```
-siminit
```

add initial statements with hierarchical refs to initialize FFs when in -noexpr mode.

```
-nodec
```

32-bit constant values are by default dumped as decimal numbers, not bit pattern. This option deactivates this feature and instead will write out all constants in binary.

(continues on next page)

(continued from previous page)

`-decimal`
dump 32-bit constants in decimal and without size and radix

`-nohex`
constant values that are compatible with hex output are usually dumped as hex values. This option deactivates this feature and instead will write out all constants in binary.

`-nostr`
Parameters and attributes that are specified as strings in the original input will be output as strings by this back-end. This deactivates this feature and instead will write string constants as binary numbers.

`-simple-lhs`
Connection assignments with simple left hand side without concatenations.

`-extmem`
instead of initializing memories using assignments to individual elements, use the '\$readmemh' function to read initialization data from a file. This data is written to a file named by appending a sequential index to the Verilog filename and replacing the extension with '.mem', e.g. 'write_verilog -extmem foo.v' writes 'foo-1.mem', 'foo-2.mem' and so on.

`-defparam`
use 'defparam' statements instead of the Verilog-2001 syntax for cell parameters.

`-blackboxes`
usually modules with the 'blackbox' attribute are ignored. with this option set only the modules with the 'blackbox' attribute are written to the output file.

`-selected`
only write selected modules. modules must be selected entirely or not at all.

`-v`
verbose output (print new names of all renamed wires and cells)

Note that RTLIL processes can't always be mapped directly to Verilog always blocks. This frontend should only be used to export an RTLIL netlist, i.e. after the "proc" pass has been used to convert all processes to logic networks and registers. A warning is generated when this command is called on a design with RTLIL processes.

10.280 write_xaiger - write design to XAIGER file

```
yosys> help write_xaiger
```

```
write_xaiger [options] [filename]
```

Write the top module (according to the (* top *) attribute or if only one module is currently selected) to an XAIGER file. Any non \$_NOT_, \$_AND_, (optionally \$_DFF_N_, \$_DFF_P_), or non (* abc9_box *) cells will be converted into psuedo-inputs and pseudo-outputs. Whitebox contents will be taken from the equivalent module in the '\$abc9_holes' design, if it exists.

```
-ascii
```

```
    write ASCII version of AIGER format
```

```
-map <filename>
```

```
    write an extra file with port and box symbols
```

```
-dff
```

```
    write $_DFF_[NP]_ cells
```

10.281 write_xaiger2 - (experimental) write module to XAIGER file

```
yosys> help write_xaiger2
```

```
write_xaiger2 [options] [filename]
```

Write the selected module to a XAIGER file including the 'h' and 'a' extensions with box information for ABC.

```
-strash
```

```
    perform structural hashing while writing
```

```
-flatten
```

```
    allow descending into submodules and write a flattened view of the design hierarchy starting at the selected top
```

```
-mapping_prep
```

```
    after the file is written, prepare the module for reintegration of a mapping in a subsequent command. all cells which are not blackboxed nor whiteboxed are removed from the design as well as all wires which only connect to removed cells
    (conflicts with -flatten)
```

```
-map2 <file>
```

```
    write a map2 file which 'read_xaiger2 -sc_mapping' can read to reintegrate a mapping
    (conflicts with -flatten)
```

(continues on next page)

(continued from previous page)

```
WARNING: THE 'write_xaiger2' COMMAND IS EXPERIMENTAL.
```

10.282 xilinx_dffopt - Xilinx: optimize FF control signal usage

```
yosys> help xilinx_dffopt
```

```
xilinx_dffopt [options] [selection]
```

Converts hardware clock enable and set/reset signals on FFs to emulation using LUTs, if doing so would improve area. Operates on post-techmap Xilinx cells (LUT*, FD*).

```
-lut4
```

Assume a LUT4-based device (instead of a LUT6-based device).

10.283 xilinx_dsp - Xilinx: pack resources into DSPs

```
yosys> help xilinx_dsp
```

```
xilinx_dsp [options] [selection]
```

Pack input registers (A2, A1, B2, B1, C, D, AD; with optional enable/reset), pipeline registers (M; with optional enable/reset), output registers (P; with optional enable/reset), pre-adder and/or post-adder into Xilinx DSP resources.

Multiply-accumulate operations using the post-adder with feedback on the 'C' input will be folded into the DSP. In this scenario only, the 'C' input can be used to override the current accumulation result with a new value, which will be added to the multiplier result to form the next accumulation result.

Use of the dedicated 'PCOUT' -> 'PCIN' cascade path is detected for 'P' -> 'C' connections (optionally, where 'P' is right-shifted by 17-bits and used as an input to the post-adder -- a pattern common for summing partial products to implement wide multipliers). Limited support also exists for similar cascading for A and B using '[AB]COUT' -> '[AB]CIN'. Currently, cascade chains are limited to a maximum length of 20 cells, corresponding to the smallest Xilinx 7 Series device.

This pass is a no-op if the scratchpad variable 'xilinx_dsp.mulonly' is set to 1.

Experimental feature: addition/subtractions less than 12 or 24 bits with the '(* use_dsp="simd" *)' attribute attached to the output wire or attached to the add/subtract operator will cause those operations to be implemented using the 'SIMD' feature of DSPs.

(continues on next page)

(continued from previous page)

Experimental feature: the presence of a ``$ge'` cell attached to the registered P output implementing the operation "`(P >= <power-of-2>)`" will be transformed into using the DSP48E1's pattern detector feature for overflow detection.

```
-family {xcup|xcu|xc7|xc6v|xc5v|xc4v|xc6s|xc3sda}
    select the family to target
    default: xc7
```

10.284 xilinx_srl - Xilinx shift register extraction

yosys> help xilinx_srl

```
xilinx_srl [options] [selection]
```

This pass converts chains of built-in flops (bit-level: `$_DFF_[NP]_`, `$_DFFE_*` and word-level: `$dff`, `$dfffe`) as well as Xilinx flops (`FDRE`, `FDRE_1`) into a `$_XILINX_SHREG` cell. Chains must be of the same cell type, clock, clock polarity, enable, and enable polarity (where relevant). Flops with resets cannot be mapped to Xilinx devices and will not be inferred.

```
-minlen N
    min length of shift register (default = 3)

-fixed
    infer fixed-length shift registers.

-variable
    infer variable-length shift registers (i.e. fixed-length shifts where
    each element also fans-out to a $shiftx cell).
```

10.285 xprop - formal x propagation

yosys> help xprop

```
xprop [options] [selection]
```

This pass transforms the circuit into an equivalent circuit that explicitly encodes the propagation of x values using purely 2-valued logic. On the interface between xprop-transformed and non-transformed parts of the design, appropriate conversions are inserted automatically.

```
-split-inputs
-split-outputs
-split-ports
    Replace each input/output/port with two new ports, one carrying the
    defined values (named <portname>_d) and one carrying the mask of which
```

(continues on next page)

(continued from previous page)

bits are x (named <portname>_x). When a bit in the <portname>_x is set the corresponding bit in <portname>_d is ignored for inputs and guaranteed to be 0 for outputs.

-split-public

Replace each public non-port wire with two new wires, one carrying the defined values (named <wirename>_d) and one carrying the mask of which bits are x (named <wirename>_x). When a bit in the <portname>_x is set the corresponding bit in <wirename>_d is guaranteed to be 0 for outputs.

-assume-encoding

Add encoding invariants as assumptions. This can speed up formal verification tasks.

-assert-encoding

Add encoding invariants as assertions. Used for testing the xprop pass itself.

-assume-def-inputs

Assume all inputs are fully defined. This adds corresponding assumptions to the design and uses these assumptions to optimize the xprop encoding.

-required

Produce a runtime error if any encountered cell could not be encoded.

-formal

Produce a runtime error if any encoded cell uses a signal that is neither known to be non-x nor driven by another encoded cell.

-debug-asserts

Add assertions checking that the encoding used by this pass never produces x values within the encoded signals.

10.286 zinit - add inverters so all FF are zero-initialized

yosys> help zinit

zinit [options] [selection]

Add inverters as needed to make all FFs zero-initialized.

-all

also add zero initialization to uninitialized FFs

BIBLIOGRAPHY

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.
- [A+02] IEEE Standards Association and others. Ieee standard for verilog register transfer level synthesis. *IEEE Std 1364.1-2002*, 2002. doi:10.1109/IEEESTD.2002.94220.
- [A+04] IEEE Standards Association and others. Ieee standard for vhdl register transfer level (rtl) synthesis. *IEEE Std 1076.6-2004 (Revision of IEEE Std 1076.6-1999)*, 2004. doi:10.1109/IEEESTD.2004.94802.
- [A+06] IEEE Standards Association and others. Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006. doi:10.1109/IEEESTD.2006.99495.
- [A+09] IEEE Standards Association and others. Ieee standard vhdl language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 26 2009. doi:10.1109/IEEESTD.2009.4772740.
- [A+10] IEEE Standards Association and others. Ieee standard for ip-xact, standard structure for packaging, integrating, and reusing ip within tools flows. *IEEE Std 1685-2009*, pages C1–360, 2010. doi:10.1109/IEEESTD.2010.5417309.
- [BHSV90] R.K. Brayton, G.D. Hachtel, and A.L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, 1990. doi:10.1109/5.52213.
- [CI00] Clifford E. Cummings and Sunburst Design Inc. Nonblocking assignments in verilog synthesis, coding styles that kill. In *SNUG (Synopsys Users Group) 2000 User Papers, section-MC1 (1 st paper)*. 2000.
- [EenSorensson03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.
- [GW13] Johann Glaser and C. Wolf. Methodology and example-driven interconnect synthesis for designing heterogeneous coarse-grain reconfigurable architectures. In Jan Haase, editor, *Advances in Models, Methods, and Tools for Complex Chip Design — Selected contributions from FDL'12*. Springer, 2013.
- [HS96] G D Hachtel and F Somenzi. Logic synthesis and verification algorithms. 1996.
- [LHBB85] Kyu Y. Lee, Michael Holley, Mary Bailey, and Walter Bright. A high-level design language for programmable logic devices. *VLSI Design (Manhasset NY: CPM Publications)*, pages 50–62, June 1985.
- [STGR10] Yiqiong Shi, Chan Wai Ting, Bah-Hwee Gwee, and Ye Ren. A highly efficient method for extracting fsms from flattened gate-level netlist. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2610–2613. 2010. doi:10.1109/ISCAS.2010.5537093.

- [Ull76] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976. doi:10.1145/321921.321925.
- [Wol13] C. Wolf. Design and implementation of the yosys open synthesis suite. Bachelor Thesis, Vienna University of Technology, 2013.
- [WGS+12] C. Wolf, Johann Glaser, Florian Schupfer, Jan Haase, and Christoph Grimm. Example-driven interconnect synthesis for heterogeneous coarse-grain reconfigurable logic. In *FDL Proceeding of the 2012 Forum on Specification and Design Languages*, 194–201. 2012.

TAG INDEX

abc (*cmd/abc*), 395
abc9 (*cmd/abc9*), 399
abc9_exe (*cmd/abc9_exe*), 402
abc9_ops (*cmd/abc9_ops*), 403
abc_new (*cmd/abc_new*), 405
abstract (*cmd/abstract*), 406
add (*cmd/add*), 408
aigmap (*cmd/aigmap*), 408
alumacc (*cmd/alumacc*), 409
anlogic_eqn (*cmd/anlogic_eqn*), 409
anlogic_fixcarry (*cmd/anlogic_fixcarry*), 409
assertpmux (*cmd/assertpmux*), 409
async2sync (*cmd/async2sync*), 410
attrmap (*cmd/attrmap*), 410
attrmvcp (*cmd/attrmvcp*), 411
autoname (*cmd/autoname*), 411
blackbox (*cmd/blackbox*), 411
bmuxmap (*cmd/bmuxmap*), 412
booth (*cmd/booth*), 412
box_derive (*cmd/box_derive*), 412
bufnorm (*cmd/bufnorm*), 413
bugpoint (*cmd/bugpoint*), 414
bwmuxmap (*cmd/bwmuxmap*), 416
cd (*cmd/cd*), 416
cellmatch (*cmd/cellmatch*), 416
check (*cmd/check*), 417
chformal (*cmd/chformal*), 418
chparam (*cmd/chparam*), 419
chtype (*cmd/chtype*), 419
clean (*cmd/clean*), 419
clean_zerowidth (*cmd/clean_zerowidth*), 420
clk2fflogic (*cmd/clk2fflogic*), 420
clkbufmap (*cmd/clkbufmap*), 420
clockgate (*cmd/clockgate*), 421
connect (*cmd/connect*), 422
connect_rpc (*cmd/connect_rpc*), 422
connwrappers (*cmd/connwrappers*), 423
coolrunner2_fixup (*cmd/coolrunner2_fixup*), 424
coolrunner2_sop (*cmd/coolrunner2_sop*), 424
copy (*cmd/copy*), 424
cover (*cmd/cover*), 424
cutpoint (*cmd/cutpoint*), 425
debug (*cmd/debug*), 426
delete (*cmd/delete*), 426
deminout (*cmd/deminout*), 426
demuxmap (*cmd/demuxmap*), 426
design (*cmd/design*), 427
dffinit (*cmd/dffinit*), 428
dfflegalize (*cmd/dfflegalize*), 429
dfflibmap (*cmd/dfflibmap*), 430
dffunmap (*cmd/dffunmap*), 431
dft_tag (*cmd/dft_tag*), 431
dump (*cmd/dump*), 431
echo (*cmd/echo*), 432
edgetypes (*cmd/edgetypes*), 432
efinix_fixcarry (*cmd/efinix_fixcarry*), 432
equiv_add (*cmd/equiv_add*), 433
equiv_induct (*cmd/equiv_induct*), 433
equiv_make (*cmd/equiv_make*), 434
equiv_mark (*cmd/equiv_mark*), 434
equiv_miter (*cmd/equiv_miter*), 434
equiv_opt (*cmd/equiv_opt*), 435
equiv_purge (*cmd/equiv_purge*), 436
equiv_remove (*cmd/equiv_remove*), 436
equiv_simple (*cmd/equiv_simple*), 437
equiv_status (*cmd/equiv_status*), 437
equiv_struct (*cmd/equiv_struct*), 437
eval (*cmd/eval*), 438
example_dt (*cmd/example_dt*), 438
exec (*cmd/exec*), 439
expose (*cmd/expose*), 439
extract (*cmd/extract*), 440
extract_counter (*cmd/extract_counter*), 442
extract_fa (*cmd/extract_fa*), 442
extract_reduce (*cmd/extract_reduce*), 443
extractinv (*cmd/extractinv*), 443
flatten (*cmd/flatten*), 443
flowmap (*cmd/flowmap*), 444
fmcombine (*cmd/fmcombine*), 445
fminit (*cmd/fminit*), 446
formalff (*cmd/formalff*), 446
freduce (*cmd/freduce*), 447
fsm (*cmd/fsm*), 448
fsm_detect (*cmd/fsm_detect*), 449

fsm_expand (*cmd/fsm_expand*), 449
fsm_export (*cmd/fsm_export*), 449
fsm_extract (*cmd/fsm_extract*), 450
fsm_info (*cmd/fsm_info*), 450
fsm_map (*cmd/fsm_map*), 450
fsm_opt (*cmd/fsm_opt*), 451
fsm_recode (*cmd/fsm_recode*), 451
fst2tb (*cmd/fst2tb*), 451
future (*cmd/future*), 452
gatemate_foldinv (*cmd/gatemate_foldinv*), 452
glift (*cmd/glift*), 452
greenpak4_dffinv (*cmd/greenpak4_dffinv*), 454
help (*cmd/help*), 454
hierarchy (*cmd/hierarchy*), 455
hilomap (*cmd/hilomap*), 456
history (*cmd/history*), 457
ice40_braminit (*cmd/ice40_braminit*), 457
ice40_dsp (*cmd/ice40_dsp*), 457
ice40_opt (*cmd/ice40_opt*), 458
ice40_wrapcarry (*cmd/ice40_wrapcarry*), 458
insbuf (*cmd/insbuf*), 458
internal_stats (*cmd/internal_stats*), 459
iopadmap (*cmd/iopadmap*), 459
jny (*cmd/jny*), 460
json (*cmd/json*), 460
keep_hierarchy (*cmd/keep_hierarchy*), 461
lattice_gsr (*cmd/lattice_gsr*), 461
license (*cmd/license*), 461
log (*cmd/log*), 462
logger (*cmd/logger*), 463
ls (*cmd/ls*), 464
ltp (*cmd/ltp*), 464
lut2mux (*cmd/lut2mux*), 464
maccmap (*cmd/maccmap*), 464
memory (*cmd/memory*), 465
memory_bmux2rom (*cmd/memory_bmux2rom*), 465
memory_bram (*cmd/memory_bram*), 465
memory_collect (*cmd/memory_collect*), 467
memory_dff (*cmd/memory_dff*), 468
memory_libmap (*cmd/memory_libmap*), 468
memory_map (*cmd/memory_map*), 469
memory_memx (*cmd/memory_memx*), 469
memory_narrow (*cmd/memory_narrow*), 470
memory_nordff (*cmd/memory_nordff*), 470
memory_share (*cmd/memory_share*), 470
memory_unpack (*cmd/memory_unpack*), 471
microchip_dffopt (*cmd/microchip_dffopt*), 471
microchip_dsp (*cmd/microchip_dsp*), 471
miter (*cmd/miter*), 472
mutate (*cmd/mutate*), 473
muxcover (*cmd/muxcover*), 474
muxpack (*cmd/muxpack*), 475
nlutmap (*cmd/nlutmap*), 475
nx_carry (*cmd/nx_carry*), 475
onehot (*cmd/onehot*), 476
opt (*cmd/opt*), 476
opt_clean (*cmd/opt_clean*), 477
opt_demorgan (*cmd/opt_demorgan*), 477
opt_dff (*cmd/opt_dff*), 477
opt_expr (*cmd/opt_expr*), 478
opt_ffinv (*cmd/opt_ffinv*), 479
opt_lut (*cmd/opt_lut*), 479
opt_lut_ins (*cmd/opt_lut_ins*), 479
opt_mem (*cmd/opt_mem*), 480
opt_mem_feedback (*cmd/opt_mem_feedback*), 480
opt_mem_priority (*cmd/opt_mem_priority*), 480
opt_mem_widen (*cmd/opt_mem_widen*), 480
opt_merge (*cmd/opt_merge*), 481
opt_muxtree (*cmd/opt_muxtree*), 481
opt_reduce (*cmd/opt_reduce*), 481
opt_share (*cmd/opt_share*), 482
paramap (*cmd/paramap*), 482
peepopt (*cmd/peepopt*), 483
plugin (*cmd/plugin*), 483
pmux2shiftx (*cmd/pmux2shiftx*), 484
pmuxtree (*cmd/pmuxtree*), 484
portarcs (*cmd/portarcs*), 484
portlist (*cmd/portlist*), 485
prep (*cmd/prep*), 485
printattrs (*cmd/printattrs*), 486
proc (*cmd/proc*), 487
proc_arst (*cmd/proc_arst*), 487
proc_clean (*cmd/proc_clean*), 488
proc_dff (*cmd/proc_dff*), 488
proc_dlatch (*cmd/proc_dlatch*), 488
proc_init (*cmd/proc_init*), 489
proc_memwr (*cmd/proc_memwr*), 489
proc_mux (*cmd/proc_mux*), 489
proc_prune (*cmd/proc_prune*), 489
proc_rmdead (*cmd/proc_rmdead*), 490
proc_rom (*cmd/proc_rom*), 490
qbfsat (*cmd/qbfsat*), 490
ql_bram_merge (*cmd/ql_bram_merge*), 492
ql_bram_types (*cmd/ql_bram_types*), 492
ql_dsp_io_regs (*cmd/ql_dsp_io_regs*), 492
ql_dsp_macc (*cmd/ql_dsp_macc*), 492
ql_dsp_simd (*cmd/ql_dsp_simd*), 493
ql_ioff (*cmd/ql_ioff*), 493
read (*cmd/read*), 493
read_aiger (*cmd/read_aiger*), 494
read_blif (*cmd/read_blif*), 495
read_json (*cmd/read_json*), 495
read_liberty (*cmd/read_liberty*), 495
read_rtlil (*cmd/read_rtlil*), 496
read_verilog (*cmd/read_verilog*), 496
read_xaiger2 (*cmd/read_xaiger2*), 500
recover_names (*cmd/recover_names*), 500
rename (*cmd/rename*), 500

rmports (*cmd/rmports*), 502
 sat (*cmd/sat*), 502
 scatter (*cmd/scatter*), 505
 scc (*cmd/scc*), 505
 scratchpad (*cmd/scratchpad*), 506
 script (*cmd/script*), 507
 select (*cmd/select*), 507
 setattr (*cmd/setattr*), 512
 setenv (*cmd/setenv*), 512
 setparam (*cmd/setparam*), 513
 setundef (*cmd/setundef*), 513
 share (*cmd/share*), 514
 shell (*cmd/shell*), 514
 show (*cmd/show*), 515
 shregmap (*cmd/shregmap*), 517
 sim (*cmd/sim*), 518
 simplemap (*cmd/simplemap*), 520
 splice (*cmd/splice*), 520
 splitcells (*cmd/splitcells*), 521
 splitnets (*cmd/splitnets*), 522
 sta (*cmd/sta*), 522
 stat (*cmd/stat*), 522
 submod (*cmd/submod*), 523
 supercover (*cmd/supercover*), 524
 synth (*cmd/synth*), 524
 synth_achronix (*cmd/synth_achronix*), 526
 synth_anlogic (*cmd/synth_anlogic*), 527
 synth_coolrunner2 (*cmd/synth_coolrunner2*), 529
 synth_easic (*cmd/synth_easic*), 531
 synth_ecp5 (*cmd/synth_ecp5*), 532
 synth_efinix (*cmd/synth_efinix*), 536
 synth_fabulous (*cmd/synth_fabulous*), 538
 synth_gatamate (*cmd/synth_gatamate*), 541
 synth_gowin (*cmd/synth_gowin*), 544
 synth_greenpak4 (*cmd/synth_greenpak4*), 546
 synth_ice40 (*cmd/synth_ice40*), 548
 synth_intel (*cmd/synth_intel*), 552
 synth_intel_alm (*cmd/synth_intel_alm*), 554
 synth_lattice (*cmd/synth_lattice*), 557
 synth_microchip (*cmd/synth_microchip*), 560
 synth_nanoxplore (*cmd/synth_nanoxplore*), 563
 synth_nexus (*cmd/synth_nexus*), 566
 synth_quicklogic (*cmd/synth_quicklogic*), 569
 synth_sf2 (*cmd/synth_sf2*), 572
 synth_xilinx (*cmd/synth_xilinx*), 574
 synthprop (*cmd/synthprop*), 578
 tcl (*cmd/tcl*), 579
 techmap (*cmd/techmap*), 579
 tee (*cmd/tee*), 583
 test_abcloop (*cmd/test_abcloop*), 583
 test_autotb (*cmd/test_autotb*), 583
 test_cell (*cmd/test_cell*), 584
 test_generic (*cmd/test_generic*), 585
 test_pmgen (*cmd/test_pmgen*), 586
 torder (*cmd/torder*), 586
 trace (*cmd/trace*), 587
 tribuf (*cmd/tribuf*), 587
 uniquify (*cmd/uniquify*), 587
 verific (*cmd/verific*), 588
 verilog_defaults (*cmd/verilog_defaults*), 590
 verilog_defines (*cmd/verilog_defines*), 590
 viz (*cmd/viz*), 591
 wbflip (*cmd/wbflip*), 592
 wrapcell (*cmd/wrapcell*), 592
 wreduce (*cmd/wreduce*), 593
 write_aiger (*cmd/write_aiger*), 593
 write_aiger2 (*cmd/write_aiger2*), 594
 write_blif (*cmd/write_blif*), 595
 write_btor (*cmd/write_btor*), 596
 write_cxxrtl (*cmd/write_cxxrtl*), 597
 write_edif (*cmd/write_edif*), 601
 write_file (*cmd/write_file*), 602
 write_firrtl (*cmd/write_firrtl*), 603
 write_functional_cxx
 (*cmd/write_functional_cxx*), 603
 write_functional_rosette
 (*cmd/write_functional_rosette*), 603
 write_functional_smt2
 (*cmd/write_functional_smt2*), 603
 write_intersynth (*cmd/write_intersynth*), 604
 write_jny (*cmd/write_jny*), 604
 write_json (*cmd/write_json*), 605
 write_rtlil (*cmd/write_rtlil*), 610
 write_simplec (*cmd/write_simplec*), 610
 write_smt2 (*cmd/write_smt2*), 610
 write_smv (*cmd/write_smv*), 613
 write_spice (*cmd/write_spice*), 614
 write_table (*cmd/write_table*), 614
 write_verilog (*cmd/write_verilog*), 615
 write_xaiger (*cmd/write_xaiger*), 617
 write_xaiger2 (*cmd/write_xaiger2*), 617
 xilinx_dffopt (*cmd/xilinx_dffopt*), 618
 xilinx_dsp (*cmd/xilinx_dsp*), 618
 xilinx_srl (*cmd/xilinx_srl*), 619
 xprop (*cmd/xprop*), 619
 zinit (*cmd/zinit*), 620

COMMAND REFERENCE

Command

abc, 395
abc9, 399
abc9_exe, 402
abc9_ops, 403
abc_new, 405
abstract, 406
add, 408
aigmap, 408
alumacc, 409
anlogic_eqn, 409
anlogic_fixcarry, 409
assertpmux, 409
async2sync, 410
attrmap, 410
attrmvcp, 411
autoname, 411
blackbox, 411
bmuxmap, 412
booth, 412
box_derive, 412
bufnorm, 413
bugpoint, 414
bwmuxmap, 416
cd, 416
cellmatch, 416
check, 417
chformal, 418
chparam, 419
chtype, 419
clean, 419
clean_zerowidth, 420
clk2fflogic, 420
clkbufmap, 420
clockgate, 421
connect, 422
connect_rpc, 422
connwrappers, 423
coolrunner2_fixup, 424
coolrunner2_sop, 424
copy, 424
cover, 424
cutpoint, 425
debug, 426
delete, 426
deminout, 426
demuxmap, 426
design, 427
dffinit, 428
dfflegalize, 429
dfflibmap, 430
dffunmap, 431
dft_tag, 431
dump, 431
echo, 432
edgetypes, 432
efinix_fixcarry, 432
equiv_add, 433
equiv_induct, 433
equiv_make, 434
equiv_mark, 434
equiv_miter, 434
equiv_opt, 435
equiv_purge, 436
equiv_remove, 436
equiv_simple, 437
equiv_status, 437
equiv_struct, 437
eval, 438
example_dt, 438
exec, 439
expose, 439
extract, 440
extract_counter, 442
extract_fa, 442
extract_reduce, 443
extractinv, 443
flatten, 443
flowmap, 444
fmcombine, 445
fminit, 446
formalff, 446
freduce, 447
fsm, 448

fsm_detect, 449
fsm_expand, 449
fsm_export, 449
fsm_extract, 450
fsm_info, 450
fsm_map, 450
fsm_opt, 451
fsm_recode, 451
fst2tb, 451
future, 452
gatamate_foldinv, 452
glift, 452
greenpak4_dffinv, 454
help, 454
hierarchy, 455
hilomap, 456
history, 457
ice40_braminit, 457
ice40_dsp, 457
ice40_opt, 458
ice40_wrapcarry, 458
insbuf, 458
internal_stats, 459
iopadmap, 459
jny, 460
json, 460
keep_hierarchy, 461
lattice_gsr, 461
license, 461
log, 462
logger, 463
ls, 464
ltp, 464
lut2mux, 464
maccmap, 464
memory, 465
memory_bmux2rom, 465
memory_bram, 465
memory_collect, 467
memory_dff, 468
memory_libmap, 468
memory_map, 469
memory_memx, 469
memory_narrow, 470
memory_nordff, 470
memory_share, 470
memory_unpack, 471
microchip_dffopt, 471
microchip_dsp, 471
miter, 472
mutate, 473
muxcover, 474
muxpack, 475
nlutmap, 475
nx_carry, 475
onehot, 476
opt, 476
opt_clean, 477
opt_demorgan, 477
opt_dff, 477
opt_expr, 478
opt_ffinv, 479
opt_lut, 479
opt_lut_ins, 479
opt_mem, 480
opt_mem_feedback, 480
opt_mem_priority, 480
opt_mem_widen, 480
opt_merge, 481
opt_muxtree, 481
opt_reduce, 481
opt_share, 482
paramap, 482
peepopt, 483
plugin, 483
pmux2shiftx, 484
pmuxtree, 484
portarcs, 484
portlist, 485
prep, 485
printattrs, 486
proc, 487
proc_arst, 487
proc_clean, 488
proc_dff, 488
proc_dlatch, 488
proc_init, 489
proc_memwr, 489
proc_mux, 489
proc_prune, 489
proc_rmdead, 490
proc_rom, 490
qbfsat, 490
ql_bram_merge, 492
ql_bram_types, 492
ql_dsp_io_regs, 492
ql_dsp_macc, 492
ql_dsp_simd, 493
ql_ioff, 493
read, 493
read_aiger, 494
read_blif, 495
read_json, 495
read_liberty, 495
read_rtlil, 496
read_verilog, 496
read_xaiger2, 500
recover_names, 500

rename, 500
rmports, 502
sat, 502
scatter, 505
scc, 505
scratchpad, 506
script, 507
select, 507
setattr, 512
setenv, 512
setparam, 513
setundef, 513
share, 514
shell, 514
show, 515
shregmap, 517
sim, 518
simplemap, 520
splice, 520
splitcells, 521
splitnets, 522
sta, 522
stat, 522
submod, 523
supercover, 524
synth, 524
synth_achronix, 526
synth_anlogic, 527
synth_coolrunner2, 529
synth_easic, 531
synth_ecp5, 532
synth_efinix, 536
synth_fabulous, 538
synth_gatamate, 541
synth_gowin, 544
synth_greenpak4, 546
synth_ice40, 548
synth_intel, 552
synth_intel_alm, 554
synth_lattice, 557
synth_microchip, 560
synth_nanoxplore, 563
synth_nexus, 566
synth_quicklogic, 569
synth_sf2, 572
synth_xilinx, 574
synthprop, 578
tcl, 579
techmap, 579
tee, 583
test_abcloop, 583
test_autotb, 583
test_cell, 584
test_generic, 585
test_pmggen, 586
torder, 586
trace, 587
tribuf, 587
uniquify, 587
verific, 588
verilog_defaults, 590
verilog_defines, 590
viz, 591
wbflip, 592
wrapcell, 592
wreduce, 593
write_aiger, 593
write_aiger2, 594
write_blif, 595
write_btor, 596
write_cxxrtl, 597
write_edif, 601
write_file, 602
write_firrtl, 603
write_functional_cxx, 603
write_functional_rosette, 603
write_functional_smt2, 603
write_intersynth, 604
write_jny, 604
write_json, 605
write_rtlil, 610
write_simplec, 610
write_smt2, 610
write_smv, 613
write_spice, 614
write_table, 614
write_verilog, 615
write_xaiger, 617
write_xaiger2, 617
xilinx_dffopt, 618
xilinx_dsp, 618
xilinx_srl, 619
xprop, 619
zinit, 620

PROPERTY INDEX

is_
is_evaluable, 391
\$alu, 290
\$fa, 291
\$lcu, 291
\$macc, 292
\$add, 242
\$and, 243
\$bweqx, 244
\$div, 244
\$divfloor, 245
\$eq, 246
\$eqx, 246
\$ge, 247
\$gt, 248
\$le, 248
\$logic_and, 249
\$logic_or, 249
\$lt, 250
\$mod, 251
\$modfloor, 251
\$mul, 252
\$ne, 253
\$nex, 253
\$or, 254
\$pow, 255
\$shift, 255
\$shiftx, 256
\$shl, 257
\$shr, 257
\$sshl, 258
\$sshr, 259
\$sub, 259
\$xnor, 260
\$xor, 260
\$_ANDNOT_, 318
\$_AOI3_, 319
\$_AOI4_, 319
\$_MUX16_, 320
\$_MUX4_, 321
\$_MUX8_, 321
\$_NMUX_, 322
\$_OAI3_, 322
\$_OAI4_, 323
\$_ORNOT_, 324
\$_AND_, 314
\$_BUF_, 314
\$_MUX_, 314
\$_NAND_, 315
\$_NOR_, 315
\$_NOT_, 316
\$_OR_, 316
\$_XNOR_, 316
\$_XOR_, 317
\$allconst, 302
\$allseq, 302
\$anyconst, 303
\$anyseq, 303
\$assert, 304
\$assume, 304
\$cover, 305
\$equiv, 305
\$fair, 305
\$initstate, 306
\$live, 306
\$_TBUF_, 391
\$lut, 296
\$sop, 296
\$bmux, 261
\$bwmux, 262
\$demux, 263
\$mux, 263
\$pmux, 264
\$tribuf, 265
\$specify2, 297
\$specify3, 298
\$specrule, 301
\$buf, 236
\$logic_not, 236
\$neg, 237
\$not, 237
\$pos, 238
\$reduce_and, 238
\$reduce_bool, 239

\$reduce_or, 240
\$reduce_xnor, 240
\$reduce_xor, 241
\$concat, 312
\$slice, 313

X-

x-aware, 391
\$bweqx, 244
\$eqx, 246
\$nex, 253
x-output, 391
\$div, 244
\$mod, 251
\$shiftx, 256
\$pmux, 264

INTERNAL CELL REFERENCE

Internal cell

\$alu, 290
\$fa, 291
\$lcu, 291
\$macc, 292
\$add, 242
\$and, 243
\$bweqx, 244
\$div, 244
\$divfloor, 245
\$eq, 246
\$eqx, 246
\$ge, 247
\$gt, 248
\$le, 248
\$logic_and, 249
\$logic_or, 249
\$lt, 250
\$mod, 251
\$modfloor, 251
\$mul, 252
\$ne, 253
\$nex, 253
\$or, 254
\$pow, 255
\$shift, 255
\$shiftx, 256
\$shl, 257
\$shr, 257
\$sshl, 258
\$sshr, 259
\$sub, 259
\$xnor, 260
\$xor, 260
\$_ANDNOT_, 318
\$_AOI3_, 319
\$_AOI4_, 319
\$_MUX16_, 320
\$_MUX4_, 321
\$_MUX8_, 321
\$_NMUX_, 322
\$_OAI3_, 322
\$_OAI4_, 323
\$_ORNOT_, 324
\$_AND_, 314
\$_BUF_, 314
\$_MUX_, 314
\$_NAND_, 315
\$_NOR_, 315
\$_NOT_, 316
\$_OR_, 316
\$_XNOR_, 316
\$_XOR_, 317
\$check, 309
\$print, 310
\$scopeinfo, 310
\$allconst, 302
\$allseq, 302
\$anyconst, 303
\$anyinit, 303
\$anyseq, 303
\$assert, 304
\$assume, 304
\$cover, 305
\$equiv, 305
\$fair, 305
\$ff, 306
\$initstate, 306
\$live, 306
\$future_ff, 307
\$get_tag, 307
\$original_tag, 307
\$overwrite_tag, 308
\$set_tag, 308
\$fsm, 287
\$_TBUF_, 391
\$lut, 296
\$sop, 296
\$mem, 279
\$mem_v2, 281
\$meminit, 283
\$meminit_v2, 284
\$memrd, 284
\$memrd_v2, 285

\$memwr, 286	\$_DFFE_PP_, 341
\$memwr_v2, 286	\$_DFFSRE_NNNN_, 342
\$bmux, 261	\$_DFFSRE_NNNP_, 342
\$bwmux, 262	\$_DFFSRE_NNPN_, 343
\$demux, 263	\$_DFFSRE_NNPP_, 343
\$mux, 263	\$_DFFSRE_NPNN_, 344
\$pmux, 264	\$_DFFSRE_NPNP_, 344
\$tribuf, 265	\$_DFFSRE_NPPN_, 345
\$adff, 267	\$_DFFSRE_NPPP_, 345
\$adffe, 267	\$_DFFSRE_PNNN_, 346
\$adlatch, 268	\$_DFFSRE_PNNP_, 346
\$aldff, 268	\$_DFFSRE_PNPN_, 347
\$aldffe, 269	\$_DFFSRE_PNPP_, 347
\$dff, 269	\$_DFFSRE_PPNN_, 348
\$dffe, 270	\$_DFFSRE_PPNP_, 349
\$dffsr, 270	\$_DFFSRE_PPPN_, 349
\$dffsre, 271	\$_DFFSRE_PPPP_, 350
\$dlatch, 272	\$_DFFSR_NNN_, 350
\$dlatchsr, 272	\$_DFFSR_NNP_, 351
\$sdff, 273	\$_DFFSR_NPN_, 351
\$sdffce, 273	\$_DFFSR_NPP_, 352
\$sdffe, 274	\$_DFFSR_PNN_, 352
\$sr, 274	\$_DFFSR_PNP_, 353
\$_ALDFFE_NNN_, 327	\$_DFFSR_PPN_, 353
\$_ALDFFE_NNP_, 328	\$_DFFSR_PPP_, 354
\$_ALDFFE_NPN_, 328	\$_DFF_NNO_, 354
\$_ALDFFE_NPP_, 329	\$_DFF_NN1_, 355
\$_ALDFFE_PNN_, 329	\$_DFF_NPO_, 355
\$_ALDFFE_PNP_, 330	\$_DFF_NP1_, 356
\$_ALDFFE_PPN_, 330	\$_DFF_N_, 356
\$_ALDFFE_PPP_, 331	\$_DFF_PNO_, 356
\$_ALDFF_NN_, 331	\$_DFF_PN1_, 357
\$_ALDFF_NP_, 332	\$_DFF_PPO_, 357
\$_ALDFF_PN_, 332	\$_DFF_PP1_, 358
\$_ALDFF_PP_, 332	\$_DFF_P_, 358
\$_DFFE_NNON_, 333	\$_FF_, 359
\$_DFFE_NNOP_, 333	\$_SDFFCE_NNON_, 359
\$_DFFE_NN1N_, 334	\$_SDFFCE_NNOP_, 359
\$_DFFE_NN1P_, 334	\$_SDFFCE_NN1N_, 360
\$_DFFE_NN_, 335	\$_SDFFCE_NN1P_, 360
\$_DFFE_NPON_, 335	\$_SDFFCE_NPON_, 361
\$_DFFE_NPOP_, 336	\$_SDFFCE_NPOP_, 361
\$_DFFE_NP1N_, 336	\$_SDFFCE_NP1N_, 362
\$_DFFE_NP1P_, 336	\$_SDFFCE_NP1P_, 362
\$_DFFE_NP_, 337	\$_SDFFCE_PNON_, 363
\$_DFFE_PNON_, 337	\$_SDFFCE_PNOP_, 363
\$_DFFE_PNOP_, 338	\$_SDFFCE_PN1N_, 364
\$_DFFE_PN1N_, 338	\$_SDFFCE_PN1P_, 364
\$_DFFE_PN1P_, 339	\$_SDFFCE_PPON_, 365
\$_DFFE_PN_, 339	\$_SDFFCE_PPOP_, 365
\$_DFFE_PPON_, 339	\$_SDFFCE_PP1N_, 366
\$_DFFE_PPOP_, 340	\$_SDFFCE_PP1P_, 366
\$_DFFE_PP1N_, 340	\$_SDFFE_NNON_, 367
\$_DFFE_PP1P_, 341	\$_SDFFE_NNOP_, 367

<code>\$_SDFFE_NN1N_</code> , 368	<code>\$reduce_or</code> , 240
<code>\$_SDFFE_NN1P_</code> , 368	<code>\$reduce_xnor</code> , 240
<code>\$_SDFFE_NPON_</code> , 369	<code>\$reduce_xor</code> , 241
<code>\$_SDFFE_NPOP_</code> , 369	<code>\$concat</code> , 312
<code>\$_SDFFE_NP1N_</code> , 370	<code>\$slice</code> , 313
<code>\$_SDFFE_NP1P_</code> , 370	
<code>\$_SDFFE_PNON_</code> , 371	
<code>\$_SDFFE_PNOP_</code> , 371	
<code>\$_SDFFE_PN1N_</code> , 372	
<code>\$_SDFFE_PN1P_</code> , 372	
<code>\$_SDFFE_PPON_</code> , 373	
<code>\$_SDFFE_PPOP_</code> , 373	
<code>\$_SDFFE_PP1N_</code> , 374	
<code>\$_SDFFE_PP1P_</code> , 374	
<code>\$_SDFF_NNO_</code> , 375	
<code>\$_SDFF_NN1_</code> , 375	
<code>\$_SDFF_NPO_</code> , 376	
<code>\$_SDFF_NP1_</code> , 376	
<code>\$_SDFF_PNO_</code> , 377	
<code>\$_SDFF_PN1_</code> , 377	
<code>\$_SDFF_PPO_</code> , 377	
<code>\$_SDFF_PP1_</code> , 378	
<code>\$_DLATCHSR_NNN_</code> , 380	
<code>\$_DLATCHSR_NNP_</code> , 381	
<code>\$_DLATCHSR_NPN_</code> , 381	
<code>\$_DLATCHSR_NPP_</code> , 382	
<code>\$_DLATCHSR_PNN_</code> , 382	
<code>\$_DLATCHSR_PNP_</code> , 383	
<code>\$_DLATCHSR_PPN_</code> , 383	
<code>\$_DLATCHSR_PPP_</code> , 384	
<code>\$_DLATCH_NNO_</code> , 384	
<code>\$_DLATCH_NN1_</code> , 385	
<code>\$_DLATCH_NPO_</code> , 385	
<code>\$_DLATCH_NP1_</code> , 386	
<code>\$_DLATCH_N_</code> , 386	
<code>\$_DLATCH_PNO_</code> , 386	
<code>\$_DLATCH_PN1_</code> , 387	
<code>\$_DLATCH_PPO_</code> , 387	
<code>\$_DLATCH_PP1_</code> , 388	
<code>\$_DLATCH_P_</code> , 388	
<code>\$_SR_NN_</code> , 389	
<code>\$_SR_NP_</code> , 389	
<code>\$_SR_PN_</code> , 390	
<code>\$_SR_PP_</code> , 390	
<code>\$specify2</code> , 297	
<code>\$specify3</code> , 298	
<code>\$specrule</code> , 301	
<code>\$buf</code> , 236	
<code>\$logic_not</code> , 236	
<code>\$neg</code> , 237	
<code>\$not</code> , 237	
<code>\$pos</code> , 238	
<code>\$reduce_and</code> , 238	
<code>\$reduce_bool</code> , 239	