Logic Synthesis with Yosys

N. Engelhardt, YosysHQ

About me

- Started out in CS
- BSc 2008, MSc 2010 of computer science at ENS Rennes, France
- Wandered around slowly getting closer to hardware
- Universität Potsdam (2010-2011) and TU Berlin (2011-2014)
- Finally ended up in EEE dept
- PhD 2019 at University of Hong Kong
    - Graph processing framework for FPGAs
- Started to work for Symbiotic EDA in Vienna after graduation
- Jan 2021: founded YosysHQ with the Yosys dev team members from SEDA
    - 8 team members, plus associated
- Now spend about 50/50 management/coding

YosysHQ

About Yosys

- "the GCC of hardware synthesis"
- Created by Claire Wolf
- Started in late 2012 as a BSc thesis project
- Intended to support synthesis for a CGRA by a PhD student in the group
- Expanded into more general infrastructure for research on synthesis
- Now has full Verilog-2005 (synthesizable subset) support
- Together with the P&R tool nextpnr, can program some FPGAs with a fully end-to-end open source flow (Lattice iCE40 and ECP5)
- Also does the synthesis portion for the OpenLANE flow targeting the SkyWater 130nm open source PDK for fully open source ASIC design
- Can also do formal verification with backends for solver formats like SMT2

About YosysHQ GmbH

- Not a startup - we don't take investment and aren't looking to be acquired
- Just trying to earn enough money to keep the devteam working on Yosys-related projects as our day job
- Multiple income sources:
  - Tabby CAD Suite licenses - integrates the Verific frontend with Yosys for support of SystemVerilog, VHDL, and SVA properties
    - request an eval license here: https://www.yosyshq.com/contact
  - Support and consulting for companies/researchers integrating Yosys in their own projects
  - Development projects for companies/researchers that want to have new features added to Yosys or the Yosys-based tools
- Recent activities:
  - New tool EQY for sequential equivalence checking of two circuits
  - Extending riscv-formal properties to privilege spec
  - Adding new FPGA architecture support to nextpnr

YosysHQ

Table of Contents

Goal: Learn how to use Yosys to analyze or transform netlists (e.g. to bridge between two tools)

- User view
  - Outline of a synthesis flow
  - Techmap
- Developer view
  - RTLIL
  - Build environment for plugins
  - How to write a custom pass

Resources

- Yosys CLI "help foo"
  - including descriptions for internal cells: `help $and` / `help $and+`
- Manual (yosys.readthedocs.io)
- Website (yosyshq.net/yosys/)
- GitHub (github.com/YosysHQ/yosys)
  - sadly, the source is the only real way to find out some things
  - searching past PRs can give insight about how things are supposed to work and why they are the way they are
- If you have a laptop/mobile device, read along in the code

YosysHQ

Interacting with Yosys

- Yosys is a collection of passes
- Each pass executes a certain operation on the in-memory design representation (RTLIL)
- The art of using Yosys is to know the right order to call the passes in
  - unlike e.g. LLVM, Yosys has no tracking of pass dependencies and very few sanity checks
  - you can get suboptimal and sometimes incoherent results by swapping order of two passes or forgetting a pass
- Input options:
  - CLI interactive mode: for playing around with the design, investigating what is happening
  - script file (.ys): for repeatable actions, changing one operation in a longer sequence
  - script pass (C++): for macro-operations that need conditional execution of passes

Outline of a Synthesis Flow

- Yosys is a synthesis tool - it only knows netlists
- There are many netlist representations for different stages of transformations
- Depending on your use case, you will need a certain representation
- Getting to a representation requires a flow, which is a script containing a sequence of calls of Yosys passes
  - There are a few built-in script passes that we can use as examples
- Let's walk through a basic synthesis flow (`synth_ice40`)
  - run `yosys -h synth_ice40` to read along
  - or find the same output in the command reference: https://yosyshq.readthedocs.io/projects/yosys/en/latest/cmd/synth_ice40.html

YosysHQ

Stages of the flow

- Load design
- Elaborate design hierarchy
- Coarse-grain optimization
- Hard block mapping
- Fine-grain logic mapping
- Fine-grain logic optimization
- Write netlist

Frontends

- 1st step: read in the source code
- Many frontends:
  - `read_verilog`, GHDL (plugin)
  - RTLIL (direct textual representation of yosys internal state)
  - aiger, blif, json, liberty
- Most will directly produce internal netlist representations of modules containing cells and wires
- But some (`read_verilog`, Amaranth-generated RTLIL) will have more "behavioral-style" representations with processes

```
read_verilog …
read_rtlil …
read_liberty …

read_verilog -D ICE40_HX
    -lib -specify
    +/ice40/cells_sim.v
```

Hierarchy

- Once you have the basic design entities, need to establish the connections between instantiations
- `hierarchy` checks that all the instantiated modules are defined
- It sets the attribute `(* top *)` on the module `<top>` (important)
- It deletes unused non-blackbox modules
- For parameterized modules, it derives the module for concrete parameter values used in instances

```
hierarchy -check
    -top <top>
```

Elaboration

- `proc` changes processes into basic logic
- `flatten` removes the hierarchy and puts everything into one module `top_mod`
- `tribuf -logic` and `deminout` remove tristate and inout constructs that can't be mapped to FPGA
- `check` looks for problems in the design (e.g. multiple drivers)

```
proc
flatten
tribuf -logic
deminout
opt_expr
opt_clean
check
```

Coarse-grain

- Now the design is in coarse-grain representation
- Still looks recognizable
- Cells are word-level operators from the internal cell library with parametrizable width (`$or`, `$add`, `$mul`, `$dff`...)
- At this level happen optimizations like const propagation, expression rewriting, trimming unused parts of wires
- Many formal backends (`write_smt2`, `write_btor`) use this representation

```
opt -nodffe -nosdff
fsm
opt
wreduce
peepopt
opt_clean
share
techmap -map
+/cmp2lut.v
    -D LUT_WIDTH=4
opt_expr
opt_clean
```

YosysHQ

opt

- Many of the common optimizations are combined in a script pass, opt
- This runs a fixed-point iteration:

```
opt_expr
opt_merge -nomux

do
    opt_muxtree
    opt_reduce
    opt_merge
    opt_share
    opt_dff
    opt_clean
    opt_expr
while <changed design>
```

Recognizing special constructs

- FSMs and hard blocks like DSP or memories and even FFs are not first-class objects in RTL
- They have to be *inferred* from patterns in the design
- Special passes to handle each
  - Facilities for building these passes: `techmap` and `pmgen`
    - `techmap` matches one cell, replaces it with one or more cells
    - `pmgen` recognizes patterns composed of multiple cells
- Detection of patterns can be affected by optimizations/transformations done previously
- E.g. enable and reset of FFs come out of "if" in always blocks and are initially (after `proc`) just muxes before the FF
- `opt_dff` can fold them in
  - fsm needs only a certain subset of FF types to be used to detect FSM-like patterns

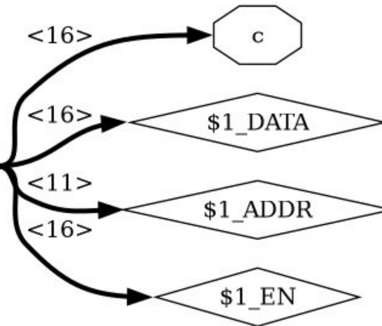
```
memory_dff
wreduce t:$mul
techmap -map
    +/mul2dsp.v -map
    +/ice40/dsp_map.v
    [..]
select a:mul2dsp
setattr -unset mul2dsp
opt_expr -fine
wreduce
select -clear
ice40_dsp
chtype -set $mul
    t:$__soft_mul
alumacc
opt
```

Example for stepping through `synth_ice40`

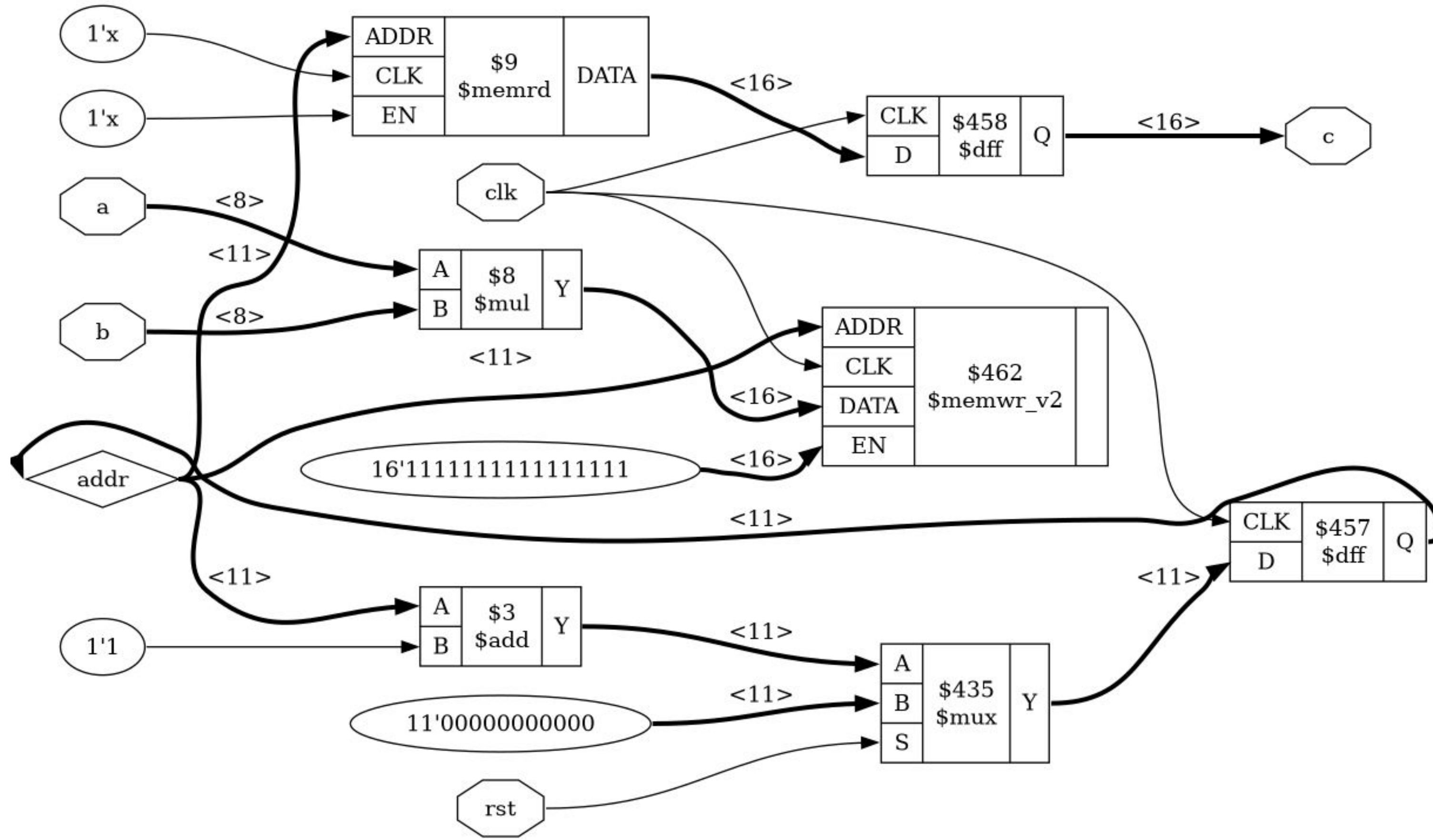
```
module example (
    input clk,
    input rst,
    input [7:0] a,
    input [7:0] b,
    output reg [15:0] c
);

reg [15:0] mem [256:0];
reg [10:0] addr;
```

```
always @ (posedge clk) begin
    if(rst) addr <= 0;
    else addr <= addr + 1'b1;
end

always @ (posedge clk) begin
    mem[addr] <= a * b;
    c <= mem[addr];
end

endmodule
```

```
read_verilog example.v
read_verilog -D
ICE40_HX
     -lib -specify
     +/ice40/cells_sim.v
hierarchy -check
     -top example
```



(`show` command output)

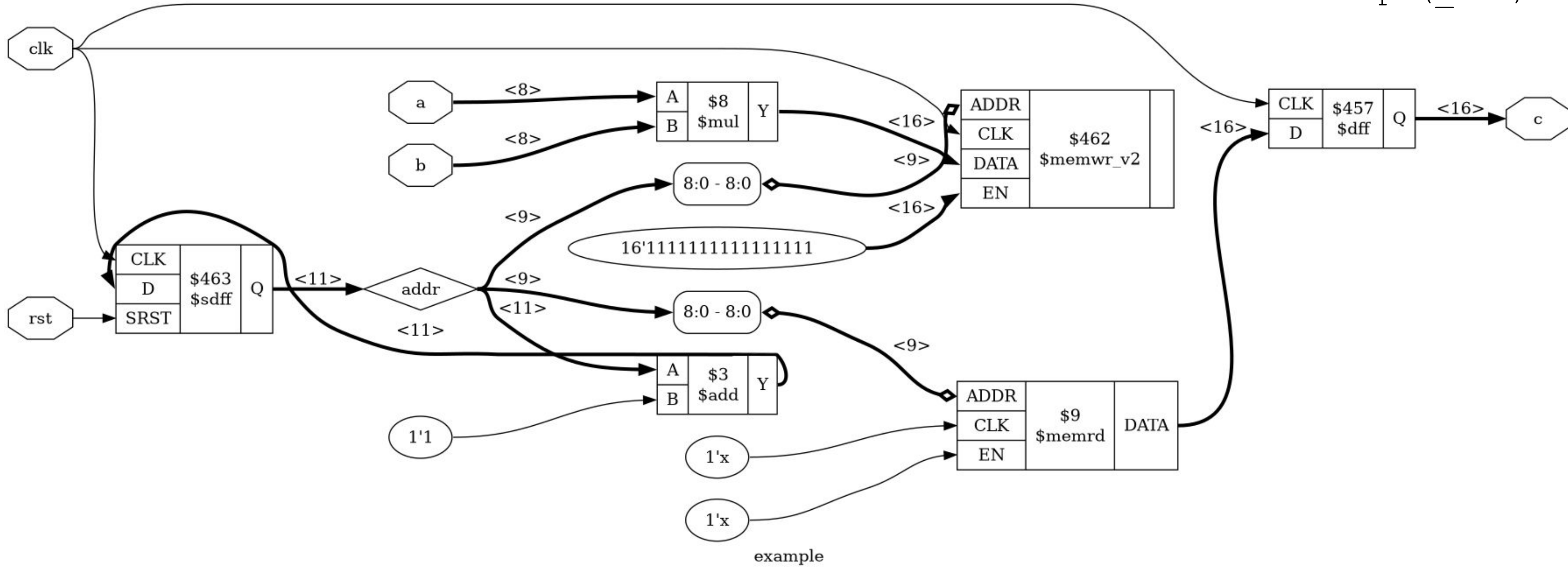


YosysHQ

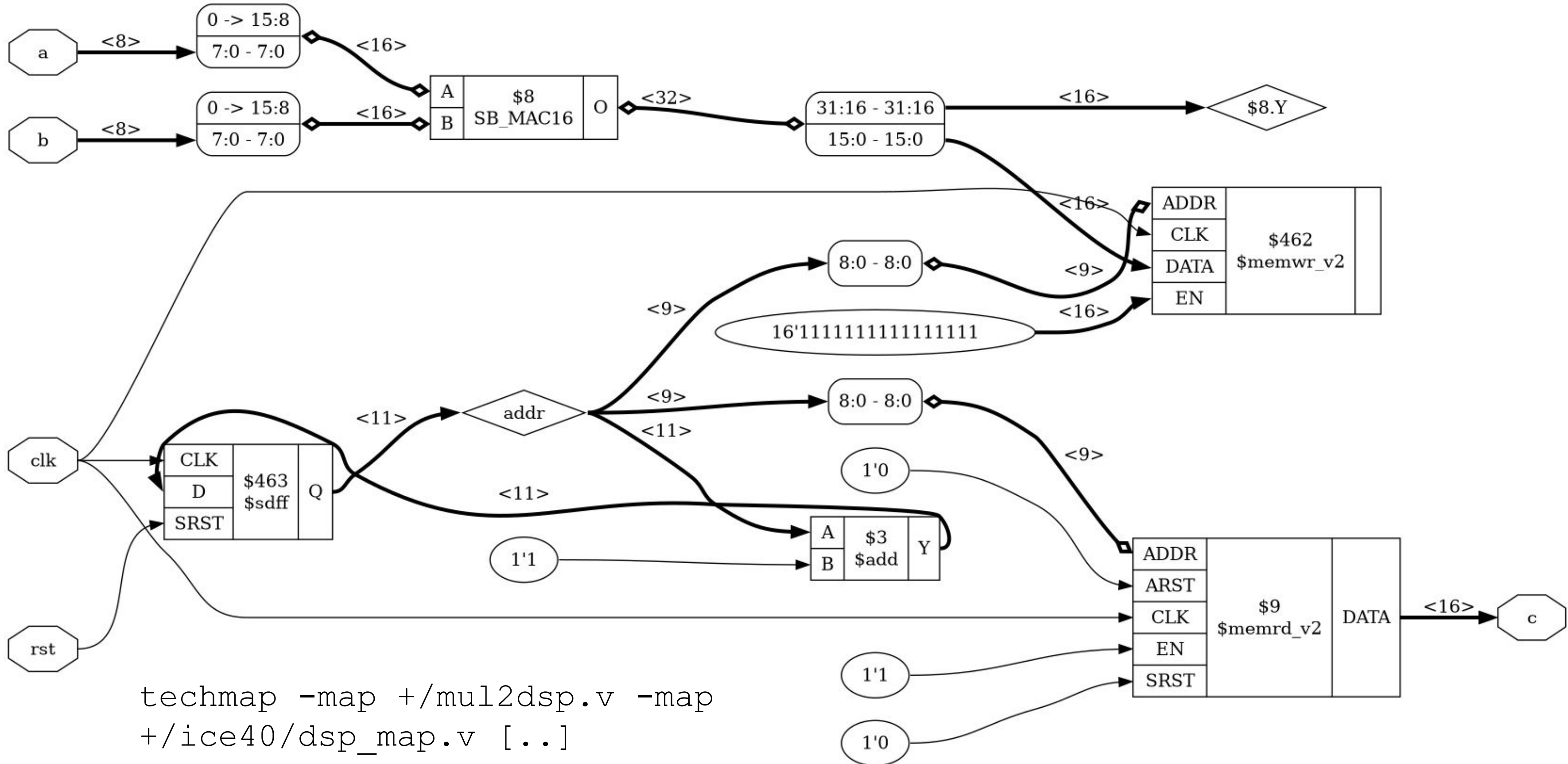
```
proc
opt_expr
opt_clean
check
```

YosysHQ

opt(_dff)

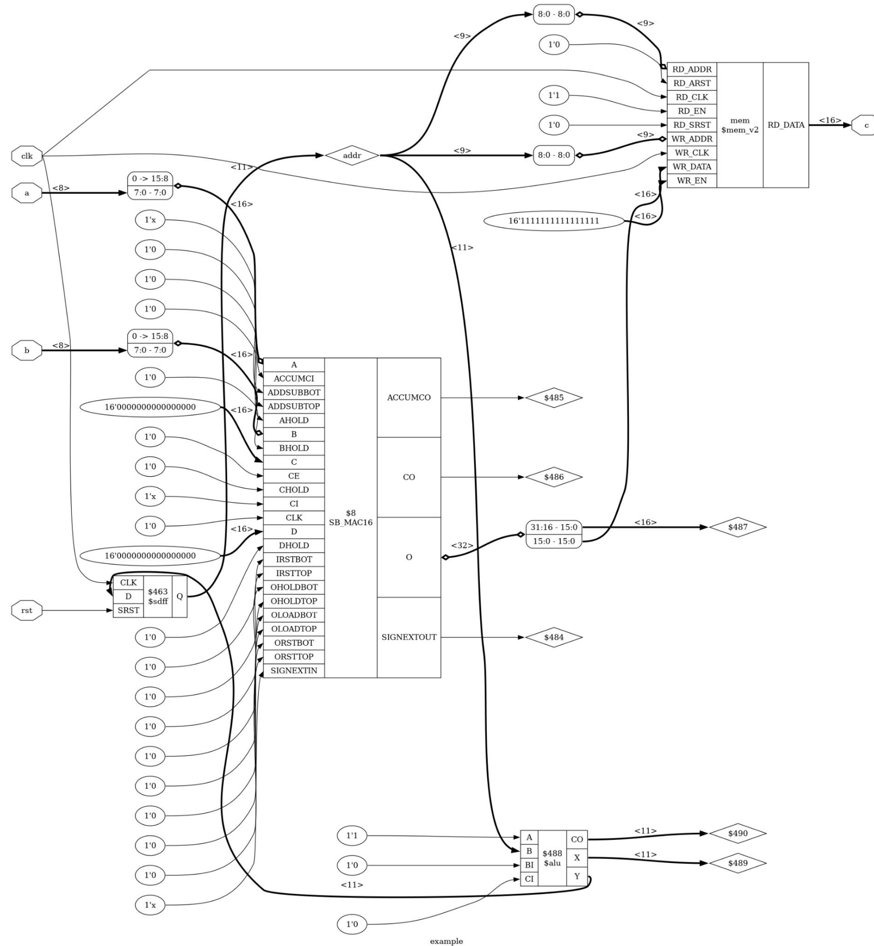


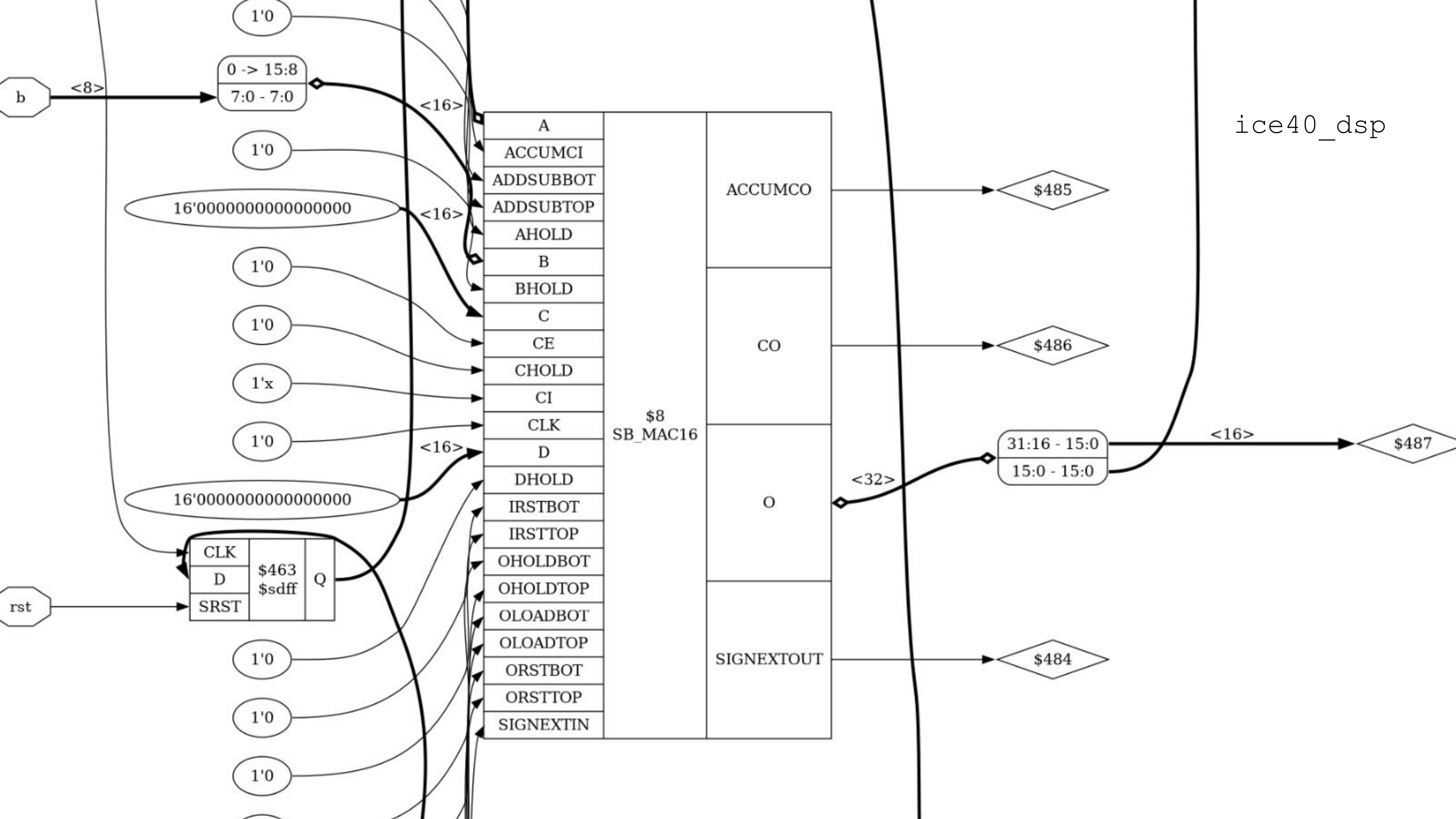
YosysHQ




```
techmap -map +/mul2dsp.v -map
+/ice40/dsp_map.v [..]
clean
```

ice40_dsp




ice40_dsp
b
<8>
0 -> 15:8
7:0 - 7:0
<16>
1'0
16'0000000000000000
1'x
rst
CLK
D
SRST
$463
$sdff
Q
A
ACCUMCI
ADDSUBBOT
ADDSUBTOP
AHOLD
B
BHOLD
C
CE
CHOLD
CI
CLK
D
DHOLD
IRSTBOT
IRSTTOP
OHOLDBOT
OHOLDTOP
OLOADBOT
OLOADTOP
ORSTBOT
ORSTTOP
SIGNEXTIN
$8
SB_MAC16
ACCUMCO
CO
O
SIGNEXTOUT
$485
$486
<32>
31:16 - 15:0
15:0 - 15:0
$487
$484

Memories

- There are two representations of memories: unpacked and packed
- Initially, individual array accesses are translated into `$memrd_v2`, `$memwr_v2`, `$meminit_v2`
- cells refer to name of the array signal
- `memory_collect` takes all cells with the same array signal and combines them into a single `$mem_v2` with a port for each `$memrd_v2` and `$memwr_v2` and a consolidated `init` attribute
- `memory_libmap` maps to a library of RAM blocks
- `memory_map` maps to logic

```
memory -nomap
opt_clean
memory_libmap -lib
    +/ice40/brams.txt
    -lib
    +/ice40/spram.txt
    -no-auto-huge
techmap -map
    +/ice40/brams_map.v
    -map
    +/ice40/spram_map.v
ice40_braminit
opt -fast -mux_undef
    -undriven -fine
memory_map
opt -undriven -fine
```

memory_libmap lib format

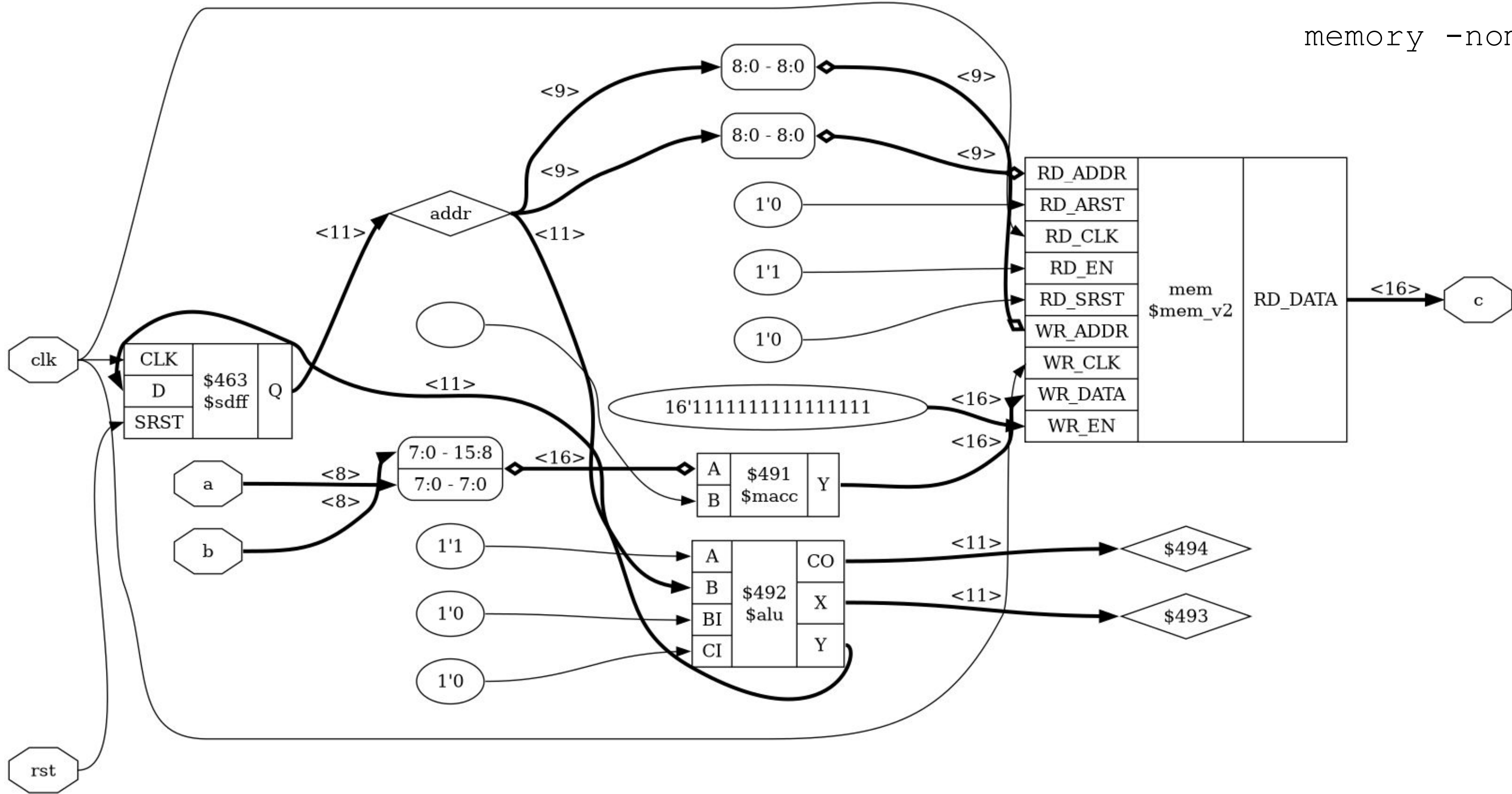
```
ram block $__ICE40_RAM4K_ {

    abits 11;
    widths 2 4 8 16 per_port;
    cost 64;

    option "HAS_BE" 1 {
     byte 1;
    }

    init any;

      port sw "W" {
       option "HAS_BE" 0 {
            width 2 4 8;
       }
       option "HAS_BE" 1 {
            width 16;
            wrbe_separate;
       }
       clock anyedge;
      }

      port sr "R" {
       clock anyedge;
       rden;
      }
}
```

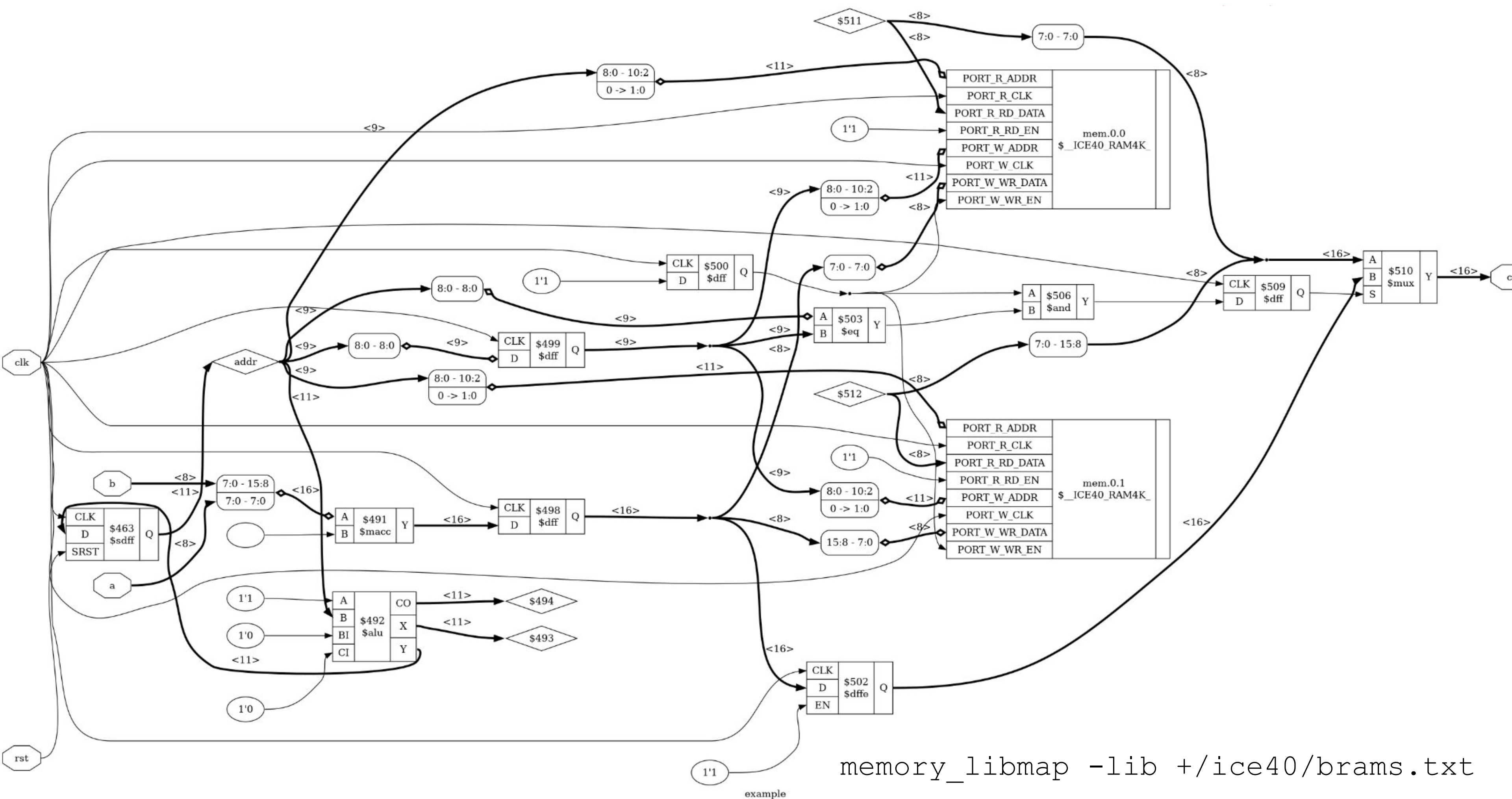
techmap rule brams_map.v

```
module $__ICE40_RAM4K_ (...);
SB_RAM40_4K #(
      .INIT_0(slice_init(0)),
     [...]
      .READ_MODE(mode(PORT_R_WIDTH)),
      .WRITE_MODE(mode(PORT_W_WIDTH))
    ) _TECHMAP_REPLACE_ (
      .RDATA(RDATA),
      .RCLK(PORT_R_CLK),
      .RCLKE(PORT_R_RD_EN),
      .RE(1'b1),
      .RADDR(RADDR),
      .WDATA(WDATA),
      .WCLK(PORT_W_CLK),
      .WCLKE(PORT_W_WR_EN),
      .WE(1'b1),
      .WADDR(WADDR),
      .MASK(MASK),
    );
endmodule
```

YosysHQ

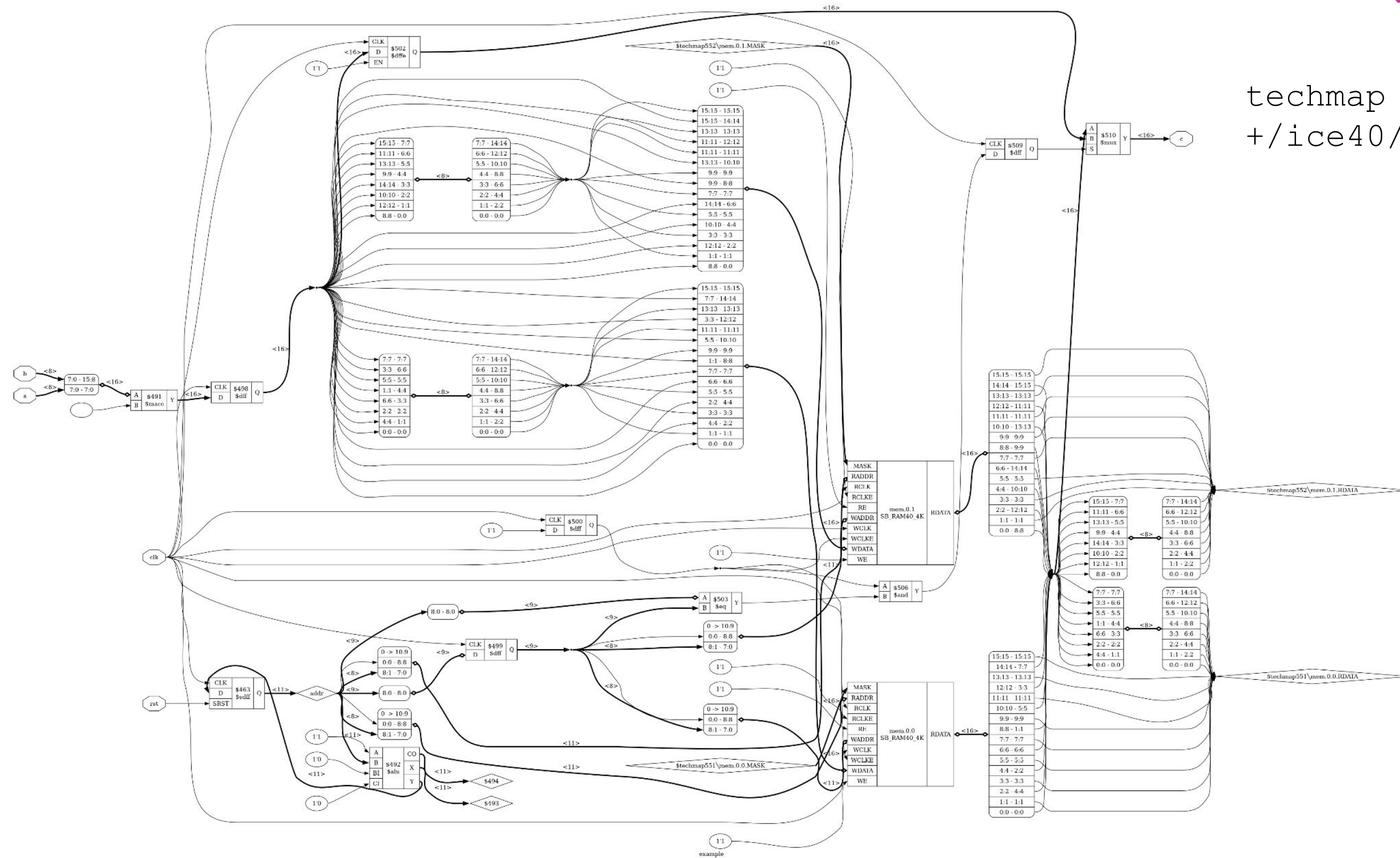
memory -nomap



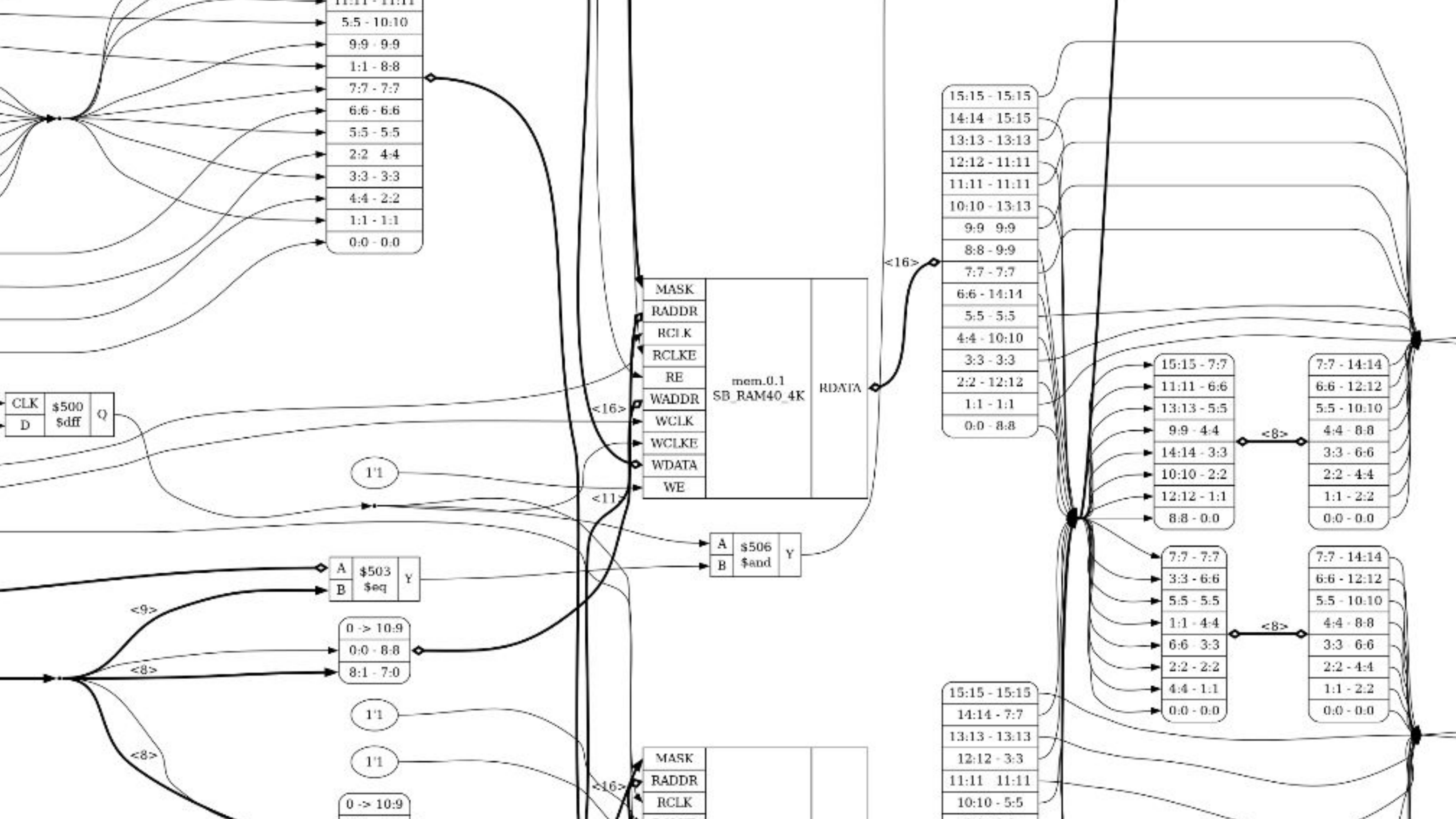


clk
rst
addr
a
b
$511
$512
7:0 - 7:0
8:0 - 10:2
0 -> 1:0
8:0 - 8:0
7:0 - 15:8
15:8 - 7:0
1'1
1'0
PORT_R_ADDR
PORT_R_CLK
PORT_R_RD_DATA
PORT_R_RD_EN
PORT_W_ADDR
PORT_W_CLK
PORT_W_WR_DATA
PORT_W_WR_EN
mem.0.0
$__ICE40_RAM4K_
mem.0.1
$__ICE40_RAM4K_
CLK
D
Q
SRST
$463
$sdff
$491
$macc
$492
$alu
A
B
BI
CI
CO
X
Y
$494
$493
$498
$dff
$499
$dff
$500
$dff
$502
$dffe
EN
$503
$eq
$506
$and
$509
$dff
$510
$mux
S
<8>
<9>
<11>
<16>
example
memory_libmap -lib +/ice40/brams.txt

YosysHQ




```
techmap -map
+/ice40/brams_map.v
```



5:5 - 10:10
9:9 - 9:9
1:1 - 8:8
7:7 - 7:7
6:6 - 6:6
5:5 - 5:5
2:2 - 4:4
3:3 - 3:3
4:4 - 2:2
1:1 - 1:1
0:0 - 0:0
CLK
D
$500
$dff
Q
1'1
A
B
$503
$eq
Y
<9>
0 -> 10:9
0:0 - 8:8
<8>
8:1 - 7:0
<8>
1'1
1'1
0 -> 10:9
MASK
RADDR
RCLK
RCLKE
RE
WADDR
WCLK
WCLKE
WDATA
WE
<16>
<11>
mem.0.1
SB_RAM40_4K
RDATA
<16>
A
B
$506
$and
Y
MASK
RADDR
RCLK
<16>
15:15 - 15:15
14:14 - 15:15
13:13 - 13:13
12:12 - 11:11
11:11 - 11:11
10:10 - 13:13
9:9 - 9:9
8:8 - 9:9
7:7 - 7:7
6:6 - 14:14
5:5 - 5:5
4:4 - 10:10
3:3 - 3:3
2:2 - 12:12
1:1 - 1:1
0:0 - 8:8
15:15 - 7:7
11:11 - 6:6
13:13 - 5:5
9:9 - 4:4
14:14 - 3:3
10:10 - 2:2
12:12 - 1:1
8:8 - 0:0
<8>
7:7 - 14:14
6:6 - 12:12
5:5 - 10:10
4:4 - 8:8
3:3 - 6:6
2:2 - 4:4
1:1 - 2:2
0:0 - 0:0
7:7 - 7:7
3:3 - 6:6
5:5 - 5:5
1:1 - 4:4
6:6 - 3:3
2:2 - 2:2
4:4 - 1:1
0:0 - 0:0
<8>
7:7 - 14:14
6:6 - 12:12
5:5 - 10:10
4:4 - 8:8
3:3 - 6:6
2:2 - 4:4
1:1 - 2:2
0:0 - 0:0
15:15 - 15:15
14:14 - 7:7
13:13 - 13:13
12:12 - 3:3
11:11 - 11:11
10:10 - 5:5

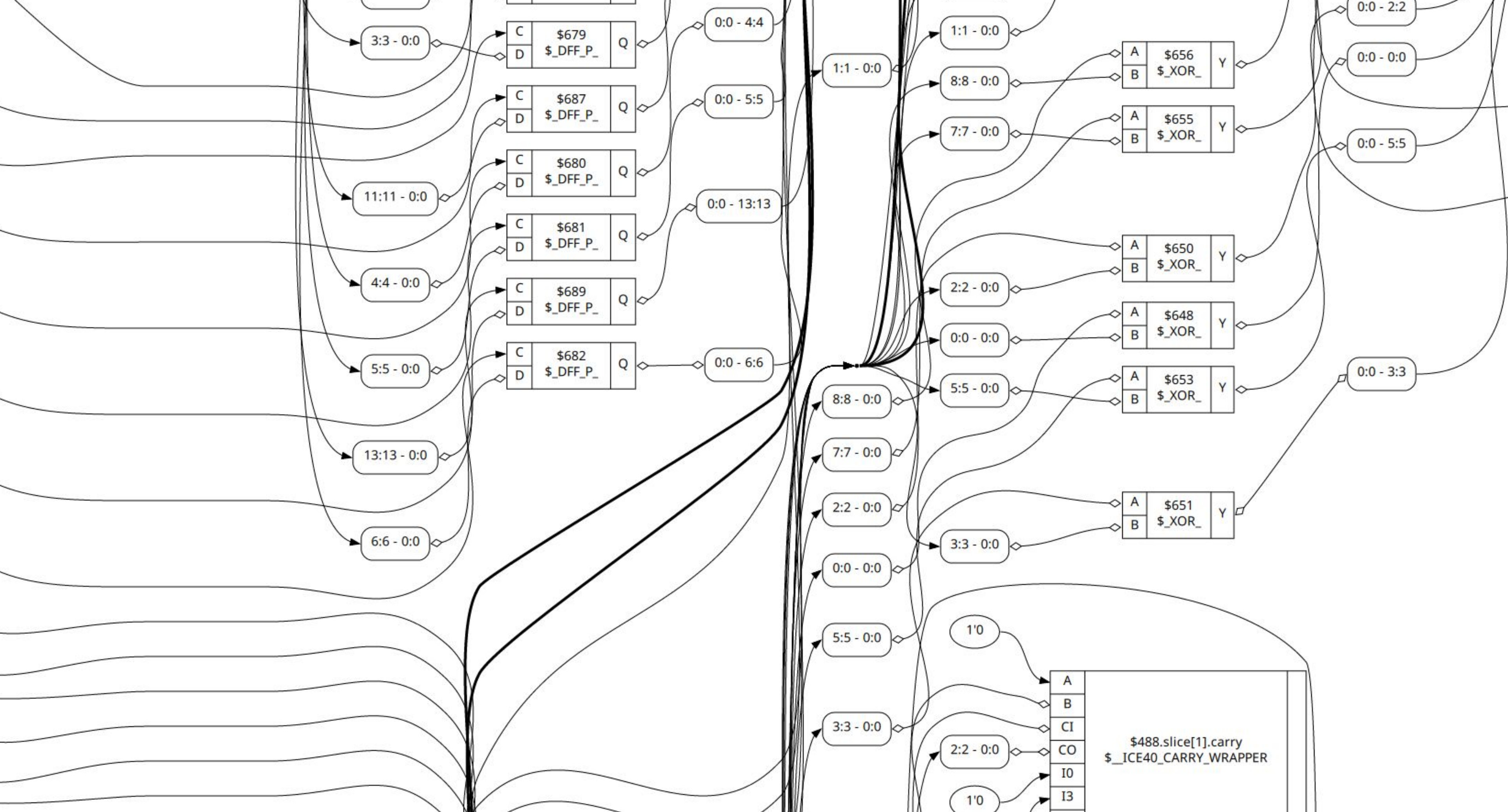
Fine-grain mapping

- `techmap/simplemap` replaces each coarse-grain cell with a collection of fine-grain cells (gates) like `$_AND_`, `$_OR_`, `$_NOT_`, `$_DFF_P_`
- `dfflegalize` transforms FF types to a restricted subset, emulating enable/reset/init where not available
- `abc` does fine-grain logic optimization and can map to LUTs or to sets of gates
  - this is actually a whole separate tool, and a lot of information is lost in the round trip
- finally internal yosys cells are mapped to target architecture cells (`techmap` again)

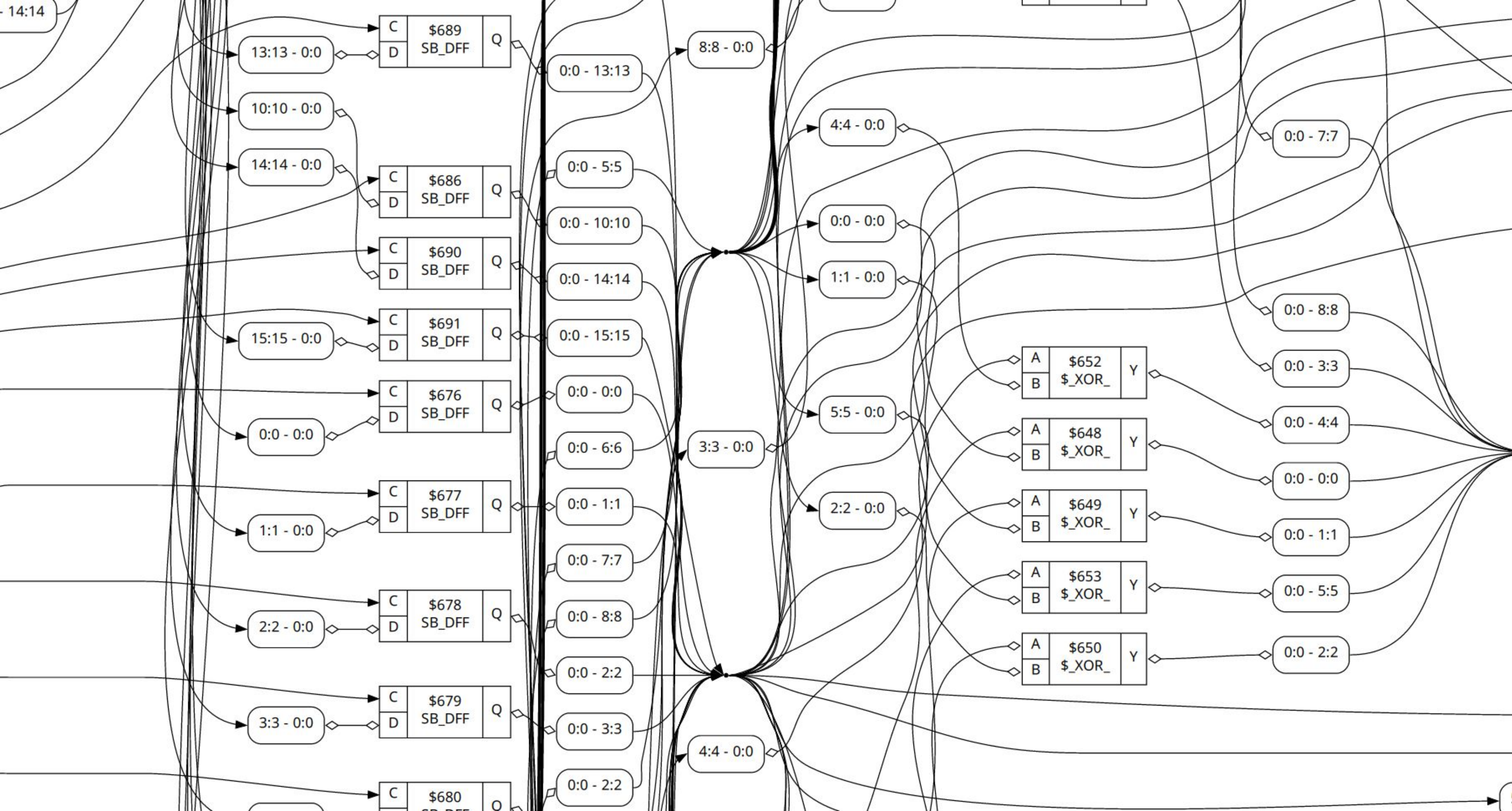
```
ice40_wrapcarry
techmap -map +/techmap.v
     -map +/ice40/arith_map.v
opt -fast
ice40_opt
dfflegalize -cell $_DFF_?_ 0
     -cell $_DFFE_?P_ 0 [..]
techmap -map +/ice40/ff_map.v
opt_expr -mux_undef
simplemap
ice40_opt -full
abc -dress -lut 4
ice40_wrapcarry -unwrap
techmap -map +/ice40/ff_map.v
clean
opt_lut -dlogic [..]
techmap -map
     +/ice40/cells_map.v
clean
```

ABC: A System for Sequential Synthesis and Verification

- by Alan Mishchenko, UC Berkeley https://people.eecs.berkeley.edu/~alanmi/abc/
- `abc` pass in yosys:
  - creates a temporary directory
  - for each module, cuts out any cell not a supported internal cell
    - this includes FF cells
  - turns the removed cell's I/O ports into module I/Os
  - outputs the result to blif
  - calls the yosys-abc executable to optimize and technology map the combinatorial circuit
  - reads the result back in
  - puts the removed cells back
- `abc -dff` includes FFs in abc module, but only supports 0-initialized FFs
  - this has a retime mode
- `abc9` is a more recent update of abc with whitebox support, for better use of hard resource blocks
  - but it doesn't have a retime mode



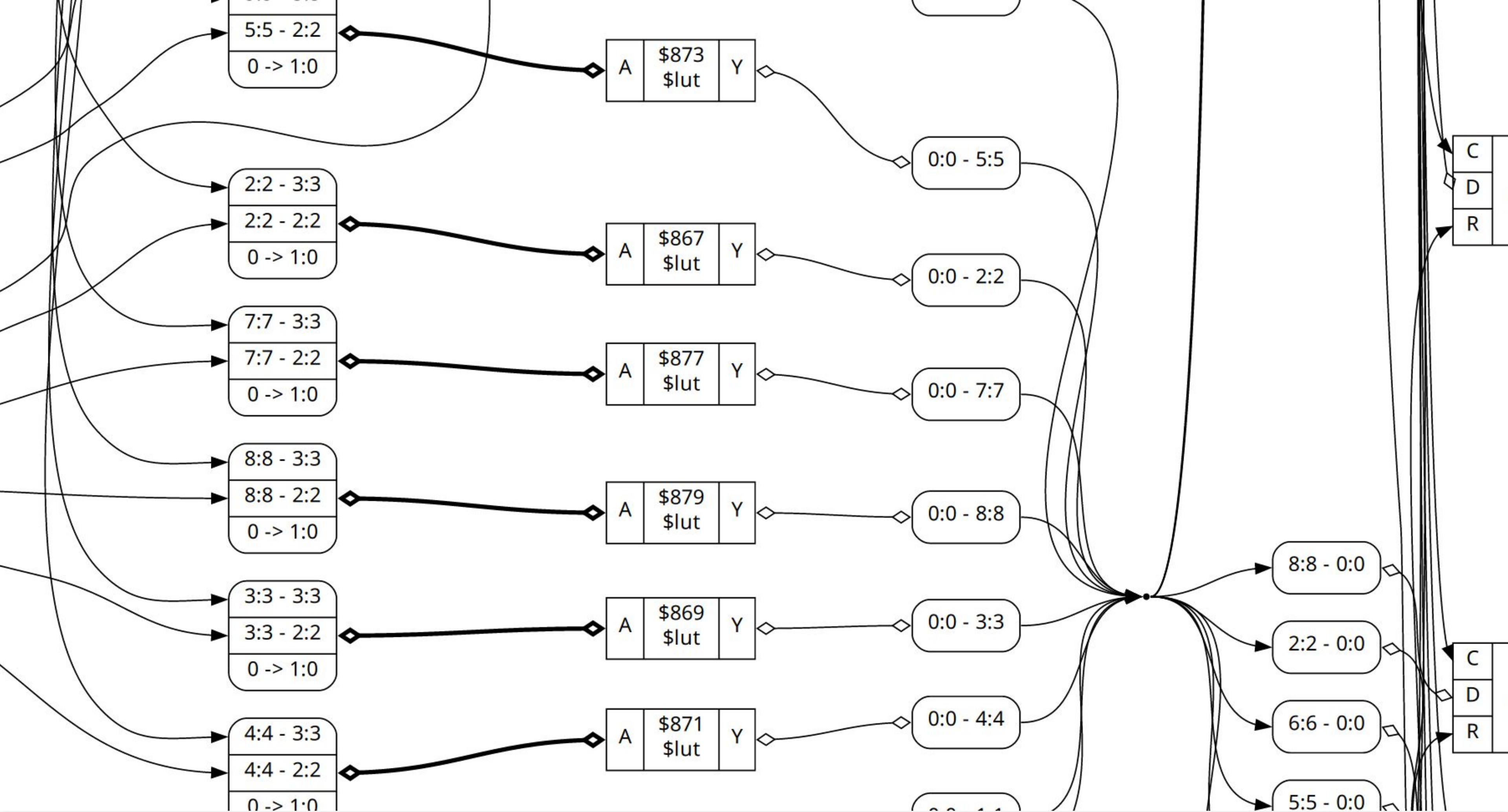
3:3 - 0:0
$679
$_DFF_P_
$687
$_DFF_P_
$680
$_DFF_P_
$681
$_DFF_P_
$689
$_DFF_P_
$682
$_DFF_P_
11:11 - 0:0
4:4 - 0:0
5:5 - 0:0
13:13 - 0:0
6:6 - 0:0
0:0 - 4:4
0:0 - 5:5
0:0 - 13:13
0:0 - 6:6
1:1 - 0:0
8:8 - 0:0
7:7 - 0:0
2:2 - 0:0
0:0 - 0:0
5:5 - 0:0
3:3 - 0:0
$656
$_XOR_
$655
$_XOR_
$650
$_XOR_
$648
$_XOR_
$653
$_XOR_
$651
$_XOR_
0:0 - 2:2
0:0 - 0:0
0:0 - 5:5
0:0 - 3:3
1'0
$488.slice[1].carry
$__ICE40_CARRY_WRAPPER
A
B
CI
CO
I0
I3



$689
SB_DFF
$686
SB_DFF
$690
SB_DFF
$691
SB_DFF
$676
SB_DFF
$677
SB_DFF
$678
SB_DFF
$679
SB_DFF
$680
$652
$_XOR_
$648
$_XOR_
$649
$_XOR_
$653
$_XOR_
$650
$_XOR_
13:13 - 0:0
10:10 - 0:0
14:14 - 0:0
15:15 - 0:0
0:0 - 0:0
1:1 - 0:0
2:2 - 0:0
3:3 - 0:0
0:0 - 13:13
0:0 - 5:5
0:0 - 10:10
0:0 - 14:14
0:0 - 15:15
0:0 - 6:6
0:0 - 1:1
0:0 - 7:7
0:0 - 8:8
0:0 - 2:2
0:0 - 3:3
8:8 - 0:0
3:3 - 0:0
4:4 - 0:0
5:5 - 0:0
0:0 - 4:4

stat

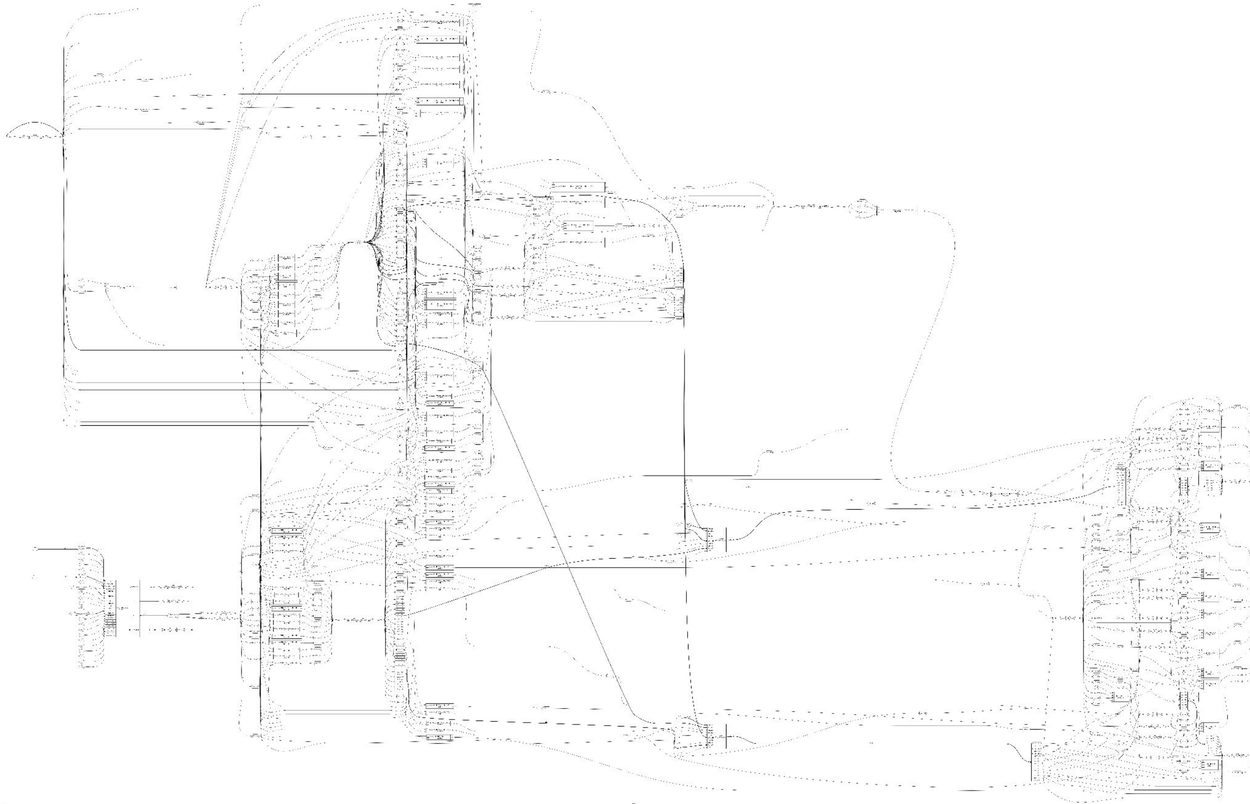
```
=== example ===

   Number of wires:                 27
   Number of wire bits:             221
   Number of public wires:          6
   Number of public wire bits:      45
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 101
     $_MUX_                         16
     $_NOT_                         2
     $_OR_                          8
     $_XOR_                         9
     $__ICE40_CARRY_WRAPPER         10
     SB_DFF                         42
     SB_DFFSR                       11
     SB_MAC16                       1
     SB_RAM40_4K                    2
```


5:5 - 2:2
0 -> 1:0
A
$873
$lut
Y
0:0 - 5:5
2:2 - 3:3
2:2 - 2:2
0 -> 1:0
A
$867
$lut
Y
0:0 - 2:2
7:7 - 3:3
7:7 - 2:2
0 -> 1:0
A
$877
$lut
Y
0:0 - 7:7
8:8 - 3:3
8:8 - 2:2
0 -> 1:0
A
$879
$lut
Y
0:0 - 8:8
3:3 - 3:3
3:3 - 2:2
0 -> 1:0
A
$869
$lut
Y
0:0 - 3:3
4:4 - 3:3
4:4 - 2:2
A
$871
$lut
Y
0:0 - 4:4
8:8 - 0:0
2:2 - 0:0
6:6 - 0:0
5:5 - 0:0
C
D
R
C
D
R

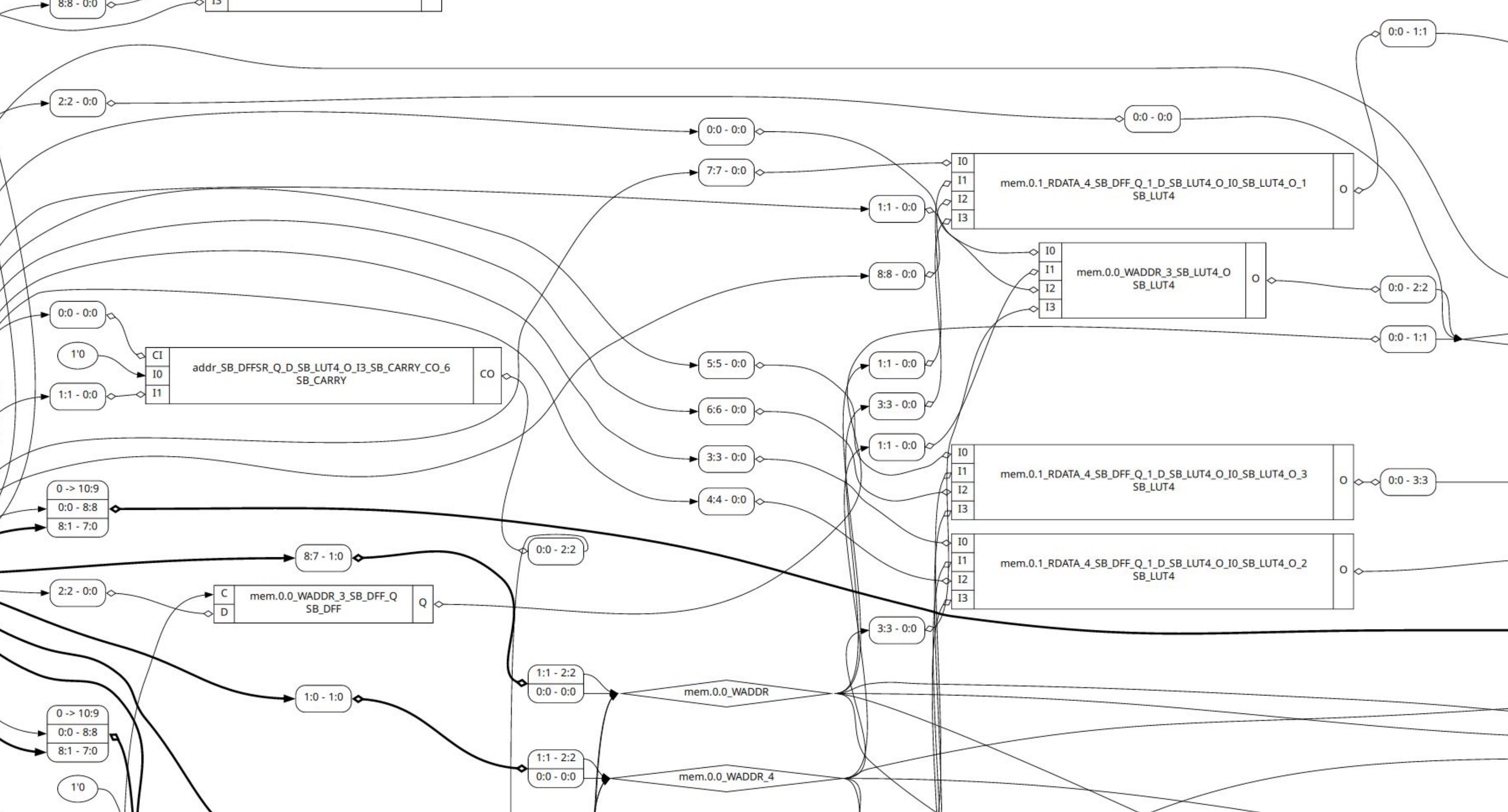
stat

```
=== example ===

   Number of wires:                 64
   Number of wire bits:             198
   Number of public wires:          6
   Number of public wire bits:      45
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 92
     $lut                           31
     SB_CARRY                       7
     SB_DFF                         42
     SB_DFFSR                       9
     SB_MAC16                       1
     SB_RAM40_4K                    2
```

Final netlist



```
techmap -map
+/ice40/cells_map.v
```



8:8 - 0:0
I3
0:0 - 1:1
2:2 - 0:0
0:0 - 0:0
0:0 - 0:0
7:7 - 0:0
I0
I1
I2
I3
mem.0.1_RDATA_4_SB_DFF_Q_1_D_SB_LUT4_O_I0_SB_LUT4_O_1
SB_LUT4
O
1:1 - 0:0
I0
I1
I2
I3
mem.0.0_WADDR_3_SB_LUT4_O
SB_LUT4
O
8:8 - 0:0
0:0 - 2:2
0:0 - 0:0
CI
I0
I1
addr_SB_DFFSR_Q_D_SB_LUT4_O_I3_SB_CARRY_CO_6
SB_CARRY
CO
1'0
1:1 - 0:0
5:5 - 0:0
1:1 - 0:0
0:0 - 1:1
6:6 - 0:0
3:3 - 0:0
3:3 - 0:0
1:1 - 0:0
I0
I1
I2
I3
mem.0.1_RDATA_4_SB_DFF_Q_1_D_SB_LUT4_O_I0_SB_LUT4_O_3
SB_LUT4
O
0:0 - 3:3
4:4 - 0:0
0 -> 10:9
0:0 - 8:8
8:1 - 7:0
I0
I1
I2
I3
mem.0.1_RDATA_4_SB_DFF_Q_1_D_SB_LUT4_O_I0_SB_LUT4_O_2
SB_LUT4
O
8:7 - 1:0
0:0 - 2:2
2:2 - 0:0
C
D
mem.0.0_WADDR_3_SB_DFF_Q
SB_DFF
Q
3:3 - 0:0
1:1 - 2:2
0:0 - 0:0
mem.0.0_WADDR
1:0 - 1:0
0 -> 10:9
0:0 - 8:8
8:1 - 7:0
1:1 - 2:2
0:0 - 0:0
mem.0.0_WADDR_4
1'0

stat

```
=== example ===

   Number of wires:                 39
   Number of wire bits:             222
   Number of public wires:          39
   Number of public wire bits:      222
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 92
     SB_CARRY                       7
     SB_DFF                         42
     SB_DFFSR                       9
     SB_LUT4                        31
     SB_MAC16                       1
     SB_RAM40_4K                    2
```

Backends

The end result is written out in a format consumable by another tool:

- structural verilog
- BLIF
- EDIF
- JSON
- FIRRTL
- …

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

write_blif -gates -attr
-param <file-name>

write_edif <file-name>

write_json <file-name>
```

Useful commands

Utility

- `show`, `dump`
- `history`: print previously used commands
- `exec`/"`!`": run a shell command
- `tee`: write log messages also to a file
- `help`: documentation
- `foo; bar`: sequence of commands, space is mandatory
- `foo;; bar` == `foo; clean; bar`
- `debug foo`: print debug messages while running foo
- `yosys -P ALL`: dump internal state to a file after each command
- `design -save`/`design -load`: keep a copy of the design state around

Select

- Nearly every pass in Yosys takes an optional selection argument and only operates on the active selection
  - e.g. `wreduce t:$mul` only trims unused bits from multiplication operations
- The select syntax is extremely powerful… but difficult to grasp
  - `help select` has the details
- Stack-based operation in reverse polish notation
- Items that can be selected are modules, cells and wires
- Select cells by type, wires by name, anything by attribute, etc.
- Combine with `show` or `dump` to look at specific portions of large designs
- `cd foo; ls; cd`: shortcuts for select -module, select -list and select -clear

Select example: Yosys one-liner

- AXI spec: “On Manager and Subordinate interfaces, there must be no combinatorial paths between input and output signals.”
- Common mistake when arbitrating between multiple AXI interfaces:
  ```
  if(axi_awvalid[0]) grant = 0; else grant = 1;
  axi_awready[grant] = 1'b1;
  ```
- One-liner to find such a mistake: flatten the design, then
  ```
  select -assert-none i:*axi_* %coe* o:*axi_* %cie* %i
  ```
  - `i:*axi_*`: input signals with name containing `axi_`
  - `%coe*`: extends selection to combinatorial output cone of last item on selection stack
  - `o:*axi_*`: new selection on stack, output signals with name containing axi_
  - `%cie*`: extends selection to combinatorial input cone
  - `%i`: takes two selections from the stack and creates their intersection

Techmap

- Techmap is the simplest of the custom cell transformation options in Yosys
- It replaces each cell of a given type with the specified logic
- Techmap rules are written in Verilog

```
techmap -map +/cmp2lut.v -D LUT_WIDTH=4
```

- At its most basic, a techmap rule just replaces a cell with the contents of the verilog module
  - flatten is actually the same code as techmap, just using the current design as techmap library
- Some special names are used for additional functionality

YosysHQ

The techmap pass

- Inputs to techmap: list of Verilog map files describing replacement modules
  - Ex:

```
(* techmap_celltype = "$eq $ne $lt $le $gt $ge" *)
module _90_lut_cmp_ (A, B, Y);
```

- techmap iterates through all the cells in the design
- If the cell matches one of the `techmap_celltype` arguments, the cell is replaced with the cell described in the `module` below
- Matches are tried in alphanumerical sort order
- Multiple matches possible, module implementation can reject match
- Recursive matching - replacement module can contain other cells which should be replaced.

`techmap` attributes

Modules in the map files can have special attributes.

- `techmap_celltype` : as seen on last slide, the type of cells matched. If this is absent, cells are matched by name. (i.e. normal instantiation)
- `techmap_simplemap` : call `simplemap` pass instead to handle this cell
- `techmap_maccmap` : call `maccmap` pass instead to handle this cell
- `techmap_wrap` : call the specified pass instead to handle this cell

In addition, individual ports can have a special attribute:

- `techmap_autopurge` : mark the port as optional. If it is not connected it is not added to the replacement cell.

Communicating with techmap

From module to techmap:

Special Wires
assigned in module

- _TECHMAP_FAIL_
- _TECHMAP_DO_*
- _TECHMAP_REMOVEINIT_<portname>_

From techmap to module:

Special Parameters
read in module

- _TECHMAP_CELLTYPE_
- _TECHMAP_CONSTMSK_<port-name>_
  _TECHMAP_CONSTVAL_<port-name>_
- _TECHMAP_WIREINIT_<port-name>_
- _TECHMAP_BITS_CONNMAP_
  _TECHMAP_CONNMAP_<port-name>_

Special wires

These are often created in generate statements in certain if/else branches.

- _TECHMAP_FAIL_ : reject the match if assigned non-zero
  Ex: If this implementation should only be used on large inputs

```
  generate
  if (0) begin end
`ifdef DSP_A_MINWIDTH
  else if (A_WIDTH < `DSP_A_MINWIDTH)
    wire _TECHMAP_FAIL_ = 1;
`endif
`ifdef DSP_B_MINWIDTH
  else if (B_WIDTH < `DSP_B_MINWIDTH)
    wire _TECHMAP_FAIL_ = 1;
`endif
```

Special wires

- _TECHMAP_DO_* : execute command(s)
  Ex: Techmap is executed after processes have been converted. If the replacement module contains generate or always blocks, they need to be converted too.

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

- _TECHMAP_REMOVEINIT_<port-name>_ : consume init value (bitmask)
  Companion to _TECHMAP_WIREINIT_<port-name>_

Special parameters

Used to change implementations based on circumstances.

- _TECHMAP_CELLTYPE_ : type name of the matched cell
  If the techmap_celltype attribute lists multiple cells that can be matched, this parameter can be used to tell which one was matched.

```
generate
    if (_TECHMAP_CELLTYPE_ == "$lt")
        localparam operation = 0;
    if (_TECHMAP_CELLTYPE_ == "$le")
        localparam operation = 1;
    if (_TECHMAP_CELLTYPE_ == "$gt")
        localparam operation = 2;
    if (_TECHMAP_CELLTYPE_ == "$ge")
        localparam operation = 3;
```

Special parameters

- _TECHMAP_CONSTMSK_<port-name>_ / _TECHMAP_CONSTVAL_<port-name>_ : specifies which inputs are constant
  Ex: replace 2-input cell with a 1-input cell with internal constant

```
else if (&_TECHMAP_CONSTMSK_B_)
    \$lut #(...) _TECHMAP_REPLACE_ (
        .A(A),
        .Y(Y)
    );
else if (&_TECHMAP_CONSTMSK_A_)
    \$lut #(...) _TECHMAP_REPLACE_ (
        .A(B),
        .Y(Y)
    );
```

Special parameters

- _TECHMAP_WIREINIT_<port-name>_ : specifies init value of wire connected to the port
  Sometimes the init value can affect which implementations are possible.
  Ex: Xilinx FF set/reset can only revert to initial value

```
module  \$_DFF_NN0_ (input D, C, R, output Q);
  parameter [0:0] _TECHMAP_WIREINIT_Q_ = 1'bx;
  generate if (_TECHMAP_WIREINIT_Q_ === 1'b1)
      $error("Spartan 6 doesn't support FFs with asynchronous reset initialized to 1");
  else
      FDCE_1 #(.INIT(_TECHMAP_WIREINIT_Q_)) _TECHMAP_REPLACE_ (.D(D), .Q(Q), .C(C), .CE(1'b1), .CLR(!R));
  endgenerate
endmodule
```

Special parameters

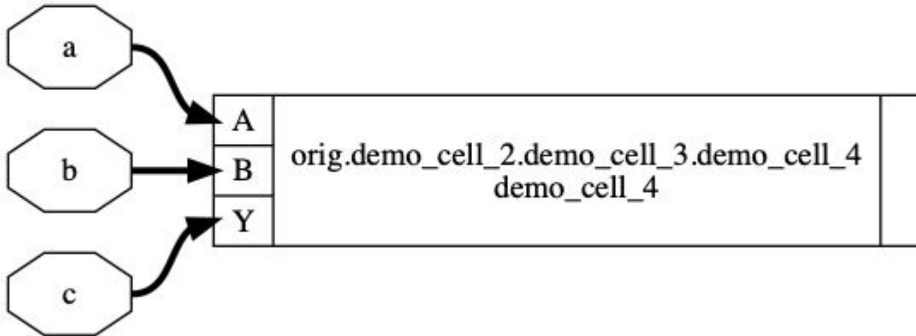
- _TECHMAP_BITS_CONNMAP_ / _TECHMAP_CONNMAP_<port-name>_ : identify input drivers
  Ex: check that read and write ports of a simple dual-port memory use the same clock

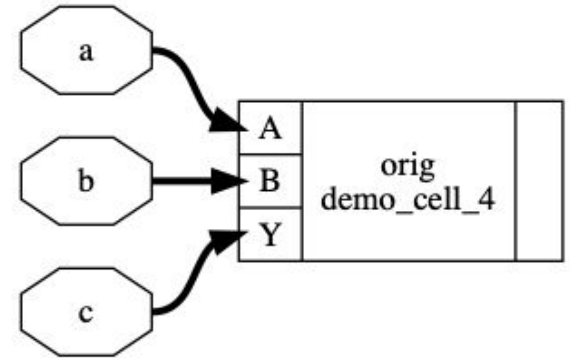
```
// read and write must be in same clock domain
if (_TECHMAP_CONNMAP_RD_CLK_ != _TECHMAP_CONNMAP_WR_CLK_)
    _TECHMAP_FAIL_ <= 1;
```

Special entity names

_TECHMAP_REPLACE_, _TECHMAP_REPLACE_.*

- modules with this name inherit the replaced cell's name
- avoids excessively long names in cases of several recursive techmap calls





3 recursive techmap calls with: named modules

_TECHMAP_REPLACE_ modules

Replacing a group of cells with “extract”

- The “extract” command finds subcircuits in the design that are isomorphic to a given reference circuit,
  - and replaces that subcircuit with an instance of the reference circuit.
  - That single cell instance can then be replaced by another circuit using techmap.
- For many real-world applications however it makes sense to write a custom pattern matcher instead of relying on “extract”.
  - Many patterns are more complex than “here’s a reference circuit”
    - have optional parts or alternative parts
    - may contain variable-width signals
  - Performance of “extract” isn’t very good for finding small circuits in large ones.
    - especially if those small circuits are heavily labeled
    - but hand-crafted matchers usually work very well for those situations

=> pmgen for generating efficient pattern matchers

Tracking correspondence to source code

- Only wires and instances are given names in source
  - these persist as public names (starting with `\`)
  - the tools do their best to preserve these
- Everything else is inferred
  - only has private names (starting with `$`)
  - private names mostly relate to the pass where the element was created
- `(* src *)` attributes refer to the original source line where available
- Consider which port may have a wire with a relevant name attached
- Sometimes it may not be possible to trace back exactly
- `autoname` tries to give things more relevant names

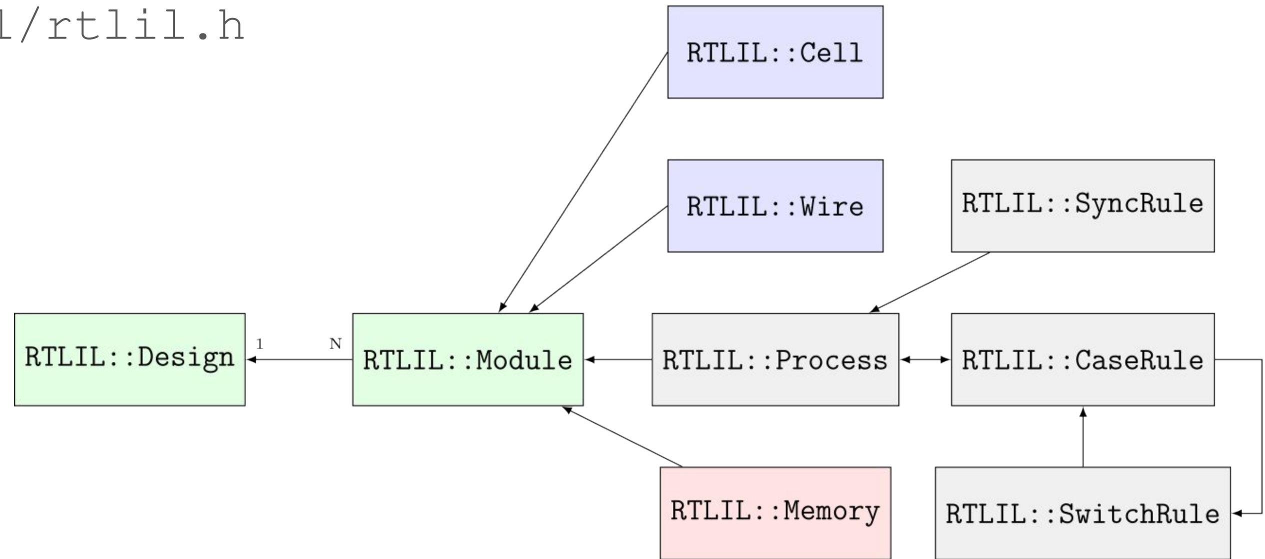
Breaktime

Core RTLIL Data Structures: the developer view

Class diagram

Design representation

read along in `kernel/rtlil.h`



RTLIL::Design

- One of these is the "current design"
- Holds all the user code
- When a pass is run, it gets the pointer to the current design
    - traverse the design from there
- Yosys can hold multiple designs in memory, they can be manipulated with the `design` command, but other passes are not aware of any but the current design (and those they create themselves)

```
for (auto mod : design->selected_modules())
```

YosysHQ

RTLIL::Module

- One module per entity declaration
- Contains many Cell, Wire, Memory objects
- Has attributes
- For parametrized modules ("generic" in VHDL), `hierarchy` will create a separate derived module for each parameter set

```
for (Cell* cell : module->selected_cells())
for (Wire* wire : module->selected_wires())
for (Mem* &mem : Mem::get_selected_memories(module));
for (pair<IdString,SigSpec> &conn : cell->connections())
```

RTLIL::Process

- Internal representation of $proc
- There is not much you can do with this beyond run proc to transform them into cells
- Most passes should refuse to work on modules with processes:

```
for (auto mod : design->selected_modules()){
    if (module->has_processes_warn())
        continue;
}
```

RTLIL::Cell

- Represents instances (of modules or primitives)
- Has a name and a type
- Has attributes
- Has ports: `dict<RTLIL::IdString, RTLIL::SigSpec>`

```
if(cell->type.in(ID($add), ID($sub)))
    sig_a = cell->getPort(ID::A)
```

YosysHQ

RTLIL::Wire

- Wires/buses in the design
- Has a width, offset, direction (upto/downto), and attributes
- Can be a module in/out/inout port

```
if(wire->width == 1)
if(wire->port_input && wire->port_output)
```

RTLIL::Memory

- This is merely an empty shell to point to, it contains only the name of the array signal
- The actual magic happens in the `Mem` helper object from `kernel/mem.h`
- Constructs all the memory handling structures when `Mem::get_selected_memories(module)` is called

FfData helper

- Similar to memories, handling other stateful elements (flops and latches) is also complex
- initialization, enables, clock sensitivities, sync or async reset or load, …
- Construct the `FfData` helper to query/manipulate FFs

```
SigMap sigmap(mod);
FfInitVals initvals(&sigmap, mod);
for (auto cell : mod->selected_cells())
{
    if (!RTLIL::builtin_ff_cell_types().count(cell->type))
        continue;
    FfData ff(&initvals, cell);
    ...
}
```

RTLIL::IdString

- kind of a string, but also kind of an int, does refcounting magic
- all of the classes we've seen so far have `name` members of this type
- most have accessors for getting member elements by name
- public IDs (from user code) start with `\`, private IDs start with `$`
- instead of `IdString("\\foo")`, you can also use `ID(foo)` and `ID::foo` (`IdString("$bar")` is `ID($bar)` or `ID::$bar`)
- passes use `NEW_ID` macro when creating new design elements - that's where all the `$foo$123` names come from

Container types

`dict<K, T>`

- A hash table, faster than `std::unordered_map<>`
- Deterministic iterator order on all platforms
- Member objects may move on insert/remove of other members (similar to `std::vector<>`)

`pool<T>`

- A replacement for `std::unordered_set<>`, similar to `dict<>`

YosysHQ

RTLIL::State and RTLIL::Const

`RTLIL::State`

- Represents a single constant bit
- Enum with following values:
  - State::S0 and State::S1
  - State::Sx -- undefined value
  - State::Sz -- high impedance
  - State::Sa -- special value for don’t cares in case statements
  - State::Sm -- special “marked” value for internal use in some passes

`RTLIL::Const`

- Essentially a vector of `RTLIL::State`

RTLIL::SigBit and RTLIL::SigSpec

`RTLIL::SigBit`

- A single bit. Either a constant (0, 1, X, Z) or a single bit from a wire

`RTLIL::SigSpec`

- Essentially a vector of `SigBit`s
- Can be a wire, wire selection, constant, concatenations
- Cell ports connect to `SigSpec`s
- e.g. `cell->getPort(ID::Q)` gets what's connected on port named `Q` - can be a mix of constants and various wire bits from other places in the design

RTLIL::SigSig and SigMap

- Represents connections between signals
- `typedef std::pair<SigSpec, SigSpec> SigSig;`
- RTLIL Wires (or wire bits) can be connected directly to each other, without the help of a “buffer” cell type.
- That means the same signal can have multiple names, which is a problem e.g. when comparing signals
- To create canonical names, create the helper object
  `SigMap sigmap(module);`

```
for(auto it: module->connections())
if(sigmap(sig1) == sigmap(sig2))
```

Indexing

- RTLIL generally does not contain indices.
- Only lookup-by-name is fast for wires and cells.
  - cellport → wire is fast because it only requires by-name lookup
  - But wire → cellports is slow because that would need different indices
- Most passes work in two phases:
  - Create the custom indices needed for the pass and
  - Do the actual work using these custom indices
- `ModWalker` is a generic index class used for read-only indices.
- `ModIndex` is a generic index class that automatically keeps indices up to date.
- these are in `kernel/sigtools.h` and `kernel/modtools.h`

Modifying the design

- The `RTLIL::Module` class has helper functions for many operations
  - `module->addWire(NEW_ID)`
  - `module->Mux()/module->addMux()` for all kinds of internal cells
  - `module->connect()` for directly connecting two wires
  - If you modify the port wires, call `module->fixup_ports()` after to reindex
  - Don't delete wires, just disconnect them and call clean after the pass
- `RTLIL::Cell`
  - `cell->getPort(ID)`
  - `cell->setPort(ID, sig)`
  - `cell->unsetPort(ID)`
  - `cell->type`
  - `cell->fixupParameters()` after you change parameters

Querying the internal cell library

- `celltypes.h` has cell information
- `yosys_celltypes` and `RTLIL::builtin_ff_cell_types()` static variable/function
- To see if a cell is an internal yosys cell: `yosys_celltypes.cell_known(c->type)`
- To see if a cell is an internal yosys FF cell: `RTLIL::builtin_ff_cell_types().count(c->type)`

Writing Custom Passes

Custom Yosys Passes

- Base-classes for passes, frontends, and backends
- Custom passes derived from Pass and ScriptPass base-class
- Design patterns commonly used in Yosys passes
- Creating and loading Yosys plugins

Base-classes for passes, frontends, backends

`Pass`

- Generic class, contains pass calling machinery
- The other three classes inherit from `Pass`

`ScriptPass`

- For a pass that's like a "script" calling other passes
- Typical examples are the `synth_*` passes (`synth_ice40`, `synth_xilinx`, etc.)

`Frontend`

- For passes that take as input a file/stream, not a design

`Backend`

- For passes that output to a file/stream

Custom passes derived from Pass base-class

Most passes are derived from `Pass`.

Each pass class has one static instance. The `Pass` constructor registers the pass with the Yosys core systems.

Use `extra_args()` to parse additional arguments for selection.

Iterate on selection with

- `design->selected_modules()`
- `module->selected_cells()`
- `module->selected_wires()`

```
#include "kernel/yosys.h"

USING_YOSYS_NAMESPACE
PRIVATE_NAMESPACE_BEGIN

struct FoobarPass : public Pass {
    FoobarPass() : Pass("foobar", "short description of foobar command") { }
    void help() YS_OVERRIDE
    {
        log("\n");
        log("    foobar [options] [selection]\n");
        log("\n");
        log("Long description of foobar command.\n");
        log("\n");
    }
    void execute(std::vector<std::string> args, RTLIL::Design *design) YS_OVERRIDE
    {
        bool m_mode = false;

        size_t argidx;
        for (argidx = 1; argidx < args.size(); argidx++) {
            if (args[argidx] == "-m") {
                m_mode = true;
                continue;
            }
            break;
        }
        extra_args(args, argidx, design);

        log_header(design, "Executing FOOBAR pass.\n");

        for (auto module : design->selected_modules())
        {
            log("Processing module %s.\n", log_id(module));
            ...
        }
    }
} FoobarPass;

PRIVATE_NAMESPACE_END
```

Worker classes

- Convention: use worker class for complex passes, one instance per module
  - member functions can have contextual module data without having to pass down too many arguments
  - example: `wreduce`
- Or worker function for less complex cases
  - example: `autoname`
- Most workers use a two-phase structure:
  - Generate SigMap for the module and other index structures
  - Do the actual work using the index structures just created

Custom passes derived from ScriptPass base-class

```
#include "kernel/yosys.h"

USING_YOSYS_NAMESPACE
PRIVATE_NAMESPACE_BEGIN

struct FoobarPass:public ScriptPass
{
    FoobarPass() : ScriptPass("foobar", "short description of foobar command") { }

    void help() YS_OVERRIDE
    {
        log("\n");
        log("    foobar [options]\n");
        log("\n");
        log("Long description of foobar command.\n");
        log("\n");
        help_script();
        log("\n");
    }

    bool m_mode;

    void clear_flags() YS_OVERRIDE
    {
        m_mode = false;
    }

    void script() YS_OVERRIDE
    {
        if (check_label("A")) {
            run("cmd1");
            run("cmd2");
            if (help_mode || m_mode)
                run("cmd3", "(with -m)");
        }

        if (check_label("B")) {
            run("cmd4");
        }
    }

    void execute(std::vector < std::string > args, RTLIL::Design * design) YS_OVERRIDE
    {
        string run_from, run_to;
        clear_flags();

        size_t argidx;
        for (argidx = 1; argidx < args.size(); argidx++) {
            if (args[argidx] == "-run" && argidx + 1 < args.size()) {
                size_t pos = args[argidx + 1].find(':');
                if (pos == std::string::npos)
                    break;
                run_from = args[++argidx].substr(0, pos);
                run_to = args[argidx].substr(pos + 1);
                continue;
            }
            if (args[argidx] == "-assert") {
                assert = true;
                continue;
            }
            break;
        }

        if (argidx < args.size())
            cmd_error(args, argidx, "Unknown option.");

        log_header(design, "Executing EQUIV_OPT pass.\n");
        log_push();

        run_script(design, run_from, run_to);

        log_pop();
    }
} FoobarPass;

PRIVATE_NAMESPACE_END
```

Logging (`kernel/log.h`)

- `log("...")` for messages
- `log_header("Executing … pass")` when starting pass
- `log_warning()` or `log_error()` for problems (error exits)
- `log_cmd_error()` for syntax errors if design was not touched (exits in script but not in yosys shell)
- all of the above take C printf format
- `log_push()`/`log_pop()` for script passes
- `log_module()`, `log_cell()`, `log_wire()` to print names of elements
- `log_signal()`, `log_const()`, `log_id()` actually don't log but return a C string (`char*`) for use with `%s` in the above log functions
- `stringf` for C printf format but with a `std::string` return value

YosysHQ

General tips

- Look in existing passes for examples doing similar things
- But the older passes use coding styles no longer recommended
  - e.g., accessing data members directly - use provided functions instead
    - `module->selected_cells()` or `module->cells()`, not `module->cells_`
- The most difficult part is figuring out the prerequisites/dependencies between passes
  - script passes in yosys can be guides
  - if all the `synth_*` passes call things in a certain order, there's probably a reason for it
  - ask!

Build environment

- OSS CAD Suite is bundled with its libraries and searches for them in `oss-cad-suite/lib/`
- To be linkable, plugins need to be compiled with the same library versions and the same compiler and flags as the yosys executable
- OSS CAD Suite is is built in a docker container running Ubuntu 20.04
  - a compatible docker file is available in https://github.com/YosysHQ/oss-cad-suite-build
  - but a local Ubuntu install should also work, such as in the provided VM
- source the file `oss-cad-suite/environment` to set up the paths
- the `yosys-config` utility handles paths and compiler flags

Creating and loading Yosys plugins

Use “yosys-config” to query informations on compiler flags and Yosys install directories.

One can also use yosys-config directly for building a plugin:

```
    yosys-config --exec --cxx --cxxflags --ldflags -o plugin.so -shared
plugin.cc --ldlibs
```

Or short:

```
    yosys-config --build plugin.so plugin.cc
```

Loading the plugin:

```
    yosys -m plugin.so …
```

or

```
    yosys> plugin -i plugin.so
```

YosysHQ

Part 2/2: Pattern Matchers

- What is the Pattern Matcher Generator (pmgen)
  - Spoiler: A code generator for finding patterns in circuits
- The .pmg file format
- API of generated pattern matchers
- Introduction to various pmgen features
- Examples

Pattern Matcher Generator (pmgen)

A common theme in technology mapping and/or netlist optimization is the requirement to detect certain groups of cells in a netlist. This problem is called *subgraph isomorphism problem* and it is NP-complete.

However, efficient algorithms exist for many subgraph isomorphisms relevant for technology mapping and/or netlist optimization. This is in part because we usually look at relatively compact subgraphs in highly labeled graphs.

The *Pattern Matcher Generator* (pmgen) is a code generator that helps with writing pattern matchers based on a backtracking algorithm.

See `yosys/passes/pmgen/` documentation and examples.

pmgen overview

The input to pmgen is a `.pmg` file.

The output is a header-only C++ library implementing the pattern matcher.

The pattern matcher itself is a single C++ class.

For example:

`foobar.pmg` ⇒ pmgen ⇒ `foobar_pm.h` (containing `foobar` class)

Pmgen documentation:
https://github.com/YosysHQ/yosys/blob/master/passes/pmgen/README.md

The .pmg file format

A .pmg file contains one or more patterns.

Each pattern contains

- “state” variables
- user-data (“udata”) variables
- “match..endmatch” blocks
- “code..endcode” blocks
- subpattern

The blocks of a pattern are executed in order, with depth-first exploration of matches.

Example:

```
pattern main

state <SigSpec> cin

match carry
	select carry->type.in(\CARRY, \CARRYX)
endmatch

code cin
	if (carry->type == \CARRY)
		carry_in = port(carry, \CI);
	else
		carry_in = port(carry, \CIX);
endcode

match lut
	select lut->type == \LUT4
	index <SigSpec> port(lut, \I1) === cin[0]
	index <SigSpec> port(lut, \I2) === cin[1]
endmatch

code
	accept;
endcode
```

C-expressions in .pmg files

The C expressions in .pmg files support some additional non-C features:

- IdString constants can be written as `\foo` or `$bar`.
- `port(cell, portname)` looks up the SigSpec for the port and normalizes it with the built-in SigMap, for canonical signal representations in all index lookups.
- Similarly `param(cell, paramname)`
- `nusers(sigspec)` returns the number of different cells connected to any of the given signal bits, plus one if any of the signal bits is also a primary input or primary output.
- In some contexts there are additional keywords or operators, such as `===` in index statement, or `accept` and `reject` in code blocks.

Pattern Matcher API

Create pattern matcher object:

```
foobar_pm pm(module, module->selected_cells());
```

Find pattern “main” and call a c++11 lambda function for each match, and return number of matches:

```
int n = pm.run_main([&](){
    log("found matching carry cell: %s\n", log_id(pm.st_main.carry));
    log("          with lut   cell: %s\n", log_id(pm.st_main.lut));
    pm.st_main.carry->type = ID(CARRYZ);
    pm.blacklist(pm.st_main.carry);
});
```

`pm.blacklist(pm.st_main.carry)` tells the pattern matcher to ignore all further matches involving the same carry cell. Blacklisting should be used whenever a cell is modified by the callback function.

Match blocks

A “match..endmatch” block matches one cell in a pattern. The “return argument” is an implicit state variable of type Cell*. It can contain the following statements:

- `if cexpr` … conditional match. simply set cell to nullptr if false
- `select cexpr` … evaluate in pattern matcher constructor. can only access cell.
- `index <type> cell-cexpr === pattern-cexpr` … eval *cell-cexpr* in constructor and *pattern-cexpr* at runtime, lookup with hash table
- `filter cexpr` … additional run-time filter

```
state <bool> with_mul

match ff
    ...
endmatch

match mul
    if ff
    select mul->type == $mul
    select nusers(port(mul, \Y) == 2
    index <SigSpec> port(mul, \Y) === port(ff, \D)
    filter foo(mul) < bar(ff)
    optional
endmatch
```

- `[semi]optional` … return nullptr when no match instead of instant backtrack. `semioptional` only explores the nullptr branch when no proper match is found.

State variables and code blocks (1/2)

Additional code..endcode blocks can update state variables, and reject or accept. The arguments to the code statement are the state variables modified by the block.

- `accept;`
  accept this match, i.e. call the callback function

- `reject;`
  reject this match, i.e. backtrack

Example:

```
state <Cell*> addA addB

state <Cell*> addAB
state <SigSpec> sigS

...

code addAB sigS
    if (addA) {
        addAB = addA;
        sigS = port(addA, \B);
    }
    if (addB) {
        addAB = addB;
        sigS = port(addB, \A);
    }
    if (sigS.is_fully_const())
        reject;
endcode
```

State variables and code blocks (2/2)

Use “branch” to explore different choices.

```
state <int> mode

code mode
    for (mode = 0; mode < 8; mode++)
        branch;
    reject;
endcode
```

```
state <IdString> portAB

code portAB
    portAB = \A;
    branch;
    portAB = \B;
endcode
```

`udata` is like `state`, but not managed by the back-tracking mechanism. Use “finally” to inject your own back-tracking code.

```
udata <vector<Cell*>> stack

code
    stack.push_back(addAB);
    ...
finally
    stack.pop_back();
endcode
```

`accept` statements can be used inside the finally section, but not `reject`, or `branch`.

Slices and choices

Cell matches can contain "slices" and "choices". Slices can be used to create matches for different sections of a cell.

- `slice` *`var cexpr`*
  slices from *var*=0 to *var*=N-1 (with N=*cexpr*)

- `choice <type> var v0 v1 v2 … vN`
  choices from var=v0 to var=vN

Slice/chice variables are local to the match block. Use the `set` statement to copy values to state variables, if necessary.

```
state <int> pmux_slice

match pmux
    select pmux->type == $pmux
    slice idx GetSize(port(pmux, \S))
    index <SigBit> port(pmux, \S)[idx] === port(eq, \Y)
    set pmux_slice idx
endmatch
```

```
state <SigSpec> foo bar
state <IdString> eq_ab eq_ba

match eq
    select eq->type == $eq
    choice <IdString> AB {\A, \B}
    define <IdString> BA (AB == \A ? \B : \A)
    index <SigSpec> port(eq, AB) === foo
    index <SigSpec> port(eq, BA) === bar
    set eq_ab AB
    set eq_ba BA
generate
```

Subpatterns

A subpattern starts with a line containing the `subpattern` keyword followed by the name of the subpattern. Subpatterns can be called from a code block using a `subpattern(`*`subpattern_name`*`);` statement.

Arguments may be passed to subpattern via state variables. The subpattern line must be followed by a `arg` *`arg1 arg2`*... line that lists the state variables used to pass arguments.

Subpatterns can be called recursively.

If a subpattern statement is preceded by a fallthrough statement, this is equivalent to calling the subpattern at the end of the preceding block.

Testing pattern matchers with generate blocks

- Match blocks may contain an optional generate section that is used for automatic test-case generation.
- The expression `rng(n)` returns a non-negative integer less than n.
- The first argument to `generate` is the chance of this generate block being executed when the match block did not match anything, in percent.
- The second argument to generate is the chance of this generate block being executed when the match block did match something, in percent.
- The special statement finish can be used within generate blocks to terminate the current pattern matcher run.

```
match mul
    ...
generate 10 0
    SigSpec Y = port(ff, \D);
    SigSpec A = module->addWire(NEW_ID, GetSize(Y) - rng(GetSize(Y)/2));
    SigSpec B = module->addWire(NEW_ID, GetSize(Y) - rng(GetSize(Y)/2));
    module->addMul(NEW_ID, A, B, Y, rng(2));
endmatch
```

Lab Preview

Exercise 1: Hello World

Familiarize yourself with the yosys CLI and how to build plugins:

- `hello.cc` contains a pass that prints "Hello World!" to the log
- build the plugin containing this pass
- load the plugin and run the pass in batch mode
- load the plugin and display its help message in interactive mode

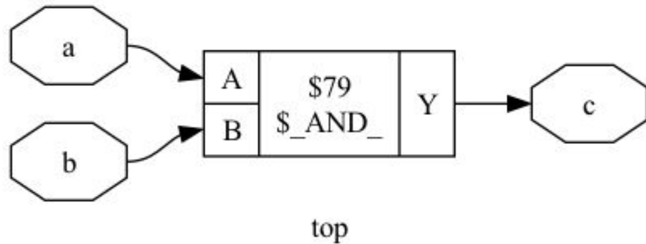
Exercise 2: Count wires and cells

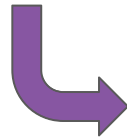
First steps in iterating through a design

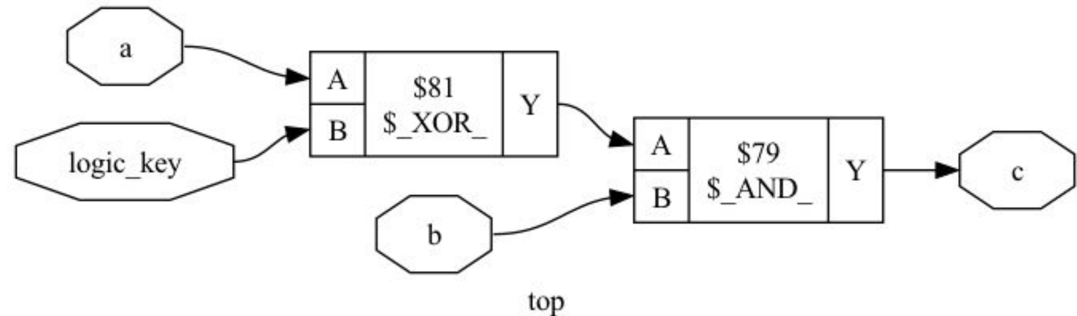
- In `count.cc`, add code that traverses the design and
- if the argument `-wires` is given, count the number of wires in the design
- if the argument `-cells` is given, count the number of cells in the design
- if both arguments are given, count both.
- Use `show` and `stat` to confirm the result.

Exercise 3: Logic Locking

- Logic locking is a form of encrypting circuits so that they only function correctly when the right key is provided
- A simple way to implement this is using XOR gates







YosysHQ

Thanks!

Any Questions?