Introduction to logic synthesis

31 May 2021 08:08 PM

RTL Design

- RTL Design
  - Behavioral representation of the required specification

```
module sample_code (
input clk,rst ,
output result,done);

always @ (posedge clk ,posedge rst)
 if(rst)
   ----
  else
   ---
endmodule
-----------------------------------------
-----------------------------------------
```

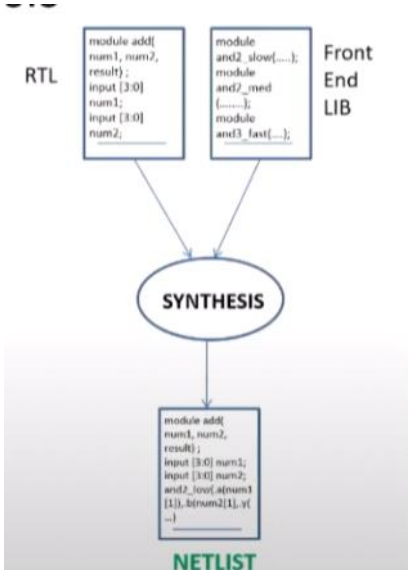
HDL

How to map from RTL code to hardware

RTL $\xrightarrow{\text{Synthesis}}$ Gate level.

* The RTL design is converted to gates and the connections are made between them.

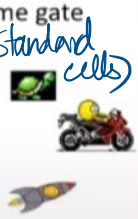
* This file is called a netlist

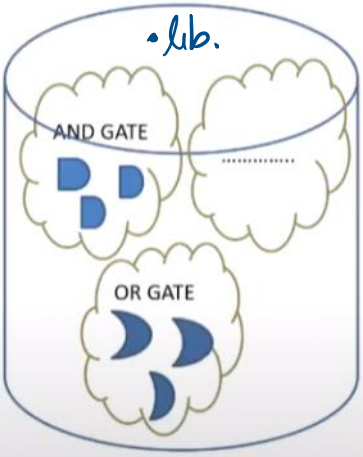


What is .lib

- .lib
  - Collection of logical modules.
  - Includes basic logic gates like And, Or , Not, etc...
  - Different flavors of same gate
    - 2 input And gate
      - Slow
      - Medium
      - Fast
    - 3 input And gate
      - Slow
      - Medium
      - Fast
    - 4 input And gate
    - .....
    - .....

why ? (Standard cells)



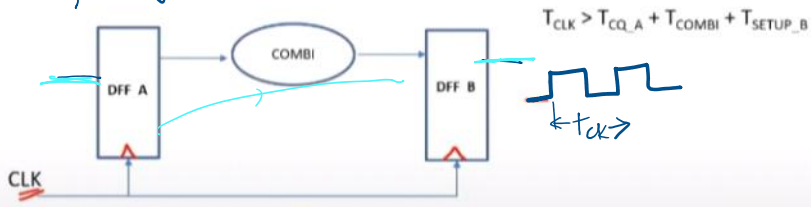


a .lib may not be exhaustive, but using its standard cells any boolean logic can be implemented

Why different flavours of gate

- Combinational delay in logic path determines the maximum speed of operation of digital logic circuit

(frequency of clk)



- So we need cells that work fast to make T_{COMBI} small
- Are faster cells sufficient ?

How large the period of the clock must be ?

* The clock period must be long enough that for a signal to complete its travel from DFF A to DFF B

* In other words, in 1 clock cycle the data must reach the DFF B input (not DFF B output)

1 clock cycle the following happens)

D_{FF} B 'y' (not D_{FF} B output)

Why? (in 1 clock cycle the following happens)

* Say both DFF A, DFF B are +ve edge triggered.
* The input to DFF A reaches DFF A out on ↑
  Meanwhile, the previous D_{FF} B input (output of combinational ckt reaches D_{FF} B out
* The D_{FF} A out now travels through the combinational ckt and reaches D_{FF} B in (it does not reflect at D_{FF} B out as it happens at the next +ve edge). This takes delay of comb ckt amount of time.
* It must reach D_{FF} B before the next +ve edge of the clock
  The time interval before which the signal must be ready at the input before arrival of ↑ is setup time (input must be stable before ↑)

∴ for data to travel from D_{FF} A in to D_{FF} B in in 1 clock cycle

$$T_{clk} > T_{prop\,DFF\,A} + T_{prop.combi} + T_{setup\,DFF\,B}$$

$$f = \frac{1}{T_{clk}} \quad (Hz)$$

* This is the minimum required clock period

If $T_{clk} < T_{clk\,min}$ then an unknown value will be sampled at B (since the data hasn't reached)

$$T_{clk\,min} = T_{prop\,DFF\,A} + T_{prop\,comb} + T_{set\,DFF\,B}$$

$$f_{clk\,max} = \frac{1}{T_{clk\,min}} = \frac{1}{T_{prop\,DFF\,A} + T_{prop.comb} + T_{set\,DFF\,B}}$$

∴ for maximum performance, we need smaller delays.

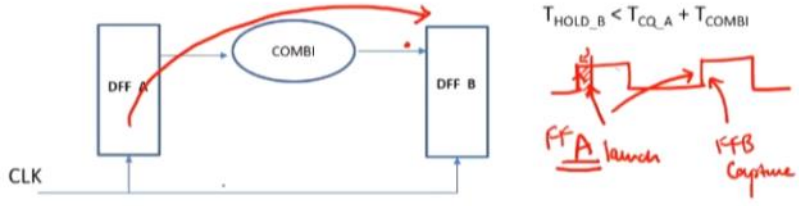
So we need faster gates

Now, qn arises

Since for faster speed we need faster gates (smaller clk prod) why we need slow and medium gates/cells in .lib

A Why do we need slow cells :-

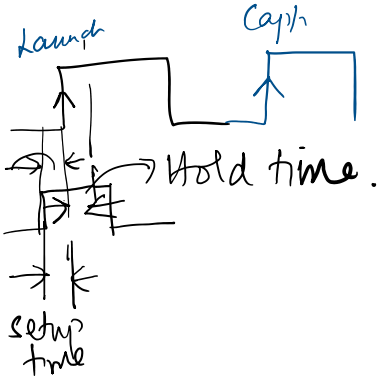
Why we need slow cells ?



- To ensure that there are no "HOLD" issues at DFF_B , we need cells that work slowly ☺
- Hence we need cells that work fast to meet the required performance and we need cells that work slow to meet HOLD
- The collection forms the .lib

Hold time

The duration after the arrival of the clk, the input must persist is the hold time



$T_{hold\ B} < T_{propA} + T_{propcomb}$

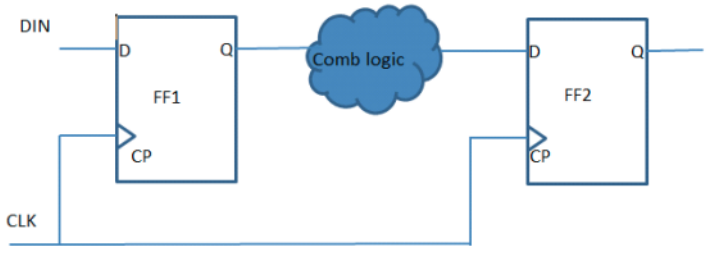
We donot require capture edge to capture the launch at A.

$T_{holdBmax} = T_{prop\ A} + T_{propcomb}$

Moreover

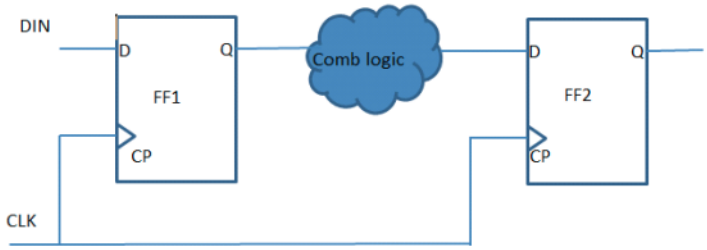
STA – Setup and Hold Time Analysis

Sini Mukundan December 16, 2013 22 Comments



STA – Setup and Hold Time Analysis

Sini Mukundan December 16, 2013 22 Comments



It is easy to get confused with the definitions of setup and hold violations. We are used to the definitions of setup and hold times for a single flipflop. The setup and hold violation checks done by STA tools are slightly different. PT aptly calls them max and min delay analysis. However, the other terminology is more common.

First a recap of the setup and hold time requirement of a flipflop.

Setup time is the minimum amount of time the data signal should be held steady before the clock event so that the data are reliably sampled by the clock. **Hold time** is the minimum amount of time the data signal should be held steady after the clock event so that the data are reliably sampled.

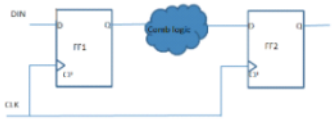
Setup Analysis (Max Delay Analysis)

Now, let us see what is meant by *setup analysis* for a timing path. Timing paths can be the following types:

1. Input port to a D pin of Flop.
2. CLK pin of Flop1 to D pin of Flop2
3. Q pin of flop to an output port
4. Input to output port through purely combinational logic.

We will take up a register to register path (2 above) for explanation.

We need to define some terms now. Consider the same clock goes to both FF1 & FF2. CLK is fed to both FF1&FF2. At clock edge A, FF1 is launching the data. That is, the D value causes a change in Q of FF1. So we call FF1 as *lauching flop* for this timing path. This signal will be captured at FF2 after one cycle.(For a single cycle path). So FF2 becomes our *capturing flop*, and clock edge B becomes our *capturing edge*.



Register to register path

Launch and Capture Edges

For ease of understanding, let us decide every component in the circuit is ideal. i.e. the flipflops have no setup and hold time requirements and clock is ideal. ie, CLK arrives at CP of FF1 & FF2 at 0 delay, edges coinciding.

The data path of the timing circuit is through CP of FF1 to D of FF2. Now let us calculate the delay encountered by data and clock while reaching FF2.

Data path delay

CLK->Q delay of FF1 + Comb path delay

In analysis, we call this the *Arrival Time*.

Clock path delay

0. (ideal clock)

However, we are checking the setup at the clock edge B. So we need to add one clock cycle to the clock path delay to get the *Required Time*.

Required Time = Clock period.

Data launched at FF1/CP should arrive at FF2/D in one clock period. So in setup check, we say a violation has occured if the data path delay is more than one clock cycle.

Now, let us increase the complication.

Clock has insertion delays. So the actual datapath delay is:

CLK delay till FF1/CP + CLK->Q delay of FF1 + Comb path delay

And the actual clock path delay is: CLK delay till FF2/CP.

Required Time = Clock period + CLK delay till FF2/CP.

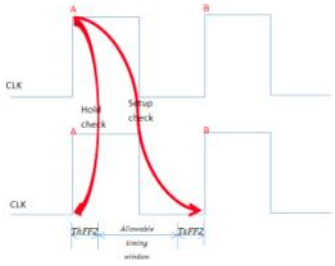
Now, we should also take into account the setup requirement of FF2. ie. Data at FF2/D should be stable for at least TsFF2 before the clock edge. So the required time for data arrival at FF2/D is *Clock period + CLK delay till FF2/CP- TsFF2* .If data arrives later than the clock path delay calculated above, the data won't be captured at edge B.

From above, it is clear that setup analysis checks for the maximum allowable delay for the timing path.

Hold Analysis (Min Delay Analysis)

Now let us move on to *Hold analysis*.

Let us take the same timing path above. Here you are verifying that the data is not captured at FF2/D on launching edge A of the CLK. ie. it is checking for the minimum delay the data should take to arrive at the second flop for the circuit to function correctly.



Timing Analysis with ideal clock

ThFF2 is the library hold time value of flop FF2. If the Data launched by FF1 reaches D of FF2 fast enough, it may be captured at the same clock edge A by the flop FF2. Hence the minimum delay requirement for the timing path is that the path to D should at least take more time than the hold time requirement of the flop FF2, so as not to corrupt the data. (Look up the .lib file for the hold time values for pin D relative to clcok pin CP of the flip flop type of FF2).

In simple terms, this makes sure the launched data does not arrive at the capture point too soon. Data launched from launching flop is allowed to arrive at the input of the second flop only after a delay greater than its hold requirement so that it is properly captured.

With clock delays in place, data path delay is
CLK delay till FF1/CP + CLK->Q delay of FF1 + Comb path delay
Arrival time = CLK delay till FF1/CP + CLK->Q delay of FF1 + Comb path delay
Clock path delay is CLK delay till FF2/CP.
Required Time = CLK delay till FF2/CP + Hold requirement of FF2

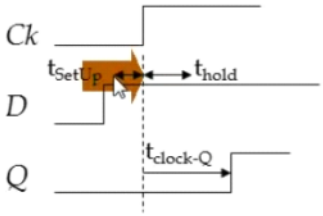
Note that we are no longer adding the clock period to the required time. This is because the hold is checked with the same clock edge, and setup with the next clock edge.

From the figure above,assuming ideal clock, there is a window of time which is between the minimum required(ThFF2) and the maximum allowed(Clock period – TsFF2) that the timing path can correctly have. If the data path takes less time that ThFF2, we say a hold violation has occurred. If the data path takes more time than Clockperiod-TsFF2, we say a setup violation has occurred.

Take a timing report and draw the clock and data path diagrams to understand this further.

What can happen with a timing violation?

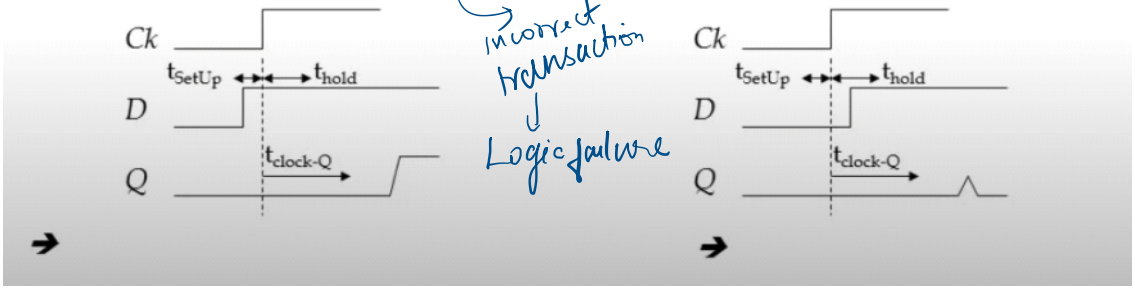
D changes outside setup and hold ➔ tclock-Q is correct



incorrect timing

slower than specified

D changes during setup and hold ➔ tclock-Q longer than specified, or Q does not transition correctly

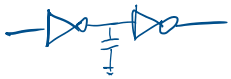


incorrect transaction → Logic failure

Faster Cells vs Slower Cell

Faster Cells vs Slower Cells

- Load in Digital Logic circuit → Capacitance



- Faster the charging / discharging of capacitance → Lesser the cell delay $\tau = RC$
  - To charge / discharge the capacitance fast , we need transistors capable of sourcing more current → wide transistors (more current) → A↑ P↑
  - Wider transistors -> Low Delay -> More Area and Power as well !! d↑
  - Narrow transistors -> More Delay -> Less Area and Power d↓
  - Faster cells donot come free , they come at penalty of area and power

$\tau = RC$

$C = \frac{\varepsilon A}{d}$

C↓ R↓ τ↓ (so more time to charge/discharge)

Power, Area, Speed ⇒ Tradeoff.

Selection of Cells

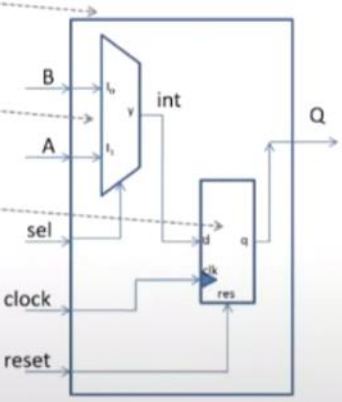
- Need to guide the Synthesizer to select the flavour of cells that is optimum for the implementation of logic circuit

This is called as Constraints.

- More use of faster cells
  - Bad circuit interms of Power and Area
  - Hold time violations ??
- More use of slower cells
  - Sluggish circuit , may not meet the performance need
- The guidance offered to the Synthesizer → "Constraints"

Synthesis (Illustration)

```
module (A,B,sel,clock, reset,Q)
input A,B,sel,clock,reset;
output Q;
wire  int;
assign int = sel ? A :B;
always @(posedge clock or posedge reset)
    begin
      if (reset)
      begin
          Q <= 1'b0;
      end
      else if (clk)
      begin
          Q <= int;
      end
    end
endmodule
```



The circuit on the right is created from RTL using the gates available in the .Lib and given out as Netlist.

Libraries naming

02 June 2021 04:06 PM

Libraries

Library Naming

Libraries in the SKY130 PDK are named using the following scheme;

> :lib_process:`<Process name>` _ :lib_src:`<Library Source Abbreviation>` _ :lib_type:`<Library Type Abbreviation>` [_ :lib_name:`<Library Name>`]

All sections are **lower case** and separated by an **underscore**. The sections are;

- The :lib_process:`Process name` in is the name of the process technology, for this PDK it is always :lib_process:`sky130`.
- The :lib_src:`Library Source Abbreviations` is a short abbreviation for who created and is responsible for the library. The table below shows the current list of :lib_src:`Library Source Abbreviations`;

Library Source	**:lib_src:`Library Source Abbreviation`**
The SkyWater Foundry	:lib_src:`fd`
Efabless	:lib_src:`ef`
Oklahoma State University	:lib_src:`osu`

- The :lib_type:`Library Type Abbreviation` is a short two letter abbreviation for the type of content found in the library. The table below shows the current list of :lib_type:`Library Type Abbreviations`;

Library Type	**:lib_type:`Library Type Abbreviation`**
Primitive Cells	:lib_type:`pr`
Digital Standard Cells	:lib_type:`sc`
Build Space (Flash, SRAM, etc)	:lib_type:`sp`
IO and Periphery	:lib_type:`io`
Miscellaneous	:lib_type:`xx`

- The :lib_name:`Library Name` is an optional short abbreviated name used when there are multiple libraries of a given type released from a single :lib_src:`library source`. If only one library of a given type is going to ever be released, this can be left out.

Creating New Libraries

Third party developers are encourage to create new and interesting libraries for usage with the SKY130 process technology. These libraries can even be included in the SKY130 PDK if it meets the following criteria;

> - It is released under an OSI approved license.
> - TODO: Finish the criteria.

:lib_type:`Primitive` Libraries

:lib_src:`Foundry` provided

```
.. toctree::
    :glob:
    :maxdepth: 1
    :caption: Foundry provided Primitives
    :name: sky130-lib-foundry-primitives

    libraries/sky130_fd_pr_*/README
```

:lib_type:`Digital Standard Cell` Libraries

:lib_src:`Foundry` provided :lib_type:`Digital Standard Cell` Libraries

```
.. toctree::
    :glob:
    :maxdepth: 1
    :name: sky130-lib-foundry-sc

    libraries/foundry-provided
    libraries/sky130_fd_sc_*/README
```

:lib_src:`Third party` provided :lib_type:`Digital Standard Cell` Libraries

```
.. toctree::
    :maxdepth: 1
    :name: sky130-lib-thirdparty-sc

    libraries/sky130_osu_sc/README
```

:lib_type:`Build Space` Libraries

The SKY130 currently offers two :lib_type:`build space` libraries. Build space libraries are designed to be used with technologies like memory compilers and built into larger macros. The provided libraries have specially crafted design rules to enable higher density compared to other libraries.

:lib_type:`IO and Periphery` Libraries

:lib_src:`Foundry` provided :lib_type:`IO and Periphery` Libraries

```
.. toctree::
    :maxdepth: 1
    :name: sky130-lib-foundry-io

    libraries/sky130_fd_io/README
```

:lib_src:`Third party` provided :lib_type:`IO and Periphery` Libraries

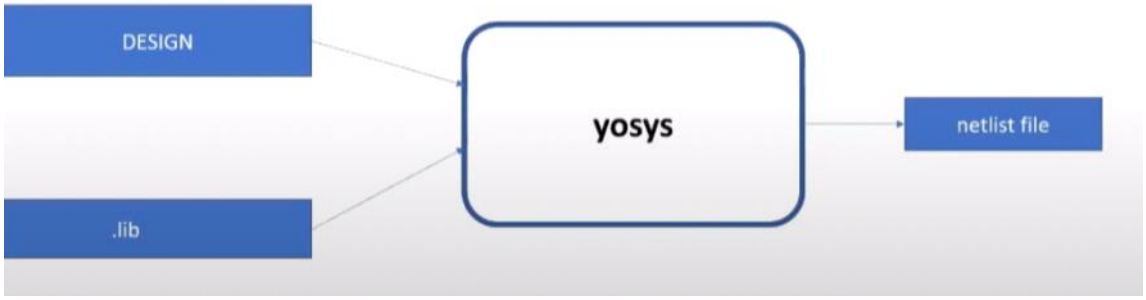
```
.. toctree::
    :maxdepth: 1
    :name: sky130-lib-thirdparty-io

    libraries/sky130_ef_io/README
```

Introduction to Yosys

01 June 2021 12:19 AM

Synthesizer

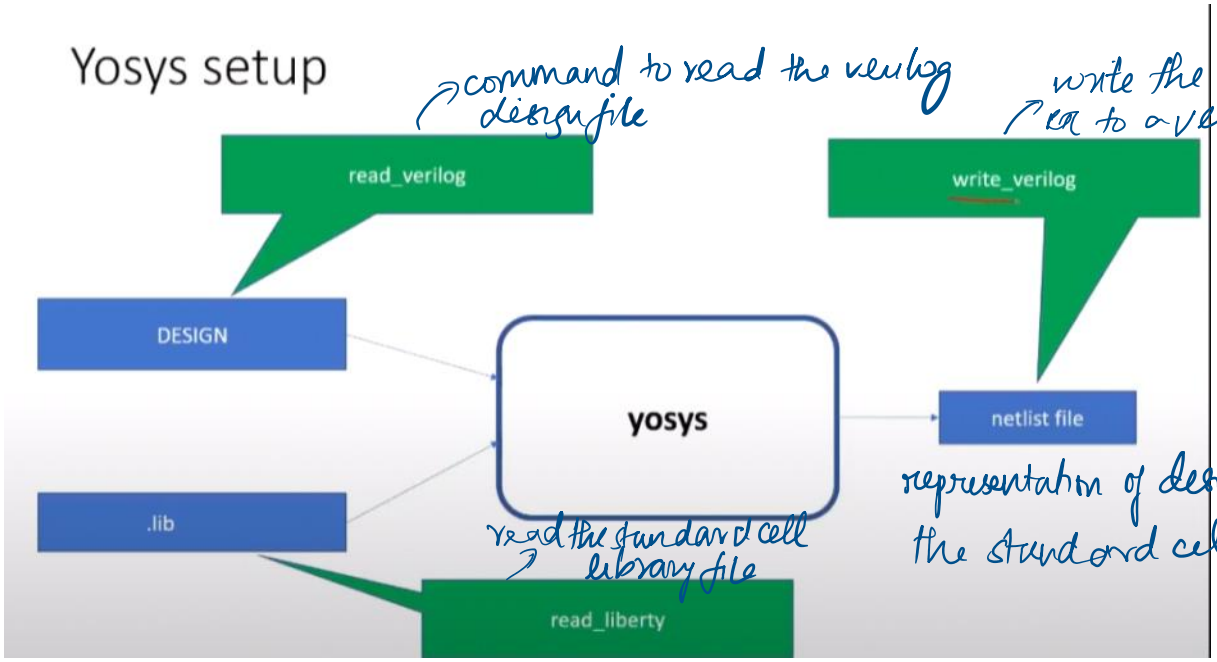
- Tool used for converting the RTL to netlist
- **Yosys** is the synthesizer used in this course



Yosys setup

command to read the verilog design file

write the synthesized o/p to a verilog netlist.

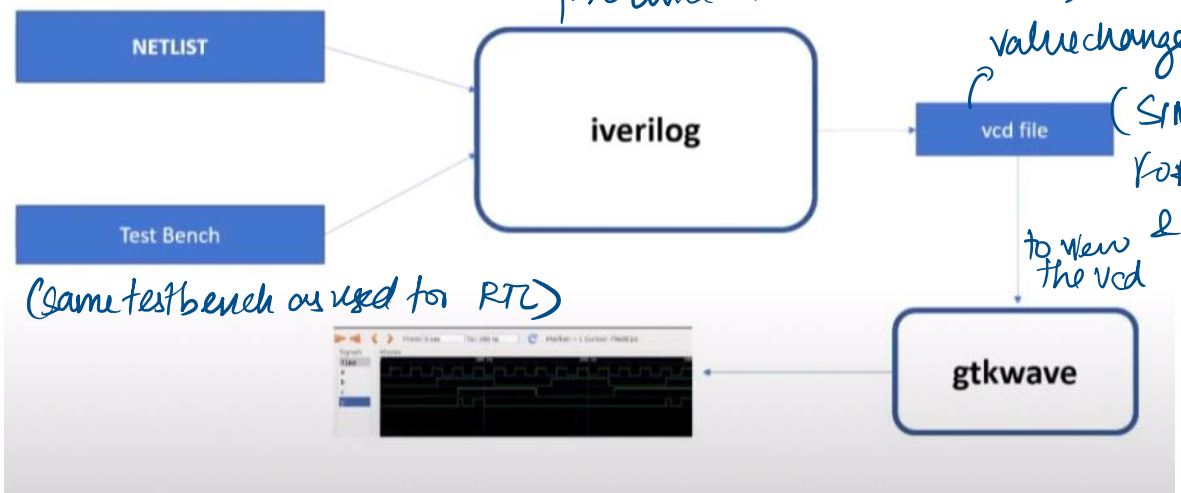


representation of design in form of the standard cells in .lib

read the standard cell library file

Verify the synthesis

(verify that the synthesized netlist and the design produce the same result)



value change dump file

(SIMULATOR ONLY SEES FOR CHANGE IN INPUT & EVALUATE THE O/P

If no change in i/p there is no evaluation of o/p)

to view the vcd

(same testbench as used for RTL)

(The obtained waveform from netlist must be same as that done in design simulation)
(RTL simulation)

NOTE

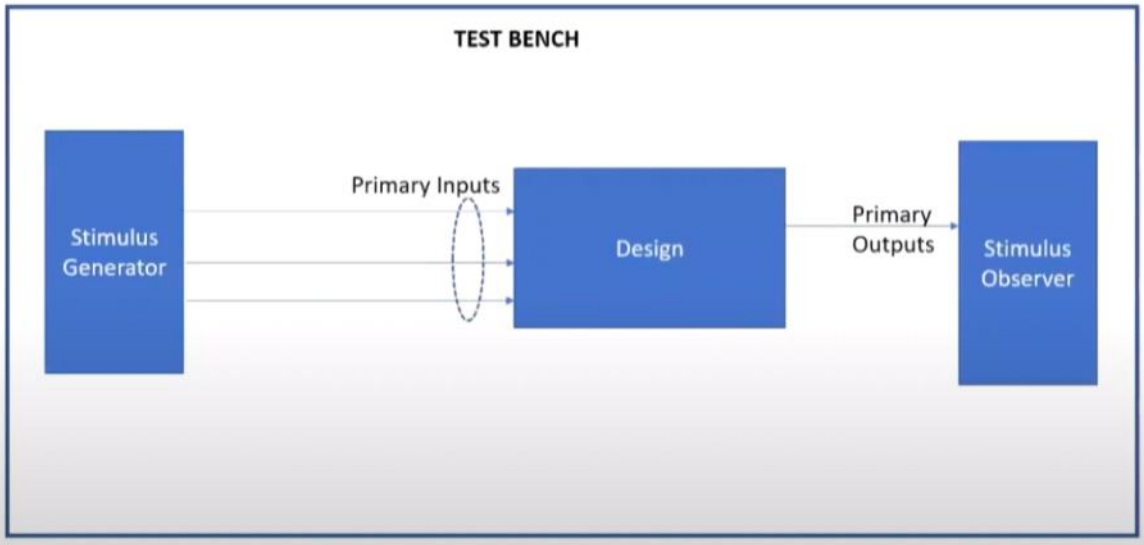
The set of Primary inputs / primary outputs will remain same between the RTL design and Synthesized netlist → Same Test bench can be used !!

iVerilog

How simulator works

- Simulator looks for the changes on the input signals
- Upon change to the input the output is evaluated
  - If no change to the input , no change to the output!
- Simulator is looking for change in the values of input!

(no change in input output will not be evaluated)



NOTE :

- **Design may have 1 or more Primary Inputs , 1 or more Primary outputs**
- **TB Doesn't have a Primary input or Primary outputs**

Lab Day 1

01 June 2021 05:42 AM

directory structure

(git clone vsdflow

Sky130 RTL Design and Synthesis Workshop:

```
(root shriharipc)-[/d/risc-v/sky130RTLDesignAndSynthesisWorkshop]
# ls
my_lib  README.md  verilog_files  yosys_run.sh
(root shriharipc)-[/d/risc-v/sky130RTLDesignAndSynthesisWorkshop]
# cd my_lib
(root shriharipc)-[/d/risc-v/sky130RTLDesignAndSynthesisWorkshop/my_lib]
# ls
lib  verilog_model
(root shriharipc)-[/d/risc-v/sky130RTLDesignAndSynthesisWorkshop/my_lib]
# cd lib
(root shriharipc)-[/d/risc-v/sky130RTLDesignAndSynthesisWorkshop/my_lib/lib]
# ls
sky130_fd_sc_hd__tt_025C_1v80.lib
(root shriharipc)-[/d/risc-v/sky130RTLDesignAndSynthesisWorkshop/my_lib/lib]
# cd ../verilog_model
(root shriharipc)-[/d/risc-v/sky130RTLDesignAndSynthesisWorkshop/my_lib/verilog_model]
# ls
primitives.v  sky130_fd_sc_hd.v
(root shriharipc)-[/d/risc-v/sky130RTLDesignAndSynthesisWorkshop/my_lib/verilog_model]
# cd ../../verilog_files
(root shriharipc)-[/d/risc-v/sky130RTLDesignAndSynthesisWorkshop/verilog_files]
# ls
bad_case_net.v          incomp_if2.v                tb_dff_async_set.v
bad_case.v              incomp_if.v                 tb_dff_const1.v
bad_counter.v           mul2_net.v                  tb_dff_const2.v
bad_latch_2.v           mult_2.v                    tb_dff_const3.v
bad_latch_net.v         mult_8.v                    tb_dff_const4.v
bad_latch.v             multiple_module_opt2.v      tb_dff_const5.v
bad_mux_net.v           multiple_module_opt.v       tb_dff_syncres.v
bad_mux.v               multiple_modules_flat.v     tb_good_counter.v
bad_shift_reg2.v        multiple_modules_hier.v     tb_good_latch.v
bad_shift_reg.v         multiple_modules.v          tb_good_mux.v
blocking_caveat_net.v   mux_generate.v              tb_good_shift_reg.v
blocking_caveat.v       mux_spice.v                 tb_incomp_case.v
comp_case.v             net.v                       tb_incomp_if2.v
counter_opt2.v          opt_check2.v                tb_incomp_if.v
counter_opt.v           opt_check3.v                tb_multiple_modules.v
demux_case.v            opt_check4.v                tb_mux_generate.v
demux_generate.v        opt_check.v                 tb_opt_check2.v
dff_ares_net.v          partial_case_assign.v       tb_opt_check3.v
dff_asyncres_net.v      pattern_detect_fsm_bad_style.v  tb_opt_check.v
dff_asyncres_syncres.v  pattern_detect_fsm.v        tb_partial_case_assign.v
dff_asyncres.v          rca.v                       tb_pattern_detect_fsm.v
dff_async_set.v         ripple_counter.v            tb_rca.v
dff_const1.v            tb_bad_case.v               tb_ripple_counter.v
dff_const2.v            tb_bad_counter.v            tb_ternary_operator_mux.v
dff_const3.v            tb_bad_latch2.v             tb_upcntr.v
dff_const4.v            tb_bad_latch.v              tb_up_dn_cntr.v
dff_const5.v            tb_bad_mux.v                tb_up_dn_cntr_with_load.v
dff_net.v               tb_bad_shift_reg2.v         tb_up_dn_cntr_with_load_with_start_stop.v
dff_syncres.v           tb_bad_shift_reg.v          ternary_operator_mux_net.v
fa.v                    tb_blocking_caveat.v        ternary_operator_mux.v
good_counter.v          tb_comp_case.v              upcntr.v
good_latch.v            tb_counter_opt.v            up_dn_cntr.v
good_mux_netlist.v      tb_demux_case.v             up_dn_cntr_with_load.v
good_mux.v              tb_demux_generate.v         up_dn_cntr_with_load_with_start_stop.v
good_shift_reg.v        tb_dff_asyncres_syncres.v
incomp_case.v           tb_dff_asyncres.v
(root shriharipc)-[/d/risc-v/sky130RTLDesignAndSynthesisWorkshop/verilog_files]
#
```

has the standard cell library (.lib file)

has the verilog model for the standard cell libraries

Verilog model for the standard cell libraries

sample verilog codes and their test benches.

Use iverilog to run the file

Saw we take good_mux.v and the testbench as tb_good_mux.v

Initially in the testbench 'include"good_mux.v" is not included so the iverilog has to be implemented

---
iverilog good_mux.v tb_good_mux.v

In the testbench there are already

$dumpfile("file.vcd");
$dumpvars;

So the value change dump (.vcd) file will be created when we run ./a.out

Instead if we include the good_mux.v as 'include in the testbench then the iverilog command will be
Iverilog tb_good_mux.v

The a.out is generated, then run the a.out file ./a.out to generate the vcd file

---
In WSL staert Vcxserv

export DISPLAY=:2
Gtkwave ilename.vcd

View the file

Stay in the verilog_files directory and open yosys. Just type yosys and give enter

First read the library file using
---

read_liberty -lib ../my_lib/lib/sky130_fd_sc_hd__tt_025c_1v80.lib
---
Read the verilog file using

Read_verilog good_mux.v

There must be no error

In case the design is spanning more than one verilog files all the files must be read

Then

Synth -top good_mux

In the place of good_mux type the top module name

```
# yosys

 /----------------------------------------------------------------------------\
 |                                                                            |
 |  yosys -- Yosys Open SYnthesis Suite                                       |
 |                                                                            |
 |  Copyright (C) 2012 - 2019  Clifford Wolf <clifford@clifford.at>           |
 |                                                                            |
 |  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.            |
 |                                                                            |
 \----------------------------------------------------------------------------/

Yosys 0.9 (git sha1 1979e0b)


yosys> read_liberty -lib ../my_lib/lib/sky130_fd_sc_hd__tt_025C_1v80.lib
1. Executing Liberty frontend.
Imported 428 cell types from liberty file.

yosys> read_verilog good_mux.v

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

```
yosys> synth -top good_mux

3. Executing SYNTH pass.

3.1. Executing HIERARCHY pass (managing design hierarchy).

3.1.1. Analyzing design hierarchy..
Top module:  \good_mux

3.1.2. Analyzing design hierarchy..
Top module:  \good_mux
Removed 0 unused modules.

3.2. Executing PROC pass (convert processes to netlists).

3.2.1. Executing PROC_CLEAN pass (remove empty switches from decision trees).
Cleaned up 0 empty switches.

3.2.2. Executing PROC_RMDEAD pass (remove dead branches from decision trees).
Marked 1 switch rules as full_case in process $proc$good_mux.v:3$1 in module good_mux.
Removed a total of 0 dead cases.

3.2.3. Executing PROC_INIT pass (extract init attributes).

3.2.4. Executing PROC_ARST pass (detect async resets in processes).

3.2.5. Executing PROC_MUX pass (convert decision trees to multiplexers).
Creating decoders for process `\good_mux.$proc$good_mux.v:3$1'.
     1/1: $0\y[0:0]

3.2.6. Executing PROC_DLATCH pass (convert process syncs to latches).
No latch inferred for signal `\good_mux.\y' from process `\good_mux.$proc$good_mux.v:3$1'.

3.2.7. Executing PROC_DFF pass (convert process syncs to FFs).

3.2.8. Executing PROC_CLEAN pass (remove empty switches from decision trees).
Found and cleaned up 1 empty switch in `\good_mux.$proc$good_mux.v:3$1'.
Removing empty process `good_mux.$proc$good_mux.v:3$1'.
```

.
.
.
.

```
=== good_mux ===

   Number of wires:                 4
   Number of wire bits:             4
   Number of public wires:          4
   Number of public wire bits:      4
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 1
     $_MUX_                         1

3.27. Executing CHECK pass (checking for obvious problems).
checking module good_mux..
found and reported 0 problems.
```

SUCCESS MESSAGE AFTER SUCCESSFUL SYNTHESIS

Now after synthesis is successful we need to generate the netlist file

---

Abc -liberty ../my_lib/lib/sky130_fd_sc_hd__tt_025C_1v80.lib

Abc -liberty path/to/the/.libfile

```
ABC: Scl_LibertyReadGenlib() skipped cell "sky130_fd_sc_hd__lpflow_decapkapwr_8" without logic function.
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__lpflow_inputisolatch_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfbbn_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfbbn_2".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfbbp_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfrbp_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfrbp_2".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfrtn_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfrtp_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfrtp_2".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfrtp_4".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfsbp_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfsbp_2".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfstp_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfstp_2".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfstp_4".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfxbp_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfxbp_2".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfxtp_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfxtp_2".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sdfxtp_4".
ABC: Scl_LibertyReadGenlib() skipped cell "sky130_fd_sc_hd__sdlclkp_1" without logic function.
ABC: Scl_LibertyReadGenlib() skipped cell "sky130_fd_sc_hd__sdlclkp_2" without logic function.
ABC: Scl_LibertyReadGenlib() skipped cell "sky130_fd_sc_hd__sdlclkp_4" without logic function.
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sedfxbp_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sedfxbp_2".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sedfxtp_1".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sedfxtp_2".
ABC: Scl_LibertyReadGenlib() skipped sequential cell "sky130_fd_sc_hd__sedfxtp_4".
ABC: Library "sky130_fd_sc_hd__tt_025C_1v80" from "/d/risc-v/sky130RTLDesignAndSynthesisWorkshop/verilog_f
iles/../my_lib/lib/sky130_fd_sc_hd__tt_025C_1v80.lib" has 334 cells (94 skipped: 63 seq; 13 tri-state; 18
no func; 0 dont_use).  Time =     0.16 sec
ABC: Memory =   16.17 MB. Time =     0.16 sec
ABC: Warning: Detected 9 multi-output gates (for example, "sky130_fd_sc_hd__fa_1").
ABC: + strash
ABC: + ifraig
ABC: + scorr
ABC: Warning: The network is combinational (run "fraig" or "fraig_sweep").
ABC: + dc2
ABC: + dretime
ABC: + retime
ABC: + strash
ABC: + &get -n
ABC: + &dch -f
ABC: + &nf
ABC: + &put
ABC: + write_blif <abc-temp-dir>/output.blif

4.1.2. Re-integrating ABC results.
ABC RESULTS:   sky130_fd_sc_hd__mux2_1 cells:       1
ABC RESULTS:        internal signals:       0
ABC RESULTS:           input signals:       3
ABC RESULTS:          output signals:       1
Removing temp directory.

yosys>
```

After the synthesis run the abc command so that the synthesised cells are realised as in the given liberty file

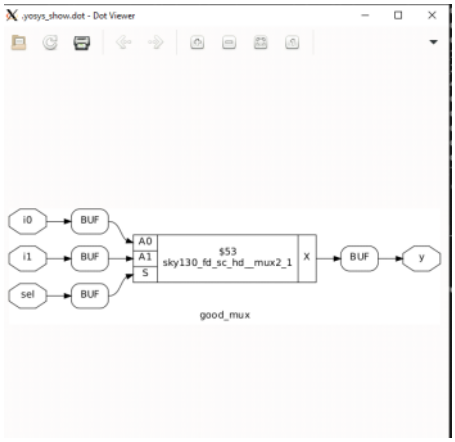
After this

Show

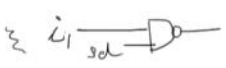
--

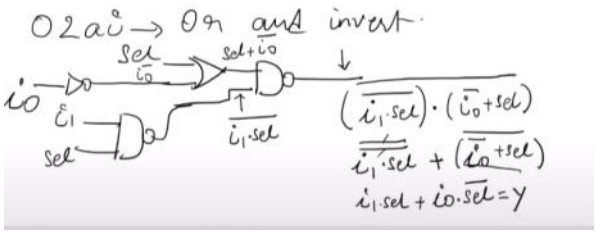
This shows the graphical version of the logic that has been realised

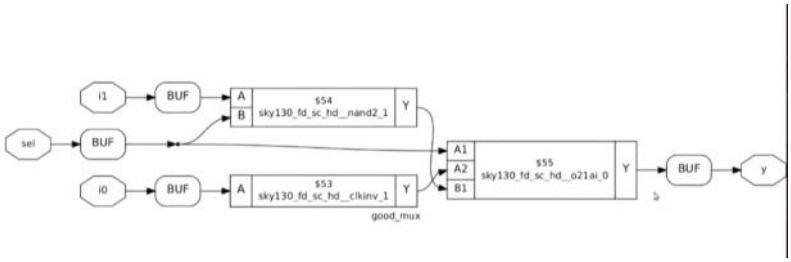
```
4.1.2. Re-integrating ABC results.
ABC RESULTS:   sky130_fd_sc_hd__mux2_1 cells:       1
ABC RESULTS:        internal signals:       0
ABC RESULTS:           input signals:       3
ABC RESULTS:          output signals:       1
Removing temp directory.
```



The output obtained in the video







Introduction to dot lib parts

01 June 2021 07:03 AM

test.sh U | sky130_fd_sc_hd__tt_025C_1v80.lib M

my_lib > lib > sky130_fd_sc_hd__tt_025C_1v80.lib

```
library ("sky130_fd_sc_hd__tt_025C_1v80") {
    define(def_sim_opt,library,string);
    define(default_arc_mode,library,string);
    define(default_constraint_arc_mode,library,string);
    define(driver_model,library,string);
    define(leakage_sim_opt,library,string);
    define(min_pulse_width_mode,library,string);
    define(simulator,library,string);
    define(switching_power_split_model,library,string);
    define(sim_opt,timing,string);
    define(violation_delay_degrade_pct,timing,string);
    technology("cmos");
    delay_model : "table_lookup";
    bus_naming_style : "%s[%d]";
    time_unit : "1ns";
    voltage_unit : "1V";
    leakage_power_unit : "1nW";
    current_unit : "1mA";
    pulling_resistance_unit : "1kohm";
    capacitive_load_unit(1.0000000000, "pf");
    revision : 1.0000000000;
    default_cell_leakage_power : 0.0000000000;
    default_fanout_load : 0.0000000000;
    default_inout_pin_cap : 0.0000000000;
    default_input_pin_cap : 0.0000000000;
    default_max_transition : 1.5000000000;
    default_output_pin_cap : 0.0000000000;
    default_arc_mode : "worst_edges";
    default_constraint_arc_mode : "worst";
    default_leakage_power_density : 0.0000000000;
    default_operating_conditions : "tt_025C_1v80";
    operating_conditions ("tt_025C_1v80") {
```

⇒ technology (cmos/FinFET etc.)

⇒ unit standardisation

P T V

sky130_fd_sc_hd__tt_025C_1v80.lib M

Process: tt ⇒ typical (it can be slow fast or typical)

Temp: 025C ⇒ 25°C

P V T ⇒ This determines how the silicon works

P → Process → Variations due to fabrication

V → Voltage → Variation due to voltage

T → Temperature. → Variation due to temperature.

Volt ⇒ 1v80

```
operating_conditions ("tt_025C_1v80") {
    voltage : 1.8000000000;
    process : 1.0000000000;
    temperature : 25.000000000;
    tree_type : "balanced_tree";
}
```

Voltage
process
Temperature.

cell definition.

```
cell ("sky130_fd_sc_hd__a2111o_1") {
    leakage_power () {
        value : 0.0017945000;
        when : "!A1&!A2&!B1&!C1&D1";
    }
    leakage_power () {
        value : 0.0105548000;
        when : "!A1&!A2&!B1&!C1&!D1";
    }
    leakage_power () {
        value : 0.0004483000;
        when : "!A1&!A2&!B1&C1&D1";
    }
    leakage_power () {
        value : 0.0009140000;
        when : "!A1&!A2&!B1&C1&!D1";
    }
    leakage_power () {
        value : 0.0004413000;
        when : "!A1&!A2&B1&!C1&D1";
    }
    leakage_power () {
        value : 0.0008205000;
        when : "!A1&!A2&B1&!C1&!D1";
    }
    leakage_power () {
        value : 0.0004191000;
        when : "!A1&!A2&B1&C1&D1";
    }
    leakage_power () {
        value : 0.0004435000;
        when : "!A1&!A2&B1&C1&!D1";
    }
    leakage_power () {
        value : 0.0017945000;
        when : "!A1&A2&!B1&!C1&D1";
    }
    leakage_power () {
        value : 0.0110122000;
        when : "!A1&A2&!B1&!C1&!D1";
    }
```

and 2 111 or

for each i/p combination.

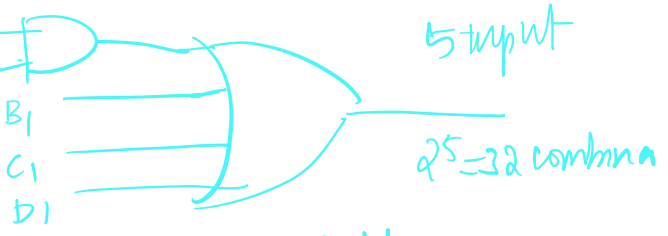
00 000

00 001

00 010

00 011

the behaviour can be seen in verilog model



5 input

$2^5 = 32$ combina

verilog model fib.

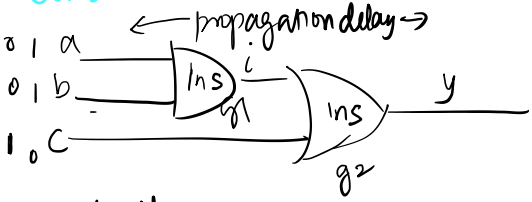
```
`ifndef SKY130_FD_SC_HD__A2111O_1_V
`define SKY130_FD_SC_HD__A2111O_1_V

/**
 * a2111o: 2-input AND into first input of 4-input OR.
 *
 *         X = ((A1 & A2) | B1 | C1 | D1)
 *
 * Verilog wrapper for a2111o with size of 1 units.
 *
 * WARNING: This file is autogenerated, do not modify directly!
 */

`timescale 1ns / 1ps
`default_nettype none


`ifdef USE_POWER_PINS
/*********************************************************/

`celldefine
module sky130_fd_sc_hd__a2111o_1 (
    X   ,
    A1  ,
    A2  ,
    B1  ,
    C1  ,
    D1  ,
    VPWR,
    VGND,
    VPB ,
    VNB
);

    output X   ;
    input  A1  ;
    input  A2  ;
    input  B1  ;
    input  C1  ;
    input  D1  ;
    input  VPWR;
    input  VGND;
    input  VPB ;
    input  VNB ;
    sky130_fd_sc_hd__a2111o base (
        .X(X),
```

```
    input  VNB ;
    sky130_fd_sc_hd__a2111o base (
        .X(X),
        .A1(A1),
        .A2(A2),
        .B1(B1),
        .C1(C1),
        .D1(D1),
        .VPWR(VPWR),
        .VGND(VGND),
        .VPB(VPB),
        .VNB(VNB)
    );

endmodule
`endcelldefine

/*****************************************************/
`else // If not USE_POWER_PINS
/*****************************************************/
```

Flip Flop coding styles

02 June 2021 09:05 AM

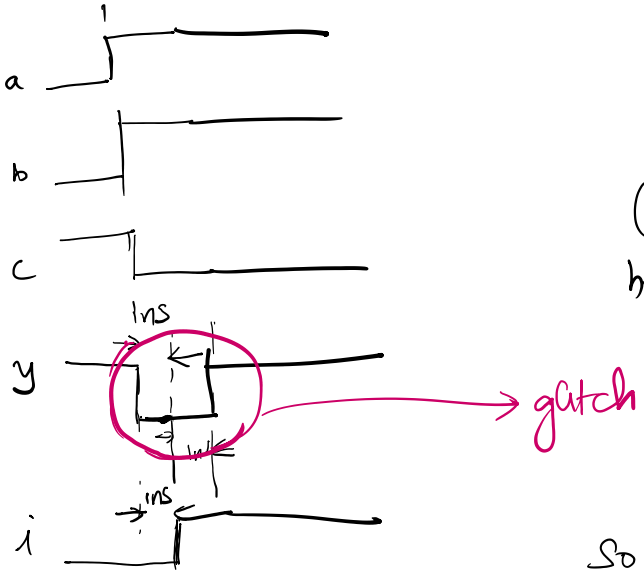
Consider a combination ckt



a	b	c	y
0	0	1	1 → initially
1	1	0	1 →

initially a=0 b=0 c=1

then at t=0s, a=1, b=1, c=0



as c is connected g2 c reaches g2 first and i was 0, so y becomes 0 (1ns it takes)

but from 0→1ns, i becomes 1 and after another 1ns y becomes 1

So for

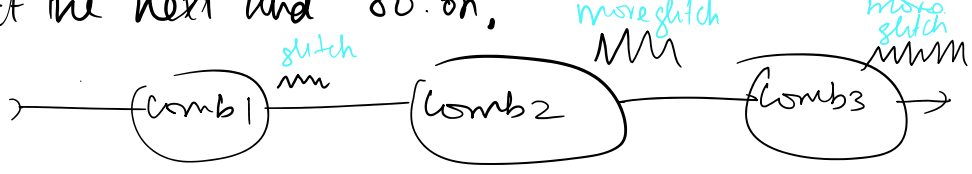
0 0 1 to 1 1 0

y must always be 1

but there is a 2ns glitch.

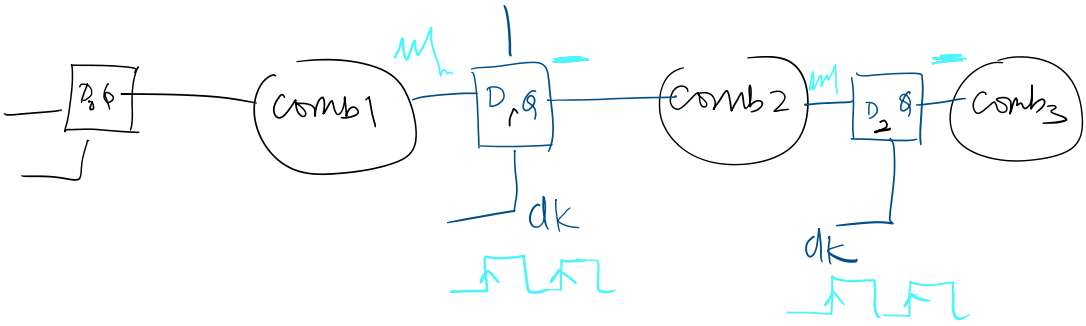
∴ The design will have combination ckts, so more the combinational ckts, the no of glitches increases

If we have many combinational circuits then the output will never settle as glitches in previous ckts affect the next and so on.



To avoid this we need an element to store the value. We use D-flipflop

value we use D-flip flop



input at D is reflected at Q only at the +ve edge of the clk
So the input to the next comb2 has a stable value.
So even though comb1 glitches, the D-ff samples the correct value

D-ff samples the value after glitch sec.

$T_{clk_0} > T_{propD_0} + T_{comb1} + T_{setup1}$ $T_{hold} < T_{propD_0} + T_{comb1}$

$T_{clk_1} > T_{propD_1} + T_{comb2} + T_{setup2}$. $T_{hold} < T_{propD_1} + T_{comb2}$

$\max(T_{clk_0}, T_{clk_1})$

Now we have inserted the flip-flops. So we have to initalise the flip flop.
If we don't initalise the flip flop then it will take a garbage value at Q before D is reflected at Q

So we have RESET / SET control pin in the flop.

Synchronous Asynchronous

Asynchronous reset.

verilog_files > dff_asyncres.v

```
1
2   module dff_asyncres ( input clk ,  input async_reset , input d , output reg q
3   always @ (posedge clk , posedge async_reset)
4   begin
5       if(async_reset)
6           q <= 1'b0;
7       else
8           q <= d;
9   end
10  endmodule
11
```

→ either clock posedge when async_reset=1 (or) async_reset posedge.

clk ↑ async_reset ↑

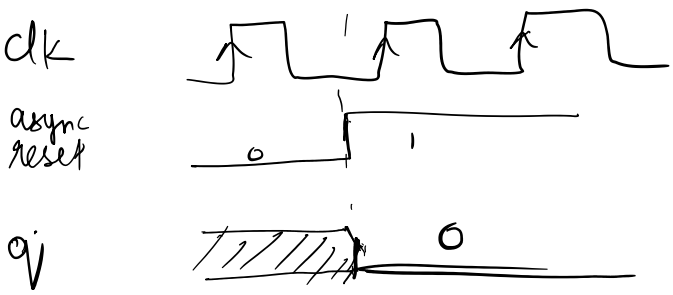
Why asynchronous reset? Because it doesn't look for/wait for a clock

(The reset signal is given in always block event control expression)

If the reset signal is not present in the event control expression but only inside the always block, then it is a synchronous reset since only at the posedge of clk reset happens.

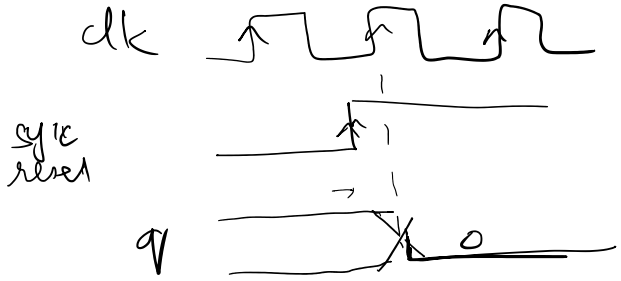
But in async_reset, irrespective of the clock, the reset can be done.

Async reset



(q has changed irrespective of clk)

Sync reset

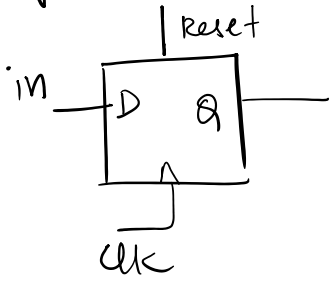


output changes only at the clk

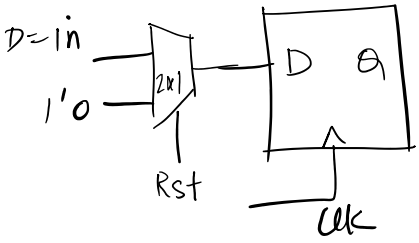
reset pulls the signal to 0

set " " 1 (q='b1) is the only difference.

Async reset



Sync reset



Async

verilog_files > dff_asyncres.v

```
1
2  module dff_asyncres ( input clk ,  input async_reset , input d , output reg q
3  always @ (posedge clk , posedge async_reset)
4  begin
5      if(async_reset)
6          q <= 1'b0;
7      else
8          q <= d;
9  end
10 endmodule
11
```

Sync

```
module dff_syncres ( input clk , input async_reset , input sync_reset , input
always @ (posedge clk )
begin
    if (sync_reset)
        q <= 1'b0;
    else
        q <= d;
end
endmodule
```

only upon clk. the ckt is evaluated.

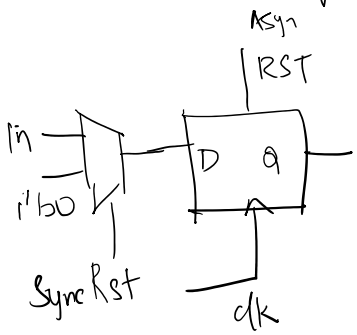
Both Async and Sync reset

test.sh U dff_asyncres_syncres.v M

verilog_files > dff_asyncres_syncres.v

```
1  module dff_asyncres_syncres ( input clk , input async_reset , input sync_reset , input d
2  always @ (posedge clk , posedge async_reset)
3  begin
4      if(async_reset)
5          q <= 1'b0;
6      else if (sync_reset)
7          q <= 1'b0;
8      else
9          q <= d;
10 end
11 endmodule
12
```

This is correct and safe.



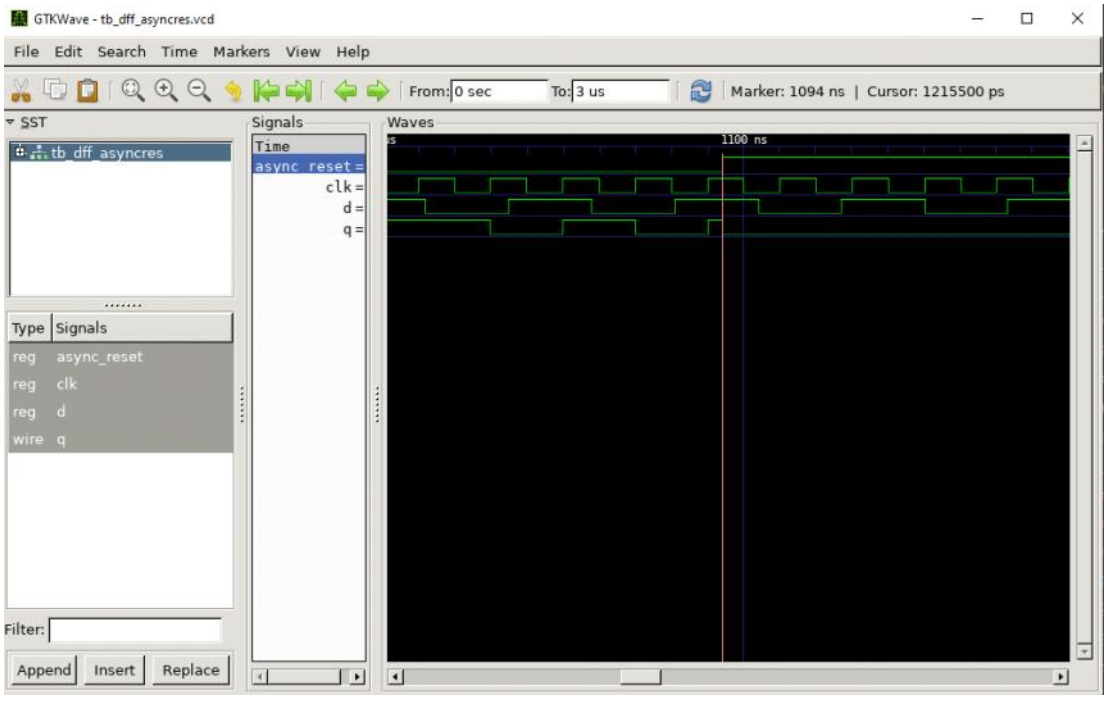
DFF synthesis

02 June 2021 01:09 PM

DFF ASYNCHRONOUS

VERIOG SIMULATION AND VIEWING WAVEFORM

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

┌──(root💀shriharipc)-[/d/RISC-V/sky130RTLDesignAndSynthesisWorkshop/verilog_files]
└─# iverilog dff_asyncres.v tb_dff_asyncres.v
┌──(root💀shriharipc)-[/d/RISC-V/sky130RTLDesignAndSynthesisWorkshop/verilog_files]
└─# ./a.out
VCD info: dumpfile tb_dff_asyncres.vcd opened for output.
┌──(root💀shriharipc)-[/d/RISC-V/sky130RTLDesignAndSynthesisWorkshop/verilog_files]
└─# export DISPLAY=:2
┌──(root💀shriharipc)-[/d/RISC-V/sky130RTLDesignAndSynthesisWorkshop/verilog_files]
└─# gtkwave tb_dff_asyncres.vcd

GTKWave Analyzer v3.3.104 (w)1999-2020 BSI

[0] start time.
[3000000] end time.
WM Destroy
┌──(root💀shriharipc)-[/d/RISC-V/sky130RTLDesignAndSynthesisWorkshop/verilog_files]
└─#
```



LOGIC SYNTHESIS IN YOSYS

read_liberty -lib /d/RISC-V/sky130RTLDesignAndSynthesisWorkshop/my_lib/lib/sky130_fd_sc_hd__tt_025C_1v80.lib

read_verilog dff_asyncres.v

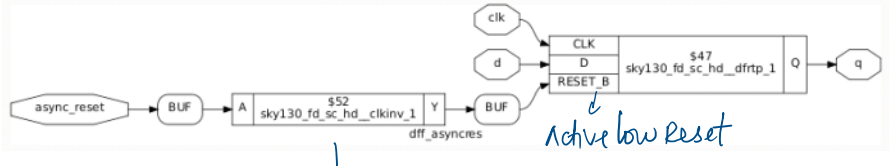
synth -top dff_asnces

dfflibmap -liberty /d/RISC-V/sky130RTLDesignAndSynthesisWorkshop/my_lib/lib/sky130_fd_sc_hd__tt_025C_1v80.lib

abc -liberty ../my_lib/lib/sky130_fd_sc_hd__tt_025C_1v80.lib

show

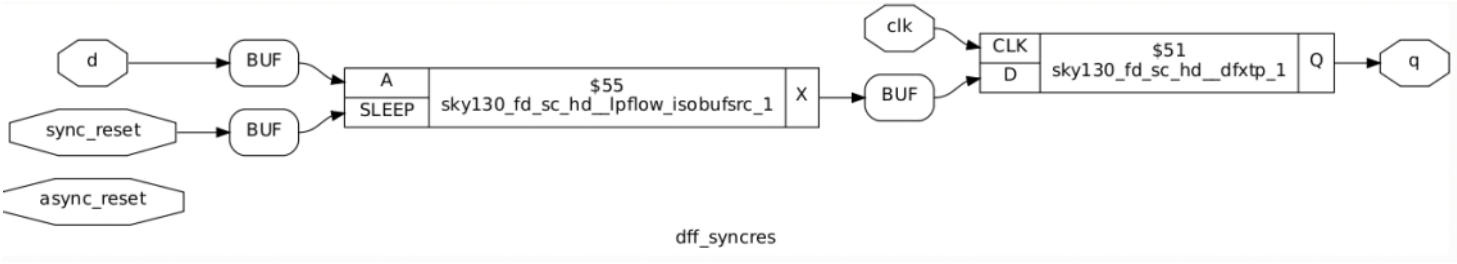
Since we are using dff we have to use a keyword called dff lib map. Because many times in the flow , there used to be a separate standard cell library and a flop library, but here we have it in the same library, so we point back to the same library



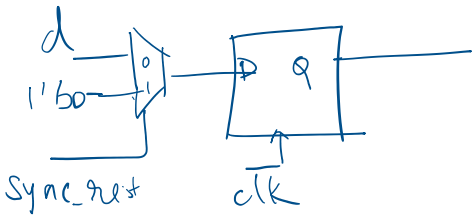
why?

We wrote a flop with active high reset, but the flop in the cell has active low reset so the tool inserted an inverter

SHOW OF SYNCHRONOUS RESET




```
/**
 * lpflow_isobufsrc: Input isolation, noninverted sleep.
 *
 *                   X = (!A | SLEEP)
 *
 * Verilog wrapper for lpflow_isobufsrc with size of 1 units.
 *
 * WARNING: This file is autogenerated, do not modify directly!
 */
```



$$D = \overline{A} + sync\_reset$$

Hierarichal Synthesis and Flat synthesis

02 June 2021 03:34 PM

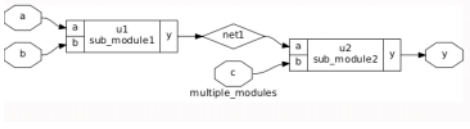
verilog_files > multiple_modules.v

```
1  module sub_module2 (input a, input b, output y);
2      assign y = a | b;
3  endmodule
4
5  module sub_module1 (input a, input b, output y);
6      assign y = a&b;
7  endmodule
8
9
10 module multiple_modules (input a, input b, input c , output y);
11     wire net1;
12     sub_module1 u1(.a(a),.b(b),.y(net1));  //net1 = a&b
13     sub_module2 u2(.a(net1),.b(c),.y(y));  //y = net1|c ,ie y = a&b + c;
14 endmodule
15
```

After synth -top multiple_modules

```
3.26. Printing statistics.

=== multiple_modules ===

   Number of wires:                 5
   Number of wire bits:             5
   Number of public wires:          5
   Number of public wire bits:      5
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 2
     sub_module1                    1
     sub_module2                    1

=== sub_module1 ===

   Number of wires:                 3
   Number of wire bits:             3
   Number of public wires:          3
   Number of public wire bits:      3
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 1
     $_AND_                         1

=== sub_module2 ===

   Number of wires:                 3
   Number of wire bits:             3
   Number of public wires:          3
   Number of public wire bits:      3
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 1
     $_OR_                          1

=== design hierarchy ===

   multiple_modules                 1
     sub_module1                    1
     sub_module2                    1

   Number of wires:                11
   Number of wire bits:            11
   Number of public wires:         11
   Number of public wire bits:     11
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 2
     $_AND_                         1
     $_OR_                          1
```

After executing abc, and using "show multiple_modules"



Here we cant see the and and the or gate but instead the sub_modules themselves.
This is the hierarichical design

Now lets write the netlist to a verilog file usting the command

write_verilog -noattr multiple_modules_hier.v //the no attr makes the file more readable by removing the attributes in (* *)

```
/* Generated by Yosys 0.9 (git sha1 1979e
module multiple_modules(a, b, c, y);
  input a;
  input b;
  input c;
  wire net1;
  output y;
  sub_module1 u1 (
    .a(a),
    .b(b),
    .y(net1)
  );
  sub_module2 u2 (
    .a(net1),
    .b(c),
    .y(y)
  );
endmodule

module sub_module1(a, b, y);
  wire _0_;
  wire _1_;
  wire _2_;
  input a;
  input b;
  output y;
  sky130_fd_sc_hd__and2_0 _3_ (
    .A(_1_),
    .B(_0_),
    .X(_2_)
  );
  assign _1_ = b;
  assign _0_ = a;
  assign y = _2_;
endmodule
```

We can see here that the hierarichy is preserved

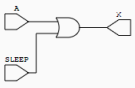
Submodule 1 is an and gate

This is an and gate

The standard cell module for 2 input and gate has been instantiated

And then the module has been rewritten as an and gate

```
`ifndef SKY130_FD_SC_HD__LPFLOW_INPUTISO1P_1_V
`define SKY130_FD_SC_HD__LPFLOW_INPUTISO1P_1_V

/**
 * lpflow_inputiso1p: Input isolation, noninverted sleep.
 *
 *                    X = (A & !SLEEP)
 *
 * Verilog wrapper for lpflow_inputiso1p with size of 1 units.
 *
 * WARNING: This file is autogenerated, do not modify directly!
 */
```



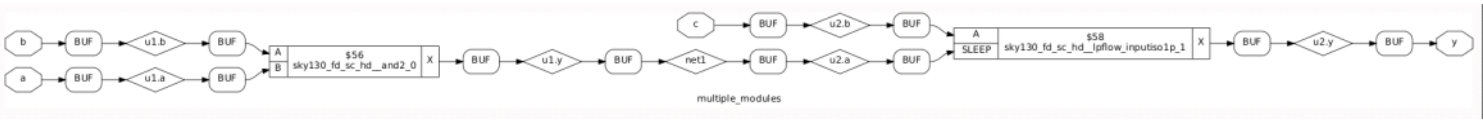
Flatten - after abc command use the flatten command

verilog_files > multiple_modules_flatten.v

```
/* Generated by Yosys 0.9 (git sha1 1979e0b) */

module multiple_modules(a, b, c, y);
  wire _0_;
  wire _1_;
  wire _2_;
  wire _3_;
  wire _4_;
  wire _5_;
  input a;
  input b;
  input c;
  wire net1;
  wire \u1.a ;
  wire \u1.b ;
  wire \u1.y ;
  wire \u2.a ;
  wire \u2.b ;
  wire \u2.y ;
  output y;
  sky130_fd_sc_hd__and2_0 _6_ (
    .A(_1_),
    .B(_0_),
    .X(_2_)
  );
  sky130_fd_sc_hd__lpflow_inputiso1p_1 _7_ (
    .A(_4_),
    .SLEEP(_3_),
    .X(_5_)
  );
  assign \u1.a  = a;
  assign \u1.b  = b;
  assign net1 = \u1.y ;
  assign _1_ = \u1.b ;
  assign _0_ = \u1.a ;
  assign \u1.y  = _2_;
  assign \u2.a  = net1;
  assign \u2.b  = c;
  assign y = \u2.y ;
  assign _4_ = \u2.b ;
  assign _3_ = \u2.a ;
  assign \u2.y  = _5_;
endmodule

```

Here we can see that all the modules are in the same netlist

After flatten


b
a
BUF
u1.b
u1.a
A
B
$56
sky130_fd_sc_hd__and2_0
X
u1.y
net1
c
u2.b
u2.a
A
SLEEP
$58
sky130_fd_sc_hd__lpflow_inputiso1p_1
u2.y
y
multiple_modules

Sub module level synthesis

02 June 2021 04:54 PM

How

Use the same file multiple_modules.v

Proceed with the same commands until synth command

In the synth command

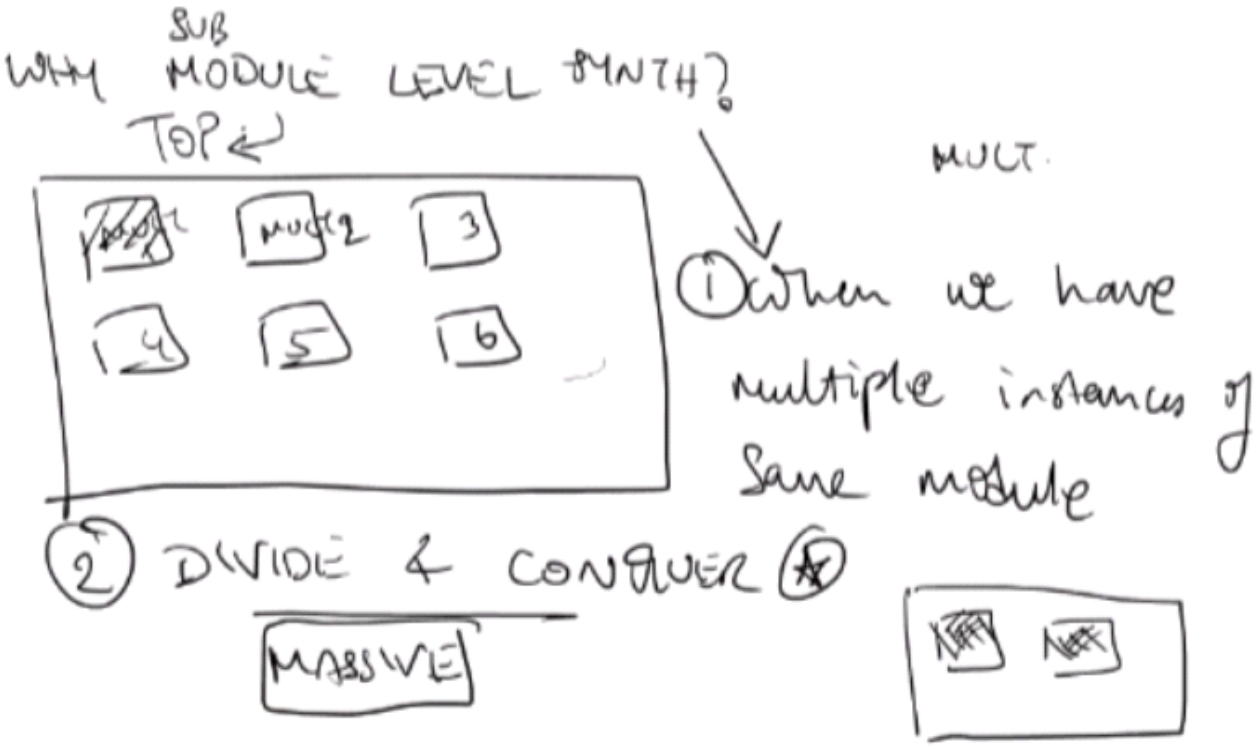
Synth -top submodule2

Abc -liberty /d/…./sky130blahblah.lib

Why synthesise only the submodule?

Sometimes if a top module has many sub modules and if many of the sub modules are repeated then it is easier to synthesise each module and then replicate it, than synthesising every module again and again

(Or) We want to divide and conquer approach. When the design is massive



Then the netlist for each can be combined module to the required design

Multiplication - Optimisations

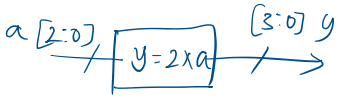
02 June 2021 05:29 PM

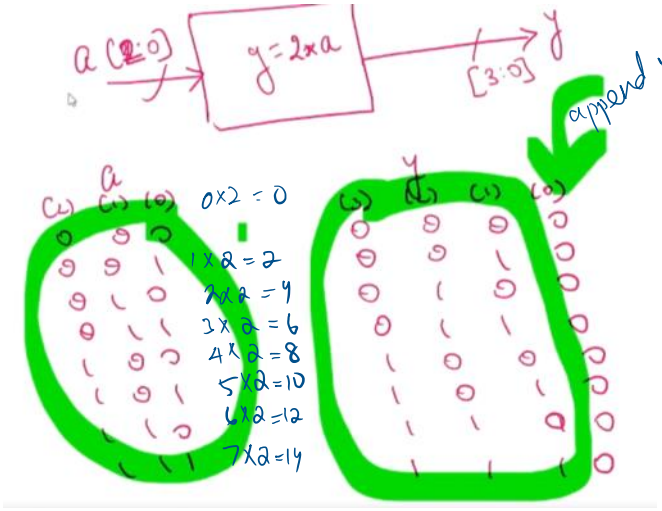
File used : mul_2.v

dff_syncres.v M | mult_2.v M

verilog_files > mult_2.v

```
1  module mul2 (input [2:0] a, output [3:0] y);
2      assign y = a * 2;
3  endmodule
4
```





we can see that

$$a[2:0] = y[3:1]$$

So multiplying with 2 is {a, 0} appending 0

So we no need any hardware for this

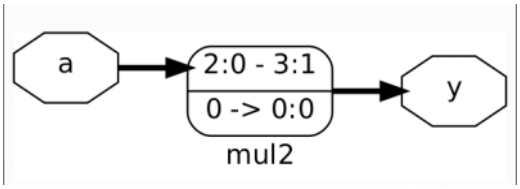
$y[3] = a[2]$
$y[2] = a[1]$
$y[1] = a[0]$
$y[0] = 0$

Multiplying by 2^n is just left shift through 'n' bits

Dividing by 2^n is right shift through 'n' bits

```
18.26. Printing statistics.

=== mul2 ===

   Number of wires:                 2
   Number of wire bits:             7
   Number of public wires:          2
   Number of public wire bits:      7
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 0

18.27. Executing CHECK pass (checking for obvious problems).
checking module mul2..
found and reported 0 problems.
```

→ we can see that yosys has internally optimised.



verilog_files > mul2_net.v

```
1  /* Generated by Yosys 0.9 (git sha1 1979e0b) */
2
3  module mul2(a, y);
4      input [2:0] a;
5      output [3:0] y;
6      assign y = { a, 1'h0 };
7  endmodule
8
```

Consider

a[2:0]
3 bits

y[5:0]
6-bits

$y = a * 9$

$y = a * [8+1]$

$y = a \times 8 + a \times 1$

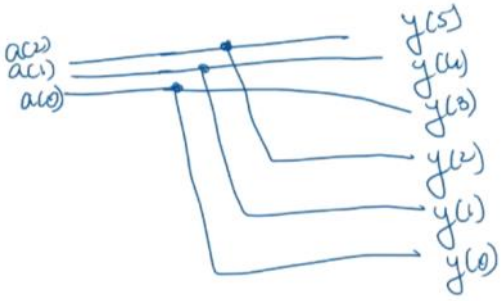
This works for only $a \Rightarrow$ 3 bits

$a[2:0]$ 000

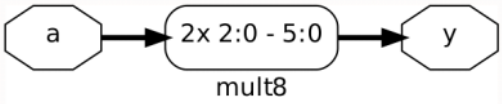
$a[2:0]$ +

$y = a[2:0]a[2:0]$

$y = aa$



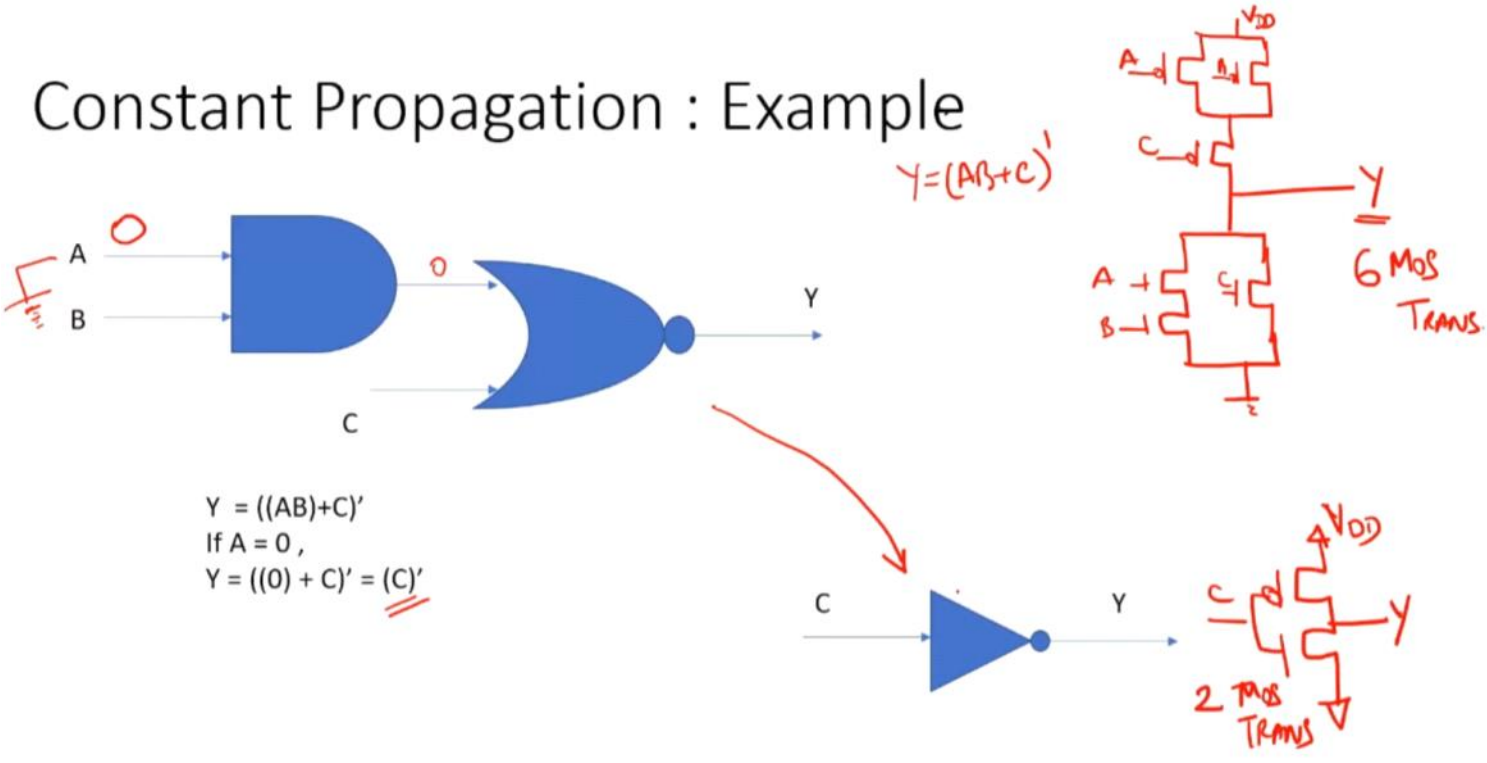

```
22.26. Printing statistics.

=== mult8 ===

   Number of wires:                 2
   Number of wire bits:             9
   Number of public wires:          2
   Number of public wire bits:      9
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 0

22.27. Executing CHECK pass (checking for obvious problems).
checking module mult8..
found and reported 0 problems.
```



Combinational Logic Optimisation

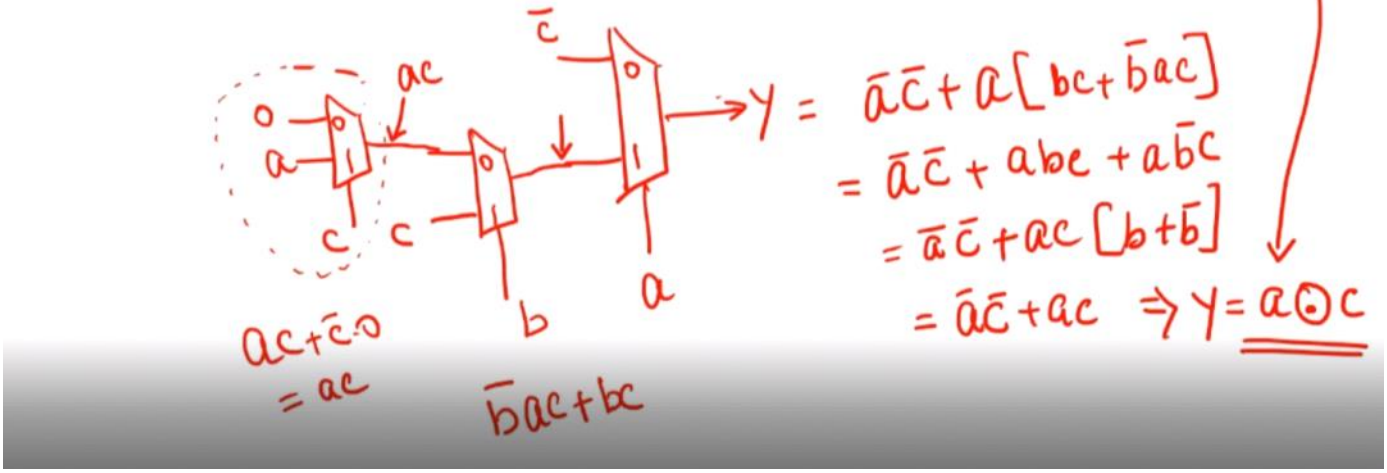
- Squeezing the logic to get the most optimised design
  - Area and Power savings
- Constant Propagation
  - Direct Optimisaton
- Boolean Logic Optimisation

Constant Propagation : Example



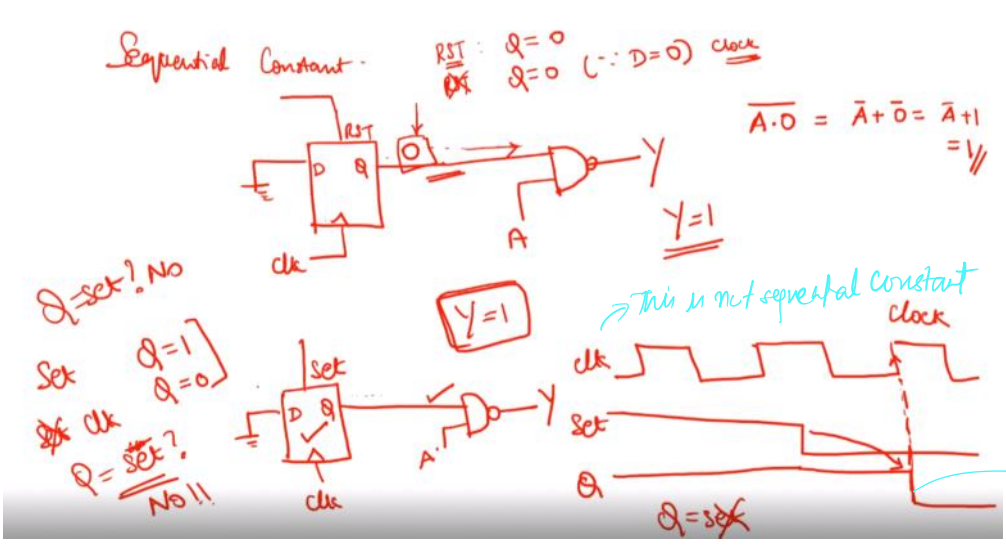
Boolean Logic Optimisation

assign y = a?(b?c:(c?a:0)):(!c)



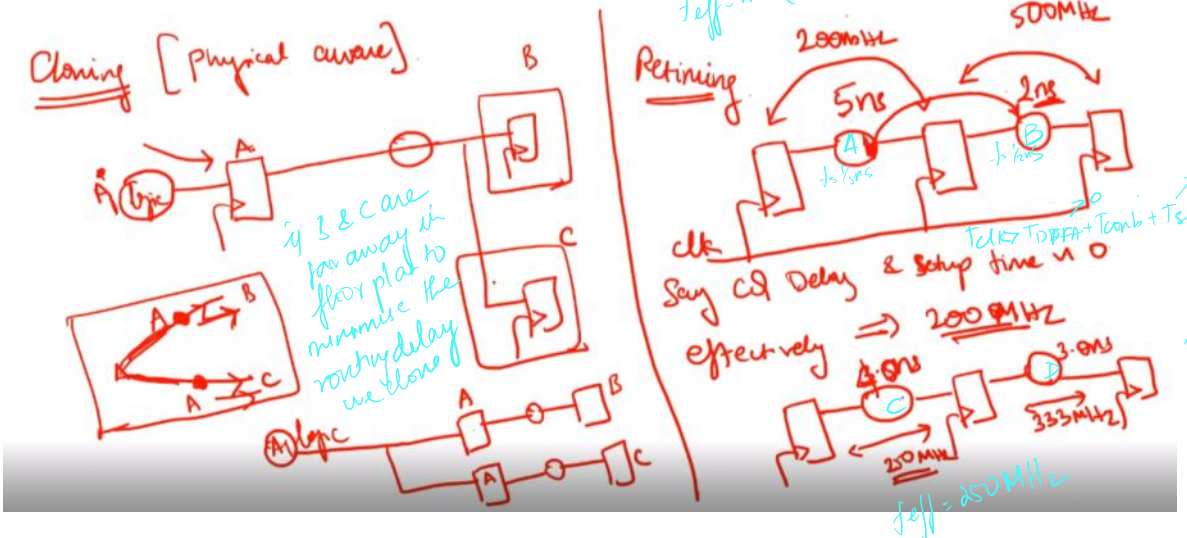
Sequential Logic Optimisations

- Basic
  - Sequential Constant propagation
- Advanced [Not covered as part of Lab]
  - State optimisation
  - Retiming
  - Sequential Logic Cloning (Floor Plan Aware Synthesis)



always @(posedge clk, posedge set)
→ Q changes only after arrival of clk once set goes back to 0 negedge.
so it won't change & we can't say Q≠set

State opt → Optimisation of Unused States (State Reduction ⇒ Refer Morris Mano)



we rearrange/ change the logic A & B, to C & D such that the functionality is taken to B part to set C & D. To improve the frequency

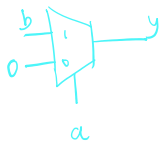
Set C & D. To
improve the frequency
of the ckt

Lab - Combinational Logic

03 June 2021 06:08 AM

File used - optcheck.v

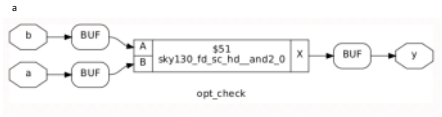
```
verilog_files > opt_check.v
1  module opt_check (input a , input b , output y);
2      assign y = a?b:0;
3  endmodule
4
```



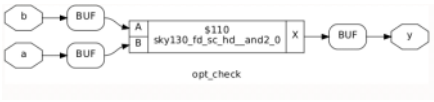
$y = ab + 0 \cdot a$

$y = ab$

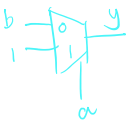
a



B after opt_clean -purge , since there are no unused wires or signals there is no difference




```
verilog_files > opt_check2.v
1  module opt_check2 (input a , input b , output y);
2      assign y = a?1:b;
3  endmodule
4
5
```

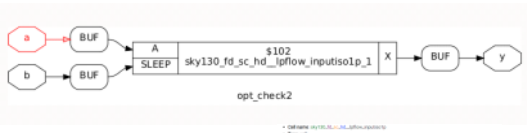


$y = a + \bar{a}b$

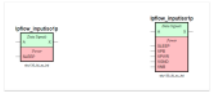
$y = (a + \bar{a})(a + b)$

$y = a + b$

a

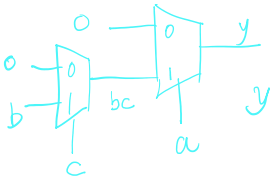


Symbols



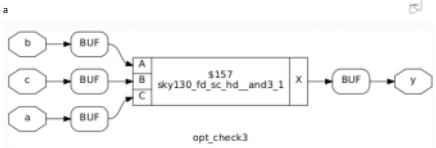
Schematic

```
verilog_files > opt_check3.v
1
2  module opt_check3 (input a , input b, input c , output y);
3      assign y = a?(c?b:0):0;
4  endmodule
5
```

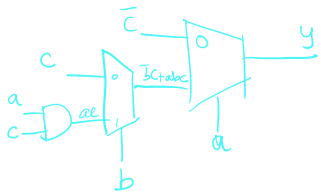


$y = abc$

a




```
verilog_files > opt_check4.v
1  module opt_check4 (input a , input b , input c , output y);
2   assign y = a?(b?(a & c ):c):(!c);
3   endmodule
4
```



$y = \bar{a}\bar{c} + a(\bar{b}c + abc)$

$y = \bar{a}\bar{c} + a\bar{b}c + abc$

$y = \bar{a}\bar{c} + ac(b + \bar{b})$

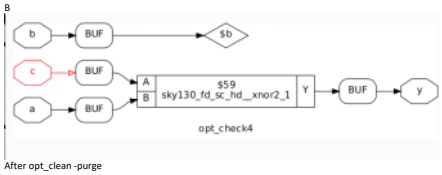
$y = \bar{a}\bar{c} + ac = a \odot c$ (XNOR)

(b is totally unused)

A- before opt_clean -purge

a

B



After opt_clean -purge

After the synthesis step

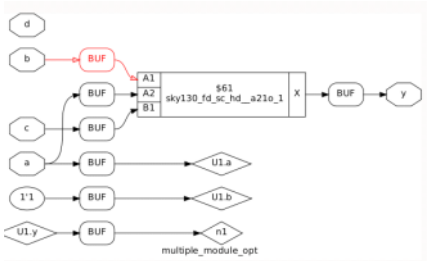
Use the command

Opt_clean -purge

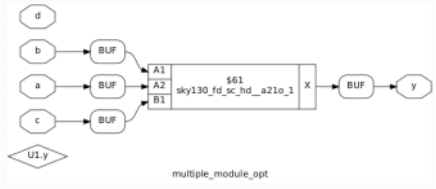
All the unused signals and wires will be removed

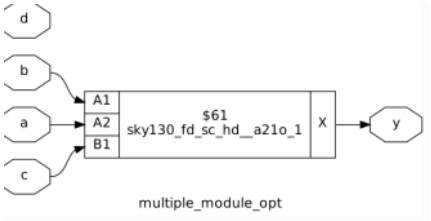
When the optimisation has to be done to the verilog files with multiple modules the files must be first flattened and then opt_clean -purge anf then abc..followed by show

File : multiple_module_opt.v



After using flatten after synth





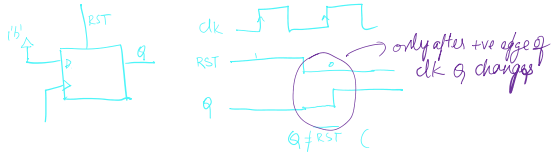
Without usinf flatten

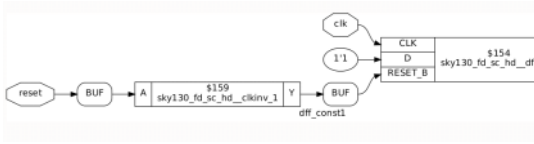
After removing the unused wires

Lab - Sequential optimisation

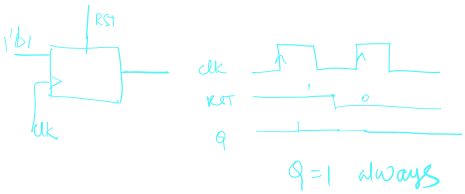
03 June 2021 06:39 AM

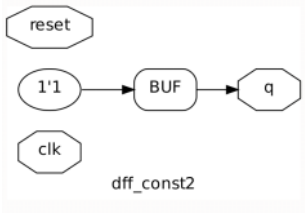
```
verilog_files > dff_const1.v
1  module dff_const1(input clk, input reset, output reg q);
2  always @(posedge clk, posedge reset)
3  begin
4      if(reset)
5          q <= 1'b0;
6      else
7          q <= 1'b1;
8  end
9
10 endmodule
11
```






```
verilog_files > dff_const2.v
1  module dff_const2(input clk, input reset, output reg q);
2  always @(posedge clk, posedge reset)
3  begin
4      if(reset)
5          q <= 1'b1;
6      else
7          q <= 1'b1;
8  end
9
10 endmodule
11
```

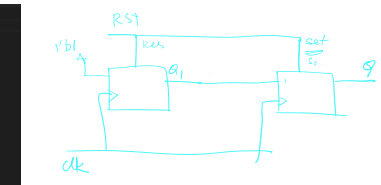


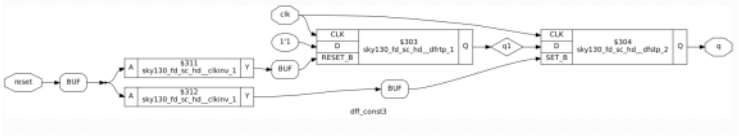



```
=== dff_const2 ===

   Number of wires:                 3
   Number of wire bits:             3
   Number of public wires:          3
   Number of public wire bits:      3
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 0
```

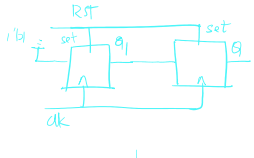
No flop is inferred. Nothing is inferred

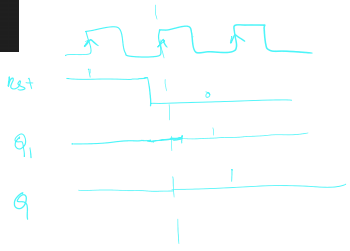
```
verilog_files > dff_const3.v
1  module dff_const3(input clk, input reset, output reg q);
2  reg q1;
3
4  always @(posedge clk, posedge reset)
5  begin
6      if(reset)
7      begin
8          q <= 1'b1;
9          q1 <= 1'b0;
10     end
11     else
12     begin
13         q1 <= 1'b1;
14         q <= q1;
15     end
16 end
17
18 endmodule
```

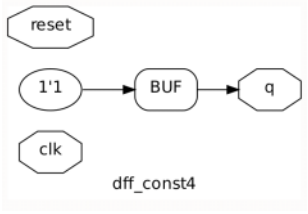





```
verilog_files > dff_const4.v
1  module dff_const4(input clk, input reset, output reg q);
2  reg q1;
3
4  always @(posedge clk, posedge reset)
5  begin
6      if(reset)
7      begin
8          q <= 1'b1;
9          q1 <= 1'b1;
10     end
11     else
12     begin
13         q1 <= 1'b1;
14         q <= q1;
15     end
16 end
17
18 endmodule
```



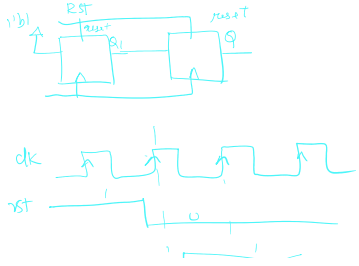


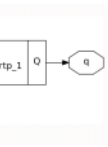


Q is always 1

There is no use of q1, as q1 is declared as a wire of reg type and not a output port

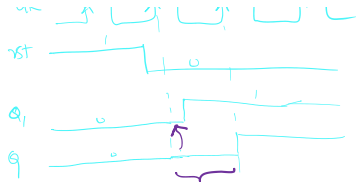
```
verilog_files > dff_const5.v
1  module dff_const5(input clk, input reset, output reg q);
2  reg q1;
3
4  always @(posedge clk, posedge reset)
5  begin
6      if(reset)
7      begin
8          q <= 1'b0;
9          q1 <= 1'b0;
10     end
11     else
12     begin
13         q1 <= 1'b1;
14         q <= q1;
15     end
16 end
17
18 endmodule
```



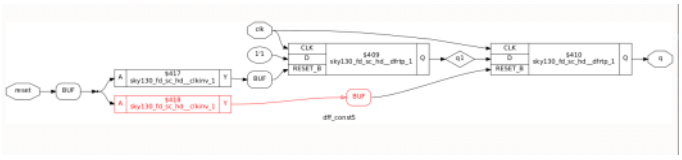



```
=== dff_const1 ===

   Number of wires:                 3
   Number of wire bits:             3
   Number of public wires:          3
   Number of public wire bits:      3
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 1
     $_DFF_PN0_                     1
```

```
        q1 <= 1'b1;
        q <= q1;
    end
endmodule
```



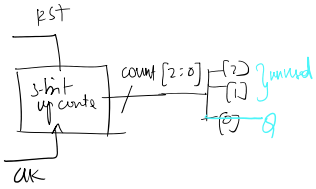
It cannot be reduced



UNUSED OUTPUT OPTIMISATION

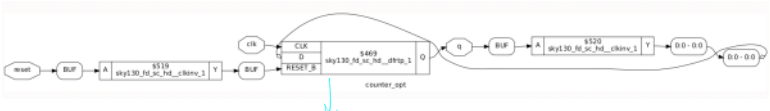
```
module counter_opt (input clk , input reset , output q);
reg [2:0] count;
assign q = count[0];

always @(posedge clk ,posedge reset)
begin
    if(reset)
        count <= 3'b000;
    else
        count <= count + 1;
end

endmodule
```

output is only count[0]



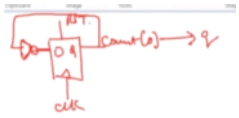
This is an upcounter counts from 0 – 7

∴ count[0] toggles on every clock cycle



only 1 DFF is used (instead of 3)

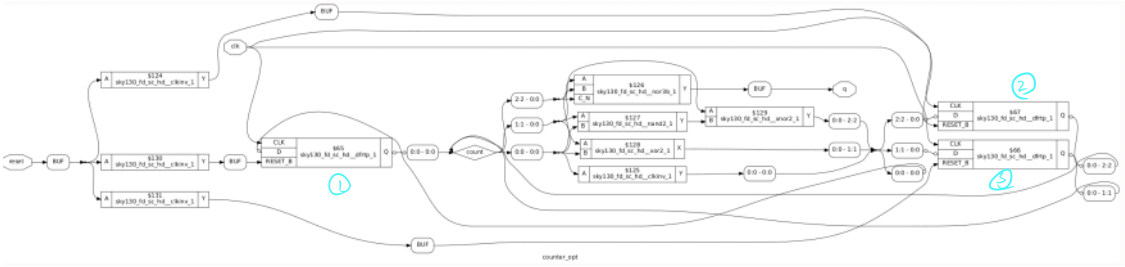
```
=== counter_opt ===

   Number of wires:                 5
   Number of wire bits:             9
   Number of public wires:          4
   Number of public wire bits:      6
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 2
     $_DFF_PP0_                     1
     $_NOT_                         1
```




```
module counter_opt (input clk , input reset , output q);
reg [2:0] count;
assign q = (count[2:0] == 3'b100);

always @(posedge clk ,posedge reset)
begin
    if(reset)
        count <= 3'b000;
    else
        count <= count + 1;
end

endmodule
```

Q becomes 1 whenevr the count 100 is reached

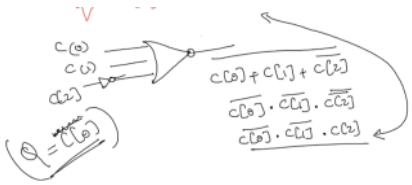
```
=== counter_opt ===

   Number of wires:                 7
   Number of wire bits:            11
   Number of public wires:          4
   Number of public wire bits:      6
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 9
     $_ANDNOT_                      1
     $_AND_                         1
     $_DFF_PP0_                     3
     $_NOT_                         1
     $_OR_                          1
     $_XOR_                         2
```



3 flip flops are used

Q = count[2]. $\overline{count[1]}$. $\overline{count[0]}$

We compare with count using XNOR



Gate Level Simulation & Synthesis-Simulation Mismatch

03 June 2021 07:46 AM

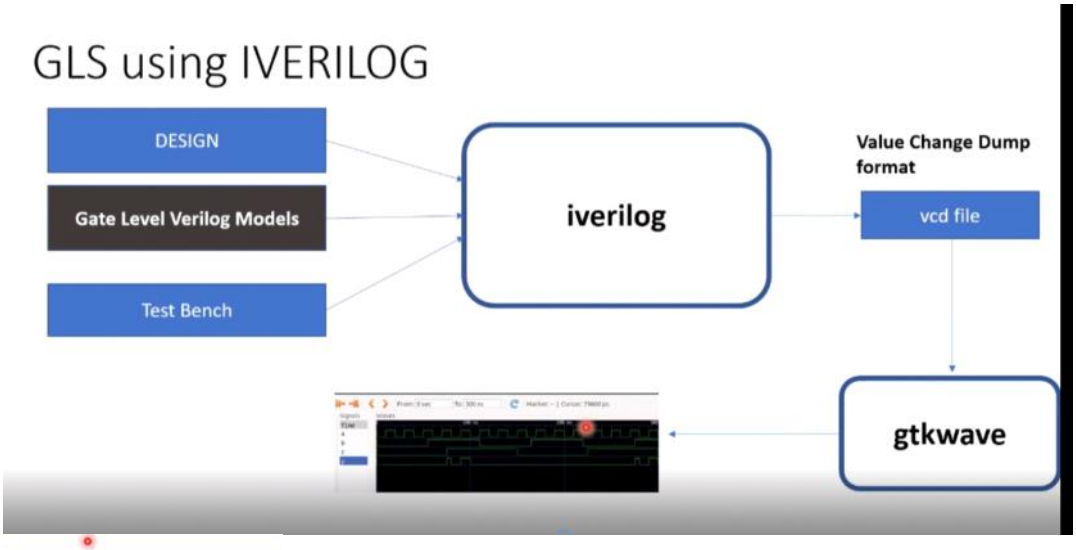
What is GLS

- Running the test bench with Netlist as Design Under Test
- Netlist is logically same as RTL Code.
  - Same Test Bench will align with the Design .

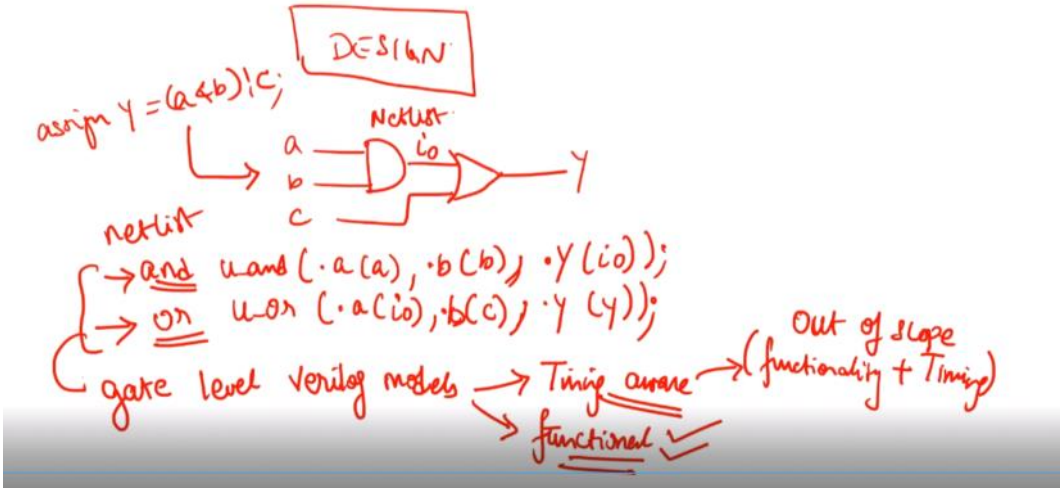
Why GLS

- Verify the logical correctness of design after synthesis
- Ensuring the timing of the design is met.
  - For this GLS needs to be run with delay annotation. (outside the scope of this discussion).

GLS using IVERILOG



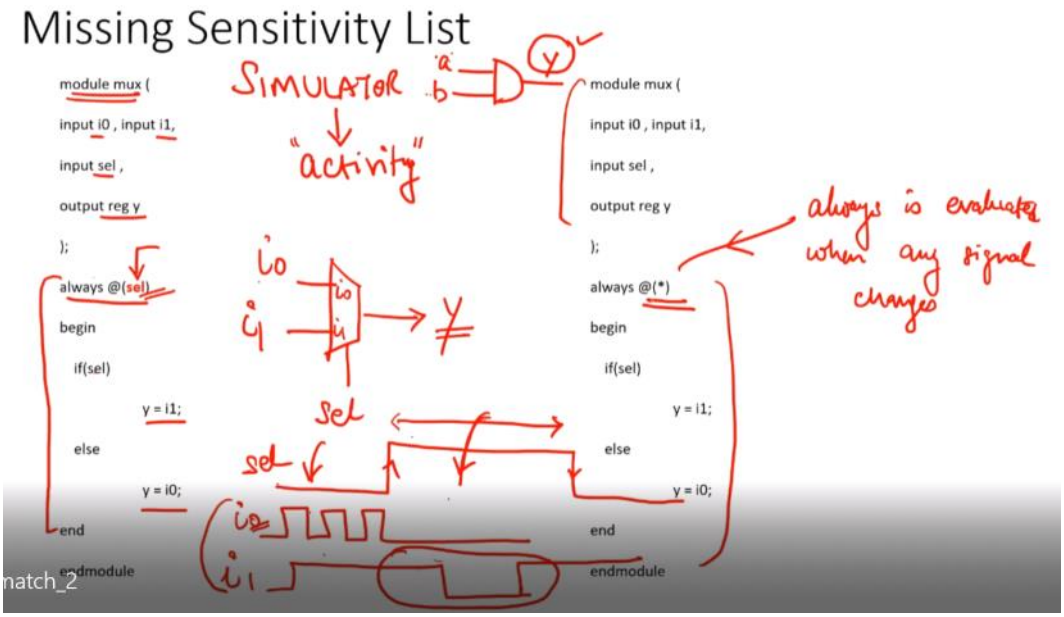
NOTE
If the Gate Level Models are delay annotated , then we can use GLS for timing validation



The netlist will have instantiation of gates. These gates are defined in the verilog_modules .v file. This verilog file can contain the functionality or both the timing an the functionality of the circuit

Generally vcd file (value change dump) only dumps the evaluated values only when there is a change in the vaues. The simulation is evaluated omly whem there is a change in the value

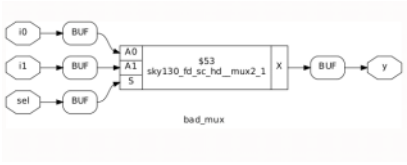
Missing Sensitivity List



Here in the left side code always@(sel) is used so only when the select changes the always block is executed. When the input signals change it is not executed and the output remains contant with the value of the inout signal when the sel was changing

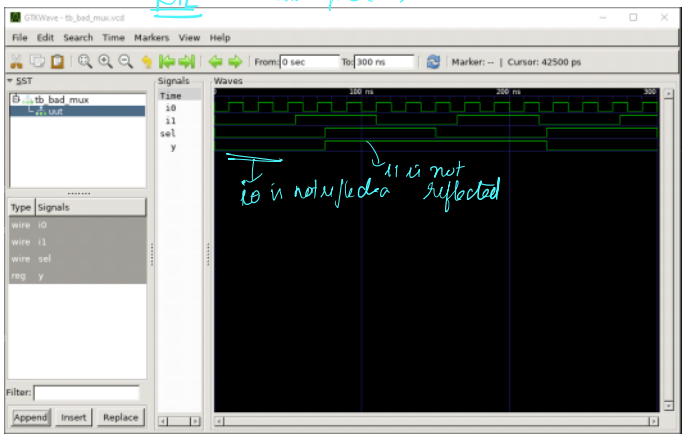
In the right side code always@(*) indicates that whenever any of the signal changes the always block will get executed

```
3   module bad_mux (input i0 , input i1 , input sel , output reg y);
4   always @ (sel)
5   begin
6       if(sel)
7           y <= i1;
8       else
9           y <= i0;
10  end
11  endmodule
12
```



File : bad_mux.v

RTL Simulation



GLS Simulation

Blocking and non blocking assignment - Simulation mismatch example

03 June 2021 08:21 AM

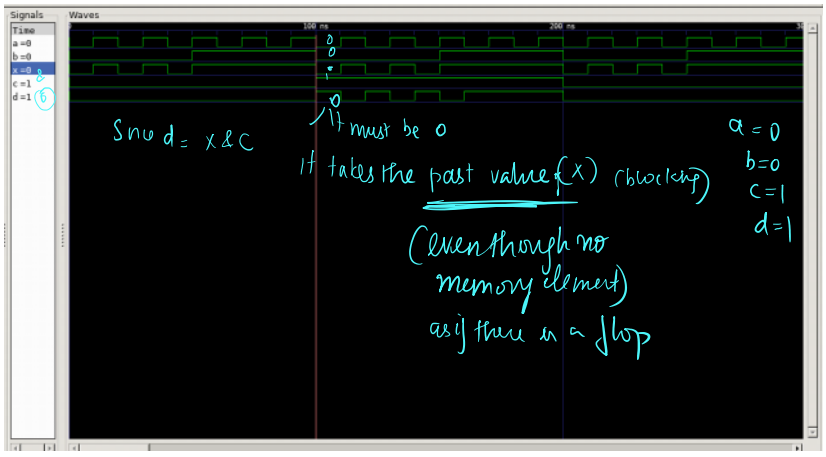
File: blocking-caveat.v

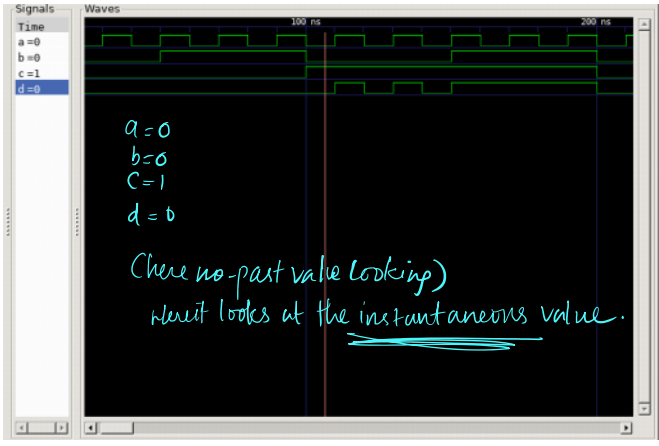
```
verilog_files > blocking_caveat.v
1  module blocking_caveat (input a , input b , input  c, output reg d);
2  reg x;
3  always @ (*)
4  begin
5      d = x & c;
6      x = a | b;
7  end
8  endmodule
9
```

= blocking (one after other)
<= non-blocking (concurrently)

Since x = a | b; is after d = x & c

first d = x & c will get executed & it will produce a flopped o/p.





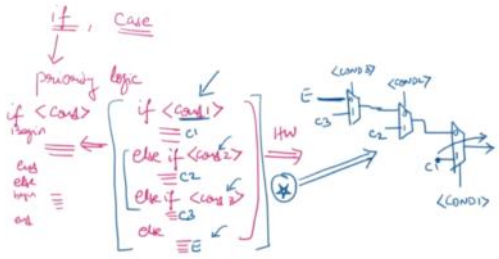
This imaginary flop is caused due to a blocking statement.

∴ synth & sim mismatch due to blocking statement

Very careful while using blocking statement
(Do not use unless required)

If and case constructs

03 June 2021 09:05 AM



A latch is inferred when the output of combinatorial logic has undefined states, that is it must hold its previous value.

Combinatorial logic does not have any flip-flop to hold state therefore the output should always be defined by the inputs.

A short example might be:

```
always @* begin
  if (a == 1'b1) begin
    b = a|y|z;
  end
end
```

What is `b` when `a == 1'b0`. `b` is not being overridden so it would hold its value. How can something hold its value when it does not have the concept of state. You have to introduce state by inferring a latch. This is normally a really bad thing.

You can imply latches and be carefull about the timing etc but inferred latches are nominally from buggy code.

A latch is inferred within a combinatorial block where the net is not assigned to a known value. Assign a net to itself will still infer a latch. Latches can also be inferred by missing signals from a sensitivity list and feedback loops.

The proper way of inferring a intended latch in Verilog/SystemVerilog are:

```
/* Verilog */                  //// /* SystemVerilog */
always @*                      //// always_latch
begin                          //// begin
  if (en) q = d;               ////   if (en) q = d;
end                            //// end
```

Ways latches are accidentally inferred:

- Signal(s) missing for the sensitivity list (this is why `@*` should be used):

```
always @(a or b) // Inferred latch :: "c" missing for the sensitivity list.
begin
  out = a + b + c;
end
```

- Missing Condition:

```
always @*
begin
  case(in[1:0])
    2'b00:  out = 1'b0;
    2'b01:  out = 1'b1;
    2'b10:  out = 1'b0;
    // Inferred latch "out" :: missing condition 2'b11/default
  endcase
end
always @*
begin
  next0 = flop0;
  next1 = flop1;
  // Inferred latch "next2" :: missing initial condition
  next3 = flop3;
  case({a[2:0]})
    3'b001:              next0 = in;
    3'b010:  if(b)       next1 = in;
    3'b100:  if(c)       next2 = in;
    default: if(!b&&!c) next3 = in;
  endcase
end
```

- Feedback Loop:

```
assign out = en ? in : out; // Inferred latch "out" :: feedback to mux
assign a = en ? z : c;
// ... any amount of code between ...
assign z = en ? a : y; // Inferred latch "a" :: feedback chain
```

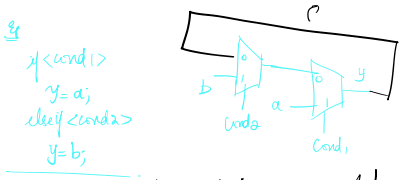
- Feedback loops can traverse through the hierarchy and design.

Inferred latches

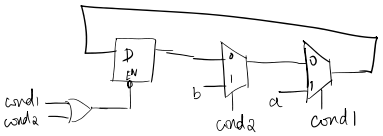
* missing sensitivity list
* net is not assigned a val
* net is assigned to itself
* feedback loops
* careless use of blocking

Eg always @(*)
begin
a = c | d
c = e & f

CAUTION WITH IF - INFERRED LATCHES
(Due to incomplete if statement, bad coding styles)



here the if statement is incomplete
it dosent specify what will happen if both cond1 and cond2 evaluate to false.
So theoritically a combinational loop is formed
To avoid this the tool will infer a latch (Inferred latch)



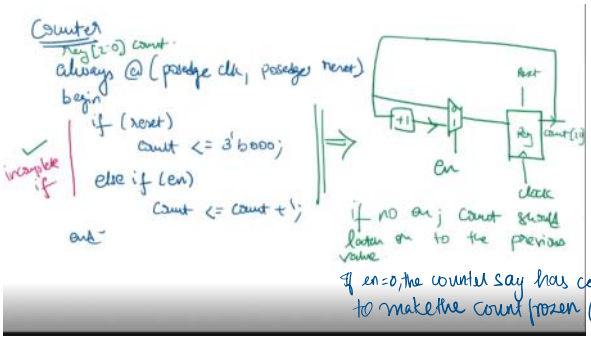
when both cond1 and cond2 fail (0), then the previous value of y stays since it is latched

This is an inferred latch due to an incomplete if statement
Don't leave the if statement incomplete unless intended (say counter etc)

How to mitigate the risk of unintended latches:

- Make intended latches simple and identifiable:
  - Put intended latches in their own always blocks with as little combinatorial logic as possible; ideally put the latches' combinatorial logic in its own separate always block. Be as explicit and identify intended latches. Use comments, labels, and if possible use the SystemVerilog `always_latch`.
- All combinatorial logic blocks need to be defined with `always @*` or SystemVerilog's `always_comb`.
- Make sure all variables assigned in a combinatorial logic blocks have an initial or default assignment.
  - `case` statements should have a `default` condition.
  - `if` statements should have a corresponding `else`.
  - When the combinatorial logic blocks is assigning many variables, giving each variable an initial value at the start of the block (before any `case` or `if`).
- Know where the inputs are coming from and where the outputs are going to.
  - The inputs of combinatorial logic should be flops **or** the outputs combinatorial logic should be flops.
- Do code reviews, use linting tools and logical-equivalency-checking tools.
  - Code review requires the reviewer(s) to know where latches could hide.
  - Using SystemVerilog's `always_comb` can help identify inferred latches with linting and logical-equivalency-checking tools.

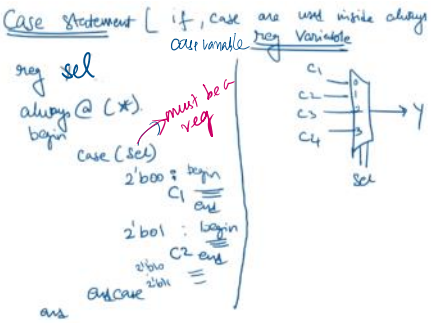
Worst case scenario, put all logic inside synchronous blocks. All inferred latches become inferred flip-flops. This is usually a bad idea because it can unnecessarily increases the gate count, create more routing, and impact timing.



If en=0, the counter say has counted to 4 before en=0, so 0 latches to count to make the count frozen (else count will go to 0)

CASE STATEMENT

Case statement [if, case are used inside always block]



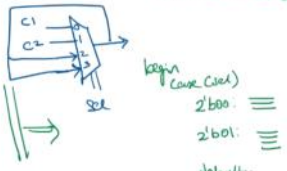
statement

has to take
revenue value

Caveats with case

① Incomplete case ⇒ Infers latches → to avoid → CODE CASE WITH DEFAULT

```
reg [1:0] sel
always @ (*)
begin
case (sel)
2'b00 : c1 =
2'b01 : c2 =
endcase
end
```

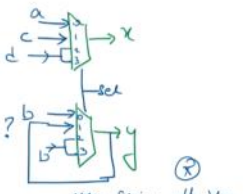



```
begin
case (sel)
2'b00 =
2'b01 =
default =
endcase
end
```

Case - Caveat ②

partial assignments in case

```
reg [1:0] sel;
reg x, y;
always @ (*)
begin
case (sel)
2'b00 : begin x = a;
        y = b;
        end
2'b01 : begin x = c; ← y = ?? ☹
        end
default : begin x = d;
          y = b;
          end
endcase
```



Note:- Assign all the outputs in all the segments of case

This leads to unpredictable outputs ⇒ overlapping case

```
Priority in if
else if
else if
else
only one portion can execute

case (sel)
2'b00 =
2'b01 =
2'b10 ✓ =
2'b1? ✓ =
endcase
```

(10) (or) (11)

unpredictable outputs (*)

(*) not have overlapping case (*)

* In if-else if - else statement there is a priority and only one block will get executed and after execution it will come outside the if-else if-else block automatically

* In case, all the cases will be executed sequentially from top to bottom unless explicitly said to come out of the case in between with a break.

For loop and For generate

04 June 2021 12:17 PM

looping

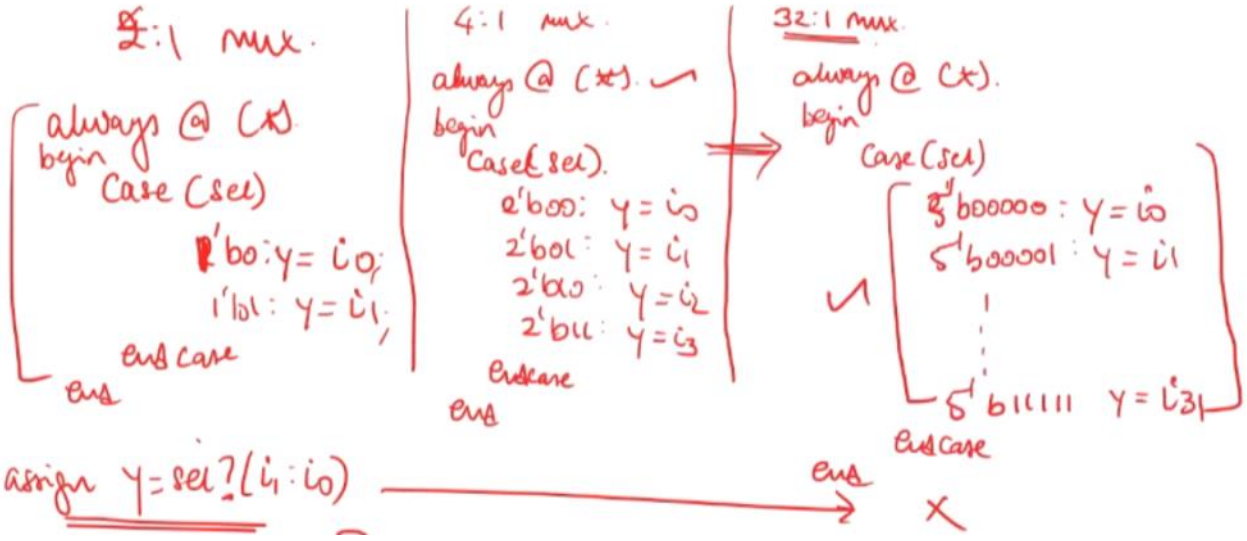
for loop

* used only within always
* evaluate expressions
* cannot instantiate other gates / modules

generate for loop

* used only outside always
* should not be used inside always
* instantiate hardware (for.g: instantiate an and gate 500 times)

Application



It is difficult to write a 32:1 mux

```
integer i
always @(*)
begin
    for (i=0; i<32; i=i+1) begin
        if (i == sel)
            y = inp[i];
    end
end
```

assumption:- inp[31:0] 32 bits

32x1 mux

256 → 1 mux

DEMUX 1 x 8 Demux

assumption op_bus[7:0] ⇒ op
input ⇒ ip
sel [2:0]

```
integer i;
always @(*)
begin
    op_bus[7:0] = 8'b0;
    for (i=0; i<8; i=i+1) begin
        if (i == sel)
            op_bus[i] = input;
```

evaluation

op_bus[7] = 0
(6) = 0
(5) = 0
input
(0) = 0

Very wide Mux / Demux

for statement is very handy

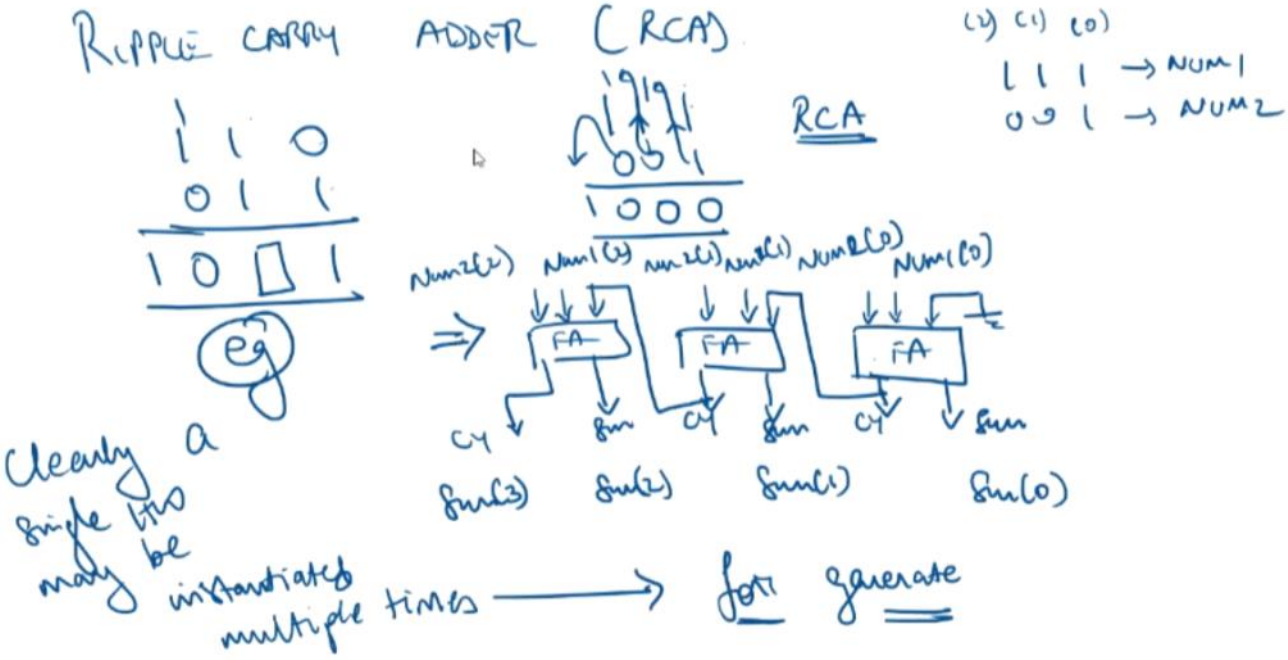
for generate :-

```
and u_and1(.a(  ),.b(  ),.y(  ));
and u_and2(.a(  ),.b(  ),.y(  ));
```

for-generate ⇒ replicating the HW [outside always block]

```
genvar i;
generate
    for (i=0; i<8; i=i+1) begin
        and u_and(.a(in1[i]), .b(in2[i]), .y(y[i]));
    end
endgenerate
```

Why we need to replicate hardware
Consider a n bit ripple carry adder



Clearly a single FA may be instantiated multiple times → for generate

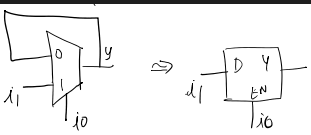
Lab - if and case

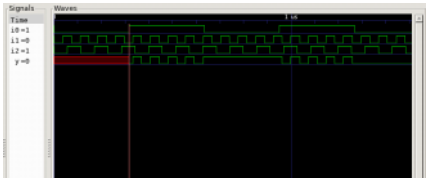
04 June 2021 01:04 PM

Incomplete if Statement

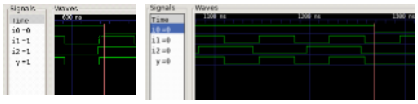
verilog_files > incomp_if.v

```
1  module incomp_if (input i0 , input i1 , input i2 , output reg y);
2  always @ (*)
3  begin
4      if(i0)
5          y <= i1;
6  end
7  endmodule
8
```





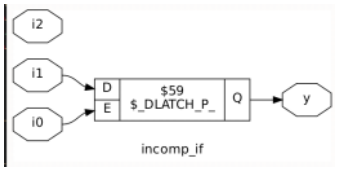
Initially i0 = 0, so y is unknown (X)
then i0 = 1, y = i1, then
i0 becomes 0, then y takes
the previous value (the
instantaneous value of i1,
just before i0 = 0, (which is 1))
or 0



y = 1 y = 0

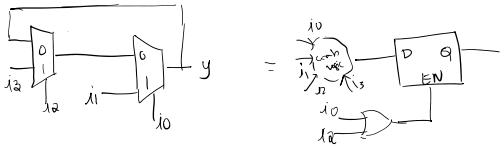
After synth -top incomp_if

```
3.26. Printing statistics.

=== incomp_if ===

   Number of wires:                 4
   Number of wire bits:             4
   Number of public wires:          4
   Number of public wire bits:      4
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 1
     $_DLATCH_P_                    1
```

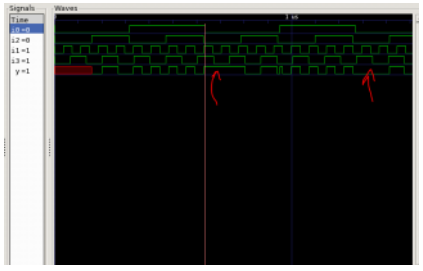


We can see that a latch has been synthesised, even though we tried to code a mux

```
2  module incomp_if2 (input i0 , input i1 , input i2 , input i3, output reg y);
3  always @ (*)
4  begin
5      if(i0)
6          y <= i1;
7      else if (i2)
8          y <= i3;
9
10 end
11 endmodule
12
```

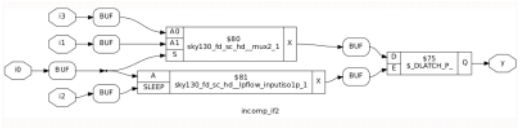


If both i0 & i2 are 0, then
the output takes the previous val



After Synthesis

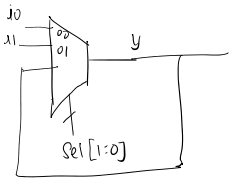
```
3.26. Printing statistics.

=== incomp_if2 ===

   Number of wires:                 7
   Number of wire bits:             7
   Number of public wires:          5
   Number of public wire bits:      5
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 3
     $_DLATCH_P_                    1
     $_MUX_                         1
     $_OR_                          1
```

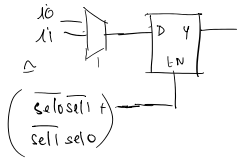


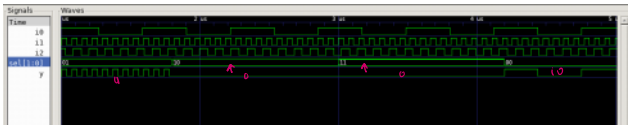
Incomplete Case statement

verilog_files > incomp_case.v

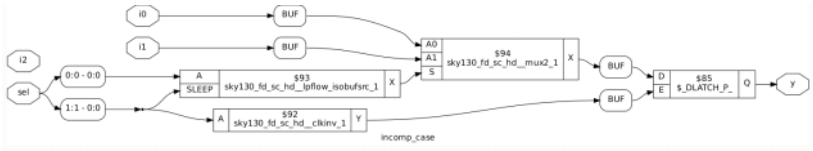
```
1
2  module incomp_case (input i0 , input i1 , input i2 , input [1:0] sel, output reg y);
3  always @ (*)
4  begin
5      case(sel)
6          2'b00 : y = i0;
7          2'b01 : y = i1;
8      endcase
9  end
10 endmodule
11
```



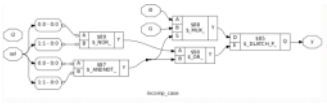





```
3.26. Printing statistics.

=== incomp_case ===

   Number of wires:                  9
   Number of wire bits:             10
   Number of public wires:           5
   Number of public wire bits:       6
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                  5
     $_ANDNOT_                       1
     $_DLATCH_P_                     1
     $_MUX_                          1
     $_NOR_                          1
     $_OR_                           1
```

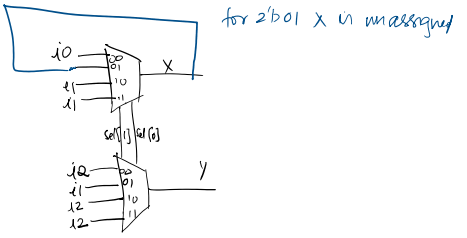


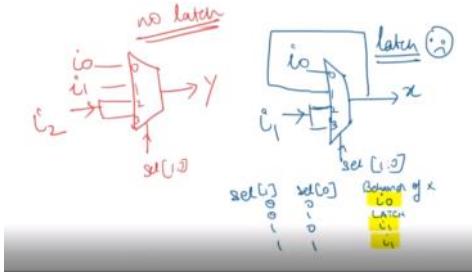
Before mapping to sky130 libraries

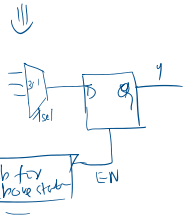


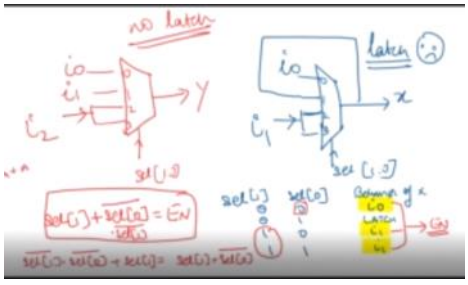
Partial case Assignment

```
verilog_files > partial_case_assign.v
1  module partial_case_assign (input i0 , input i1 , input i2 , input [1:0] sel, output reg y , output reg x);
2  always @ (*)
3  begin
4      case(sel)
5          2'b00 : begin
6              y = i0;
7              x = 12;
8              end
9          2'b01 : y = i1;
10         default : begin
11                   x = i1;
12                 y = i2;
13                 end
14     endcase
15 end
16 endmodule
17
```

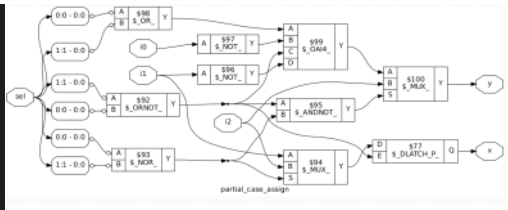


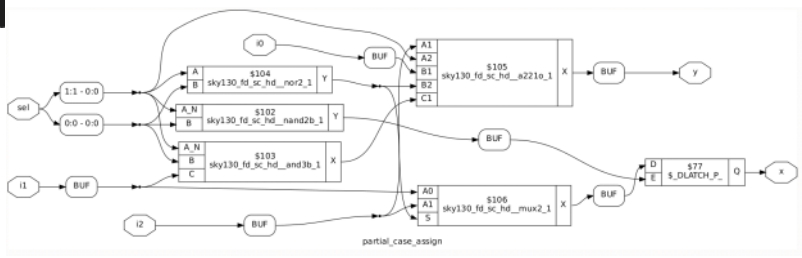







```
3.26. Printing statistics.

=== partial_case_assign ===

   Number of wires:                 14
   Number of wire bits:             15
   Number of public wires:           6
   Number of public wire bits:       7
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                 10
     $_ANDNOT_                       1
     $_DLATCH_P_                     1
     $_MUX_                          2
     $_NOR_                          1
     $_NOT_                          2
     $_OAI4_                         1
     $_ORNOT_                        1
     $_OR_                           1
```

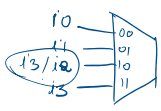


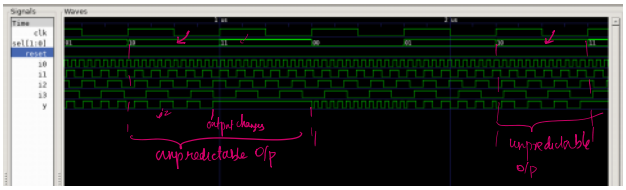


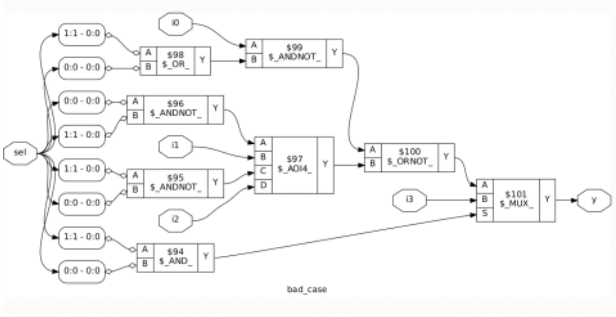
OVERLAPPING CASE (Unpredictable outputs) (NOT latching)

Multiple Assignments

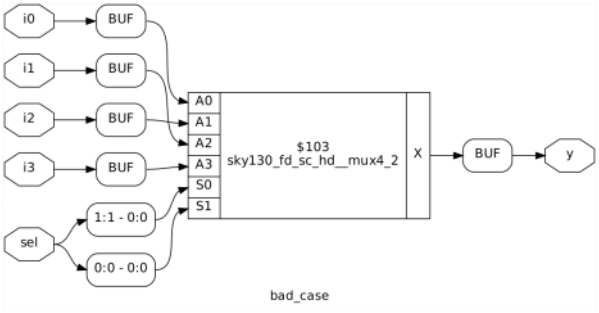
```
verilog_files > bad_case.v
1  module bad_case (input i0 , input i1, input i2, input i3 , input [1:0] sel, output reg y);
2  always @(*)
3  begin
4      case(sel)
5          2'b00: y = i0;
6          2'b01: y = i1;
7          2'b10: y = i2;
8          2'b11: y = i3;
9          //2'b11: y = i3;
10     endcase
11 end
12
13 endmodule
```







After abc - liberty



However there are no latches inferred, such case assignments lead to unpredictable behavior in the RTL simulation. After synthesis it dosent matter here(occassionally depending on the synthesizer)

In GLS there is no problem

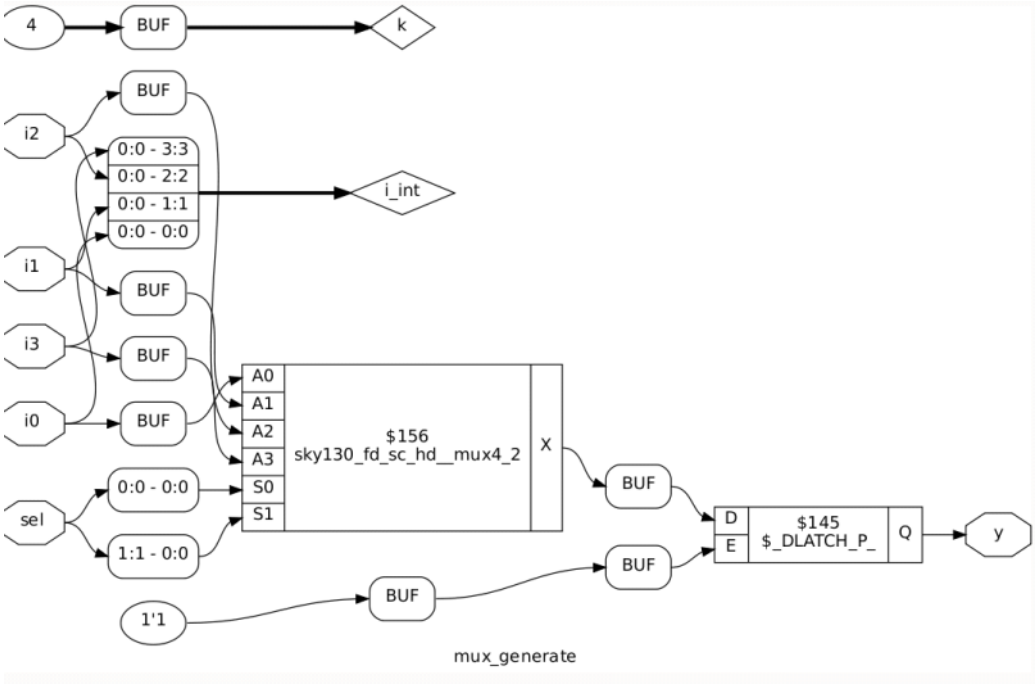
Lab - for & generate for

04 June 2021 02:40 PM

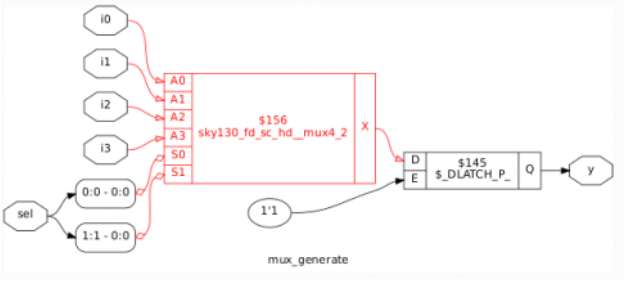
verilog_files > mux_generate.v

```
1  module mux_generate (input i0 , input i1, input i2 , input i3 , input [1:0] sel  , output reg y);
2  wire [3:0] i_int;
3  assign i_int = {i3,i2,i1,i0};
4  integer k;
5  always @ (*)
6  begin
7  for(k = 0; k < 4; k=k+1) begin
8      if(k == sel)
9          y = i_int[k];
10 end
11 end
12 endmodule
13
```

5x1 mux
Initially i_int is initialised to a 4 bit value i3i2i1i0 by contatinating them
This way all the discrete inputs are concatinated and then assigned to a vector. This way in the for loop for the kth selection the K th bit of the i_int can be assigned to the output



Opt_clean -purge



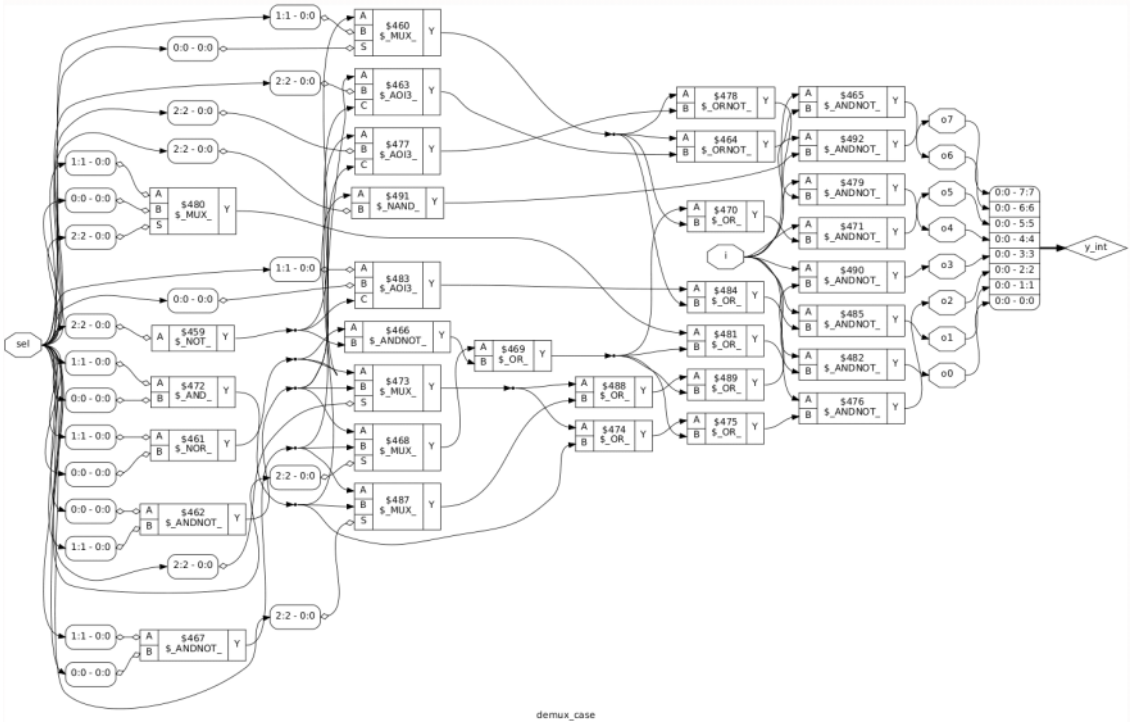
?? Why dlatch

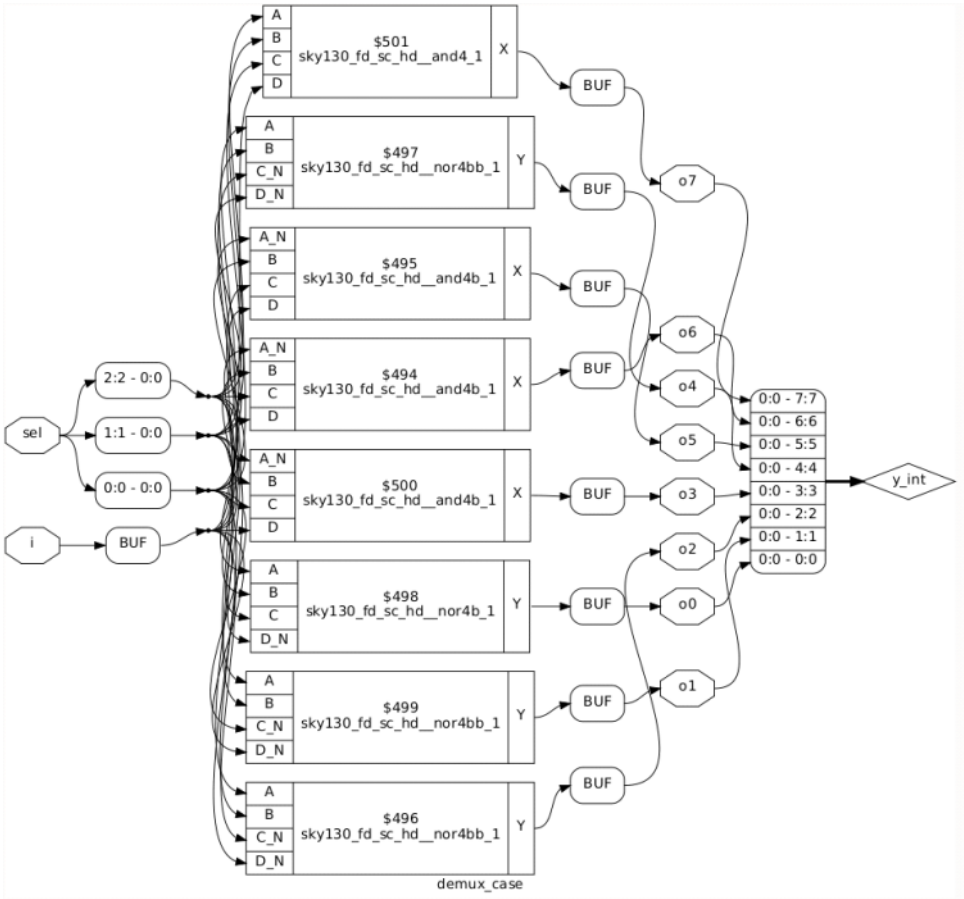
DEMUX

verilog_files > demux_case.v

```v
module demux_case (output o0 , output o1, output o2 , output o3, output o4, output o5, output o6 , output o7 , input [2:0] sel  , input i);
reg [7:0]y_int;
assign {o7,o6,o5,o4,o3,o2,o1,o0} = y_int;
integer k;
always @ (*)
begin
y_int = 8'b0;
    case(sel)
        3'b000 : y_int[0] = i;
        3'b001 : y_int[1] = i;
        3'b010 : y_int[2] = i;
        3'b011 : y_int[3] = i;
        3'b100 : y_int[4] = i;
        3'b101 : y_int[5] = i;
        3'b110 : y_int[6] = i;
        3'b111 : y_int[7] = i;
    endcase

end
endmodule
```

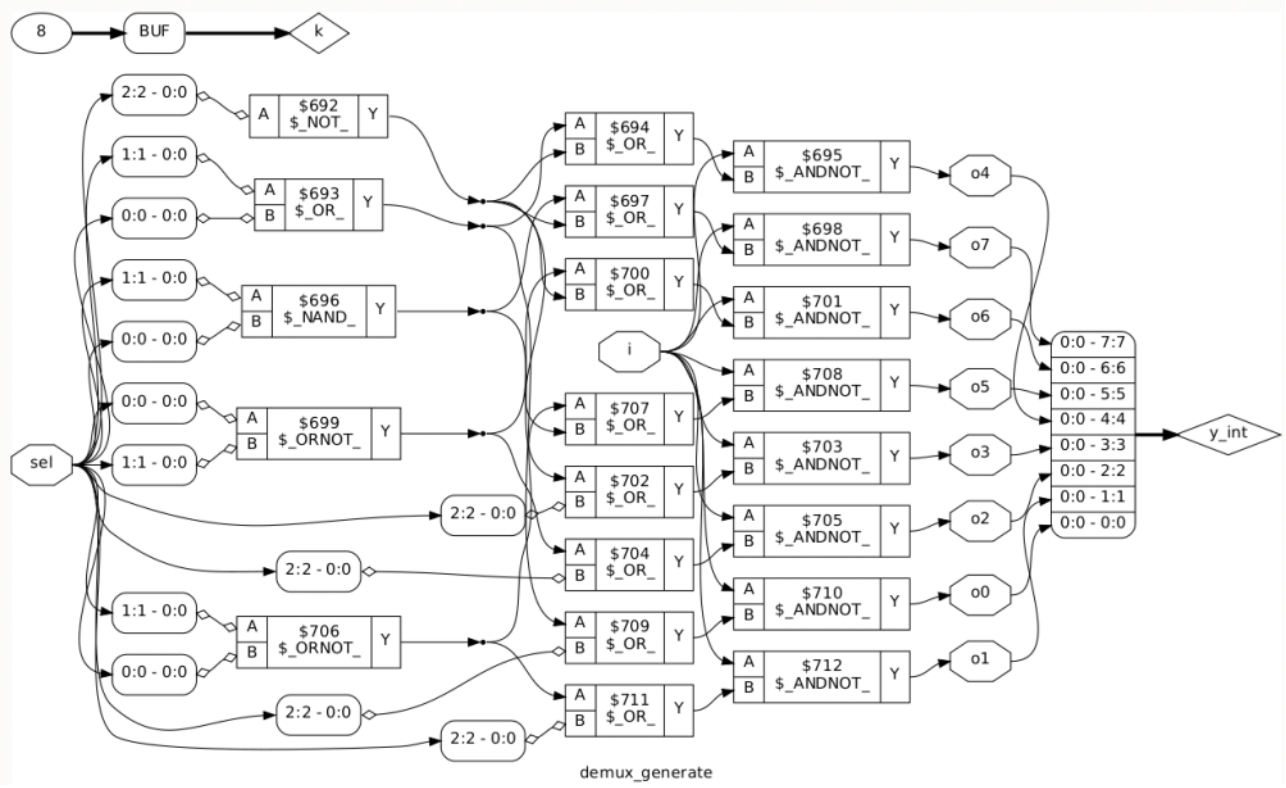
Syth

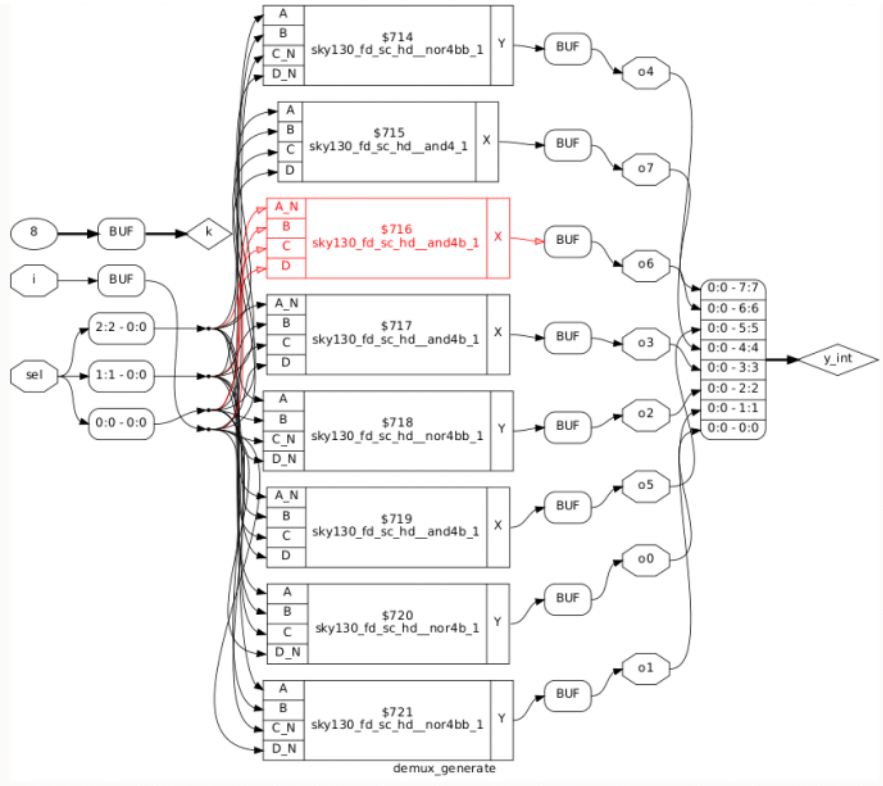


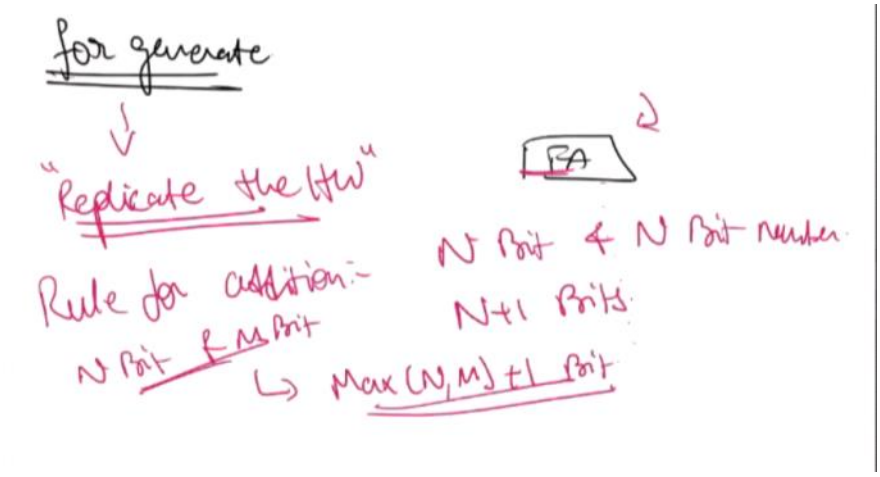


verilog_files > demux_generate.v

```
1
2  module demux_generate (output o0 , output o1, output o2 , output o3, output o4, output o5, output o6 , output o7 , input [2:0] sel  , input i);
3  reg [7:0]y_int;
4  assign {o7,o6,o5,o4,o3,o2,o1,o0} = y_int;
5  integer k;
6  always @ (*)
7  begin
8  y_int = 8'b0;
9  for(k = 0; k < 8; k++) begin
10     if(k == sel)
11         y_int[k] = i;
12 end
13 end
14 endmodule
```


8
BUF
k
2:2 - 0:0
1:1 - 0:0
0:0 - 0:0
sel
$692 $_NOT_
$693 $_OR_
$696 $_NAND_
$699 $_ORNOT_
$706 $_ORNOT_
$694 $_OR_
$697 $_OR_
$700 $_OR_
$707 $_OR_
$702 $_OR_
$704 $_OR_
$709 $_OR_
$711 $_OR_
i
$695 $_ANDNOT_
$698 $_ANDNOT_
$701 $_ANDNOT_
$708 $_ANDNOT_
$703 $_ANDNOT_
$705 $_ANDNOT_
$710 $_ANDNOT_
$712 $_ANDNOT_
o4
o7
o6
o5
o3
o2
o0
o1
0:0 - 7:7
0:0 - 6:6
0:0 - 5:5
0:0 - 4:4
0:0 - 3:3
0:0 - 2:2
0:0 - 1:1
0:0 - 0:0
y_int
demux_generate

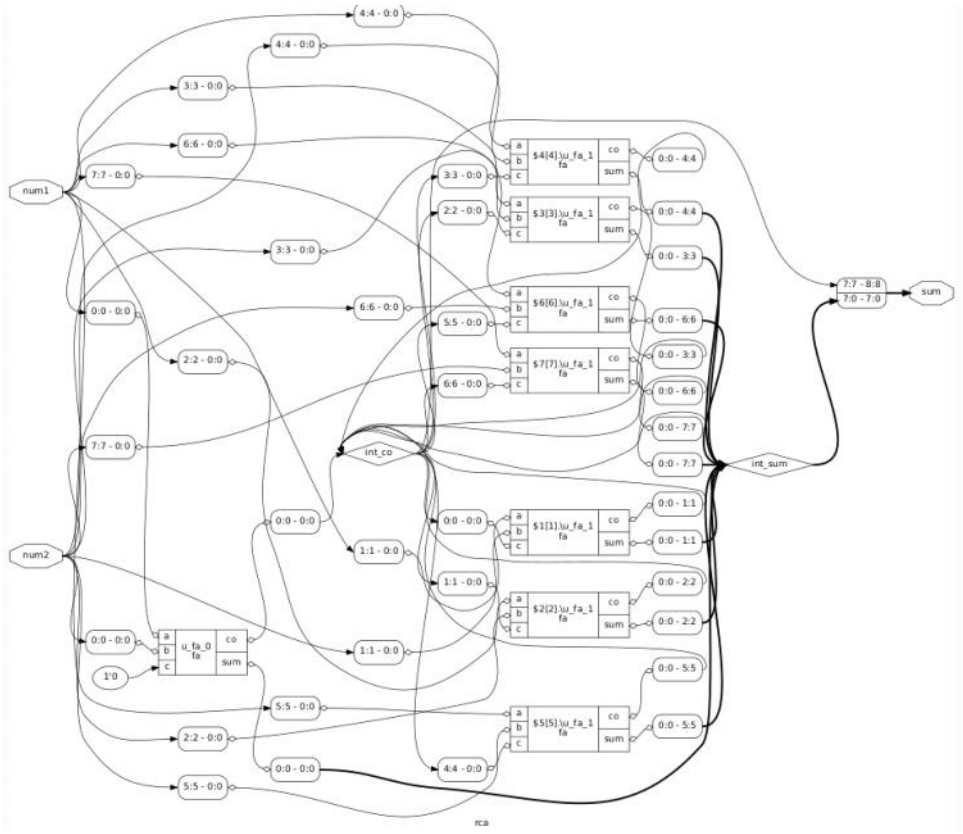

8
BUF
k
i
sel
2:2 - 0:0
1:1 - 0:0
0:0 - 0:0
$714 sky130_fd_sc_hd__nor4bb_1
$715 sky130_fd_sc_hd__and4_1
$716 sky130_fd_sc_hd__and4b_1
$717 sky130_fd_sc_hd__and4b_1
$718 sky130_fd_sc_hd__nor4bb_1
$719 sky130_fd_sc_hd__and4b_1
$720 sky130_fd_sc_hd__nor4b_1
$721 sky130_fd_sc_hd__nor4bb_1
o4
o7
o6
o3
o2
o5
o0
o1
0:0 - 7:7
0:0 - 6:6
0:0 - 5:5
0:0 - 4:4
0:0 - 3:3
0:0 - 2:2
0:0 - 1:1
0:0 - 0:0
y_int
demux_generate



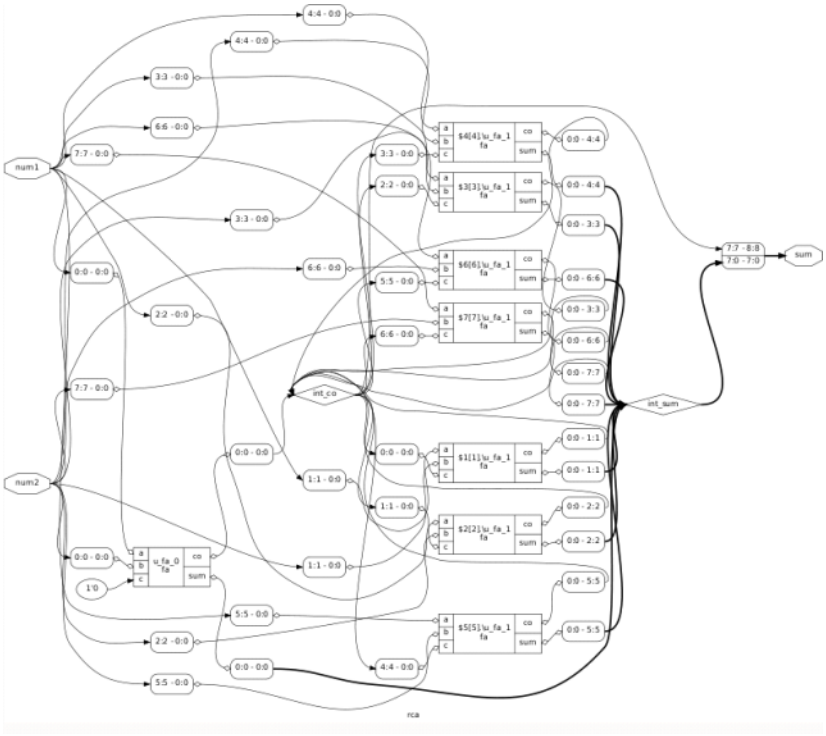
verilog_files > rca.v

```
1   module rca (input [7:0] num1 , input [7:0] num2 , output [8:0] sum);
2   wire [7:0] int_sum;
3   wire [7:0]int_co;
4
5   genvar i;
6   generate
7       for (i = 1 ; i < 8; i=i+1) begin
8           fa u_fa_1 (.a(num1[i]),.b(num2[i]),.c(int_co[i-1]),.co(int_co[i]),.sum(int_sum[i]));
9       end
10
11  endgenerate
12  fa u_fa_0 (.a(num1[0]),.b(num2[0]),.c(1'b0),.co(int_co[0]),.sum(int_sum[0]));
13
14
15  assign sum[7:0] = int_sum;
16  assign sum[8] = int_co[7];
17  endmodule
18
19
```

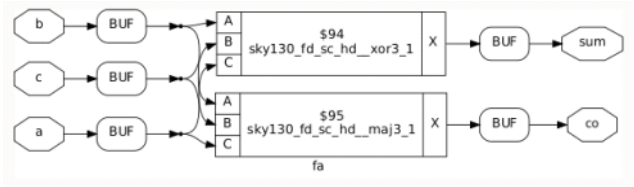
Show rca



After sky maping show rca


num1
num2
sum
inst_co
inst_sum
rca

Show fa


b
c
a
BUF
A
B
C
$94
sky130_fd_sc_hd__xor3_1
$95
sky130_fd_sc_hd__maj3_1
X
sum
co
fa