5

(4) 6 2

primary ip & op.

__/__/____

27/4/22 Physical Design Workshop → VSD (1) installation snap.

Day 1

Theory

4: iverilog and testbench.

simulator → to check RTL design's spec

Design Testbench → setup to apply stimulus to the design.

↓

test vectors.

Simulator working → change in ip ⇒ op is evaluated

output of a simulator → vcd file

↳ value change dump format.

↓

gtkwave → to view waveform.

Lab: (1) VLSI → git done from kunal's git page. → new directory.

Lab: (2) verilog_files → to load a file into iverilog

design file & tb_<designname> file.

command- iverilog good_mux.v tb_good_mux.V

[./a.out] gives vcd file

↳ to execute.

UUT → unit under test

cmd: gtkwave tb_good_mux.vcd

drag & drop inputs & ops.

zoom fit

to look for transitions → select signal

→| symbol and arrow to move.

$dumpfile

$dumpvars.

look into files → using gvim

to open multiple files → use -o.

Intro to yosys

synthesizer → RTL to netlist

tool used.

DESIGN .lib → [yosys] → netlist file

↓

representation of design in the form of standard cells.

commands: read_verilog → to read the design
read_liberty → to read the library files
write_verilog → to obtain netlist

How to verify synthesis:

NETLIST
TB → iverilog → vcd → gtkwave → observe output

TB ✓ same as RTL tb.

Should be same as op observed during RTL simulation.

Logic Synthesis → design is converted into gates.

RTL → Gate Level.

* What is RTL Design

• lib → collection of logical modules
↳ all basic gate — different flavours of the same gate
eg: take AND
2-ip, 3-ip, 4-ip
slow, medium, fast

• lib ↓ not exhaustive but rich enough to implement design

Why different flavours of the same gate?

$T_{clk} > T_{CQ\_A} + T_{COMB1} + T_{SETUP\_B}$

T_{clk} ↓ min ⇓ f_{max} ↑ clk.

T_{SETUP_B} ↓ train example.

T_{clk}

ip → op should be one clock cycle.

Are fast cells sufficient? or why do we need slow cells?

$T_{HOLD\_B} < T_{CQ\_A} + T_{COMB1}$.

Requirement: → ip of B should change after this window.

edge of clk, after which data shouldn't change.

FFA launch, FFB capture

B shouldn't capture here. (for the same cycle)

Sol: to guarantee a minimum delay. ⇓

Fast cells to meet setup criteria
Slow cells to meet hold criteria
} Thus variety of cells are used.

Fast cells vs slow cells → digital logic circuit

Load of DLC → capacitance

fast charging and discharging of C → less delay.

↓

transistors capable of sourcing more current.

↓

Wide transistors

Wide → less delay → Area & Power ↑

Narrow → more delay → Area & Power ↓

Selection of cells → 'constraints'

↳ guidance to pick correct set of cells.

Synthesis illustration.

Lab:

Introduction to yosys synthesizer.

commands: yosys → to invoke synthesizer.

read_liberty -lib ../my_lib/<filename>

↓

has significance

read_verilog good_mux.v

synth -top good_mux

↳ module name.

abc -liberty ../

↳ path to lib file

RTL to netlist command.

ABC report -x -x

↳ note signals & cells.

show

↳ graphical version of logic realized.

flop

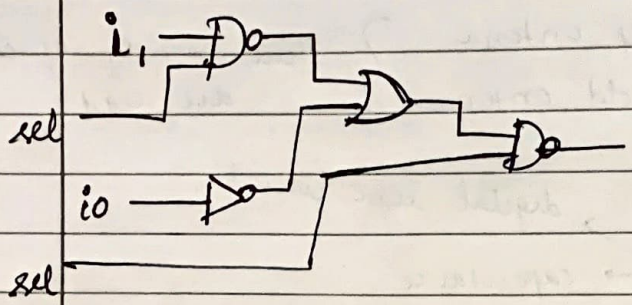
↓

another word to be execute

4 cm
3 lat ort.

02 ai → or and

Gate level interpretation



to generate netlist file

cmd: write_verilog good_mux_netlist.v

modify the switch?

cmd: write_verilog -noattr good_mux_netlist.v

Day 2

Lab: (4) Intro to timing .lib file

→ open library file in gvim.

Name of the library file → sky130_

tt → typical

025C → 25°C

Any lib file → PVT impt.

Process, voltage, temperature.

Process → variations due to fabrication. (bread toasting example)

semiconductors are sensitive to temp. conditions

wrt .lib file

- 1V80 → 1.8 volts.
- technology
- delay model → NLT.
- unit of time, voltage, current, power, res. & cap.
- operating conditions

↳ contains definitions of different cells.

↓

features → leakage power mentioned for all possible ip combs. and delay.

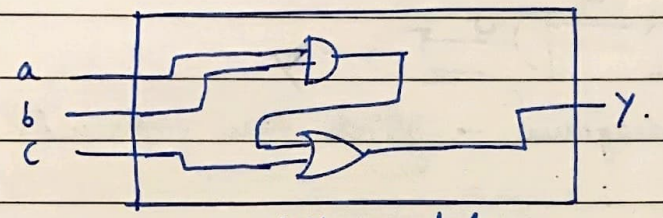
cells can be better understood by viewing its equivalent verilog model

- ★ area
- ★ power port info
- ★ details wrt each pin

__/__/____

Lab: (5) Hierarchial and Flat Synthesis

cmd: gvim multiple_modules.v



multiple module

* launch yosys
* read liberty file
* read verilog file
* synth -top multiple_modules.v
* generate netlist (abc)
* show

output: instances shown and not the gates.
↳ hierarchical design.

* write netlist ⇒ hierarchy preserved.
* view gvim

OR gate realised in terms of NAND. → classical action of DeMorgan's.

Why?

cmd: flatten

cmd: write_verilog multiple_modules_flat.v

* view the netlist

NAND implementation is preferred over NOR
↓
stacking PMOS is bad.
↓
because more width reqd. for good logical effort.

Task: given multiple modules, how to synthesize only submodule?

change in synth -top command. (mention submodule name.)

Why do we need to do submodule synthesis?

(1) If same module is replicated n times in a time, it can be synthesised once and the results can be replicated & put together.

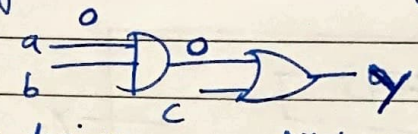
(2) Divide and conquer method
↳ for massive designs.

Lab 6 files: dff_async_rst dff_syncres.
dff_asyncres
dff_asyncres_syncres

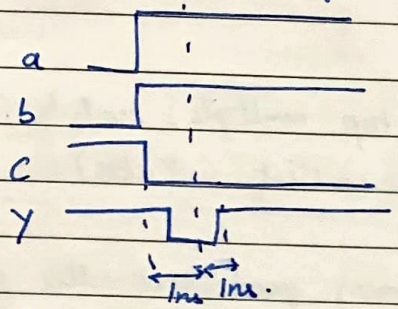
Flops and Flop coding style.

Propagation delay in comb. circuits → glitches @ output.

Example:



using timing diagram → glitch was explained.



chain of combinational circuits → output(s) will continuously be glitchy.

Sol: D FF.

Flops provide shielding. → Main purpose of using flops in digital circuits.

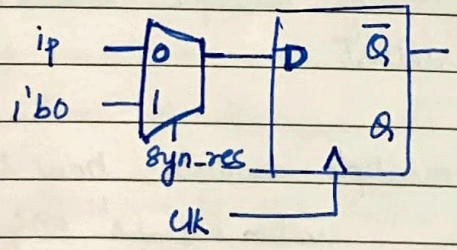
Reset / Set pin → to initialise a flop

Synchronous — Asynchronous.

async → irrespective of clk.

sync ⇒ D pin of the flop.

depends on clock.



same flop can have sync & async reset.

set and reset together can cause race conditions.

Lab ⑥: Synthesis and simulation of flops:

* simulation of all types of flops given. → snaps of op.

sync reset given preference

CO2 standard cells & its are kept separately @ liberty

dfflibmap → after this 'abc' step.

__/__/____

Lab (7): Interesting optimisations. two modules → mult_2.v, mult_8.v.

mult_2 → truth table observation.

a x 2 ⇒ a appended with 0 → in binary

Take example of a x 9:

↳ a x [8+1] ⇒ a x 8 + a x 1.

6 bits: a followed by three zeros. ⇒ y will be [a][a] (3 bits, 3 bits).

Day 3

Combinational and Sequential Optimisations

Introduction to optimisations → Logic

Combinational → Why? → to squeeze the logic to get the most optimised design.

↳ Area & Power.

Techniques:

* Constant Propagation → Direct. → eg = Y = (AB+C)' where A always 0.
* Boolean Logic Optimisation

↓ given expr. → $\bar{a}\bar{c} + ac$.

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

Simplified expr Y = a ⊙ c

Sequential

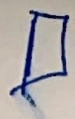
* Basic → sequential constant propagation
* Advanced

State optimisation → condensation

Retiming → splitting logic equally for improving max op. frequency.

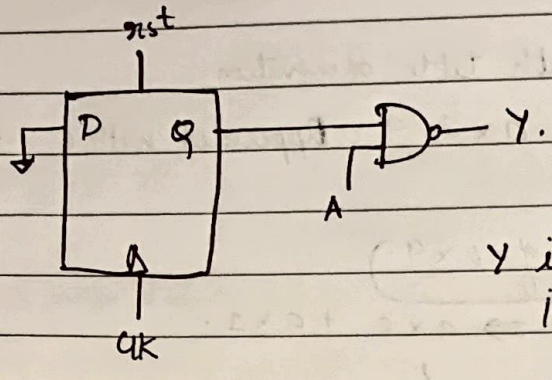
Sequential logic Cloning (Floor Plan Aware Synthesis) → A → B, A → C → to minimize routing delay (huge -ve slack prerequisite)

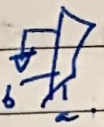
'de-asserted'.



Sequential constant propagation

example:





$\bar{a} \cdot 0 \neq ab$

Y is always 1 irrespective of D, clk, A and rst.

Same example with set option

↳ constant propagation cannot be applied.

Lab (08): (6 as per the course)

Combinational Logic Optimisations part 1.

files → all opt files.

cmd: opt_clean -purge

opt_check.

opt_check2

opt_check3

for multiple modules: first flatten and then optimise.

Lab (09): (7 as per the course)

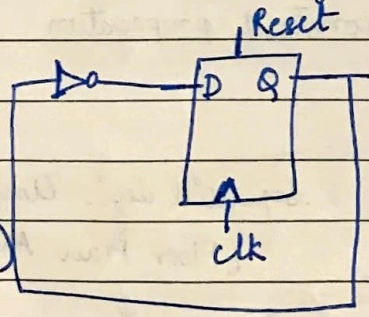
Sequential Logic Optimisation

files → dff_const1.

const 4 & 5 → task.

Sequential optimisation of unused outputs.

counter_opt.v → synthesized module.



modify counter and observe op.

q = (count[2:0] == 3'b100)

Day 4

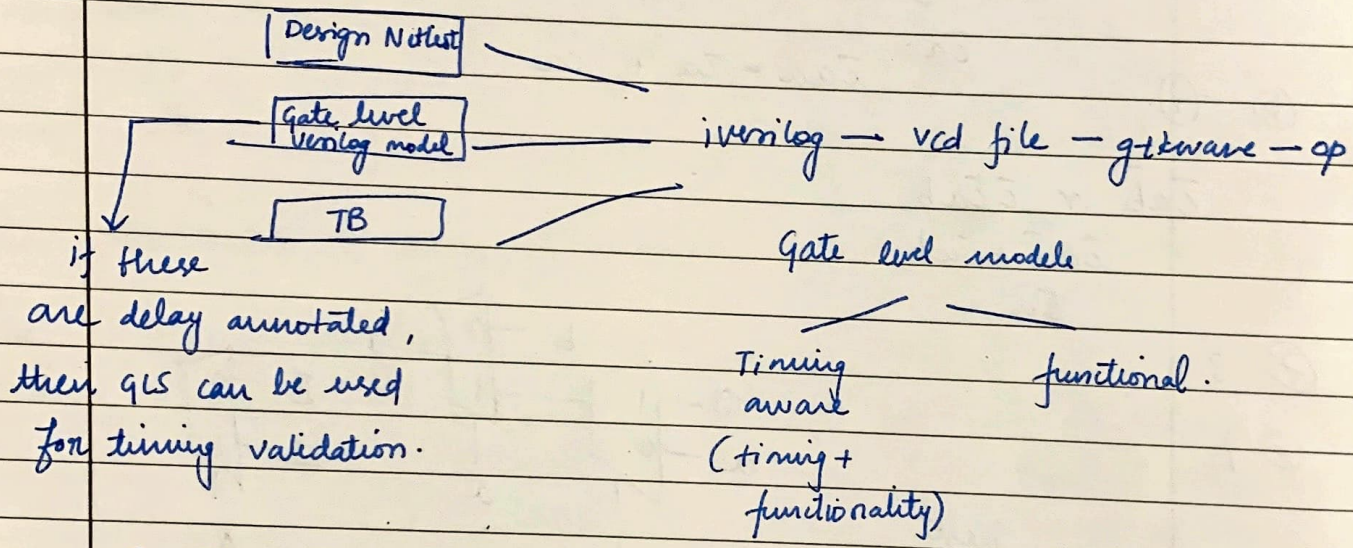
Gate level Simulation (GLS)

- running TB with netlist as DUT.
  - ↓
  - logically same as RTL code

Why ?

- to verify logical correctness
- to ensure timing (run with delay annotation) to be

GLS using iverilog



Synthesis – simulation mismatch

- ★ Missing sensitivity list
- ★ Blocking vs non-blocking assignments
- ★ Non standard verilog coding.

comes into picture only inside the 'always' block.

= → blocking → sequential execution

<= → non blocking → concurrent execution.

↳ RHS evaluated first.

Caveats with blocking statements

↓

SR example.

* bad_mux
* good_mux.

__/__/____

Lab (10): (8 as per course)

cmd to read verilog models.

iverilog ../my_lib/[redacted]verilog_models/primitives.v
sky130_fd_sc_hd.v
and then netlist. and test bench.

Lab (11): (9 as per course)

Mismatch due to blocking statements.
file blocking_caveat.v

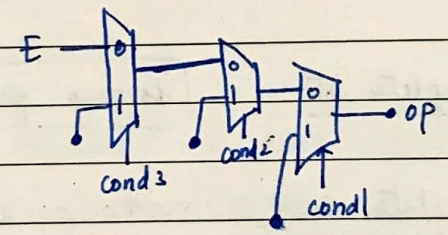
Day 5

If case constructs

if → to create priority logic

```
if <cond1>
  :
elsif <cond2>
  :
else if <cond3>
  :
else
  :
  :
```

HW



Danger / caution → inferred latches.

↳ Bad coding style

↳ effect of incomplete if statement.

incomplete if → latch → required @ times
ex: counter.

no inferred latches in combinational circuits.

variable to which the value is assigned inside a if or case statement should be a 'reg'.

③ overlapping case. ② partial assignments. ① incomplete case

Caveats with case statement

① → With default, we avoid inferred latches.

② assume a situation where case controls two output signals.
partial assignment → inferred latch.
moral: assign all the outputs in all the segments of case.

priority clear.

if
else if
else if
else

case:
no defined priority
bad coding might lead to two or more case segments being executed.
↓
unpredictable outputs.

Never have overlapping cases.

Lab (12): Incomplete IF — incomp_files

Lab (13): Incomplete case — incomp-case.v, comp_case.v, bad_case.v, partial_case_assign.v

Redundancy theorem.
overlapping case → synthesis - simulation mismatch.

Looping constructs
- for loop → inside always → to evaluate expressions.
- generate for loop → outside always → instantiating hardware.

example: 32:1 mux

assuming input is a 32 element array

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

__/__/____

for - generate → to replicate hardware

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

// creates 8 instances of and gates.

Example: Ripple Carry Adder. (RCA)

if generate can also be used ↓ outside the always block.

Lab (14): mux_generate.v rca.v

demux_case.v fa.v

demux_generate.v

01 sel[1] & sel[0]

Rule for addition

N-Bit + N-Bit number → N+1 bit output

N-Bit + M Bit → max(N, M) +1 bit output.