Verilog HDL

1. Bitwise operators
2. Parameter
3. Different data-types reg, wire. (i/p, o/p by implicitly wire)
4. Comparission operator
5. $ finish
6. Blocking, Non-blocking, continuous assignment
7. Vector [MSB:LSB]
8. Arithmatic operators.
9. Concardination operator.
10. always @ block
11. *missing sencitivity list
    else
    default -> missing case condition
12. if, case, for
13. *nested if else
14. $display, $monitor
15. posedge, negedge
16. Order in blocking assignment.
17. Replication operator.
18. Ternary operator
19. Function & task (static / automatic)
20. Stratified event queue

21. primitive (built-in)

22. while, forever

26-09-2024

Hardware modeling using verilog

- Indranil Senegupta

Verilog → hardware discription language

Behavioral, structural

Area, speed, energy consumption

1961 - First IC

Technology CMOS, FinFET (14nm)
(22nm)

Design flow Specification, Synthesis, Simulation, Layout, Testing & anal

Use of CAD tool

Behavioral → RTL → gate → transistor.

Design Idea → Behavioral Design

Data path Design
netlist (Blocks connected) => Structural

Logic Design
gate flipflop, standard cell

Physical Design
regular polygon.

Manufacturing

1. Introduction about lecture
2. Design Representation.
3. Getting started with verilog.
4. VLSI Design Styles.

30/09/2024

Book- Digital logic design using verilog.

Verilog HDL

- Vivado
- Questasim

- Vivav

→ IEEE ~~1995~~ 1364
1995 2001, 2005

1983-84 Gate automation
→ Phil morby

Why HDL:

- in C++, designing a counter in it or any high level language.
  - not hardware counter
    - no description abt hardware
      - i/o ports
      - no direction control
      - no trigger (raising edge pos edge
        falling edge Neg edge)

- to

1) Model digital design
2) direction control input
   output
   input output in out.
3) Edge sensitive pos edge
   neg edge
4) Time control generate signals after delay.
   in a sequence.
5) Switch level modeling.

※ | _in - input
_out - output | industrial standard to avoid confusion since delay.

Design flow

Specifications

↓

Architecture & micro architecture design

↓

RTL Design

↓

Gate level design

↓

Transistor level design

~~RTN~~
RTL - Register transfer level.

→ Synthesizable construct..
Testbench
→ non-synthesizable construct.

Verilog (or) System verilog. IEEE 1800-2009

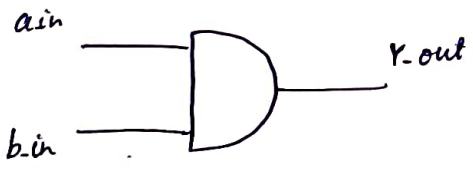
Verilog HDL

Synthesizable & non-synthesizable constructs.

↳ RTL Design.

↳ Behavioral
used in verification
stimulus generation

intension → not to infer logic.



Syntax Verilog HDL explained.

Dont use red for signal

:X: → red is for dontcare

```
module <module_name> (
                    input <port_nos>,
                    ...           ,
                    ...           ,
                    output <port_nams>
                    );

// functionality of block
...
...
...

endmodule
```

Combinational logic

assign

↳ all are assigned concurrently (not line by line as in any compiled (or) interpreted language)

Forced level simulation

→ force clock on each i/p

complexity increase 2^n → no of i/ps

2

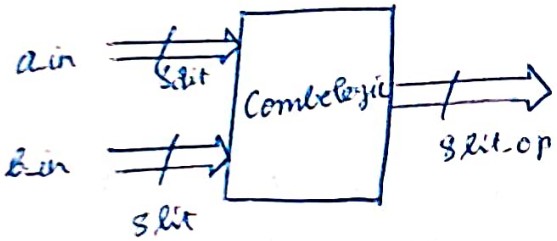
01-10-2024

Vector n bit i/o

Bitwise operator & their application in RTL Design

Almost inputs & outputs are vector (ie) multibit inputs (or) o/p as a bus. Logical operators are not suitable, bitwise operator to be used, following are bitwise operator

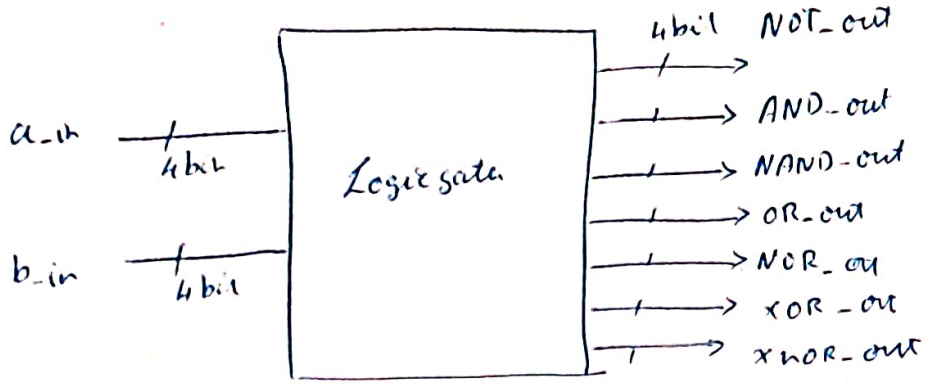
& - Bitwise and
| - Bitwise or
~ - Bitwise not
^ - Bitwise exor
~^ } Bitwise exnor
^~ }




```
module logic_gates(
    input [7:0]  a_in,
    input [7:0]  b_in,
    output [7:0] y_out
);

assign  y_out = a_in & b_in

end module.
```

Exercise 2

RTL Design



parameter <parameter-n> (name usually capital letter)

```
module and_bit #(parameter DATA_width=4
( input [DATA_width-1:0] a_in,
  input [DATA_width-1:0] b_in,
  output [DATA_width-1:0] Y_out);

assign Y_out = a_in & b_in;

endmodule.
```

Exercise 3

RTL Design. use parameter DATA_WIDTH = 4

Test Bench ⇒ uses non synthesizable construct

↳ tb_nor

(.DATA_WIDTH (_))

```
// test bench use non synthesizable constr
module tb_and_bit();
input = output  reg [3:0] a_in;
are             reg [3:0] b_in;
wire <-         wire [3:0] Y_out;

and_bit #(.DATA_WIDTH (4)) DUT
(
  .a_in (a_in),
  .b_in (b_in);
  .Y_out (Y_out));
```

→ execute only once procedural block used in test bench to generate a stimulus.

```
initial
begin
  a_in = 0; b_in = 0;
#10 a_in = 4'b1010; b_in = 4'b1010;
#20 a_in = 4'b1100; b_in = 4'b1011;
  #200 finish.
end
```

* a_in, b_in need to be assign specific value within initial block these should be of type reg, not wire

Error: (procedural assignment to non-register not permitted)

reg/integer/time/genvar

Proper industrial standards & requirement.

-> usage of naming convention _in input
_out output.
_io input/output

-> improve readability of design & testbench (ie) use comments wherever required.

-> use synthesizable construct in RTL design file.

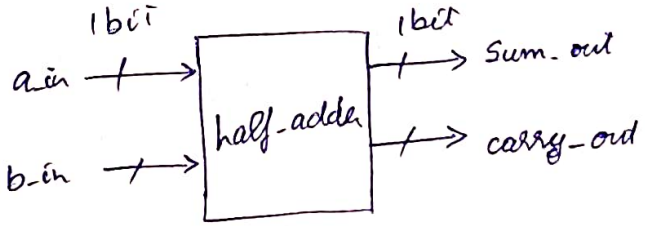
-> use non-synthesizable constructs in testbenches

-> to check for functional correctness of design use testbench not force level simulation.

Evening session

Exercise 6

Complete RTL design of half adder



Exercise 7

tb_half_adder

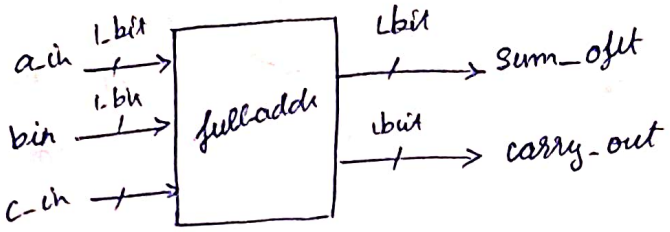
Complete a testbench to check functional correctness of exercise

※ Relaunch → to see waveform.

Exercise 8

Use mon no of half adder & OR gate to have 1-bit full adder

instance of half adder.

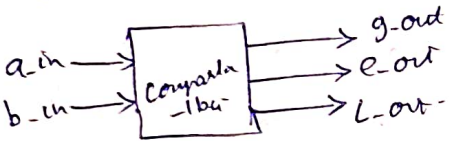


Exercise_9

Compile a test bench of exercise 8.

Exercise_10

RTL Design 1-bit comparator



Exercise_11

Testbench exercise 10

$finish

> < == >= <= } comparision operator.

03-10-2024

Exercise 12

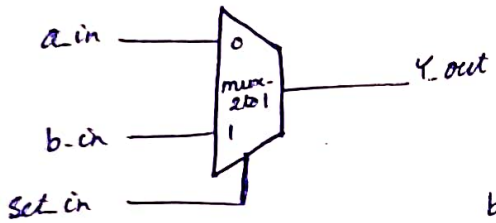
Check order of continious assignit → Neither blocking nor non-blocking

→ Blocking (BA) =
→ Non-blocking. (NBA) <=] used for assignment in procedural bloc

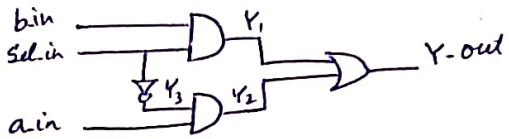
Procedural blocks are
- initial, always.

Continious assign → in module

Consider RTL design of 2:1 MUX



Sel_in	Y_out
0	a_in
1	b_in




```
module mux2to1(
    input a_in, b_in, sel_in,
    output Y_out);
wire Y1, Y2, Y3;
assign Y1 = Sel_in & b_in;
assign Y2 = ~(Y3) & a_in;
assign Y3 = ! sel_in;
assign Y_out = Y1 | Y2;

end module
```

→ here Y_3 is used in previous line (ie) before assign to Y_3 that exution order doesn't matter.
all exute parallely.

Exercise 13

Test bench for exercise 12.

[7:0]
$D_7 \ldots D_0$
MSB LSB

[0:7]
$D_0 \ldots D_7$
MSB LSB

Part select

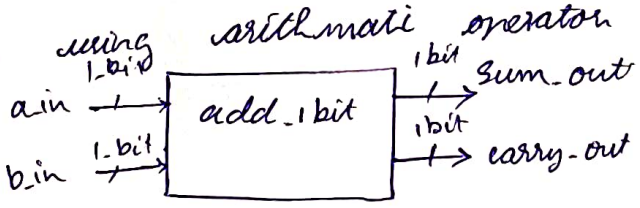
```
module ...
    (input [7:0] a_in,
      :
    );

wire [3:0] Y1;
wire Y2;

assign Y1 = a_in[3:0];
assign Y2 = a_in[7];
      :
end module
```

Arithmatic operators

+, -, *, /, %, ** (power)

Exercise 14

Complete RTL design to perform addition of 2-1bit binary numbers using arithmatic operator



concardination operator

{ ..., --, ..., ... } ⇒ to assign value as extra overflow

Exercise 16

add_n bit

Exercise 17

Test bench for exercise 16 above

Evening session

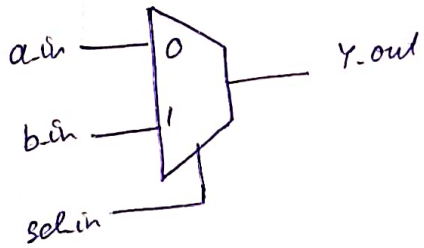
Exercise 18, 19 → Subtractor_n bit

Exercise 20, 21 → 2:4 decoder

always @, if...else

always @ (<sensitivity list variables>)
begin
:
end

(eg) consider 2:1 MUX



sel_in	Y_out
0	a_in
1	b_in

```
module mux_2to1( input a_in,
                 input b_in,
                 input sel_in,
                 output y_out);

always @(a_in, b_in, sel_in)
begin
  if(sel_in == 1'b1)
     Y_out = b_in;
  else
     Y_out = a_in;
end
```

Inside if block all assignment are blocking in combinational logic (=)

non-blocking assignment are used to model sequential logic (<=)

⇒ Why?

In this design.

y_out is output & can't be of wire type as it is assigned within always procedural block, so y_out should be of a reg type

Exercise 22

Complete RTL design of 2:1 mux & test bench. use always procedural block with if ... else

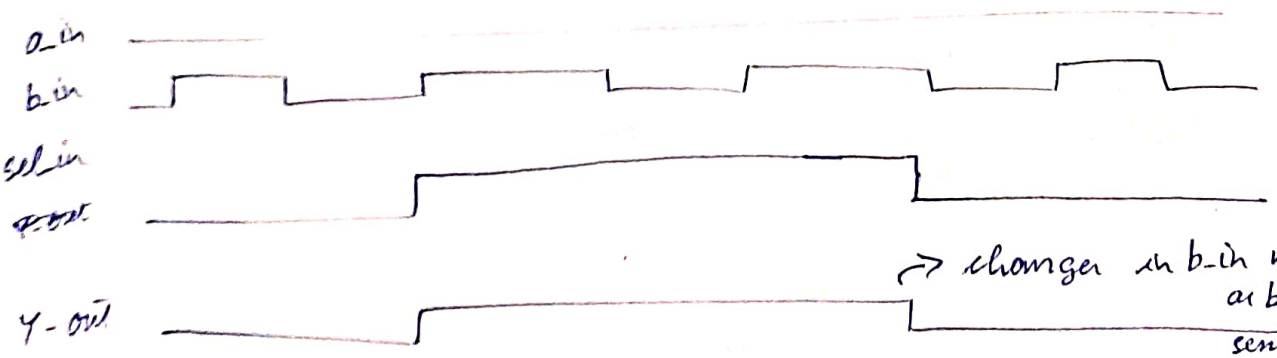
* missing sensitivity list

for example if b_in is not included in the sensitivity list tool will through a warning ↳ not a error of missing sensitivity list signal

(i) reference signal b_in should be on a sensitivity list.

⇒ Schematic synthesis will work as expected

as b_in is missing always procedural block will not be invoked & there will be simulation / synthesis mismatch.



always @(*) => introduced in IEE 1364-2001

-> indicates include all required inputs in sensitivity list

* missing else

↳ will generate a latch & tool will generate a warning.

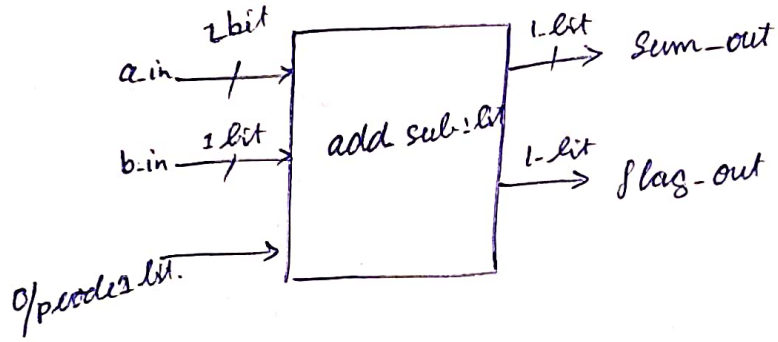
↳ intelligent tool will flash a warning on inteded latch in a design

04-10-2024

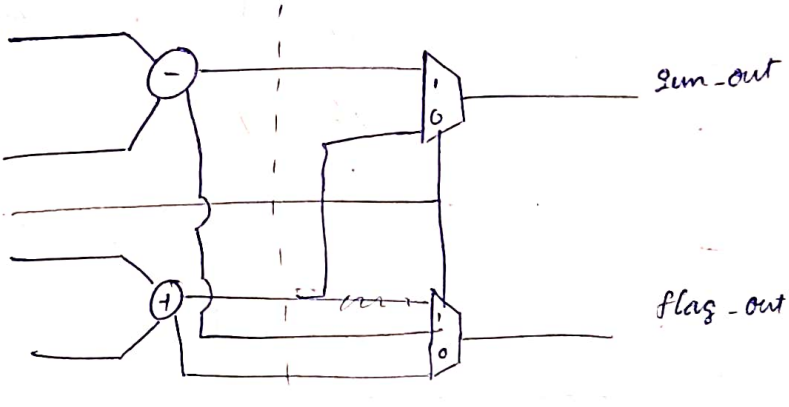
Exercise 23

RTL design for following.

Design 1-bit adder subtractor



opcode	
0	add
1	sub



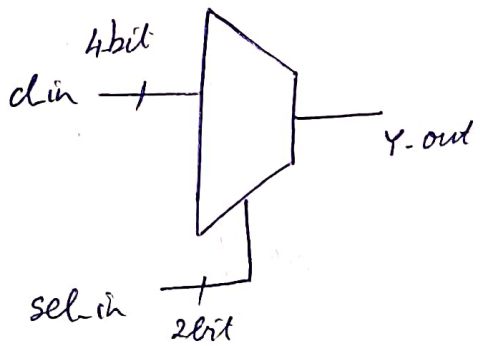
multiple always @ block execute concurrently

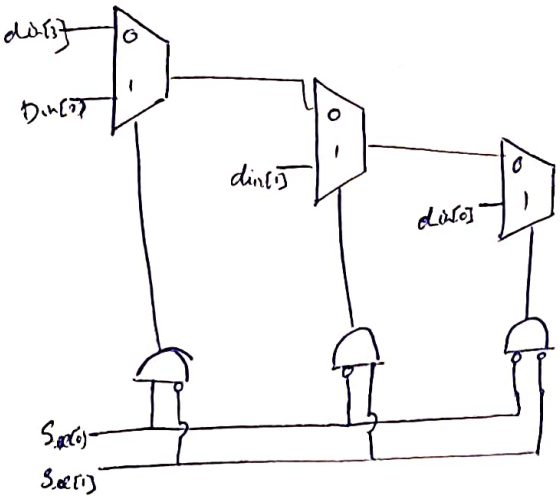
Nested if... else

```
always @*
 begin
   if (condition1)
    begin
    :
    end
   else if (condition2)
    begin
    :
    end
   :
   else
    begin
    :
    end
```

Exercise 26

RTL design of 4:1 MUX using nested if else





nested if else will infer priority logic so never use nested if else

dangling if generate latch in design generated.

case(?)... endcase

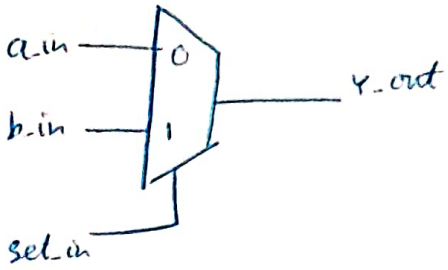
```
case (<variable>)
   <condition-1> : <statement to exute if condition-1>;
   :               begin
   ;               end.
   default     : (                    );
endcase
```

missing case condition will infer a latch

casex, casez are two other variants ⇒ discuss subsequently. :)

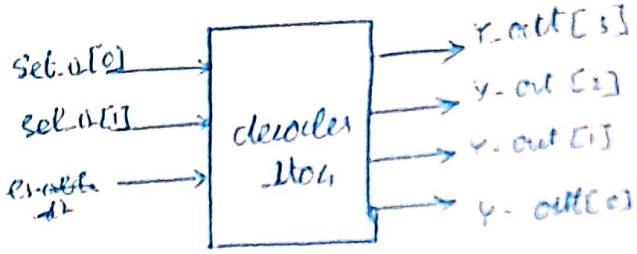
Evening session

Exercise 27 RTL design of 2:1 MUX using case construct



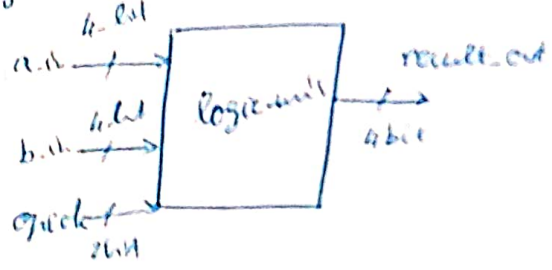
Exercise 28

Complete RTL design of 2:4 decoder using case construct



Exercise 29

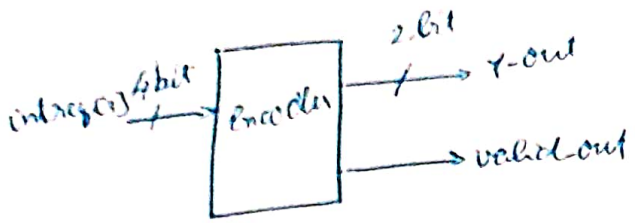
Design logic unit to perform following operation on two 4 bit binary numbers



opcode	operation
0	NOT
1	OR
2	AND
3	XOR

Exercise 30

Design priority encoder to



int_req[3] has highest priority

& int_req[0] has lowest priority

valid out = 0 when all int_req

Exercise 31

Create a testbench.

time unit / time precision

$monitor( ) -> used to display the required i/ps and o/ps at various time (eg)

05-10-2024

always @*

@ (posedge)

@ (negedge)

} ⟹ always @ (posedge) // executes on every posedge of clock.

begin

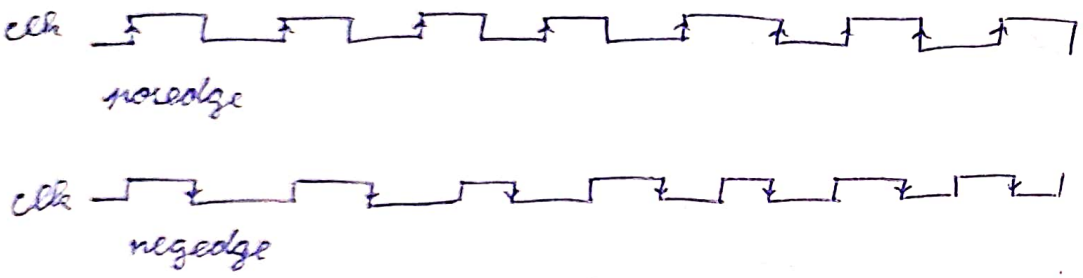
q_out <= d_in; // non-blocking assignment

end

Blocking assignment, Non-blocking assignment BA & NBA

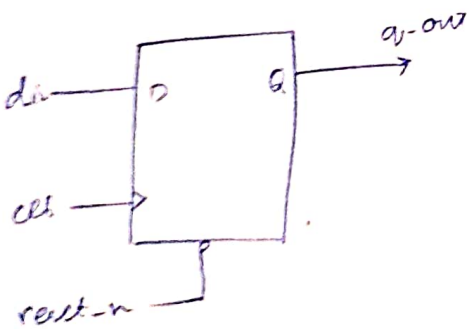
As discussed earlier blocking assignment are used to model combinational logic '=' is used in the assignment within procedural block

It is recommended to use NBA within sequential logic (register) NBA assignments are using '<=', Consider positive edge sensitive D-flip flop. we can use posedge clock where posedge indicates low to high transition. To have negative edge sensitive flip flop we can use negedge of clock which is high to low transition



Example -

Rising ~~active~~ ~~low~~ edge sensitive flip flop with a async active low reset



Don

```
always @(posedge clk, negedge rest_n)
    if (!rest_n)
```

Mix of ~~level~~ sensitive & ~~level~~ sensitive are not ~~sensitive~~ synthesizable

to edge se triggered

$ display

$ monitor.

Exercise 33

Synchronom reset D-ff

Exercise 34

Test bench.

Active low sync reset

```
always @(negedge clk)
begin
    if (!reset_n)
        q_out <= 1'b0;
    else
        q_out <= d_in;
end
```

Exercise 35

negedge DFF active low async reset

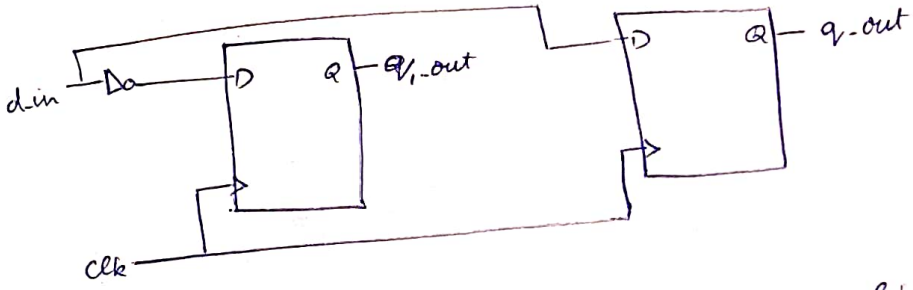
Exercise 36

modify RTL of above to have active low sync rest.

Exercise 37

Find out logic inferred by following design

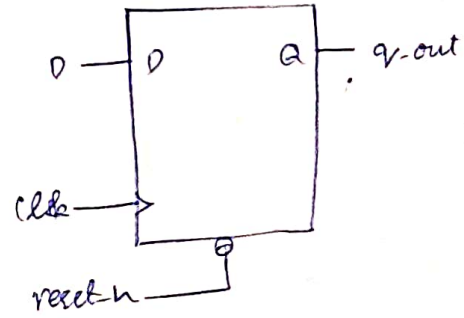
```
always @ (posedge clk, negedge reset_n)
begin
    if (!reset_n)
    begin
        q_out <= 0;
        qb_out <= 0;
    end
    else
    begin
        q_out <= d_in;
        qb_out <= !d_in;
    end
end
```



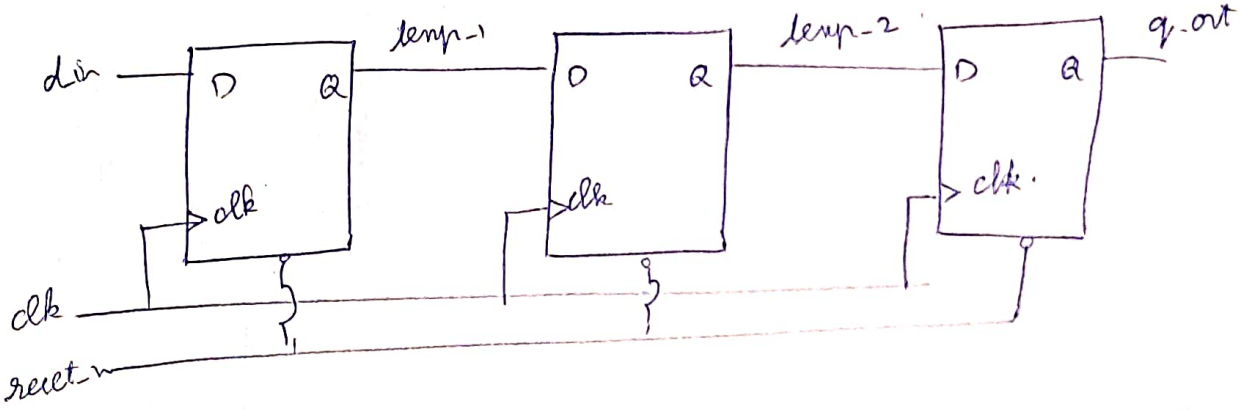
If designer intension is to have shift register having 3 FF and consider the following design which use blocking assignment as shown. this will infer a single FF not a shift register.

```
module shift_reg (input d_in
                  input rest_n, clk,
                  output reg q_out
                  );
reg temp1, temp2;
always @ (posedge clk, negedge reset_n)
begin
    temp_1 = d_in;
    temp_2 = temp1;
    q_out = temp_2;
end
```

Change this using NBA it will infer shift register.



```
always @( posedge clk, negedge reset_n)
begin
    if (!reset_n)
    begin
      temp_1 <= 1'b0;
      temp_2 <= 'b0;
       q_out <= 'b0;
    end
    else
    begin
       temp_1 <= d_in;
       temp_2 <= temp_1;
       q_out <= temp_2)
    end
end
```



If any NBA is change in order no change in logic inferred

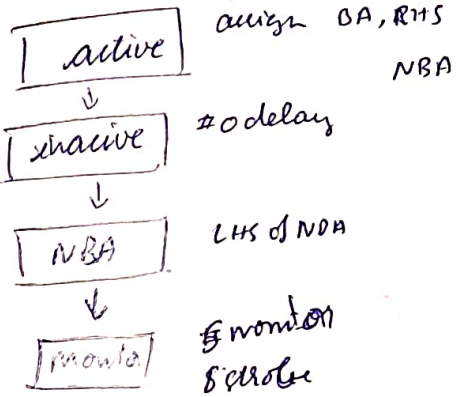
* Reordering of blocking assignment

changes the number of flip flop from no of assignment → 1

* Don't mix blocking, non-blocking assignments.

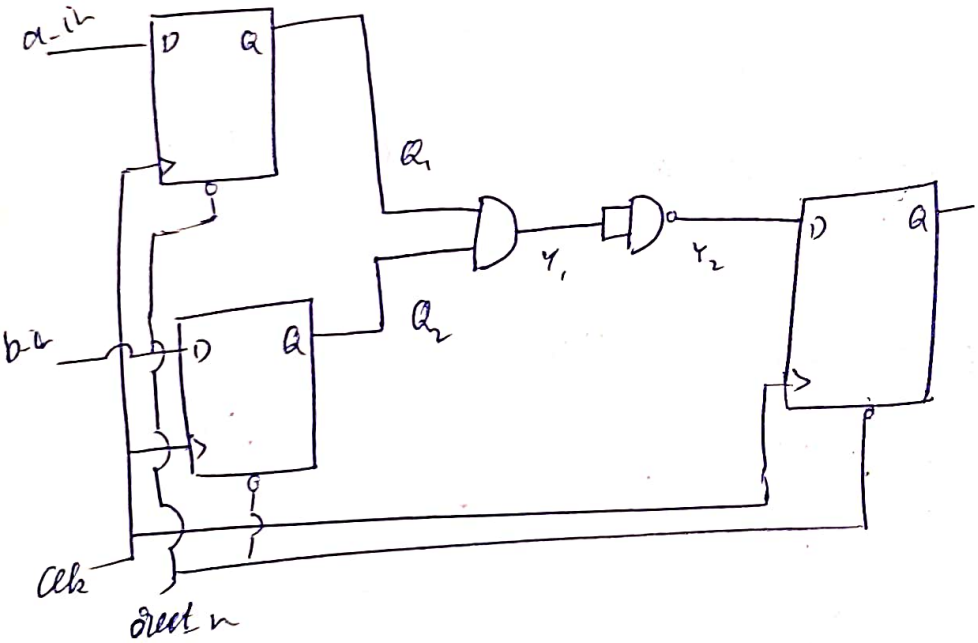
③ FF	② FF	① FF
q_out = temp_2;	q_out = temp_2;	temp_1 = d_in;
temp_2 = temp_1;	temp_1 = temp_2;	temp_2 = temp_1;
temp_1 = d_in;	temp_2 = temp_1;	q_out = temp_2;

queue in verilog



Exercise 35

Compute RTL design for the following.



Exercise 40 counter 2bit up async reset

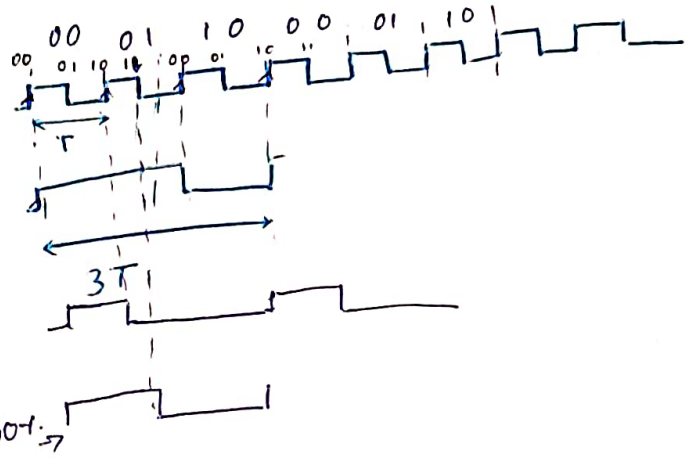
Exercise 42 counter n bit up/down async reset

Replication operator → {. DATA_WIDTH {'b0}}; ⇒ gives warning

Exercise 44 with load

Exercise 46 ÷ 2

Exercise 48 ÷ 3 counter



Exercise 50 50% duty cycle. 50% →

07-10-2024

Exercise 51 Mod 5 counter, load, async active low reset (up-dn-in)

52 Test bench

&

Exercise 53 Ring counter 4-bit, load 8 at initial async active low reset

54 Test bench. qout <= {q_out[0], qout[3:1]};

1000
0100
0010
0001

Exercise 55 0, 8, 12, 14, 15, 7, 3, 1, 0 ... Sequence generator.

8 4 2 1 -8 -4 -2 -1

↳ Johnson counter (twisted ring counter)

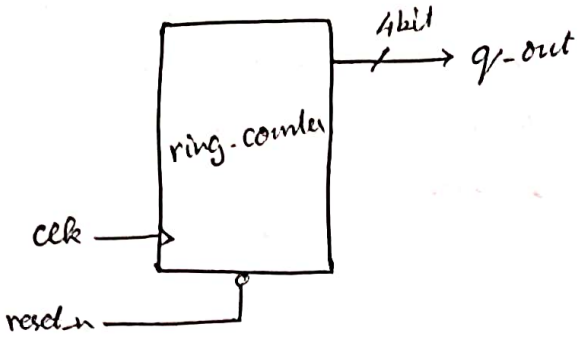
0000
1000
1100
1110
1111
0111
0011
0001
0000

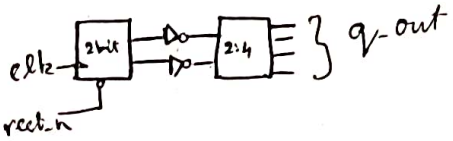
56 Test bench

q_out <= {~q_out[0], q_out[3:1]})

Evening session Exercise 57

RTL Design of 4 bit ring counter using 2:4 decoder & 2 bit syn counter





00, 01, 10, 11 → 1000, 0100, 0010, 0001

11 10 01 00

Exercise 58

Test Bench

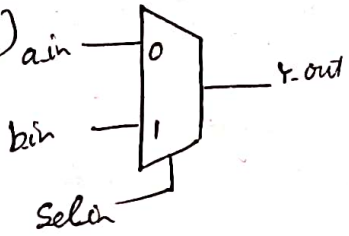
Exercise 59

Shift register, 4 bit left shift, right shift;

Exercise 60

Test Bench & monitor.

Conditional operator (Ternary)



assign Y_out = (sel_in) ? b_in : a_in

08-10-2024

4 to 1 MUX using ternary

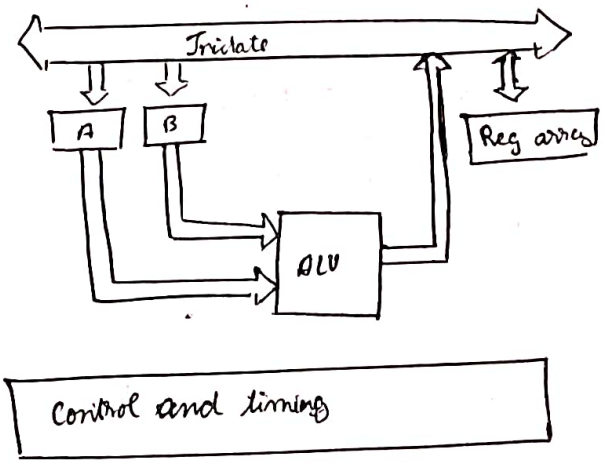
1-bit adder, subtractor conditional operators.

a_in — result_out

b_in — flag_out

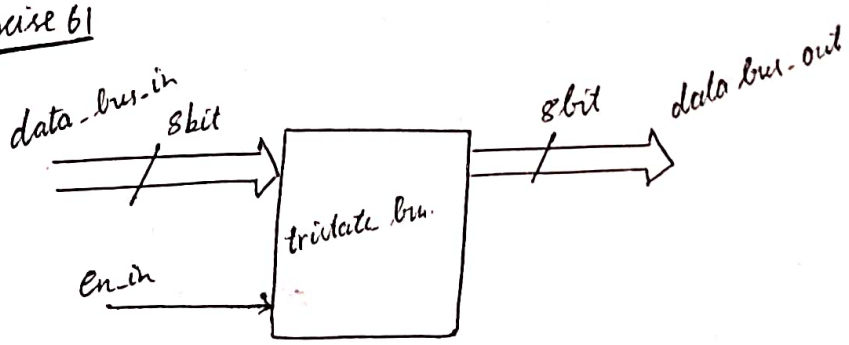
opcode

ALU



opcode ADD(A,B)

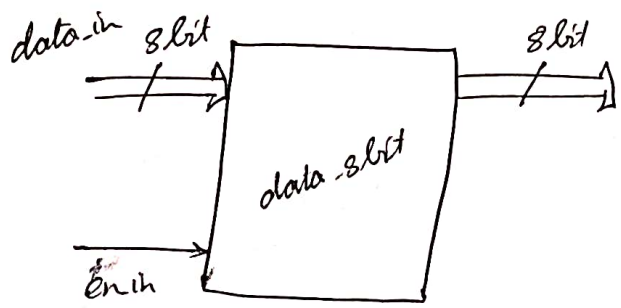
Exercise 61

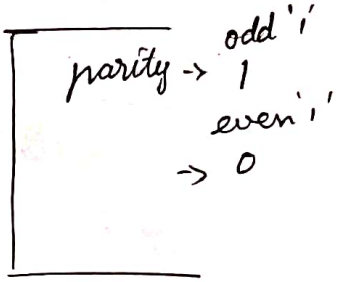


en_in	operation
1	data bus in = data bus_out.
0	high_impedance = Z (or) z

Exercise 62

Intentional latch positive level sensitive 8 bit latch





Test bench

09-10-2024

✓ assign
✓ always @*
✓ always @ (posedge (or) negedge)
↳ sync, async reset
✓ Ternary operators.

Function & Task

function <return type> <name of function> <arguments> (portlist)
..
..
endfunction

Rules..
- fn should have atleast one i/p arg
- fn can call another function but can't call task
- fn can't consist of time control construct
- fn can't consist of output or inout arg
- fn cannot have non-blocking assignment deassign, force, release

#, @, wait
posedge, negedge

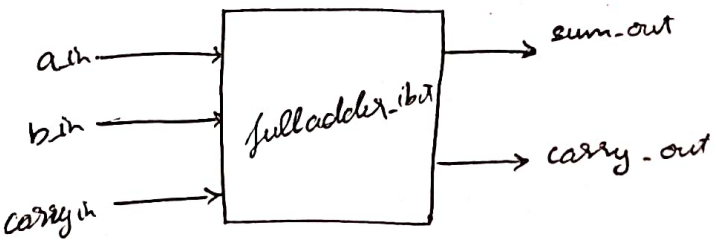
→ sometime we need to perform some operation multiple times in such scenario we can use a function call (function, endfunction) are keywords

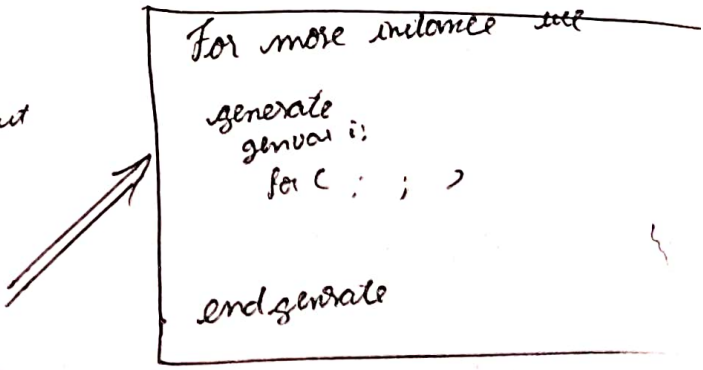
(eg) function [3:0] addition (input [3:0] a_in, b_in);

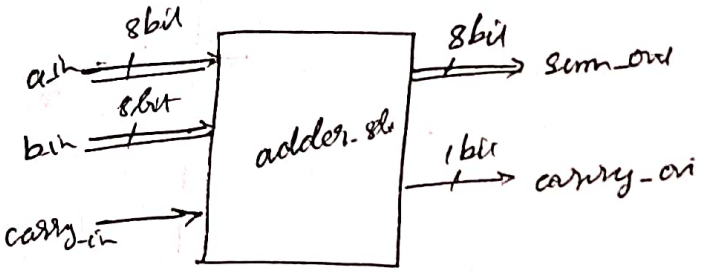
```
begin
    addition = a_in + b_in;
end
endfunction
```

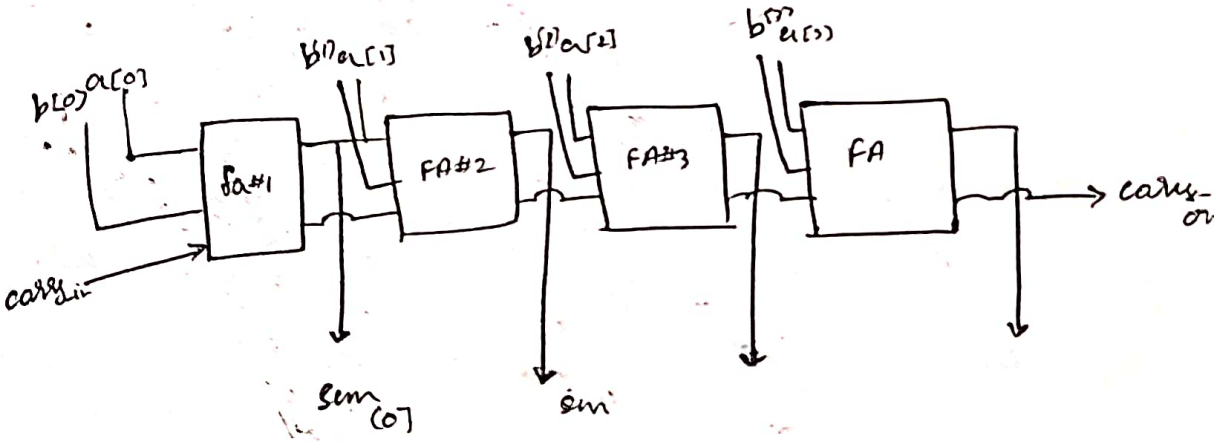
Exercise 63 addition 4bit using function

Excercise 64 1 bit full adder using 1 bit half adder.



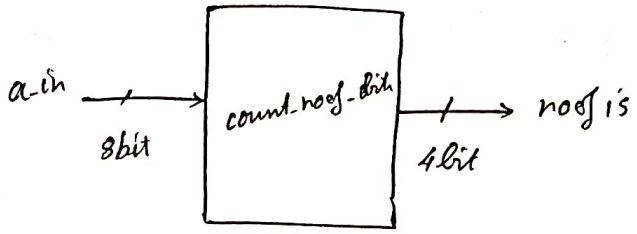






Exercise 65

RTL design to find no of 1's in i/p use function & for loop




```
count=1 (a_in)
for (i=0; i<8; i=i+1)
begin
   if (a_in[i])
end count=count+1;
```

Events section

Verilog95 module declaration

```
module <name> (<parameter>);
  input ..., ...;
  output ..., ...;
endmodule
```

function declaration

```
function <name>;
  <port_assignment>
    (i/p & o/p)
endfunction
```

Task

Task declaration

```
task <name of task> < arg, i/p, o/p, inout >;
  ....
  ....
  ...
endtask
```

Rules for a task

- → task can have i/p, o/p, inout args
- → task can enable another task or function.
- → task will not, can't return any value
  but, same result is achieved using output arg.
- → task can consist of time control constructs
  (#, @, posedge, negedge, wait)
- → we have static task, automatic task
  ↳ task is re-entrant.

Tcl → used in legacy system

puts

set <variable_name> <value>

for {set <var>} {$i <= $num} {incr <var>} {

[expr $sum + $i]

}

[format %c <var>]

---

task call inside always @*

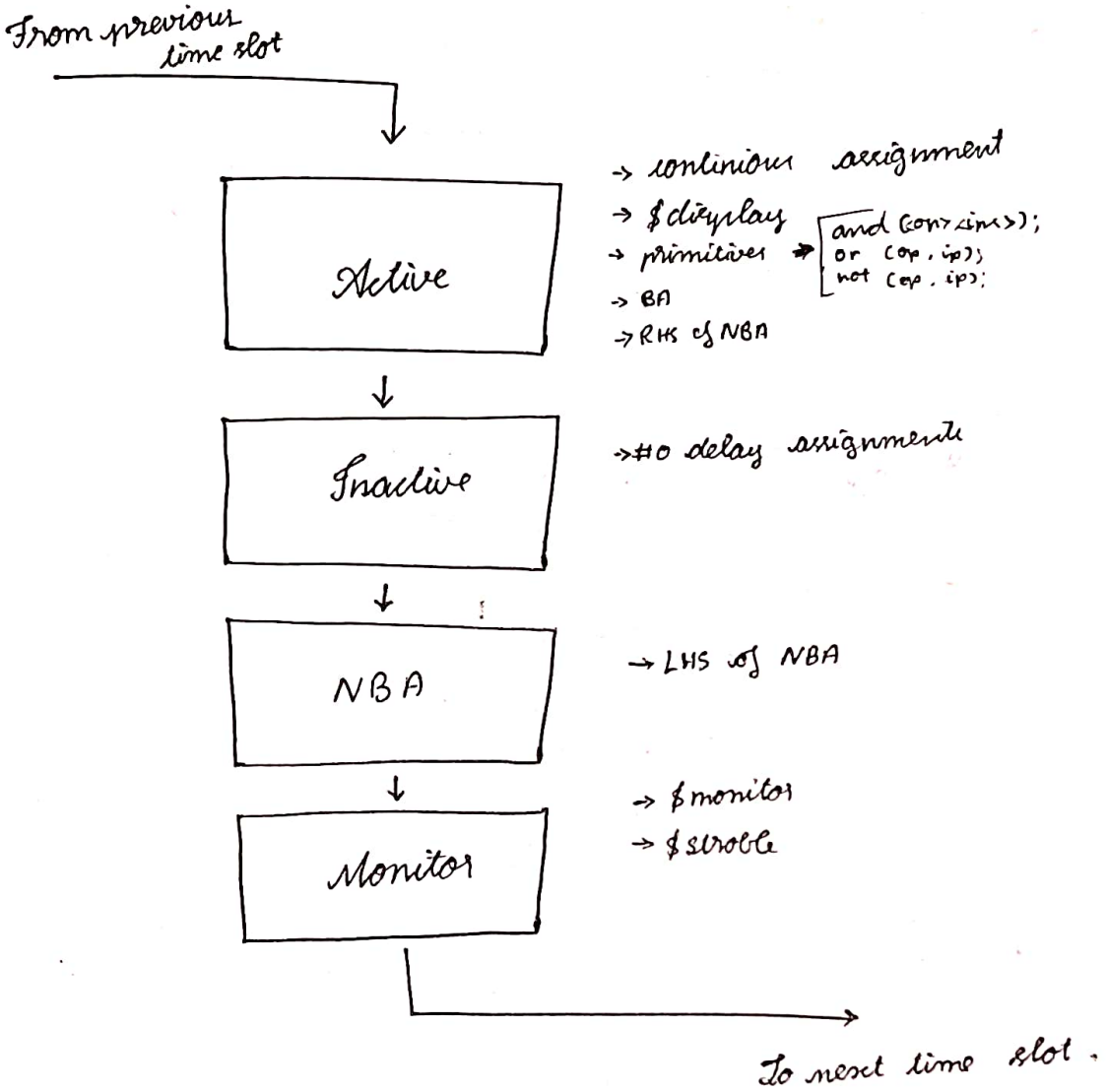
Global task can be declared outside module

Exercise 67 RTL design of posedge D-ff having active low async reset (reset_n) using static task as well as global task

10-10-2024

Exercise 68 RTL Design to find no of zeros from 8 bit i/p using task

Stratified Event Queue



mux 2 to 1 using primitive

full adder using primitive
→ with instance of halfadder with primitive

fork
--
---
join

Evening session

* Static task vs automatic task

```
module tb_design;
initial
  print();
initial
  print();
  ⋮
task print();
    ⋮ for (increment)
endtasks
```

```
task automatic print();
```

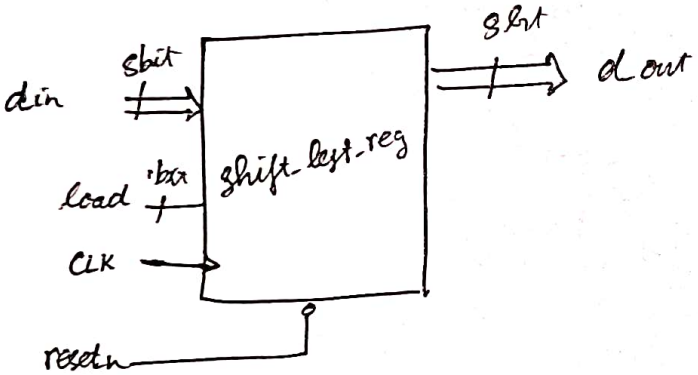
* Loops in design

→ for

Exercise 10 RTL design to have 8-bit shift 1-bit

1) without for loop

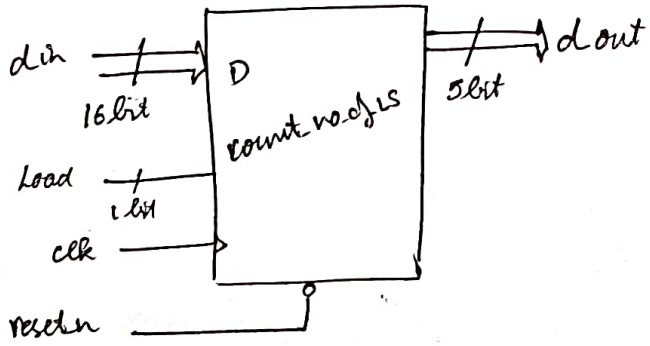
2) with for loop.



8 →

$\lceil \log_2(\text{no of bit}) \rceil$

Exercise 11 RTL design to find no of 1's in 16 bit



11.10.2020

While loop

```
while (condition)
begin
  :
end
```

→ careful about condition
→ not advised to use loops in synthesis
infer unneeded logic, for combo design.
eg) for loop.

forever loop → infinite loop used with time controlled construct

always vs forever

```
always #5 clk = ~clk;
initial
begin
   clk=0;
   #250 $finish;
end.
```

100MHz clk

→ 2 times assignment of variable in 2 different procedural block.

non-deterministic race condition

multidriven net
-> if assigned 2 times in always procedural block

```
initial
begin
   clk=0;
   forever # 5 clk = ~clk;
end

initial
begin
   #250 $finish;
end.
```

---

intra & interdelay

↳ difference in BA, NBA

fork inst1 begin
: :
join end

↳ all execute concurrently.

Exercise 76

fork, join (half adder → tb)

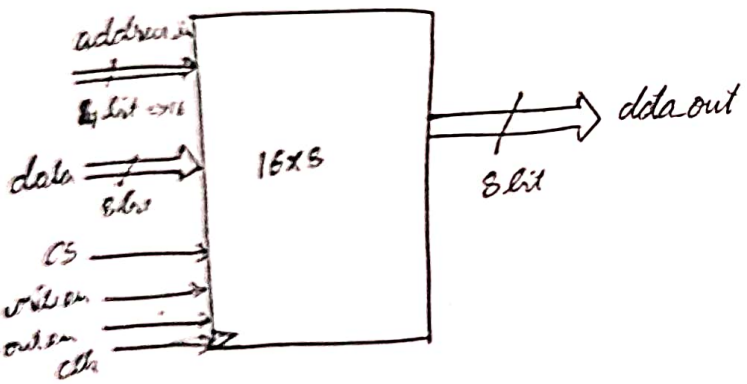
Error in code

Repeat loop

repeat (number)
begin
:
end

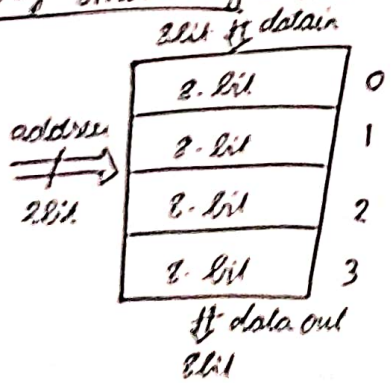
16 bit - 64kB
32 bit - 4 GB
64 bit - 16 Exa B

ROM/RAM operation



CS	write_en	output	operation
0	1	0	→ write to memory → o/p High imp
1	0	1	→ o/p as per addr.
0	x	x	000
1	0	0	store_mem_temp

Memory modeling - consider 1, location of memory




```
reg [7:0] memory [0:3];
:
memory [address_in] <= data_in;
:
data_out <= memory[address_in];
```

Exercise-7

RAM 16×8 bit table & i/p from above.

13-10-2024

Test bench for above

-> clk - period 10ns (ie) clock toggle for 5ns

-> to write 16 memory location we need 160ns similary to read.

-> address to be incremented by '1' for every clock cycle (ie) for every 10ns

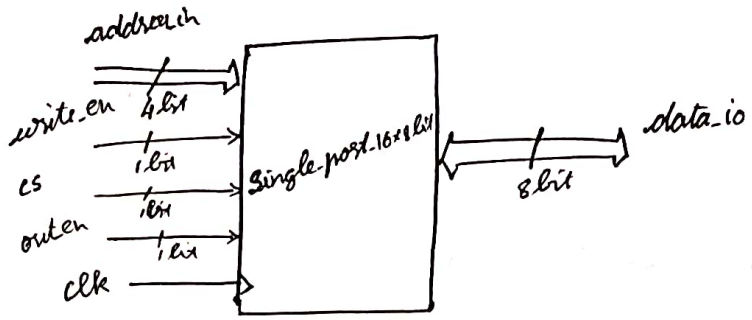
-> chip select 'cs' atleast 320 ns (ie) 160 for read & 160 for write

-> following are test cases.

```
always #5 clk = ~clk;
always #10 address_in = address_in + 1;
always #160 write_en = ~write_en;
always #400 cs = ~cs;          // >320
always #10 data_in = data_in + 1;
always #160 out_en = ~out_en;

initial
begin
   data_in = 0
   address_in = 0
   write_en = 1
   cs = 1
   out_en = 0
   clk = 0
end
```

Exercise 78

RTL Design of single port RAM 16 x 8 bit having bidirectional inout port



cs	write_en	out en
0	x	x
1	1	x
1	0	0
1	0	1

operation:

No operation
write to memory

```
wire data_io [7:0];
reg temp_data [7:0];
assign data_io = (!out_en & write_en & cs) ? temp_data : 8'b zzzz_zzzz;
```

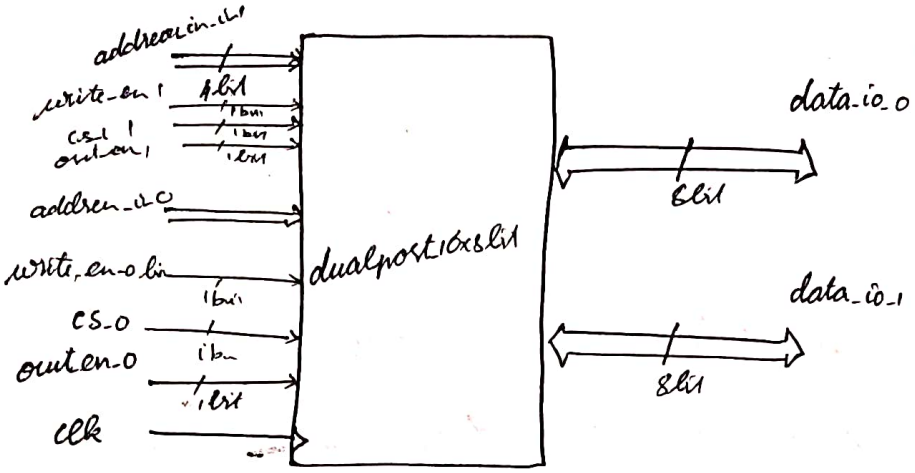
$random

For inout data bus initially write a data in ~~data~~ memory for that concider temp_data & assign that to data_io for write condition

13-10-2024

Exercise 79

RTL design of dual port RAM 16x8 bit having

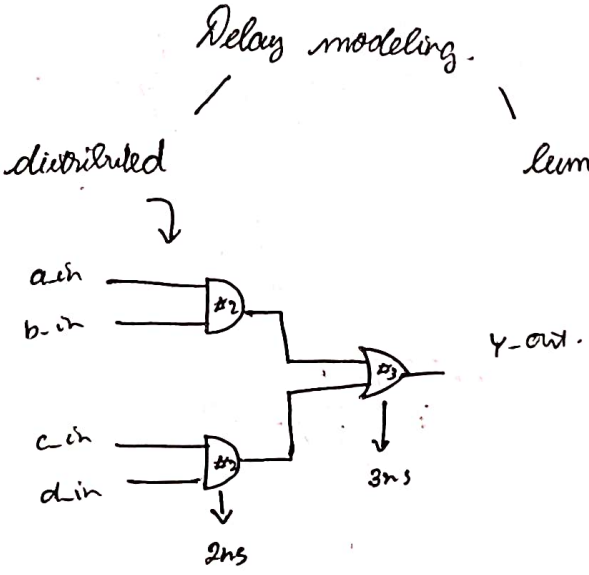


Same operation as before.

Delay modeling

Funtional verification -> without delay

-> to check funtional correctness

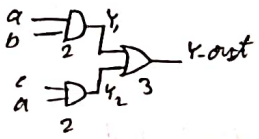


-> delay distributed

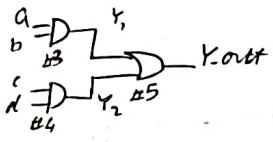
```
module delay_modeling_distributed(input a_in, b_in, c_in, d_in,
                                  output Y_out);
wire Y1, Y2;
assign #2 Y2 = a_in && b_in;
assign #2 Y2 = c_in && d_in;
assign #3 Y_out = Y1 | Y2;
```

⇒ Functional verification is without delays, goal is to check functional correctness of design. But, in delay modeling we can except output of gate after some delay. We have 2 types of delay 1. distributed delay & 2. lumped delay. // In distributed delay delays are specified per gate basis, they are distributed across design // Where as lumped delays are specified as per model basis (ie) single delay on the gate.

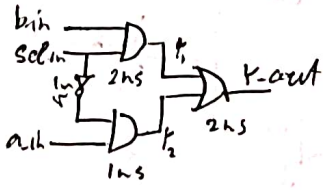
Exercise 80 ⇒ RTL design for distributed delay of



Exercise 81 ⇒ using gate primitive

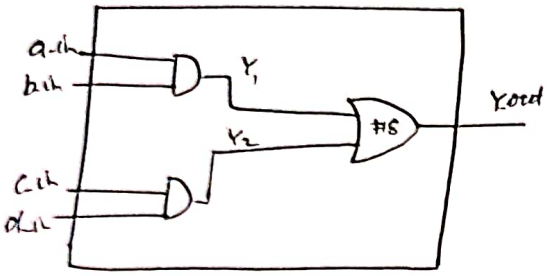


Exercise 81. ⇒ RTL design of 2:1 MUX



Lumped delay

There are module level delays (ie) delays are specified per module. #8 ns is specified.

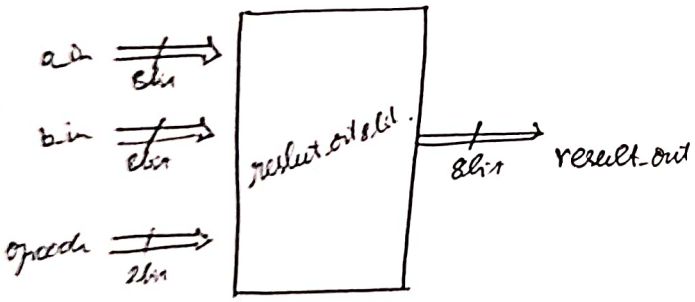


`define <name> <value>

`<name>

⟷

RTL design of 8-bit logic unit to perform following operation.



opcode	operator
00	NOT
01	XOR
10	AND
11	OR

Exercise 84

Consider lumped delay of module

⇒ `include " <name>.vh "

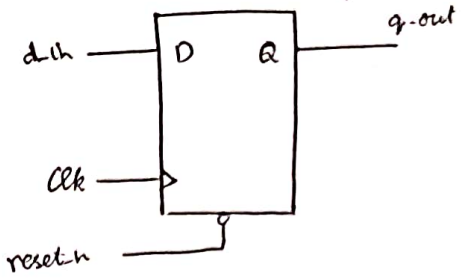
↳ verilog headerfiles

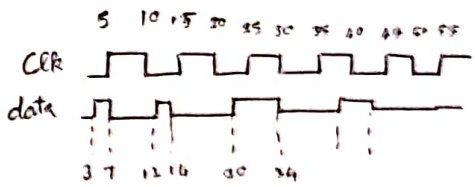
Timing check ⟹

$ setup

$ hold.

$ width





assign D1 = d_in;

assign CLK1 = clk;

specify

specparam tsu = 2ns;

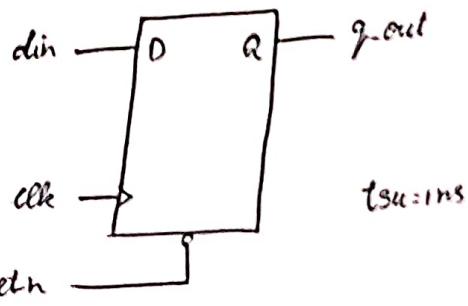
$ setup (D1, CLK1, tsu);

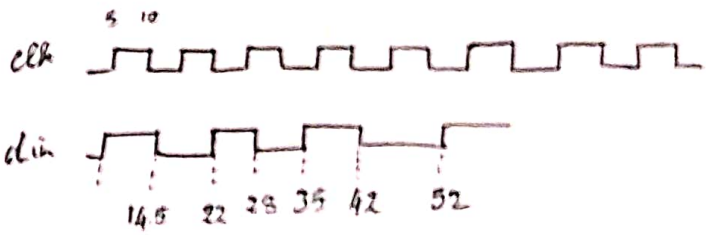
end_specify

14-10-2024

use timing check setup & check for timing violation

$ setup (data_change, reference, limit);





$setup is timing check system task all timing check must be within specify block.

data change is signal changes that is checked against reference.

reference is clk signal or any other signal that is used a reference.

limit is time between two events.

Violation is reported when,

$$T_{reference} - T_{data\_change} < limit$$

in $setup → if data changes within given time limit
it reports timing violation.

Following are scenearios it capture

→ if data change a event occurs at the same time there is no-violation

$hold → it checks ~~check~~ weather data is stable in specified interval of time after clock edge

→ Violation reported if data event & reference event occurs at same time

if the following is true then tool will report

$$T_{dataevent} - T_{referene} < limit$$

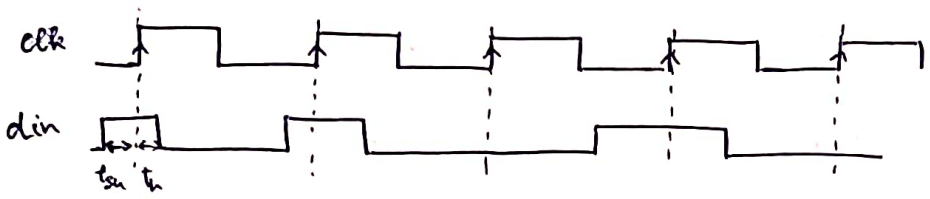
$setuphold(reference_event, data_event, tsu, th);

22-10-2024

→ # width (reference, time limit)

↓ first transition of signal → min time req between T_1 & T_2

Violation occurs if $T_{ref_2} - T_{ref_1} < limit$



$t_{su} = 1ns$

$t_h = 0.5nS$

$\Rightarrow t_{su} + t_H = 1 + 0.5$

$= 1.5ns$

width (posedge din, 1.5);

Questa

New project →

→ Library name (work)

create new file dff.v

tb_dff.v

select file and compile.

simulate

wlf

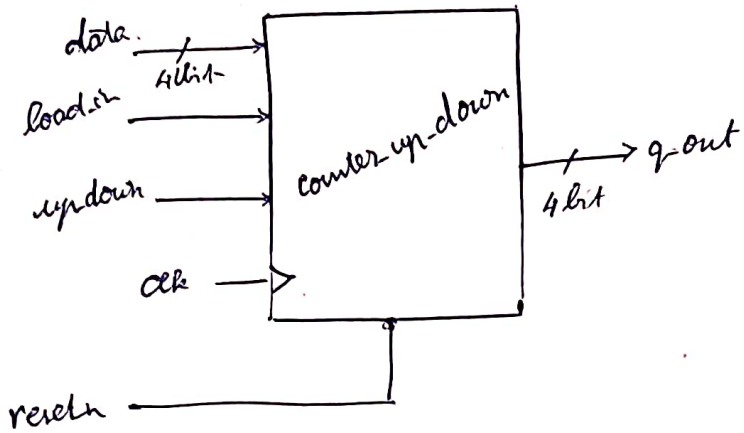
Simulation.

Code coverage. report

tools view code cov.;

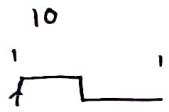
23-10-2024

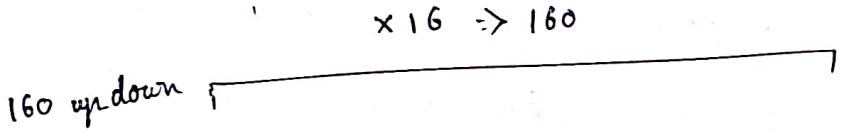
RTL design of 4 bit binary up down counter & get 100% code coverage



resetn	clk	load_in	up_down	operation
0	x	x	x	q_out <= 0
1	↑	1	x	q_out <= din
1	↑	0	1	up counting
1	↑	0	0	down counting

5. clk





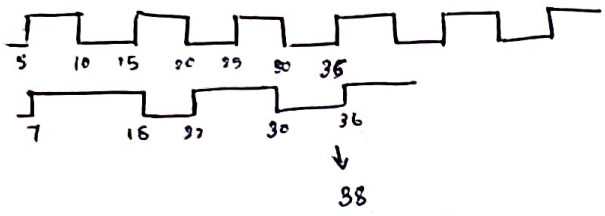
→ toggle coverage of data_in

$skew (<signal to compare>, < >, tskew);

It is used to check the timing violation depending on the arrival of active edge of clk. will know the clk skew is difference between arrival of active edge of clk if diff clks are used in a design which are not sync $skew will

report timing violation when active edge of clk occur outside the time limit allowed for the other clk. when both event will occur at the same time $ skew will not report timing violation.

Tb generate clk1, clk2




```
wire D1;
wire E1;

assign D1 = clk;
assign E1 = en;

specify
  specparam trecovery = 2;
  $recovery (posedge E1, D1, trecovery);
endspecify
```

Tomorrow to see

$ nochange $ period $ recovery.

24-10-2024

$recovery (reference, data, limit);

reference event - edge triggered (pos_edge or, negedge)

-> violation if time interval between edge triggred reference event the lim, if both occurs at same time then violation reported

* NOTE: If reference event arg without edge a error is reported.

Consider tristate buffer.

$nochange (reference_event, data_event, start_time, end_time);

-> checks if data signal is stable in an interval of start time or start offset & end time or end offset. If signal has changed timing violation reported, reverence event argument can be pos_edge or neg edge.

To understand this consider 2 i/p and gate as shown in following

start edge offset $\Rightarrow$ tstart = -2 tend = 2

$period (ref_event, limit);

buff(dout, d_in);

-> to check if period of event sufficiently long or not reference event is edge specific, data event is not specified directly by object at ref event, $period report violation when time of ref event ~ data in less limit.

Consider a buffer

25-10-2024

Conditional compilation

```
`ifdef
  :
`else
  :
`endif
```

Consider PIPO register without reset, that can be included by `ifdef.

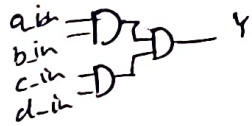
```
`define INCLUDE_RST
module PIPO_8bit (input clk,
                  `ifdefine INCLUDE_RST
                    input reset_n,
                  `endif
                    input [9:0]din,
                    output [7:0] dout);
                  );

  always @(negedge clk)
   begin
     `ifdef INCLUDE_RST
       if (!reset_n)
       begin
         d_out <= 8'b0000_0000;
       end
     `endif
       begin
         d_out <= din;
       end
   end
endmodule
```

Exercise 95

async reset 16 bit PIPO

Exercise 96

`define SynTheris.



`if index has exactly seven

26-10-2024

Generate block → no specify specparam.

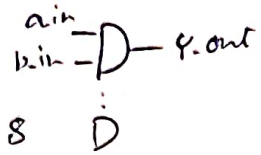
Multiply module instantiation

- Verilog construct, generate block which allow to multiply module (ie) to perform conditional instantiation of any module.
- generate block cannot contain
  port, parameter, specparam, specify block.
- all generate instance are written within a module within generate, ... endgenerate. Generated instance can have assignment, module, always or initial block & user defined primitive

Generate construct are of two types loops using generate for loop. Conditional ~~const~~ generate if..else or case.

Loop variable can be declared using key word genvar. which gives information to compiler or tool that variable to be used during elaboration of generate block

Application consider - replication of instance of AND gate N times. Generate for loop can be used.



Code coverage report → Expression coverage, Branch coverage, Statement coverage, FSM coverage, Toggle coverage

Exercise 98

PIPO.sbit generate d-ff

Exercise 99

full adder n-bit

Conditional generate

generate if → a name indicate use of if else inside generate construct a objective is to select between two different implementation

(eg) 2 i/p NAND based implementation or 2 i/p NOR based implementation in top module a parameter NAND-~~SELECT~~ SELECT = 1 to select a choice of 2 i/p NAND consider the following design

Exercise 100

```
parameter NAND_sel = 1;
generate
    if (NAND_SEL)
        design_and u1 (a_in, b_in, y, a1);
    Else
        design nor u2 (                  );
```

```
case ( )
   :
   :
   :
endcase
```

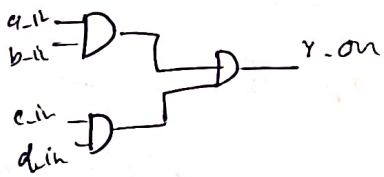
Setenv LD_LIBRARY_PATH /home/student/questa_trial

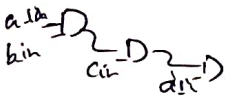
```
vlib work
# design.
vlog -work work ....v
... .. .. .. .v

# tb
... .. v

# runsim
vsim -c -quiet work.<modulename> -do "run -all; quit"
```

Error Generate case.

It need to choose a instance of a module depending on some expression. (eg: consider 2 i/p NAND gate, 2 i/p NOR gate & if gate_type = 1 NAND instance should be selected otherwise NOR instance in such scenario we use generate case.

Exercise 103





Exercise 104

diff wrt diff & diff read.

Exercise 105

if else case.

Already discussed about inbuilt primitives like and, or, not, nand, but not always useful in complex logic. So, to model complex logic we may have to use UDP (user defined primitive) we can have combinational or seq UDP.

1. All UDP should have exactly '1' op can be 0, 1, x never Z

2. Verilog UDP are written in same level as module definition but never between module ... endmodule

2. UDP can have many i/p ports. bidirectional ports are not allowed. All port signals should ~~have~~ be scalar type.

Hardware behaviour is described as primitive state tables which consists of various combinational i/p & outputs within table ... endtable.

Eg: for combinational UDP following can be various values

0 - logic 0, 1 - logic 1, x unknown either 0 or 1, used as i/p or o/p, ? -> logic 0, 1 or x cannot be o/p of UDP. consider inbuild gate level primitive 'and' where single op & multiple i/p

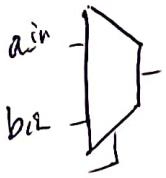
and (y_out, a_in, b_in)

Combinational UDP of and we can model using following

Simulate it & instantiation in test bench is same as model instantiation.

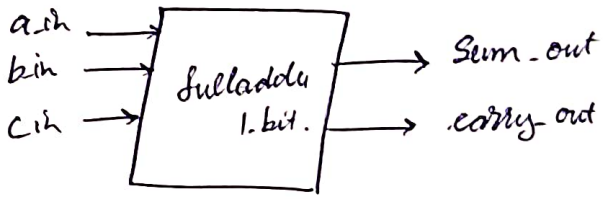
```
primitive and_primitive (Y_out, a_in, b_in);
    output Y_out;
    input  a_in, b_in;
table
// ain   bin    : Yout
   0     0      : 0;
   0     1      : 0;
   1     0      : 0;
   1     1      : 1;
endtable
endprimitive
```

Exercise 10

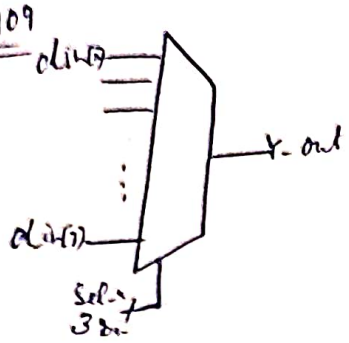


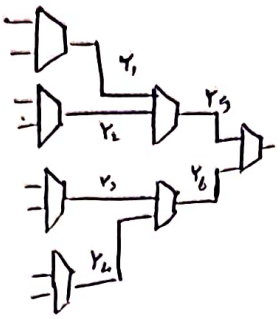

```
sel   a_in   b_in   Y_out
0     1      ?    : 1 ;
0     0      ?    : 0 ;
1     ?      0    : 0 ;
1     ?      1    : 1 ;
```

27-10-2024 Design full adder 1-bit using combinational UDP



Exercise 109





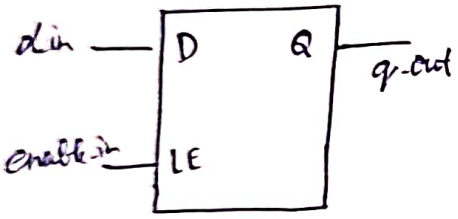
Sequential UDP

Level sensitive
- Latch

Edge sensitive
- flip flop
- counter

Exercise 110

Positive level sensitive




```
primitive latch_primitive
  // enable din : q_ot : q_outff
  table
  end table
end primitive
```

---

* same as ?? indicate any change in i/p
- at o/p only indicate no change

r rising (0→1)

f falling edge (1-0)

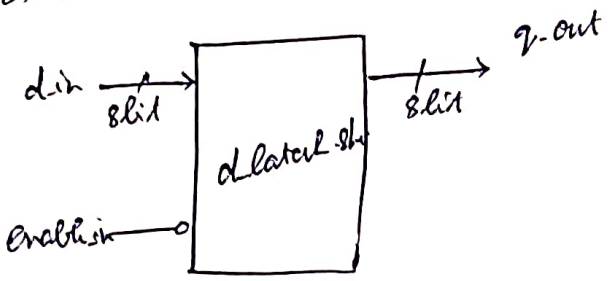
(ab) → a to b

(10) → transition from 1 to 0

(01) → transition from 0 to 1

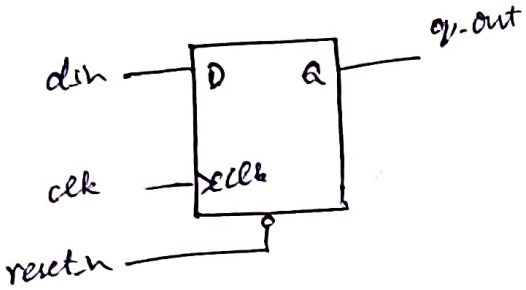
Exercise 111

8 bit D latch having active low enable



Exercise 112

D flip-flop having active low async reset.



Exercise 113.

UDP of negative edge sensitive dff having async active low reset

complete tb & get 100% code coverage.

Exercise 114

Modify rising edge sensitive dff (Exercise 112) to have sync_active low reset_n complete tb & get 100% code coverage

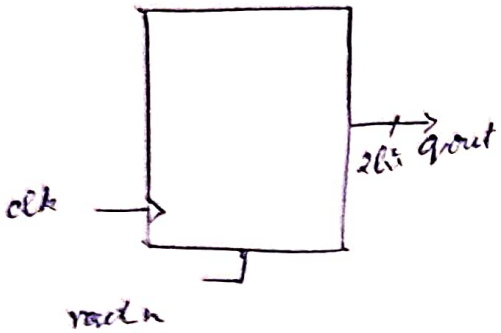
Exercise 115

Using UDP sequential of rising edge sensitive D-ff complete 8 bit PIPO register. consider async reset_n complete

29-10-2024

Example 11b

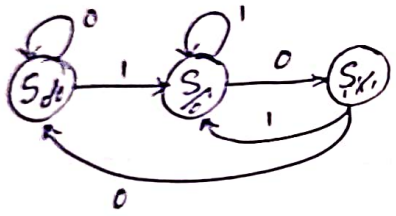
Async Binary 2Bit up counter



Moore sequence detector

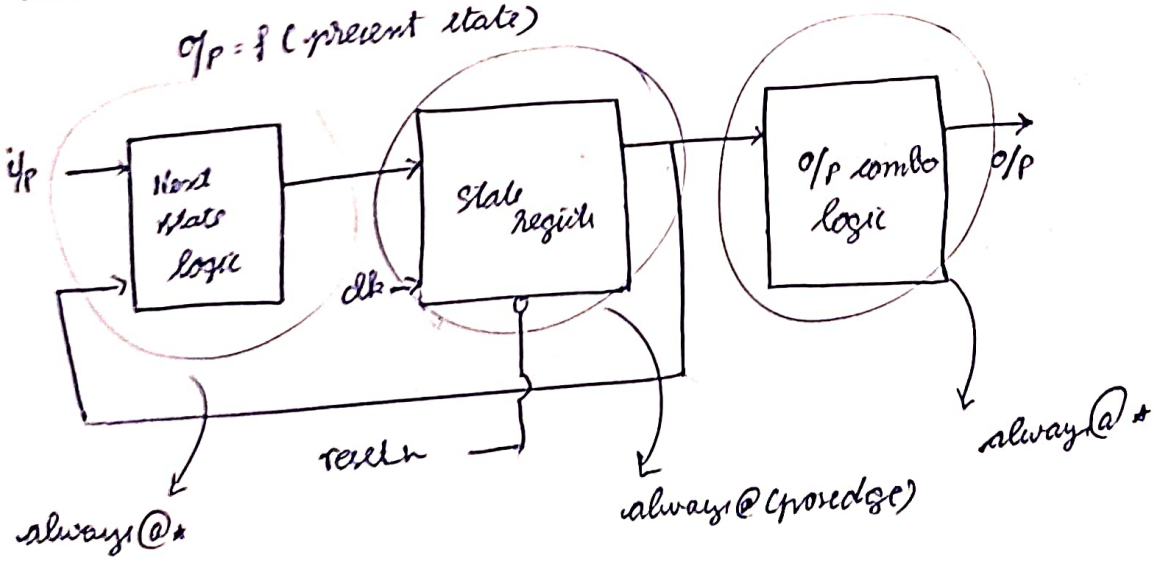
din = 0100 0 10 1 0
q_out = 001000 100 1

10 sequence detector



⇒ Evening session

Moore sequence detector

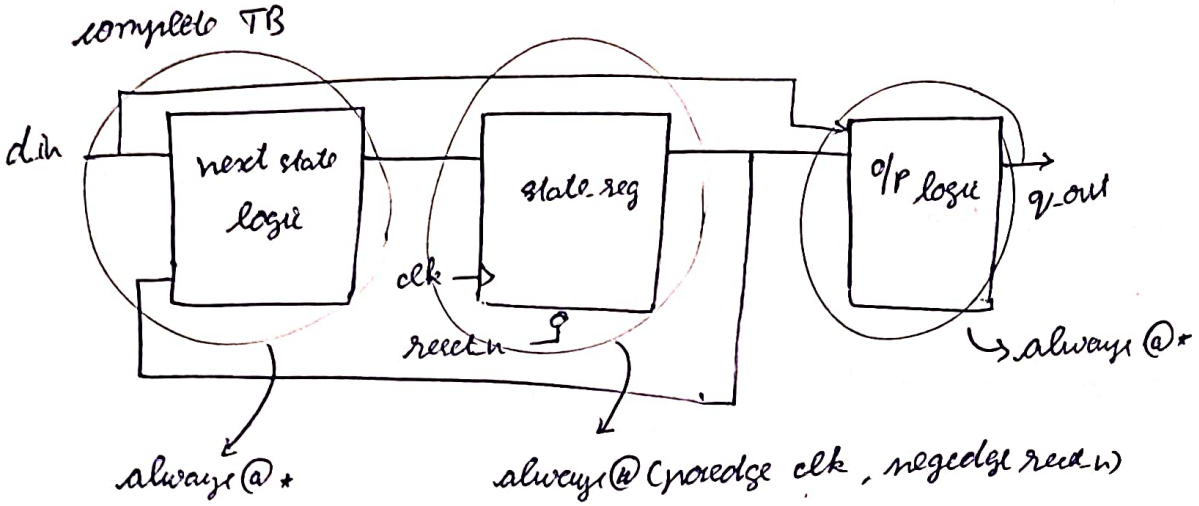


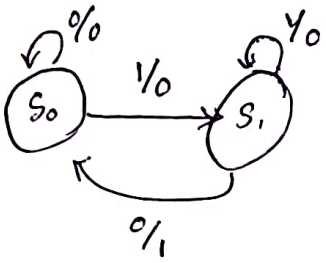
FSM sequence detect using verilog: 10

```
parameter S0 = 2'b00;
          S1 = 2'b01;
          S2 = 2'b10;

reg [1:0] present_state, nextstate;

//State register
always @(posedge clk , negedge reset_n)
  begin
    if (reset_n)
      present_state <= 0;
    else
      present_state <= next_state;
  end
// next state logic
 always @*
 begin
    case (present state)
     S0: if(d_in) nextstate = S1;
         else     S0;
     S1: if (!d_in) next_state = S2;
         else       S1;
     S2: if (!d_in) next_state = S0;
         else       S1;
     default: nextstate = S0;          // all not all binary of met
    endcase

// o/p logic
 always @ (present_state)
    begin
      case ( present_state)
       S0 :  q_out = 1'b0;
       S1 :  q_out = 1'b0;
       S2 :  q_out = 1'b0;
       defaul q_out = 1'b0;
```

Exercise 117

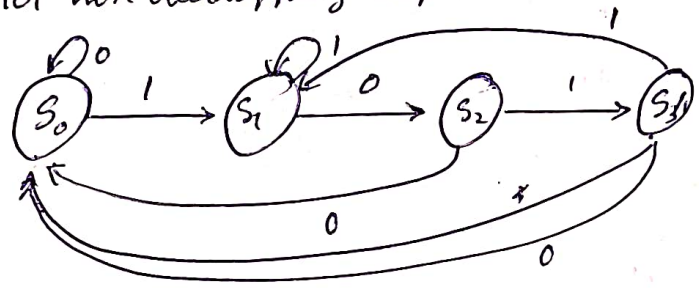
RTL design sequence detector to detect 10 sequence





Exercise 118

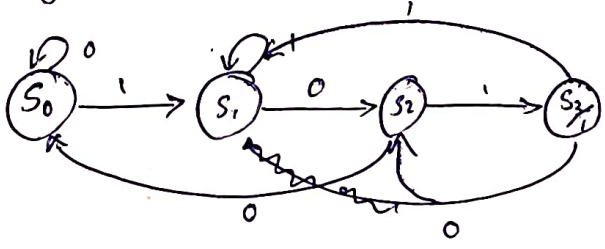
RTL design 101 non overlapping sequence detector. moore



29-10-2024

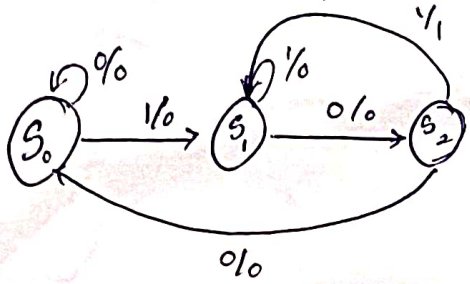
Exercise 119

RTL Design of moore overlapping sequence detector 101



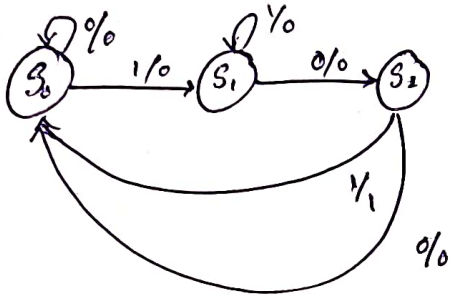
Exercise 120

RTL Design of Mealy overlapping sequence detector to detect 101



Exercise 121

RTL design 101 mealy non-overlapping sequence detector.



→□

Exercise 122.

RTL design of mealy overlapping 10101

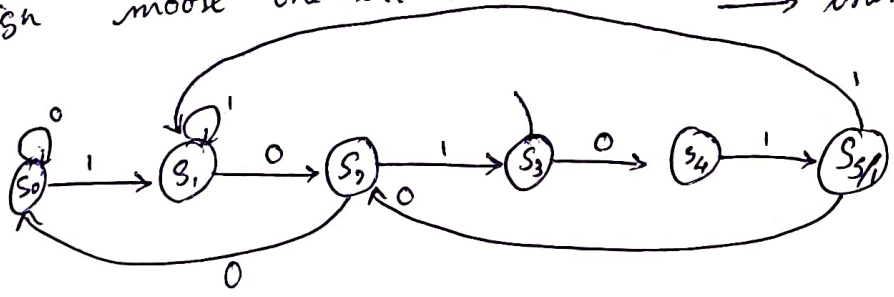
Exercise 123

RTL design of mealy overlapping 101 sequen. detector → gray encoding

Exercise 124

RTL design moore one-hot overlapping → one hot encoding.

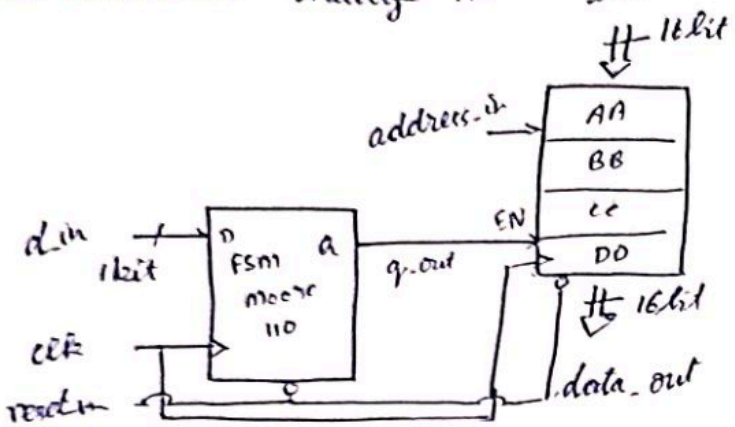
10101



04-11-2024

FSM Moore

FSM controller ~~Mealy~~ 110

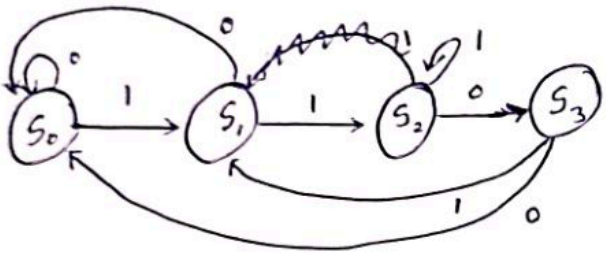


d_in = 1100 110 1100 11 110

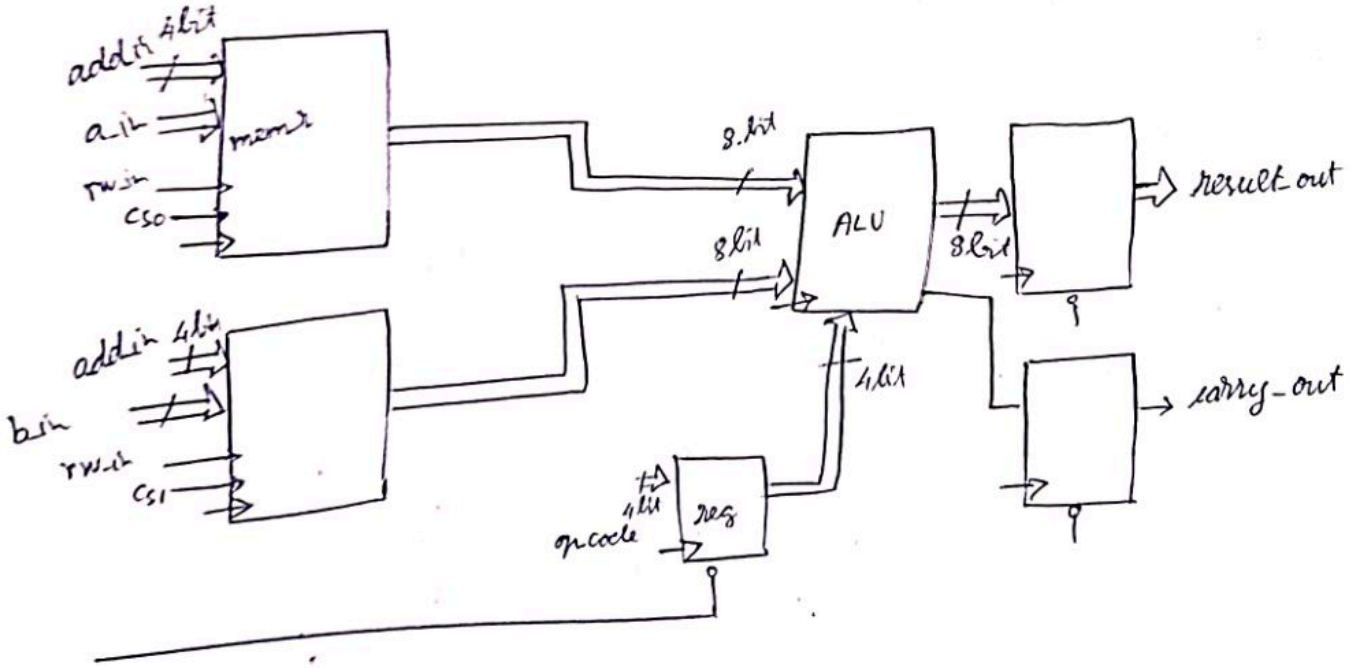
q_out = 001 000 1 001 0 0001

↓ AA ↓ BB ↓ cc ↓ DD

data_in



Complex examples and RTL design

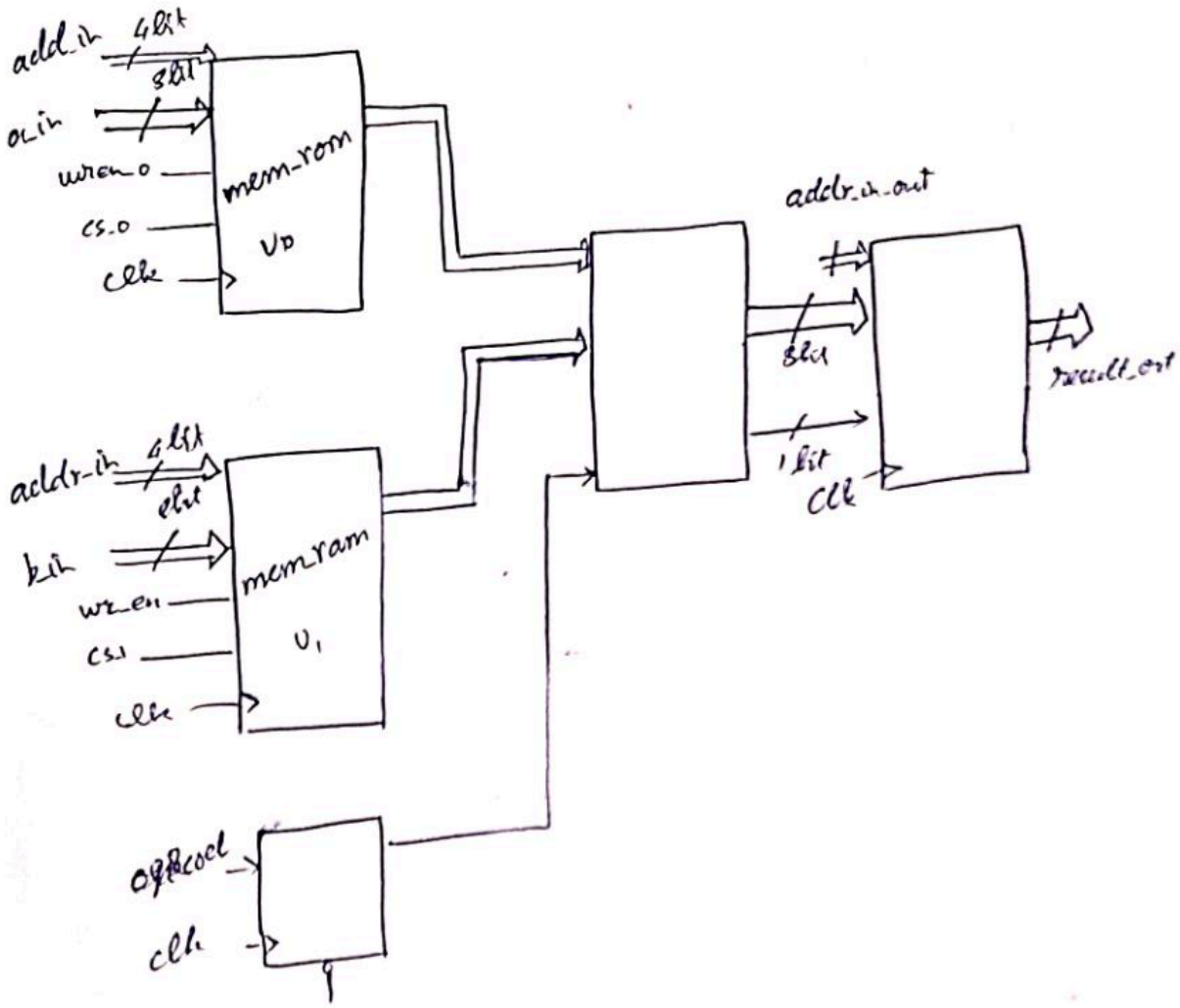


memory

CS	wr_en	operation
1	1	write
1	0	read
0	x	No operation

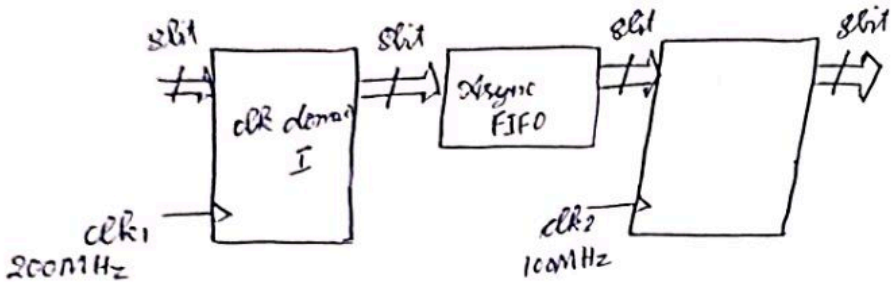
opcode	operation
0000	Transfer a_in
0001	a_in + b_in
0010	a_in + b_in + 1
0011	a_in - b_in
0100	a_in - b_in - 1
0101	a_in + 1
0110	a_in - 1
0111	Transfer b_in
1000	a_in OR b_in
1001	a_in XOR b_in
1010	a_in AND b_in
1011	NOT of a_in

Exercise 13?

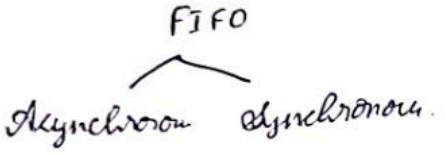

addr_in
4bit
8bit
di_in
wren_0
cs_0
clk
mem_rom
U0
addr_in_out
8bit
1bit
clk
addr_in
4bit
8bit
di_in
wr_en
cs1
clk
mem_ram
U1
offcool
clk

05-10-2024

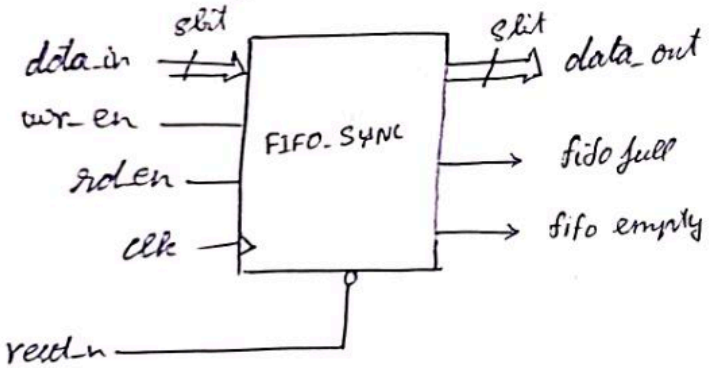
FIFO design



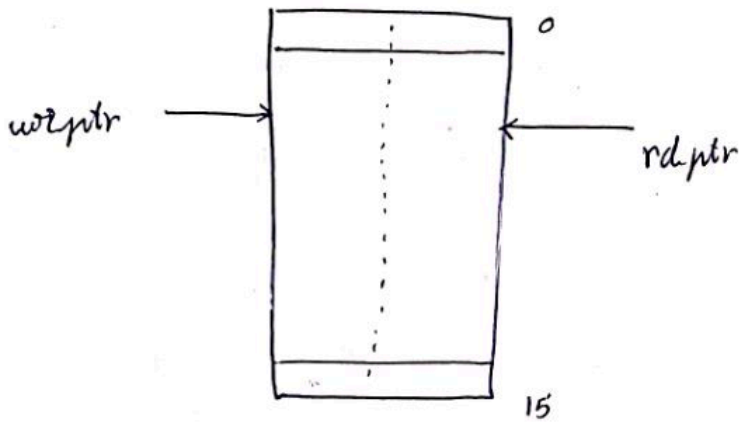
In multiple clock domain design



Synchronous FIFO



1. Design the memory 16x8 bit & develope rd_ptr & wr_ptr logic
2. Develop a logic for FIFO full, FIFO empty
3. Complete RTL Design
4. Complete TB
5. 100% code coverage.

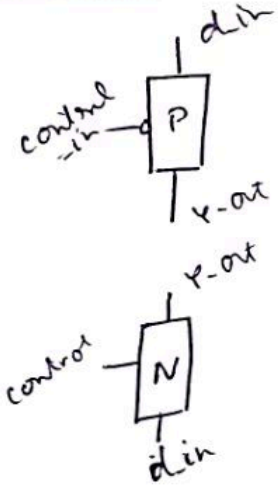


if (fifo_full == 0)
↳
if (fifo_empty == 0)

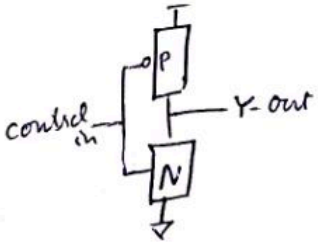
$clog2( );

06-11-2024

Switch level modeling



CMOS inverter.



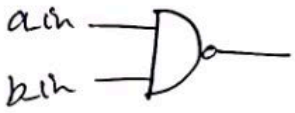
p-MOS & n-MOS switches can be used in switch level modelling

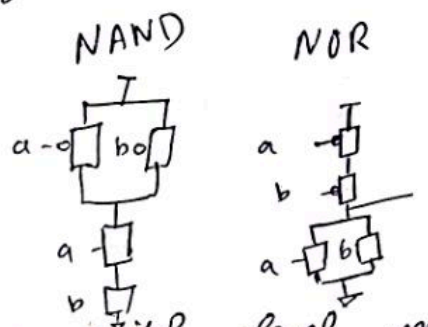
n-MOS can be used to pass strong 0, pMOS to pass strong 1

```
pmos u1 (Y_out, d_in, control_in);
nmos u2 (Y_out, d_in, control_in);
```

Exc 133

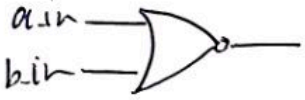
Implement 2 i/p NAND using switch level modeling



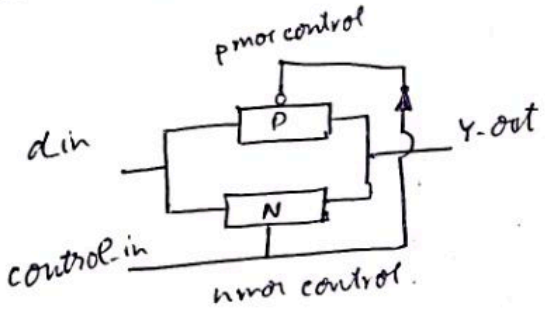


Exc 134

Implement 2 i/p NOR using switch level modeling



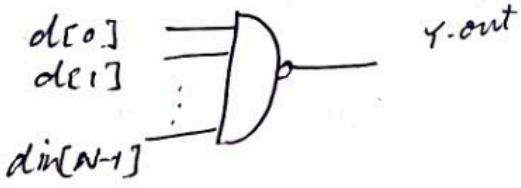
CMOS switch



Switch level modeling of CMOS

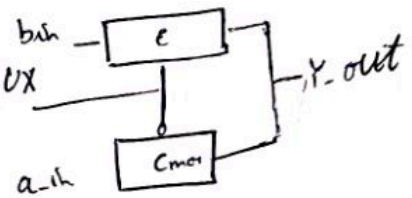
Exercise 136 Implement N i/p NAND using switch level modelling

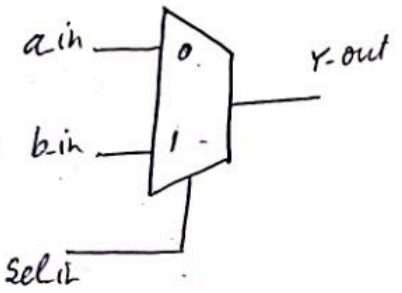
N=4



06-10-2024

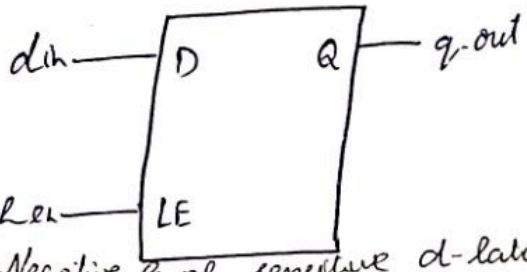
Exercise 138 Switch level modeling of 2:1 MUX





Sel_in	Y_out
0	a_in
1	b_in

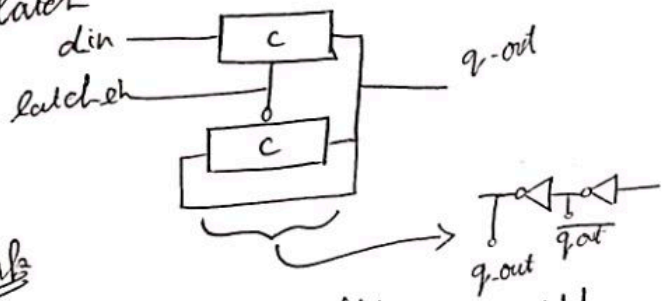
Exercise 139 Switch level modeling of D-latch positive level sensitive



latch_en	q_out
0	No change
1	d_in

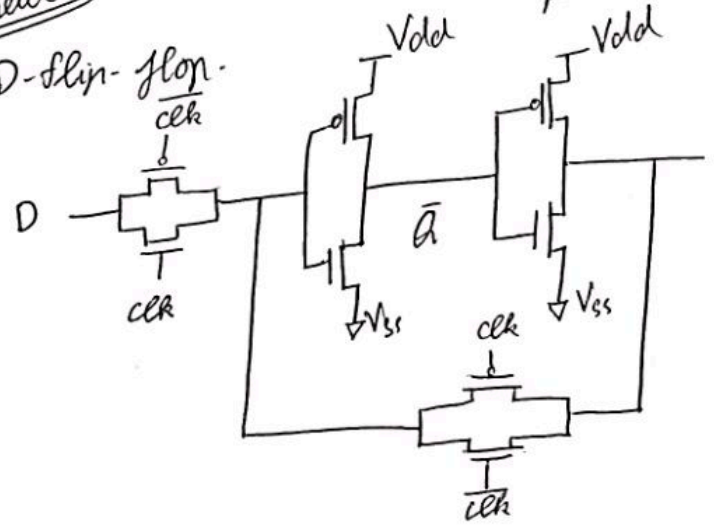
Exercise 140 → Negative level sensitive d-latch.

D-latch



Homework

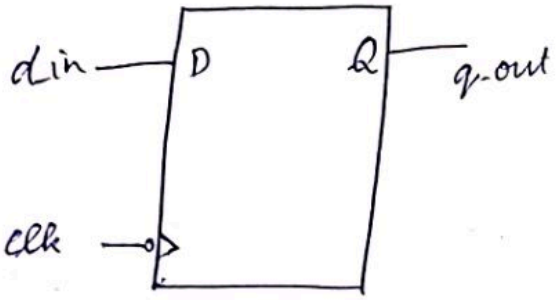
D-flip-flop



data-input - d_in
clock - clk
output - q_out

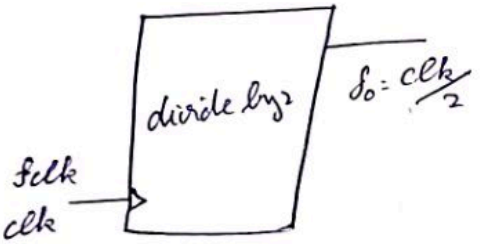
07-11-2024

Exercise 141 Switch level modeling of negative edge sensitive D flip flop.

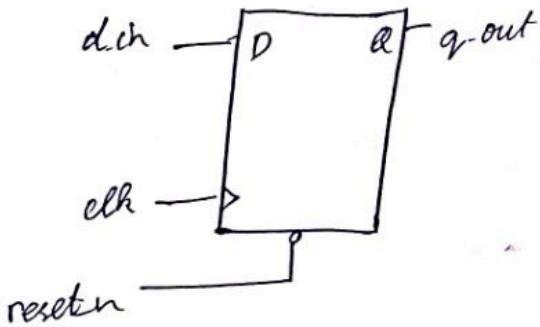


clk	q_out
↓	d_in
⌐	No change.

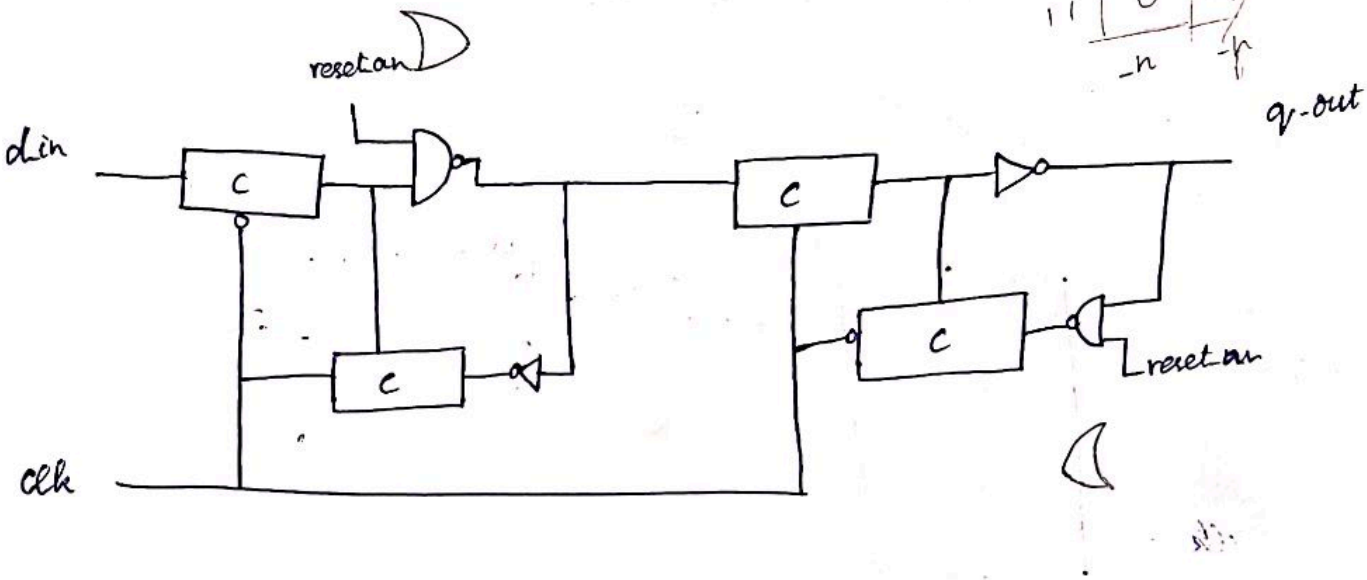
Exercise 142 Cmos switch level modeling



Exercise 143 Include active low asynchronous reset_n in positive edge triggered D flip flop.



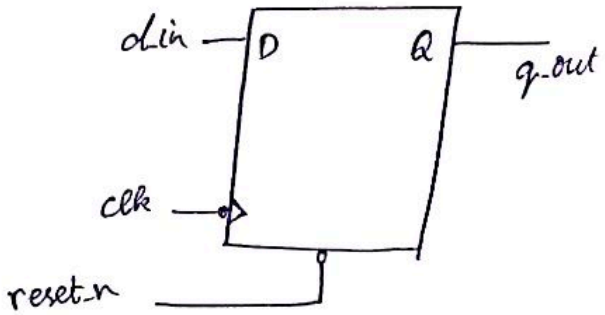
A B	$\overline{A.B}$	$\overline{A+B}$
0 0	1	0
0 1	1	1
1 0	1	1
1 1	0	1
	-n	-p

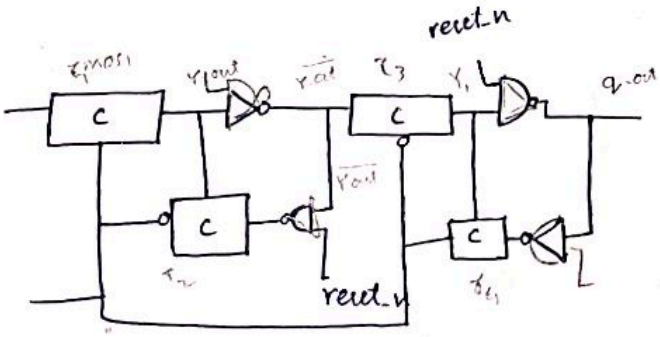


Asynchronous reset an

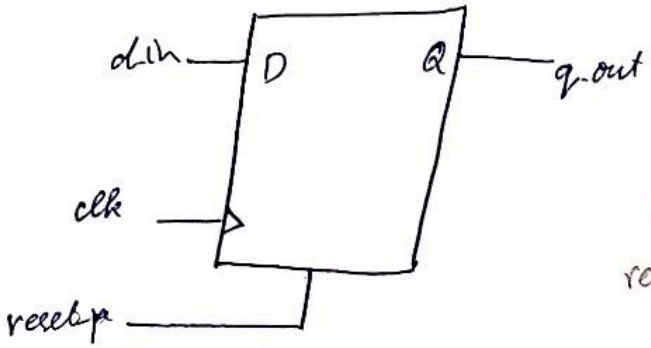
08-11-2024

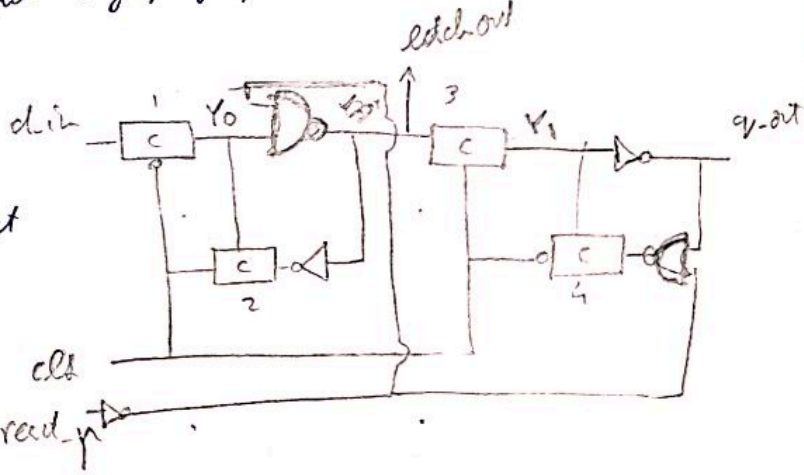
Exercise 144 -> negative edge triggered D flip flop with async active low reset_n



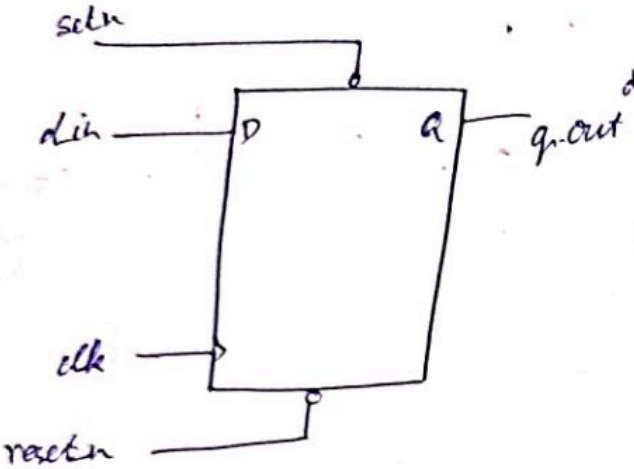


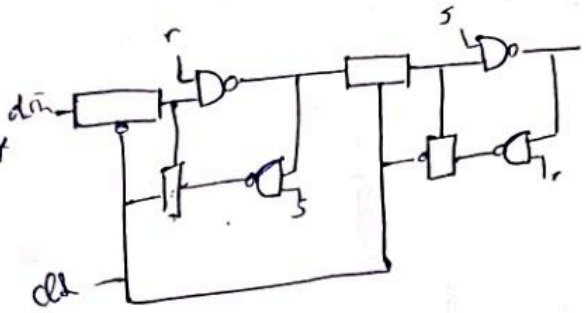
Exercise 145 -> positive edge trigged D flip flop having active high asynchronous reset





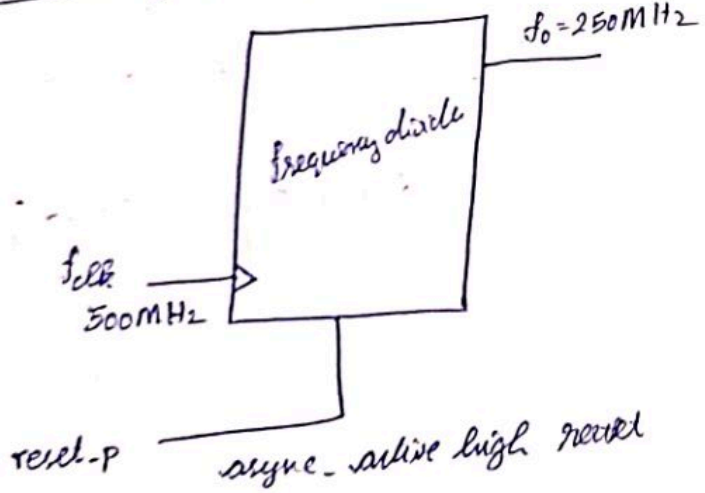
Exercise 146 positive edge triggered D ff having async- active low reset_n & async active low set_n



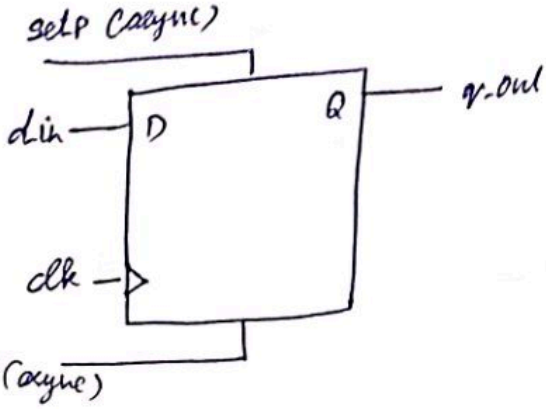


Evening Session.

Exercise 147 frequency divider



Exercise 148



09-11-2024

Applications of Verilog constructs

-> Inter delay assignment

```
#5   a=0;
#10  a<=0;
```

↓ LHS side

-> Intera delay assignment

```
a = #5 0;
b = #10 1;
```

↓ RHS

Difference between Blocking assignment Non-Blocking assignment in Test Bench.

Delay assignment.

Verilog has two type of delay assignment inter delay assignment & intera delay assignment

Inter delay - delay value on LHS of operator, which indicates the statement is executed after delay expire.

(eg) Consider initial procedural block having inter delay assignment (mostly used). These assignment will execute when delay expires. ->

```
initial begin
    #5  a =0;
    #10 a <=0;
end
```

Intra Delay.

In this delay is on RHS of operator. Here statement is evaluated and value of all signals on RHS are captured first. then assigned to reluctant signal after delay expires.

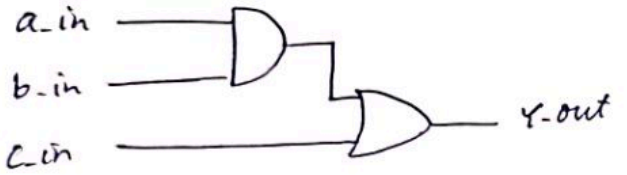
(Eg) initial begin

a = #5 0;

a <= #10 1;

end

RTL design



t	a_in	b_in	c_in	Y_out
0	0	0	0	0
10	1	0	1	0
15	1	0	1	1

Testbench NBA

a_in <= 0, b_in <=0, a c_in <= 0;

#10 a_in <= 1, b_in <= 1, c_in <= 1;

#50 $finish;

Both NBA & BA given same delay

Consider simulation source. having inter-delay assignment with NBA within initial procedural block checking for Y_out

```
module beh_design;

  reg a_in;
  reg b_in;
  reg c_in;
  reg Y_out;

initial begin
  $monitor C...

  a_in <= 0;  b_in <= 0   cin <= 0
#10 a_in <= 1;            a_in <= 1

#5  Y_out <= (a_in < b_in) | Cin;

#60 $finish;

end
endmodule
```

t	a_in	b_in	c_in	Y_out
0	0	0	0	0
10	1	0	1	0
15	1	0	1	1

Inter assignment & intera assignment delay & NBA in a Test bench

```
  :
  Y_out <= #5 (a_in < b_in) | c_in;
  :
```

t	a_in	b_in	c_in	Y_out
0	0	0	0	x
10	1	0	1	x
15	1	0	1	0

```
#5  a_in <= 1;      c_in<=1
    a_in = 1;       c_in=0
```

NBA
BA

0	
10	
15	1 0 1 1

	a_in	b_in	cin	Y_out
#5	<=0	<=0	<=0	<=0
#10	=1	=0	=1	
#25	<=0	<=1	<=1	
	=1	=0	=1	

Y_out <= #10 (a_in < b_in) | cin;

0
0
0
0
0
0
#50 1

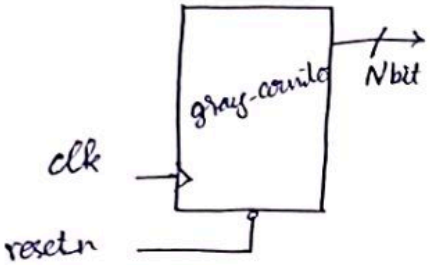
	a_in	b_in	c_in	Y_out
0	0			0
5	1			0
10	0			1
25	1			1

fork

:

join

Exercise 149 RTL Design of N bit gray counter, use async active low resetn



Complete TB

Have 100% Code coverage.

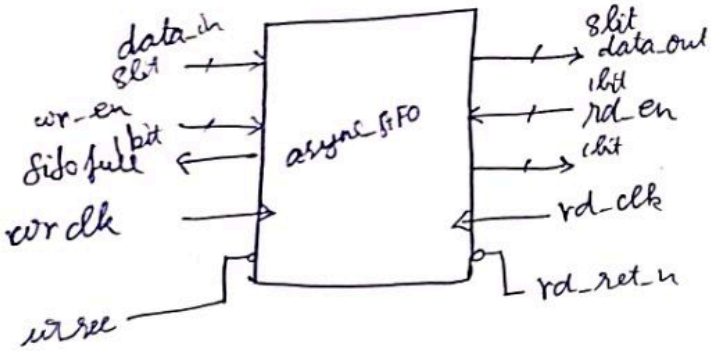
Application Multiple Clock domain data path

wr_clk = 100MHz

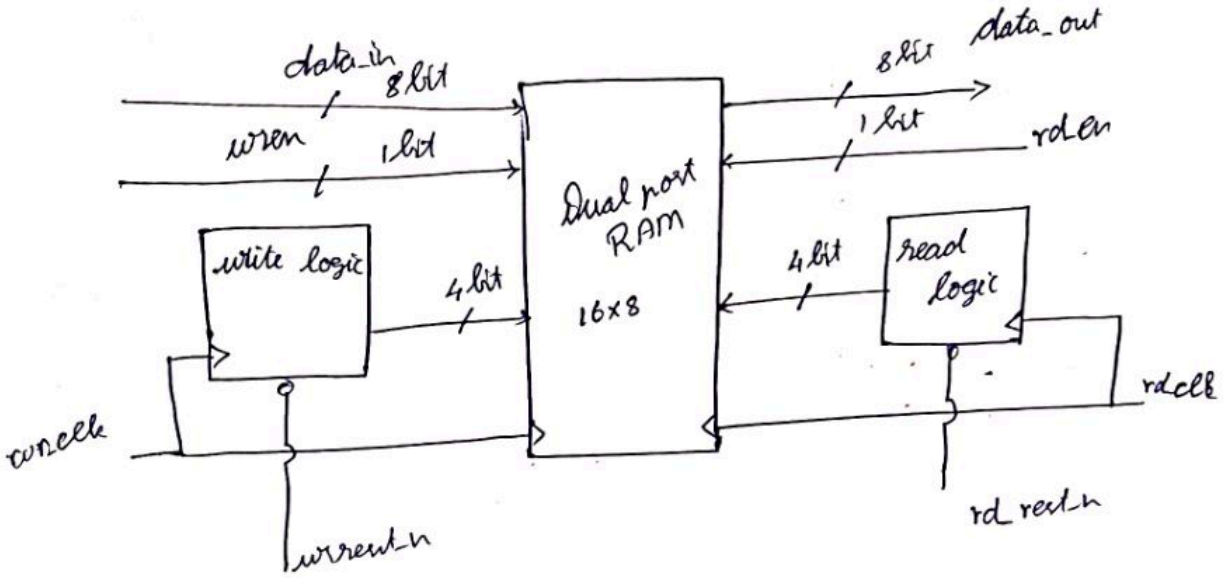
rd_clk = 50MHz

Burst size = 32 byte

Depth of FIFO = ? ⇒16 byte



- Dual port RAM
- Synchronizer
- write logic
- read logic

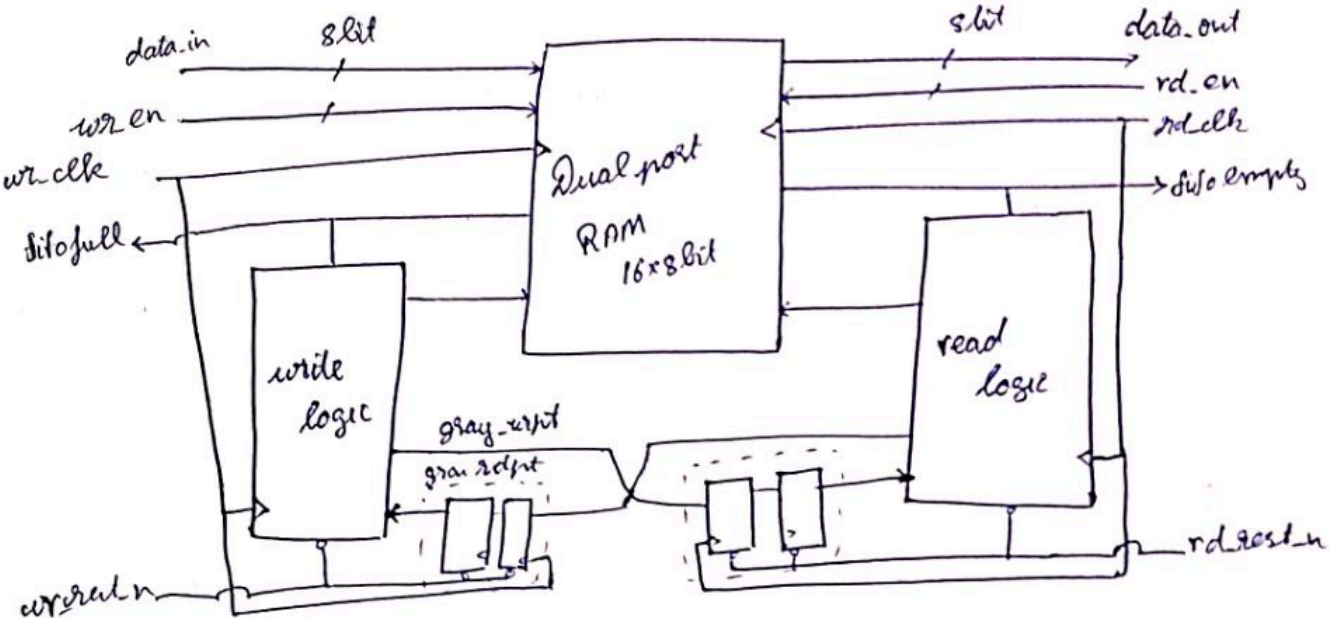


FIFO full

gray_wrt_ptr == {~(gray_rd_ptr [WIDTH:WIDTH-1]), gray_rd_ptr [WIDTH-2:0]}

FIFO empty

gray_wr_ptr == gray_rd_ptr



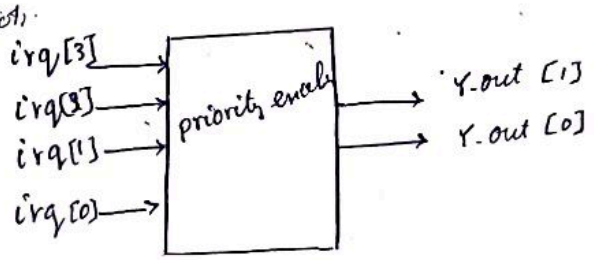
11-11-2024

Three variants of case

case
casex → x propagation.
casez
↳ 2'bIZ =

Priority encoder

Highest Priority



Exercise 152

↳ Priority encoder using nested if else

Exercise 153

... ... case x

Exercise 154

... . casez

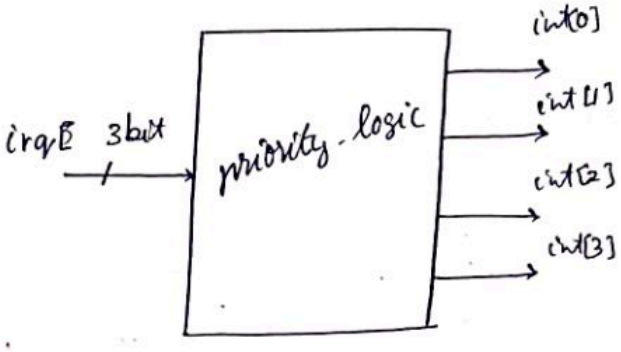
Verilog defines 3 version of case statement case, casex, & casez

casex it allows x, z, ?? to be treated as a don't care in either case expression (or) case item. But, in case x statement x propagation can cause problem problem. Hence not advised to use in synthesizable designs.

casez in case z, z& ? to be treated as don't care in either case statement or case item if 2'bIZ = 2'bI?

2'b10 = 2'bz
2'b11
2'bIx

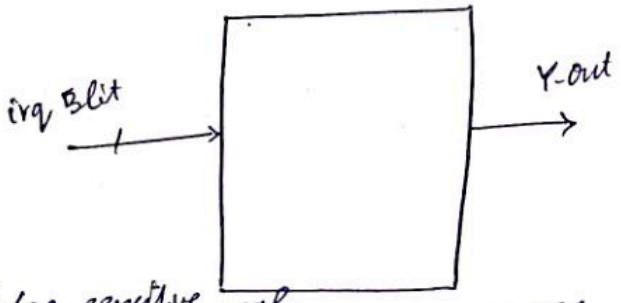
In this if more than one expression matches the case item the first matching case is taken. → By using this we can infer a priority logic.




```
casez (irq)
  3'b1?? : int[2] = 1'b1;
  3'b?1? : int[1] = 1'b1;
  3'b??0 : int[0] = 1'b1;
endcase
```

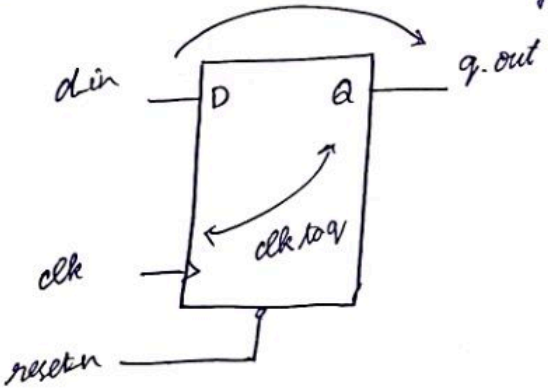
Full case & parallel case

↳ // Synonyms full_case




```
case2                          // Synonyms parallel_c
  3'b1?? : Y_out = irq[2];
  3'b?1? : Y_out = irq[1];
  3'b??1 : Y_out = irq[0];
endcase
```

12-11-2024 Edge sensitive paths dtoq full & parallel connection.




```
specify
specparam
    dtoq = 2ns;

    clktoq = 1ns;
    (din => q_o) = dtoq
    (clk => q_o) = clktoq;
endspecify
```

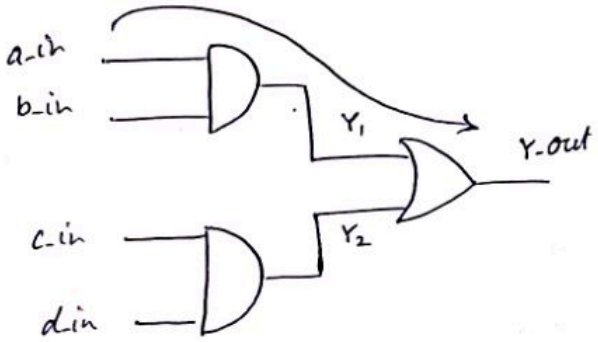
Practically we use ~~for~~ special parameter inside specify block (eg) d_in, clk, reset_n, q => here D to Q delay are specified in specify block

=> RTL design, check simulation get 100% code coverage.

Difference between $finish & $stop

Full connection & Parallel connection

(a_in => Y_out) = delay1;



Condition delay.

```
if (a_in)
  (a_in => Y_out) = 5;
if (~a_in)
  (a_in => Y_out) = 6;
```

Consider above circuit -> we can specify pin -> pin delay. Practically, depending on state of i/p signal pin -> pin delay may change. Verilog allows path delays based on condition. => are called as conditional path delays (or) state dependent path delays

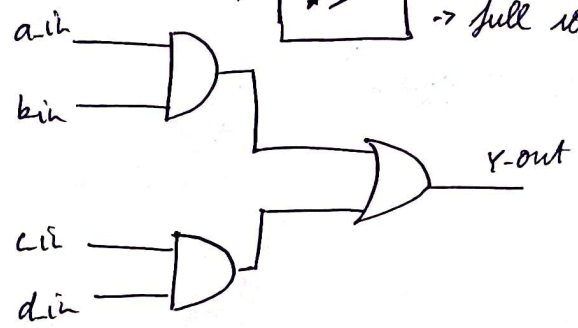
Objective to understand parallel connection for different pin to pin depending on state of i/p state.

15-11-2024

Full connection for different pin to pin timing.

↳ `*>` -> full connection operator.

(clr, pre *> q) = (tRise control, tFall cont)



// Use Specify & have pin to pin delay

```
if ({a,b} == 2'b10)
    (a,b *> Y_out) = 2;
if ({a,b} != 2'b10)
    (a, b *> Y_out) = 1;
```

// Similarly check condition

Exercise 160

Use parallel connection & full connection for pin to pin delay

a	delay
0	1
1	2

b	delay
0	2
1	3

c	delay
0	1
1	2

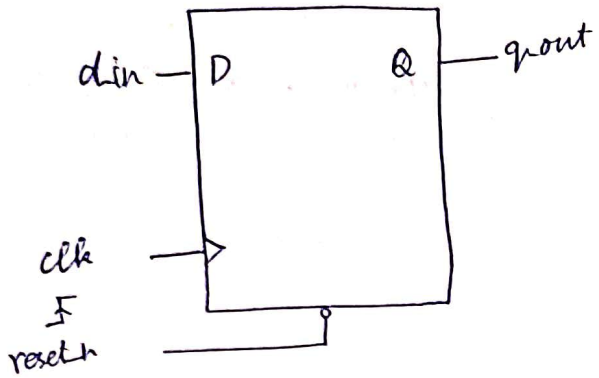
d	delay
0	1
1	2

ab	delay
01	3
other	2

cd	delay
11	4
other	3

Edge sensitive path

d_in => q_out path



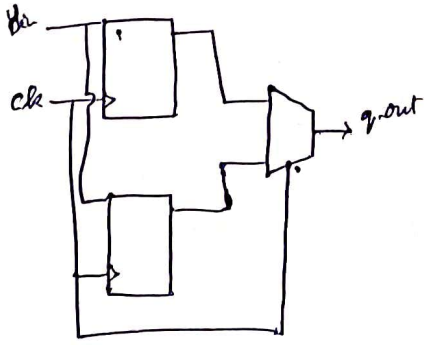

```
specify
    specparam   trise = 3 ;
    specparam   tfall = 2 ;

    (posedge clk => (q_out +: d_in)) = (trise, tfall)

endspecify
```

timing in input to o/p delay in edge, edge identifier can be used as posedge, negedge polarity +, - / parallel connection

[edge_identifier] input_pin => outputpin [polarity] : data_input)) = delay

DDR →

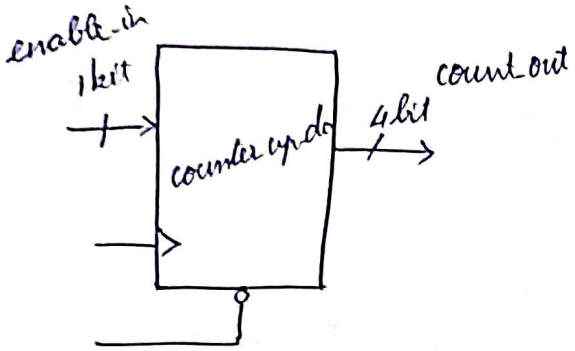
	delay (ns)
d_q	2
c_q	2



16-11-2024

DUT.UP_DOWN = 0

defparam,



UP_DOWN = 0 down counts

UP_DOWN = 1 up_counts.

reset_n	clk	enable_in	count_od
0	x	x	0 No_change
1	↓	0	operation_updown.
1	↓	1	

→ use of module name - same as in document (architecture) also of the body

→ use parameter deleration for every module name

→ dont use hardcoded values.

→ _in, _i for input

→ improve readability use comments for the required intermediate reg, wire declerations.

→ use blocking assignment (=) for combinational logic always @*

→ non-blocking assignment to be used for sequential logic which is sensitive to either posedge or negedge of block.

→ common defines have to common_define.v file.

`include that file in the required module.

-> don't mix blocking (=) & and non-blocking (<=) in single procedure block

-> during synthesis check warnings & design should not infer unintentional latches.

-> to infer priority logic use case or nested if else.

-> parallel logic use case construct.

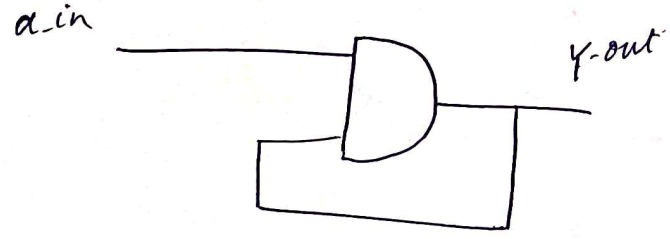
-> Avoid combinational loops or oscilatory behaviour.

-> Synthesis tool should not give warning for combinational or timing loops.

---

localparameter -> declared using localparam.

- cannot be overridden or modified in testbench, If tried to modify in testbench elabration : ERROR
- modify test bench

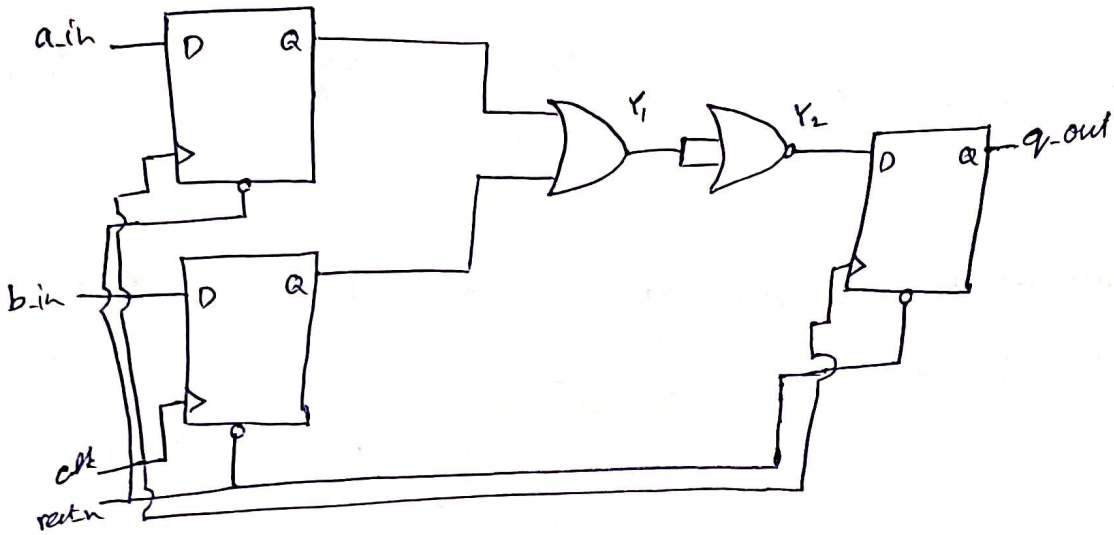


assign Y_out = a_in & Y_out;

Find f_{max} = ? t_{su} = 1ns th = 0.5 ns

t_{clock} = 1ns t_{o2} = 1 ns

t_{nand} = 1ns



VCD

'dump.vcd' → default

```
$ dumpfile ("wave.vcd");
$ dumpvars (0, tb_<..>);
```

↓ denotes depth of modules.

↓ to specify variables that are to be recorded in the file.

no arg → dump all variables

first_arg → no of hierarchy levels.

0 - all (or) (0, tb_) ⇒ dump all variable in tb_

1 - only that module, not in submodules.

$dumpoff; → pause dumping process. ↗ all selected variable will be dumped with 'x'

$dumpon;

$dumplimit (<size>); → size specified in bytes.

$dumpp_all ; ——→ current variable value
↳ create a check point

Programming language interface.

→ to define system tasks or functions, in design & simulation environment.

Role

→ to provide set of interface routines to read and write to internal data representation.

→

Task/Function tf_ | first generation PLI |

① → Used for operation user defined task / function arguments
② → utility functions
③ → call back mechanism
④ → writing data to output device.

acc_ | second generation PLI |

① → used to object oriented access directly into verilog HDL structural description
② → used to access and modify the objects in verilog design.

Vpi_ | third generation PLI |

① → There are superset of the functionality of acc & tf_ routines.

PLI structure (Verilog design & simulation)

Design & stimulus using verilog constructs	user defined system tasks

Design stimulus compilation	Internal design representation

Now internal design representation is used to run the simulation	Outputs of simulator

What PLI allows :

User defined system tasks	Links with the user defined C routine

C routine are described by standard library & PLI interface subroutine	Used to access the internal data representation

PLI library with Verilog simulator	The PLI interface allows

⇒

- Read internal data structure
- Modify internal data structure
- Access simulation environment

Application of PLI -

- Used to define additional system tasks and function (eg) debugging tasks, monitoring tasks
- Extract design information such as hierarchy, connectivity fanout, logic elements number etc.

- used to write the output display routines
- useful to have the routines that used to provide stimulus.

```
#include "veriuser.h"

int verilog_design_verification()
{
    io_printf(" ...");
}
```

Linking PLI task

$verilog_design_verification is system task →

↓ hverilog created at end

invoke PLI task

```
module tb_ ;

initial
    $verilog_design_verification;

endmodule
```

Access and utility routines

↓ tf

↓ acc

↓ object_handle,

top_handle

Handle Routine → acc_handle

Next Routine → acc_next

VCL Routine → Value, change link acc_vcl

Fetch Routine → acc_fetch

Utility Routine → acc_initialize();
↳ acc_close

Modify Routine →

Utility Routine

purpose → ◦ to get information about verilog system task invocation.

◦ argument list & values of arguments

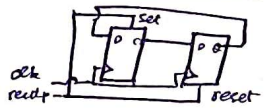
◦ used to pass new values of arguments to calling system task. Even they monitor changes in value of arguments

◦ Used to save the work ~~as~~, or points to do long arithmetic

◦ Used to display messages, halt terminate & restore simulato

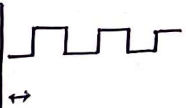
tf_ routine

Race condition



Deterministic

```
always #5 clk = ~clk;

initial begin
  clk=0;
end
```



Non-deterministic

```
always @(posedge clk (or) posedge reset_p)
  if (reset_n)
    q1_out = 0;
  else
    q1_out = q2_out;

always @(posedge clk (or) posedge reset_p)
  if (reset_n)
    q2_out = 1;
  else
    q2_out = q1_out;
```

Race condition when two or more than two statements scheduled to execute in the same simulation time step, But give different results when the order of statement execution is changed.

We know that blocking assignment block trailing assignment in the always procedural block, and are evaluted in active event queue region. Problem with event blocking occurs if RHS ~~value~~ variable of one assignment is also the LSH variable of another assignment in another procedural block, & both equation are scheduled in same simulation timestep eg; on clock edge that indicates race condition can occur if blocking assignment are not ordered properly. according to IEEE 1364 standard we have determinism. for garunteed order of execution. (ie) deterministic race condition. Non-determinism indicates statement donot have garunteed order of execution.

In, Feedback oscillator with ~~procedure~~ blocking assignment

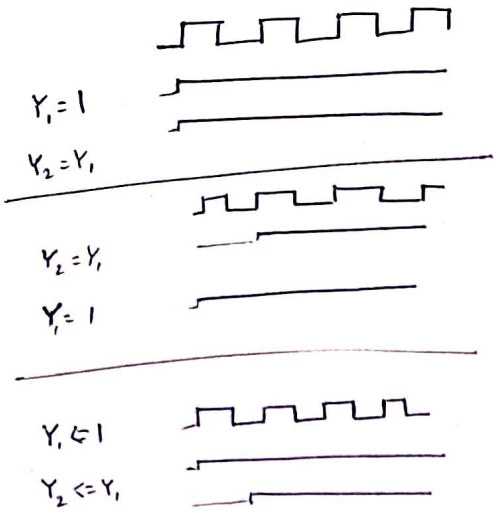
$\Longrightarrow$ NOTE -> Don't use blocking assignments in sequential clock use non-blocking assignment, as RHS of ~~nba~~ NBA in active region, LHS updated in NBA region.

The always block scheduled first garuntees correct behaviour of oscillation. so replace BA (=) by NBA (<=)

Write read or read write race condition

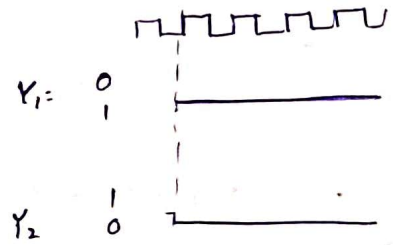
If we consider two procedural blocks sensitive to posedge of clock as shown.

```
module tb;
reg Y1, Y2;
always @(posedge clk) begin
    Y1 = 1;
end
always @(posedge clk) begin
    Y2 = Y1;
end
```

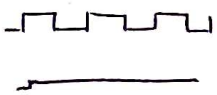


Write write race condition

```
always @(posedge clk) begin
    Y1 = 0;
end
always @(posedge clk) begin
    Y2 = Y1;
end
```



```
fork
   Y1 = 0;
   Y2 = 1;
join
```

clk 

```
fork
   Y1 = 1;
   Y2 = Y1;
join
```

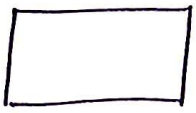
clk 

Race condition due to variable initialisation

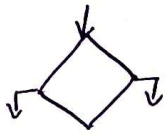
```
reg Y1, Y2;
reg clk = 0;

initial begin

    clk = 1;
    forever #5 clk = ~clk;

end
```

Algorithmic state Machine (or) Algorithmic state machine with data path

Components of ASM



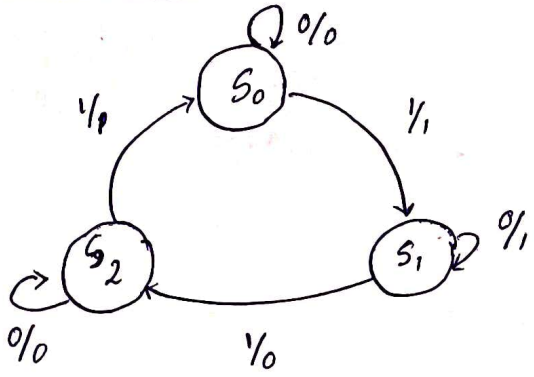
State box



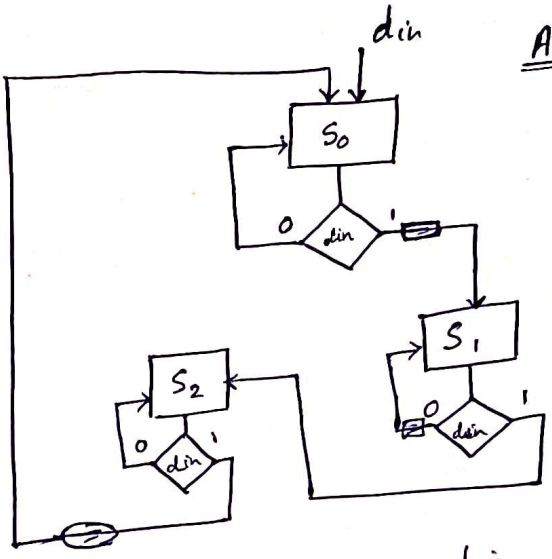
Decision box

Condition box

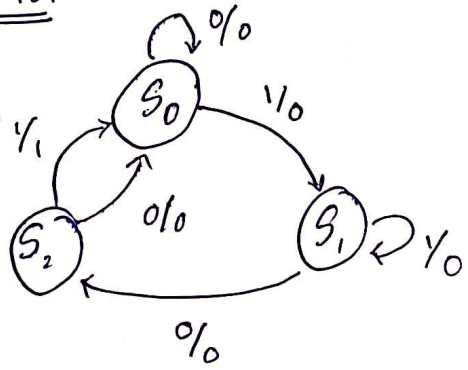
Mealy machine

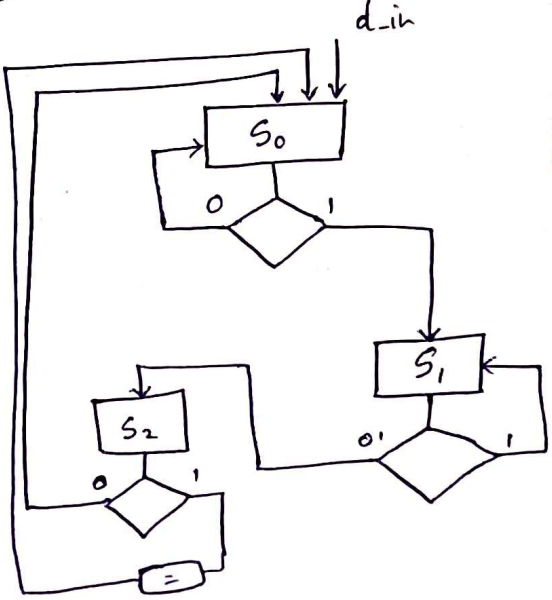


ASM chart

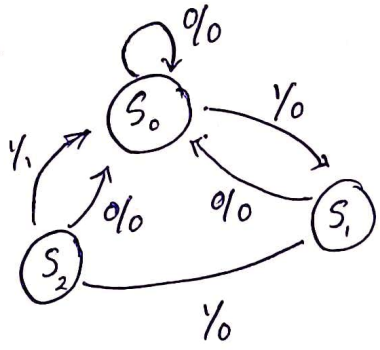


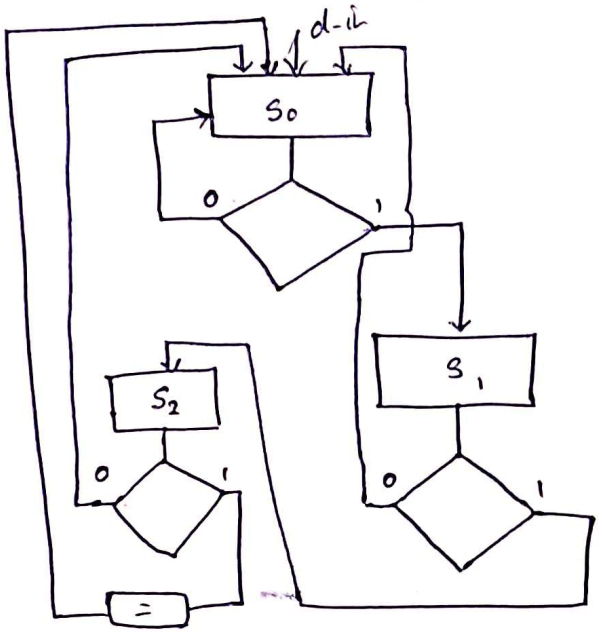
Melay 101





Mealy 111





Event control

```
@r rega = regb;                    // controlled by any change in value of r
@(negedge clock) rega = regb;      // controlled by posedge change in clock
```

Wait

```
wait (<trig>) ;
```