ECE 551
Digital Design And Synthesis

Fall '21

Coding Flops (review)
Blocking vs non-blocking
Proper Coding of Counters & Shifters
Stratified Event Queue
if/else statements

1

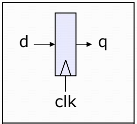
Administrative Matters

- Readings
  - Cummings SNUG Paper *(Verilog Styles that Kill) (posted on webpage)* ***Quiz on this paper*** *during class on Weds Feb 26th.*
- HW2 Due soon (Mon. Sept 28th).

2

2

Implying Flops



Standard D-FF with no reset

```
reg q;

always_ff @(posedge clk)
  q <= d;
```

It can be A vector too

```
reg [11:0] DAC_val;

always_ff @(posedge clk)
  DAC_val <= result[11:0];
```

Be careful… Yes, a non–reset flop is smaller than a reset Flop, but most of the time you need to reset your flops.

Always error on the side of reseting the flop if you are at all uncertain.

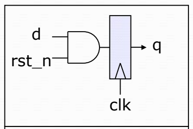
3

3

Implying Flops (synchronous reset)

```
reg q;

always_ff @(posedge clk)
  if (!rst_n)
    q <= 1'b0;      //synch reset
  else
    q <= d;
```

How does this synthesize?



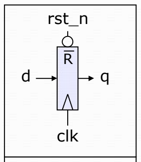
Cell library might not contain a synch reset flop. Synthesis might combine 2 standard cells

Many cell libraries don't contain synchronous reset flops. This means the synthesizer will have to combine 2 (or more) standard cell to achieve the desired function… Hmmm? Is this efficient?

4

4

Implying Flops (asynch reset)



D-FF with asynch reset

```
reg q;

always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    q <= 1'b0;
  else
    q <= d;
```

Cell libraries will contain an asynch reset flop. It is usually only slightly larger than a flop with no reset. This is probably your best bet for most flops.

Reset has its affect asynchronous from clock. What if reset is deasserting at the same time as a + clock edge? Is this the cause of a potential meta-stability issue?

Know your cell library

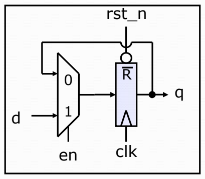
- What type of flops are available
  - + or – edge triggered (most are positive)
  - Is the asynch reset active high or active low
  - Is a synchronous reset available?
  - Do I have scan flops available?
- Code to what is available
  - You want synthesis to be able to pick the least number of cells to implement what you code.
  - If your library has active low asynch reset flops then don't code active high reset flops.

What about conditionally enabled Flops?

```
reg q;

always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    q <= 1'b0;      //asynch reset
  else if (en)
    q <= d;         //conditionally enabled
  else
    q <= q;         //keep old value
```

Dragons you will have to face though. The benefits of clock gating (particularly power benefits) are too great to ignore just because it is "hard to do". In this class we "chicken out" and use recirculating flops style.

How does this synthesize?



How about using a gated clock?

It would be lower power right?

Be careful, there be dragons here!

Flop Inference Verilog vs System Verilog

```
module simple_ff(input clk,d,rst_n,
                 output q);

reg q;

always @(posedge clk, negedge rst_n)
  if (!rst_n)
    q <= 1'b0;
  else
    q <= d;
endmodule
```

Standard verilog

In standard verilog anything assigned inside an always block must be of type **reg**.

```
module simple_ff(input clk,d,rst_n,
                 output q);

logic q;

always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    q <= 1'b0;
  else
    q <= d;
endmodule
```

System verilog

System Verilog contains a type called "**logic**" this can be used for signals that are assigned in **always** blocks or in structural or dataflow.

always_ff … no different, but Synthesis tool will warn if didn't infer a flop.

Behavioral: Combinational vs Sequential

- Combinational
  - Not edge-triggered
  - All “inputs” (RHS nets/variables) are triggers
  - Does not depend on clock
- Sequential
  - Edge-triggered by clock signal
  - Only clock (and possibly reset) appear in trigger list
  - Can include combinational logic that feeds a FF or register

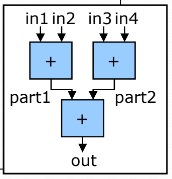
9

9

Blocking vs non-Blocking

- Blocking “Evaluated” sequentially
- Works a lot like software **(danger!)**
- Used for combinational logic

```
module addtree(output reg [9:0] out,
                   input [7:0] in1, in2, in3, in4);

reg [8:0] part1, part2;
always @(in1, in2, in3, in4) begin
        part1 = in1 + in2;
        part2 = in3 + in4;
        out = part1 + part2;
end
endmodule
```



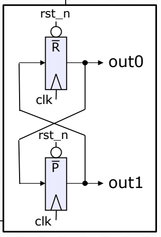
10

10

Non-Blocking Assignments

- “Updated” simultaneously if no delays given
- Used for sequential logic

```
module swap(output reg out0, out1, input rst, clk);

always @(posedge clk, negedge rst_n) begin
        if (!rst_n) begin
                out0 <= 1'b0;
                out1 <= 1'b1;
        end
        else begin
                out0 <= out1;
                out1 <= out0;
        end
end
endmodule
```



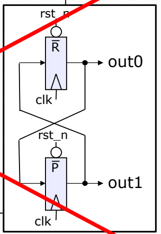
11

11

Swapping if Done With Blocking

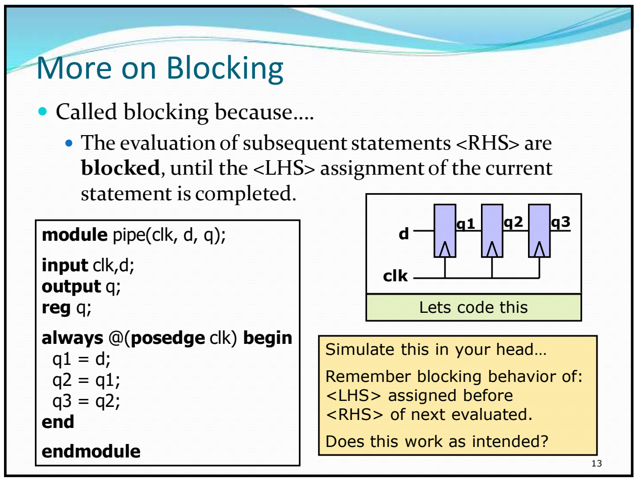
- temp signal will be required

```
module swap(output reg out0, out1, input rst, clk);
reg temp;
always @(posedge clk, negedge rst_n) begin
        if (!rst_n) begin
                out0 = 1'b0;
                out1 = 1'b1;
        end
        else begin
                temp = out0
                out0 = out1;
                out1 = temp;
        end
end
endmodule
```



12

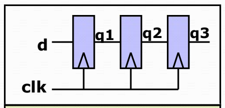
12



More on Blocking

- Called blocking because….
  - The evaluation of subsequent statements <RHS> are **blocked**, until the <LHS> assignment of the current statement is completed.

```
module pipe(clk, d, q);
input clk,d;
output q;
reg q;
always @(posedge clk) begin
  q1 = d;
  q2 = q1;
  q3 = q2;
end
endmodule
```



Lets code this

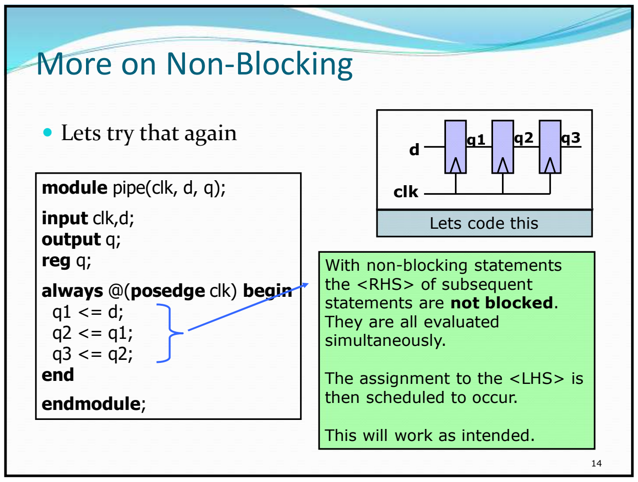
Simulate this in your head…

Remember blocking behavior of: <LHS> assigned before <RHS> of next evaluated.

Does this work as intended?

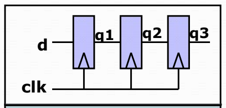
13

13



More on Non-Blocking

- Lets try that again



Lets code this

```
module pipe(clk, d, q);
input clk,d;
output q;
reg q;
always @(posedge clk) begin
  q1 <= d;
  q2 <= q1;
  q3 <= q2;
end
endmodule;
```

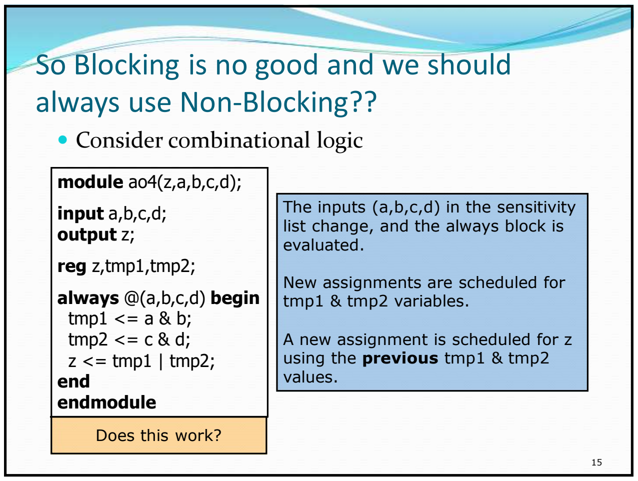
With non-blocking statements the <RHS> of subsequent statements are **not blocked**. They are all evaluated simultaneously.

The assignment to the <LHS> is then scheduled to occur.

This will work as intended.

14

14



So Blocking is no good and we should always use Non-Blocking??

- Consider combinational logic

```
module ao4(z,a,b,c,d);
input a,b,c,d;
output z;
reg z,tmp1,tmp2;
always @(a,b,c,d) begin
  tmp1 <= a & b;
  tmp2 <= c & d;
  z <= tmp1 | tmp2;
end
endmodule
```

Does this work?

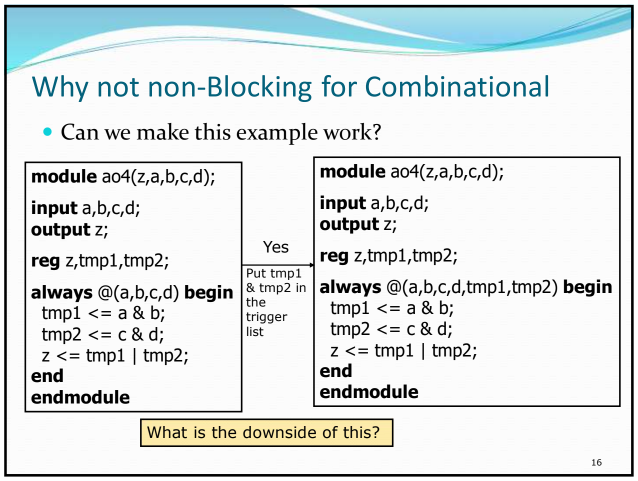
The inputs (a,b,c,d) in the sensitivity list change, and the always block is evaluated.

New assignments are scheduled for tmp1 & tmp2 variables.

A new assignment is scheduled for z using the **previous** tmp1 & tmp2 values.

15

15



Why not non-Blocking for Combinational

- Can we make this example work?

```
module ao4(z,a,b,c,d);
input a,b,c,d;
output z;
reg z,tmp1,tmp2;
always @(a,b,c,d) begin
  tmp1 <= a & b;
  tmp2 <= c & d;
  z <= tmp1 | tmp2;
end
endmodule
```

Yes

Put tmp1 & tmp2 in the trigger list

```
module ao4(z,a,b,c,d);
input a,b,c,d;
output z;
reg z,tmp1,tmp2;
always @(a,b,c,d,tmp1,tmp2) begin
  tmp1 <= a & b;
  tmp2 <= c & d;
  z <= tmp1 | tmp2;
end
endmodule
```

What is the downside of this?

16

16

Cummings SNUG Paper

- Posted on ECE551 website
  - Well written easy to understand paper
  - Describes this stuff better than I can
  - Read it!
- Outlines 8 guidelines for good Verilog coding
  - Learn them
  - Use them

17

17

end of section I of Lecture04

18

18

What Have We Learned?

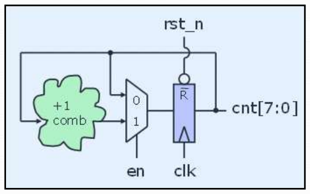
1) Sequential elements (flops & latches) should be inferred using non-blocking "**<=**" assignments
2) Combinational logic should be inferred using blocking "**=**" statements.
3) Blocking and non-Blocking statements should not be mixed in the same **always** block.
4) Plus 5 other guidelines of good coding outlined in the Cummings SNUG paper.

19

19

Engineers are paid to think, Pharmacists are paid to follow rules

- Counters are commonly needed blocks.



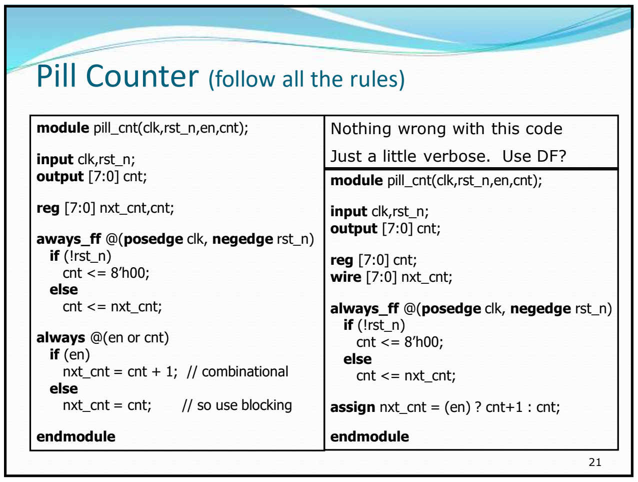
8-bit counter with reset & enable

Increment logic & mux are combinational ➔ blocking

Flop is seqential. ➔ non-blocking

20

20



Pill Counter (follow all the rules)

```
module pill_cnt(clk,rst_n,en,cnt);

input clk,rst_n;
output [7:0] cnt;

reg [7:0] nxt_cnt,cnt;

aways_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    cnt <= 8'h00;
  else
    cnt <= nxt_cnt;

always @(en or cnt)
  if (en)
    nxt_cnt = cnt + 1;  // combinational
  else
    nxt_cnt = cnt;       // so use blocking

endmodule
```

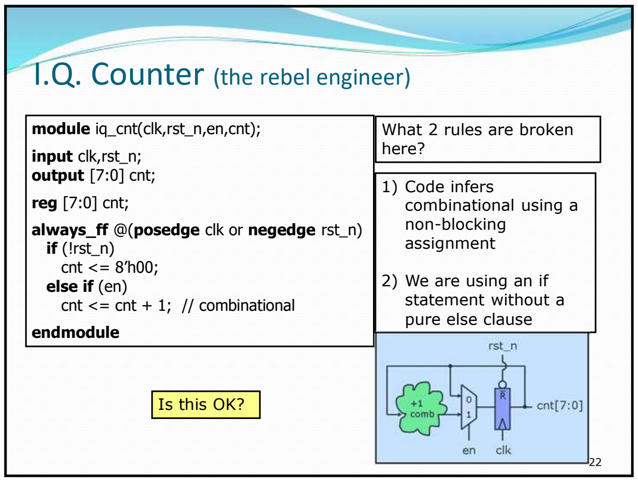
Nothing wrong with this code

Just a little verbose. Use DF?

```
module pill_cnt(clk,rst_n,en,cnt);

input clk,rst_n;
output [7:0] cnt;

reg [7:0] cnt;
wire [7:0] nxt_cnt;

always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    cnt <= 8'h00;
  else
    cnt <= nxt_cnt;

assign nxt_cnt = (en) ? cnt+1 : cnt;

endmodule
```

21

21



I.Q. Counter (the rebel engineer)

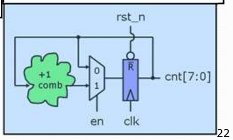
```
module iq_cnt(clk,rst_n,en,cnt);
input clk,rst_n;
output [7:0] cnt;
reg [7:0] cnt;
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n)
    cnt <= 8'h00;
  else if (en)
    cnt <= cnt + 1;  // combinational
endmodule
```

Is this OK?

What 2 rules are broken here?

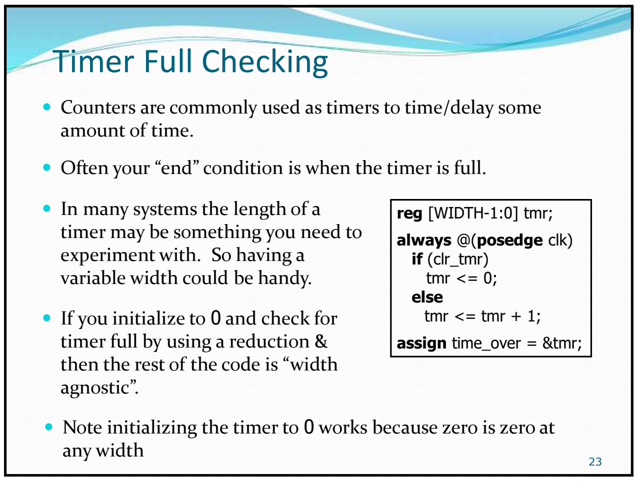
1) Code infers combinational using a non-blocking assignment

2) We are using an if statement without a pure else clause



22

22



Timer Full Checking

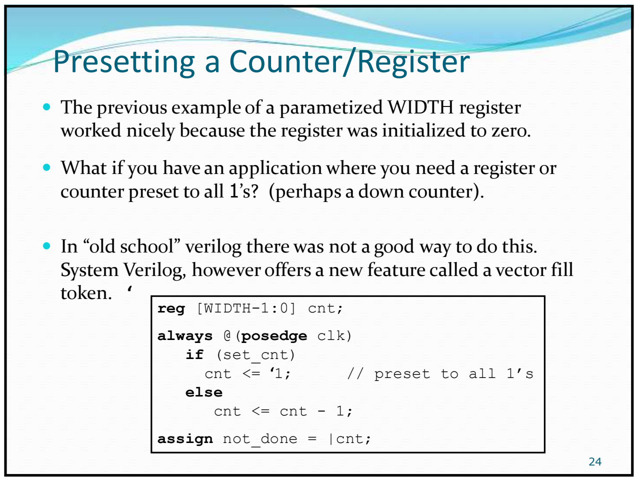
- Counters are commonly used as timers to time/delay some amount of time.
- Often your "end" condition is when the timer is full.
- In many systems the length of a timer may be something you need to experiment with. So having a variable width could be handy.
- If you initialize to 0 and check for timer full by using a reduction & then the rest of the code is "width agnostic".

```
reg [WIDTH-1:0] tmr;
always @(posedge clk)
  if (clr_tmr)
    tmr <= 0;
  else
    tmr <= tmr + 1;
assign time_over = &tmr;
```

- Note initializing the timer to 0 works because zero is zero at any width

23

23



Presetting a Counter/Register

- The previous example of a parametized WIDTH register worked nicely because the register was initialized to zero.
- What if you have an application where you need a register or counter preset to all 1's? (perhaps a down counter).
- In "old school" verilog there was not a good way to do this. System Verilog, however offers a new feature called a vector fill token. '

```
reg [WIDTH-1:0] cnt;
always @(posedge clk)
   if (set_cnt)
     cnt <= '1;       // preset to all 1's
   else
      cnt <= cnt - 1;
assign not_done = |cnt;
```

24

24

Ring Counter

```
module ring_counter (count, enable, clock, reset);
  output reg    [7: 0]   count;
  input                enable, reset, clock;

  always @  (posedge clock or posedge reset)
    if (reset == 1'b1)      count <= 8'b0000_0001;
    else if (enable == 1'b1) begin
        case (count)
           8'b0000_0001: count <= 8'b0000_0010;
           8'b0000_0010: count <= 8'b0000_0100;
           …
           8'b1000_0000: count <= 8'b0000_0001;
           default: count <= 8'bxxxx_xxxx;
        endcase
    end
endmodule
```

What do you think of this code?

25

25

Ring Counter (a better way)

```
module ring_counter (count, enable, clock, reset_n);
  output reg    [7: 0]   count;
  input                enable, reset_n, clock;

  always_ff @  (posedge clock or negedge reset_n)
    if (!reset_n)                   count <= 8'b0000_0001;
    else if (enable)                count <= {count[6:0], count[7]};
endmodule
```

- Use vector concatenation in this example to be more explicit about desired behavior/implementation
  - More concise
  - Does not rely on synthesis tool to be smart and reduce your logic for you.

26

26

Rotator

```
module rotator (Data_out, Data_in, load, clk, rst_n);
  output reg      [7: 0]    Data_out;
  input   [7: 0]     Data_in;
  input              load, clk, rst_n;

always @  (posedge clk or negedge rst_n)
     if (!rst_n)      Data_out <= 8'b0;
     else if (load)  Data_out <= Data_in;
     else if (en)     Data_out <= {Data_out[6: 0], Data_out[7]};
     else             Data_out <= Data_out
endmodule
```

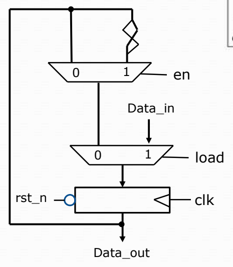
- Think what this code implies…How will it synthesize?
- What would such a block be used for?

27

27

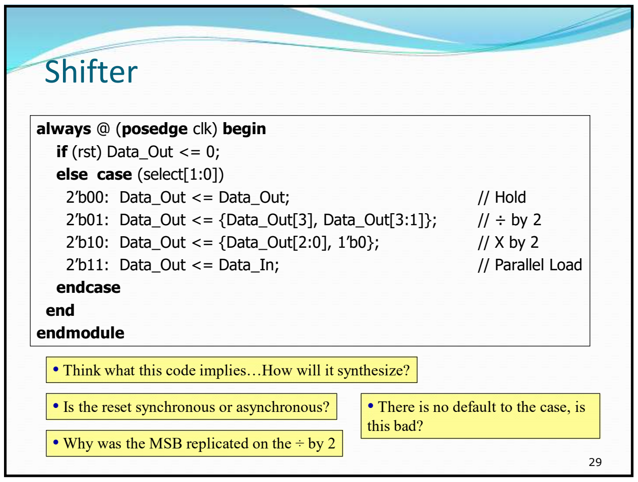
Rotator Synthesis

```
module rotator (Data_out, Data_in, load, clk, rst_n);
  output reg      [7: 0]    Data_out;
  input   [7: 0]     Data_in;
  input              load, clk, rst_n;

always @  (posedge clk or negedge rst_n)
     if (!rst_n)      Data_out <= 8'b0;
     else if (load)  Data_out <= Data_in;
     else if (en)     Data_out <= {Data_out[6: 0], Data_out[7]};
     else             Data_out <= Data_out
endmodule
```



28

28



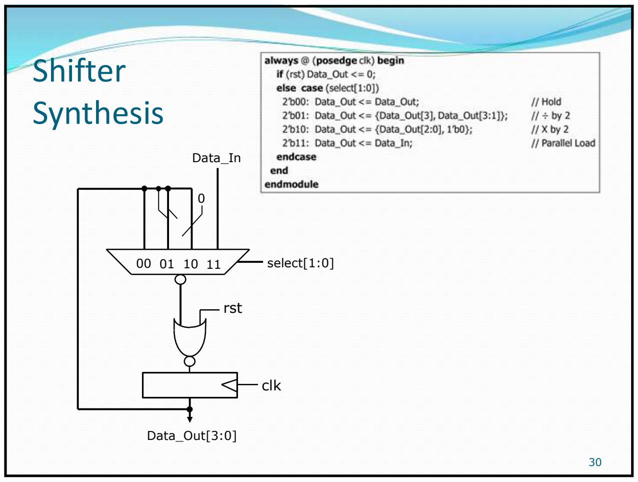
Shifter

```
always @ (posedge clk) begin
   if (rst) Data_Out <= 0;
   else  case (select[1:0])
     2'b00:  Data_Out <= Data_Out;                          // Hold
     2'b01:  Data_Out <= {Data_Out[3], Data_Out[3:1]};      // ÷ by 2
     2'b10:  Data_Out <= {Data_Out[2:0], 1'b0};             // X by 2
     2'b11:  Data_Out <= Data_In;                           // Parallel Load
   endcase
 end
endmodule
```

- Think what this code implies…How will it synthesize?
- Is the reset synchronous or asynchronous?
- Why was the MSB replicated on the ÷ by 2
- There is no default to the case, is this bad?

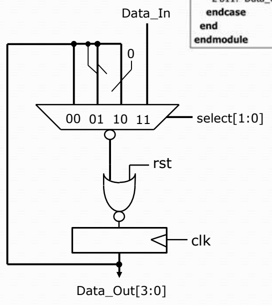
29

29



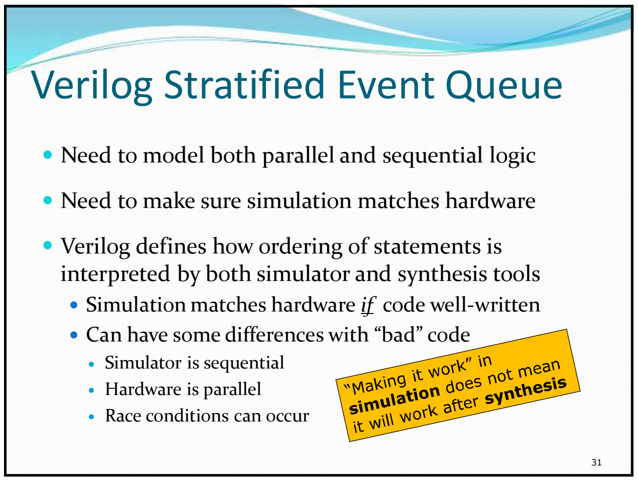
Shifter Synthesis

```
always @ (posedge clk) begin
   if (rst) Data_Out <= 0;
   else  case (select[1:0])
     2'b00:  Data_Out <= Data_Out;                          // Hold
     2'b01:  Data_Out <= {Data_Out[3], Data_Out[3:1]};      // ÷ by 2
     2'b10:  Data_Out <= {Data_Out[2:0], 1'b0};             // X by 2
     2'b11:  Data_Out <= Data_In;                           // Parallel Load
   endcase
 end
endmodule
```



30

30



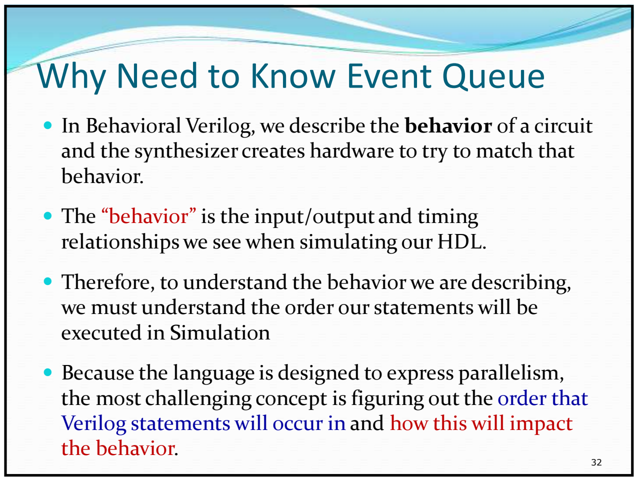
Verilog Stratified Event Queue

- Need to model both parallel and sequential logic
- Need to make sure simulation matches hardware
- Verilog defines how ordering of statements is interpreted by both simulator and synthesis tools
  - Simulation matches hardware *if* code well-written
  - Can have some differences with "bad" code
    - Simulator is sequential
    - Hardware is parallel
    - Race conditions can occur

"Making it work" in **simulation** does not mean it will work after **synthesis**

31

31

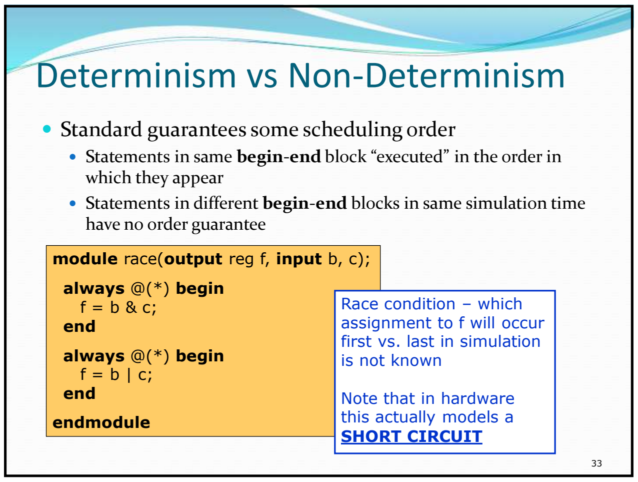


Why Need to Know Event Queue

- In Behavioral Verilog, we describe the **behavior** of a circuit and the synthesizer creates hardware to try to match that behavior.
- The "behavior" is the input/output and timing relationships we see when simulating our HDL.
- Therefore, to understand the behavior we are describing, we must understand the order our statements will be executed in Simulation
- Because the language is designed to express parallelism, the most challenging concept is figuring out the order that Verilog statements will occur in and how this will impact the behavior.

32

32



Determinism vs Non-Determinism

- Standard guarantees some scheduling order
  - Statements in same **begin**-**end** block "executed" in the order in which they appear
  - Statements in different **begin**-**end** blocks in same simulation time have no order guarantee

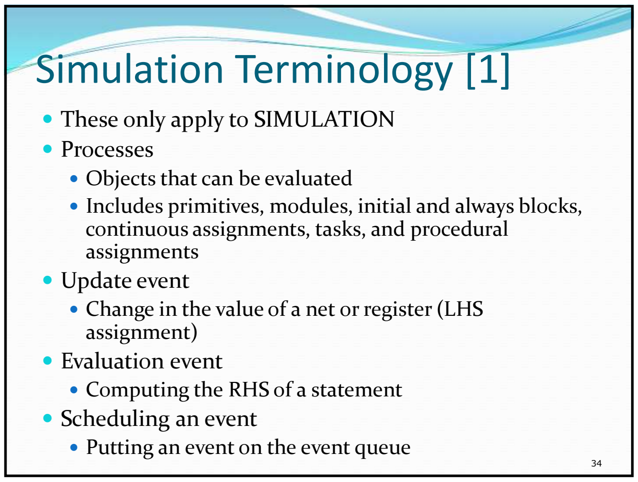
```
module race(output reg f, input b, c);
  always @(*) begin
    f = b & c;
  end
  always @(*) begin
    f = b | c;
  end
endmodule
```

Race condition – which assignment to f will occur first vs. last in simulation is not known

Note that in hardware this actually models a **SHORT CIRCUIT**

33

33

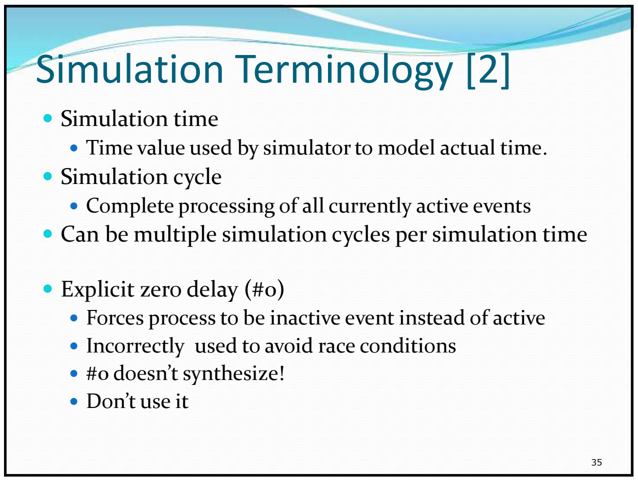


Simulation Terminology [1]

- These only apply to SIMULATION
- Processes
  - Objects that can be evaluated
  - Includes primitives, modules, initial and always blocks, continuous assignments, tasks, and procedural assignments
- Update event
  - Change in the value of a net or register (LHS assignment)
- Evaluation event
  - Computing the RHS of a statement
- Scheduling an event
  - Putting an event on the event queue

34

34



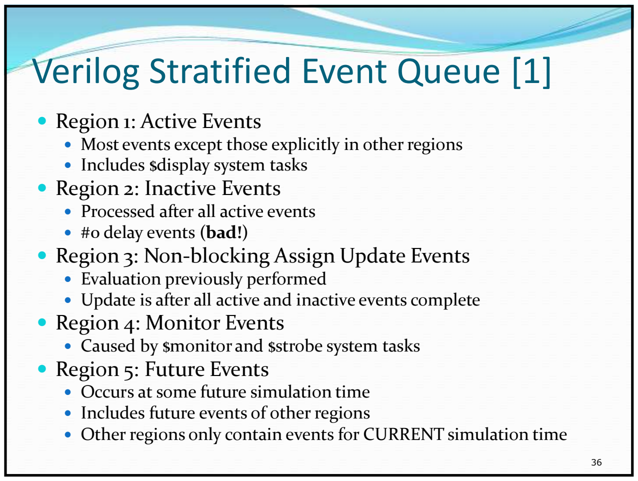
Simulation Terminology [2]

- Simulation time
  - Time value used by simulator to model actual time.
- Simulation cycle
  - Complete processing of all currently active events
- Can be multiple simulation cycles per simulation time

- Explicit zero delay (#0)
  - Forces process to be inactive event instead of active
  - Incorrectly used to avoid race conditions
  - #0 doesn't synthesize!
  - Don't use it

35

35



Verilog Stratified Event Queue [1]

- Region 1: Active Events
  - Most events except those explicitly in other regions
  - Includes $display system tasks
- Region 2: Inactive Events
  - Processed after all active events
  - #0 delay events (**bad!**)
- Region 3: Non-blocking Assign Update Events
  - Evaluation previously performed
  - Update is after all active and inactive events complete
- Region 4: Monitor Events
  - Caused by $monitor and $strobe system tasks
- Region 5: Future Events
  - Occurs at some future simulation time
  - Includes future events of other regions
  - Other regions only contain events for CURRENT simulation time

36

36

Verilog Stratified Event Queue [2]

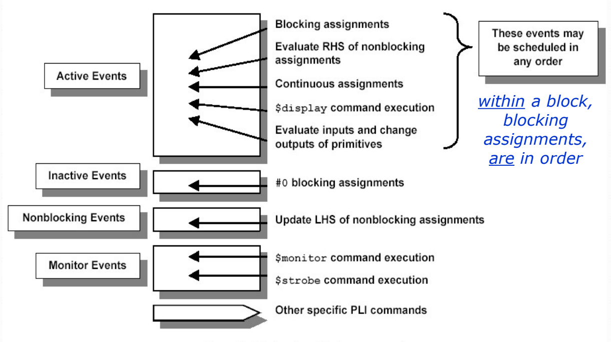

Active Events
Inactive Events
Nonblocking Events
Monitor Events
Blocking assignments
Evaluate RHS of nonblocking assignments
Continuous assignments
$display command execution
Evaluate inputs and change outputs of primitives
These events may be scheduled in any order
within a block, blocking assignments, are in order
#0 blocking assignments
Update LHS of nonblocking assignments
$monitor command execution
$strobe command execution
Other specific PLI commands

Figure 1 - Verilog "stratified event queue"

37

Simulation Model

```
Let T be current simulation time
while (there are events) {
   if (no active events) {
            if (inactive events) activate inactive events
            else if (n.b. update events) activate n.b. update events
            else if (monitor events) activate monitor events
            else {     advance T to the next event time
                       activate all future events for time T }
   }
   E = pull event from event queue
   if (E is an update event) {
            update the modified object
            add evaluation events for sensitive processes to the event queue
   } else {                              // evaluation event (of non-blocking):
            evaluate the process
            add update event(s) to the event queue
   }
}
```

38

38

Race Condition

```
assign p = q;

initial begin
        q = 1;
        #1 q = 0;
        $display(p);
end
```

- What is displayed?
- What if **$strobe**(p) were used instead?

39

39

Hand Simulation of:

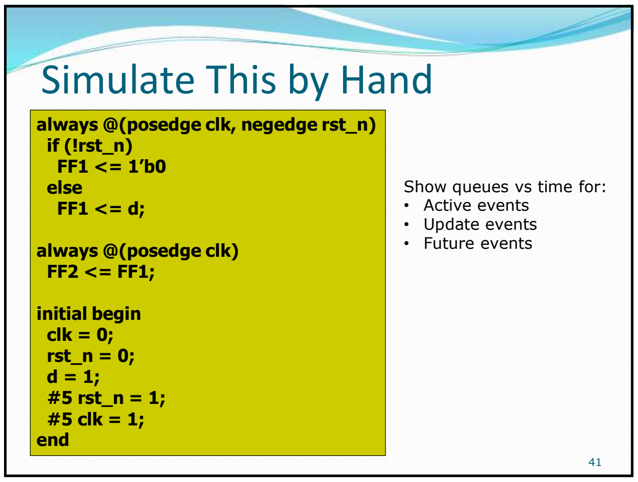
```
assign p = q;

initial begin
        q = 1;
        #1 q = 0;
        $display(p);
end
```

Active:	NB Update:	Future:
~~p=x~~		#1 q = 0,
~~q=1~~		$display(p)
~~p=1~~		
q = 0,		
$display(p)		
p = 0		

Time	Event/Action
0	p = x
0	q = 1
0	p = 1
1	q = 0
1	$display(1)
1	p = 0

40

40



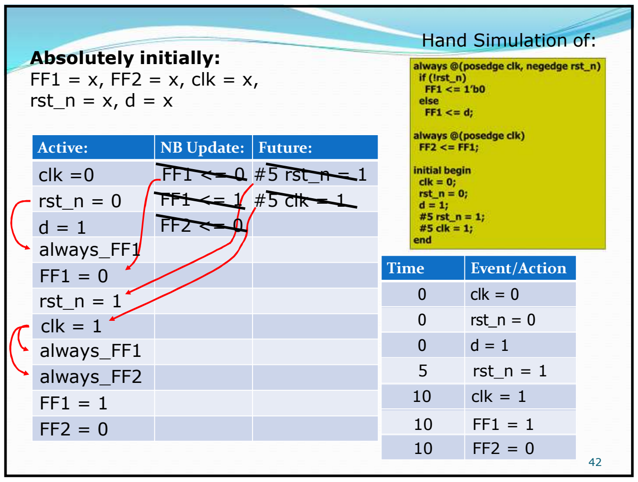
Simulate This by Hand

```
always @(posedge clk, negedge rst_n)
  if (!rst_n)
    FF1 <= 1'b0
  else
    FF1 <= d;

always @(posedge clk)
  FF2 <= FF1;

initial begin
  clk = 0;
  rst_n = 0;
  d = 1;
  #5 rst_n = 1;
  #5 clk = 1;
end
```

Show queues vs time for:
- Active events
- Update events
- Future events

41

41



Hand Simulation of:

```
always @(posedge clk, negedge rst_n)
  if (!rst_n)
    FF1 <= 1'b0
  else
    FF1 <= d;

always @(posedge clk)
  FF2 <= FF1;

initial begin
  clk = 0;
  rst_n = 0;
  d = 1;
  #5 rst_n = 1;
  #5 clk = 1;
end
```

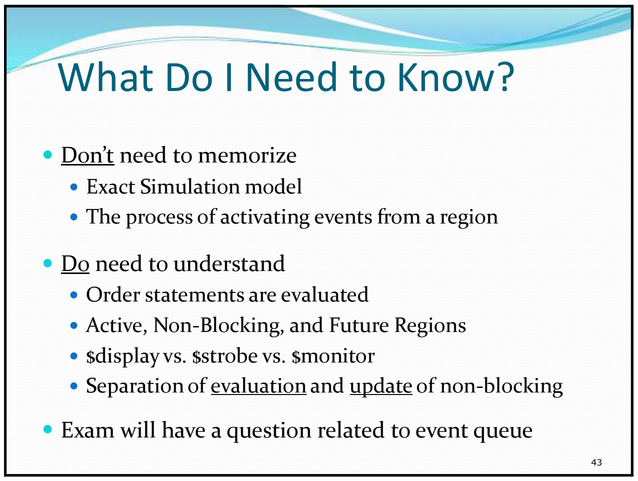
Absolutely initially:
FF1 = x, FF2 = x, clk = x,
rst_n = x, d = x

Active:	NB Update:	Future:
clk =0	~~FF1 <= 0~~	~~#5 rst_n = 1~~
rst_n = 0	~~FF1 <= 1~~	~~#5 clk = 1~~
d = 1	~~FF2 <= 0~~	
always_FF1		
FF1 = 0		
rst_n = 1		
clk = 1		
always_FF1		
always_FF2		
FF1 = 1		
FF2 = 0		

Time	Event/Action
0	clk = 0
0	rst_n = 0
0	d = 1
5	rst_n = 1
10	clk = 1
10	FF1 = 1
10	FF2 = 0

42

42

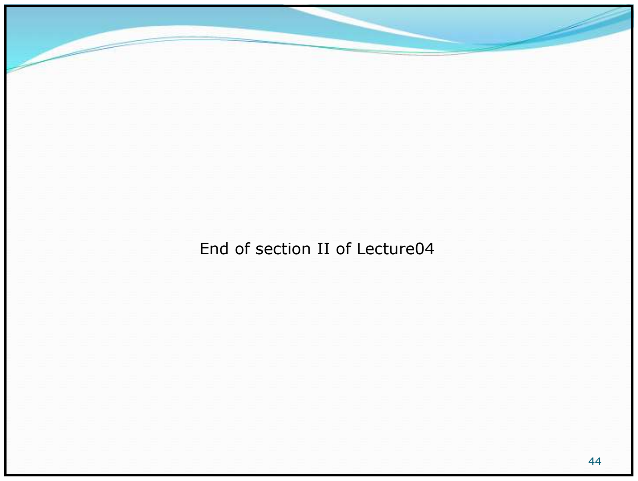


What Do I Need to Know?

- Don't need to memorize
  - Exact Simulation model
  - The process of activating events from a region
- Do need to understand
  - Order statements are evaluated
  - Active, Non-Blocking, and Future Regions
  - $display vs. $strobe vs. $monitor
  - Separation of evaluation and update of non-blocking
- Exam will have a question related to event queue

43

43



End of section II of Lecture04

44

44

if…else if…else statement

- General forms…

```
if (condition) begin
   <statement1>;
   <statement2>;
end
```

```
if (condition)
  begin
    <statement1>;
    <statement2>;
  end
else
  begin
    <statement3>;
    <statement4>;
  end
```

```
if (condition)
  begin
    <statement1>;
    <statement2>;
  end
else if (condition2)
  begin
    <statement3>;
    <statement4>;
  end
else
  begin
    <statement5>;
    <statement6>;
  end
```

Of course the compound statements formed with **begin/end** are optional.

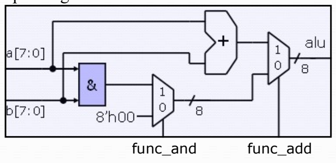
Multiple else if's can be strung along indefinitely

45

How does and **if…else if…else** statement synthesize?

- Does not conditionally "execute" block of "code"
- Does not conditionally create hardware!
- It makes a multiplexer or selecting logic
- Generally:
  - ✓Hardware for both paths is created
  - ✓Both paths "compute" simultaneously
  - ✓The result is selected depending on the condition

```
if (func_add)
   alu = a + b;
else if (func_and)
   alu = a & b;
else
   alu = 8'h00;
```

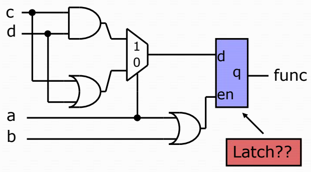


46

if statement synthesis (continued)

```
if (a)
  func = c & d;
else if (b)
  func = c | d;
```

How does this synthesize?



What you ask for is what you get!

func is of type **reg** (or **logic**). When neither **a** or **b** are asserted it didn't not get a new value.

That means it must have remained the value it was before.

That implies memory…i.e. a **latch!**

Always have an **else** to any **if** to avoid unintended latches.

47

Latch Avoidance in System Verilog

```
reg func;

always @(a,b,c,d)
  if (a)
    func = c & d;
  else if (b)
    func = c | d;
```

Traditional verilog

Produces a latch, but gives no warning (unless hdlin_check_no_latch is set true).

`func` has to be of type **`reg`** even though intent is not to infer a sequential element

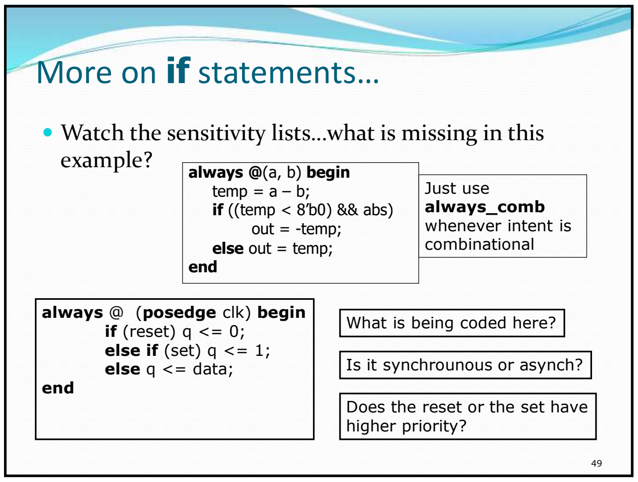
```
logic func;

always_comb
  if (a)
    func = c & d;
  else if (b)
    func = c | d;
```

System verilog

Produces a latch, but gives warning since always block said we intended combinational. No sensitivity list needed.

`func` can be of type logic, which makes more sense than **`reg`**

48



More on **if** statements…

- Watch the sensitivity lists…what is missing in this example?

```
always @(a, b) begin
    temp = a – b;
    if ((temp < 8'b0) && abs)
          out = -temp;
    else out = temp;
end
```

Just use **always_comb** whenever intent is combinational

```
always @  (posedge clk) begin
          if (reset) q <= 0;
          else if (set) q <= 1;
          else q <= data;
end
```

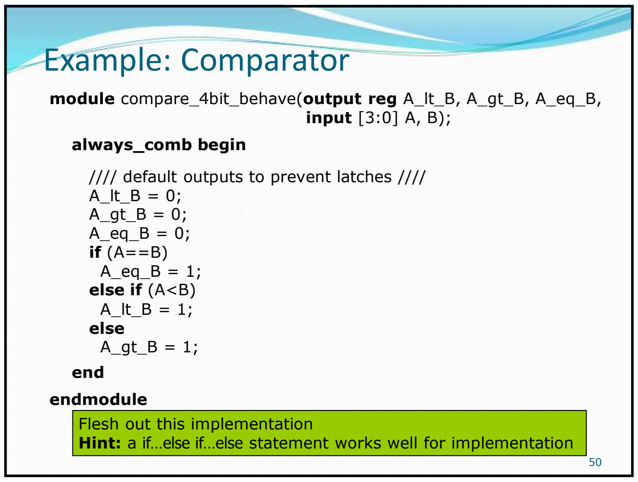
What is being coded here?

Is it synchrounous or asynch?

Does the reset or the set have higher priority?

49

49



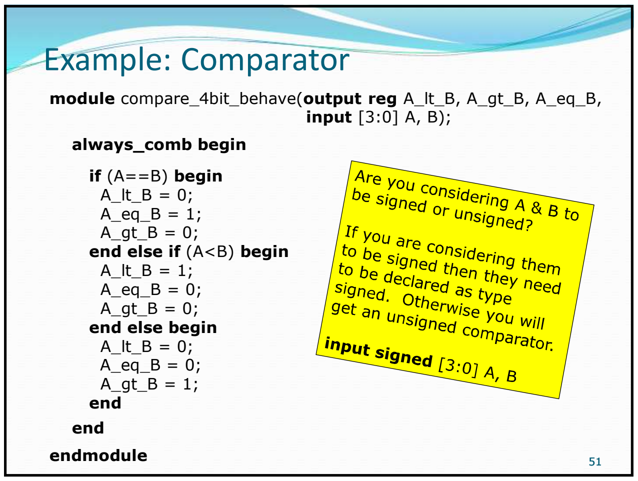
Example: Comparator

```
module compare_4bit_behave(output reg A_lt_B, A_gt_B, A_eq_B,
                           input [3:0] A, B);

  always_comb begin

    //// default outputs to prevent latches ////
    A_lt_B = 0;
    A_gt_B = 0;
    A_eq_B = 0;
    if (A==B)
      A_eq_B = 1;
    else if (A<B)
      A_lt_B = 1;
    else
      A_gt_B = 1;
  end

endmodule
```

Flesh out this implementation
Hint: a if…else if…else statement works well for implementation

50

50



Example: Comparator

```
module compare_4bit_behave(output reg A_lt_B, A_gt_B, A_eq_B,
                           input [3:0] A, B);

  always_comb begin

    if (A==B) begin
      A_lt_B = 0;
      A_eq_B = 1;
      A_gt_B = 0;
    end else if (A<B) begin
      A_lt_B = 1;
      A_eq_B = 0;
      A_gt_B = 0;
    end else begin
      A_lt_B = 0;
      A_eq_B = 0;
      A_gt_B = 1;
    end
  end

endmodule
```

Are you considering A & B to be signed or unsigned?

If you are considering them to be signed then they need to be declared as type signed. Otherwise you will get an unsigned comparator.

input signed [3:0] A, B

51

51