ET5080E Digital Design Using Verilog HDL

Fall ‘21

- Behavioral Verilog
  - always & initial blocks
  - Coding flops
  - if else & case statements
- Blocking vs Non-Blocking
- Simulator Mechanics part duo

Administrative Matters

- Readings
  - Text Chapter 7 *(Behavioral Modeling)*
  - Cummings SNUG Paper *(Verilog Styles that Kill) (posted on webpage)*

- Cover HW solution for asynch reset and tran count

Behavioral Verilog

- **initial** and **always** form basis of all behavioral Verilog
  - All other behavioral statements occur within these
  - **initial** and **always** blocks cannot be nested
  - All <LHS> assignments must be to type **reg**
- **initial** statements start at time 0 and execute once
  - If there are multiple **initial** blocks they all start at time 0 and execute independently. They may finish independently.
- If multiple behavioral statements are needed within the initial statement then the initial statement can be made compound with use of **begin/end**

More on **initial** statements

- Initial statement very useful for testbenches
- Initial statements don’t synthesize
- Don’t use them in DUT Verilog (stuff you intend to synthesize)

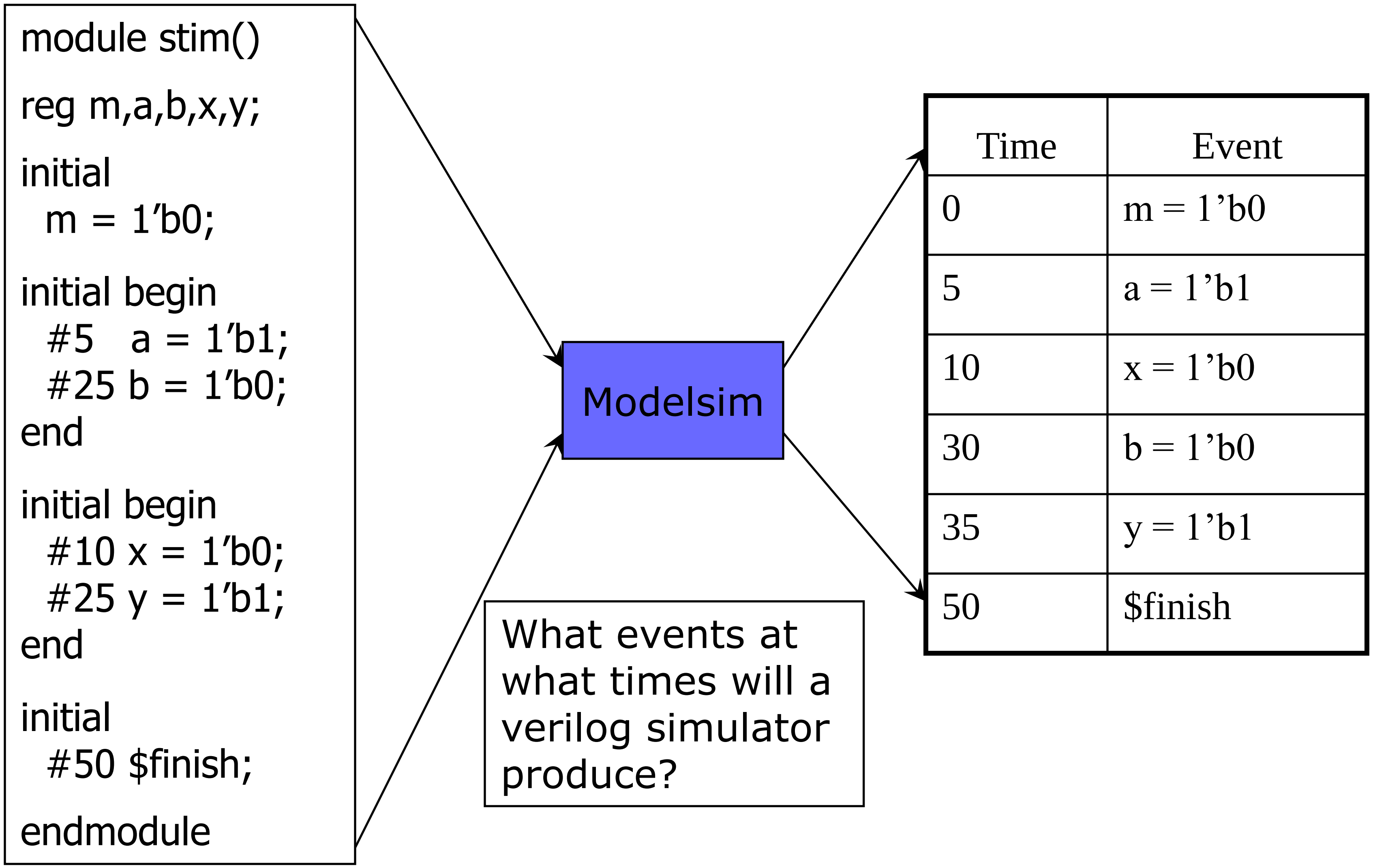
initial Blocks

```
`timescale 1 ns / 100 fs
module full_adder_tb;
    reg [3:0] stim;
    wire s, c;

    full_adder(sum, carry, stim[2], stim[1], stim[0]);        // instantiate DUT

    // monitor statement is special - only needs to be made once,
    initial $monitor("%t: s=%b c=%b stim=%b", $time, s, c, stim[2:0]);

    // tell our simulation when to stop
    initial #50 $stop;

    initial begin     // stimulus generation
        for (stim = 4'h0; stim < 4'h8; stim = stim + 1) begin
            #5;
        end
    end
endmodule
```

all initial blocks start at time 0

*multi-statement block enclosed by **begin** and **end***

Another **initial** Statement Example



```
module stim()

reg m,a,b,x,y;

initial
  m = 1'b0;

initial begin
  #5   a = 1'b1;
  #25 b = 1'b0;
end

initial begin
  #10 x = 1'b0;
  #25 y = 1'b1;
end

initial
  #50 $finish;

endmodule
```

Time	Event
0	m = 1’b0
5	a = 1’b1
10	x = 1’b0
30	b = 1’b0
35	y = 1’b1
50	$finish

What events at what times will a verilog simulator produce?

always statements

- Behavioral block operates CONTINUOUSLY
  - Executes at time zero but loops continuously
  - Can use a *trigger list* to control operation; @(a, b, c)
  - In absense of a trigger list it will re-evaluate when the last <LHS> assignment completes.

```
module clock_gen (output reg clock);

initial
  clock = 1'b0;            // must initialize in initial block

always                     // no trigger list for this always
  #10 clock = ~clock;      // always will re-evaluate when
                           // last <LHS> assignment completes
endmodule
```

always vs initial

```
reg [7:0] v1, v2, v3, v4;

initial begin
       v1 = 1;
    #2 v2 = v1 + 1;
       v3 = v2 + 1;
    #2 v4 = v3 + 1;
       v1 = v4 + 1;
    #2 v2 = v1 + 1;
       v3 = v2 + 1;
end
```

```
reg [7:0] v1, v2, v3, v4;

always begin
       v1 = 1;
    #2 v2 = v1 + 1;
       v3 = v2 + 1;
    #2 v4 = v3 + 1;
       v1 = v4 + 1;
    #2 v2 = v1 + 1;
       v3 = v2 + 1;
end
```

- **What values does each block produce?**
    - **Lets take our best guess**
    - **Then lets try it in Silos simulator**

Trigger lists (Sensitivity lists)

- Conditionally "execute" inside of **always** block
  - Any change on trigger (sensitivity) list, triggers block

```
always @(a, b, c) begin
        ...
end
```

- Original way to specify trigger list

  always @ (X1 **or** X2 **or** X3)

- In Verilog 2001 can use **,** instead of **or**

  always @ (X1, X2, X3)

- Verilog 2001 also has ***** for *combinational only*

  always @ (*)

Some mixed simulation tools in use today still do not support Verilog 2001.

Do you know your design?

Example: Comparator

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

  always@(        ) begin




  end

endmodule
```

Flush out this template with sensitivity list and implementation
Hint: a if…else if…else statement is best for implementation

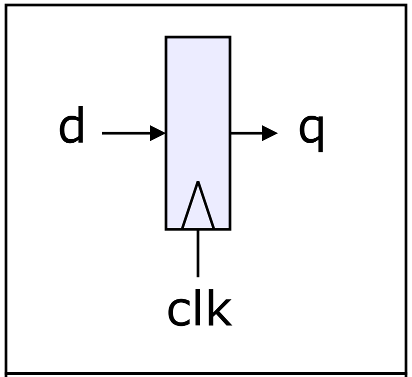
FlipFlops (finally getting somewhere)

- A negedge is on the transitions
  - 1 -> x, z, 0
  - x, z -> 0
- A posedge is on the transitions
  - 0 -> x, z, 1
  - x, z -> 1
- Used for clocked (synchronous) logic (i.e. Flops!)

```
always @ (posedge clk)
  register <= register_input;
```

Hey! What is this assignment operator?

Implying Flops (my way or the highway)



Standard D-FF with no reset

```
reg q;

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

It can be A vector too

```
reg [11:0] DAC_val;

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

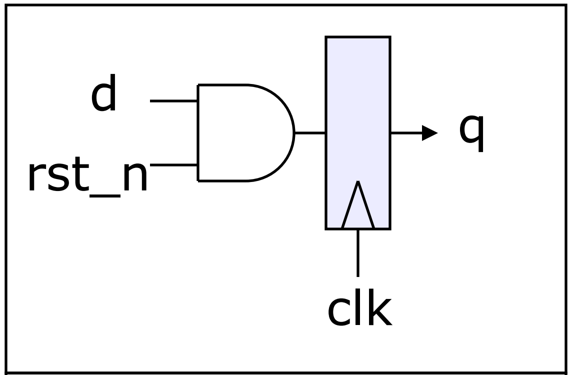
Be careful… Yes, a non−reset flop is smaller than a reset Flop, but most of the time you need to reset your flops.

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

Implying Flops (synchronous reset)

```
reg q;

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

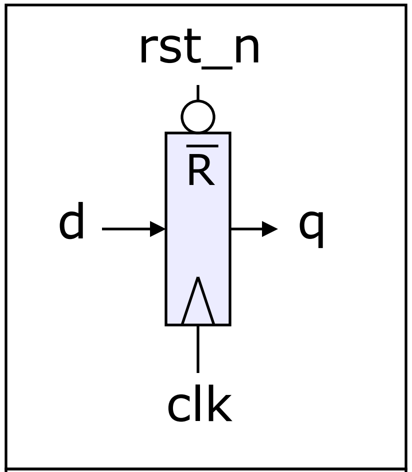
How does this synthesize?



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?

Implying Flops (asynch reset)



D-FF with asynch reset

```
reg q;

always @(posedge clk or 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?

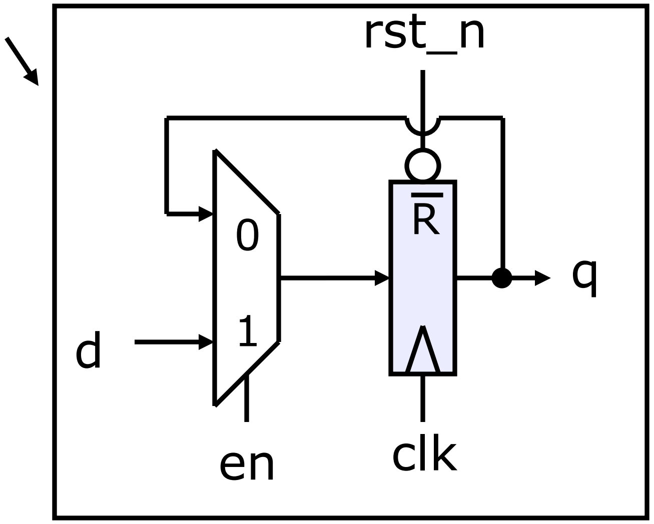
```
reg q;

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

How does this synthesize?

How about using a gated clock?

It would be lower power right?

Be careful, there be dragons here!



Behavioral: Combinational vs Sequential

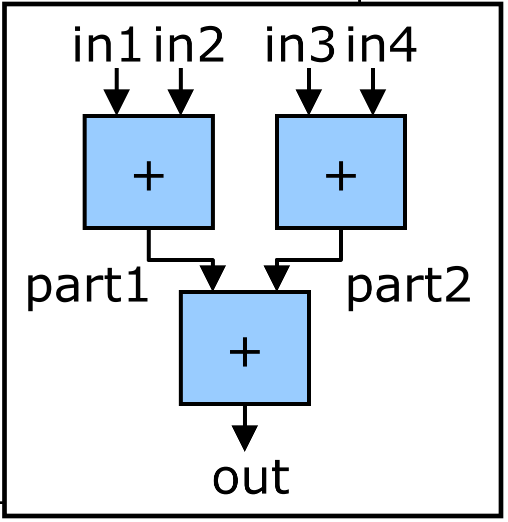
- Combinational
  - Not edge-triggered
  - All “inputs” (RHS nets/variables) are triggers
  - Does not depend on clock

- 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

Blocking vs non-Blocking

- Blocking "Evaluated" <u>sequentially</u>
- Works a lot like software **(danger!)**
- Used for <u>combinational</u> logic

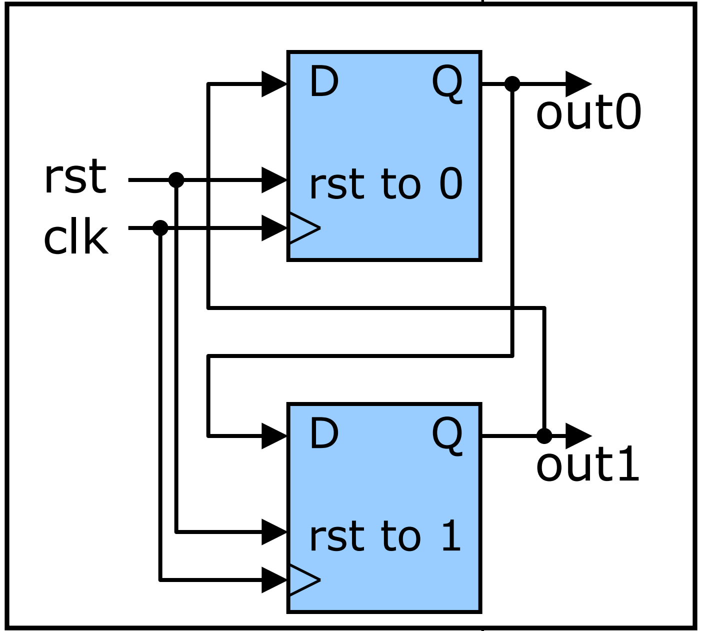
```
module addtree(output reg [9:0] out,
                           input [7:0] in1, in2, in3, in4);

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



Non-Blocking Assignments

- "Updated" simultaneously if no delays given
- Used for sequential logic

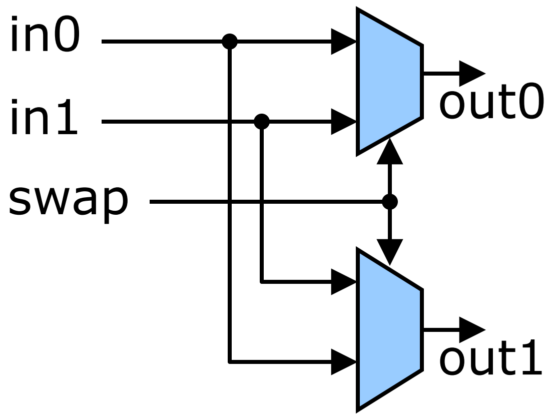
```
module swap(output reg out0, out1, input rst, clk);

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



Swapping if done in Blocking

- In blocking, need a “temp” variable

```
module swap(output reg out0, out1, input in0, in1, swap);

reg temp;
always @(*) begin
        out0 = in0;
        out1 = in1;
        if (swap) begin
                temp = out0;
                out0 = out1;
                out1 = temp;
        end
end
endmodule
```

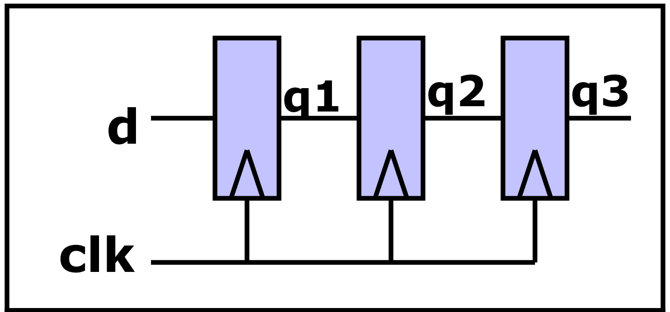


Which values get included on the sensitivity list from *?

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…

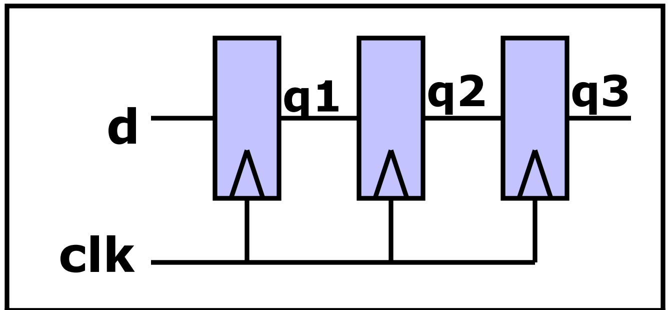
Remember blocking behavior of: <LHS> assigned before <RHS> of next evaluated.

Does this work as intended?

More on Non-Blocking

- Lets try that again

```
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

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.

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.

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?

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

Verilog Stratified Event Queue

- Need to be able 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

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

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

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

Verilog Stratified Event Queue [2]

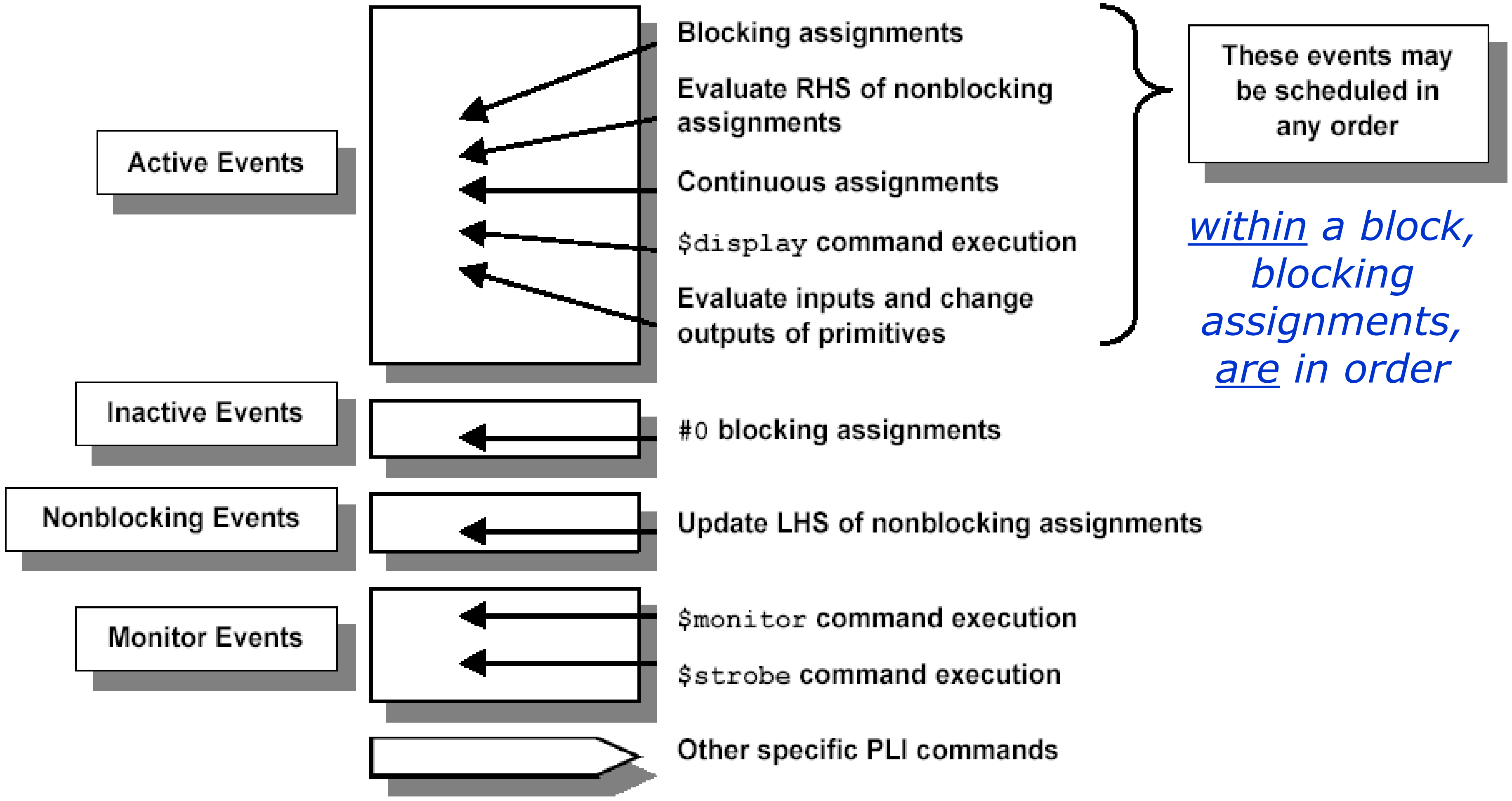

Active Events
Blocking assignments
Evaluate RHS of nonblocking assignments
Continuous assignments
$display command execution
Evaluate inputs and change outputs of primitives
These events may be scheduled in any order
Inactive Events
#0 blocking assignments
Nonblocking Events
Update LHS of nonblocking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

Figure 1 - Verilog "stratified event queue"

within a block, blocking assignments, are in order

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 = any active event;              // can cause non-determinism!
    if (E is an update event) {        // in y = a | b, the assigning of value to y
        update the modified object
        add evaluation events for sensitive processes to the event queue
    } else {   // evaluation event:    in y = a | b, the evaluation of a | b
        evaluate the process
        add update event(s) for LHS to the event queue
    }
}
```

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.

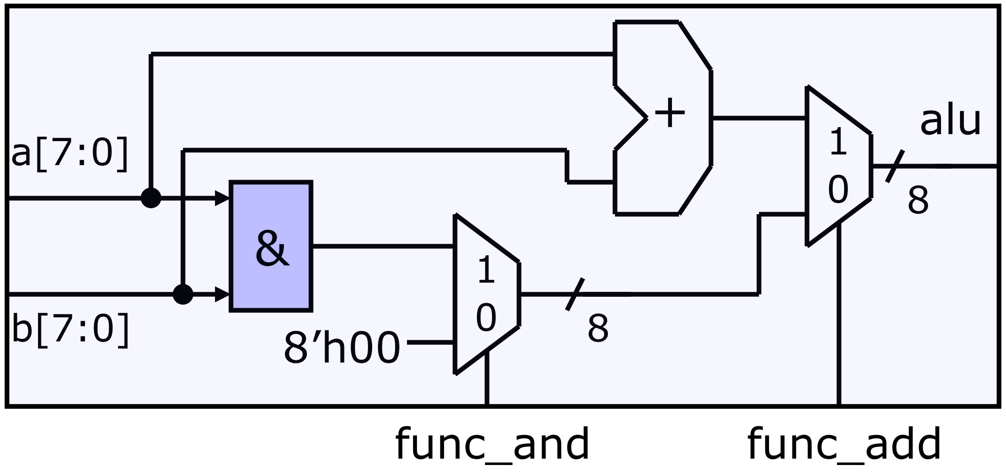
Multiple else if’s can be strung along indefinitely

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;
```

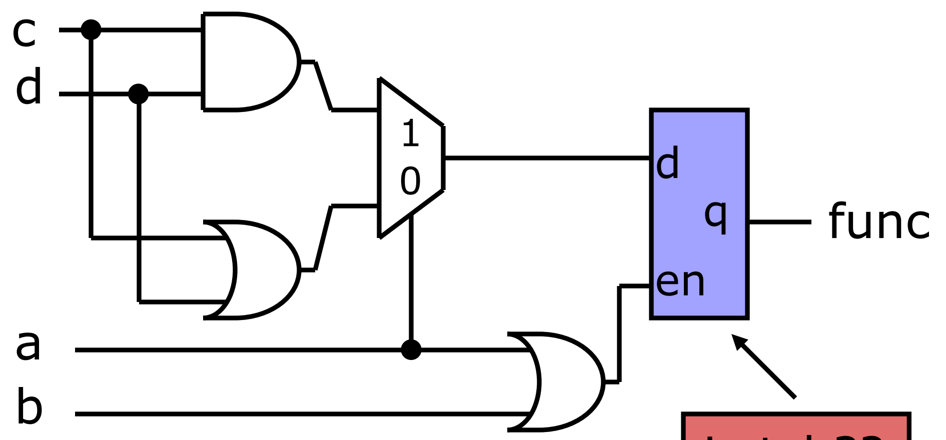
$\longrightarrow$



if statement synthesis (continued)

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

How does this synthesize?



Latch??

What you ask for is what you get!

func is of type register. 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.

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
```

```
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?

case Statements

- Verilog has three types of case statements:
  - **case**, **casex**, and **casez**
- Performs bitwise match of expression and case item
  - Both <u>must</u> have same bitwidth to match!
- **case**
  - Can detect **x** and **z**! (good for testbenches)
- **casez**
  - Uses **z** and **?** as "don't care" bits in case items and expression
- **casex**
  - Uses **x**, **z**, and **?** as "don't care" bits in case items and expression

Case statement (general form)

```
case (expression)
   alternative1 : statement1;          // any of these statements could
   alternative2 : statement2;          // be a compound statement using
   alternative3 : statement3;          // begin/end
   default : statement4                // always use default for synth stuff
endcase
```

```
parameter AND = 2'b00;
parameter OR = 2'b01;
parameter XOR = 2'b10;

case (alu_op)
   AND          : alu = src1 & src2;
   OR           : alu = src1 | src2;
   XOR          : alu = src1 ^ src2;
   default      : alu = src1 + src2;
endcase
```

Why always have a default?

Same reason as always having an else with an if statement.

All cases are specified, therefore no unintended latches.

Using **case** To Detect **x** And **z**

- Only use this functionality in a testbench!
- Example taken from Verilog-2001 standard:

```
case (sig)
        1'bz:           $display("Signal is floating.");
        1'bx:           $display("Signal is unknown.");
        default:        $display("Signal is %b.", sig);
endcase
```

casex Statement

- Uses **x**, **z**, and **?** as single-bit wildcards in case item and expression
- Uses first match encountered

```
always @ (code) begin
        casex (code)                              // case expression
                2'b0?: control = 8'b00100110; // case item1
                2'b10: control = 8'b11000010; // case item 2
                2'b11: control = 8'b00111101; // case item 3
        endcase
end
```

- What is the output for code = 2'b01?
- What is the output for code = 2'b1x?

casez Statement

- Uses z, and ? as single-bit wildcards in case item and expression

```
always @ (code) begin
    casez (code)
           2'b0?: control = 8'b00100110; // item 1
           2'bz1: control = 8'b11000010; // item 2
           default: control = 8b'xxxxxxxx; // item 3

    endcase
end
```

- What is the output for code = 2b'01?
- What is the output for code = 2b'zz?

Use of default case vs casex

- If **casex** treats x,z,? as don't care then one can create a case in which all possible alternatives are covered.
  - This would prevent unintended latches
  - Therefore default branch of case would not be needed
  - Might synthesize better?
- OK, I can accept that reasoning…but…
  - The default branch method is typically easier to read/understand
  - Sometimes it is good to have a default behavior, especially in next state assignments of state machines.