ECE 551

Digital Design And Synthesis

Fall '21

Simulator Mechanics & timing
Testbench Basics (stimulus generation)
Intro to Behavioral verilog
- Always & initial blocks
- Basic flop inference

1

Administrative Matters

- HW2 Posted and due in 2 weeks. Mon. Sept. 28th
- Watch Videos on rest of Lecture03 materials
- Friday will be in class exercise
- Monday will be a quiz on Lecture03 materials.

2

2

Analog Simulation (Spice Engine)

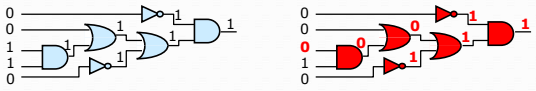
- Divide "time" into slices
- Update information in whole circuit at each slice
- Used by SPICE
- Allows detailed modeling of current and voltage
- Computationally intensive and slow
- Don't need this level of detail for most digital logic simulation

3

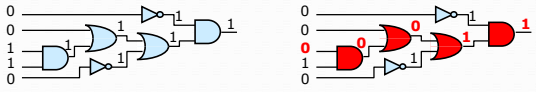
3

Digital Simulation

- Could update every signal on an input change



- Could update just the full path on input change



- Don't even need to do that much work!

4

4

Event-Driven Simulation

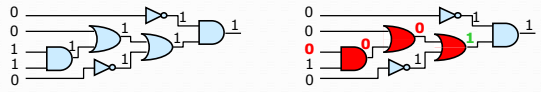
- When an input to the simulating circuit changes, put it on a "changed" list
- Loop while the "changed" list isn't empty:
  - Remove a signal from the "changed" list
  - For each sink of the signal
    - ✓Recompute its new output(s)
    - ✓For any output(s) that have changed value, add that signal to the "changed" list
- When the "changed" list is empty:
  - Keep simulation results
  - Advance simulation time to next stimulus (input) event

5

5

Simulation

- Update only if changed



- Some circuits are very large
  - Updating every signal => very slow simulation
  - Event-driven simulation is much faster!

6

6

Timing Controls For Simulation

- Can put "delays" in a Verilog design
  - Gates, wires, & behavioral statements
- Delays are useful for Simulation only!
  - Used to approximate "real" operation while simulating
  - Used to control testbench
- SYNTHESIS
  - Synthesis tool IGNORES these timing controls
    - ✓Cannot tell a gate to wait 1.5 nanoseconds
    - ✓Delay is a result of physical properties
  - Only timing (easily) controlled is on *clock-cycle* basis
    - ✓Can tell synthesizer to attempt to meet cycle-time restriction

7

7

Types Of Delays

- Inertial Delay (Gates)
  - Suppresses pulses shorter than delay amount
  - In reality, gates need to have inputs held a certain time before output is accurate
  - This models that behavior
- Transport Delay (Nets)
  - "Time of flight" from source to sink
  - Short pulses transmitted
- Not critical for our project, however, in industry
  - After APR an SDF is applied for accurate simulation
  - Then corner simulations are run to ensure design robust

8

8

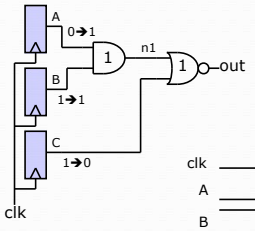
Delay Examples

- wire #5 net_1; // 5 unit transport delay
- and #4 (z_out, x_in, y_in); // 4 unit inertial delay
- assign #3 z_out = a & b; // 3 unit inertial delay
- wire #2 z_out; // 2 unit transport delay
- and #3 (z_out, x_in, y_in); // 3 for gate, 2 for wire
- wire #3 c; // 3 unit transport delay
- assign #5 c = a & b; // 5 for assign, 3 for wire

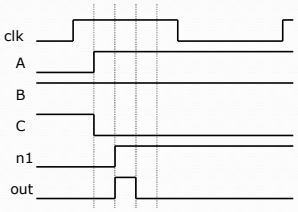
9

9

Combinational Logic will Generate Pulses



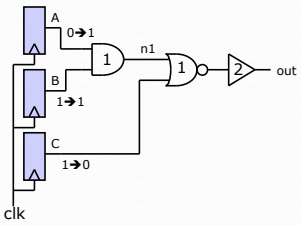
- Have 3 Flops with a unit delay clk2q
- Feeds simple combinational logic with unit gate delays
- A,B,C make the transitions shown
- What does "out" do?



10

10

Inertial Delay (tries to model gate delay)



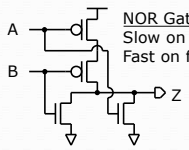
- If we add a buffer with a 2 unit delay to the end of this circuit, now what happens?
- Remember the inertial gate delay will filter any pulse shorter than the specified delay…The pulse is eaten.

11

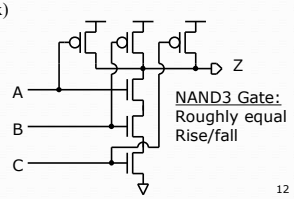
11

Gate Delay Variation (Verilog Handles This)

- Gates don't have same delay in a rising transition vs falling
- Gates delays are not constant
  - Vary with PVT (Process, Voltage, Temperature)
  - You will hear reference to PVT corners
    - ✓(fff,typ,sss) = (min,typ,max)



NOR Gate:
Slow on rise
Fast on fall

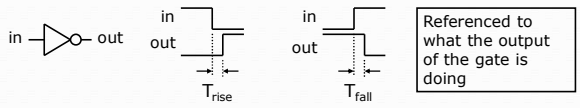


NAND3 Gate:
Roughly equal Rise/fall

12

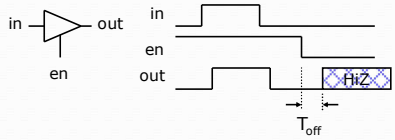
12

Rise/Fall/Turn Off



Referenced to what the output of the gate is doing

```
not #(2,1) inv1(out,in);     // not gate with 2 unit rise and 1 unit fall
```



Turn Off delay = Time from cntrl to high impedance

```
bufif1 #(1,1,2) tri1(out,in,en);    // tri-state gate with unit rise/fall and 2 unit off
```

13

13

Min/Typ/Max Delay

- Speed, Speed, Speed ....
  - If you need to ensure speed of your circuit you want to perform your worst case analysis with the longest (max) delays.
- Perhaps more dangerous is a min-delay case. Race condition. If circuit has a race that is not met, it will fail at any clock speed.
  - To ensure your circuit is immune to race conditions (i.e. clock skew) you want to perform your worst case analysis with shortest (min) delays.

14

14

Min/Typ/Max

- Verilog supports different timing sets.

```
and #(1:2:3) g1(out,a,b);          // 1 ns min, 2ns typical, 3ns max delay
```

- Can specify min/typ/max for rise and fall separate)

```
and #(2:3:4, 1:2:3) g1(out,a,b);    // gate has different rise,fall for min:typ:max
```

- Selection of timing set can is typically done in simulation environment

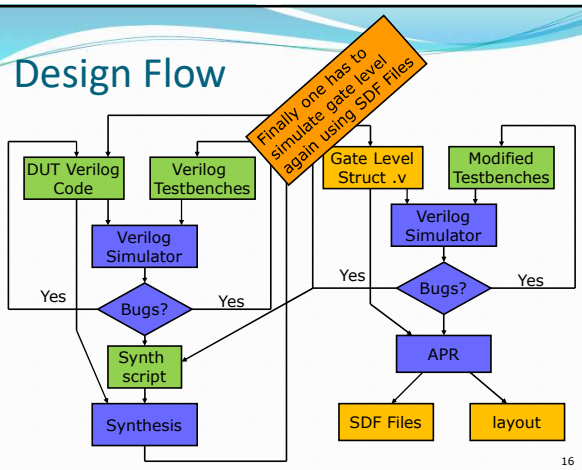
```
% verilog test.v +maxdelays
```

Invoke command line verilog engine (like Verilog XL) selecting the maxdelay time set.

15

15

Design Flow



16

16

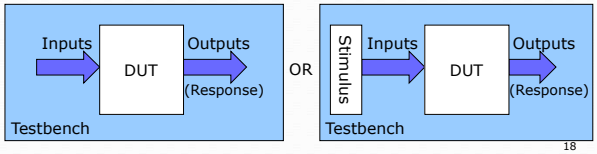
End of section I of lecture03

17

17

Testbench Basics

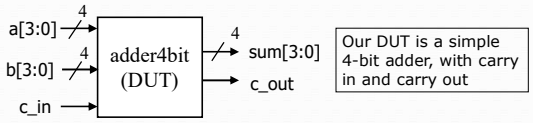
- Need to verify your design
  - "Design Under Test" (DUT)
- Use a "testbench"
  - Verilog module with no ports
  - Generates or routes inputs to the DUT
  - Outputs can be checked by human staring at waveforms or by a self-check in the testbench



18

18

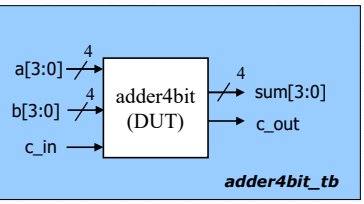
Simulation Example



Our DUT is a simple 4-bit adder, with carry in and carry out

Use a consistent naming convention for your test benches:

I usually add _tb to the end of the unit name



19

19

Simulation Example

```
`timescale 1ns /10 ps                    // time_unit/time_precision
module adder4bit_tb;
    reg[8:0] stim;                       // inputs to DUT are regs
    wire[3:0] S;                         // outputs of DUT are wires
    wire C4;

    // instantiate DUT
    adder4bit(.sum(S), .c_out(C4), .a(stim[8:5]), .b(stim[4:1]), .c(stim[0]));

    // stimulus generation
    initial begin
        stim = 9'b0000_0000_0;           // at 0 ns
        #10 stim = 9'b1111_0000_1;       // at 10 ns
        #10 stim = 9'b0000_1111_1;       // at 20 ns
        #10 stim = 9'b1111_0001_0;       // at 30 ns
        #10 stim = 9'b0001_1111_0;       // at 40 ns
        #10 $stop;                       // at 50 ns – stops simulation
    end
endmodule
```

20

20

Testbench Requirements

- Instantiate the unit being tested (DUT)
- Provide input to that unit
  - Usually a number of different input combinations!
- Watch the "results" (outputs of DUT)
  - Can watch ModelSim Wave window…
  - Can print out information to the screen or to a file
- This way of monitoring outputs (human interface) is dangerous & incomplete.
  - Subject to human error
  - Cannot be automated into batch jobs (regression suite)
  - Self checking testbenches are desired

21

21

Output Test Info

- Several different system calls to output info
  - $monitor
    - Output the given values whenever one changes
    - Can use when simulating Structural, RTL, and/or Behavioral
  - $display, $strobe
    - Output specific information like a printf in a C program
    - Used in Behavioral Verilog
- Can use formatting strings with these commands
- Only means anything in simulation
- Ignored by synthesizer

22

22

Output Format Strings

- Formatting string
  - %h, %H hex
  - %d, %D decimal
  - %o, %O octal
  - %b, %B binary
  - %t time
- $monitor("%t: %b %h %h %h %b\n",
  $time, c_out, sum, a, b, c_in);
- Can get more details from Verilog standard

23

23

Output Example

```
module adder4bit_tb;
    reg[8:0] stim;                          // inputs to DUT are regs
    wire[3:0] S;                            // outputs of DUT are wires
    wire C4;

    // instantiate DUT
    adder4bit(.sum(S), .c_out(C4), .a(stim[8:5]), .b(stim[4:1]), .c(stim[0]));

    initial $monitor("%t A:%h B:%h ci:%b Sum:%h co:%b\n",$time,
                     stim[8:5],stim[4:1],stim[0],C4,S);
// stimulus generation
    initial begin
          stim = 9'b0000_0000_0;            // at 0 ns
          #10 stim = 9'b1111_0000_1;        // at 10 ns
          #10 stim = 9'b0000_1111_1;        // at 20 ns
          #10 stim = 9'b1111_0001_0;        // at 30 ns
          #10 stim = 9'b0001_1111_0;        // at 40 ns
          #10 $stop;                        // at 50 ns – stops simulation
    end
endmodule
```

24

24

Generating Clocks

- Wrong way:

```
initial begin
   #5 clk = 0;
   #5 clk = 1;
   #5 clk = 0;
   … (repeat hundreds of times)
end
```

- Right way:

```
initial clk = 0;

always
  #5 clk = ~clk;
```

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

25

25

Exhaustive Testing

- Practical for combinational designs w/ up to 8 or 9 inputs
  - Test ALL combinations of inputs to verify output
  - Could enumerate all test vectors, but don't…
  - Generate them using a "for" loop!

```
logic [4:0] x;
initial begin
      for (x = 0; x <  16; x = x + 1)
            #5;    // need a delay here!
end
```

- Need to use "**reg** or **logic**" type for loop variable? Why?

26

26

Why Loop Vector Has Extra Bit

- Want to test all vectors 0000 to 1111

```
logic [3:0] x;
initial begin
      for (x = 0; x <  16; x = x + 1)
            #5;   // need a delay here!
end
```

- If **x** is 4 bits, it only gets up to 1111 => 15
  - 1100 => 1101 => 1110 => 1111 => 0000 => 0001
- x is never >= 16… so loop goes forever

27

27

Example: DUT

```
module Comp_4 (A_gt_B, A_lt_B, A_eq_B, A, B);
output  A_gt_B, A_lt_B, A_eq_B;
input   [3:0] A, B;
   // Code to compare A to B
   // and set A_gt_B, A_lt_B, A_eq_B accordingly
endmodule
```

28

28

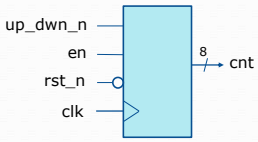
Example: Testbench

```
module Comp_4_tb();
  wire   A_gt_B, A_lt_B, A_eq_B;
  logic  [4:0] A, B;                    // sized to prevent loop wrap around

  Comp_4 M1 (A_gt_B, A_lt_B, A_eq_B, A[3:0], B[3:0]);          // DUT

  initial $monitor("%t  A: %h  B: %h  AgtB: %b  AltB: %b  AeqB:
    %b", $time, A[3:0], B[3:0], A_gt_B, A_lt_B, A_eq_B);

  initial #2000 $finish;                // end simulation, quit program

  initial begin
    for (A = 0; A < 16; A = A + 1) begin          // exhaustive test of inputs
      for (B = 0; B < 16; B = B + 1) begin #5;  // may want to test x's and z's
      end // first for
    end // second for
  end // initial

endmodule
```

29

29

Stimulus for Sequential Circuits



- Imagine an 8-bit counter with an active low reset signal.
- When en=0 the count is frozen
- The counter counts up if up_dwn_n=1 or down if up_dwn_n=0

- What should the testbench look like?
- How should the stimulus be generated?

30

30

Stimulus for Sequential Circuits

```
module up_dwn_tb();

logic clk, rst_n;
logic en, up_dwn_n;

// Instantiate DUT //
up_dwn iDUT(.clk(clk), .rst_n(rst_n), .en(en), .up_dw

initial begin
  clk = 0;
  rst_n = 0;        // assert reset
  en = 0;           // start with it disabled
  up_dwn_n = 1;     // count up at first
  #16 rst_n = 1;    // deassert clock
  #5 en = 1;        // start counting
  #50 en = 0;       // stop after 5 clocks
  #10 en = 1;       // start after 1 clock freeze
  #20 up_dwn_n = 0; // start counting down
end

always
  #5 clk = ~clk;

endmodule
```

- Can do it using delays as shown here.
- Signal timings are calculated based on your clock period
- Figuring it out is a pain
- What if you want to change the clock period?
- There is a better way

31

31

Stimulus for Sequential Circuits

```
module up_dwn_tb();

logic clk, rst_n;
logic en, up_dwn_n;

// Instantiate DUT //
up_dwn iDUT(.clk(clk), .rst_n(rst_n), .en(en), .up_dwn_

initial begin
  clk = 0;
  rst_n = 0;                // assert reset
  en = 0;                   // start with it disabled
  up_dwn_n = 1;             // count up at first
  @(posedge clk);           // wait one clock cycle
  @(negedge clk) rst_n = 1; // deassert reset on negati
  @(posedge clk) en= 1;     // start counting
  repeat (5)@(posedge clk); // wait 5 clock cycles
  en = 0;                   // stop counting
  @(posedge clk) en = 1;    // for 1 clock cycle
  repeat(2) @(posedge clk); // wait 2 clock cycles
  up_dwn_n = 0;             // start counting backwards
end

always
  #5 clk = ~clk;
```

- Can use clock edges as trigger for stimulus events
- Easier to think about, no arithmetic with clock period
- Code is independent of clock period
- Can use repeat() loop to wait a given number of clocks

32

32

Force/Release In Testbenches

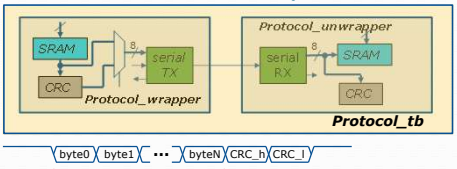
- Allows you to "override" value FOR SIMULATION
- Doesn't do anything in "real life"
- How does this help testing?
  - Can create a "short cut" to a state that is difficult to get to through normal paths.
    - Force a state in a SM that would take many sequences to get to through "normal operation"
    - Force an error condition in a protocol if you wanted to test how your error correction mechanism worked.
- Can help achieve code coverage *(more on that later)*

Force/Release Example

```
assign y = a & b;
assign z = y | c;
initial begin
  a = 0; b = 0; c = 0;
  #5 a = 0; b = 1; c = 0;
  #5 force y = 1;
  #5 b = 0;
  #5 release y;
  #5 $stop;
end
```

T	a	b	c	y	z
0	0	0	0	0	0
5	0	1	0	0	0
10	0	1	0	1	1
15	0	0	0	1	1
20	0	0	0	0	0

Force/Release Example



16-bit CRC formed from N bytes of payload

A protocol has to transfer a chunk of data in an SRAM via a serial protocol. There is potential for bit corruption in the serial protocol, so data integrity is checked via a CRC

Given the structure of this testbench how do you check the protocol unwrapper's behavior to a CRC mismatch?

One possibility would be to use force/release to force the Protocol wrapper to "corrupt" a few bits of the CRC transmitted.

End of section II of Lecture03

Behavioral Verilog

- Instead of describing what the hardware looks like, describe what you want the hardware to do
- **Goal:** Abstract away the details of the hardware implementation to make design easier
- The synthesizer creates a hardware structure that does the same thing as your description
  - … but the synthesizer has to be able to realize your description using real hardware constraints
  - This is why not all Verilog constructs are supported
  - You still need to "think hardware" then describe it using verilog

37

37

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** or **logic**
- **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**

38

38

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)
  - If you are tempted to use **initial** blocks in DUT code it means you are not resetting flops you need to reset.
  - All state is held in flops and the reset of your logic occurs through resetting flops. The reset condition then flows through the combinational logic

39

39

initial Blocks

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

    full_adder iDUT(stim, 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

This is not a good example:
- *Was there a need for 3 initial blocks?*
- *Are these begin/end pairs necessary?*

40

40

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
```

Modelsim

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?

41

41

always statements

- Behavioral block operates CONTINUOUSLY (***always***)
  - 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
```

42

42

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?**

43

43

Trigger lists (Sensitivity lists)

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

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

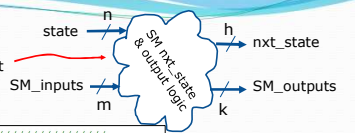
- Original way to specify trigger list
  always @ (X1 **or** X2 **or** X3)
- In Verilog 2001 can use **,** instead of **or**
  always @ (X1, X2, X3)
- Verilog 2001 also has * for *combinational only*
  always @ (*)
- System Verilog introduced the **always_comb**

44

44

always_comb

SM state transition logic & output logic are purely combinational




```
////////////////////////////////////////////
// Infer state transition logic & SM outputs //
////////////////////////////////////////////
always_comb begin
  ////////////////////////////////////
  // Default outputs and nxt_state logic //
  ////////////////////////////////////
  nxt_state = state;
  capture_cmd = 0;
  nxt_cmd = 0;
  go = 1;

  case (state)
    STOP: begin
      go = 0;              // only state we are not going
      if (cmd_rdy & line_present) begin
      // if (line_present) begin
        capture_cmd = 1;
        nxt_state = GOING;
      end
    end
    GOING : begin
      if (!line_present) begin
```

always_comb block works nice for SM transition logic.

NOTE: always_comb does not have a sensitivity list

NOTE: can use a **case** statement and **if** statements inside an always block

45

45

FlipFlops (now we're 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!)

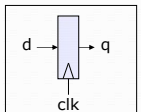
```
always @ (posedge clk)
  register <= register_input;
```

Hey! What is this assignment operator?

46

46

Implying Flops (my way or the highway)



Standard D-FF with no reset

It can be A vector too

```
reg q;

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

```
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.

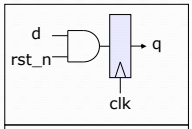
47

47

Implying Flops (synchronous reset)

```
reg q;

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

How does this synthesize?



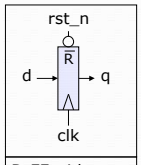
Cell library might not contain a synch reset flop. Synthesis might combine 2 standard cells

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?

48

48

Implying Flops (asynch reset)



D-FF with asynch reset

```
reg q;

always @(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?

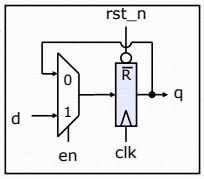
49

49

What about conditionally enabled Flops?

```
reg q;

always @(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
```

How does this synthesize?



50

50

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

51

51