ECE 551

Digital Design And Synthesis

Fall '21

case statements
Proper SM Coding
Random Misc Stuff

1

Administrative Matters

- Cummings paramdesign paper for hdlcon (posted on class website)
- HW3 Posted ***(a difficult one)***
- Midterm I
  - Weds Oct 7th @ classtime.

2

2

case Statements

- System Verilog has four types of case statements:
  - **case**, **casex**, **casez,** & **case** () **inside**
- Performs bitwise match of expression and case item
  - Both must have same bitwidth to match!
- **case**
  - Can detect **x** and **z**! (good for testbenches & when no wildcards needed)
- **casez & casex**
  - Used x, z, ? As wild cards, but in both case item & case expression. made them error prone, obsoleted by **case () inside**.
- **case** () **inside**
  - Uses **x**, **z**, and **?** as "don't care" bits (in case items only). This is good for decoding instructions where some bits are don't care.

3

3

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
```

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

always_comb
 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.

4

4

case statement (improved example)

```
typedef enum logic [1:0] {AND, OR, XOR, ADD} alu_op_t;

alu_op_t alu_op_i;
alu_op_i = alu_op_t'(alu_op);

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

If **alu_op** was a port/signal of type alu_op_t already, then this casting to intermediate **alu_op_i** would not be necessary.

Use of enum type makes debug easier because waveforms display as names (AND,OR,XOR), not encoded values.

5

Using case To Detect x And z

- Only use this functionality in a testbench!

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


6

casex Statement

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

```
always @(op_code) begin
        casex (op_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
                default: $display("opcode invalid");
        endcase
end
```

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

7

```
always_comb begin
   casex (case_exp)
      3'b0??: $display("case1");
      3'b10?: $display("case2");
      3'b111: $display("case3");
     default: $display("default case");
   endcase
end
```

```
always_comb begin
   case (case_exp) inside
      3'b0??: $display("case1");
      3'b10?: $display("case2");
      3'b111: $display("case3");
     default: $display("default case");
   endcase
end
```


8

case () inside Statement

- Uses x, z, and ? as single-bit wildcards in **case item only**

```
always_comb begin
   case (thumb_opcode) inside
         6'h001???: begin        // thumb LD/S[illegible]irect+imm
           ...
         6'b00????: begin        // th[illegible] immediate instructions
           ...
         6'b0101??: begin        // thumb LD/ST indirect
         ...
   endcase
end
```

Might be good to remember this if you take 552 or 554

- **case () inside** really shines when performing instruction decoding for most processor ISA's.
- This example shows a possible decode of a few ARM thumb opcodes.

9

9

Use of Range with case () inside

```
always_comb
   case (graduation_range) inside
         [1950:1959] : $display("Did you kiss at the drive in?");
         [1960:1969] : $display("What was tripping on LST like?");
         [1970:1979] : $display("Are you sad disco died?");
         [1980:1989] : $display("How big was your hair back then?");
         [1990:1999] : $display("Were you hip hop or grunge?");
   endcase
```

- This range checking is synthesizeable and could also be useful in decode applications.

10

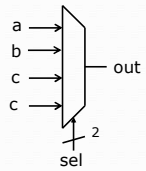
10

How Does a case(x) Statement Synthesize?

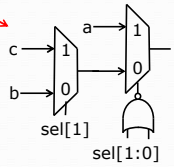
Remember the First Match is taken
(there is an inferred priority)

```
always_comb
  case (sel)
    2'b00 : out = a;
    2'b01 : out = b;
    default : out = c;
  endcase
```

Synthesis Tool



or this?



Depends on several factors including some synthesis directives. However the 2nd case is what we expect it to do without any directives. System verilog gives us more control.

11

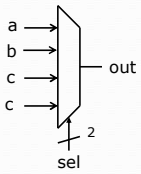
11

System Verilog Case Options

unique means one and only one case will match. So can synthesize it flat

```
always_comb
  unique case (sel)
    2'b00 : out = a;
    2'b01 : out = b;
    default : out = c;
  endcase
```

Synthesis Tool




```
always_comb begin
  nxt_state = state1hot;
  unique case (1'b1)
    state1hot[0] : nxt_state = 3'b010;
    state1hot[1] : nxt_state = 3'b100;
    state1hot[2] : nxt_state = 3'b001;
  endcase
end
```

This would throw a warning in synthesis because it clearly does not cover all cases

This would also throw a warning in simulation if **state1hot** was 3'b111 because multiple conditions would match

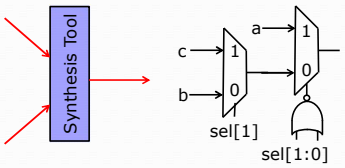
12

12

System Verilog Case Options

priority tells the synthesizer *and simulator* that at least one of the cases will always match

```
always_comb begin
  out = c;
  priority case (sel)
    2'b00 : out = a;
    2'b01 : out = b;
    2'b10 : out = c;
  endcase
end
```

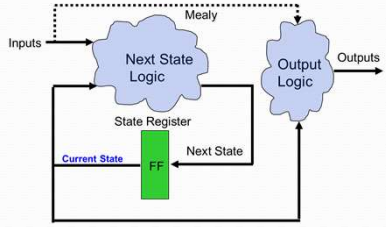


This would throw a warning in simulation if **sel** was 2'b11 because it does not match any case item.

13

13

State Machines



State Machines:

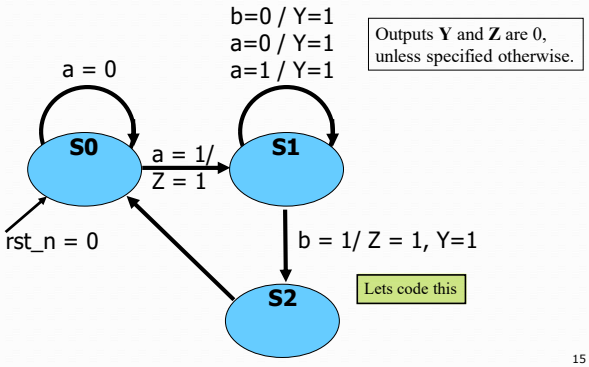
- Next State and output logic are combinational blocks, which have outputs dependent on the current state.
- The current state is, of course, stored by a FF.

- **What is the best way to code State Machines?:**
  - ✓ Best to separate combinational (blocking) from sequential (non-blocking)
  - ✓ Output logic and state transition logic can be coded in same always block since they have the same inputs
  - ✓ Output logic and state transition logic are ideally suited for a case statement

14

14

State Diagrams



Outputs **Y** and **Z** are 0, unless specified otherwise.

Lets code this

15

15

SM Coding

```
module fsm(clk,rst_n,a,b,Y,Z);
input clk,rst_n,a,b;
output reg Y,Z;
typedef enum reg [1:0] {S0, S1,
                        S2} state_t;
state_t state,nxt_state;
always_ff @(posedge clk,
            negedge rst_n)
  if (!rst_n)
    state <= S0;
  else
    state <= nxt_state;
```

What problems do we have here?

```
always_comb
  case (state)
    S0 : if (a) begin
           nxt_state = S1;
           Z = 1; end
         else
           nxt_state = S0;
    S1 : begin
           Y=1;
           if (b) begin
             nxt_state = S2;
             Z=1; end
           else
             nxt_state = S1;
         end
    S2 : nxt_state = S0;
  endcase
endmodule
```

16

16

SM Coding (2nd try of combinational)

```
always_comb
  nxt_state = S0;  // default to reset
  Z = 0;           // default outputs
  Y = 0;           // to avoid latches

  case (state)
    S0 : if (a) begin
           nxt_state = S1;
           Z = 1;
         end

    S1 : begin
           Y=1;
           if (b) begin
             nxt_state = S2;
             Z=1; end
           else nxt_state = S1;
         end
    default : nxt_state = S0;
  endcase
```

Defaulting of assignments and having a default to the case is highly recommended!

17

17

SM Coding Guidlines

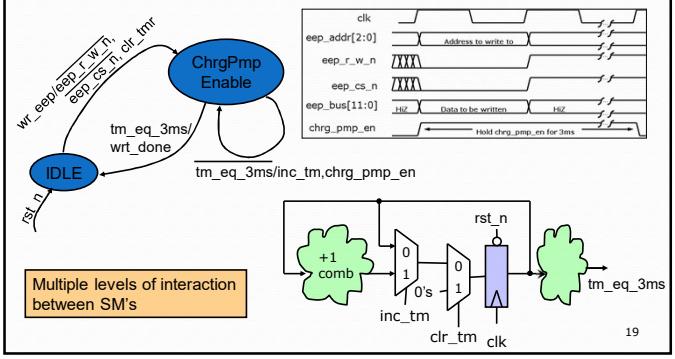
1) Keep state assignment in separate **always_ff** block using non-blocking "<=" assignment
2) Code state transition logic and output logic together in a **always_comb** block using blocking assignments
3) Assign default values to all outputs, and the ***nxt_state*** registers. This helps avoid unintended latches
4) Remember to have a default to the case statement.
   - Default should be (if possible) a state that transitions to the same state as reset would take the SM to.
   - Avoids latches
   - Makes design more robust to spurious electrical/cosmic events.

18

18

SM Interacting with SM

- A very common case is a state that needs to be held for a certain time.
  - ✓The state machine in this case may interact with a timer (counter).



Multiple levels of interaction between SM's

19

19

EEPROM Write SM Example [1]

```
module eeprom_sm(clk,rst_n,wrt_eep,
wrt_data,eep_r_w_n,eep_cs_n,
eep_bus,chrg_pmp_en,wrt_done);

input clk,rst_n,wrt_eep;
input [11:0] wrt_data;      // data to write
output eep_r_w_n,eep_cs_n;
output chrg_pmp_en;         // hold for 3ms
inout [11:0] eep_bus;

typedef enum reg {IDLE,
                  CHRG} state_t;
state_t state, nxtState;
reg [20:0] tm;      // 3ms => 21-bit timer
reg clr_tm, inc_tm, bus_wrt;
```

```
//// implement 3ms timer below ////
always_ff @(posedge clk,negedge rst_n)
  if (!rst_n)        tm <= 21'h000000;
  else if (clr_tm)   tm <= 21'h000000;
  else if (inc_tm)   tm <= tm+1;

//// @500MHZ cnt of 16E360 => 3ms ////
assign tm_eq_3ms = (tm==21'h16E360) ?
                   1'b1 : 1'b0;

//// implement state register below ////
always_ff @(posedge clk or
            negedge rst_n)
  if (!rst_n) state <= IDLE;
  else state <= nxtState;
```

20

20

EEPROM Write SM Example [2]

```
//// state transition logic & ////
//// output logic ////
always_comb begin
    nxtState = IDLE;    // default all
    bus_wrt = 0;        // to avoid
    clr_tm = 0;         // unintended
    inc_tm = 0;         // latches
    chrg_pmp_en = 0;

    case (state)
      IDLE : if (wrt_eep) begin
        clr_tm = 1;
        bus_wrt = 1;
        nxtState = CHRG;
      end
```

```
      default : begin        // is CHRG
                  inc_tm = 1;
                  chrg_pmp_en=1;
                  if (tm_eq_3ms)
                    begin
                      wrt_done = 1;
                      nxtState = IDLE;
                    end
                  else nxtState = CHRG;
                end
    endcase
  end

  assign eep_r_w_n = ~bus_wrt;
  assign eep_cs_n = ~bus_wrt;
  assign eep_bus = (bus_wrt) ?
                       wrt_data : 12'bzzz;
endmodule
```

21

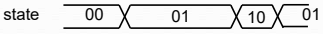
21

The Beauty of the Enumerated type

- One construct old school verilog was missing was an enumerated type.
- For state definition we use **localparam** was used for code readability

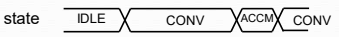
```
localparam IDLE = 2'b00;
localparam CONV = 2'b01;
localparam ACCM = 2'b10;
```

However, in waveform viewing it still shows up as digits. One has to always refer back to the encoding while debugging.



- System verilog corrected this by adding an enumerated type.

```
typedef enum reg [1:0] { IDLE, CONV, ACCM } state_t;
state_t state, nxt_state;          // declare state and nxt_state signals
```



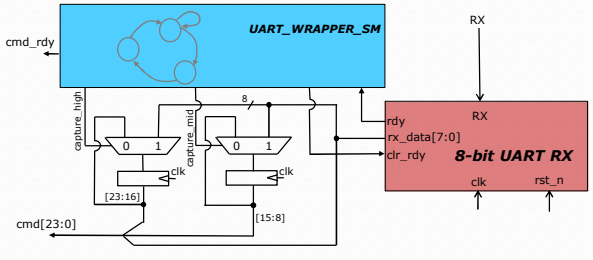
Makes debug much easier

22

22

UART Wrapper

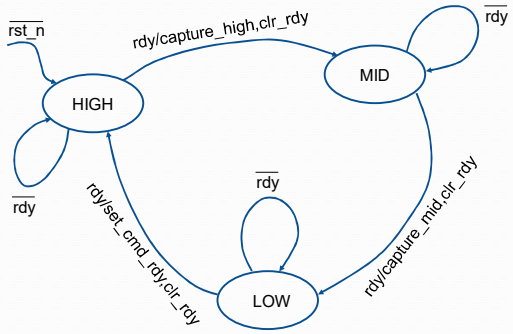
- Consider This Application. Collecting 3-bytes from a UART receiver and packaging them into a 24-bit command.



23

23

UART Wrapper Bubble Diagram



24

24

SM in System Verilog

```
module UART_wrapper_sm(input clk, rst_n, ...

typedef enum reg [1:0] {HIGH,MID,LOW} state_t;
state_t state, nxt_state;

logic capture_high,capture_mid,set_cmd_rdy,
      clr_cmd_rdy,clr_rdy;

////////// infer state flops ////////////
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    state <= HIGH;
  else
    state <= nxt_state;

always_comb begin
  ///// default outputs //////
  capture_high   = 0;
  capture_mid  = 0
  set_cmd_rdy  = 0;
  clr_cmd_rdy = 0
  clr_rdy  = 0;
  nxt_state = HIGH;
```

```
  case (state)
    MID : if (rdy) begin
      capture_mid = 1;
      clr_rdy = 1;
      nxt_state = LOW;
    end else
      nxt_state = MID;
    LOW : if (rx_rdy) begin
      set_cmd_rdy = 1;
      clr_rdy = 1;
      nxt_state = HIGH;
    end  else
      nxt_state = LOW;
    ///// default case = HIGH /////
    default : if (rdy) begin
      capture_high = 1;
      clr_rdy = 1;
      clr_cmd_rdy = 1;
      nxt_state = MID;
    end
  endcase
end
```

25

25

Default nxt_state to state

Allows less coding if a state is to remain in same state

```
module UART_wrapper_sm(input clk, rst_n, ...

typedef enum reg [1:0] {HIGH,MID,LOW} state_t;
state_t state, nxt_state;

logic capture_high,capture_mid,set_cmd_rdy,
      clr_cmd_rdy,clr_rdy;

////////// infer state flops ////////////
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)
    state <= HIGH;
  else
    state <= nxt_state;

always_comb begin
  ///// default outputs //////
  capture_high   = 0;
  capture_mid  = 0
  set_cmd_rdy  = 0;
  clr_cmd_rdy = 0
  clr_rdy  = 0;
  nxt_state = state;
```

```
  case (state)
    MID : if (rdy) begin
      capture_mid = 1;
      clr_rdy = 1;
      nxt_state = LOW;
    end else
      nxt_state = MID;
    LOW : if (rx_rdy) begin
      set_cmd_rdy = 1;
      clr_rdy = 1;
      nxt_state = HIGH;
    end  else
      nxt_state = LOW;
    ///// default case = HIGH /////
    default : if (rdy) begin
      capture_high = 1;
      clr_rdy = 1;
      clr_cmd_rdy = 1;
      nxt_state = MID;
    end
  endcase
end
```

26

26

End of Lecture05 Part I

27

27

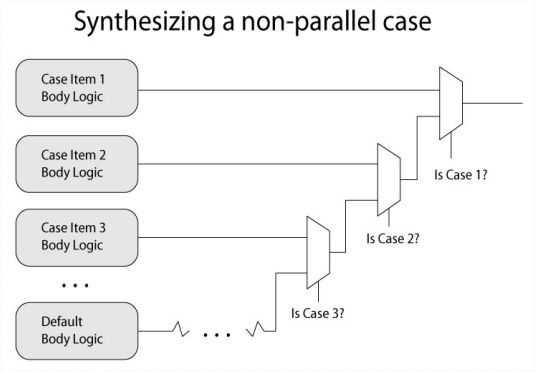
Random Misc Topics

Next slides are a bunch of stuff I wasn't sure where to put, but seemed like good information.

28

28

Consider how a **case/if else** will synthesize



29

29

SystemVerilog – Control Constructs

- *unique* keyword modifier

```
unique case (sel)
    CASE1: ...
    CASE2: ...
    CASE3: ...
endcase
```

- unique tells the synthesizer *and simulator* that one, and only one, case will be selected
- Also works with if: unique if(...) ...

30

30

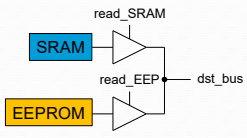
unique if Example

```
always_comb begin
  capX = 1'b0;
  capY = 1'b0;
  unique if (X_accel<min_accel)
    capX = 1'b1;
  else if (Y_accel<min_accel)
    capY = 1'b1;
end
```

This example is not a good use of **unique if**. It is likely the both the X-axis and Y-axis accelerometer values could be less than the min_accel. In which case there would be a run-time warning thrown in simulation because the match is not unique.

```
always_comb begin
  unique if (read_SRAM)
    dst_bus = RAM_data;
  else if (read_EEP)
    dst_bus = EEP_data;
  else
    dst_bus = 16'hzzzz;
end
```

This code would be a good use of **unique if** when you know read_EEP and read_SRAM are mutex



31

31

SystemVerilog – Control Constructs

- *priority* keyword modifier

```
priority case (sel)
    CASE1: ...
    CASE2: ...
endcase
```

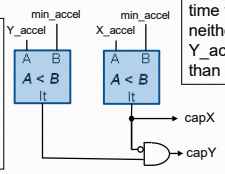
- priority tells the synthesizer *and simulator* that at least one of the cases will always match. If this doesn't happen in simulation it will warn you.
- Also works with if: priority if(...) ...

32

32

priority if Example

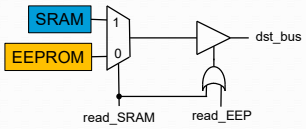
```
always_comb
   priority if (X_accel<min_accel) begin
      capX = 1'b1;
      capY = 1'b0;
   end else if (Y_accel<min_accel) begin
      capX = 1'b0
      capY = 1'b1;
   end
```



This will synthesize as shown, and would throw a run-time warning if neither X_accel or Y_accel were less than min_accel

```
always_comb begin
   priority if (read_SRAM)
      dst_bus = RAM_data;
   else if (read_EEP)
      dst_bus = EEP_data;
   else
      dst_bus = 16'hzzzz;
end
```

This would synthesize as shown. **priority if**'s will throw a run-time warning if no conditions match, but since this code has a pure **else** that will not happen.



33

33

Encoder With case

```
module encoder (output reg [2:0] Code, input [7:0] Data);
always @ (Data)
  // encode the data
  case (Data)
      8'b00000001      : Code = 3'd0;
      8'b00000010      : Code = 3'd1;
      8'b00000100      : Code = 3'd2;
      8'b00001000      : Code = 3'd3;
      8'b00010000      : Code = 3'd4;
      8'b00100000      : Code = 3'd5;
      8'b01000000      : Code = 3'd6;
      8'b10000000      : Code = 3'd7;
      default          : Code = 3'bxxx; // invalid, so don't care
  endcase
endmodule
```

How do we think it will synthesize?

34

34

Priority Encoder With case () inside

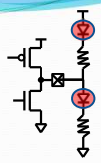
```
module priority_encoder (output reg [2:0] Code, input [7:0] Data);

always_comb // encode the data
  case (Data) inside
      8'b1xxxxxxx : Code = 7;
      8'b01xxxxxx : Code = 6;
      8'b001xxxxx : Code = 5;
      8'b0001xxxx : Code = 4;
      8'b00001xxx : Code = 3;
      8'b000001xx : Code = 2;
      8'b0000001x : Code = 0;
      8'b00000001 : Code = 0;
      default     : Code = 3'bxxx; // should be at least one 1, don't care
  endcase
endmodule
```

Will this synthesize larger or smaller than the previous example?

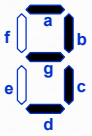
35

35

Seven Segment Display



```
module Seven_Seg_Display (Display, BCD, Blanking);
  output reg [6: 0]      Display;          // abc_defg
  input  [3: 0]          BCD;
  input                  Blanking;
  localparam   BLANK   = 7'b111_1111;      // active low
  localparam   ZERO    = 7'b000_0001;      // h01
  localparam   ONE     = 7'b100_1111;      // h4f
  localparam   TWO     = 7'b001_0010;      // h12
  localparam   THREE   = 7'b000_0110;      // h06
  localparam   FOUR    = 7'b100_1100;      // h4c
  localparam   FIVE    = 7'b010_0100;      // h24
  localparam   SIX     = 7'b010_0000;      // h20
  localparam   SEVEN   = 7'b000_1111;      // h0f
  localparam   EIGHT   = 7'b000_0000;      // h00
  localparam   NINE    = 7'b000_0100;      // h04
```



Defined constants – can make code more understandable!

36

36

Seven Segment Display [2]

```
always @ (BCD or Blanking)
   if (Blanking) Display = BLANK;
   else
     case (BCD)
        4'd0:        Display = ZERO;
        4'd1:        Display = ONE;
        4'd2:        Display = TWO;
        4'd3:        Display = THREE;
        4'd4:        Display = FOUR;
        4'd5:        Display = FIVE;
        4'd6:        Display = SIX;
        4'd7:        Display = SEVEN;
        4'd8:        Display = EIGHT;
        4'd9:        Display = NINE;
        default:     Display = BLANK;
     endcase
endmodule
```

Using the defined constants!

37

37

Mixing Flip-Flop Styles (1)

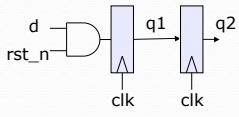
- What will this synthesize to?

```
module badFFstyle (output reg q2, input d, clk, rst_n);
  reg q1;

  always @(posedge clk)
    if (!rst_n)
      q1 <= 1'b0;
    else begin
      q1 <= d;
      q2 <= q1;
    end
endmodule
```

38

38

Mixing Flip-Flop Styles (2)



It synthesizes like this right?

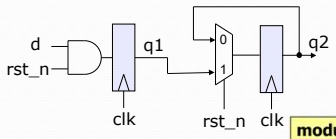
```
module badFFstyle (output reg q2,
                   input d, clk, rst_n);
  reg q1;

  always @(posedge clk)
    if (!rst_n)
      q1 <= 1'b0;
    else begin
      q1 <= d;
      q2 <= q1;
    end
endmodule
```

If !rst_n then q2 is not assigned…
It has to keep its prior value

39

39

Mixing Flip-Flop Styles (3)




```
module badFFstyle (output reg q2,
                   input d, clk, rst_n);
  reg q1;

  always @(posedge clk)
    if (!rst_n)
      q1 <= 1'b0;
      q2 <= q2;
    else begin
      q1 <= d;
      q2 <= q1;
    end
endmodule
```

40

40

Mixing Flip-Flop Styles (4)

```
module goodFFstyle (output reg q2, input d, clk, rst_n);
  reg q1;

  always @(posedge clk)
    if (!rst_n) q1 <= 1'b0;
    else q1 <= d;

  always @(posedge clk)
    q2 <= q1;

endmodule
```

Only combine like flops (same reset structure) in a single **always** block.

If their reset structure differs, split into separate **always** blocks as shown here.

Eric's Coding Guideline

- Unless flops have the same conditions under which they are reset and enabled separate them into different **always** blocks.

Inter vs Intra Statement Delays

- Inter-assignment delays block both evaluation and assignment
  - #4 c = d;
  - #8 e = f;
- Intra-assignment delays block assignment but not evaluation
  - c = #4 d;
  - e = #8 f;
- Blocking statement is still blocking though, so evaluation of next statements RHS still does not occur until after the assignment of the previous expression LHS.
  - What?? How is it any different then? Your confusing me!

Inter vs Intra Statement Delays (Blocking Statements)

```
module inter();
integer a,b;
initial begin
  a=3;
  #6 b = a + a;
  #4 a = b + a;
end
endmodule
```

Compare these two modules

```
module intra();
integer a,b;
initial begin
  a=3;
  b = #6 a + a;
  a = #4 b + a;
end
endmodule
```

Time	Event
0	a=3
6	b=6
10	a=9

Yeah, Like I said, they are the same!

Or are they?

Time	Event
0	a=3
6	b=6
10	a=9

Intra Statement Delays (Blocking Statements)

```
module inter2();
integer a,b;
initial begin
  a=3;
  #6 b = a + a;
  #4 a = b + a;
end

initial begin
  #3 a=1;
  #5 b=3;
end

endmodule
```

Time	Event
0	a=3
3	a=1
6	b=2
8	b=3
10	a=4

```
module inter2();
integer a,b;
initial begin
  a=3;
  b = #6 a + a;
  a = #4 b + a;
end

initial begin
  #3 a=1;
  #5 b=3;
end

endmodule
```

Time	Eval Event	Assign Event
0	nxt_b= 6	a=3
3	--	a=1
6	nxt_a=7	b=6
8	--	b=3
10	--	a=7

45

45

Non-Blocking: Inter-Assignment Delay

- Delays both the evaluation and the update…effectively becomes a blocking statement

```
always @(posedge clk) begin
      b <= a + a;
  # 5  c <= b + a;
  # 2  d <= c + a;
end

initial begin
  a = 3; b = 2; c = 1;
end
```

Time	Event
0	clk pos edge
0	b=6
5	c=9
7	d=12

46

46

Non-Blocking: Intra-Assignment Delay

- Delays the update, but not the evaluation. Does not block

```
always @(posedge clk) begin
    b <= a + a;
    c <= #5 b + a;
    d <= #2 c + a;
end

initial begin
  a = 3; b = 2; c = 1;
end
```

Time	Event
0	clk pos edge
0	b=6
2	d=4
5	c=5

This is more like modeling the Clk2Q delay of a Flop
(it captures on rising edge, but has a delay till output)

47

47

Intra-Assignment Review

```
module bnb;
reg a, b, c, d, e, f;

initial begin // blocking assignments
  a = #10 1; // a will be assigned 1 at time 10
  b = #2 0; // b will be assigned 0 at time 12
  c = #4 1; // c will be assigned 1 at time 16
end

initial begin // non-blocking assignments
  d <= #10 1; // d will be assigned 1 at time 10
  e <= #2 0; // e will be assigned 0 at time 2
  f  <= #4 1; // f will be assigned 1 at time 4
end
endmodule
```

Note: In testbenches I mainly find blocking inter-assignment delays to be the most useful. Delays really not used outside of testbenches that much during the design process.

48

48