ET5080E
Digital Design Using Verilog HDL

Fall '21

Counters are Common
Shifters/Rotators
Parameters
Proper SM Coding
Random Misc Stuff

Administrative Matters

- Readings
  - Cummings paramdesign paper for hdlcon (posted on class website)

What Have We Learned?

1) Sequential elements (flops & latches) should be inferred using non-blocking “**<=**“ assignments

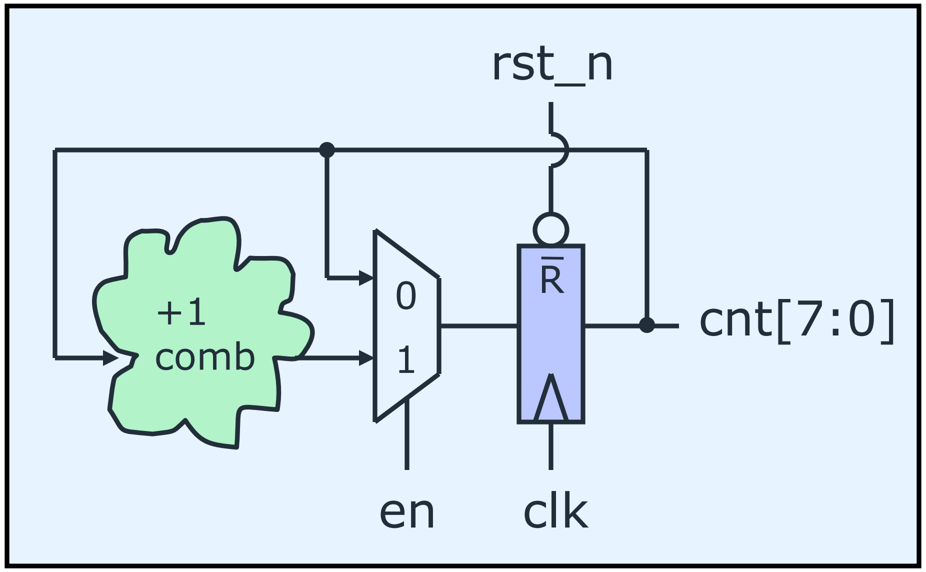
2) Combinational logic should be inferred using blocking “**=**“ statements.

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.

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

Pill Counter

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

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

reg [7:0] nxt_cnt,cnt;

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

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

endmodule
```

I.Q. Counter (the rebel engineer)

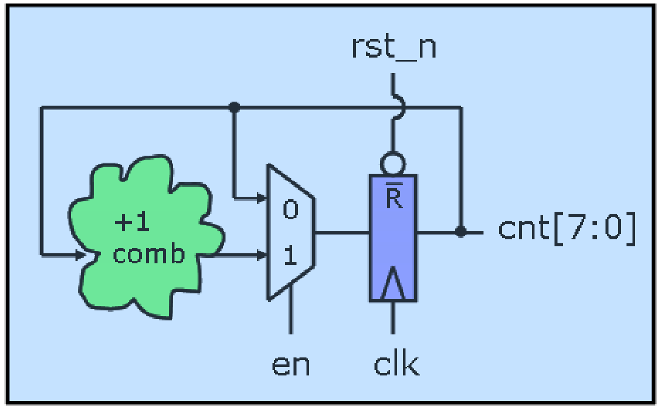
```
module iq_cnt(clk,rst_n,en,cnt);

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

reg [7:0] cnt;

always @(posedge clk or negedge rst_n)
  if (!rst_n)
    cnt <= 8'h00;
  else if (en)
    cnt <= cnt + 1;  // combinational

endmodule
```

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

Is this OK?



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?

Ring Counter (a better way)

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

  always @  (posedge clock or negedge reset_n)
    if (!reset_n)                         count <= 8'b0000_0001;
    else if (enable == 1'b1)      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.

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?

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?
- There is no default to the case, is this bad?
- Why was the MSB replicated on the ÷ by 2

Aside (a quick intro to parameters)

- **parameter** → like a **local `define**
  - Defined locally to the module
  - Can be overridden (passed a value in an instantiation)
  - There is another method called **defparam** (don't ever use it) that can override them

- **localparam** → even more local than parameter
  - Can't be passed a value
  - **defparam** does not modify
  - Only available in Verilog 2001

Aside (a quick intro to parameters)

```
module adder(a,b,cin,sum,cout);

parameter WIDTH = 8; // default is 8

input [WIDTH-1:0] a,b;
input cin;
output [WIDTH-1:0] sum;
output cout;

assign {cout,sum} =  a  +  b + cin

endmodule
```

```
module alu(src1,src2,dst,cin,cout);
  input [15:0] src1,src2;
  …
  /////////////////////////////////////
  // Instantiate 16-bit adder //
  ///////////////////////////////////
  adder #(16) add1(.a(src1),.b(src2),
                   .cin(cin),.cout(cout),
                   .sum(dst));


  …
endmodule
```

Instantiation of module can override a parameter.

Aside (a quick intro to parameters)

- Examples:

```
parameter Clk2q = 1.5,
              Tsu = 1,
              Thd = 0;
```

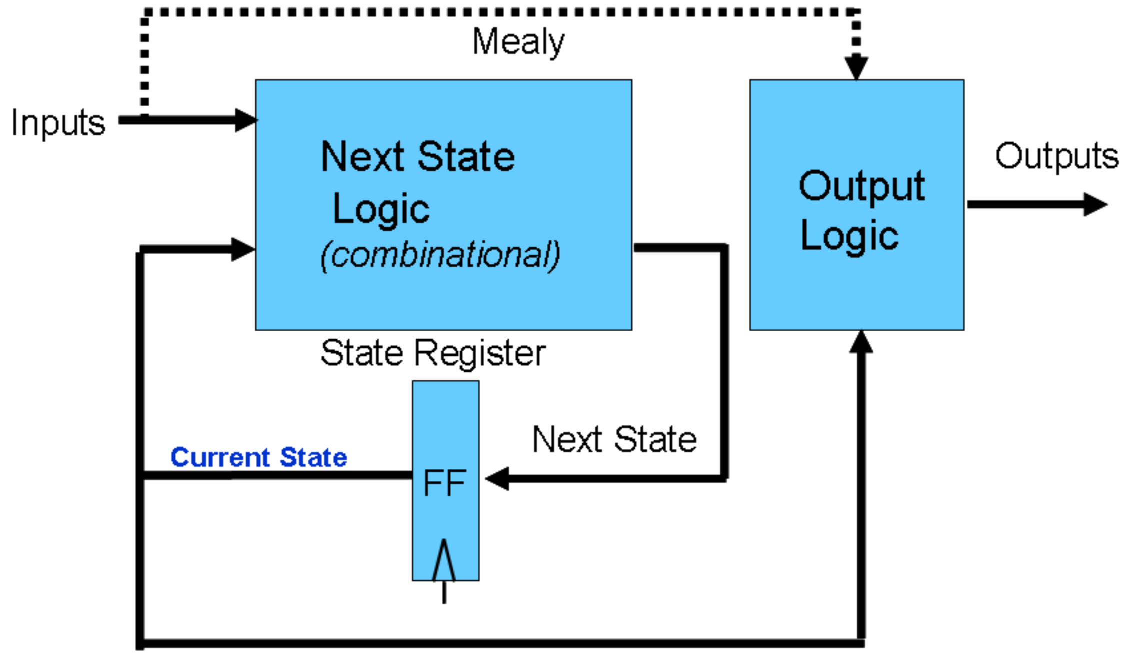
```
parameter IDLE   = 2'b00;
parameter CONV  = 2'b01;
parameter ACCM  = 2'b10;
```

```
module register2001 #(parameter SIZE=8)
  (output reg [SIZE-1:0] q, input [SIZE-1:0] d,
   input clk, rst_n);

always @(posedge clk, negedge rst_n)
  if (!rst_n) q <= 0;
  else q <= d;

endmodule
```

Verilog 2001 allows definition in module header

- Read Cummings *paramdesign* paper for hdlcon posted on class website

State Machines

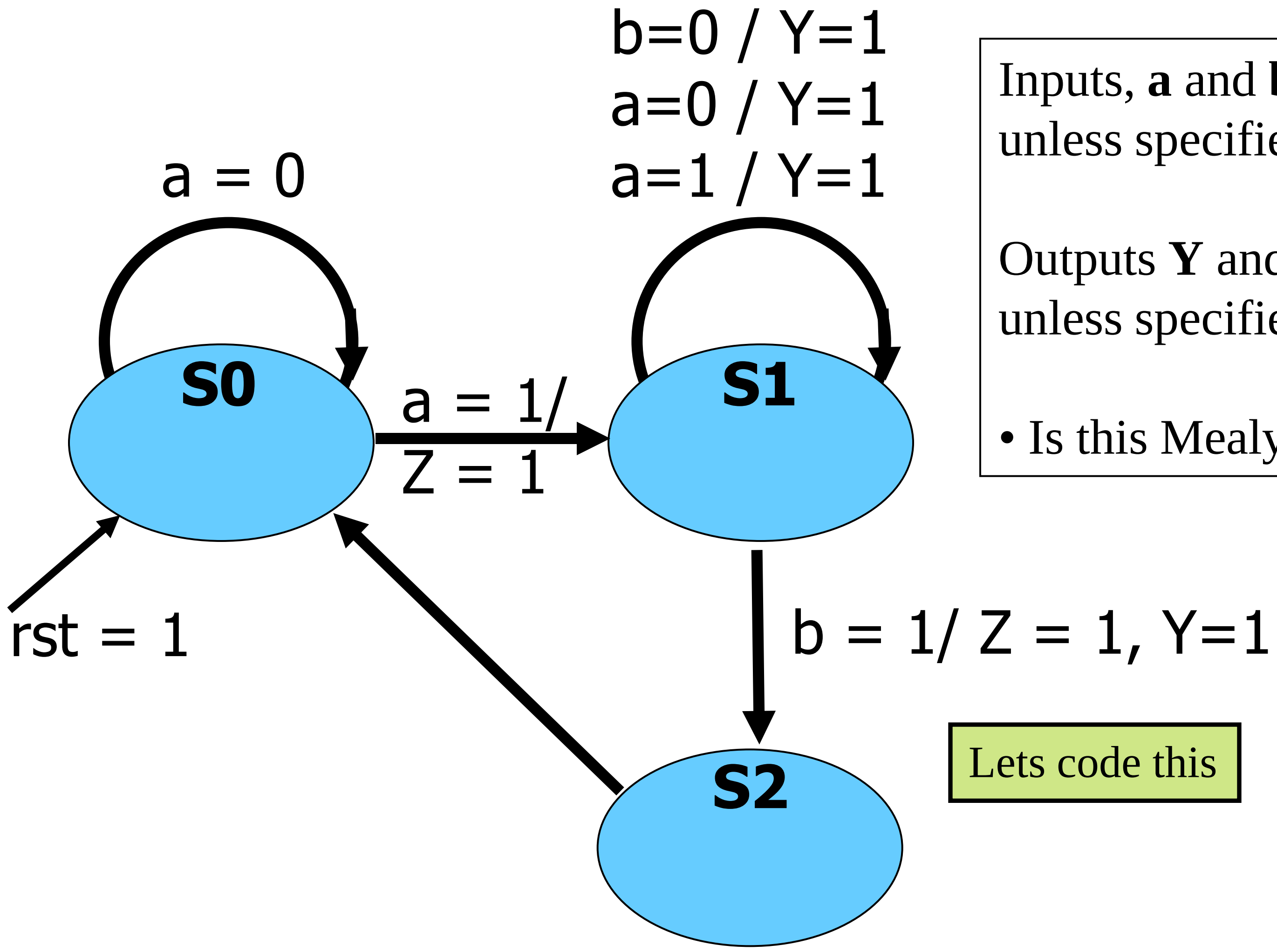


State Machines:

- 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

State Diagrams



Inputs, **a** and **b** are 0, unless specified otherwise

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

- Is this Mealy or Moore?

Lets code this

SM Coding

```
module fsm(clk,rst,a,b,Y,Z);

input clk,rst,a,b;
output Y,Z;

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

reg [1:0] state,nxt_state;

always @(posedge clk, posedge rst)
  if (rst)
     state <= S0;
  else
     state <= nxt_state;
```

What problems do we have here?

```
always @ (state,a,b)
   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
```

SM Coding (2nd try of combinational)

```
always @ (state,a,b)
   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
endmodule
```

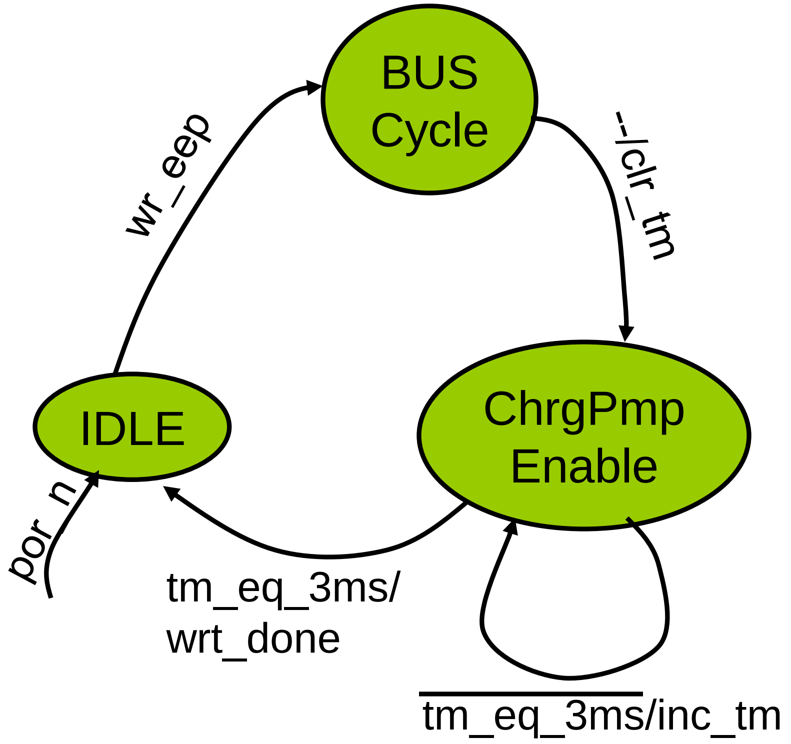
Defaulting of assignments and having a default to the case is highly recommended!

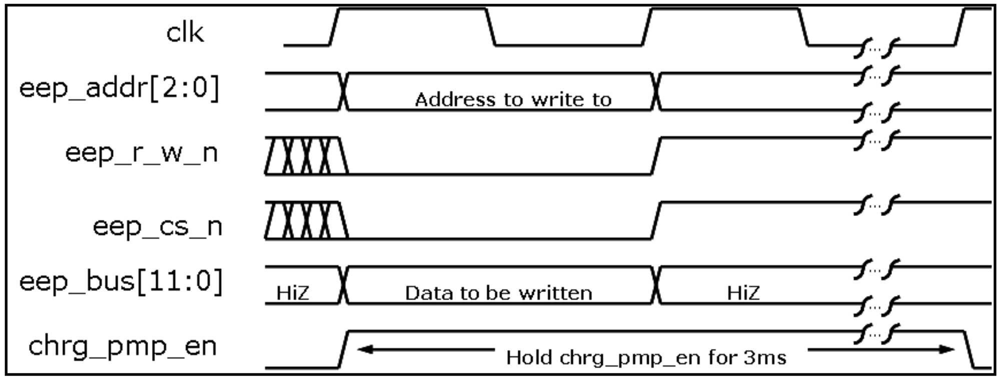
SM Coding Guidlines

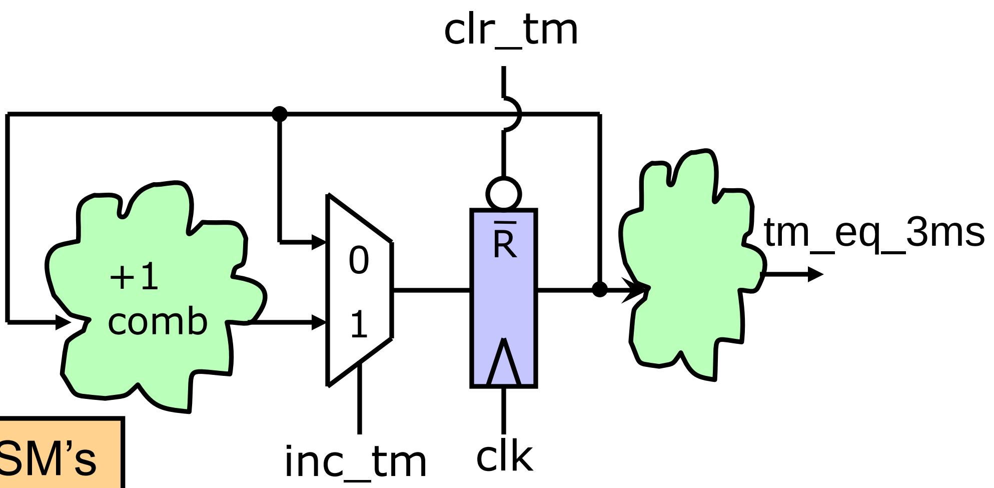
1) Keep state assignment in separate always block using non-blocking “<=“ assignment
2) Code state transition logic and output logic together in a always block using blocking assignments
3) Assign default values to all outputs, and the *nxt_state* registers. This helps avoid unintended latches
4) Remember 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.

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

EEPROM Write SM Example [1]

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

parameter IDLE    = 2'b00,
          BUS     = 2'b01,
          CHRG = 2'b10;

input clk,por_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;

reg [13:0] tm;         // 3ms => 14-bit timer
reg clr_tm,inc_tm,bus_wrt;
reg [1:0] state,nxtState;
```

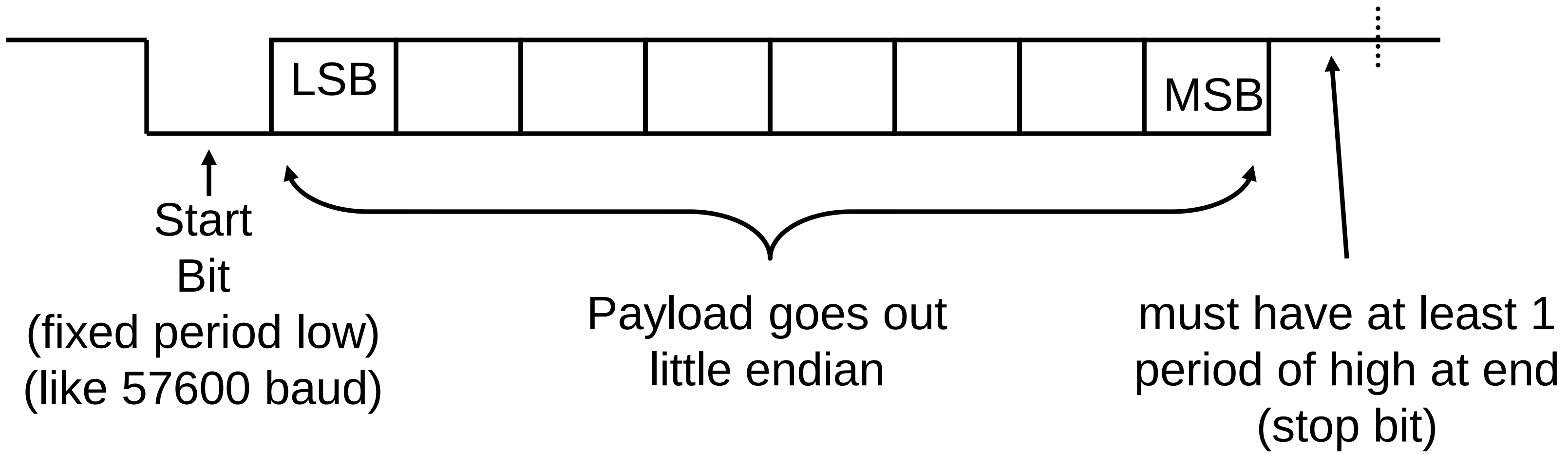
```
//// implement 3ms timer below ////
always @(posedge clk or
                posedge clr_tm)
  if (clr_tm)         tm <= 14'h0000;
  else if (inc_tm)  tm <= tm+1;

//// @4MHZ cnt of 2EE0 => 3ms ////
assign tm_eq_3ms = (tm==14'h2EE0) ?
                               1'b1 : 1'b0;

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

EEPROM Write SM Example [2]

```
//// state transition logic & ////
//// output logic ////
always @(state,wrt_eep,tm_eq_3ms)
  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)
        nxtState = BUS;
      BUS : 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'bzzzz;
endmodule
```

Are there optimizations that can be made?

USART (RS232) Example



Assume we have a 4MHz clock running our digital system

We want to make a RS232 transmitter with a baud rate of 57,600

How many clock cycles do we hold each bit?

$$\boxed{Cycles = \frac{4MHz}{57600\,baud} \approx 69}$$

$$\boxed{69 = 7\text{'b}1000101}$$

USART Example

```
module usart_tx(clk,rst_n,strt_tx,tx_data,tx_done,TX);

input clk, rst_n, strt_tx;      // start_tx comes from Master SM
input [7:0] tx_data;            // data to transmit
output TX;                      // TX is the serial line
output tx_done;                 // tx_done asserted back to Master SM
          .
          .
          .
endmodule;
```

1) Go over HW3 problem statement

Random Misc Topics

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

Mux With **case**

```
module Mux_4_32_(output [31:0] mux_out, input [31:0] data_3,
    data_2, data_1, data_0, input [1:0] select, input enable);
      reg           [31: 0]  mux_int;
     // choose between the four inputs
    always @ ( data_3 or data_2 or data_1 or data_0 or select)
      case (select)  (* synthesis parallel_case *)
        2'b00:              mux_int = data_0;
        2'b01:              mux_int = data_1;
        2'b10:              mux_int = data_2;
        2'b11:              mux_int = data_3;
      endcase
    // add the enable functionality
    assign         mux_out = enable ? mux_int : 32'bz;
endmodule
```

Synthesis directive:
Lets the synthesis tool know to use parallel (mux) scheme when synthesizing instead of priority encoding. Called an attribute in the IEEE spec

- Case statement implies priority unless use parallel_case pragma

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
```

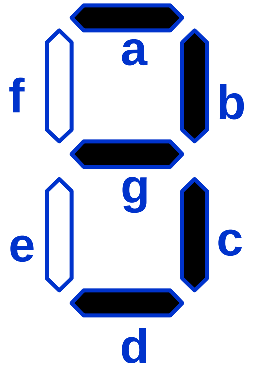
How do we think it will synthesize?

Priority Encoder With **casex**

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

assign valid_data = |Data; // "reduction or" operator
always @  (Data)
  // encode the data
  casex (Data)
       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 = 1;
       8'b00000001  : Code = 0;
       default      : Code = 3'bxxx; // should be at least one 1, don’t care
  endcase
endmodule
```

Seven Segment Display

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



Defined constants – can make code more understandable!

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!

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
```

```
module intra();
integer a,b;

initial begin

  a=3;

  b = #6 a + a;
  a = #4 b + a;

end
endmodule
```

Compare these two modules

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

Yaa, 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

Non-Blocking: Inter-Assignment Delay

- Delays both the evaluation and the update

```
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	c=12

Non-Blocking: Inter-Assignment Delay

- Delays both the evaluation and the update

```
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)

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.