

ECE 551

Digital Design And Synthesis

Fall '17

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

Administrative Matters

- Readings

- Cumming SNUG paper quiz on Weds Oct 11th
- Quiz on Video09 and Video10 (Lecture05) on Monday Feb 9th
- Cummings paramdesign paper for hdlcon (posted on class website)

- HW3 Posted (*a difficult one*)

- Midterm

- Weds Oct 25th 7:15PM in EH1800
- Alternate is 48hrs earlier in ??
- Taking the alternate has to be cleared with Hoffman

2

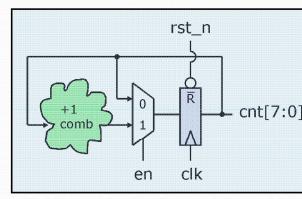
What Have We Learned?

- 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.

3

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 sequential. → non-blocking

4

Pill Counter (follow all the rules)

```
module pill_cnt(clk,rst_n,en,cnt);
  input clk,rst_n;
  output [7:0] cnt;
  reg [7:0] nxt_cnt,cnt;
  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;
  always @(*(en or cnt))
    if (en)
      assign nxt_cnt = (en) ? cnt+1 : cnt;
  endmodule
```

5

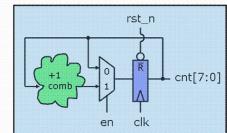
I.Q. Counter (the rebel engineer)

```
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?



6

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?

7

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)
      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.

8

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?

9

Shifter

```
always @ (posedge clk) begin
  if (rst) Data_Out <= 0;
  else case (select[1:0])
    2'b00: Data_Out <= Data_Out;
    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

10

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 & newer

11

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
```

Instantiation of module can override a parameter.

```
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
```

12

Aside (a quick intro to parameters)

- Examples:

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

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

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

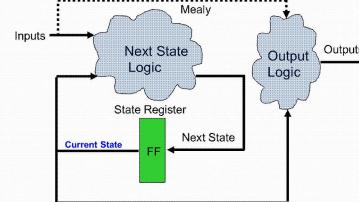
  always @ (posedge clk, negedge rst_n)
    if (!rst_n) q <= 0;
    else q <= d;

endmodule
```

• Read Cummings
paramdesign paper
for hdlcon posted on
class website

13

State Machines



State Machines:

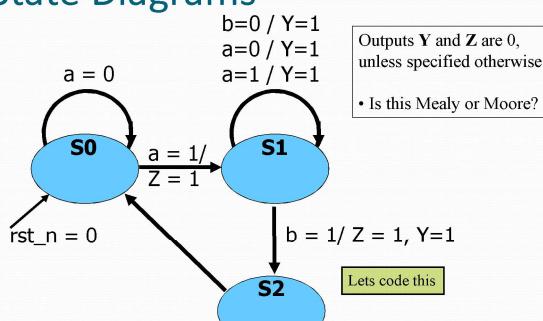
- Next State and output logic are combinational blocks, which have outputs dependent on the current state.
- The current state is, of course, stored by a FF.

• 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

State Diagrams



15

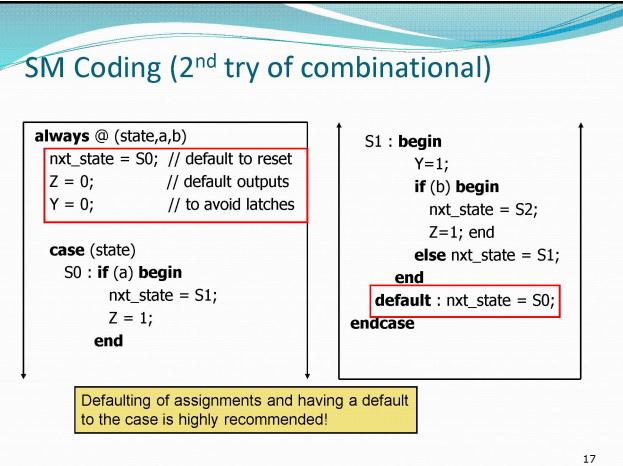
SM Coding

```
module fsm(clk,rst_n,a,b,Y,Z);
  input clk,rst_n,a,b;
  output reg Y,Z;
  localparam S0 = 2'b00;
  localparam S1 = 2'b01;
  localparam S2 = 2'b10;
  reg [1:0] state,nxt_state;
  always @(posedge clk,
            negedge rst_n)
    if (!rst_n)
      state <= S0;
    else
      state <= nxt_state;
```

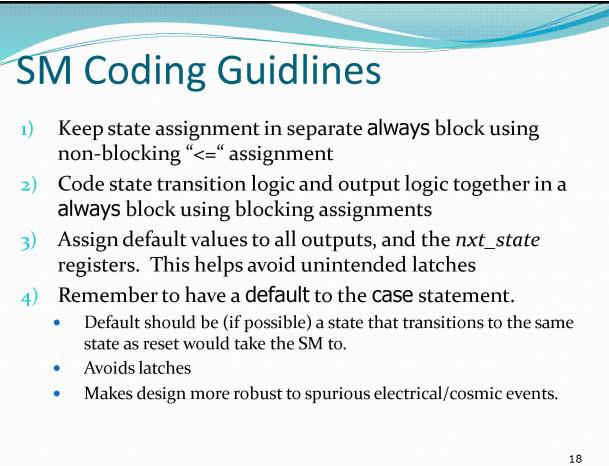
What problems do
we have here?

```
  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
```

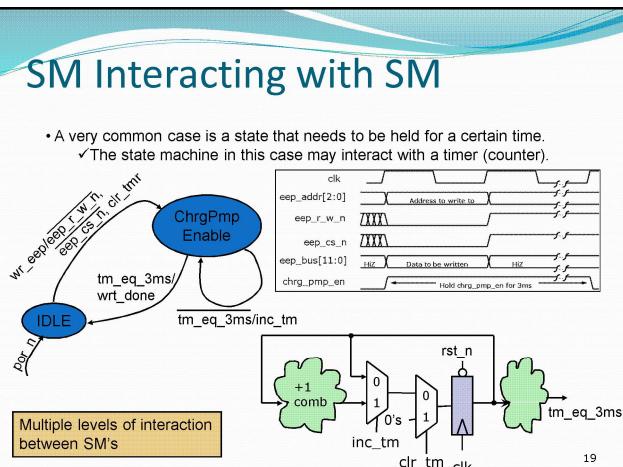
16



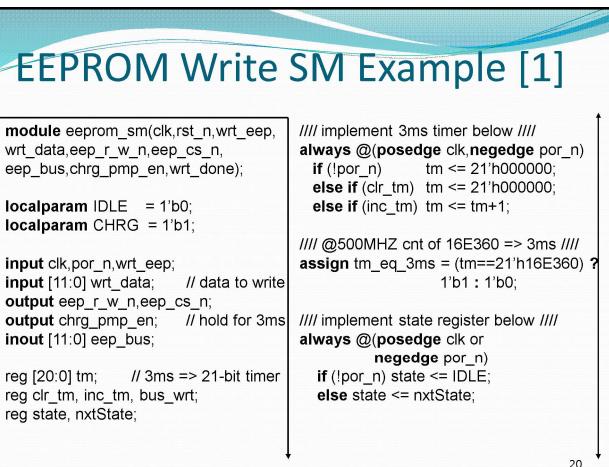
17



18



19



20

EEPROM Write SM Example [2]

```


// state transition logic & //
// output logic //
always @ (state,wrt_eep,tm_eq_3ms)
begin
    nextState = 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;
            nextState = CHRG;
        end
    endcase
end

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

endmodule


```

System Verilog SM Coding

- One construct verilog was missing was an enumerated type.
- For state definition we use **localparam** in verilog to make code more readable

```

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 adds an enumerated type.

```

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

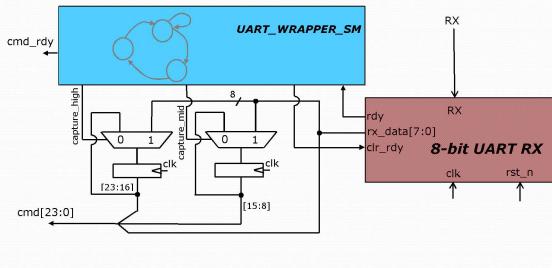
```

state IDLE X CONV X ACCM X CONV Makes debug much easier

22

UART Wrapper

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



23

SM in System Verilog

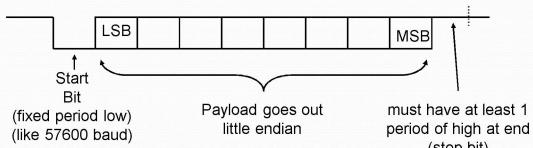
```

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

```

24

UART (RS232) Example



Assume we have a 50MHz clock running our digital system

We want to make a RS232 transmitter with a baud rate of 19,200

How many clock cycles do we hold each bit?

$$Cycles = \frac{50MHz}{19200baud} \approx 2604$$

$$2604 = 12'hA2C$$

25

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

26

Random Misc Topics

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

27

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

28

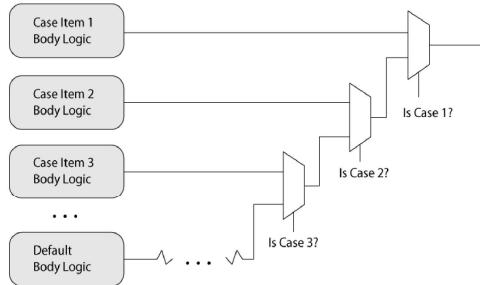
SystemVerilog – Control Constructs

- Case and **if/else** have priority by default in Verilog.
- To match this behavior in synthesis, we need a cascade of muxes.
- Designers commonly use **synopsis parallel_case** to force the synthesizer to make a single mux instead.
- Synthesizer pragmas give different information to the synthesis tool than to the simulator.
- This is fundamentally bad. We want **synthesis tool and simulator to have the same information!**

29

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

Synthesizing a non-parallel case



30

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(...)** ...

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(...)** ...
 - Easy way to avoid accidental latches!

32

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'bxx; // invalid, so don't care
endcase
endmodule
```

33

How do we think it will
synthesize?

Priority Encoder With casex

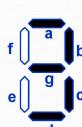
```
module priority_encoder (output reg [2:0] Code, input [7:0] Data);
always @ (Data)
// encode the data
casex (Data)
8'b1xxxxxxx : Code = 7;
8'b01xxxxxx : Code = 6;
8'b001xxxxx : Code = 5;
8'b0001xxxx : Code = 4;
8'b00001xxx : Code = 3;
8'b000001xx : Code = 2;
8'b0000001x : Code = 1;
8'b00000001: Code = 0;
default        : Code = 3'bxx; // should be at least one 1, don't care
endcase
endmodule
```

34

Will this synthesize larger or smaller
than the previous example?

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!

35

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!*

36

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!

37

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

Yaa, Like I said, they are the same!
Or are they?

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

38

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
```

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

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

39

40

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)

41

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.

42