EE108a: Verilog Examples

Version 1.0 – David Black-Schaffer

Building Counters Veriog Example

There are many different ways to write code in Verilog to implement the same feature. In ee108a you should strive to make your code as easy to read and debug as possible.

The counter example in the book instantiates a flip flop for storing the count, and then uses a case statement to build a mux to choose the next input to the flip flop based on the control signals. Let’s take a look at two ways to write this in Verilog.

Example 1:

This is the up/down counter code from the course reader:

```
module UDL_Count1(clk, rst, up, down, load, in, out) ;
      parameter n = 4 ;
      input clk, rst, up, down, load ;
      input [n-1:0] in ;
      output [n-1:0] out ;
      wire [n-1:0] out ;
      reg [n-1:0] next ;

      DFF #(n) count(clk, next, out) ;

      always@(rst, up, down, load, in, out) begin
            casex({rst, up, down, load})
                  4’b1xxx: next = {n{1’b0}} ;
                  4’b01xx: next = out + 1’b1 ;
                  4’b001x: next = out - 1’b1 ;
                  4’b0001: next = in ;
                  default: next = out ;
            endcase
      end
endmodule
```

This code is fairly easy to read except that it concatenates all of the bits of the control (rst, up, down, load) into one signal and then does a case on them. Understanding what the 4’b001x: case is requires the reader to look at what the case statement is doing. Comments would help here, but we’d like to make it simpler.

Example 2:

This is the same[1] up/down counter as the code from the course reader, but it is a lot easier to understand:

```
module UDL_Count2(clk, rst, up, down, load, in, out) ;
      parameter n = 4 ;
      input clk, rst, up, down, load ;
      input [n-1:0] in ;
      output [n-1:0] out ;
      wire [n-1:0] out ;
      reg [n-1:0] next ;

      DFF #(n) count(clk, next, out) ;

      always@* begin
            if (rst)
                  next = {n{1'b0}};
            else if (load)
                  next = in;
            else if (up)
                  next = out + 1'b1;
            else if (down)
                  next = out - 1'b1;
            else
                  next = out;
      end
endmodule
```

It is immediately apparent that the else if (down) case is what happens when counting down, unlike the 4'b001x case in Example 1.

So what is the tradeoff here? Well, Example 2 is clearly easier to read and understand, but Example 1 is more explicit about what logic should be built. In general you should write the most clear and easy to debug code you can. You should only revert to explicitly telling the tools what to do (as in Example 1) when you see that 1) the tools are doing a bad job with your code and 2) you've determined that it is that chunk of code that really matters. You may well run into this issue in lab 4 and the final project!

[1] Technically this version chooses the results with priority. (I.e., if the first if statement is true then it won't evaluate any of the following statements.) However, that this is exactly what the casex statement in Example 1 is doing by having the least specific to most specific cases listed.

Inferred state

We've warned you several times about not inferring latches in ee108a. **Whenever you need state storage (counters, FSMs, etc.) you must explicitly instantiate a flip flop from the provided ff_lib.v file**. This file explicitly infers state to generate a flip flop. To understand what you need to do to avoid doing this yourself, let's take a look inside the flip flop library:

The EE108a Flip Flop Module:

```
module dff (d, clk, q);
      parameter WIDTH = 1;
      input clk;
      input [WIDTH-1:0] d;
      output [WIDTH-1:0] q;
      reg [WIDTH-1:0] q;

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

You should note two things here that you haven't seen before in Verilog. The first is the @ (posedge clk) and the second is the <= operator. (You are not allowed to use either of these in ee108a so that's why you haven't seen them yet.)

So what is that always block doing? Well, because it is not using always @* it will only evaluate when @ (posedge clk) is true. This means that on every rising edge of the clock (and only on the rising edges of the clock) this block will be evaluated. The q <= d statement says that q will get the last value of d, and will only change to that value when everything else is done being evaluated. (That way the output of the flip flop will only change once even if the input changes multiple times during simulation.)

Now the real question is what is the value of q when we're not on the rising edge of the clock? What happens is that Verilog infers a latch for q, and remembers its value whenever the clock is not a rising edge. This is exactly the behavior we want for a flip flop, but we don't want this behavior anywhere else in our designs.

Mistakes that infer state

Now that we've seen how to build a flip flop on purpose, let's see how we can build them by accident.

Verilog will only infer state in your design if you don't build combinational logic in always blocks. That is, **as long as every single output is defined for every single combination of inputs you will never infer state**. This should make sense: if all the outputs are defined for all the inputs then the block is purely combinational (the outputs only depend on the current inputs). If an output is not defined for a given set of inputs then Verilog will infer a latch to remember the previous output to use in that case. You are not allowed to do this ee108a and we will take off points if you do.

Let’s look at an example:

```
reg result;
reg status;
always @* begin
      if (input_button) begin
            status = 1’b1;
            result = 1’b0;
      end
      else if (done_signal) begin
            status = 1’b0;
      end
end
```

This code does not define every output for every combination of inputs. There are two different problems here. The first is that we don’t define the value of the result output for the case where done_signal is true. The second is that we don’t define the value of either status or result in the case where both done_signal and input_button are false. As a result this will infer latches.

The correct code is as follows, with changes underlined:

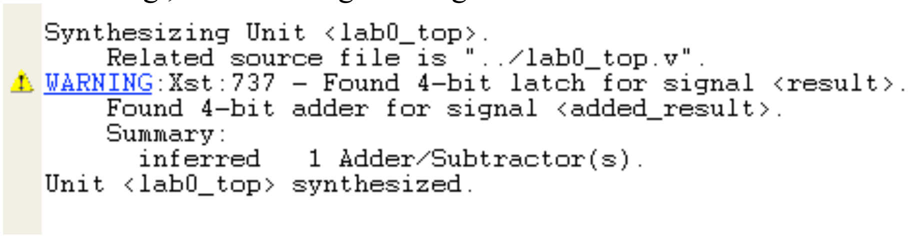
```
reg result;
reg status;
always @* begin
      if (input_button) begin
            status = 1’b1;
            result = 1’b0;
      end
      else if (done_signal) begin
            status = 1’b0;
            result = 1’b1;
      end
      else begin
            status = 1’b0;
            result = 1’b0;
      end
end
```

This version is purely combinational because it defines all outputs for all combinations of inputs. The same thing can happen with a case statement where you don’t include a default.

The basic rules to avoid this problem are:

1. Always have an else for every if
2. Always define the same set of signals in all cases or if clauses
3. Always have a default for every case
4. Always read the warnings in Xilinx ISE about inferred latches.

You will only see warnings about this in Xilinx because ModelSim assumes that you just wanted to build a latch and so it just gives you one. When you run Xilinx you need to look at the output from the synthesis step and make sure you don’t see any warnings except where you have instantiated a flip flop from the ff_lib.v module. E.g., the following warning is not allowed in this class:



```
Synthesizing Unit <lab0_top>.
    Related source file is "../lab0_top.v".
WARNING:Xst:737 - Found 4-bit latch for signal <result>.
    Found 4-bit adder for signal <added_result>.
    Summary:
      inferred   1 Adder/Subtractor(s).
Unit <lab0_top> synthesized.
```