The Missing Lecture of 271 Winter 2020

When in-person classes were cancelled, we had about 1 lecture of important stuff left. I was also going to do the transistor basis for all of our gates, but that’s optional stuff put at the end of the class because it is useful, but can be cancelled in case of snowdays, etc.

In the scans directory are a set of slides named slides209-215.pdf You’ll want to look at those. They have the start of the discussion in white, then what the slides would look like at the end in yellow (my class notes). We’ll go through those here. You probably want to look at the white and think about the question, then read through my discussion, then look at the final result on the yellow page.

Review Problem 56.

Is this code legal? Seems odd that the “b” is only defined in the “if” section, and “c” in the “then” portion, so are we violating the rule that “if you ever write a variable in an always_comb block, you must always write that variable”? Looks like this could give us a “latches” error because it is sequential.

Actually, this code is just fine. If you look at the code you ALWAYS get a value for b and c, since you initialize them at the top. This is actually how I often write ALWAYS_COMB blocks – put default values in at the top (possibly X’s if that makes sense), then override them with more specific values later in the code. I find that it results in fairly clean code.

So now let’s figure out what the code does. B is set to 0, and is only changed to D if A is true. So if A is false B is false, and if A is true then B is equal to D. When is B true? Only when both A and D are true. So we get B = A & D.

What about C? Same idea. C is only true if A is false, and (D|E) is true. So C = ¬A & (D | E)

Slide 209

Remember, I’m only showing snippets of code, so don’t worry that ps is not defined, etc. We’re focusing on portions of the code.

What’s going on here in the case statement? Well, for various values of PS and W, we set the value of NS. This is really just a giant truth table – given PS and W, we determine NS. For the parameter constants (A, B, and C) we have a specific encoding listed towards the top of the code, so saying NS = B is just saying NS = 2’b01.

BTW, this code is buggy – notice that the CASE statement doesn’t handle the situation when PS == 11. We need to add a line “default: ns = 2’bXX”.
On the yellow version you can see the resulting truth table. Quartus, when it is converting this Verilog to hardware, will essentially create this truth table, and solve the resulting two K-maps, one for NS[1], and the other for NS[0].

**Slide 210**

Where slide 209 was the combinational logic for a finite state machine, this is the DFFs. We’ve got an always_ff statement, that defines what happens on the clock edge. Because it is creating a 2-bit value “PS”, this is two DFFs, one for PS[1], the other for PS[0].

Now inside of this is an “if (reset)” statement. This is just like the modes we had in registers and counters: based on the value of reset, we either send in “Start” or NS to PS. So this is just a mux before the D input of each DFF. The Yellow version shows these structures.

Note that this is an example of RESET’ing to something other than all 0’s. Our reset state is Start, and Start = 2'b01. Turns out that the DFFs inside of an FPGA can handle this just fine – when the “reset” signal is true, PS[1] is reset to 0, and PS[0] is “set” to 1. “Set” is just a way of saying “it resets, but resets to 1 instead of 0”. Common term in digital logic. If you are curious, look up the SR latch or SR flipflop on the web or in the class textbook to see more on this.

**Slide 211**

Here we’ve got a counter – this isn’t a “deep” slide, but one I want to use before talking about things in slide 212 actually... Kinda like slide 210, the “if (reset) ... else ...” creates muxes controlled by the reset signal, and we have one DFF for each bit of OUT, so 4 DFFs. Overall structure is shown in the yellow sheet. BTW, how do we do the “+”? With an adder... See slides from the previous lecture on how we might have implemented that. In fact, since the adder has one input tied to a constant 4'b0001, this logic would get simplified down to a hard-coded incrementer, again exactly like we did in the previous lecture.

One note – we don’t have a final else. We know what to do if reset is true, and what to do if incr is true, but what if neither is true? Is this a latches error?

Nope, this is just fine. If you ever assign to a variable in an always_comb block you must always assign to it. But, this is an ALWAYS_FF block. What happens when INCR and RESET are both false? The output stays the same. A “hold” FF block. This is just fine. If you wanted to write “else out <= out;” that would be legal, though redundant.

**Slide 212**

This is a histogrammer, a chunk of hardware that counts how many times a given type of event occurs. Why this? Turns out to be an interesting piece of Verilog, and actually a piece of hardware people often want to build. Why? Well, imagine you are building the Large Hadron Collider in Europe, or a PET Scanner (medical imaging) system here in the US. When our research group worked on each of these,
you realize what happens is an energetic particle slams into some electrically active element (a detector, sometimes a crystal of special materials, other times an electronic sensor) that screams out “I SAW SOMETHING” when the energetic particle arrives. However, each of these sensors have differing sensitivities – some over-react, others under-react, and what we need to do is tune them. What we do is we just histogram the data coming off of all of the sensors, and then adjust the “gain” (how sensitive each sensor is, via electronic tuning knobs) so that the average values seen are the same for all of them. Hence, we histogram and then tune...

This code builds the histogrammer. For this simple case we have a counter for each value, and just add one to whatever event we see at that moment. Notice that we are doing all of this inside an always_ff block, so it does the math and the DFFs in one chunk of logic. Compare this to slide 211 – notice any similarity? Yep, each of those lines is just a counter, with the logic from the yellow on slide 211.

But, how do we control the INCR signal for each counter? Check the code. Based on inVal, we set to 1 one of the INCR signals, corresponding to the inVal we are seeing, and leave the other INCR signals at 0, so that only one counter actually counts. What’s that circuit?

The circuit that sets one output to true that corresponds to a given input pattern is just a decoder – pass in a binary number, and out comes the wire whose number is specified. Case statements like this will often give rise to decoders.

**Slide 213**

This is a chunk of code to do a comparison operator – is A less than B. How do we do this hardware? You guys did it for homework #5, so just check that solution. This unit is a comparator, and we’ll use it for the next slide.

**Slide 214**

This is a meaty one. You want to look at the yellow version, and read it from top to bottom, one line at a time.

So, we’ve written tons of ALWAYS_COMB blocks this quarter, but how does THAT equal hardware? Well, the simple version is that Verilog describes what the hardware should do, and the Quartus tool synthesizes circuits that do the same thing.

An ALWAYS_COMB block runs infinitely fast, and the “output” is whatever the value of the variables are when you reach the bottom of the ALWAYS_COMB block. So, read through the always_comb block, watching what happens to the variables, and that set of transformations is what the corresponding hardware does.

Easier to understand via an example. Let’s walk through this code, and we’re going to go line-by-line, from top to bottom. Don’t peek at the final circuit at the bottom just yet!
This block computes o0 and o1. At the beginning of the ALWAYS_COMB block these o0 and o1 each have an initial value – let’s assume o0 has initial value A, and o1 has initial value B. The values will stay the same unless changed by further lines in the code.

The first line says “o0 = i0”. What is this doing? It is overwriting the current value of o0 with the input i0, and is leaving the rest of the values alone. So now, we delete the “A” value of o0, and replace it with i0. What happened to o1? Nothing. It stays at B. Notice how on the yellow sheet, to the right of the Verilog, we’ve written i0 under the column for o0, yet just let the B in o1 continue down.

The next line says “o1 = i1”. Similar thing to the previous paragraph. The B in i1 is thrown away, replaced with i1, while the value of o0 is left alone.

Next line, “if (o1 < o0)”. Take the current value of o1 and see if it is less than the current value of o0. How do I do a comparison? Look above at slide 213. What are the current values of o1 and o0? Check the table we’re building on the yellow sheet. O1 is i1, and o0 is i0, so (o1 < o0) is the same as (i1 < i0), which is implemented in a comparator. That comparator is drawn on the yellow slide for you.

What about the “if” portion of that line? It says “if the comparison is true, run the lines in the “then” portion. If not, leave those outputs alone”.

The next line is o1 = i0, but this only happens if the comparator returns TRUE. If the comparator returns FALSE? Keep o1 at its old value. So if the comparison is FALSE then o1 stays at i1, but if the comparison is TRUE then o1 becomes i1. This is just a mux, controlled by the output of the comparison, as shown in the yellow sheet. The new value of o1, at the end of this line of Verilog, is the output of that mux – sometimes i1, sometimes i0, all controlled by the comparator.

The next line is o0 = i1, which is pretty much the same as the previous line.

Finally we get to the “end”. The “result” of the ALWAYS_COMB block, the thing that the hardware must do, is react exactly the same as our reading of that Verilog code. So, what happens? O1 and o0 each are the outputs of muxes, controlled by a comparator. You can see that in the table of information, line-by-line, next to the Verilog on the yellow sheet. I clean that all up at the bottom of the page, showing the equivalent hardware.

BTW, what does this Verilog do? It makes sure that o0 is the smaller of i1 and i0, and o1 is the larger of i1 and i0. It is a simple sorter.

Slide 215 (the final slide)

On the previous slide we figured out how ALWAYS_COMB blocks become hardware. What happens if there is a FOR loop in the code? Well, it does the same thing – figure out what happens line-by-line, and create hardware that does the equivalent. But for the FOR loop, we just walk through the body of the loop as many times as the FOR loop specifies. So, for this code we do the body of the loop 3 times, once each for i=1, 2, and 3. Look at the yellow sheet and see.

Notice that there’s an “if” statement inside the loop body. “if (vals[i] != 2’b00)”. How do we do that? Go back to the early slides on advanced Verilog. That IF condition is true as long as both bits of vals[i] are not zero. What’s that? An OR gate...
So we have an OR gate as the condition of an IF – that’s just an OR gate on the control of a mux. If you look on the yellow sheet at “loop_1”, you’ll see the corresponding hardware. The value of “result” for the first iteration of the loop is the output of a MUX (actually 2 muxes, since result is a 2-bit-signal), controlled by an OR gate. That value of result enters the loop for the second iteration (loop_2), which hits another mux. Do the loop the full three times, and you get the hardware shown.

BTW, what does that hardware do? Take a look and try to figure it out...

...

What does it do? It makes RESULT be the highest entry in the VALS array that is non-zero. So, if VALS[3] is non-zero, RESULT is that. If VALS[3] is zero, but VALS[2] is not, RESULT is VALS[2]. Return the highest non-zero entry.

Okay, you can see the hardware, and that’s pretty simple. But, what happens if we made the loop iterate 127 times, not just 3? Well, you’d get roughly 127 muxes in series!!!! BIG circuit, with a HUGE critical path – each clock cycle you’ve got to go through 127 muxes in series. HORRIBLE circuit. Are you going to get only 1MHz out of it, something like that. Yeah, bad.

How could you do better? Design it differently. Here’s how I would do it:

1. For I = 0 ... 126, in steps of 2, compare entries I and I+1 via a mux like the one we already did – return VALS[i+1] if it is non-zero, or VALS[i] otherwise. You now have 64 outputs – let’s call these STAGE_ONE.
2. Take the STAGE_ONE array and do the same thing. Compare entries I and I+1. You’ll now have 32 muxes, and 32 results. Call these STAGE_TWO.
3. Continue doing this. There will be 16 entries in STAGE_THREE, 8 entries in STAGE_FOUR, 4 entries in STAGE_FIVE, 2 entries in STAGE_SIX, and a final mux for the overall result.

What did all of that do? Do the computation as a tree. With 7 muxes in series (though 127 muxes total) you can compute the final result. It is the same # of muxes as the previous version, but the critical path is 7 muxes in series, not 127.

BTW, being able to understand what our Verilog turns into is key in spotting improvements like that. Yeah, the original FOR loop is simple, and a decent way to do things if you were writing software, but my tree version is RADICALLY more efficient for hardware.

Okay, that’s all. We’re skipping the rest of the slides. If you are interested, take a look at the slides and you should be able to follow them, especially with a little Googling. Or use BING – if you do, your web search will actually be computed on ALTERA FPGAs, just like the ones you’ve programmed this quarter.

Have a good break, and see you all next quarter, or next year.