Verilog Lesson 3

tag correction

Verilog Lesson 3 by T. Miller

Before we go further, I figure we should have a short quiz.

Explain this code in detail:

```
module whatsit(a, b, d, c);
input [3:0] a, b;
input d;
output [3:0] c;
wire [3:0] e, f;
assign e = a & b;
assign f = a + b;
assign c = d ? e : f;
endmodule
```

Also, some information is being lost. Identify it.

Ok, on to the lesson. Continuous assignments like what we saw are powerful but limiting in their forms of expression. To augment is expressiveness, Verilog has what are called behavioral blocks or "always" blocks. Tonight, we'll just cover a handful of things that you can do. Behavioral blocks look a lot more like C programming, and they allow you to express algorithms in a way that looks sequential (even when it's not in hardware). In fact, we'll see later how to express sequential logic (logic with registers and clock signals).

The syntax for a behavioral block is:

```
always @(sensitivity_list) <statements>
```

The sensitivity list must include every signal that is an input to anything expressed in the behavioral block. Doing this incorrectly won't affect synthesis, but it will affect simulation. The sensitivity list is a sequence of inputs separated by the keyword "or":

```
always @(a or b or d) <statements>
```

Now, signals on the right-hand-side of an expression can be of either type wire or of type reg. But the left-hand-side must how always be of type reg:

```
reg [3:0] c;
always @(a or b or d) begin
   c = <something using the inputs>;
end
```

Basically the difference between reg and wire is that reg is the type on the lhs of an expression in a behavioral block. You'd think they'd make 'reg' mean a register (flipflops), but not necessarily, and this is a point of confusion for beginners.

The assignments we're using in this lesson, using the "=" operator, are called "blocking" assignments. For simulation, that means that the expression is computed and the signal is assigned its value immediately so that the result is available for subsequent statements. We'll explain non-blocking assignments next time. As a rule of thumb, we use blocking assignments in combinatorial logic and non-blocking assignments in sequential logic, and we avoid mixing them up.

You can express chains of operations to be done with all sorts of dependencies between them, and the synthesizer will just combine them together, perhaps creating rather long combinatorial chains. For instance:

```
wire [3:0] a, b, c, d, e;
reg [3:0] f, g, h, j;
always @(a or b or c or d or e) begin
   f = a+b;
   g = f & c;
   h = g | d;
   j = h - e;
end
```

And this is equivalent to:

```
reg [3:0] j = (((a+b) & c) | d) - e;
```

f, g, and h may or may not exist as identifiable signals once the synthesizer is done with this, depending on whether or not some other piece of logic uses them. If, for instance, f is never used anywhere else, the synthesizer will give you a warning, saying that it is assigned but not used. That's okay, because its result is used as part of the bigger expression. All it means is that nothing in the netlist is going to be named "f".

In behavioral blocks, we can now do some more interesting things, like if-statements, case-statements, loops, etc. For instance, to express the quiz problem behaviorally, we can do this:

```
reg [3:0] c;
always @(a or b or d) begin
   c = d ? (a & b) : (a + b);
end
```

Or we can do it this way:

```
reg [3:0] c;
always @(a or b or d) begin
   if (d) begin
       c = a & b;
   end else begin
       c = a + b;
   end
end
```

In the place of 'd', you can have any expression. Zero means false and any non-zero value is true.

Case statements are just like switch statements in C. They're a great way to express multiplexers over more than two inputs or to select between different actions.

The basic syntax is:

```
case (selector)
    option1: <statement>;
    option2: <statement>;
    default:  <if nothing else statement>;  // optional
endcase
```

Here's an example:

```
wire [1:0] option;
wire [7:0] a, b, c, d;
reg [7:0] e;
always @(a or b or c or d or option) begin
   case (option)
       0: e = a;
       1: e = b;
       2: e = c;
       3: e = d;
   endcase
end
```

This will synthesize to eight 4-to-1 muxes.

You can also do goofy things like this with case statements:

```
wire [3:0] x;
always @(...) begin
   case (1'b1)
       x[0]: <something1>;
       x[1]: <something2>;
       x[2]: <something3>;
       x[3]: <something4>;
   endcase
end
```

Logically speaking (and literally in simulation), the case statement walks down the list of cases and executes the first one that matches. So here, if the lowest 1-bit of x is bit 2, then something3 is the statement that will get executed (or selected by the logic).

The latter is great for 1-hot encoded state machines which we'll also have to cover another time.

Ok, we've seen case and if. How about a loop. For synthesis, a loop must have a fixed number of iterations, otherwise, it can't map to a fixed amount of logic. A good example perhaps is a priority encoder. Given a bit vector, we would like to know the highest bit that is set, and we'd like that number to be the output of our block.

```
wire [15:0] in_data;
reg [3:0] high_bit;
reg valid;
integer i;
always @(in_data) begin
   valid = 0;    // Assume invalid by default
   high_bit = 0;   // Set this to something by default to avoid inferring a latch
   for (i=0; i<16; i=i+1) begin
       if (in_data[i]) begin
           valid = 1;  // We found one, so set it valid
           high_bit = i;  // and note the number
       end
   end
end
```

Chew on that for a moment before going on. It's looping over the bits from the bottom up. Whenever it hits a bit, it notes the position. When the loop has completed, high_bit will contain the index of the highest 1 bit encountered. If you wanted the lowest bit, you would do it this way:

```
always @(in_data) begin
   valid = 0;    // Assume invalid by default
   high_bit = 0;   // Set this to something by default to avoid inferring a latch
   for (i=15; i>=0; i=i-1) begin
       if (in_data[i]) begin
           valid = 1;  // We found one, so set it valid
           high_bit = i;  // and note the number
       end
   end
end
```

One reason I use an integer for the loop count is just so loops like this will work. regs and wires are unsigned numbers. Only integer is signed. So when the loop counter rolls around from 0 to -1, only the integer will represent it correctly and have the >=0 comparison yield false.

Ok, I'm going to go out on a limb here. The next thing I'm going to show you is doable with everything we've covered up to this point, but it's really advanced for only being the third lesson. I suggest, therefore, that people pick it apart or perhaps suggest that it not be included in official lessons if it's a problem.

This list is dedicated to graphics, and so I'd like to show you how we can now do something that is fundamental to 2D graphics, which is to apply a planemask and a raster operation to two pixel values. One value is the destination pixel that was read from the target location in the framebuffer. The other value is the source pixel that was generated from drawing engine logic. OGA does this logic around about where it also does alpha blending between source and dest.

Here are your two input pixel values:

```
wire [31:0] src;
wire [31:0] dst;
```

And here's your planemask. The planemask is used to select which src bits are written out and which dst bits are preserved. A 1 in the planemask means to write the src bit and a 0 means to leave the dst bit alone.

```
wire [31:0] pmsk;
```

Now, let's combine the pixels:

```
reg [31:0] result;
integer i;
always @(src or dst or pmsk) begin
   for (i=0; i<32; i=i+1) begin
       if (pmsk[i]) begin
           // 1 means to use the src
           result[i] = src[i];
       end else begin
           // 0 means to keep the dst
           result[i] = dst[i];
       end
   end
end
```

Here's a functionally equivalent way to express this (but it may or may not produce identical logic):

```
wire [31:0] result = (pmsk & src) | (~pmsk & dst);
```

Next, there's the raster-op (ROP). Given any two bits, there are 16 ways to combine them. These

include the basics like AND, OR, and XOR, but also things like src-only (copy), dst-only (no-op), invert (~dst), copy-invert (~src), and the degenerate cases clear (0) and set (all 1 bits).

Since there are 16 ROPs, we can use a 4-bit number to express them. Have a look in "/usr/include/X11/X.h" and search for GXcopy. You'll find names for all 16 rops and their numbers.

The easiest way to express this in hardware is to concatenate the two bits together into a 2-bit number and use that to mux amongst the four ROP bits. If you don't see this right away, don't fret. I didn't either. I suggest you play about a bit with ROP numbers and input bit values and see it in action. Also, we're using the mux a bit strangely. Usually, we have inputs to the mux and some number to select amongst them on the select input. In this case, the "inputs" are a constant, and the pixel bits are being used as the select inputs. Here's the behavioral logic for that:

```
wire src_bit, dst_bit;
wire [3:0] rop;
reg result_bit;
always @(src_bit or dst_bit or rop) begin
   case ({src_bit, dst_bit})
       0: result = rop[0];
       1: result = rop[1];
       2: result = rop[2];
       3: result = rop[3];
   endcase
end
```

You may have noticed a pattern. In fact, we can use a bit-select to express the same thing more concisely:

```
wire result_bit = rop[{src_bit, dst_bit}];
```

Let's try an example. GXxor is function number 6, which in binary is 4'b0110.

If both src_bit and dst_bit are 0's, we want the exclusive or to be 0,and so we see that rop[0] is in fact 0.
If src_bit=0 and dst_bit=1, we want the result to be 1. Looking at rop[1], we do in fact find a 1.
If src_bit=1 and dst_bit=0, we want the result to be 1. Looking at rop[2], we find a 1, as expected.
Finally, if they're both 1's, the xor needs to be zero, and we find rop[3] to be 0.

Now that we have that out of the way, let's look at how we can combine ROPs and planemasks and pixels together in one chunk of logic:

```
module mask_and_rop(
   src, dst,
   pmask, rop,
   result);

input [31:0] src, dst, pmask;
input [3:0] rop;
output [31:0] result;

reg [31:0] result;  // override wireness, making result a reg

integer i;  // loop counter
always @(src or dst or pmask or rop) begin
   // Loop over all bits in the pixels
   for (i=0; i<32; i=i+1) begin
       if (pmask[i]) begin
           // If pmask bit is true, we combine src and dst
           // based on ROP
           result[i] = rop[{src[i], dst[i]}];
       end else begin
           // Otherwise, we preserve the dst bit
           result[i] = dst[i];
       end
   end
end

endmodule
```

We could also make that loop a little more concise like this:

```
always @(src or dst or pmask or rop) begin
   for (i=0; i<32; i=i+1) begin
       result[i] = pmask[i] ? rop[{src[i], dst[i]}] : dst[i];
   end
end
```

For completeness, Windows GDI defines something called a ROP4. A ROP4 combined four things:

- A source pixel (color)
- A dest pixel (color)
- A pattern pixel (color)
- A mask pixel (monochrome)

Typically, the pattern is 8x8 and logically repeated over the area of the drawing surface. The mask is typically 32x32 (IIRC), and it's also repeated. (Windows doesn't define a planemask, but we have it in the hardware for X11.)

Here's the logic to combine these together:

```
module src_and_dst_and_patt_and_mask_and_planemask_and_rop(
   src, dst, patt, mask,
   planemask, rop,
   result);

input [31:0] src, dst, patt, planemask;
input mask;
input [15:0] rop;   // Notice the 16-bit ROP4
output [31:0] result;

reg [31:0] result;  // override wireness, making result a reg

integer i;  // loop counter
always @(src or dst or pmask or rop) begin
   // Loop over all bits in the pixels
   for (i=0; i<32; i=i+1) begin
       if (planemask[i]) begin
           // If planemask bit is true, we combine src and dst
           // based on ROP
           result[i] = rop[{mask, patt[i], src[i], dst[i]}];
       end else begin
           // Otherwise, we preserve the dst bit
           result[i] = dst[i];
       end
   end
end

endmodule
```

Turns out to be painfully simple in the end.

Created by: hamish last modification: Wednesday 19 of December, 2007 [16:43:33 UTC] by vininim