Verilog Lesson 4

backlinks...

Verilog Lesson 4 by T. Miller

For this lesson, I don't think there's a whole lot to cover. We've been over arithmetic/logic expressions and behavioral code.
Sequential logic is basically just behavioral code that is sensitive to a clock edge rather than signal inputs. In this case, rather than just combinatorial logic, we also get flipflops (registers) inferred to store the signals we're assigning. With combinatorial logic, the computations in the always block are "instantaneous" with respect to the change in any input (they happen at the time of the change). With sequential logic, the inputs may change arbitrarily, but they only matter at the time of the clock edge, after which the signals are assigned.

We need to cover again blocking vs. nonblocking assignments. Blocking assignment (using the '=' operator) are called "blocking" because, conceptually, the logic pauses to finish the computation and make the assignment before moving on.

Nonblocking assignments (using the '<=' operator, only available in always blocks) post the computation to be assigned a time-delta later; conceptually, the logic runs past the assignment statement (does not block), setting it up but not completing it immediately.

Generally, we prefer to use blocking assignments with combinatorial code, and we prefer to use nonblocking assignments with sequential code. This is less of a convention than it is a result of the way we think about the two types of logic behaving. In your mind, you imagine that the clock edge occurs simultaneous for all logic that is sensitive to it (and the synthesizer makes sure that timing constraints make that effectively true). Unlike with combinatorial logic, it's perfectly normal for a physical signal to feed back on itself in sequential logic, such as with a counter. I'll give some examples to show the difference.

First, syntax. Here's a trivial sequential block with no reset:

```
module my_mod (clock, in, out);

input clock;
input [[5:0] in;
output [[5:0] out;

reg [[5:0] out;   // must declare reg type
always @(posedge clock) begin
    out <= in;
end

endmodule
```

Note that 'in' could be replaced by any expression that you could already do with a combinatorial always block. Also, in place of
'posedge' (transition from 0 to 1), you can use 'negedge' (from 1 to 0); this simply inverts the clock.

If you have a reset input, like this:

```
module my_mod2 (clock, reset, in, out);
...
input reset;
...
```

You can have different sorts of resets, such as synchronous active-high:

```
always @(posedge clock) begin
    if (reset) begin
        ... assign reset values to registers ...
    end else begin
        ... logic using your registers and other signals ...
    end
```

```
end
```

Synchronous active-low:

```
always @(posedge clock) begin
    if (!reset_) begin
        ... assign reset values to registers ...
    end else begin
        ... logic using your registers and other signals ...
    end
end
```

Asynchronous active-high:

```
always @(posedge clock or posedge reset) begin
    if (reset) begin
        ... assign reset values to registers ...
    end else begin
        ... logic using your registers and other signals ...
    end
end
```

Asynchronous active-low:

```
always @(posedge clock or negedge reset_) begin
    if (!reset_) begin   // we like to mark active low signals, like with _
        ... assign reset values to registers ...
    end else begin
        ... logic using your registers and other signals ...
    end
end
```

The way I'm coding this, with reset as the first thing we test in an 'if', with the 'else' for everything else, is a coding convention that synthesizer heuristics are designed to look for. There are other ways to do it that the synthesizer would recognize and all sorts of things that would work in simulation but not for synthesis. I code it this way, and all synthesizers know what I'm asking for; do it differently, and you risk the synthesizer doing something else that doesn't take advantage of built-in reset logic for your flipflops.

For real hardware, resets and other external signals are all you have to set initial values for your registers. There is no such thing as 'initialization', other than what you create explicitly based on signals.

Now, let's compare blocking and nonblocking assignments. Consider this code:

```
wire [[3:0] in;
reg [[3:0] x, y, z;
always @(posedge clock) begin
    x <= in + 1;
    y <= x + 1;
    z <= y + 1;
end
```

This will synthesize three adders and three physical registers. The time sequence (time zero is before the first clock edge) would go something like this (with the input set to, say, 3):

time: values
0: in=3, x='hx, y='hx, z='hx
1: in=3, x=4, y='hx, z='hx

2: in=3, x=4, y=5, z='hx
3: in=3, x=4, y=5, z=6

If you were to use blocking assignments, this is what you'd get:

```
always @(posedge clock) begin
    x = in + 1;
    y = x + 1;
    z = y + 1;
end
```

time: values
0: in=3, x='hx, y='hx, z='hx
1: in=3, x=4, y=5, z=6

Also, in the first case, all three regs will get registers. In the blocking case, if you don't actually use x and y, their registers will get ripped out, and you'll likely end up with a lousy way to assign in+3 to z.

Let's mix it up by changing the order of statements. I won't even bother rearranging the nonblocking code, because in this case, order doesn't matter. The first line assigns to x, but the new value isn't assigned until after the clock edge, so y gets the old value of x (plus 1). If you reorder the blocking assignments in reverse order, you'll get the same result as with the nonblocking assignments. Instead, let's just swap the last two statements.

```
always @(posedge clock) begin
    x = in + 1;
    z = y + 1;
    y = x + 1;
end
```

time: values
0: in=3, x='hx, y='hx, z='hx
1: in=3, x=4, y=5, z='hx
2: in=3, x=4, y=5, z=6

The key idea behind using nonblocking assignments is that you can take advantage of the time delta delay in the assignment, making your order of assignments generally irrelevant. The only time the order matters is when you assign to the same signal. For instance, this:

```
reg z;
always @(posedge clock) begin
    z <= 0;  // by default, z is zero
    if (in_signal) begin
        z <= 1;  // unless the input signal says otherwise
    end
end
```

Assignments like this are a really good way to set a default value that is followed by complex logic that may or may not assign something else to the signal. While case statements have a 'default', it can be painful to make sure you've covered every 'else' in nested 'if' statements.

Let's now try out a simple sequential circuit.

Here's a counter with async reset (to zero) and a 'load' signal that starts at some arbitrary input value:

```
module counter (clock, reset, load, in, out);
input clock, reset, load;
input [[6:0] in;
```

```
output [[6:0] out;
reg [[6:0] out;

always @(posedge clock or posedge reset) begin
    if (reset) begin
        out <= 0;
    end else begin
        out <= out + 1;
        if (load) begin
            out <= in;
        end
    end
end

endmodule
```

This is logically equivalent to:

```
always @(posedge clock or posedge reset) begin
    if (reset) begin
        out <= 0;
    end else begin
        if (load) begin
            out <= in;
        end else begin
            out <= out + 1;
        end
    end
end
```

It's common to do it either way, but sometimes one is more convenient or semantically revealing to the reader than the other. (With resets, always stick to the outlined convention.)

Note that if we were to put the load first and then the counter, the load would never have any effect, because the last assignment to 'out' is the one that takes.

Let's do a more complex design. Hamish has made a lot of progress with a Bresenham line generator. How about we try a fixed-point
version?

In this design, X and Y are 8-bit values. The major increment (whichever axis is steeper) is 1 or -1, and the minor increment is a
fraction, where we have another 8 bits of precision to the right of the binary point. This is lengthy, so do please ask questions to make sure you get it. (Also, it's untested. That's an exercise for you.)

```
module FixP_line(
    clock, reset,

    // Are we working on a line?
    busy,

    // For starting a new line
    MajStart,  // Initial value of major axis
    MinStart,  // Initial value of minor axis
    MajInc,    // Major increment: 0 -> -1, 1 -> 1
    MinInc,    // Minor increment
    Length,    // Number of pixels to generate
    XMaj,      // Is X the major axis?
    do_line,   // Start drawing

    // Output coordinates
    X,
    Y,
    valid_out
);

input clock, reset;
input [[7:0] MajStart;  // Integer
input [[15:0] MinStart;  // 8.8 FP
input MajInc;
input [[9:0] MinInc;   // 2.8 FP, signed (by the way we code it)
```

```
input XMaj, do_line;
input [[7:0] Length;

output [[7:0] X, Y;
output valid_out;
reg [[7:0] X, Y;
reg valid_out;

output busy;
reg busy;

reg [[7:0] counter;  // Counter over length of line
reg [[7:0] Maj;      // Major axis counter
reg [[15:0] Min;     // Minor axis counter

// The minor axis, as output, needs to be rounded.
// Here, I just add 0.5 by adding the highest fractional bit
// to the integer.  There are other and better ways to do this.
wire [[7:0] MinOut = Min[[15:8] + Min[[7];

always @(posedge clock) begin
    if (reset) begin
        // sync reset
        {X, Y, busy, valid_out, counter} <= 0;
        {Maj, Min} <= 0;
    end else begin
        // By default, we're not sending a pixel out
        valid_out <= 0;

        if (counter != 0) begin
            // Compute on line
            valid_out <= 1;
            X <= XMaj ? Maj : MinOut;
            Y <= XMaj ? MinOut : Maj;
            Maj <= MajInc ? (Maj+1) : (Maj-1);
            Min <= Min + {{6{MinInc[9]}}, MinInc};
            counter <= counter - 1;

            // This is the part I have trouble working out
            // in my head.
            // We want busy=0 when counter=1 so it rolls
            // straight from 1 to the next line length.
            // This way, the counter never reaches zero
            // as long as we have lines to draw, so we
            // have no idle cycles.
            // If counter==2 now, it's 1 on the next cycle.
            // If busy=0, there's a one cycle delay before loading
            // the next set of parameters, so the zero counter
            // will be replaced by the new length.
            // Your exercise:  Write a test harness and
            // use icarus to simulate this.
            busy <= counter > 2;
        end

        if (!busy && do_line) begin
            Maj <= MajStart;
            Min <= MinStart;
            counter <= length;
            busy <= 1;
        end
    end
end

endmodule
```

Refer to this list post for a full discussion on dealing with busy and start signals:
http://lists.duskglow.com/open-graphics/2006-June/005885.html

As another exercise, there is something very foolish I'm doing here with some inputs that will totally break things if you handle busy and do_line like in the above list post. What do I need to add to fix things?

BTW, people have their own coding styles, and I don't want to dictate how you do things. I'm sure mine's a bit quirky, because it places begins and ends like C, while the usual Ada/Pascal convention is different. But if you want me to look at a piece of code that you've written, I'll be able to read and understand it a lot quicker if your coding style is more like mine. My indents are 4 spaces, kinda like in the X server, but I don't ever use hard tabs.

Created by: hamish last modification: Tuesday 07 of August, 2007 [20:09:59 UTC] by chrisost

source history similar **1 comment**

RSS Wiki RSS Blogs

Übersetzen Sie diese Seite ins Deutsche
Traduzca esta paginación a español
Traduisez cette page en français
Tradurre questa pagina in italiano
Traduza esta página em português
翻译这页成汉语 **(CN)**
日本語にこのページを翻訳しなさい **(Nihongo)**
한국인으로 이 페이지를 번역하십시요 **(Hangul)**