Verilog Docs

Modules and I/O Ports

Modules are defined in the format below. Each file should typically contain one module with the same name as the file itself.

```
module myFirstModule(port1, port2, ...);
      ...
endmodule
```

Now, for the remainder of this example, suppose we have two ports. All ports must be declared as an input or output (or both – more on that later!). We define a module with one input and one output as follows.

```
module myFirstModule(
      input in,
      output [2:0] out
);
      ...
endmodule
```

The above indicates a module that takes in a 1-bit input named `in` – the name can be anything you want – and produces a 3-bit output `out`, whose constituent bits are labeled as bit 2 to bit 0.

Now assume we have a module `topLevel` that will "use" this module, i.e. integrate the functionality of the module above as part of its overall design, with wires `send` and `receive` corresponding to our module's input `in` and output `out`. To instantiate this module, you'd write the following code in `topLevel`.

```
      myFirstModule instanceName(
            .in(send),
            .out(receive)
```

```
    );
```

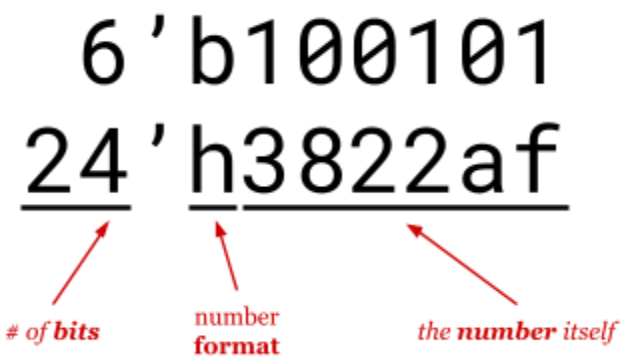
Note that the instance name can be anything you want – it's really only used during simulation and in compilation logs. You can also use a shorthand format:

```
    myFirstModule instanceName(send, receive);
```

In this syntax, it is implied that `send` maps to the first port in `moduleName` (a.k.a. `in`) and `receive` maps to the second. In many cases, this shorthand is acceptable, but sometimes for clarity's sake you should use the first one. Use good judgment!

Numbers

Numbers in Verilog are described with the following syntax:



Options for the number format include (b)inary, (d)ecimal, (o)ctal, and (h)exadecimal.

Optionally, the number of bits can be omitted, in which case the number will consist of the minimum bits required to contain the number. The number format, when omitted, will default to decimal.

X and Z

`X` and `Z` are special bit values in Verilog.

`X` represents a "don't care" bit. In typical digital circuits, the don't-care bit is used to indicate a value or case that is irrelevant to the circuit because it'll never be reached. In Verilog, `X` is used in simulations to represent certain errors in bit assignment; for example, if a bit is never assigned with `0` or `1`, or if it's driven by multiple combinational assignments, your simulation software may use `X` to represent it instead.

`Z` is a "high-impedance" bit. When your module assigns `Z` to a wire, it pulls the wire neither high nor low; as a result, other devices can drive that wire without causing a short. This is often used in digital communication protocols where multiple devices send data over the same wire.

Wires, Regs, Logic, and Bit Width

Modules have different ways of storing bits and passing them around. Recall that a `wire` is used exclusively to connect different ports and modules – much like a physical wire! On the other hand, a Verilog `reg` can synthesize to either a wire or a latch, flipflop, register, etc. For a more detailed description of the differences between these two elements, refer to this link. However, the TL;DR is that you would use `reg` any time you're driving your value from an `always` block.

All that being said, SystemVerilog provides us with a handy shortcut that allows us to never have to think about the differences: the `logic` keyword. `logic` can essentially be used to tell the compiler to automatically figure out based on usage whether a port should be a wire or reg.

Both `wire`s and `reg`s, like inputs and outputs, can contain multiple bits. We specify the number of bits by indicating a range of indices into those bits. Typical convention is to specify a larger number for the MSB (most significant bit) and a smaller number for the LSB. The following lines declare a wire and reg.

```
wire  [7:0] w = 8'b0001000; // w[3] is 1
reg   [3:0] r = 4'b0010;    // r[1] is 1
logic [1:0] l = 2'b01;      // l[0] is 1
```

In each line, the bits are numbered such that the leftmost bit is indexed with the number to the left of the colon and the rightmost bit is indexed with the number to the right of it.

Concatenation

Curly braces are used to "concatenate" or combine multiple bits or sets of bits into one wire or reg.

```
logic [3:0] l  = 4'b0010;

logic [7:0] l2 = {l, l};                    // l2 is 00100010
logic [3:0] l3 = {l[1], l[3], l[2], l[0]}; // l3 is 1000
```

The `assign` statement

The `assign` keyword is used to drive a `wire` or individual bits in a `wire` with combinational logic. It is used like so:

```
logic [7:0] w1, w2;

assign w1    = 8'b01010101; // w1 is now 01010101
assign w2[7] = 1;           // w2 is now 1XXXXXXX
```

Assign statements CANNOT be used inside `initial` or `always` constructs; if you use one, it should be outside any blocks.

The `always` block

The `always` block is a procedural construct – in other words, the lines inside all execute sequentially*. The block is triggered based on conditions specified in a *sensitivity list*, a list of ports that must change in order to trigger the `always` construct.

```
always @(posedge clk) begin
    ...
end
```

This `always` block will run every time the port `clk` has a positive edge (i.e. goes from low to high). You can also use `negedge` to trigger the block when the clock goes from high to low. Omitting the keyword will cause the block to trigger on both edges, i.e. any time `clk` changes.

If the `always` block is exclusively driven by a well-defined clock, your `reg`s inside will synthesize to properly-clocked registers and all will be well. If it's driven by something else, your `reg`s will synthesize to latches instead, which is a problem. Always trigger your `always` block with clocks!

You can also write an always block that synthesizes to a purely combinational circuit. This must trigger when *anything* changes, so we write it in the following way:

```
always @(*) begin
    ...
end
```

If you're using SystemVerilog (which, in DAV, we are!), we have some additional bells and whistles, one of which is a special construct that allows the synthesizer to detect various issues in your combinational circuit (such as latches) and raise errors.

```
always_comb begin
    ...
end
```

Assignment Operators

There are two different ways to assign a value to a reg or wire in Verilog. The first type is the one we've seen so far: *blocking* assignment, represented by an equals sign (=):

```
logic r1, r2;
always_comb begin
     r1 = 1;
     r2 = r1;
end
```

Blocking assignment is used in combinational blocks and indicates that we want the assignment to finish BEFORE we move to the next line. In the block above, `r1` gets assigned the value of 1, THEN `r2` gets assigned the value of `r1` – which is guaranteed to be 1 because the previous assignment has finished already.

In contrast, nonblocking assignments all occur in parallel. You must use these in sequential blocks because delaying the execution of an assignment in a clocked block could lead to timing violations. In the block below, `r1` gets assigned the value of some input `in` and `r2` gets assigned the value of `r1` from the previous clock cycle (because `r1`'s assignment in the current clock cycle hasn't finished).

```
logic r1, r2;
always @(posedge clk) begin
     r1 <= in;
     r2 <= r1;
end
```

Operations

Verilog supports many arithmetic operators that you've probably seen in other programming languages. Remember that unlike other languages you may be used to, in a typical module, these operations will function on two sets of *bits*; therefore, standard rules in bit math apply. Notably, this implies the following:

- Any and all overflow bits will be lost.
- If any operation causes a divide-by-zero error, the resulting output will be an X (don't-care bit).

```
w1 = a + b;  // addition
w1 = a - b;  // subtraction
w1 = a * b;  // multiplication
w1 = a / b;  // integer division
w1 = a % b;  // modulo
w1 = a ** b; // exponent (a^b)
```

Verilog also supports traditional bitwise operators. If one operand is shorter than the other, it gets zero-extended to match the length of the longer one.

```
w1 = ~a;    // bitwise NOT
w1 = a & b; // bitwise AND
w1 = a | b; // bitwise OR
w1 = a ^ b; // bitwise XOR
w1 = a ~& b; // bitwise NAND
w1 = a ~| b; // bitwise NOR
w1 = a ~^ b; // bitwise XOR
```

`if`-statements and `case`-statements

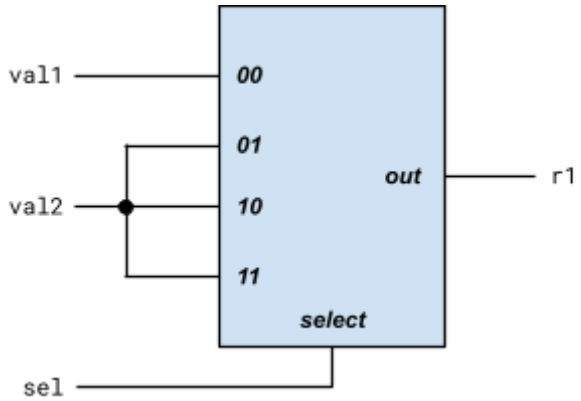
Much like "traditional" programming languages, Verilog `if-` and `case`-statements allow us to ensure that certain operations only happen when a condition is met. In combinational logic, these statements will either synthesize to a multiplexer or create a latch. In order to create a multiplexer for a reg `r1`, (the ideal result), the following conditions must be met:

- Every single possible value of the select lines must lead to a block of the `if-` or `case`-statement, and
- Every path through the statement must drive `r1`.

The following is an example of a valid combinational if-statement.

```
    always_comb begin
        if (sel == 2'b00)
            r1 = val1;
        else
            r1 = val2;
    end
```

This block would synthesize to the following multiplexer:



Case-statements are similar; you can use them to compare one specific wire or reg against multiple possible values. The syntax looks like this:

```
    always_comb begin
        case (sel)
           2'b00: begin
              r1 = val1;
           end
           2'b01: begin
              r1 = val2;
           end
           default: begin // if none of the other cases match
              r1 = val3;
           end
        endcase
    end
```

Arrays

If you want to maintain multiple wires or buses that logically make more sense as an array of independent ones, please don't just create a bunch of nets like `value1`, `value2`, `value3`, etc. Verilog also has arrays! The syntax to declare one is as follows:

```
wire  [7:0] w [0:3]; // 4x 8-bit wires:
                     // each wire is indexed 7 (MSB) to 0 (LSB),
                     // but the array itself is indexed 0 to 3

reg   [3:0] r [0:2]; // 3x 4-bit regs:
                     // each reg is indexed 3 (MSB) to 0 (LSB),
                     // but the array itself is indexed 0 to 2
```

You can even declare inputs and outputs as arrays. Just be careful to not confuse them with wires or regs of multiple bits! *The following things are NOT equivalent*:

```
reg [7:0] r1; // 8-bit reg, indexed 7 (MSB) to 0 (LSB)
reg r2 [7:0]; // array of 8x 1-bit regs,
              // indexed 7 (first element) to 0 (last element)

r1 = 8'b10101010; // works!
r2 = 8'b10101010; // throws an error!
```

Parameters

You may recall that C++ has **macros**, where you can **#define** things. Verilog parameters are quite similar. In a module declaration, you can define a parameter:

```
module randomModule #(PARAMNAME=defaultValue) (ports go here);
```

And from the main module, you can set your parameter value during module instantiation with

```
randomModule #(value) instanceName(ports go here);
```

Parameters allow us to create "customizable" modules without using FPGA blocks and registers to modify their behavior.

The `$clog2` function

SystemVerilog provides us with a convenient function called `$clog2()` that takes in a number as input and returns $\lceil \log_2(N) \rceil$, a.k.a. the **c**eiling of the **log** base **2** of N. In more practical terms, this function will return, for a given value of N, how many bits you need to represent N values with a distinct number for each one. For example, an FSM with 5 states requires at least $\lceil \log_2(5) \rceil = 3$ bits. It's a very helpful function for figuring out how many bits you need for a register or wire.

Frequently Asked Questions

Quartus and Questa

Q: Error deleting "msim_transcript"

When trying to run Questa, I get an error that says `error deleting "msim_transcript": permission denied.` *Whatever shall I do?*

A: If you're getting this error, it means you have Questa open already and you're trying to launch it again. Close Questa before you try reopening it from Quartus.

Q: Module missing from Questa

When I open Questa, I can't find my testbench in the list of modules. Why not?

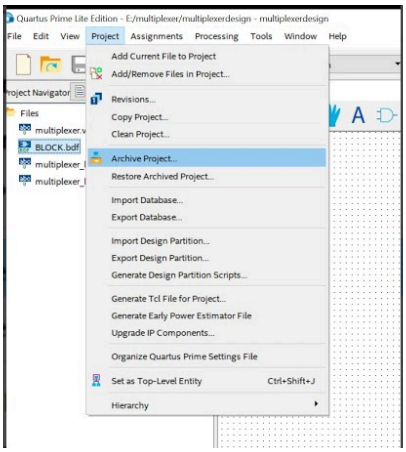
A: If this is happening to you or a loved one, you may be entitled to financial compensation!

Just kidding. This is usually because Questa has found some error with your testbench or other module that the Quartus compiler didn't catch. In the little window at the bottom labeled **Transcript**, you should be able to see a log from the Questa compiler, which will feature a line that says "Errors: #, Warnings: #" (but with actual numbers). If you scroll up above that, you should be able to see the exact module and line number where some sort of error exists. Fix those errors, re-run synthesis, and run Questa again – and you should be good to go!

Q: Making a QAR

How do I send my entire Quartus project to the leads for debugging?

In your project, on the top bar, click on Project → Archive Project. Use the screenshot on the right as reference.



Lab 1

Q: Illegal reference to net

The Questa compiler gives me an error saying `Illegal reference to net`. Why, oh why, is this happening?

A: In Verilog, we have a few different keywords we can use to declare ports; `wire`, `reg`, and `logic` are the main ones. Usually, this error indicates that you declared something as a `wire` when it should have actually been a `reg`. *This can even apply to outputs!* If the particular net or register causing problems is an output port, you may need to declare it with `output reg`.

Read the section Wires, Regs, Logic, and Bit Width for more information. You'll probably want to just use the `logic` keyword for all of your ports; this will simplify everything a great deal, and you don't have to think about which keyword to use.

Lab 2

Q: Timer counts weirdly

When I synthesize my alarm clock onto the FPGA, the timer isn't counting second-by-second; instead, it's jumping inconsistently, in weird increments, or there's other strange and unexplained behavior. Why is my life such a tragedy?

A: Congratulations! You have likely just discovered your first **timing violation**. Our guess is that your clock divider is driving the output clock combinationally instead of sequentially.

We're not 100% sure, but we believe the issue is due to the nature of combinational assignments and their effect on clocks. Unlike a sequential assignment to a register, a combinational assignment can happen arbitrarily whenever the inputs change; and even if we know that the inputs can only change on the clock edge, the compiler doesn't know that because we're telling it that the output clock is generated combinationally, so it probably synthesizes in hardware to something that isn't "clock-like." On the other hand, if you assign the output clock sequentially, it is guaranteed that the slower output clock will still remain in sync (i.e. the positive edges lining up) with the original input clock.

Therefore, the solution is simply to make your output clock a register that gets assigned its values on the clock edge. You should still compute the clock value combinationally, but the actual assignment of this computed value to the output clock register should happen in an `always @(posedge clk)` block.

Q: Buzzer doesn't buzz

When I test my clock divider, the buzzer isn't making any noise. What could be the issue?

A: If your buzzer doesn't buzz, try the following things:

1. Refer to the answer above regarding timing violations; this could be a source of buzzer errors as well.

2. Check your **pin assignments**. The input to your top-level module should be the 50 MHz clock pin on the FPGA, identifiable on the Pin Sheet as **MAX10_CLK1_50** or **MAX10_CLK1_50**. The output should be a singular GPIO pin, either in the GPIO rows at the top or the Arduino IO header row just below it. Check the datasheet for more details. (**NOTE**: This link downloads a PDF of the datasheet to your computer.)
3. Ensure that your clock divider produces an output clock with a **duty cycle of 50%**. In other words, your output clock should be low for half of the counter values and high for the other half.
4. Verify that the output frequency you're generating is within the **range of human hearing**, i.e. somewhere in between 20 Hz and 20000 Hz.

Q: Port is multiply driven

I keep getting an error telling me that a port is "multiply driven." What does this mean? Why does everything hurt?

A: If your port is multiply driven, it usually means that some part of your module is attempting to give the same port multiple values at the same time. In hardware, doing something like this would theoretically result in a short, which is why the synthesis tool usually just does something unexpected in this case. There are a number of different things you might have done to cause this error:

1. Driving a port from both a combinational and sequential `always` block.
2. Driving a port using an `always_comb` block and an `assign` statement.
3. Driving a port from multiple `assign` statements or multiple `always_comb` blocks.
4. Driving a port from multiple distinct sequential `always` blocks.

Basically, the only situation in which you're *allowed* to "multiply" drive something is if you give a register an initial value and then thereafter drive it with only ONE sequential `always` block.