

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:

6 ' b100101
24 ' h3822af

The diagram illustrates the syntax of Verilog numbers. It shows two examples: `6 ' b100101` and `24 ' h3822af`. The number `24` is underlined, and the number `h3822af` is also underlined. Red arrows point from the underlined numbers to the text `# of bits` and `the number itself` respectively. Another red arrow points from the `h` in `h3822af` to the text `number format`.

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 a latch, flipflop, register, etc.

You would typically use `reg` any time you’re driving your value from a clocked `always` block, because you use these blocks whenever your system requires *memory*; in other words, you want to reliably store a value over time. In an `always_comb` block, you usually assign values to wires; this is because the value of a wire changes instantly based on the current values of the inputs, without reference to the past values.

Note that you cannot drive the value of a register from multiple places (within both a combinational and a sequential block, for instance).

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 wires and regs, 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'b00001000; // 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.

What if your left-hand and right-hand logic elements have different bit widths? You'll want to take care to ensure this doesn't happen very often in your code, because issues arising here are tricky to debug. However, the behavior goes like this:

```
logic [3:0] a = 8'b00001000; // Case 1
logic [7:0] b = 4'b0010; // Case 2
```

In the first case, we are assigning an 8-bit value to a 4-bit element. The result is the most-significant four bits will be truncated. Then, $a = 4'b1000$.

In the second case, we are assigning a 4-bit value to an 8-bit element. If the right-hand side is signed, then an arithmetic sign-extension (MSB of RHS operand duplicated, whether it is 0 or 1), and a logical extension will be used if the right-hand side is unsigned. Here, $b = 8'b00000010$.

In SystemVerilog, values are unsigned by default; an explicit declaration is needed for something like a logic or reg value to be treated as signed. Also, mixing signed and unsigned operands results in *all* operands being treated as unsigned.

Concatenation

Curly braces are used to “concatenate” or combine multiple bits or sets of bits into one wire or reg. Read more [here](#).

```
logic [3:0] l1 = 4'b0010;

logic [7:0] l2 = {l1, l1}; // l2 is 00100010
logic [3:0] l3 = {l1[1], l1[3], l1[2], l1[0]}; // l3 is 1000
```

Left-hand side concatenation is also possible; see the examples [here](#). Think about why the example marked “invalid” fails, and when you would want to use left versus right-hand side concatenation!

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 `regs` inside will synthesize to properly-clocked registers and all will be well. If it's driven by something else, your `regs` 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:

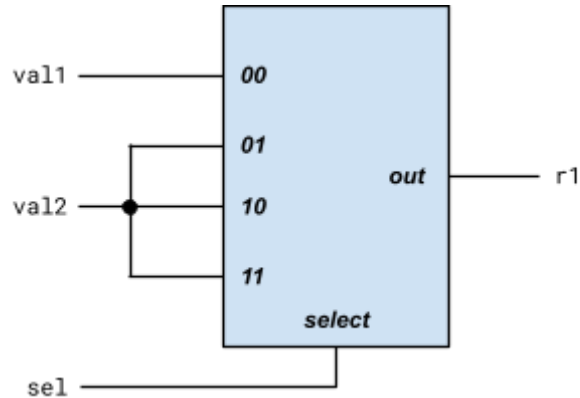
- Every single possible value of the select lines must lead to a block of the `if`- or `case`-statement, and
- Every path through the statement must drive `r1`.

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 **ceiling** 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?

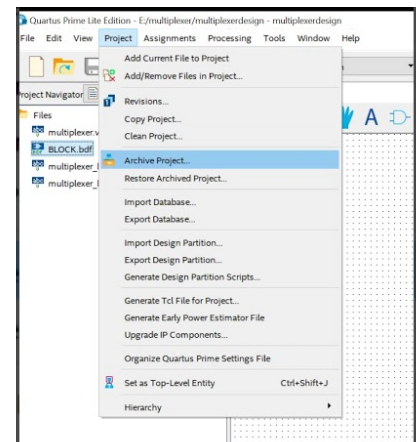
A: If this is happening to you or a loved one, you may be entitled to financial compensation!

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 combinatorially 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 combinatorially, 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 combinatorially, 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.

SystemVerilog Example [IN PROGRESS]

Let's take a look at a small project. The inputs are five switches; two correspond to one operand, the next two correspond to a second operand, the fifth controls an operation.

Compare the following two implementations:

A. Calculatorinator 1

`calc_top.sv`

```
module calc_top(  
    input sw,  
    input [2:0] operand1,  
    input [2:0] operand2,  
    output [3:0] answer  
);  
  
    calc CALCULATOR(.mode(sw), .op1(operand1), .op2(operand2),  
    .ans(answer));  
  
endmodule
```

`calc.sv`

```
module calc(  
    input mode,  
    input [1:0] op1,  
    input [1:0] op2,  
    output [3:0] ans  
);  
  
    assign ans = mode ? op1 + op2 : op1 - op2;  
  
endmodule
```

We can convert our top-level module into a testbench as follows:

```

calc_top.v
`timescale 1ns/1ns

module calc_top(
    //    input sw,
    //    input [2:0] operand1,
    //    input [2:0] operand2,
    output [3:0] answer
);

    reg sw = 1'b0;
    reg [2:0] operand1;
    reg [2:0] operand2;

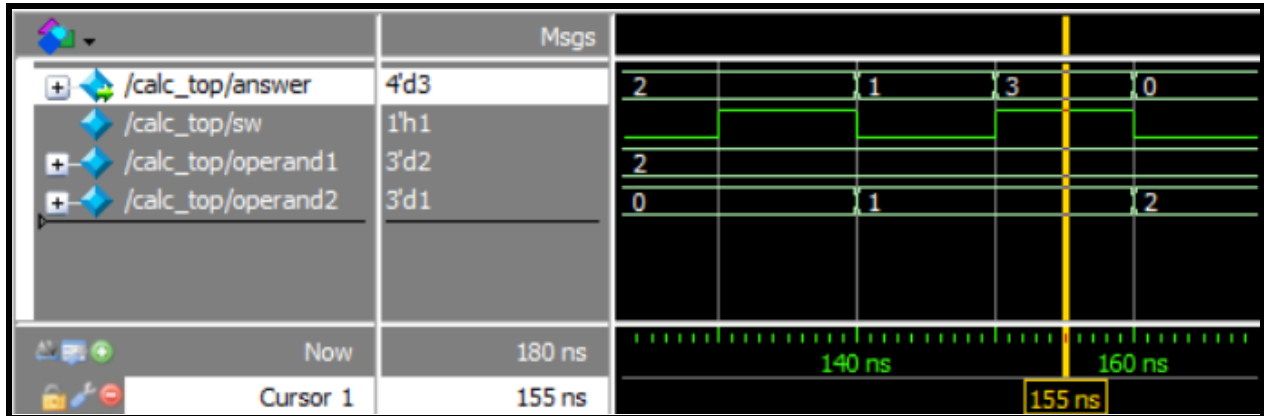
    calc CALCULATOR(.mode(sw), .op1(operand1), .op2(operand2),
    .ans(answer));

    initial begin
        for (int i = 2'b00; i < 2'b11; i = i + 2'b01) begin
            for (int j = 2'b00; j < 2'b11; j = j + 2'b01) begin
                operand1 = i;
                operand2 = j;
                sw = 1'b0;
                #10;
                sw = 1'b1;
                #10;
            end
        end
    end
    $stop;
end

endmodule

```

In Questa, here is an example from waveforms:



After a delay of 155 nanoseconds, the values of operand1, operand2 and sw are as shown. Since sw is 1, we add the operands to get the value of 3 for answer.

B. Calculatorinator 2

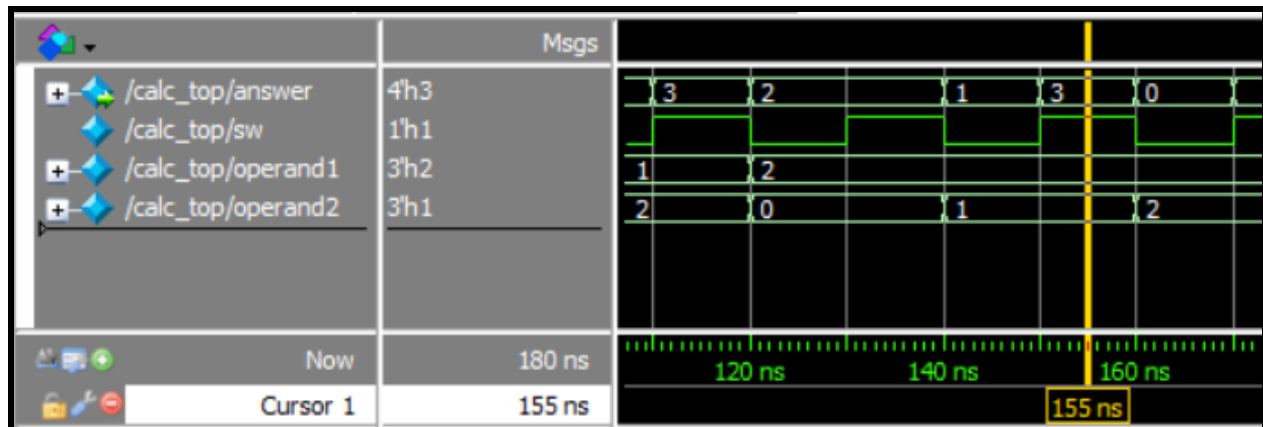
We put the assignment to ans inside an always_comb block and declare the output as reg.

```
calc.v
module calc(
    input mode,
    input [1:0] op1,
    input [1:0] op2,
    output reg [3:0] ans
);

    always_comb begin
        ans = mode ? op1 + op2 : op1 - op2;
    end

endmodule
```

The Questa output is:



The output is identical. So what's the difference between the implementations?

The `assign` statement requires that the left-hand-side be a wire; constantly driven. So, in the first implementation we do away with `always` blocks and simply use the `assign` statement to drive `ans` with the result of the operation. The logic on the right-hand-side synthesizes to some combination of gates, multiplexers, decoders – memory-less elements.

In the second implementation, the `always_comb` block executes only when any of the input signals change. `ans` is updated accordingly, but in between signal changes it needs to retain a value. That value needs to be based on the current values of the input signals alone.

Suppose there was a case statement inside the `always_comb` block. If for every branch within, `ans` is assigned a value, then the `always_comb` block will synthesize to strictly combinational logic, which is what we want. Otherwise, if for some branches `ans` is *not* assigned, then a latch is inferred; in the absence of a new value to assign to `ans`, the old value is retained and memory is implied.

Long story short, `reg` (or `logic`) is used for `always_comb` blocks to create complex combinational circuits. Though `reg` and `logic` both enable memory, the key point is that by ensuring every possible input combination results in some value being assigned to a `reg` variable, the `reg` variable depends only on the current values of the inputs and thus behaves exactly as a combinational circuit should. It thus

synthesizes to a purely combinational circuit. We have basically used memory to create a memory-less circuit.