SystemVerilog intro

Table of Contents

Variable Types

Input:

What is it:

An input is a type specifically for modules in a design file. They are specifically of type wire, however; you are not allowed to change the value of an input inside a module. While I doubt SystemVerilog will give you an error, it is extremely bad practice.

When to use it:

Use it inside a module in your design file to determine outputs. It can also be used like a clock edge if used as the parameter for an always block causing the code inside the always block to run whenever the input goes high or low (depends on what you set it to).

Output

What is it:

An output is also a type specifically for modules in a design file. They are originally of type wire but can be typecasted to a reg using output reg. OUTPUT REG IS USELESS FOR 341. Don't use output reg because it is only used for local memory to a specific module. A regular output has an indeterminate value at the start of your module and should be set by the end of your module. Whatever is set will be passed out of this module back into the testbench or previous module that called this module.

When to use it:

Use it inside your design file to get values out of your module that you can use.

Reg

What is it:

A register by definition is a combination of 2 or more flip-flops. A flip-flop in circuitry simply holds a bit and does not change the value of the bit until a clock signal tells the flip-flop to update its value to the new incoming value. Registers in SystemVerilog work the same way. You can't set a value in a register unless it is in an always blocked controlled by a clock edge, or at the start of your program denoted by initial. You can't set the value in a register any other way. This is contrary to wires which will continually update.

When to use it:

For the context of 341, you only need to use it for inputs to a module from the testbench. The unchanging aspect of registers make it optimal for inputs that shouldn't change value in the middle of executing a module. Outside of 341, registers are used for behavioral logic to write SystemVerilog code in a C like fashion. Since most systems are controlled

by clocks, registers update inside clock controlled always blocks and assign their values to outputs creating the overall system.

Wire

What is it:

A wire is literally a strip of metal. It holds no value until you put something into it which makes it optimal for output values. However, the key aspect of wires is that as soon as you change the input to the wire, the output of the wire immediately changes in contrast to registers which require a clock. Therefore, you don't need an always block to assign values to wires (you actually can't assign values to wires inside an always block). This also means that you can only apply gate and module logic to wires.

When to use it:

Use it as temporary variables inside a module to hold results of gates or other logic. Also use it to pass in outputs to a module from a testbench.

Array

What is it:

In actual hardware, an array is very similar to a bus. An array can be of type input, output, reg, or wire. The size inside the brackets is inclusive on both ends and typically has the larger value first. Having the larger value first means that your MSB will be on the left of your value and your LSB will be on the right. You can switch this order by changing the larger value to second value instead of the first.

When to use it:

Anywhere you need to have a collection of bits that share a purpose and make organization easier. For instance, you could make an 8-bit adder using 17 inputs and 8 outputs, but it's easier to use 2 8-bit array input, a Carry In and 1 8-bit array output.

Double Array

What is it:

This doesn't actually exist in hardware, but it makes it easier to access values greater than one bit. A single array can only hold one bit at each location in the array. A double array allows the user to hold a larger value at each array index. This size is specified by the first pair of brackets and the size of the array is denoted in the second pair of brackets.

When to use it:

You typically will only use it to read in memory values since SystemVerilog does not allow inputs or outputs to be a double array, only wires or regs. Really annoying, but it makes sense in terms of hardware.

A short bit on scalared vs vectored

A scalared variable means that you have an array of values in which you can access individual bits from the array. Vectored arrays on the other hand require all bits to be used or changed at the same time.

Additional Note

On a separate note, there are about 15 other data types available in SystemVerilog, however you will not use them for your college career at UB and they have too many caveats to explain in what was supposed to be a short description of SystemVerilog.

Operations

Module:

The keyword module dictates the start of a method in SystemVerilog. Modules that are used to implement a design will have inputs and outputs. Modules that test designs won't need inputs or outputs but will supply them to the design to test its behavior.

Endmodule:

The keyword endmodule is like the closing brace for a method. One key part of endmodule is that it is the only non-directive line that doesn't require a semicolon after it.

Initial:

The keyword initial is used to run at the very start of the program. In a testbench, it is the first piece of code run after variable declaration. In a module, it is the first piece of code to run once the module is called, again after variable declaration. You can only assign values to registers in the initial block, however this is useful for setting initial inputs to a module or reading from files.

Begin:

Begin is like the opening brace for an initial or always block. It encapsulates all the code from begin to end inside the initial or always block if there is more than one line required.

End:

End is like the closing brace for an initial or always block.

Always:

An always block runs like a while loop that runs every clock edge. The way to dictate the clock to use is with the character @ followed by the variable that acts as the clock in parentheses. For example, always @(posedge a) sets the always block to only run when a goes from 0 to 1. Without posedge, the always block will run on both posedge and negedge. Without any variable present, the always block runs continuously with the system clock acting as its clock. A key part of an always block is that you can only update registers inside.

Assign:

Assign is the most complicated of all the keywords and stands for continuous assignment. The basic gist of assign is that whenever you want to set a wire equal to something, it must do so in such a way that it is not reliant on a clock or external variables to update. Assign allows the value of the wire to continuously take the value of whatever it is being equated to even if that value changes. Because of its continuous nature, assign can't be used in an always or initial block and can only be used on wires or outputs. This is especially useful if you want to set a part or the entire value of a register or input to an output or wire as seen in inst_mem_split from lab 7.

`include:

Include allows you to import other DESIGN FILES IN THE SAME PLAYGROUND into a design or testbench. The syntax is as follows: `include "ALU1bit.sv" No semicolon and the character in front of include is on the top left of your keyboard (not the apostrophe).

$finish:

The finish keyword tells your testbench, and therefore your entire program, how many timescales it should run for. This time should be directly dependent on your always block in your testbench that tests your module. The syntax is as follows: initial #___ $finish;

$display:

The keyword display is the exact same as println. One key part of display is that it only runs once when it is called. The syntax is as follows: $display ("Hello World"); Semicolon.

$monitor:

The keyword monitor is similar to display but prints every time one of the variables referenced in its printing changes value. You can have different types of values print out. %b is binary, %d is decimal, #h is hex (I think). Syntax is as follows: $monitor("This is x: %b", x);

`timescale:

The keyword timescale needs to be in both your design and testbench. This tells you how fast your code will run and is typically only useful if you want to actually see your values update vs whether you want to see how fast the code can run if implemented on actual hardware. A larger gap in timescale means values update slower and are more visible.

Modules

Modules allow you to create and test code in SystemVerilog. Therefore, it is imperative you know how to create and call modules correctly. When creating a module, you should have the keyword 'module' followed by the name of the module and then in parentheses, just the names of all inputs and outputs to the module. Then it should end with a semicolon:

```
module example_design(input1, input2, a, b, output1, x);
```

In this case, example_design is the name of the module. It is impossible to tell at this point whether any of the variables are inputs or outputs yet or their size. So, right after declaring the module, you should also declare your inputs and outputs and their sizes

```
module example_design(input1, input2, a, b, output1, x);
input input1, input2;        //your inputs can be separated into different lines.
input [7:0] a, b;            //inputs of different sizes must be separated. In this case, both
                             //a and b are eight bits long.
output output1;              //output can also be separated onto different lines and must be
output [7:0] x;              //separated by size.
```

Now that you have a module, you can write your gate logic inside and end the module using the keyword endmodule.

But if you want to call a module inside either your testbench or different module, you need to follow a rather unconventional syntax. There is a reason for this syntax though which will be explained after the syntax example.

The syntax for calling a module starts with the name of the module, followed by a random variable name, and then port assignments and a semicolon. The following example shows all the steps to call a module:

```
reg input1_tb,  input2_tb;      //declaring my inputs using registers. These could also be
reg [7:0] a_tb, b_tb;           //inputs or wires if the wires are preset.
wire output1_tb;                //declaring my outputs using wires. These could also be
wire [7:0] x_tb;                //outputs if calling from a module.
initial                         //This block is used to initialize my register values.
    begin
        input1_tb = 0;          //These two can only be 0 or 1 (1 bit max).
```

```
        input2_tb = 0;
        a_tb = 0;              //These can be any value from 0 to 255 unsigned (I think).
        b_tb = 37;             //The value is then translated into an 8-bit array.
    end

example_design dut (.input1(input1_tb), .input2(input2_tb), .a(a_tb) , .b(b_tb) ,
.output1(output1_tb) , .x(x_tb) );          //this calls the module using the inputs and outputs
```

As you can see when I called the module, there are a lot more variable names and dots and commas. The way the syntax works for the portlist WHEN CALLING A MODULE is that you start with a '.' to gain access to the module and then the VARIABLE NAME FROM THE MODULE YOU ARE CALLING. You assign a value to this variable in the parentheses following the variable name with a VARIABLE NAME FROM THE CURRENT MODULE.

The reason for this syntax is because you are literally assigning a variable, not a value, to a pin in a circuit. Your module simulates a real circuit so the '.' connects the pin from your module to the variable you assign it to in your testbench. Or if you call from a module, you are literally connecting two pins with the dot. Please note that YOU DON'T NEED TO PUT THE SIZE INTO THE PORTLIST EVER.

Design vs Testbench

So, as I described before, your design simulates a literal circuit, but doesn't have anything to run it. That's where your testbench comes in: to feed data to the circuit so you can see how it works before making the actual hardware. That's what makes SystemVerilog a hardware descriptive language (HDL). Since your testbench is only testing a module, it doesn't need inputs or output for itself. So, its portlist is left empty or not included.

When you call a module from the testbench (or anywhere else) and pass in the inputs or outputs, that module runs constantly and infinitely until the end of the program. However, if you don't change any of the inputs of that module, your outputs also won't change because that's simply how circuitry and black boxes work.

But, if you change even a single value of an input into a module, that module will immediately update (in the context of CSE 341) its outputs. To test your circuit, that means you only have to call the module once and then start changing the inputs to every possible combination of inputs possible to test all the outputs. The always block in your testbench lets you update your register inputs every few time increments. And you can have more than one always block in one module in which they will run simultaneously.

You'll notice that if you only call the module and update the registers in an always loop, you won't have any idea what the module is actually doing. That's because you need to print out the values so you can see what is happening. This is where monitor comes in handy. Just like module calls, monitor also runs constantly and infinitely and prints a line whenever one of its variable inputs changes.

Therefore, the key contrast between the testbench and design is that your design is running constantly based on what you feed it from the testbench.

One extra note, since you are getting values out of a module into your wire outputs from your testbench, you can actually use those wires as inputs into other modules called after that module. You can see an example of this in the example code at the end.

Memfiles

Memfiles that end in .dat (short for data I'm pretty sure) are specific files to read information in from while coding in SystemVerilog. You already have the code necessary to extract data from a file using the keyword $readmemh. The h at the end stands for hex and can be changed to b for binary. readmemb can only read binary values while readmemh can only read in hex. So if it reads a 0 with readmemh, it's translated to 0000 binary.

The way the module provided in lab 7 works is that the memfile is read in inst_mem using the initial block and assigned into the variable RAM. RAM is a register which allows it to be updated in an initial block. But to get the data out of the module, you need to put it into an output. Therefore, assign is used at the end to continuously update the value of the output variable data based on what the address is. The address input is simply picking what number instruction to read.

After pulling data out of RAM and returning it, your testbench now has access to it through the output variable. So, it can use this value AFTER THE MODULE CALL TO INST_MEM for parsing, and eventually, ALU/processor operations.

You can edit the Memfile by removing or adding lines and the size that can be read is only dictated by the size of RAM. In the code, RAM is a 32-value array where each value is 32 bits (or 8 hex digits or one instruction in MIPS). You can change the first bracket values to change the size of each value read in or change the second bracket values to change the number of instructions read in.

Memfiles will be extremely important for testing because it's much easier to edit a memfile than a testbench.

EXAMPLE CODE

Edit code - EDA Playground

Please use this link to view the code with comprehensive comments.