Introduction to Structural Verilog

CS141 Spring 2019

Introduction

Verilog is both a language and a simulator for describing digital systems. Using Verilog you can describe a model of a digital circuit as logic gates, and then use it to simulate how signals will propagate through the system over time.

When we use Verilog, we talk about two different "flavors:" structural and behavioral Verilog. Structural Verilog refers to using modules and wires explicitly to build circuits. Behavioral Verilog uses the inference features of Verilog to infer what a circuit should be based on code which is often higher level and should be executed multiple times (i.e. on a clock edge). Structural Verilog is usually used for making combinational circuits and behavioral Verilog is used for making sequential circuits. For now, we will only worry about structural Verilog. We will begin using behavioral Verilog for synthesis starting with programming assignment 3.

It is important to note that there are many differences between Verilog and a traditional programming language. Since your code will be compiled to static logic gates you shouldn't think of your code as being executed in a procedural way. It is often most helpful to write Verilog after designing a schematicfor your circuit.

These examples have been adapted from *Logic Design and Verification Using SystemVerilog* by Donald Thomas, along with the figures.

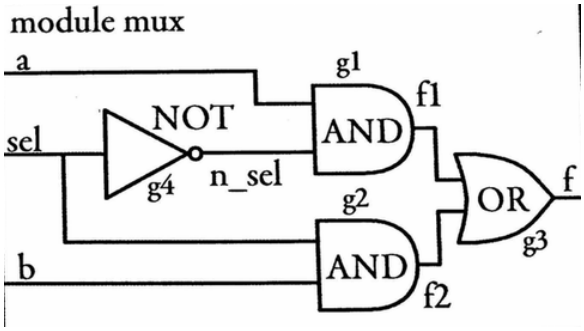
A Structural Example

In Verilog the *module* is the basic building block. Every module defines inputs and outputs as well as any logic for the contents. We will begin with a simple combinational circuit: a two-to-one multiplexer. This gate takes 3 inputs a, b, and sel and produces one output f. We define it as follows: if sel is low, then f=a, otherwise f=b. We can express this using AND, OR, and NOT gates as $f = a \cdot \overline{sel} + b \cdot sel$.

```
module mux(f, a, b, sel);
    input wire a, b, sel;
    output wire f;

    wire n_sel, f1, f2;

    and #2 g1 (f1, a, n_sel),
           g2 (f2, b, sel);
    or #2 g3 (f, f1, f2);
    not g4 (n_sel, sel);
endmodule
```



We instantiate two *AND* gates into the design on lines 5 and 6. On line 7 we add an *OR* gate and on line 8 we add a *NOT* gate. Each gate instance is given a name (`g1` through `g4`) corresponding to the gates in the diagram. When we instantiate a gate we pass in the wires that the inputs and outputs should be connected to. In this example the output if the first argument and the inputs follow. The `#2` indicates that the specific gate has a propagation delay of 2 time units. The *NOT* gate is not given a delay so it computes in 0 time units.

We should note that primitive gates such as *AND*, *NAND*, *OR*, *NOR*, *NOT*, and *XOR* are predefined in Verilog.

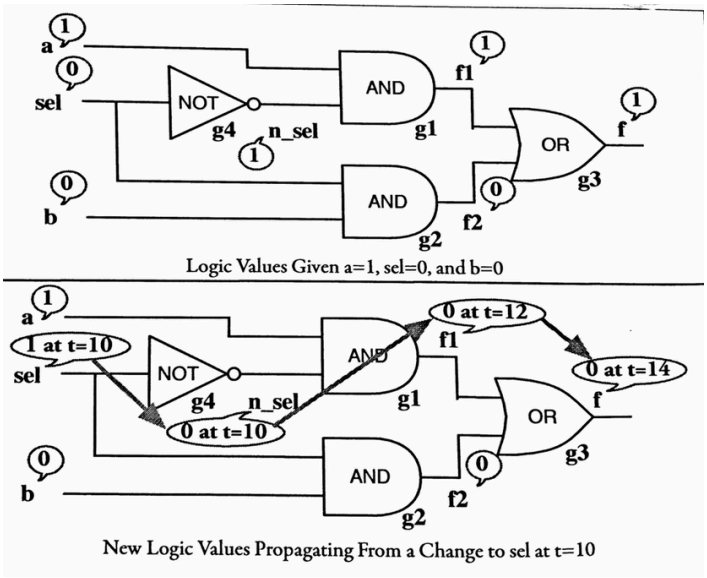
In this example we see several important concepts: instantiation, interconnection, and modules. We see how modules can be instantiated, how to connect their inputs and outputs via wires and how this can help us create a black box that can help to isolate complexities from the rest of the program.

We should note how this is different from a traditional programming language such as C. Always keep in mind that your Verilog code is being compiled in hardware. Thus the order that we instantiate the gates in does not matter. After compilation we can simulate the behavior of the module using Verilog's simulation features. Since we have given the gates in the example propagation delays, we can consider how a signal might propagate over time.

Simulation

Let's assume that the inputs to the module have been constant for a long time and that the input values are `a=1`, `b=0`, and `sel=0`. Thus `n_sel=1`, `f1=1`, and the output `f=1`. If `sel` is changed to be 1 at time 10, some of the values in the circuit will change, but there will be a time taken for the signal to propagate.

The following figure illustrates the propagation of the signal.



Note that the flow of logic values does not correspond to the order in which the gates were instantiated.

Let's see how to simulate this in Verilog. We can create a testbench module:

```
module testmux;
    reg a, b, sel;
    wire f;

    mux uut (f, a, b, sel);

    initial begin
        $monitor ($time, " a = %b b=%b sel=%b f=%b", a, b, sel, f);
        a = 1;
        b = 0;
        sel = 0;
        #10 sel = 1; // switch sel to 1 after waiting 10 time units
        #6 $finish;
    end
endmodule
```

First note that we declare the inputs as `reg`. This means that they serve as variables that can store values rather than just as interconnects between modules. We will see `reg` in greater depth later but for now we use it for assigning test values in a testbench.

The *initial* statement is a procedural statement that begins executing when the simulation starts. Procedural here means that you can read the statements within the block sequentially, that is they execute one after the other like a traditional language. In this example the initial statement begins executing at time 0. First it executes the `$monitor` statement. This is a simple debugging statement which will print the time and the string specified whenever any of the variables are modified (if you read the optional detailed section below, you will see exactly when the `monitor` statement is executed).

We then assign initial values to *a*, *b*, and *sel*. The `#10` tells the simulator to wait 10 time units before executing the next statement, when we assign *sel* to 1. Finally we simulate for another 6 time units before terminating the program.

When we execute this testbench we see the following output:

```
0 a = 1 b=0 sel=0 f=x
4 a = 1 b=0 sel=0 f=1
10 a = 1 b=0 sel=1 f=1
14 a = 1 b=0 sel=1 f=0
```

This is exactly what we predicted earlier: when we flip *sel* at *t=10* we only see the output *f* change at *t=14*, since the signal takes time to propagate.

A note about `f=x`: all logic variables are set to `x` when the simulator starts. Thus to use a variable you must give it an initial value. In this example at time 0 the inputs have not yet propagated to the gate output.

A More Detailed View of Simulation (optional)

The previous section provides a basic explanation of how the Verilog simulator functions by keeping track of times and values, either through the gate execution model (where it calculates gate logic and propagates the values) or through the execution of procedural statements in an initial block. This section gives a more detailed view of the internal structure of the Verilog simulator. The main feature of the simulator is its ability to keep track of time, or as it is often called *virtual time*. The simulator maintains a time-ordered list of *update events* which represent the change in a value at a certain virtual time. In the same list, it also keeps track of *execution events* which specify that a certain procedural

statement should begin executing at a certain time (for example any statement in an `initial begin` block).

The simulator sets all logic values to `x` at startup and then works as follows:

1. The simulator *retrieves* all events for the current time.
2. It goes through the list and *updates* the current state of the logic (values at each gate).
3. Then it *follows* connections in the logic and propagates these changes to further gate inputs.
4. It *executes* the followed gate models with the new inputs to see if the outputs change and if they do then it *schedules* the changes in the event list as new *update events* to be executed in the future.
5. If the simulator sees a delay indicator (`#n`) in a procedural block, it schedules the next procedural statement as an *execution event* to execute at `current time + n`.
6. The simulator repeats this cycle until there are no more events.

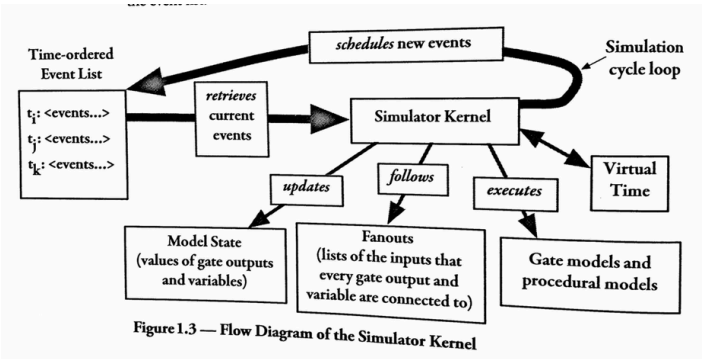


Figure 1.3 — Flow Diagram of the Simulator Kernel

This isn't the full explanation of the simulator but it provides more depth than the previous section. Specifically we learned that there are two kinds of events: *update events* which schedule new values for gate outputs at a specific time and *execution events* which specify that a certain line should begin executing at a given time.

Let's consider the previous mux module with a new testbench:

```
initial begin
    $monitor($time, " a=%b, b=%b, sel=%b, f=%b", a, b, sel, f);
    a = 0;
    b = 0;
    sel = 0;
    #12 a = 1;
    #6 $finish;
end
```

Since we change `a` to 1 after 12 time units, we expect that after 4 time units `f` will change to 1. Indeed if we run this we see that `f` changes at time 16:

```
 0 a=0, b=0, sel=0, f=x
 4 a=0, b=0, sel=0, f=0
12 a=1, b=0, sel=0, f=0
16 a=1, b=0, sel=0, f=1
```

Now let's reason about how exactly this block will be simulated. First the simulator schedules all *initial* and *always* (we'll learn about these later) blocks for execution at time 0. Since the simulator has just started, all logic values are set to `x`. All events scheduled for time 0 are then retrieved. These events set the values of *a*, *b*, and *sel* to 0. When the simulator arrives at `#12` it schedules an *execution event* for time 12 to begin executing the following events (in this case `a = 1`). The simulator also schedules *update events* by propagating logic. Namely it schedules `n_sel=1` for time 0 (no delay through the NOT gate), `f1=0` and `f2=0` for time 2 (there is a delay of 2 time units through the AND gates and we can deduce the result because one of the inputs is already 0). Note that `n_sel` does not change in this simulation cycle - the update is simply scheduled for time 0 and put back in the event list. Since all current events have been handled, this marks the end of a *simulation cycle*, and we start the process again.

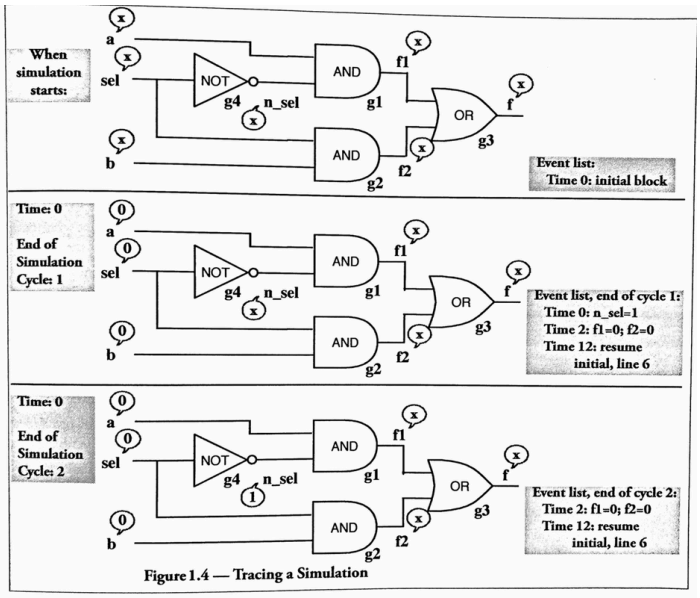


Figure 1.4 — Tracing a Simulation

The simulator now retrieves any events for the current time (still 0) and thus now updates `n_sel=1`. It also propagates this value forward to `f1` and `f2` but since this change has no effect it does not schedule anything. This marks the end of simulation cycle 2 of time 0. See figure 1.4 for an illustration.

Now the simulator sees that there are no more events at the current virtual time (time 0), so the `$monitor` statement executes. We see that `f=x` (and `n_sel` would be `1` while `f1` and `f2` would also be `x`). The simulator then finds the next smallest time and sets the current time to that (time 2 in this case). It retrieves the update events for `f1` and `f2` and propagates them by scheduling an update event at time 4 setting `f=0`.

Now the simulator starts another simulation cycle, sets the virtual time to 4, and updates `f` to 0. Since there are no more events scheduled for time 4, and a monitored variable has been updated, it executes the monitor statement.

The next event in the event list is scheduled for time 12, where an *execution event* is scheduled to continue the `initial` block. The simulator sets `a=1`, schedules an update event for `f1=1` for time 14, and schedules an execution event to resume the initial block at time 18. See figure 1.5. In addition, since `a` has changed, the monitor statement is executed.

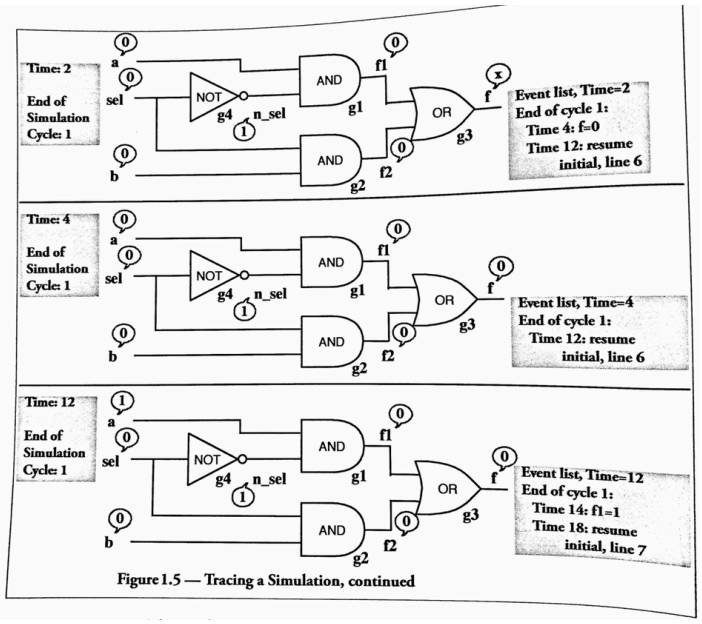
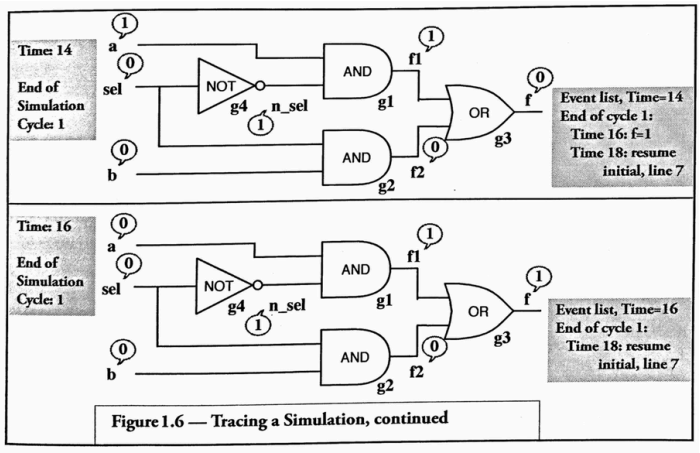


Figure 1.5 — Tracing a Simulation, continued

The simulation continues, updating `f1` to 1, updating `f` to 1, and finally halting execution when the

`$finish` statement is found at time 18. These final simulation cycles are illustrated below:



We have been describing times in terms of "time units." In Xilinx a *timescale* can be provided to the simulator to associate units to a value and precision (for rounding). You'll find that in the programming assignment files the timescale is set by the command

```
`timescale 1ns / 1ps
```

This means that 1 time unit is 1 nanosecond with 1 picosecond of precision.

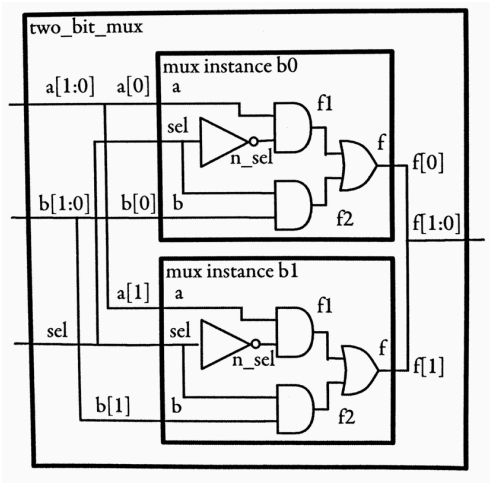
Hopefully this section has helped you gain a more in-depth understanding of how Verilog can enable us to simulate logical systems with possibly millions of components that are concurrent. When implementating in physical hardware for the first time, it is important that our code does not have bugs, and having a good simulator and a good understanding of the simulator enables us to build complex circuits that we can have confidence in.

Module hierarchy

In addition to the gate primitives we saw earlier (*AND*, *OR*, etc…) we can also use modules we have defined to construct more complex modules. In the following example we use our previous mux to create a 2-bit two-to-one mux. This time a and b are 2-bit values, while *sel* remains a single bit value. In Verilog we can define multi-bit logic variables using the `[n:0]` notation.

```
module two_bit_mux(f, a, b, sel);
    input wire [1:0] a, b;
    input wire sel;
    output wire [1:0] f;

    mux b0 (f[0], a[0], b[0], sel);
    mux b1 (f[1], a[1], b[1], sel);
endmodule
```



Here we define *f*, *a*, and *b* as 2-bit quantities using `[1:0]`. This means that we can now access the bits in these binary vectors using `a[1]` and `a[0]`.

From this example we can see that modules are useful because they allow us to reuse code. Additionally we see that each module has its own namespace.

Verilog Cheat Sheet

Verilog provides many more features than what we have seen here. Notably large amounts of shorthand for making the language easier to work with. However it is still good to understand what is really going on under the hood. The Verilog cheat sheet on the course website should provide a good summary of what shorthand you can use with Verilog. Also note that until programming assignment 3 we will only use structural Verilog in synthesis (though we may use behavioral Verilog for testbenches).