1 Introduction and Warning

The purpose of this document is to teach you the Verilog you will need to know for this class (which is a lot, but much less than all of Verilog). But first, what is Verilog? A clear understanding of what Verilog is (and what it is not) is very important.

First and foremost, Verilog is a hardware description language (HDL). Never never never forget that: when you are writing Verilog, you are describing a hardware design. It is sometimes easy to lose sight of that fact in this course, since you never actually build the hardware you are so busy describing, but maintaining a hardware mindset is key to success.

Though you will not build any actual hardware, you will verify the correctness of your hardware design by simulating its operation. There are a number of tools on the market for simulating hardware designs described in Verilog. These tools use discrete event simulation: time is divided up into discrete points, and at each point, the tool evaluates models of the hardware blocks in your design to find the values of their outputs. The fastest ones, like VCS, get their speed by compiling custom binaries based on your Verilog description. Because the simulation uses discrete rather than continuous time, and because at each discrete time point the tool must evaluate hardware blocks sequentially rather than in parallel, the behavior of the simulation in some corner cases does not always match what the hardware would do. In general, if your hardware design is correct, the simulation will be correct. However, if your hardware design has bugs, the simulation behavior may be erratic. Thus when you are debugging your design, it can be important to understand how Verilog (or more precisely VCS) works as a simulation language in order to diagnose the symptoms of your misbehaving system. The simulation aspects of Verilog can also be significant on the fringes of your design, as when initializing state or building test benches. However, you should not forget that these aspects are merely artifacts of simulating your hardware design.

Because Verilog syntax looks very much like C, and your Verilog compile/simulate/debug/edit loop feels much like the compile/execute/debug/edit loop you use in software development, far too many students are tempted to think of Verilog as a programming language. In the broadest sense, that is not untrue—you can write arbitrary programs in Verilog—but this mindset is the surest path to failure. While you should certainly take advantage of the high-level C-like abstractions of Verilog (typing ’+’ sure beats designing an adder out of gates!) you should never forget that you are designing parallel hardware (that happens to be simulated sequentially), and not writing a sequential program.
Verilog like C is case sensitive and has C style comments (// and /* */), although we prefer //). In addition many of the operators are the same (+,-,/,*,>,<<,|, etc). Some of the reserved words are the same (for, if) and there are new reserved words (always, module, input, output, etc).

With Verilog you are designing parallel hardware. We know we have said this many times, but it’s something new students have trouble with so here it is again. Under very few circumstances will the Verilog mimic the sequential nature of C. In this class you will use Verilog for testing, simulating, and synthesizing. Each of these tasks entails different functionality, and thus allows different parts of the language to be used.

Verilog design has a steep learning curve. Once you learn the rules and think about the Verilog in the appropriate way at the appropriate time Verilog will only be somewhat frustrating but will allow you to design complicated hardware rather quickly. Until then it will likely be very frustrating and will take you far longer to get anything done than you think possible.

2 Language Syntax

Below we describe some of the syntax of the verilog language. This is by no means a complete description, but it should be enough to get you heading in the correct direction.

2.1 Building blocks

2.1.1 Variables

There are two basic kinds of “variables”: wires and regs. Wires cannot have state and are thus always evaluated in terms of other values. They are highly restrictive and so are very safe to use. Regs can have state (but don’t necessarily have state) and almost anything can be done with them, so they are more flexible and more dangerous. All outputs default to wire, but can be redeclared as regs. All variables have a size (in bits), and default to one bit if no size is specified. Size is given as a range from one bit number to another. In a sense, many variables are like an array of bits. Either endianness is possible, but we suggest big-endian (as in these examples).

wire [7:0] an_8bit_wire, another_8bit_wire;
// an_8_bit_wire[0] is the rightmost bit, an_8_bit_wire[1:0] is
// the two rightmost bits
reg a_single_bit_register;

2.1.2 Literals

Verilog allows you to use literals represented in many different bases. A literal takes the following format: A size in bits, followed by the ’ character (the “right” tick which should be on the same key as ”), followed by a base indicator (most common are b=binary, d=decimal, h=hex), followed by the value in that base.

If you just write the value (without the base information), it will default to base 10 and use as many bits as necessary. You should try to develop the habit of including all the base information because it will make your code a lot more legible later. In general, the value is 0-extended to the left to fit the specified size.
Examples:

- $8'b00001010$ a byte with decimal value 10
- $32'd10$ a 32-bit word with decimal value 10
- $16'ha$ a 16-bit half-word with decimal value 10

0 and 1 are not the only values that exist in Verilog, x and z also have meaning. x means unknown and z means disconnected or high-impedence. In general, you will see these as outputs when something is not functioning properly, however, it is perfectly possible to introduce x values into your code on purpose in order to help the synthesis make faster circuits or z values in your code for a shared bus. When a combinational circuit has an x value as its output, the synthesis tools will pick whatever output makes the fastest circuit. So if some value should never appear when the system is working properly, you might set the outputs to x for that output. This is not recommended until you are comfortable with Verilog.

### 2.1.3 Macros

Like C, Verilog has the ability to define and use macros, however the syntax is a bit different but should be clear for the below example.

```verilog
'define MY_CONSTANT 3'b010
a = b + 'MY_CONSTANT; // really a = b + 3'b010;
```

### 2.2 Operators

The operators $+, -, *, >, <, >=, <=, !=, <>, (), |, ~, \&, \land, ?, :$ are all like their C counterparts and can be used in any expression. A short description of the ones less common in C:

- $\&$ Bitwise and
- $|$ Bitwise or
- $\sim$ Bitwise negation (can be combined with another operator so $\sim \&$ is bitwise nand)
- $\land$ Bitwise xor
- $<<$ Left shift
- $>>$ Right shift
- $?:$ a $?$ b $:$ c yields b if a is true and c if a is false (note that just like in C, 0 is false and 1 is true). In hardware terms, this is a two-input mux.

Additionally there are the $\{\}$ operators

- $\{a, b, c\}$ concatenation: puts a, b, and c consecutively as single value (i.e. $\{2'b01, 3'b101, 4'b0110\}$ becomes $9'b011010110$)
- $\{n\{m\}\}$ makes a single value that is n copies of m one after the other

### 2.3 Setting values

#### 2.3.1 Assign and Always

There are two main ways to describe the hardware you’re trying to design assign statements and always blocks.
Assign statements are simple descriptions of combinational logic. The syntax is `assign <wire> = <expression>` The output of an assign statement must be a wire (not a reg). The combinational logic inputs may be either wires or regs. Since assign statements are single lines of code, you cannot use if/then/else statements, but you may use the `a?b:c` syntax described above (and nested versions of it) to accomplish the same thing.

Always blocks are descriptions of events that should occur when a certain condition is met. You define an always block using `always @(condition)` and the block of code to be executed must be enclosed with `begin` and `end`. Outputs set within an always block must be declared as regs, regardless of whether they're combinational or sequential outputs. For 470, you will mostly use two types of always blocks: `always @*` is updated continuously and is effectively the same as combinational logic. `always @(posedge clock)` is updated on the positive edge of the clock signal and functions as a positive edge-triggered flip-flop.

For examples of assign statements and always blocks, refer to the example at the end of this document.

### 2.3.2 Initial blocks

Initial blocks are used to initialize wires with certain values. You will use these in testbenches in order to run a sequence of test values. Do not use these within functional modules. If you want to reset signals within modules to a certain starting value, use a top-level reset signal controlled by the testbench.

### 2.3.3 Blocking and non-blocking operators

Within an always block, combinational outputs are set using C-style assignments. These types of assignments are processed in order. These are called blocking assignments, because earlier lines of code block later lines of code from executing even if they are independent.

Blocking assignments are suitable for combinational logic, since all the operations occur in a certain sequence. However, for sequential logic we may want certain things to happen in parallel (like latching values into a register). For this, we need to use non-blocking assignments.

Non-blocking assignments use the non-blocking operator `=`. All assignments that use the non-blocking operator occur at (approximately) the same time. However, different always blocks are processed in a certain order, so two non-blocking assignments in different blocks may happen in series rather than in parallel. To avoid this, a delay can be added to the non-blocking assignment. The delay reads the current value of the input, but doesn’t modify the output until later. A few examples:

```vhdl
//blocking operation 1
always @*
begin
  x = new_val1;
  y = new_val2;
  last_sum = x+y; //last_sum = new_val1 + new_val2 here
end

//non-blocking operation 1
```
always @(posedge clock)
begin
    x <= new_val1;
y <= new_val2;
    last_sum <= x+y; //last_sum uses the old values of x and y
end

//non-blocking operation 2
always @(posedge clock)
begin
    x <= new_val1;
y <= new_val2;
end

always @(posedge clock)
begin
    last_sum <= x+y; //last_sum = ??? depends on order...
end

//non-blocking operation 3
always @(posedge clock)
begin
    x <= #1 new_val1;
y <= #1 new_val2;
end

always @(posedge clock)
begin
    last_sum <= #1 x+y; //last_sum uses old values of x,y
end

One last important thing to note: You cannot use both blocking operators and non-blocking operators to write to the same signal. Additionally you shouldn’t have both blocking and non-blocking operators within a single always block.

Example (don’t do this):
always @(posedge clock)
begin
    if(cond)
        x <= y;
    else
        x = z;  //???? Mixing blocking and non-blocking. BAD!
end
2.4 Modules

These are the basic organizing units in Verilog. They are like “black boxes.” They have inputs and outputs and internals that the outside world does not see. An example declaration (defines what one of these boxes looks like):

```verilog
module my_simple_mux(select_in,a_in,b_in,muxed_out);
  //all inputs and outputs listed
  input select_in, a_in, b_in;
  output muxed_out; //defaults to wire but can be redefined
  assign muxed_out=select_in?b_in:a_in;
endmodule
```

Modules can be instantiated in two ways (makes an instance of one box inside a bigger box):

```verilog
my_simple_mux m1(.a_in(a),.b_in(b),.select_in(s),.muxed_out(m));
//using this syntax, the order of the operands is not important
```

```verilog
my_simple_mux m1(s,a,b,m);
//using this syntax, the order of the operands must match the
//order in the module declaration
```

2.5 Debugging with $display and $monitor

$display is like printf in C and uses very similar format strings (but no need for a trailing newline, that is added automatically). It displays something every time the statement is executed. $monitor sets up a background watcher on some signals and prints something every time one of them changes.

These are both useful tools for debugging because the GUI-based debugger is slower and sometimes harder to interpret than a simple command-line simulation. These will be ignored by the synthesis tools because they don’t really have anything to do with actual hardware, just the simulation. Examples:

```verilog
$monitor("At time %4.0f: r=%d, u_d=%d", $time, r, u_d);

$display("Got to else part and the instruction is %h", inst)
```

3 Designing the hardware

While a simulator lets you check the functional behavior of your hardware design, a synthesis tool compiles your Verilog to a low-level hardware design suitable for fabrication. Verilog is flexible enough to allow you to describe hardware that cannot actually be built, and even when your design
is feasible, synthesis tools are not always smart enough to figure that out. Thus if you plan to use a synthesis tool (which you will), you must follow certain rules and conventions in your Verilog description. This restricted subset of Verilog is referred to as “synthesizable Verilog”. We require synthesizable Verilog in 470 for everything except test benches. Perhaps the most important constraint is that you need to describe sequential logic and combinational logic different ways. In particular sequential logic is very restricted. Here is an example. Suppose we are implementing the 2-bit predictor state machine reproduced here from page 198 in the textbook:

How do we implement this in hardware? Here is a diagram.

Note that there are other possible ways to implement the same functionality. In this example the combinational logic is known to be the transition function of the state machine. Also, the clock and reset connections are left out since we are clocking all flip-flops on the positive edge of the same clock and all resets are triggered by the same reset signal, so there is no need to explicitly show these signals. You will often want to detail portions of the combinational logic as well, especially muxes. In fact, here is a different design of the state machine that does that:
4 Planning and writing the code

With the design in hand it should not be that difficult to describe it in Verilog. In general, each piece of hardware in the design should turn into one block of code in the Verilog module. If a block of the design is a complicated piece with its own design it should probably be its own module instantiated in the larger module. On the following page is sample Verilog code for both designs: The first version is file `tbp1.v` and the second version is `tbp2.v`.

Notice how in `tbp1.v` the transition function became an always block and the register became an always block, and in `tbp2.v` the mux and equality check became an assignment to an internal wire and the two registers were combined into an always block.

For larger designs there may be too many details or copies of things to just write the code in a straightforward way. Later in the class we will discuss using arrays or trees to generate these.

```verilog
module two_bit_pred(clock, reset, taken, transition, prediction);

  input clock, reset, taken, transition;
  output prediction;

  reg [1:0] state;
  reg [1:0] next_state; // synthesis should make this a wire

  assign prediction = state[1];

  always @*
  begin
    case(state)
      2'b00 : next_state = taken ? 2'b01 : 2'b00;
      2'b01, 2'b10 : next_state = taken ? 2'b11 : 2'b00;
      2'b11 : next_state = taken ? 2'b11 : 2'b10;
    endcase
  end

  always @(posedge clock)
  begin
    if(reset)
state <= #1 2'b01;
else if(transition)
    state <= #1 next_state;
end
endmodule

module two_bit_pred(clock, reset, taken, transition, prediction);

input clock, reset, taken, transition;
output prediction;

reg last_taken, prediction;
wire next_prediction;

assign next_prediction = (taken==last_taken) ? taken : prediction;
always @(posedge clock)
begin
    if(reset)
        begin
            last_taken <= #1 1'b1;
            prediction <= #1 1'b0;
        end
    else if(transition)
        begin
            last_taken <= #1 taken;
            prediction <= #1 next_prediction;
        end
end
endmodule

5 Testing your design

Writing good tests is very important for finding bugs in your design as early as possible. You will use Verilog to write tests for the modules you create, however a “testbench” is written in a very different style of Verilog. No longer are you describing hardware (for the most part). Instead, you are describing how the inputs to the hardware change and either checking the results against something (known correct results or the results of a different version of the same module) or displaying the results so that you can determine if they are correct, perhaps by comparing the results to known correct results with the diff utility for example. A straightforward test-bench should:
1. declare regs for the module inputs
2. declare wires for the module outputs
3. instantiate the module
4. maybe set up a correctness checker (not shown in this test-bench)
5. maybe set up a clock with an always block
6. describe the tests in the initial block which does:
   (a) maybe set up the debug info so the debugger can be used
   (b) maybe set up a monitor to track all changes to certain variables (similar things can be done with always blocks and $display)
   (c) initialize the inputs (and clock, if necessary)
   (d) describe how the inputs change over time.

The actual tests can be:
1. Hand-written (this test-bench is just some hand-written tests)
2. Exhaustive (tests every possible input combination)
3. Random (very effective for certain problems but not really this one)

Common pitfalls/tricks when writing tests:
• Never have the inputs change right on the rising edge of the clock. Real hardware generally does wacky unpredictable things when this happens and Verilog does too. This is a race condition.
• It is easy to lose track of the clock in the middle of writing the tests. @(negedge clock); is a great command that waits till the next falling edge of the clock which is generally a very safe place to change things.
• When your testbench makes you think you have a bug in your code, check the test-bench for bugs as well. You wrote both and either could have problems, especially if you changed how your module interfaces to the outside world.

On the next page is a testbench for the sample modules: This is file tbp_tb.v.

module test_bench;

    reg clock, reset, taken, transition;

    wire prediction;

    two_bit_pred tbp(.clock(clock), .reset(reset),
always
begin
  #5;       // 5 tick delay
  clock = ~clock; // generate the clock.
  // This will make a 10 tick (ns) clock
end

initial // initialize values and run the test sequence
begin

  'ifdef DUMP // this is necessary for GUI debugging
    $vcdpluson;
  'endif

  $monitor("Time:%4.0f clock:%b reset:%b taken:%b transition:%b"
   "prediction:%b", $time, clock, reset, taken, transition, prediction);

clock = 1'b0; // must initialize clock!
reset = 1'b1;
taken = 1'b0;
transition = 1'b1;

  @(negedge clock);
  @(negedge clock);
  reset = 1'b0;
  @(negedge clock);
  taken = 1'b1;
  @(negedge clock);
  transition = 1'b0;
  @(negedge clock);
  @(negedge clock);
  transition = 1'b1;
  #3 transition = 1'b0;
  @(negedge clock);
  transition = 1'b1;
  @(negedge clock);
  taken = 1'b0;
  @(negedge clock);
  @(negedge clock);
$finish;
end
endmodule

This particular test-bench displays the inputs and outputs of your design over a short test-vector and requires manual inspection to verify correctness. Test-benches of this form are the easiest to produce but are only effective at verifying the smallest designs. Project 1 will require the use of self-checking test-benches that verify circuit correctness at runtime and end simulation if an error is discovered. The next page has an example of such a testbench using random test-vectors. Assume that `two_bit_pred_gold` is a `two_bit_pred` module that we are sure works correctly.

Later in the semester you will be designing large state machines that cannot easily make use of self-checking test-benches. In these instances you will make use of other means to verify that the test-bench output is correct. For example, when testing the processor you will be using in project 3 and your final project you will need to verify the contents of main memory at the end of program execution as well as the contents within the pipeline at each cycle. The GSIs will cover methods for verifying the behavior of these larger designs when they are assigned later in the semester.

### 6 Debugging

A large portion of your time this semester will be spent debugging your Verilog projects. It is strongly recommended that you familiarize yourself with the various debugging tools and approaches presented in this section early in the semester. There are certain instances in which each of these methods excels and fails at debugging. Knowing how and when to use each of these tools will save you a great deal of time later in the semester.

The first and simplest method of debugging a design is the 'printf' style of debugging. This is a rather ad-hoc method but it can be very effective at quickly checking signal values and determining the specific code blocks that are executing at each time step. There are two ways to accomplish this in Verilog: `$display` and `$monitor` statements. Both types of statements have syntax very similar to C’s `printf` function.

`$display` is the Verilog equivalent to C’s `printf` function. It uses the same syntax as `printf` but always inserts new-lines at the end of each string. Any number of `$display` statements may be placed throughout your design but must be placed within code blocks such as `always` and `initial`.

```verilog
$display("time %4.0f: Executing always block A with values %d %d", $realtime, a, b);
```

The `$monitor` statement is a variation on `$display` that automatically prints to the terminal whenever there is a change in the value of any of its function arguments. Only a single `$monitor` statement can be placed within the module you would like to debug. This statement should be placed toward the beginning of the initial block.

```verilog
$monitor("time %4.0f: Signal values %d %d %d", $realtime, a, b, c);
```

One of the most useful applications of `$display` statements is to determine which paths are being taken through an if-else block or case statement.
A more formal testing method is to use the graphical interface to VCS to view waveform data from your circuit simulation. This is discussed in the tutorial.

The synthesis tools present more challenges to debug but also provide some useful debugging tools. They will be discussed later.

```verilog
module test_bench;

    reg clock, reset, taken, transition, quit;

    wire prediction, prediction_gold, correct;

    two_bit_pred tbp(.clock(clock), .reset(reset), .taken(taken), .transition(transition), .prediction(prediction));

    two_bit_pred_gold tbp_gold(.clock(clock), .reset(reset), .taken(taken), .transition(transition), .prediction(prediction_gold));

    assign correct = (prediction==prediction_gold);

    always @ (negedge correct)
    begin
        #1;
        if(!correct) begin //check to make sure this isn’t a glitch
            $display("FAILED");
            $finish;
        end
    end

    always begin
        #5;
        clock =~clock;
    end

    initial begin

        ifdef DUMP
            $vcdpluson;
        endif

        clock = 1’b0;
        reset = 1’b1;
```
taken = 1'b0;
transition = 1'b1;
quit = 1'b0;

@(negedge clock);
@(negedge clock);
reset = 1'b0;
quit <= #10000 1'b1;
while(!quit) begin
  @(negedge clock);
  {taken, transition} = $random;
end
@(negedge clock);
@(negedge clock);
$display("PASSED");
$finish;
end
endmodule

7 Creating Synthesizable Verilog

Synthesizable Verilog is used for designing actual hardware. Hardware registers and combinational logic are two very different things. The programming styles and guidelines for sequential and combinational logic are different but easily confused. Separating sequential and combinational logic into different blocks allows one to check for errors much more easily.

7.1 Sequential logic

All assignments use the nonblocking <= operator with `SD (or some other delay). Note that multiple registers can be set within a single always block, but only if they are of the same type (i.e. all have reset or none do). There are a few more advanced forms (enables and memories write ports) that will be explained and allowed in later assignments. Finally, no clock or reset gating is allowed. Note that there may not be any path that sets a single register more than once. Any registers not set are assumed to keep their last value.

64-bit register a_reg with input a (output is a_reg), no reset

reg [63:0] a_reg;

always @(posedge clock)
begin
  a_reg <= 'SD a;
end
8-bit register `b_reg` with input `b` (output is `b_reg`), and synchronous reset

```verilog
class [7:0] b_reg;
always @(posedge clock)
begin
    if(reset)
        b_reg <= 'SD 0;
    else
        b_reg <= 'SD b;
end
```

### 7.2 Combinational logic via assign

- There is no always block.
- Outputs must be declared as “wire”.
- Safest (no way to accidentally create state).

1 bit wire `do_something` set to output of `(do_a or do_b)`

```verilog
wire do_something;
assign do_something = do_a | do_b;
```

### 7.3 Combinational logic via always block

- All assignments use blocking `=` operator with NO ‘SD
- All inputs listed in always statement (or use `@*`)
- Output “wires” declared as reg (will synthesize as wires, not registers)
- All outputs must be assigned to on all paths through code (use of default initialization and/or defaults for cases recommended) This one is very important because if there is a path in the always block that does not set a wire, Verilog will give it a state and create crazy problems.
- Acts sequentially like C where the final assignment to an output is the output of the logic.
  This means the same signal can be set multiple times, but only the last one will take effect.
- If the block is too large, sometimes the tools will create false dependencies. This is a real
  problem because circular combinational logic is strictly prohibited.

This code routes one of three inputs (based on `input_select`) to one of three outputs (based on `output_select`) and clears all non-selected outputs to 0. When an invalid input is provided to the device, an error output (`control_error`) is set to 1. The circuit looks something like the figure on the right but with the addition of some simple error detection.
`define DEVA_ID 2'b00
`define DEVB_ID 2'b01
`define DEVC_ID 2'b10

reg control_error;
reg [63:0] deva_output, devb_output, devc_output;
reg [63:0] selected_output; // intermediate variable

always @*
begin
    control_error=0;
    deva_output=0;
    devb_output=0;
    devc_output=0;
    selected_output=0;
    case(input_select)
        `DEVA_ID: selected_output=deva_input;
        `DEVB_ID: selected_output=devb_input;
        `DEVC_ID: selected_output=devc_input;
        default: control_error=1;
    endcase
    case(output_select)
        `DEVA_ID: deva_output=selected_output;
        `DEVB_ID: devb_output=selected_output;
        `DEVC_ID: devc_output=selected_output;
        default: control_error=1;
    endcase
end