Introduction to Verilog

Some material adapted from EE108B *Introduction to Verilog* presentation

In lab, we will be using a hardware description language (HDL) called Verilog. Writing in Verilog lets us focus on the high-level behavior of the hardware we are trying to describe rather than the low-level behavior of every single logic gate.

Design Flow

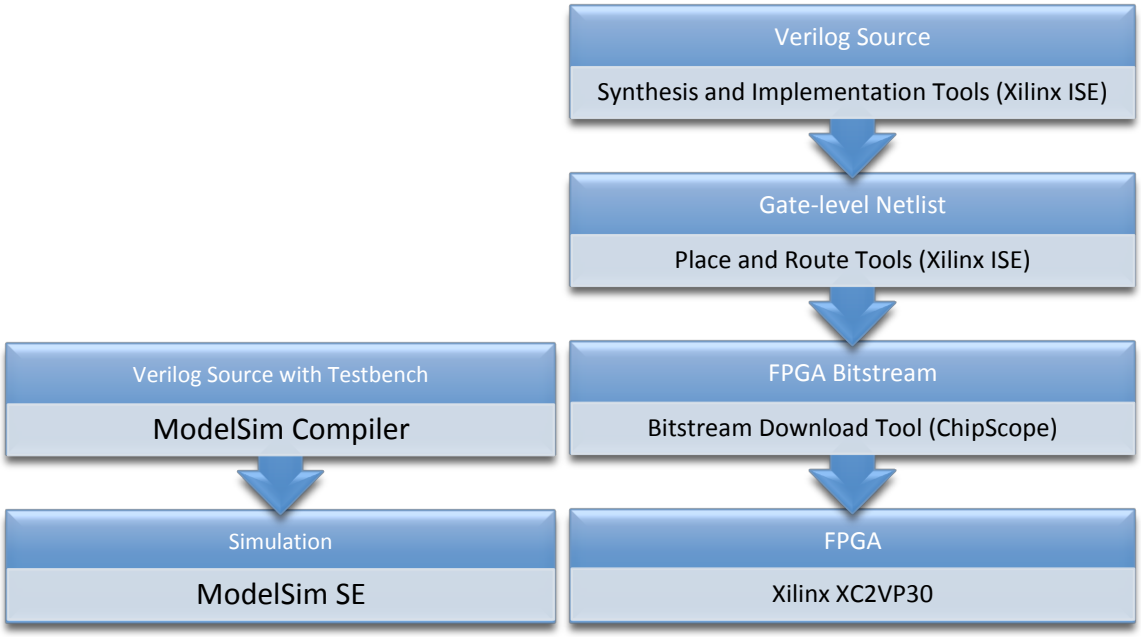

Figure 1. Simulation flow (left) and synthesis flow (right)

The design of a digital circuit using Verilog primarily follows two design flows. First, we feed our Verilog source files into a simulation tool, as shown by the diagram on the left. The simulation tool simulates in software the actual behavior of the hardware circuit for certain input conditions, which we describe in a testbench. Because compiling our Verilog for the simulation tool is relatively fast, we primarily use simulation tools when we are testing our design.

When we are confident that design is correct, we then use a hardware synthesis tool to turn our high-level Verilog code to a low-level gate netlist. A mapping tool then maps the netlist to the applicable resources on the device we are targeting—in our case, a field programmable grid array (FPGA). Finally, we download a bitstream describing the way the FPGA should be reconfigured onto the FPGA, resulting in an actual digital circuit.

Philosophy

Verilog has a C-like syntax. However, it is philosophically different than most programming languages since it is used to describe hardware rather than software. In particular:

- Verilog statements are concurrent in nature; except for code between *begin* and *end* blocks, there is no defined order in which they execute. In comparison, most languages like C consist of statements that are executed sequentially; the first line in main() is executed first, followed by the line after that, and so on.
- Synthesizable Verilog code is eventually mapped to actual hardware gates. Compiled C code, on the other hand, is mapped to some bits in storage that a CPU may or may not execute.

Synthesizable Combinational Verilog Syntax

Modules

The basic building block of Verilog is the module statement. It is somewhat analogous to defining a function in C:

```
module <module_name>(<input_list>, <output_list>);
input <input_list>;
output <output_list>;

endmodule
```

Here is a module that takes in three inputs: two 5-bit operands called a and b, and an enable input called en. The module's name is comparator.

```
module comparator(a, b, en, a_gt_b);
input [4:0] a, b;
input en;
output a_gt_b;

endmodule
```

In this state, the module just does nothing, for two reasons. First, there is no code in the body of the module—both the inputs and outputs are dangling. Secondly, defining a module in and of itself does nothing (unless it is the top level module). We need to create an instance of a module in our design to actually use it.

Instantiating Modules

We can include an instance of a module within another module using the following syntax:

```
<module_name> <instance_name>(<port_list>);
```

For example, to instantiate a comparator module with the name comparator1, input wires in1, in2, and en, and an output wire gt, we could write:

```
comparator comparator1(in1, in2, en, gt);
```

This instantiation depends on the ordering of the ports in the comparator module. There is an alternate syntax for instantiating modules which does not depend on port ordering, and is thus usually vastly preferred. The syntax is:

```
<module_name> <instance_name>(.<port_name>(ioname), ...);
```

Continuing from the last example, we could instead write:

```
comparator comparator1(.b(in2), .a(in1), .en(en),
.a_gt_b(gt));
```

Notice that although we switched the order of ports b and a in this example, the instantiation will still work because we have named which ports we are connecting to.

Comments

Comments in Verilog are exactly the same as in C.

```
// This is a comment
/* Multi-line
   comment */
```

Numerical Literals

Many modules will contain numerical literals. In Verilog, numerical literals are unsigned 32-bit numbers by default, but in this class you should probably get into the habit of declaring the width of each numerical literal. This leads to less guesswork when, for example, you concatenate a wire and a numerical literal together (as shown later).

Here are a few example numerical literals:

```
/* General syntax:
   <bits>'<base><number>
   where <base> is generally b, d, or h */

wire [2:0]  a = 3'b111;       // 3 bit binary
wire [4:0]  b = 5'd31;        // 5 bit decimal
wire [31:0] c = 32'hdeadbeef; // 32 bit hexadecimal
```

We have not yet defined what a wire is, but we will soon.

Constants

We can use `define to define global constants in our code (like the #define preprocessor directive in C). Note that unlike C, when referencing the constant, we need to append a backtick to the front of the constant: e.g., in our case we had to use `FRI instead of FRI. Also, do not append a semicolon to the `define statement.

```
`define RED   2'b00 // DON'T add a semicolon to these
`define WHITE 2'b01 // statements, just as with C's #define
`define BLUE  2'b10

wire [1:0] color1 = `RED;
wire [1:0] color2 = `WHITE;
wire [1:0] color3 = `BLUE;
```

Wires

To start with, we will declare two kinds of data types in our modules: wires and registers. You can think of wires as modeling physical wires—you can connect them either to another wire, an

input or output port on another module, or to a constant logical value. To declare a wire, we use the wire statement:

```
wire       a_wire;
wire [1:0] two_bit_wire;
wire [4:0] five_bit_wire;
```

We then use the assign statement to connect them to something else. Assuming that we are in a module that takes a two bit input named two_bit_input, we could do the following:

```
assign two_bit_wire = two_bit_input;
// Connect a_wire to the lowest bit of two_bit_wire
assign a_wire = two_bit_wire[0];

/* {} is concatenation – 3 MSB will be 101, 2 LSB will be
   connected to two_bit_wire */
assign five_bit_wire = {3'b101, two_bit_wire};

// This is an error! You cannot assign a wire twice!
// assign a_wire = 1'b1;
```

Note that these are *continuous* assignments. That means that in the previous example, whenever the input two_bit_input changes, so do the values of two_bit_wire, a_wire, and five_bit_wire. There is no "order" by which they change—the changes occur at the same time. This is also why you cannot assign the same wire twice in the same module—a wire cannot be driven by two different signals at the same time. This is what we mean when saying Verilog is "naturally concurrent."

Finally, there is a shortcut that is sometimes used to declare and assign a wire at the same time:

```
// Declares gnd, and assigns it to 0
wire gnd = 1'b0;
```

Registers

The other data type we will use is register. Despite the name, registers do *not* imply memory. They are simply a language construct denoting variables that are on the left hand side of an always block (and in simulation code, initial and forever blocks). You *declare* registers, like wires, at the top level of a module, but you *use* them within always blocks. You cannot assign registers values at the top level of a module, and you cannot assign wires while inside an always block.

Always Blocks

Always blocks are blocks which model behavior that occurs repeatedly based on a sensitivity list. Whenever a signal in the sensitivity list changes values, the statements in the always block will be run *sequentially* in the simulator. In terms of actual hardware, the synthesis tool will synthesize circuits that are logically equivalent to the statements within the always block.

In the degenerate case, a register in an always statement acts like a wire data type, as in this simple module:

```
module bitwise_not(a_in, a_out);
```

```
input  [1:0] a_in;
output [1:0] a_out;

/* Declare the 2-bit output a_out as a register, since it
   is used on the LHS of an always block */
reg [1:0] a_out;

// better to use always @* - see next example
always @(a_in) begin
  a_out = ~a_in; // out = bitwise not of in
end

endmodule
```

So whenever the input a_in changes, the code within the always block is evaluated—a_out takes the value of a_in. It is as if we declared a_out to be a wire, and assigned it to be ~a_in.

If and Case Statements

More interestingly, we can place case and if statements into always blocks. We can usually think of these case and if statements as being synthesized into some sort of multiplexer. In this class, Prof. Dally encourages you to use only case statements, but in other classes you may see if-else being used more.

Here is a simple circuit which utilizes a case statement within an if statement and utilizes some of the other concepts above:

```
module alarm_clock(day_i, hour_o, minute_o);
input  [2:0] day_i;
output [4:0] hour_o;
output [5:0] minute_o;

wire [2:0] day_i;

/* Declare hour_o and minute_o to be regs since
 * they are on LHS of an always block */
reg [4:0] hour_o;
reg [5:0] minute_o;

// have is_weekday take the value of a comparator
wire is_weekday;
assign is_weekday = (day_i <= `FRI);

always @* begin
  if (is_weekday) begin
    hour_o = 5'd8;
    minute_o = 6'd30;
  end
  else begin
    case (day_i)
      `SAT: {hour_o, minute_o} = {5'd11, 6'd15};
      `SUN: {hour_o, minute_o} = {5'd12, 6'd45};
     default: {hour_o, minute_o} = 11'd0;
    endcase
  end
end

endmodule
```

In particular, note:

- Case and if statements must be placed within an always block.
- The use of always@*. This is a new Verilog-2001 construct that automatically populates the sensitivity list with all variables listed in the right hand side of the always block. Unless otherwise noted, **you should always use always @*** for your always block sensitivity lists in this class in code you write, even though the lecture notes have not been updated to reflect this. It will save you hours of debugging.
- The use of begin…end to delineate multi-line blocks (instead of the usual {} found in other C-like languages). You may omit the begin…end if the assignments in your case or if statements are only a single line long, as in the case statement above.
- The fact that every case statement has a matching default statement and every if statement has a matching else. **Do not forget this, or you will generate latches!** We will go over why this happens in section or lab.

Parameters

Often, we want to create some generic module that can be customized by a few parameters when the module is instantiated. This is where the parameter statement comes in. The following is an example of a parameterized ALU, which defaults to 32-bits if no parameter is given during module instantiation:

```
`define ADD  3'd0
`define LESS 3'd1
`define EQ   3'd2
`define OR   3'd3
`define AND  3'd4
`define NOT  3'd5

module ALU(opcode, op_a, op_b, result);
parameter N = 32;
input  [2:0]   opcode;
input  [N-1:0] op_a, op_b;
output [N-1:0] result;

// result used in LHS of always block -> must be reg
reg [N-1:0] result;

always @* begin
  case (opcode)
     `ADD:  result = op_a + op_b;
     `LESS: result = op_a < op_b;
     `EQ:   result = op_a == op_b;
     `OR:   result = op_a | op_b;
     `AND:  result = op_a & op_b;
     `NOT:  result = ~op_a;
    default: result = 0;
  endcase
end

endmodule
```

Then, to instantiate this ALU within another module, use the #() symbols within the instantiation line. For example, this line instantiates a 16-bit ALU:

```
ALU #(16) alu1(...)
```

Testbenches

Up until now, we have written purely synthesizable Verilog—Verilog that will be synthesized, translated, and mapped to actual hardware (in our case, the FPGA). But before we do that costly step, we need to be extremely confident that our modules function correctly. This is where testbenches and simulation software come into play.

A Verilog testbench is a special file that instantiates the module (or modules) that we need to test. This testbench is not synthesized into hardware. Rather, it provides input stimuli into the instantiated modules so that we can run the testbench in a software simulator of our projected hardware design. Because they do not have to be synthesized, testbenches can be written in a different style than normal synthesizable Verilog.

Delay Statements

Simulation proceeds in hardware across discrete time units. To make actions in simulation go in a defined order, we often need to present the input stimuli at different time periods, rather than all at the same time. We do this by using delay statements:

```
// General syntax: #<n> -- delay for n time units

#5;   // delay this block for 5 time units
#100; // delay for 100 time units

// Can be compounded next to another statement to delay
// that statement

#3 $display("hi"); // wait 3 time units, then display “hi”
```

Initial Blocks

Initial and forever blocks are like always blocks in that the statements within an initial block execute in order when triggered. Also, only registers are allowed on the left hand side of an initial block. However, while an always blocks executes every time a condition changes, initial blocks are executed once—at the beginning of the program.

The following code sets opcode, op_a, and op_b to 0, 10, and 20 respectively at t=0 in the simulation, and then changes those values to 2, 10, and 20 respectively at t=5 in the simulation:

```
reg [2:0] opcode;
reg [4:0] op_a, op_b;

initial begin
  opcode = 3’b000;
  op_a = 5’d10;
  op_b = 5’d20;

  #5 opcode = 3’b010;
  op_a = 5’d10;
  op_b = 5’d20;
end
```

Display Statement

The $display statement can be used to display the value of a variable using printf-like syntax. It automatically inserts a newline at the end of the printing.

```
wire [3:0] ten = 4'd10;
$display("10 in hex: %h, dec: %d, bin: %b", ten, ten, ten);
```

Sample Testbench

Using just these statements, we can make a very crude testbench of our ALU module. More advanced testbench techniques are discussed in the lecture notes.

```
module ALU_test;

/* Declare as regs since we'll be changing these
   values in always blocks */
reg [2:0] opcode;
reg [15:0] op_a, op_b;
wire [15:0] result;    // just connected to module

// Instantiate the ALU module
ALU #(16) alu(.opcode(opcode), .op_a(op_a),
              .op_b(op_b), .result(result));

initial begin
  opcode = `ADD;
  {op_a, op_b} = {16'd32, 16'd5};
  // Wait 1 time unit for result to settle
  #1 $display("%b + %b = %b", op_a, op_b, result);
  #5;

  opcode = `OR;
  {op_a, op_b} = {16'd8, 16'd7};
  #1 $display("%b | %b = %b", op_a, op_b, result);

  // etc.
end

endmodule

/* Output of simulator:
# 0000000000100000 + 0000000000000101 = 0000000000100101
# 0000000000001000 | 0000000000000111 = 0000000000001111 */
```

This testbench is far from complete. We tested only one set of inputs for only two of the functions of the unit. We should test more cases, especially the corner cases. Also, an automated test is better than a manual test such as this. But it gives an idea of how to start programming testbenches. More examples of describing combinational logic in Verilog and creating testbenches can be found in chapters 7-11 of your course reader.