Chapter 1: The Verilog HDL

The Verilog language provides a module-driven approach to the creation of digital circuits and syntax similar to the C programming language. A module-driven approach allows the designer to create modules with clearly specified inputs, outputs, and internal signals that perform a specified digital function. These modules can be instantiated within other modules and used to build very large, complex designs. The code written for a digital system using the Verilog language is frequently referred to as RTL, as Verilog provides a Register Transfer Level description of a digital circuit.

Verilog provides many different tools for the creation and verification of logical circuits. This chapter will provide a brief overview of Verilog, including discussions about nets (signals) in Verilog, the basic structure of a module, constants in Verilog, parameterization, and the various constructs for creating logical circuits provided by Verilog.

Nets in Verilog

Verilog provides several different classes of nets (nets can be thought of as signals), but we will only use two classes of signals known as *reg*s and *wire*s. These nets can implement the same functionality but the desired functionality is assigned in different ways. *Wire*s are used for “continuous” assignments; continuous assignment implies that the value of the expression is either *assign*ed by an expression, or the *wire* is continuously connected to the output of another module. *Regs* are used with the other assignment methods (*always* and *initial* blocks) which will be discussed later in this chapter.

In Verilog, nets can either be single or multiple bits, and can be organized in arrays. An instance of a single bit *wire* signal can be instantiated by using the following syntax:

```
      wire <name>;
```

where <name> is the desired name of the signal. The instantiation of a multiple bit *reg* signal has similar syntax, as demonstrated below for a signal of width *w* and with name *name*:

```
      reg [<w-1>:0] <name>;
```

The addition of the text within the braces (“[<w-1>:0]”) in the multiple bit *logic* signal instantiation lays the only difference between the two instantiations. This text determines the positions of the bits in the signal, with the number before the colon corresponding to the position of the most significant bit (MSB) and the number after the colon corresponding to the position of the least significant bit (LSB). While these numbers can be set to any values, by convention we place the LSB at position zero; as a consequence of this, the MSB should be at the position equal to the width of the signal minus one. A net instantiated without the text within these colons (as seen in the single bit *wire* example) will automatically have a width of one bit.

Aside: Nets in SystemVerilog/Verilog-2005

In Verilog-2005 and SystemVerilog, a new type of net called *logic* is added to the mix. Nets with type *logic* can perform any functionality that can be implemented using either *reg*s or *wire*s and can use any of the assignment methods (*assign*, *always*, *initial*, or port connection) available in Verilog. Currently, SystemVerilog is supported by ModelSim, but is not supported by Xilinx ISE.

Net Arrays

In Verilog, arrays of nets can be created, allowing for the simple representation of array-like structures such as memories or multiplexers, and also allowing related signals with similar names and related functionality to be combined into an array in order to make the RTL more readable.

The syntax for the instantiation of a *reg* array is derived from the syntax for the instantiation of a multiple bit *logic* signal, with one small addition. A *reg* array with width *w*, size *s*, and name *name* is instantiated using the following syntax:

```
    reg [<w-1>:0] <name> [0:<s-1>];
```

Similar to the MSB and LSB position, the start and end of the array need not be at 0 and s-1, but by convention they are. We require that you follow these conventions.

Inputs and Outputs

In Verilog, the inputs and outputs from a *module* must be instantiated using the *input* and *output* keywords. Signals instantiated using the *input* keyword automatically are of type *wire* and cannot have a value assigned to them within the module, but can be used within logical expressions in the module. Signals instantiated using the *output* keyword automatically are of type *wire*, but this can be overridden and instantiated with other types. Example syntax for declaration of an *input* signal and an *output* signal with type *reg* is shown below, with width *w* and names *input_name* and *output_name*:

```
    input [<w-1>:0] input_name;
    output reg [<w-1>:0] output_name;
```

The inclusion of the *reg* keyword in the *output* net instantiation overrides the type of the signal; *input*s are by definition of type *wire* and cannot be overridden. There is an additional class of net known as *inout* which can be used for tri-state signals, but this type of net will not be used in EE108A.

Pitfall: Misspellings of Net Names

When the Verilog standard was created, it was decided that if a net was mentioned without being formally instantiated, it would automatically be instantiated as a *wire* with width one. While this provided a shortcut for experienced designers, it can cause trouble, as a misspelled net name will not be caught as an error during compilation (unlike other languages such as C or Java, which would catch the use of an undefined variable), but will instead be instantiated as a *wire* of width 1. As a result of this, if a circuit is not working in simulation and the logic appears to be correct, it is prudent to make sure that all net names are correctly spelled.

Implementing Logic Using Verilog: Operators

Verilog provides a rich array of operators for the implementation of digital logic; these operators can be used effectively with both the *assign* statement and *always* blocks to implement a wide array of circuits.

Concatenation, Replication, and Bit Selection

In Verilog, it is frequently useful to concatenate multiple signals together, replicate a signal, or to select only a few bits from a larger signal. Verilog provides operators for each of these specific functions.

To concatenate multiple signals together, Verilog uses braces ({}), with the names of the specific nets separated by commas (,). The following RTL would concatenate three signals named *a*, *b*, and *c*:

```
      {a, b, c}
```

The replication operator is similar to the concatenation operator. The syntax for this operator is an open bracket, followed by the number of times to be replicated, followed by another open bracket, followed by the signal to be replicated, and then two closed brackets. To replicate the signal *a* four times, we would use the following expression:

```
      {4{a}}
```

Selecting specific bits or a specific bit from a signal in Verilog is similar to addressing an array in C/C++/Java. To select specific bit(s) *n* from a net named *name*, we would use the following syntax:

```
      <name> [n]
```

N can either be a single bit, or a range of bits. The syntax for *n* is identical for the syntax used for instantiating a net.

Implementing Logic Using Verilog: Assign Statements

Assign statements are simple statements that allow a designer to use the operators available within the Verilog language to assign a logical equation to a specific signal.

Assign statements consist of the keyword *assign*, a net, and an expression. The *assign* statement below demonstrates two signals *a* and *b* being added together, and assigned to a net named *sum*:

```
      assign sum = a + b;
```

Implementing Logic Using Verilog: Always Blocks

Always blocks are among the most powerful constructs within the Verilog language, allowing any logical function to be implemented. *Always* blocks consist of two parts, a sensitivity list and a block body; every time that a signal in the sensitivity list changes the body of the block is evaluated. *Always* blocks can function to implement combinational logic, flip flops, and latches, depending on how the sensitivity list is structured.

Always blocks take the following form:

```
always @ (/* sensitivity list */)
     begin
          /* always block body */
     end
```

The Sensitivity List

The sensitivity list specifies when the body of an *always* block is evaluated. Depending on the way that the sensitivity list is specified, combinational logic, or logic with a flip-flop will be inferred. **In EE108A, we will only use *always* blocks for combinational logic; there will be a flip-flop library that is provided.**

If we desire to specify exactly what signals should cause the revaluation of the *always* block, the sensitivity list should specify the specific signals that cause revaluation separated by *or*. For a sensitivity list containing the example signals *trigger*, *on*, and *me*, the block would look as follows:

```
always @ (trigger or on or me)
     begin
          /* always block body */
     end
```

In combinational logic, every time any input to a circuit changes, the value of the circuit may change and should be revaluated. Verilog provides the wildcard operator (*), which can be used in a sensitivity list to include all signals within the *always* block in the sensitivity list. This leads to the Verilog simulating correctly, as any change in any net referenced within the *always* block will cause the block to be revaluated. <u>It is important to note that no matter what you put in your sensitivity list, the synthesis tool will always treat it as if it was *always @ (*)*. You are required to use this syntax.</u> This block would look as follows:

```
always @ (*)
     begin
          /* always block body */
     end
```

For a flipflop to be inferred, the *posedge* or *negedge* keywords must be included in front of a signal in the sensitivity list. This causes the block to be executed on the positive or negative transitions of a signal, similar to the behavior of a flipflop. A standard flipflop would look as follows:

```
always @ (posedge clk)
     begin
          /* always block body */
     end
```

Case Statements

The *case* statement can be considered as being identical to the switch statement that exists in most conventional programming languages. *Case* statements can be very useful for implementing finite state machines (FSM), multiplexers, and other related circuits. The syntax for a *case* statement with two cases (<*case 1*> and <*case 2*> are constants) and a default case is as follows:

```
case (/* net to switch on */)
     <case 1> : begin
          /* body of case 1 */
     end
     <case 2> : begin
          /* body of case 2 */
     end
     default : begin
          /* body of default case */
     end
endcase
```

It is important to have a *default* case that catches inputs that do not match with any of the selected cases. This helps to prevent inferred latches (see below for more information on the problem of inferred latches) and can also make it easier to catch improper functionality when verifying the circuit.

If-Else If-Else Statements

Verilog provides an *if-else* construct, similar to most conventional programming languages. Similar to the *case* statement, *if-else if-else* statements are generally most useful for implementing FSMs and complex multiplexors. The syntax for an *if-else if-else* statement with two expressions (<*expression 1*> and <*expression 2*>) is as follows:

```
if (<expression 1>) begin
     /* body for first case */
end else if (<expression 2>) begin
     /* body for second case */
end else begin
     /* body for all other cases */
end
```

While *if-else if-else* statements do not have a "*default*" case (which *case* statements have), any *if-else if-else* statement should end in an unqualified *else*. If the statement does not end in an unqualified *else*, and all possible permutations of inputs (inputs being any combination of values for any of the signals used in any expression) are not covered, a latch will be inferred.

Pitfall: Inferred Latches

In Verilog, if an *always* block is created for sequential logic, but there is a possible condition that might cause the block to be evaluated but values not assigned to one or more nets, this will cause the synthesis tool to infer that the last value on any of those nets should be latched. While this is occasionally desired behavior, this generally means that there is an error in the RTL that is causing unintended behavior. Additionally, this can cause complications during synthesis, especially on FPGAs.

A simple way to prevent latches from being inferred is to assign all nets in the *always* block to some value at the immediate start of the block body. This will ensure that a value is always assigned to all signals in the block once the block is evaluated.

Initial Blocks

Although most of Verilog is not procedural, *initial* blocks are the main procedural construct in Verilog and are not synthesizable.[1] Within an *initial* block, the values of *regs* can be manipulated at specific times, through the usage of time delays and events (e.g. rising/falling edge of a signal occurs). *Initial* blocks are one of the main tools used for performing verification, as they allow the verifier to apply specific test stimuli.

Modules in Verilog

Modules are the basic logical block in Verilog used to build up all large circuits. Within a *module*, nets can be instantiated, have values assigned to them, and additional sub*modules* can be instantiated. *Modules* consist of several syntactical elements, including the module declaration, the input/output list, and the module body.

Module Declaration and Input/Output List

In Verilog, every *module* must have a declaration that indicates the start of a *module* and attaches a name to the *module*. Immediately after this declaration, the inputs into the *module* and the outputs exiting the *module* must be instantiated in a list. The syntax for the module declaration and input/output list for a *module* with name *name* is as follows:

```
module <name> (
                /* input/output list */
                );
```

[1] There are rare cases when *initial* blocks are part of a synthesizable construct, such as initializing a RAM. Outside of these rare cases, *initial* blocks cannot be synthesized.

For a *module* used to create a two-bit adder named *adder* and with two bit inputs *a* and *b*, two bit output *sum*, and one bit output *carry*, the module declaration and input/output list would look as follows:

```
module adder (
              /* inputs to be summed */
              input [1:0] a,
              input [1:0] b,

              /* sum and carry outputs */
              output logic carry,
              output logic [1:0] sum
              );
```

Notice that the *input*s and *output*s are separated by commas (,). Normally, lines in Verilog are separated by semicolons (;), but input/output lists are one of several special exceptions. Please clearly comment on what specific *input*s and *output*s are to be used for in your input/output declarations; it is very bad coding practice to not clarify what your nets do, and although it may be crystal clear to you, it might not be to your partners, or to the TAs grading your labs.

Module Body

Following the module declaration and the input/output list is the module body, where nets can be instantiated, digital logic can be described using *always and/*or *initial* blocks and/or *assign* statements, and other *module*s can be instantiated. At the end of the module body, the *endmodule* keyword must occur, to let Verilog know that the end of the module has been reached.

We recommend that modules be structured as follows, for best readability:

```
/**
 * Module: <name>
 * Author: <author name>, <email>@stanford.edu
 * Date: <date>
 *
 * <description of functionality>
 */
module <name> (
                /* input/output list */
                );

     /* net instantiations */

     /* sub-module instantiations */

     /* assign statements (if any) */

     /* always/initial blocks (if any) */
```

```
endmodule /* module <name> */
```

Large digital designs are generally made by groups or teams. In these large projects, the engineers who implement the design may not work with the engineers who verify the design's functionality or the engineers who ready the design for manufacture. Since there is very little opportunity for direct communication, it is **imperative** to clearly document your designs. At the top of any module, there should be a detailed comment that describes the functionality of the module, lists the authors and their contact information, and lists the date of the most recent edit. Within the module, every net declaration should be commented, with a description of the signal's functionality. All module instantiations, *assigns*, and *always*/*initial* blocks should be clearly commented with a description of the functionality they implement.

Parametrization

One of the single most useful features implemented by Verilog is the support for changing the implementation of a module by changing the values of parameters. Verilog supports two different types of parameters: *parameter*s, which can be overridden by the instantiating module, and *localparam*s, which cannot be overridden by the instantiating module. These parameters can be very useful for designing modules that are frequently used, but used in different configurations (combinational/arithmetic logic with varying widths) and for storing the encoding of different states.

Here is an example for a circuit that multiplexes a signal between its current value, and a constant value. This module is designed to deal with signals with parametrizable widths, constant values, and can be changed between either active high or low.

```
module constant_mux #(
                      /* default width is 1 */
                      parameter WIDTH = 1,
                      /* default constant is 0 */
                      parameter CONST = 0
                      ) (
                         input  [WIDTH-1:0] in,
                         input  select,
                         output [WIDTH-1:0] out
                         );

     /* active high select */
     localparam SELECT = 1;

     assign out = (select == SELECT) ? CONST : in;

endmodule // constant_mux
```

Instantiating Modules

In Verilog, there are two ways to instantiate modules. One is known as named instantiation, as signals are connected by name, and the other is known as positional instantiation, as signals are connected by position. We require students to use named instantiation, but will describe positional instantiation so that you are familiar with both styles.

In named instantiation, the designer explicitly specifies how nets in the current *module* are connected to the sub-module that is being instantiated. **This is the preferred style.** In positional instantiation, the designer specifies the connection between nets in the current *module* and the sub-module that is being instantiated by ordering the connections. Roughly speaking, positional instantiation takes the following form:

```
<module name> #(<value of first parameter>,
                <value of second parameter>,
                …
                <value of last parameter>
                ) <instance name> (
                                   <first connection>,
                                   <second connection>,
                                   …
                                   <last connection>
                                   );
```

Named instantiation takes the following form:

```
<module name> #(.<name of parameter>(<value of parameter>),
                …
                .<name of parameter>(<value of parameter>)
                ) <instance name> (
                .<name of port>(<connection to port>),
                …
                .<name of port>(<connection to port>)
                );
```

To demonstrate the differences between these two styles, let's create an example module that logically ANDs together three signals, and a constant mask:

```
module and3mask #(parameter WIDTH,
                  parameter MASK) (
                                   input [WIDTH-1:0] a,
                                   input [WIDTH-1:0] b,
                                   input [WIDTH-1:0] c,
                                   output [WIDTH-1:0] out
                                   );

     /* and signals and mask together */
     assign out = a & b & c & MASK;

endmodule // and3mask
```

In our current module, let's say we have wires named *one*, *two*, *three*, and *anded*. We desire to connect *one* to input *a*, *two* to input *b*, *three* to input *c*, and *anded* to output *out*. In both cases, we want to name this instance of the *and3mask* module *ander*, we want to set parameter *WIDTH* to 3, and parameter *MASK* to 3'b110. Using positional instantiation, we'd write:

```
and3mask #(3, 3'b110) ander (one, two, three, anded);
```

Using named instantiation, we'd write:

```
and3mask #(.WIDTH(3), .MASK(3'b110)) ander (
                                              .a(one),
                                              .b(two),
                                              .c(three),
                                              .out(anded)
                                              );
```

With named instantiation, we can permute the order of the parameters and inputs/outputs as we please. Although named instantiation requires us to write more lines of RTL, it is clearer to read, and easier to update if the inputs/outputs/parameters of a module change.

Verilog Precompiler Directives

Similar to C/C++, Verilog has support for precompiler directives that can be used to modify written Verilog at compile time. Before support for parameters was added in Verilog-2001, engineers seeking to create parametrizable designs were forced into using precompiler defines to propagate parametrizable constants across a design. Although the use of parameters is preferable, the precompiler directives supported by Verilog can still be very useful.

Defines

To define a precompiler constant in Verilog, we use a define statement. As mentioned above, precompiler constants were used before Verilog-2001 to allow for parametrized designs. Although the need for this functionality has been largely eliminated by *parameters* and *localparams*, precompiler constants can be useful for large designs where certain constant values are reused across the design and it is desirable to have the constant values defined in one central location.

When compiling Verilog, we first pass it through a precompiler, which replaces all precompiler constants with their defined constant value. The syntax for defining the value of a precompiler constant (*END_COUNT* in this example) is as follows:

```
`define END_COUNT 16'b1000
```

To use this *END_COUNT* constant, we would use the following syntax:

```
assign reset_counter = count == `END_COUNT;
```

When the precompiler runs across our Verilog, it will replace every instance of *`END_COUNT* with it's defined value (16'b1000 in the above example). For students who are used to programming in C/C++, it is important to notice two subtle differences: *defines* come after a ` (a tick, not an apostrophe) instead of a hash (#), and text that is to be replaced must come after a tick as well (in C/C++, text that is to be replaced does not follow any specific character).

Although we recommend using *parameter*s and *localparams* in place of precompiler defines, it is important to be familiar with precompiler defines, as they are frequently used by tools and can serve as great time savers.

Pitfall: Reusing Constants Across Files

One of the main reasons that it is recommended to use *parameter*s and *localparam*s over precompiler defines in Verilog stems from how certain tools treat precompiler defines. In some tools (Modelsim, specifically), if multiple constants with the same name are defined across multiple files, the compiler will apply the value of the constant from the file that was read last to all of the files. For example, let's say we define a precompiler constant named *STATE_WIDTH* in three different files containing Verilog modules. In file 1, *STATE_WIDTH* is defined as having value 5, in file 2, *STATE_WIDTH* is defined as having value 7, and in file 3, *STATE_WIDTH* is defined as having value 3. In tools where the value from the file compiled last is used, we will see errors if file 3 is compiled last.

Includes

For designs where the designer desires desire to define a slew of constants for use across the whole design, it can be prudent to place all constants in one file, and to then grab the constants from this file for use across the design. To do this, we use precompiler includes.

In the example below, assume that there is a file called *constants.v* that contains multiple precompiler defines that we desire to use in one of our modules, to ensure consistency across our design. We can do this with the following syntax:

```
`include constants.v
```

This precompiler directive takes the contents of the file *constants.v*, and prints them at the location of the *`include*. For this reason, *`include*s are generally put at the head of a file.

Ifdef/Ifndef/Else

The *ifdef*/*ifndef*/*else* precompiler directives are used to selectively hide/show lines in a file depending on whether a precompiler constant has been defined. In hardware designs, these directives are most frequently used to replace hard IP blocks[2] that are difficult to simulate with simpler models for simulation, but to

[2] Hardware designers frequently use "IP cores" in their designs. An IP core is a piece of hardware that has already been implemented and tested, and can just be dropped into a design, allowing the

explicitly instantiate these same IP blocks in the actual, synthesized design. Synthesis tools will frequently define a constant named *SYNTHESIS* for this purpose. In the example below, we replace hard IP for a clock divider with a simpler model of a clock divider for simulation:

```
`ifdef SYNTHESIS

wire CLK_divby2;
/* instantiate clock divider ip */
clkdiv clkdiv (.CLKIN(CLK), .CLKOUT(CLK_divby2));

`else

/* simulate dividing clock by 2 */
reg CLK_divby2 = 0;
always@(posedge CLK)
      CLK_divby2 <= ~CLK_divby2;

`endif
```

The *`ifdef* directive allows the text following it to be seen by the compiler if and only if the constant it references has been defined in the current file, or has been defined in a file included into the current file. Alternatively, the *`ifndef* directive allows the text following it to be seen by the compiler if and only if the constant it references has not been defined. *`ifdef/`ifndef/`else* directives should always be terminated with an *`endif*.

engineer to not need to implement a specific feature that has already been implemented. IP cores come in two varieties: soft and hard IP. Soft IP comes in the form of code that has already been designed and tested; a good source for IP of this form is OpenCores. Hard IP does not come as code, but instead exists as modules that can be instantiated that are physically implemented by the synthesis tool. Two sources of hard IP are Synopsys' DesignWare library, and the libraries used by Xilinx's Coregen tool.