Hardware Modeling with Verilog

1.0 Introduction to Verilog

Verilog is a hardware description language (HDL) that allows the user to describe digital systems at a variety of different levels of abstraction. The system can be described simply in terms of its hardware components and their interconnections or it can be used to model the behavior of those components at levels ranging from mathematical expressions down to the behavior of the transistors that actually comprise the various components. Here we will concentrate on specifying interconnections and modeling behavior at the very highest levels. There are a number of books written about Verilog which may also be consulted.

2.0 Hierarchical Descriptions of Hardware Systems

The Verilog language describes a digital system as a set of modules. Each of these modules contains a set of input and output ports as well as a description of the contents of the module. The contents can be described in a structural or behavioral manner or in some combination of the two. A structural description is composed of the actual logical structure of the module, while a behavioral description contains no information about the actual internal structure of the module, but rather describes the module's behavior in a manner similar to higher level programming languages. Behavioral descriptions are not concerned with logical implementations, but rather with the task the module is to perform.

Modules can be thought of as building blocks. A Verilog description of a hardware system contains a top level module which may be composed of several lower level modules, each of which may be composed of several modules themselves. This hierarchical approach presents a very flexible environment in which to describe digital systems. A system may start out with just a few behavioral modules described and evolve into a more accurate description as behavior is replaced with structure in the form of lower level modules.

The register file module from the MIPS Light Verilog model is used as an example throughout this document to demonstrate Verilog concepts. It appears in its entirety in appendix A.

3.0 Defining and Instantiating Modules

A Verilog module definition is very similar to a procedure definition in other languages. The syntax for defining a module is shown in figure 1.

```
module <module_name> [(<list_of_ports>)];
    <input declarations>
    <output declarations>

    <statements>

endmodule
```

Figure 1

The optional list of ports contains the names of input and output ports with which the module communicates with other modules in the hardware model. A module that contains no input or output ports is totally isolated and most likely useless. While the top level module in the hierarchy will have no ports, all other modules should. After the opening module definition

statement which includes the parenthesized list of ports, each port must be declared as either an input or an output port. Figure 2 shows the module definition for the register file module used in MIPS Light.

```
module  rf(WCLK, RSaddr, RTaddr, RDaddr, RS, RT, RD);
input           WCLK;           // Write clock for RD write port
input   [4:0]   RSaddr;         // Read address for source read port
input   [4:0]   RTaddr;         // Read address for target read port
input   [4:0]   RDaddr;         // Store address for dest. write port
output  [31:0]  RS;             // Source read port
output  [31:0]  RT;             // Target read port
input   [31:0]  RD;             // Destination write port

   <statements removed>

endmodule       // rf -- Register File
```

Figure 2

The name of the module is **rf** and it contains seven ports—five input and two output ports. The register file used in MIPS Light has two read ports and one write port allowing it to produce the contents of two registers and write to one register simultaneously. Accordingly, it must have three inputs that indicate which registers are to be read and which is to be written. These three registers are known as the source, target and destination registers and their addresses (simply their register number) are indicated to the register file through the **RSaddr**, **RTaddr** and **RDaddr** input ports. Notice that each of these ports is declared to be a 5–bit vector allowing for the selection of any of the 32 registers. Square brackets (“[]”) are used to declare the range of bit numbers that an object contains with the most significant bit position first. Hence, the range [4:0] declares a five–bit vector (bits numbered 4 through 0) and the range [31:0] declares a 32–bit vector (bits numbered 31 through 0). If the range is omitted, the object is assumed to be composed of a single bit.

The register file must also contain ports to allow for the transfer of data to and from the indicated registers. Two of these ports are output ports used when reading a register (**RS** and **RT**) and one is an input port used when writing to a register (**RD**). Each of these ports is declared as a 32–bit vector as described above.

The remaining port, WCLK, is declared to be a single bit input. This port represents the write clock, used to indicate to the register file at what time the value on the **RD** input port should actually be written to the destination register indicated by **RDaddr**.

The Verilog code shown in figure 2 contains no information about the behavior or functionality of the register file, but it does completely identify the structure of the module as seen by other modules in the system. Hence, we can now discuss how modules are instantiated within other modules.

```
rf regfile(WCLK, RSaddr, RTaddr, RDaddr, RS, RT, RD);
```

Figure 3

The code in figure 3 simply instantiates a register file named **regfile** and connects its ports to the variables listed. In the example, the variables have the same names as those in the rf module definition, but they are not required to be the same. This style of instantiation closely resembles a procedure call in other programming languages.

4.0 Wire and Reg Data Types

Verilog provides for two fundamental data types called nets and registers. For our purposes, we will use the terms net and wire interchangeably although wires are actually only one type of net. The fundamental property of wires, unlike registers, is that they do not store values. Wires are used in Verilog to transmit values which are driven on them. Registers, on the other hand, have the distinct capability to store values. This distinction may seem trivial, but it is important to remember when creating a module in Verilog. One of the simplest and most confusing errors to make is to incorrectly declare the data type of an object.

Figure 5 provides an example from the MIPS Light register file module.

```
reg     [31:0]  RAM [0:31];     // A 32 x 32 bit memory array
reg     [31:0]  RS, RT;         // Declare read ports as registers
```

Figure 5

The first line of the code in figure 5 declares the variable **RAM** to be a 32 element array of 32–bit values. Each of these elements is declared to be a register. Note that the vector size of the elements precedes the variable name, while the array size follows the variable name. The variable **RAM** will be used by the register file module to store the values contained by each of the 32 registers. Since these values are to be stored rather than transmitted by the variable, **RAM** is declared to be of type register.

The second line of the code may be a bit more confusing. It declares that the output ports **RS** and **RT** (described above) are to be 32–bit registers. Since these ports appear to be used simply to transmit values from the register file to its outside connections, you may wonder why they are declared as registers rather than wires. This will become more clear in our discussion of behavioral modeling, but for now it is sufficient to understand that these ports must maintain their values for a specific amount of time and thus be declared as registers.

You may notice that other than **RS** and **RT** none of the register files ports are type defined. The default Verilog type is wire, so rather than explicitly define the remaining ports, we simply let Verilog implicitly type them as wires.

5.0 Behavioral Modeling

Modeling behavior with Verilog is very similar to programming in languages like C and Pascal. Traditional constructs such as if–then–else and case statements are available as well as a number of different loop constructs. These features of the language will not be described in detail here. In addition to more common programming language constructs, Verilog also provides a number of new statements useful for modeling hardware behavior. Three of these are discussed below.

5.1 The Initial Statement

The initial statement is generally used to initialize waveforms and other simulation or module variables before the actual simulation begins. All statements within an initial block occur simultaneously unless they are specifically separated by a delay. Once all statements within an initial block are executed, the block becomes inactive for the remainder of the simulation. The syntax of the initial statement is shown in figure 6.

initial
<statement>

Figure 6

<statement> represents either a single statement to be executed or multiple statements enclosed in a begin–end pair. Modules may contain multiple initial statements or none at all. All initial statements found throughout the model are executed concurrently at simulation time = 0. Figure 7 presents an example initial statement from the register file module.

```
initial
begin
    RAM[0] = 32'b0;             // Initialize register r0 with zeros
    RAM[29] = (`MEM_SIZE-8)*4;  // Initialize sp to end of memory
end
```

Figure 7

When the MIPS Light simulation begins, this initial statement is executed concurrently with other initial statements found in other modules. The first line of this initial block fills the first element of our register array with zeroes. The constant "32'b0" specifies both a value and a size. It represents the 32–bit operand whose value is specified in binary as 0. The second line initializes register r29 (the stack pointer and the twenty–ninth element in the array) to point to the end of our memory. (Actually it points to the seventh to the last word in the memory because a small number of instructions are placed in the last few words of memory to facilitate orderly program termination.) The constant `MEM_SIZE is defined elsewhere to be the number of words in the memory. Verilog allows macro definitions in a manner similar to C. The macro definition of `MEM_SIZE might look like this:

```
`define MEM_SIZE 1000
```

Notice that the "#" symbol used in C is replaced by "`" in Verilog. The same symbol must precede any use of the macro in the Verilog code.

5.2 The Always Statement

The basic Verilog statement for describing a process is the always statement. The always statement is similar to the initial statement except that it continually repeats rather than becoming disabled after having executed only once. Just as with the initial statement, a module may contain any number of always statements, including none at all. Description of the behavior of a module is placed in an always statement which may be thought of as a process acting and interacting concurrently with other processes described by other always statements. A module without an always statement is simply a structural description of hardware containing instantiated submodules and their interconnections.

The syntax of the always statement is identical to that of the initial statement. Three example always blocks from the register file module are shown in figure 8.

```
always @(negedge WCLK)          // At negedge clock...
begin
    if (RDaddr != 0)      // Register r0 is read-only
    RAM[RDaddr] = RD;     // Write RD data to cell addressed by RDaddr
end

always @(RAM[RSaddr])
begin
    RS = RAM[RSaddr];           // Fetch RS data using RSaddr
```

```
end

always @(RAM[RTaddr])
begin
    RT = RAM[RTaddr];           // Fetch RT data using RTaddr
end
```

Figure 8

The first thing to notice about these always statements is that they contain an additional specification not present in our discussion of initial statements. The first always block contains the phrase "@(negedge WCLK)." This part of the always statement serves to sensitize the block, effectively restricting the time during which it is active. Recall that always statements, unlike initial statements, continually repeat themselves. Sensitizing an always statement specifies when that statement is to become active, rather than allowing it to be active continuously throughout the simulation. The first always block is sensitized so that it is active only at the negative edge of **WCLK**. This always block becomes active when the **WCLK** signal changes from a 1 to a 0. This first block indicates that if the register to be written to is not r0, then the value of **RD** is stored in the element of our array corresponding to the destination register.

The second always block is activated when the value specified by **RAM[RSaddr]** changes. (negedge and posedge are keywords restricting the direction of the noticed change from 1 to 0 and 0 to 1 respectively.) When the value of **RAM[RSaddr]** changes, it is stored to the output port register **RS**. We can now see why the output ports **RS** and **RT** are defined to be registers. Since these always statements are not always active, the appropriate values are not always driven on the output ports which would be required if the ports were wires.

The third always block is nearly identical to the second; the difference is that it controls the **RT** port instead of the **RS** port.

5.3 The Assign Statement

The assign statement is not used in the register file module we have been discussing, but it might have been. You may have been tempted to declare the **RS** and **RT** ports as wires and simply not sensitize the always statements controlling them. While this may seem logical, it is incorrect. An unsensitized always statement is continuously executing and unless there is some delay associated with one of the statements it executes, the simulation clock will never advance. This situation produces what amounts to an infinite loop in the model and prevents the simulation from proceeding as expected. The solution to this problem is the assign statement which produces the desired result while at the same time allowing the simulation clock to advance. If we had declared the output ports to be wires, we could have replaced the two relevant always statements with the assign statements in figure 9.

```
assign RS = RAM[RSaddr];        // drive RS with value of reg RSaddr
assign RT = RAM[RTaddr];        // drive RT with value of reg RTaddr
```

Figure 9

If any of the inputs on the right hand side of the assignment change, the expression is evaluated and assigned to the port or variable on the left hand side. The choice between using always statements and assign statements where they are interchangeable is up to the programmer.

6.0 Delays

The MIPS Light model does not include any delays in its modules with the necessary exception of the module which generates the clock. Excluding delays from the model allows us to concentrate on the functionality of the machine without worrying about the timing requirements of hardware components with which the machine might actually be built.

A delay is specified by a pound sign and a number indicating the length of the delay in simulation time. **#10** specifies a delay of 10 simulation time units. Delays may be attached to almost any object in Verilog from wires to modules and they may be explicitly specified between statements in initial and always blocks as well. Figure 10 shows the code used to generate the clock for MIPS Light.

```
always
begin
        CLK = 0;
        # 10            // Delay 10 time units
        CLK = 1;
        # 10
        CLK = 0;
end
```

Figure 10

The code in figure 10 specifies an always block that executes continually. It produces a 50% duty cycle clock signal on CLK which is then routed to other modules in the model for use in determining event timing.

7.0 Verilog Comments

Comments can be represented in Verilog code in two ways. The first is identical to the method used in C whereby comments are delimited by the symbols **/*** and ***/**. The second allows comments to be placed on a single line and begins with the symbol **//** and terminates with the newline character. Thus, single line comments may be alone on a line or may follow code as in many of the examples used above.

8.0 System Tasks for Displaying Information

Verilog provides three very useful system tasks used for displaying information. They are **display**, **gr_waves** and **gr_regs**. Display simply prints text to the current xterm in a manner very similar to the printf function in C. Figure 11 shows an example call to the display task.

```
$display("data=%h address=%h",Din,Daddr);
```

Figure 11

The values of Din and Daddr are inserted in place of the two **%h** format controls just as you would expect if this were a printf call in C. The display task automatically appends a new line character to the end of the string. If you don't want to print a new line, use the write task which is identical to display except that a new line character is not automatically appended.

Another useful system task is the gr_waves task. This task produces the waves window which you invoked in programming assignment one by supplying the **+waves** command line argument. An example from the MIPS Light model is shown in figure 12.

```
$gr_waves (
   "CLK",`TOP.CLK,
   "PC",`CPU.PC,
   "IR_if",`CPU.Iin,
   "IR_rd",`CPU.IR2,
   "IR_ex",`CPU.IR3,
   <...>
);
```

Figure 12

As seen in figure 12, arguments of the gr_waves task must come in pairs. The first argument of the pair is a string indicating how the wave should be labeled in the waves window. The second argument in the pair is the signal that should be displayed.

The third useful system task is gr_regs. This task produces the register window which was activated in programming assignment one by supplying the command line argument **+regs**. It is very similar to the display task in that a number of strings with embedded format controls are specified delimited by commas. Like display, gr_regs implicitly appends a new line character to the end of each of these strings. Following the strings are comma separated operands which are to replace the format controls as they did with the display task. A simple example of the gr_regs task is shown in figure 13.

```
$gr_regs("Register 1 = %h",
         "Register 2 = %h",
         Reg1, Reg2
);
```

Figure 13

Table 1 contains descriptions of the various format control characters as recognized by both display and gr_regs.

h or H	display in hexadecimal
d or D	display in decimal
o or O	display in octal
b or B	display in binary
c or C	display ASCII character
m or M	display hierarchical name
s or S	display string

Table 1

Appendix A: Register File Example

```
//*****************************************************************************
// EE282 MIPS R3000 verilog model
//
// rf.v
//
// Register File
//
// The register file is a three ported memory device with 31
// read/write locations, each 32 bits wide.  A read-only cell
// (register r0) provides a 32-bit zero.
//*****************************************************************************

module  rf(WCLK, RSaddr, RTaddr, RDaddr, RS, RT, RD);

input          WCLK;          // Write clock for RD write port
input   [4:0]   RSaddr;        // Read address for source read port
input   [4:0]   RTaddr;        // Read address for target read port
input   [4:0]   RDaddr;        // Store address for dest. write port
output  [31:0]  RS;            // Source read port
output  [31:0]  RT;            // Target read port
input   [31:0]  RD;            // Destination write port

reg     [31:0]  RAM [0:31];    // A 32 x 32 bit memory array
reg     [31:0]  RS, RT;        // Declare read ports as registers

initial
begin
   RAM[0] = 32'b0;             // Initialize register r0 with zeros
   RAM[29] = (`MEM_SIZE-8)*4;  // Initialize sp to end of memory
end

always @(negedge WCLK)         // At negedge clock...
begin
   if (RDaddr != 0)        // Register r0 is read-only
   RAM[RDaddr] = RD;       // Write RD data to cell addressed by RDaddr
end

always @(RAM[RSaddr])
begin
   RS = RAM[RSaddr];           // Fetch RS data using RSaddr
end

always @(RAM[RTaddr])
begin
   RT = RAM[RTaddr];           // Fetch RT data using RTaddr
end

endmodule       // rf -- Register File
```

Appendix B: Why Doesn't It Work?

You've just finished making some changes to your Verilog code and you're sure it's correct, but it doesn't produce the results you expected. Presented here are a few of the most common errors which might produce unexpected results.

1. **An always statement is not properly sensitized.** As discussed in sections 5.2 and 5.3, an unsensitized always statement can produce an infinite loop which will prevent the simulation clock from being advanced. This will stall the simulation at time 0 and will result in no observable output. Additionally, if an always statement is not sensitized to all of the variables used in expressions within the always block you may notice a change in a signal and be confused about why another signal which is dependant on the first did not change. This is a common error and should be one of the first things you check if your model doesn't work the way you expect.

2. **A register or wire name is misspelled.** Because of the implicit typing Verilog uses, any previously unencountered variable is assumed to be a new variable of type wire. If, for example, you declare a port named **Out** to be a register and then forget to capitalize its name when you reference it, the new variable **out** will be implicitly declared to be a wire and will be used in place of the variable you intended. This error is sometimes difficult to recognize because Verilog will implicitly accept the misspelled variable and continue as if nothing was wrong.