7.4 Using a hardware description language

(Original section[1])

Today most digital design of processors and related hardware systems is done using a *hardware description language*. Such a language serves two purposes. First, it provides an abstract description of the hardware to simulate and debug the design. Second, with the use of logic synthesis and hardware compilation tools, this description can be compiled into the hardware implementation.

Harware description language: A programming language for describing hardware, used for generating simulations of a hardware design and also as input to synthesis tools that can generate actual hardware.

In this section, we introduce the hardware description language Verilog and show how it can be used for combinational design. In the rest of the appendix, we expand the use of Verilog to include design of sequential logic. In the optional sections of COD Chapter 4 (The Processor) that appear online, we use Verilog to describe processor implementations. In the optional section from COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) that appears online, we use system Verilog to describe cache controller implementations. System Verilog adds structures and some other useful features to Verilog.

Verilog is one of the two primary hardware description languages; the other is *VHDL*. Verilog is somewhat more heavily used in industry and is based on C, as opposed to VHDL, which is based on Ada. The reader generally familiar with C will find the basics of Verilog, which we use in this appendix, easy to follow. Readers already familiar with VHDL should find the concepts simple, provided they have been exposed to the syntax of C.

Verilog: One of the two most common hardware description languages.

VHDL: One of the two most common hardware description languages.

Verilog can specify both a behavioral and a structural definition of a digital system. A *behavioral specification* describes how a digital system functionally operates. A *structural specification* describes the detailed organization of a digital system, usually using a hierarchical description. A structural specification can be used to describe a hardware system in terms of a hierarchy of basic elements such as gates and switches. Thus, we could use Verilog to describe the exact contents of the truth tables and datapath of the last section.

Behavioral specification: Describes how a digital system operates functionally.

Structural specification: Describes how a digital system is organized in terms of a hierarchical connection of elements.

With the arrival of *hardware synthesis* tools, most designers now use Verilog or VHDL to structurally describe only the datapath, relying on logic synthesis to generate the control from a behavioral description. In addition, most CAD systems provide extensive libraries of standardized parts, such as ALUs, multiplexors, register files, memories, and programmable logic blocks, as well as basic gates.

Hardware synthesis tools: Computer-aided design software that can generate a gate-level design based on behavioral descriptions of a digital system.

Obtaining an acceptable result using libraries and logic synthesis requires that the specification be written with an eye toward the eventual synthesis and the desired outcome. For our simple designs, this primarily means making clear what we expect to be implemented in combinational logic and what we expect to require in sequential logic. In most of the examples we use in this section and the remainder of this appendix, we have written the Verilog with the eventual synthesis in mind.

Datatypes and operators in Verilog

There are two primary datatypes in Verilog:

1. A *wire* specifies a combinational signal.
2. A *reg* (register) holds a value, which can vary with time. A reg need not necessarily correspond to an actual register in an implementation, although it often will.

Wire: In Verilog, specifies a combinational signal.

Reg: In Verilog, a register.

A register or wire, named X, that is 64 bits wide is declared as an array: `reg [63:0] X` or `wire [63:0] X`, which also sets the index of 0 to designate the least significant bit of the register. Because we often want to access a subfield of a register or wire, we can refer to a contiguous set of bits of a register or wire with the notation [`starting bit: ending bit`], where both indices must be constant values.

An array of registers is used for a structure like a register file or memory. Thus, the declaration

```
reg [63:0] registerfile[0:31]
```

specifies a variable registerfile that is equivalent to a LEGv8 registerfile, where register 0 is the first. When accessing an array, we can refer to a single element, as in C, using the notation `registerfile[regnum]`.

The possible values for a register or wire in Verilog are

- 0 or 1, representing logical false or true
- X, representing unknown, the initial value given to all registers and to any wire not connected to something
- Z, representing the high-impedance state for tristate gates, which we will not discuss in this appendix

Constant values can be specified as decimal numbers as well as binary, octal, or hexadecimal. We often want to say exactly how large a constant field is in bits. This is done by prefixing the value with a decimal number specifying its size in bits. For example:

- `4'b0100` specifies a 4-bit binary constant with the value 4, as does `4'd4`.
- `-8'h4` specifies an 8-bit constant with the value -4 (in two's complement representation)

Values can also be concatenated by placing them within `{ }` separated by commas. The notation `{x{bitfield}}` replicates `bitfield x` times. For example:

- `{32{2'b01}}` creates a 64-bit value with the pattern 0101 ... 01.
- `{A[31:16],B[15:0]}` creates a value whose upper 16 bits come from `A` and whose lower 16 bits come from `B`.

Verilog provides the full set of unary and binary operators from C, including the arithmetic operators (+, -, *./), the logical operators (&, |, ~), the comparison operators (==, !=, >, <, <=, >=), the shift operators (<<, >>), and C's conditional operator (?, which is used in the form `condition ? expr1 : expr2` and returns `expr1` if the condition is true and `expr2` if it is false). Verilog adds a set of unary logic reduction operators (&, |, ^) that yield a single bit by applying the logical operator to all the bits of an operand. For example, `&A` returns the value obtained by ANDing all the bits of `A` together, and `^A` returns the reduction obtained by using exclusive OR on all the bits of `A`.

PARTICIPATION ACTIVITY 7.4.1: Check yourself: Constant values.

Which of the following define the same value as $1111\ 0000_{two}$?

1) `8'hF0`
   - True
   - False

2) `8'd240`
   - True
   - False

3) `{{4{1'b1}}, {4{1'b0}}}`
   - True
   - False

4) `{4'b1, 4'b0}`
   - True
   - False

Structure of a Verilog program

A Verilog program is structured as a set of modules, which may represent anything from a collection of logic gates to a complete system. Modules are similar to classes in C++, although not nearly as powerful. A module specifies its input and output ports, which describe the incoming and outgoing connections of a module. A module may also declare additional variables. The body of a module consists of:

- `initial` constructs, which can initialize `reg` variables
- Continuous assignments, which define only combinational logic
- `always` constructs, which can define either sequential or combinational logic
- Instances of other modules, which are used to implement the module being defined

Representing complex combinational logic in Verilog

A continuous assignment, which is indicated with the keyword `assign`, acts like a combinational logic function: the output is continuously assigned the value, and a change in the input values is reflected immediately in the output value. Wires may only be assigned values with continuous assignments. Using continuous assignments, we can define a module that implements a half-adder, as the figure below shows.

Figure 7.4.1: A Verilog module that defines a half-adder using continuous assignments (COD Figure A.4.1).

```
module half_adder (A,B,Sum,Carry);
```

Assign statements are one sure way to write Verilog that generates combinational logic. For more complex structures, however, assign statements may be awkward or tedious to use. It is also possible to use the `always` block of a module to describe a combinational logic element, although care must be taken. Using an `always` block allows the inclusion of Verilog control constructs, such as *if-then-else*, *case* statements, *for* statements, and *repeat* statements, to be used. These statements are similar to those in C with small changes.

An `always` block specifies an optional list of signals on which the block is sensitive (in a list starting with @). The `always` block is re-evaluated if any of the listed signals changes value; if the list is omitted, the `always` block is constantly re-evaluated. When an `always` block is specifying combinational logic, the *sensitivity list* should include all the input signals. If there are multiple Verilog statements to be executed in an `always` block, they are surrounded by the keywords `begin` and `end`, which take the place of the { and } in C. An `always` block thus looks like this:

```
always @(list of signals that cause reevaluation) begin
   Verilog statements including assignments and other control statements
end
```

Sensitivity list: The list of signals that specifies when an `always` block should be re-evaluated.

`Reg` variables may only be assigned inside an `always` block, using a procedural assignment statement (as distinguished from continuous assignment we saw earlier). There are, however, two different types of procedural assignments. The assignment operator = executes as it does in C; the right-hand side is evaluated, and the left-hand side is assigned the value. Furthermore, it executes like the normal C assignment statement: that is, it is completed before the next statement is executed. Hence, the assignment operator = has the name *blocking assignment*. This blocking can be useful in the generation of sequential logic, and we will return to it shortly. The other form of assignment (*nonblocking*) is indicated by <=. In nonblocking assignment, all right-hand sides of the assignments in an `always` group are evaluated and the assignments are done simultaneously. As a first example of combinational logic implemented using an `always` block, the figure below shows the implementation of a 4-to-1 multiplexor, which uses a `case` construct to make it easy to write. The `case` construct looks like a C `switch` statement. COD Figure A.4.3 (A Verilog behavioral definition of a LEGv8 ALU) shows a definition of a LEGv8 ALU, which also uses a `case` statement.

Blocking assignment: In Verilog, an assignment that completes before the execution of the next statement.

Nonblocking assignment: An assignment that continues after evaluating the right-hand side, assigning the left-hand side the value only after all right-hand sides are evaluated.

Figure 7.4.2: A Verilog definition of a 4-to-1 multiplexor with 32-bit inputs, using a case statement (COD Figure A.4.2).

The **case** statement acts like a **C switch** statement, except that in Verilog only the code associated with the selected case is executed (as if each case state had a break at the end) and there is no fall-through to the next statement.

```
module Mult4to1 (In1,In2,In3,In4,Sel,Out);
   input [31:0] In1, In2, In3, In4; /four 32-bit inputs
   input [1:0] Sel; //selector signal
   output reg [31:0] Out;// 32-bit output
   always @(In1, In2, In3, In4, Sel)
   case (Sel) //a 4->1 multiplexor
      0: Out <= In1;
      1: Out <= In2;
      2: Out <= In3;
      default: Out <= In4;
   endcase
endmodule
```

Figure 7.4.3: A Verilog behavioral definition of a LEGv8 ALU (COD Figure A.4.3).

This could be synthesized using a module library containing basic arithmetic and logical operations.

```
module LEGv8ALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [63:0] A,B;
   output reg [63:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
   always @(ALUctl, A, B) begin //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0;
      endcase
    end
endmodule
```

Since only reg variables may be assigned inside **always** blocks, when we want to describe combinational logic using an **always** block, care must be taken to ensure that the reg does not synthesize into a register. A variety of pitfalls are described in the elaboration below.

Elaboration

*Continuous assignment statements always yield combinational logic, but other Verilog structures, even when in **always** blocks, can yield unexpected results during logic synthesis. The most common problem is creating sequential logic by implying the existence of a latch or register, which results in an implementation that is both slower and more costly than perhaps intended. To ensure that the logic that you intend to be combinational is synthesized that way, make sure you do the following:*

1. *Place all combinational logic in a continuous assignment or an **always** block.*
2. *Make sure that all the signals used as inputs appear in the sensitivity list of an **always** block.*
3. *Ensure that every path through an **always** block assigns a value to the exact same set of bits.*

The last of these is the easiest to overlook; read through the example in COD Figure A.5.15 (A Verilog behavioral definition of a LEGv8 ALU) to convince yourself that this property is adhered to.

PARTICIPATION ACTIVITY 7.4.2: Check yourself: Blocking and nonblocking assignment.

Assuming all values are initially zero, what are the values of A and B after executing the following Verilog code inside an always block?

```
C = 1;
A <= C;
B = A;
```

1) **A**

- ○ 1
- ○ 0

2) B

- 1
- 0

(*1) This section is in original form.

Provide feedback on this section