A Verilog Primer

Carolyn Chen Matthew Matl Victor Ying

ELE 206 / COS 306

Contents

Introduction to Verilog

What is Verilog?

Welcome to ELE 206/COS 306's Verilog tutorial! Before you get started learning the details of Verilog, it's best to take a step back and gain some context. What exactly *is* Verilog, why is it useful, and where does it fit in the toolkit of a digital circuit designer?

Let's begin with some motivation. For now, it may seem to you that all digital design can accomplished by hand. First, you draw a circuit, and then you go build that circuit, hook it up to an oscilloscope, and test it out. For small circuits, this might be feasible, but modern circuits are often comprised of hundreds of millions to billions of transistors, with highly-complex modules interacting with one another in parallel all across a chip. Since pen-and-paper circuit designs do not scale to these sizes, engineers turned to Hardware Description Languages (HDLs).

An HDL is a programming language that can be used to describe a digital circuit. This type of programming language offers several major benefits over pen-and-paper design:

- **Scalability** – HDLs offer a simple, scalable way to represent large digital circuits as a sum of smaller modules. Each individual component can be created and thoroughly tested on its own, and then a large number of these components can be wired together with little effort on the designer's part.

- **Portability** – Working with a programming language offers many advantages over drawn designs. It's easier for a large number of engineers to collaborate when they're using a common language and platform, and using an HDL allows engineers to use version control and other helpful software-engineering constructs.

- **Abstraction and Generality** – HDL's can be used to directly describe logic gates, wire pathways, and physical connections in great detail. However, they also afford programmers the chance to abstract away some of these details and simply describe the desired behavior of a circuit. For example, Verilog allows users to use an edge-triggered register circuit *without* specifying exactly how that circuit should be constructed at the gate level!

  These abstractions, known generally as *behavioral modeling*, allow digital circuit designers to focus on the big, logical picture of how their circuit should work and separate the functionality of their circuits from implementation details. Designers can specify how their circuit works and then deal with lower-level issues (like what style of register to use or how to size transistors) later in the manufacturing process – and they can even automate these design decisions away and leave them up to a circuit synthesizer! This allows designers to be much more productive and makes it much easier to understand circuit designs at a high level.

- **Simulation** – Simulation tools are available that allow circuit designers to thoroughly test their designs before attempting to manufacture them into a physical circuit. This saves both time and money throughout the design process. Additionally, HDL's allow thorough testing of individual components of a larger circuit, helping designers to track down bugs in a speedy and direct manner.

- **Synthesis** – Finally, designs written in an HDL can be immediately compiled to run on a Field-Programmable Gate Array (FPGA) or transformed into a manufacturing design for an Application-Specific Integrated Circuit (ASIC). The process of programatically translating HDL designs into physical circuit plans is known as synthesis, and rapid, automatic synthesis makes HDLs especially valuable as digital design tools.

Verilog is a widely popular HDL that is used throughout both academia and industry. Although Verilog is a programming language, it and high-level programming languages, such as C and Java, have some fundamental differences:

- **Concurrency** – Verilog allows for the expression of conccurent computation, a common characteristic of digital circuits. Generally, statements in Verilog are concurrent in nature, i.e. all statements are executed in parallel. In comparison, languages like C are sequential, consisting of statements that are executed in sequence.

- **Complexity** – Verilog allows bit-level control of the hardware constructs it is used to model. Although Verilog is used to design digital systems at a relatively high-level of abstraction, languages like C and Java offer higher levels of abstraction. It can become challenging to build complex structures and algorithms in Verilog because the programmer controls how they work on a bit-level. Therefore, Verilog requires more careful computation design, especially considering Verilog's concurrency.
- **Hardware Mapping** – Often after simulation, Verilog code is eventually mapped to actual hardware gates, such as those on a Field-Programmable Gate Array (FPGA). This mapping is completed through a process called synthesis. However, some constructs and operations found in languages like C, such as division, are not synthesizable. Since there are many complex alternative implementations for such operators, the language does not provide for a single, quickly synthesized option. Therefore, a Verilog programmer must approach their design with the final hardware implementation in mind. In comparison, compiled C code, for example, is mapped to storage as bits that may or may not be executed on a processor.

The remainder of this book will teach you the details of Verilog and show you how to model complex digital circuits with simple programming constructs. This book does not present the entirety of the Verilog language, but the subset shown is sufficient to build any digital circuit.

Chapter 1

Verilog Basics and Combinational Logic

This chapter explains the fundamental building blocks that Verilog provides for constructing and simulating combinational logic circuits. We begin with a discussion of Verilog's basic datatypes, and then we provide an overview of logical, bitwise, and arithmetic operators that Verilog provides for manipulating those datatypes. After a brief description of Verilog's numerical constant specifications, we conclude with a full description of how to write combinational logic circuits.

1.1 Wires and Registers

Verilog has two basic datatypes – the `wire` and the `reg`. Each has its own specific properties and use cases, although they both share some useful properties. All Verilog code is designed to actually describe a physical circuit, and the `wire` and `reg` datatypes are the core structures through which data can flow through Verilog's high-level circuit descriptions. This section will describe each type in detail.

Wires

A Verilog `wire`, much like a real wire, is used for connections. A `wire` does not store a value, but instead derives its value from whatever is driving it. Usually, wires are either driven by some sort of combinational logic element (for example, an AND gate) or directly by a state-storing element (for example, a flip-flop).

Wires are usually driven with a digital value (either a logical zero or one), but they can also be left to float in a high-impedance state. In simulation, the `wire` can also take on an indeterminate value. A table of these values is shown below:

Signal	Description
0	Logical Low
1	Logical High
Z	High Impedance
X	Indeterminate

Declaring a wire is quite simple - just use the `wire` keyword, followed by an identifier (a name) for the wire. In Verilog, identifiers must begin with an alphabetic character or an underscore and may only contain alphanumeric characters, underscores, and dollar signs. More than one wire can be declared in a single Verilog statement by separating the identifiers with a comma.

```
1 wire Name1;
2 wire Name2, Name3, Name4;
```

A single `wire` can carry only one bit at a time. However, many logical structures require signals with a larger width. If more than one bit is needed to represent a signal, a `vector` of `wires` can be created using the following syntax:

```
1 wire [3:0] wire_vector;
```

The bracketed numbers form a *range specification*, which declares the width of the named vector in bits and assigns numbers to each of the bits for later reference. In this example, `wire_vector` is declared to be 4 bits wide, with the bits numbered from 3 to 0. The ordering of the numbers is arbitrary, but it is good practice to place the larger number to the left of the colon and to begin numbering from zero (unless there is a good reason not to).

The result of this wire declaration is illustrated in Figure 11.

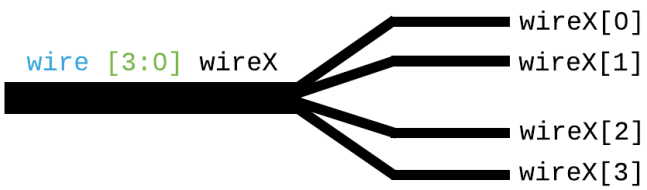


Figure 11: Illustration of a `wire` Vector

In order to access the individual bits of the wire, simply use a *bit selection* operator. The syntax should be familiar to programmers who have used a C-like language before – simply use brackets after the wire's identifier and place a number in the brackets to select that bit. For example,

```
1 wire_vector[3]
```

would correspond to the highest-order bit in the 4-bit `wire_vector` vector. Additionally, Verilog offers the ability to select a contiguous range of bits from a signal. For example,

```
1 wire_vector[2:1]
```

would correspond to the two middle bits (bit 2 and bit 1) of the `wire_vector` vector.

In addition to `vector` types, Verilog also permits the creation of arrays. Arrays are generally used when one structure must store several multi-bit signals. Declaring an array works in the same way as a range specification, except that the brackets now come *after* the wire's identifier in the declaration.

```
1 wire [3:0] wire_vector_array [63:0];
2 wire wire_array [63:0];
```

The first statement declares an *array* of 64 four-bit `wire` *vectors*, while the second declares an array of 64 one-bit `wire` types. Array access has a higher precedence than bit access in an array of vectors. In the following example, the first statement references the highest-numbered `wire` vector in the array (a 4-bit signal), and the second statement references the highest-order bit in the highest-numbered `wire` vector in the array (a 1-bit signal).

```
1 wire_vector_array[63]    // four bits
2 wire_vector_array[63][3] // one bit
```

Note that double forward slashes produce a comment – whatever comes after them on a line is ignored by the Verilog compiler.

Generally, arrays should only be used when you must group *multi-bit* signals. This is usually not required for wires, but is often used to create register files. If you are simply grouping a set of one-bit signals, just make a vector instead of an array.

Registers

A Verilog register, or `reg`, shares many properties with `wire` types. However, while the wire must be driven by a source and cannot maintain a value, a Verilog `reg` can serve as a state-storing element and actually drive `wire` types. Typically, `reg` types are used in sequential logic as edge-triggered flip flops, but with some special syntax they can be coerced to act exactly like a `wire`. In this way, the `reg` datatype serves a dual function in Verilog – it can act either as a connection element for combinational logic or as a storage element for sequential logic. The syntax that differentiates these uses of the `reg` type will be explained in the sections on combinational logic and sequential logic later in this book.

Like a wire, a `reg` can hold a single bit containing one of the four values listed above. The syntax for declaring a `reg` works in the exact same way as for a `wire`, and the `reg` can be instantiated as a vector and/or an array with a range specification in the exact same manner:

```
1 reg register1;
2 reg register2, register3;
3 reg [3:0] four_bit_wide_reg;
4 four_bit_wide_reg[2:1]; // middle two bits of reg
5 reg [3:0] four_bit_wide_array[31:0]; // 32 four-bit registers
```

1.2 Numerical Constants

In addition to `wire` and `reg` datatypes, Verilog also allows for the use of numerical constants in code. Numerical constants in Verilog are usually specified as a *based constant number*, which consists of three parts:

1. The size of the value in bits
2. An apostrophe, followed by the base indicator
3. The digits representing the value

The bitwidth of the constant is always specified as a decimal integer. The base indicator can take on any of the four following values:

Base Indicator	Type
d	Decimal (base 10)
h	Hexadecimal (base 16)
o	Octal (base 8)
b	Binary (base 2)

Finally, the actual value can be an arbitrary string of numbers and letters that represent a number in the specified base. Be careful that the specified bitwidth is adequate for the number represented – if the number is too large, the higher-order bits will be truncated.

For example, here are several ways to define a 4-bit constant for the decimal number ten:

```
4'd10
4'hA
4'b1010
```

If the constant's bitwidth can be inferred from the surrounding code (for example, if it is being used to set a value on a wire of known width), then you don't have to provide a bitwidth. The constant will have higher-order zero bits added to bring it up to the appropriate width, or its high-order bits will be removed to bring it down to the appropriate width.

Additionally, if you don't provide a base indicator, the constant will be inferred to be a decimal (base 10) number. For example, the following numbers are all equivalent.

```
6
3'd6
3'b110
```

Simply using decimal constants for numbers is perfectly acceptable, but it is often good practice to use *based constant number* form. Doing so makes it easier to find tricky bugs caused by bitwidth errors and truncation, and it will often make your code clearer.

1.3 Operators

Wires, registers, and constants are quite wonderful on their own, but they're not worth much if you can't manipulate them. This is where Verilog's operators come in.

Generally, operators will operate on either one or two `wire` or `reg` types to produce a new value. The new value produced by an operation is implicitly a `wire` with no name. Behind the scenes, this implicit `wire` is driven by a set of gates that performs the desired operation on the operands. To preserve the result of the operation, the value of this unnamed `wire` must be *assigned* to a named `wire` or `reg`.

The sections on combinational logic and sequential logic will explain how to save the result of an operation via *assignment*, so this section only explains the output created by each of Verilog's operators.

Arithmetic

Operator	Meaning
`+`	Addition
`-`	Subtraction

Verilog has full support for addition and subtraction, which are both operations that can be achieved via combinational circuits. Verilog also seemingly supports multiplication, division, and modulo operations, but **beware**. These operations are not *synthesizable*, which means that Verilog will not be able to convert them to actual hardware. They'll work in simulation, but fail when you try to program an FPGA. Do **not** use these operators – instead, stick to addition and subtraction.

Addition works in the simplest way possible. The two signals to be added together can be of different bitwidths. If they are, the smaller one will be padded with higher-order zeros before addition. The implicit `wire` that holds the result of the addition will always have enough bits to include a possible carry-out – for example, adding two four-bit numbers could produce a 5-bit result!

Subtraction works in the exact same way as addition, except that the second number is made negative via its two's complement representation before addition takes place. If we subtracted `4'b0001` from a 4-bit register containing `4'b0100`, then `4'b0001` would be converted into it's two's complement form (`4'b1111`). Then, this would be added to the `4'b0100` stored in the register, producing the result `5'b10011`. Usually, this will be stored back into a 4-bit register or used to drive a 4-bit wire, and the top bit will be truncated – thereby giving the correct, expected answer of `4'b0011`.

Bitwise

Operator	Meaning
`&`	AND
`|`	OR
`^`	XOR
`~`	NOT

Verilog supports the bitwise operators listed above. When applied to wires or registers, these operators will act on each bit in turn, and the result will be the same width as the largest operand.

These operations can also be chained – for example `~&` is the bitwise NAND operator.

Note that the NOT operator takes a single argument, and the operator appears just before the signal you intend to invert.

Concatenation

The concatenation operator, represented by curly brackets ({}) joins the bits from two or more wires/registers.

Simply place a list of the values to be concatenated inside the brackets to create a larger signal.

```
1 wire [2:0] a;
2 wire [2:0] b;
3 reg  [1:0] r;
4
5 // now, we can do something like {r, b, a}
```

In this example, the concatenation operator produces an 8-bit signal that has its highest-order two bits (`[7:6]`) set to the value of `reg r`, its middle three bits (`[5:3]`) set to the value of `wire b`, and its lowest three bits (`[2:0]`) set to the value of `wire a`.

Logical

Operator	Meaning
`==`	Equality
`!=`	Inequality
`!`	Logical Negation
`&&`	Logical AND
||	Logical OR
>	Greater Than
<	Less Than
>=	Greater Than or Equal
<=	Less Than or Equal

Verilog also supports several logical operators. These operators are very different from the bitwise operators, so **be careful**. Each logical operator will create a *one-bit value* – either a zero or a one. However, if *any* bit in any operand is a Z (high impedance) or X (unknown), then the logical expression will evaluate to Z.

The equality and inequality operators perform a bitwise comparison of their operands. They are exact inverses of each other, so either one or the other will evaluate to one.

The logical negation operator produces a one if all bits in the provided signal are zero. Otherwise, it produces a zero.

Logical AND and OR work in much the same way, treating multi-bit signals as true/one if any bit is one, and as false/zero only if all bits are set to zero. These operators do **not** take the bitwise AND/OR of their operators, but instead the logical AND/OR. This means that `4'b1000 && 4'b0001 == 1'b1`, while `4'b1000 & 4'b0001 == 4'b0000`. Be careful when deciding whether to use a logical or bitwise operator!

The comparison operators work as you might expect – however, be careful! They work on all numbers as if they are unsigned, zero-extending smaller signals automatically. Just be careful and realize that Verilog by default does not treat ANY numbers as signed. The only exception to this rule is the automatic use of two's complement during subtraction.

Shift

Operator	Meaning
>>	Logical Right Shift
<<	Logical Left Shift
>>>	Arithmetic Right Shift
<<<	Arithmetic Left Shift

Verilog also supports several shift operators. For each operator, the signal to be shifted is specified to the left of the operator and the shift amount is specified to the right. For example, the following code fragment produces a 5-bit signal equal to `wire x` shifted right by two bits.

```
1 wire [4:0] x;
2 // shift x right by 2: x >> 2
```

The logical operators always shift zeros into the result. The arithmetic left shift operator works in the same way as its logical counterpart.

However, the arithmetic right shift operator will fill in the most-significant-bit positions with the prior value of the highest-order bit to preserve sign. For example, `4'b1000 >> 1 == 4'b0100`, while `4'b1000 >>> 1 == 4'b1100`.

Other Operators

There are an assortment of other operators available in Verilog. Be cautious when using these, as not all of them are synthesizable into hardware. In particular, do **not** use the `===`, `!==`, division, power, or modulus operators in any code that you intend to be synthesized and run on an FPGA. It's fine to use these operators in simulation-specific code, but do not use them in hardware-ready code. Furthermore, minimize the use of more complex operators, e.g. multiplication, as it can significantly affect the time required for synthesis.

1.4 Structural and Behavioral Verilog

In Verilog, the level of abstraction for circuit design is typically classified as structural, behavioral, or a mix of the two.

Structural Verilog takes a bottom-up approach to digital circuit design. A structural model can be seen as a textual representation of a circuit schematic, where the circuit is abstracted at the level of logic gates. This offers the designer the flexibility to explore different hardware implementations of the same

function. For example, an exclusive-OR, XOR, function can be implemented using two AND-gates and one OR-gate or, alternatively, using one AND-gate and two OR-gates. Thus, structural Verilog gives the designer greater control over the final circuit.

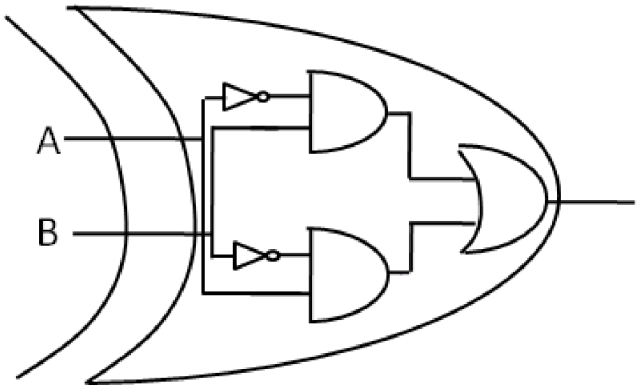


Figure 12: XOR implemented using two AND-gates and one OR-gate.

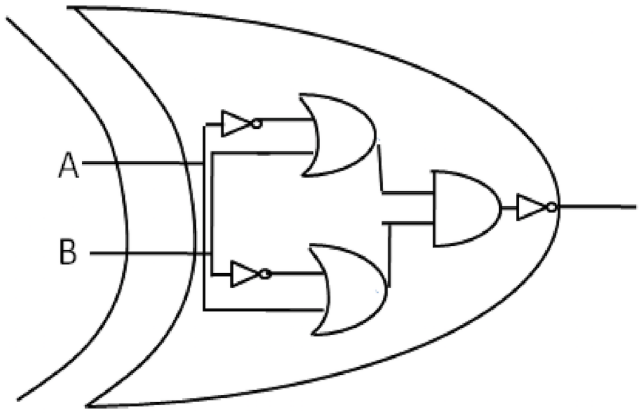


Figure 13: XOR implemented using one AND-gate and two OR-gates.

On the otherhand, *Behavioral* Verilog takes a top-down approach. A behavioral model expresses the behavior of a circuit without explicitly describing its structure. For example, instead of implementing an XOR function structurally, it's behavior is easily expressed in one Verilog statement:

```
1 assign Out = A ^ B;
```

In this case, the actual hardware implementation will be handled by the synthesizer. Thus, the designer now has little control of the gates used in the final circuit, but complex circuits can be abstracted more easily. Most production Verilog code contains a mixture of both structural and behavioral Verilog.

1.5 Combinational Logic

This section covers the details of how to represent combinational logic in Verilog. Combinational logic is continuous logic – logic that is stateless and does not require registers or a clock. The outputs of a combinational logic block are only dependent on its inputs at any instant in time.

Generally speaking, combinational logic in Verilog simply involves taking a few inputs from existing `wire` or `reg` types, performing a set of operations on them, and then driving an output `wire` by assigning the result of those operations to it.

Combinational logic naturally assigns to `wire` data types. However, as will be discussed later in this section, `reg` data types can be coerced to act like wires with some special syntax, and complex logic blocks will frequently target wire-like `reg` types instead of `wire` types.

There are two primary ways to construct combinational logic blocks in Verilog. If one is assigning to `wire` data types, then the style that is used is called *continuous assignment*. If one is assigning to `reg` data types (which are coerced to act like connection-oriented wires), then the style that is used is called *procedural assignment*. Each of these styles is described in detail in the following sections.

Continuous Assignment

The simplest way to create a combinational logic structure is the continuous `assign` statement. This statement sets the drivers for a `wire` or a vector of `wire` types from the output of an operation. The structure of the statement is represented below:

```
1 wire A;
2 assign A = B && C;
```

The identifier to the left of the equals sign is always the `wire` being assigned to. The objects to the right of the equals sign can be either wires or registers. Whenever the value of any structure on the right side of the equals sign changes, the value of the assigned `wire` is re-evaluated. In this example, `wire A` will always be equal to the logical AND of `B` and `C`.

This may seem a bit strange – in normal programming languages, we'd expect `A` to be set to `B && C` once, but when `B` or `C` change later on, we wouldn't expect `A` to change. However, Verilog is a *hardware-description language*, and hardware is inherently continuous and parallel. The operation `B && C` is continuous (specifically, a 1-bit AND gate), and using continuous assignment ensures that `A` is *continuously, constantly* equal to the result of `B && C` – even when `B` or `C` change. Whenever `B` or `C` change, `A` will change instantaneously.

A wire can be declared and continuously assigned to in the same statement. In this case, the `assign` keyword is dropped:

```
1 wire A = B && C;
```

Continuous assignment can also work on vector `wire` structures – just be sure to check that the result of the operation has the same bitwidth as the `wire` you're assigning to.

```
1 wire [4:0] A;
2 assign A = B & C;  // assigns values to all 5 bits in A
```

If the operation produces a result with a smaller bit-width than the wire being assigned to, the higher-order bits of the wire will be set to zero. If the operation's result has too many bits, the higher-order bits will be dropped when the result is assigned to the `wire`.

The assigned `wire` can be a vector created by using the *concatenation* operator. For example, the following code takes the bitwise AND of `B` and `C`, assigning the top bit to `A` and the lower bits to `D`.

```
1 wire A;
2 wire [3:0] D;
3 assign {A, D} = B & C;
```

Additionally, individual bits within a `wire` vector can be assigned to by using bit selection. For example, the following code sets the middle bit of `A` without assigning to its other bits.

```
1 wire [2:0] A;  // 3 bits wide
2 reg R;
3 wire B;
4
5 assign A[1] = R && B;
```

A final useful construct that is frequently used in continuous assignment statements is the *conditional operator*. This operator acts as a switch or multiplexer – it can set the value of its target to one of two inputs based on a conditional signal. The syntax for this operator is identical to the similar C operator:

```
1 assign A = D ? B : C;
```

If the `wire` or `reg` before the question mark holds a non-zero value, the assigned `wire` takes on the value of the variable between the question mark and colon. Otherwise, it takes on the value of the variable after the colon. It is valid to use `vector` wires or operation results as the conditional value – the conditional value will be treated as one as long as any bit in it is set to one.

The above code sample has the following truth table:

D	A
1	B
0	C

Continuous assignment is best for simple logical statements, like ANDing a set of values together or creating a simple multiplexor. If the logic for determining a signal is complex, consider breaking the logic up into multiple assignments or using easier-to-read procedural assignment, which is described next.

Procedural Assignment

Procedural assignment can be used to create a combinational logic block as well. In this case, the target of assignments is a `reg` datatype. Normally, a `reg` is a state-holding element, but when it is used in a specific procedural block, it becomes equivalent to a `wire`.

The syntax for combinational procedural assignment looks like this:

```
1 reg R;
2 always @(A, B) begin
3   R = A & B;
4 end
```

This statement is complex, so let's break it into parts. First, we have the `always` statement with the "at" sign followed by several signals in parentheses. This statement, known as the *always procedure*, runs the block of code between the `begin` and `end` keywords whenever the value of any of the signals

in parentheses changes. Thus, the signals in parentheses are known as the *sensitivity list*.

The code within the `always` procedure is stored between `begin` and `end` keywords. In Verilog, whitespace and indentation are insignificant, so blocks of code are marked with those keywords to make it clear which statements belong to which procedure.

Finally, we have a single statement in the body of the procedure that looks a lot like continuous assignment – just without the `assign` keyword. This type of assignment is known as *blocking assignment*, and may only be performed within a procedure. Whenever a procedure is run (in this example, when either `A` or `B` changes), statements that use *blocking assignment* are executed in-order from the top to the bottom of the procedural block. As each right-hand side expression is evaluated, its result is immediately stored in the left-hand side's `reg` variable.

Let's look at another example.

```
1 reg R, S;
2 always @(S, A, B) begin
3   S = A | B;
4   R = S & B;
5 end
```

Let's say that `A` changes its value. Immediately, the `always` block will be executed. The first statement will re-evaluate `(A | B)` and assign the result to `S`. Next, the second statement will evaluate `(S & B)` with the *new* value of `S` and then store it into `R`. Thus, `S` will always equal `(A | B)`, and `R` will always equal `(S & B)`.

What happens when the order of the statements is reversed?

```
1 reg R, S;
2 always @(S, A, B) begin
3   R = S & B;
4   S = A | B;
5 end
```

When `A` changes, the first statement will re-evaluate `(S & B)`, and `R` will not change. Next, `S` will receive its new value of `(A | B)`. However, despite the fact that `S` is in the sensitivity list for the procedure and has now changed, the procedure will **not** run again! This is because, in Verilog, a change to a variable within an `always` block cannot trigger that *same* `always` block. Any other `always` block that is sensitive to the value of `S` will immediately be scheduled to be run again, but the block where the change took place will not.

This example illustrates that the order of the statements in a combinational procedural block *does, in fact, matter* in some cases. In general, order

your statements so that a variable that is both assigned to and used within a single `always` block is assigned to before it is used. Alternatively, if you find that you are both using and assigning to a variable inside an `always` block, you can simply split that block in two.

For example, the following block will work correctly – `S` will always equal `(A | B)`, and `R` will always equal `(S & B)`. This is because a change to `S` in one block will cause an update to be scheduled in the other and vice-versa. Verilog only prevents updates to a variable within a block from scheduling another run of that same block – other blocks *will* be scheduled to update correctly.

```
1 reg R, S;
2 always @(S, B) begin
3   R = S & B;
4 end
5 always @(A, B) begin
6   S = A | B;
7 end
```

As a final note, if two statements in a procedural block assign to the same variable, the last statement executed will take precedence. For example, the following code assigns `S` to always equal `(A | B)`.

```
1 reg S;
2 always @(A, B) begin
3   S = A & B;
4   S = A | B; // takes precedence
5 end
```

Frequently, procedural blocks for combinational logic get very large, and it's often difficult to remember what signals should be in the sensitivity list. In order to ease development, Verilog has a catch-all sensitivity list. This special list automatically captures every identifier that is on the right-hand side of an assignment or used as a control-flow condition (basically, all identifiers that could cause a change in the way the procedure executes). The format for this catch-all sensitivity list is as follows:

```
1 always @( * ) begin
2   // statements
3 end
```

It is good practice to *always* use the *-based sensitivity list. It's far too easy to forget a variable when manually creating sensitivity lists, so always use the *-based list when creating procedural combinational logic.

Procedural combinational logic allows for the utilization of several common programming constructs. Inside procedures, we can use `if-else` and `case` statements to construct complex logical structures.

If-Else Blocks

Verilog provides `if-else` logic that should be familiar to anyone who has programmed before. Simply put, one can perform conditional assignment using `if` and/or `else` blocks within any procedure. The syntax for `if-else` blocks is simple:

```
reg X;
always @( * ) begin
  if ( Y & Z ) begin
    X = A;
  end
  else if ( Y & ~Z ) begin
    X = B;
  end
  else if ( ~Y & Z ) begin
    X = C;
  end
  else begin
    X = D;
  end
end
```

The `else if` and `else` blocks are optional, and any number of `else if` blocks may come after an `if` block.

An `if` block will execute if the value in parentheses is non-zero. If the value is a `vector`, it considered to be non-zero if any bit in the vector is a one.

An `else if` block works the same was as an `if`, except that it won't execute if any block before it in the `else if` chain was successfully executed during the current execution of the procedure.

The `else` block will execute if and only if all of the blocks in the `else if` chain above it failed to execute.

Thus, the above example has the following truth table for X:

Y	Z	X
0	0	D
0	1	C
1	0	B
1	1	A

As you can see, this logical block forms a multiplexer.

Be careful to always place `begin` and `end` statements at the beginning and end of each `if` and `else` block. Otherwise, unexpected behavior will occur when the blocks have more than one statement.

Case Statement

The `case` statement is an alterative to lengthy `else if` chains. It operates analogously to the C/C++ `switch` statement. The following example produces the same behavior as code in the if-else blocks above.

```
reg X;
wire W = {Y, Z};
always @( * ) begin
  case (W)
    2'd0: begin
      X = D;
    end
    2'd1: begin
      X = C;
    end
    2'd2: begin
      X = B;
    end
    default: begin
      X = A;
    end
  endcase
end
```

Simply put, the `case` statement will choose a block to execute based on the value of the variable in parentheses. If the value matches one of the labels for the block (like 2'd0 or 2'd1), then that block is executed. If no named blocks are matched, the `default` block (if it exists) will be executed.

The block labels must be constant values – do not try to use a variable as a case block label.

Avoiding Implicit Latching

When defining a procedural block for combinational logic, be sure that *every* variable that is assigned to in the block is *always* assigned to on *every* possible execution path through the procedure. A common way to achieve this is to set a default value for every variable at the beginning of the procedure. If a later statement also assigns to that variable, that later statement will take priority.

If a variable is not always given an updated value on every iteration of the procedure, it will seem to the compiler that the `reg` being assigned to should hold state in certain cases. Verilog will then synthesize a latch as part of the logic. A latch is a structure with memory, and is probably *not* what you want in a combinational block.

Follow good practice – if you have a variable in the block that is not assigned to on every path, just set a default value at the top of the procedure. In the following example, `X` will be set to its default value of zero unless Y is non-zero.

```
reg X;
always @( * ) begin
  X = 0; // set default value
  if ( Y ) begin
    X = 1;
  end
end
```

Tri-State Logic

Verilog also allows for the creation of tri-state buffers using combinational logic. These buffers can be used to create safe buses, which are wires that can be driven by several sources (just not simultaneously).

A tri-state buffer is a simple circuit that has an input and an enable signal. If the enable signal is active, the input will directly drive the buffer's output. Otherwise, the output will be left to *float*, or take on a high-impedance value until another circuit drives that output wire.

To create a tri-state buffer, use the following logic:

```
module TriBuffer(
  input in,
  input en,
  output out
);

  assign out = (en) ? in : 1'bz;

endmodule
```

If `en` is zero, then `out` is driven with Z, the high-impedance value. What this practically means is that `out` is no longer driven by `in` – but it's free to be driven by any other circuit connected to it! Assigning Z's to a wire will not override values from another circuit driving that wire.

1.6 Chapter 1 Review Questions

1.1 Describe the fundamental differences between a program written in a high-level language, e.g. Java, and a program written in Verilog? How does your approach to program design change when programming in Verilog?

1.2 What does it mean for a `wire` to be driven? What value does a `wire` take when it is not being driven? What about when it is being driven by more than one driver?

1.3 What is key difference between a `wire` and a `reg`? Why is a `reg` necessary in sequential logic design?

1.4 Read and understand the following Verilog code:

```
1 wire [3:0] x;
2 wire y;
3 assign y = (x == 4'd8) ? 1 : 0;
```

a) Explain what the indices `[3:0]` mean.

b) Explain what the literal `4'd8` means. What are **three** other ways this constant could be represented?

c) What is the `?` operator? How does it work?

1.5 Read and understand the following Verilog code:

```
1 wire [3:0] a;
2 wire [1:0] b;
3 wire [___] y;
4
5 assign a = 4'd14;
6 assign b = 2'b10;
7 assign y = a + b;
```

a) Although `wire a` and `wire b` are different widths, will they still be added correctly? Why?

b) What should the width of `wire y` be to handle all possible sums? What would happen in cases where the width were smaller or larger than that value?

1.6 Read and understand the following Verilog code:

```
wire [5:0] a, b, x, y, z;

assign a = 6'b101010;
assign b = 6'b010101;
assign x = a & b;
assign y = a && b;
assign z = !a;
```

a) Will the values for `x` and `y` be the same? Why?

b) What type of operator is `&`? What type of operator is `&&`? What is the difference between the two types of operators?

c) Assume the programmer was expecting `z` to equal `6'b010101` using the code above. What is the value of z? Is this the desired result? If not, how could you fix it without using `b`?

1.7 Consider you are designing a digital circuit to be implemented on a Field Programmable Gate Array (FPGA).

a) What is the process of mapping Verilog code to physical logic gates called?

b) What considerations must you make when designing your circuit in Verilog? Are these considerations the same if you were not implementing your circuit on an FPGA?

c) What could be possible effects of using a non-synthesizable operator, e.g. division, in your design?

1.8 What is the difference between continuous and procedural assignment in Verilog? Which one describes hardware more naturally? Why?

1.9 What is combinational logic? Intuitively, is a `wire` or `reg` more natural for the result of combinational logic? Why?

1.10 Read and understand the following Verilog code:

```
wire [5:0] x;

assign x = 5'b00001;
assign x = x + x;
```

a) What will the value of `x` while this circuit is in operation? Why?

b) Why can there not be two assignments to the same wire in a circuit? How does this extend to the physical properties of digital circuits on hardware?

1.11 Read and understand the following Verilog modules:

```
module SimpleCircuitA(
input a,
input b,
input c,
output f
);
  wire d;

  assign d = a || b;
  assign f = d && c;
endmodule
```

```
module SimpleCircuitB(
input a,
input b,
input c,
output f
);
  reg d;

  always @( a, b, c, d ) begin
    d = a || b;
    f = d && c;
  end
endmodule
```

a) Classify each module as either a structural or behavioral model.

b) How does the execution of code differ between these two implementations?

c) The sensitivity list for the `always` procedure in `SimpleCircuitB` contains four signals. Let's say the value of the signal `a` changes and the procedure is triggered. Consequently, the value of `d` changes, but is the procedure triggered again? Does the signal `d` need to be in the sensitivity list?

d) How might you rewrite the sensitivity list to be better?

e) For this circuit, which style of writing seems more intuitive? Why?

1.12 Read and understand the following Verilog code:

```
1  reg [3:0] x;
2  always @( * ) begin
3    // x is given some value
4    if (x) begin
5      // do something
6    end
7    else begin
8      // do something
9    end
10 end
```

Under what circumstances does the `if` block execute?

Chapter 2

Modules and Hierarchy

Verilog is used to describe and simulate digital circuits. If the circuit is simple, it can adequately be described with `reg` and `wire` instances in a single file. However, modern digital circuits are massive, and putting all of the wires and registers for an entire CPU into a single location would lead to duplication (think about how many adders or AND gates are in a CPU) and unreadable code.

A Verilog *module* is designed to solve these problems. A module is a description of a complete digital circuit – such as an AND gate, a flip-flop, or an adder – including its inputs, its outputs, and an internal description of how the circuit actually works. When a module is defined, it can be *instantiated* and used over and over again without having to re-describe the logical structures (wires, registers, operations, and assignments) that make up the module.

In this way, a module prevents duplication of code and allows engineers to break up the design of a massive system into small, easy-to-test parts.

Modules are such an important part of Verilog that *every* circuit design must be declared within a module!

2.1 Module Basics

Defining a Module

The first step to using modules is to define one. For example, let's design an AND gate together.

A module definition begins with a line that performs three functions. First, it uses the `module` keyword to let Verilog know that a module is being defined. Next, it sets a name for the module – for example, `MyAndGate`. Finally, it lists all of the inputs and output *ports* of the module. A *port* is simply a

connection used to send data to the module and receive data from the module. For example, a simple AND gate has two inputs (the signals to be ANDed) and one output (the result of the AND). The beginning of an AND gate's description could look like the following:

```
module MyAndGate(
  input  a,
  input  b,
  output res
);
```

The ordering of the input and output lists does not matter, but it is generally good practice to list the inputs first and the outputs afterwards. Extra spacing helps readability, although it is not required.

There are three types of ports: `input`, `output`, and `inout`. Input ports are always of type `wire` within the module, and they cannot be assigned to – they are inputs, and are therefore read-only. Output ports are also wires by default, but they can also be made into `reg` types by specifying that explicitly in the port list:

```
module X(output reg res);
```

Finally, `inout` ports are used for tri-state buses that the module can either drive or read from. They are always `wire` types.

Ports can be vector types, like so:

```
module X(
  input  [4:0] a,
  output [3:0] b
);
```

However, arrays can `never` be used in a port declaration. The following statement is illegal:

```
module Illegal(
  input [2:0] a[3:0], // illegal
  output b[4:0]  // also illegal
);
```

The next part of a module declaration is the body. It actually describes the functionality of the module using wires, registers, other modules, and combinational and sequential logic.

For example, the internals of an AND gate could be created using the bitwise AND operator and continuous assignment:

```
assign res = a & b;
```

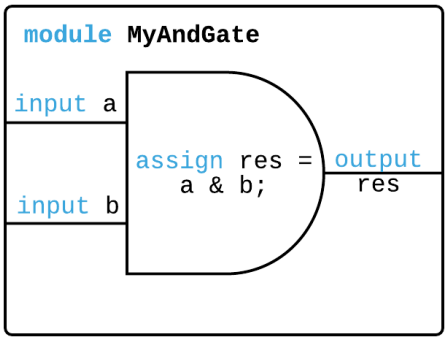
This simple body fully describes our AND gate – we put the inputs through a logical function to produce our output.

The final part of the module is the `endmodule` statement. Simply put, just add the following line of code at the end of a module to conclude its definition:

```
endmodule
```

So, in summary, our module definition looks like this:

```
// Simple two-input AND gate
module MyAndGate(
  input  a,
  input  b,
  output res
);

  // Internal Logic
  assign res = a & b;

endmodule
```



Instantiating a Module

Now that our module is fully defined, we probably want to put it to work. The process of actually using a module definition to create a useful Verilog logic block is called *instantiation*. Modules are re-usable and can be instantiated many times – all we have to do is provide a unique name for each instantiation. You can think about the module declaration as a description of the module – a blueprint, so to speak – and the instance of the module as an actual useful, physical unit. For example, Ford's schematics for the F-150 are the equivalent of a module declaration, while each actual truck is an instance of that module.

Module instantiation is easy. Simply state the name of the module, the name of the instance, and attach signals to the new instance's ports. Attaching signals to the ports is done by naming the port with a dot operator and then providing the name of the signal to connect to that port in parentheses:

```
wire signalA;
wire signalB;
wire result;

MyAndGate andgate1(
  .a(signalA),
  .b(signalB),
  .res(result)
```

```
);
```

In the above code snippet, we have created an instance of `MyAndGate` and named it `andgate1`. The parameter list links the inputs and outputs of the module instance with wires or registers in the current code block. For example, in this instance, `result` will be linked to and driven by the output `res` of the module. Additionally, `signalA` and `signalB` will be linked to inputs `a` and `b`, respectively. Thus, `result` will always equal the bitwise AND of `signalA` and `signalB`.

Note that either `wire` or `reg` types may be connected to the `input` ports of a module, but only `wire` types may be connected to the `output` or `inout` ports of a module. Modules directly drive whatever is connected to their `output` ports, and `wire` types are always safe to drive directly.

While not necessary, it is often good practice to declare a single module per file and to give the file the same name as the module. For example, the module we've declared would be saved in `MyAndGate.v`. If I need to instantiate this module in another file, I just use an include statement at the top of that file:

```
`include "/path/to/file/filename.v"

// more code that uses the file's module(s)
```

Note that the symbol before pre-processor directives like `include` is a backtick, the symbol on the key just above tab.

The `include` directive will essentially open the provided file and copy-paste it into the top of the current file during compilation. That way, module definitions from other files can be made available in the current file for usage and instantiation.

Top-Level Modules and Hierarchy

Modules can be defined with instantiations of other modules in their body. For example, the definition of a shift register might instantiate several AND or OR gate instances. In this example, each AND and OR instance would be a *child* of the shift register instance. Child-parent module relationships form a sort of hierarchical tree in Verilog, which we usually simply call the *hierarchy*.

The root of this hierarchy is always called the topmodule – a module with no ports and cannot be instantiated, but instead serves as the starting point for the full description of a circuit. Usually, topmodules are used as a base for actually testing a circuit via simulation – sort of like the `main()` method in a test client for a C++ or Java class.

Declaring a topmodule is very easy – it's the exact same thing as a normal module, but without the port list.

```
1 module MyModuleName;
2
3 // body here
4
5 endmodule
```

More description of how to use topmodules for simulation and testing will be provided further along in this text.

2.2 Built-in Logic Gates

Verilog has several built-in modules that represent the basic logic gates. These include `and`, `or`, `nand`, `nor`, `xor`, and `xnor`. The first port in each of these gates is the output, and all later ports are the inputs. The number of inputs is arbitrary except for the `not` gate, which has only one input.

To use one of these built-in gates, simply use the same syntax as above for instantiating a module. In this case, the ports are not named, so you can simply use position to hook up to the correct ports. For example, the following code declares a four-input AND gate:

```
1 wire out, in1, in2, in3, in4;
2
3 // Ports hooked up by positional order
4 and myand1(out, in1, in2, in3, in4);
```

In fact, you can always connect signals to module ports by positional matching, but it's best to explicitly name the connections using the dot operator as explained in the previous section.

2.3 Chapter 2 Review Questions

2.1 Explain why breaking down complex digital circuit designs into *modules* can be useful for hardware designers.

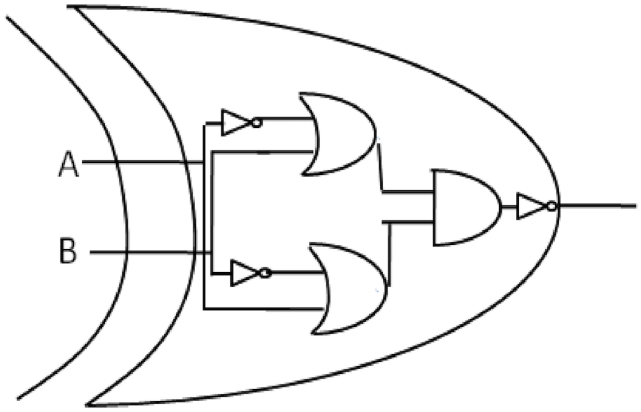
2.2 What is a *port*? What is the difference between a module `input` and `output` port?

2.3 **a)** Design a module for an OR gate.

b) Provide an example instantiation of the module.

2.4 How do you reference another file in Verilog? Is instantiating a module from another file different from instantiating a module from within the same file?

2.5 The following is an implementation of an XOR gate:



a) How could a programmer abstract this design into modules?

b) What is a *top-level module* in Verilog? What are they usually used for?

Chapter 3

Sequential Logic

This chapter covers the details of how to represent sequential logic in Verilog. Sequential logic is logic whose outputs depend on the present value of input signals *and/or* the past value of those inputs. This type of logic, unlike combinational logic, requires *memory*, which usually comes in the form of registers controlled by a clock.

In Verilog, sequential logic is created though the use of the `reg` datatype and a special class of `always` procedure.

3.1 Sequential Procedures

Typically, sequential circuits involve a storage mechanism – usually, a latch or a register – that is controlled by a clock. In particular, we focus on edge-triggered registers (such as the D flip-flop). This memory element stores a new value only on the rising or falling edge of a clock signal. Between edges, the register maintains its value.

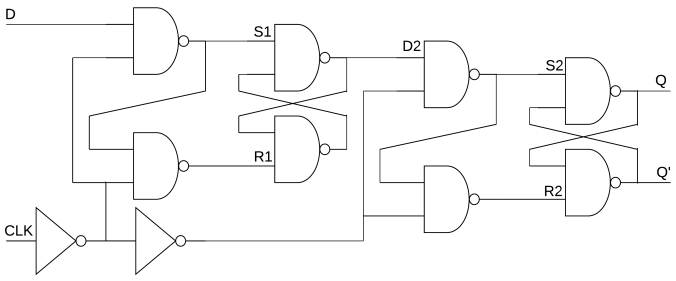


Figure 31: Master-Slave D Flip-Flop (Rising Edge)

Implementing a flip-flop can be done with complex combinational logic. For example, a classic master-slave D flip-flop (as shown in Figure 31), can be implemented as follows:

```
module DFF(
  input d,
  input clk,
  output q
);

  wire s1, r1, d2, s2, r2, qp;

  assign s1 = ~(d  & ~clk);
  assign r1 = ~(s1 & ~clk);
  assign d2 = ~(s1 & ~(r1 & d2));
  assign s2 = ~(d2 & clk);
  assign r2 = ~(s2 & clk);
  assign qp = ~(r2 & q);
  assign q  = ~(s2 & qp);

endmodule
```

However, the use of registers and memory elements is so common in hardware design that Verilog provides a much simpler way to create an edge-triggered flip-flop.

By using a `reg` datatype and specifying logic to update its values within a special `always` procedure, designers can greatly reduce the amount of code required to represent memory elements. Additionally, using the sequential `always` procedure allows for a synthesis program to pick the best flip-flop or register implementation for a given application.

Here's a simple rising-edge-triggered register written with a sequential procedure. It has the exact same functionality as the combinational code above, but it's much simpler and clearer.

```
module DFF(
  input d,
  input clk,
  output reg q
);

  always @(posedge clk) begin
    q <= d;
  end

endmodule
```

This looks a lot like the `always` procedure we used for combinational logic. However, there are a few subtle differences that are extremely important:

1. The sensitivity list no longer contains all right-hand-side signals. Instead, it contains a special modifier, `posedge`, that indicates that the procedure should be run on the *rising edge* (i.e. a zero-to-one transition) of the indicated signal. In this case, the procedure will run on the rising edge of the clock signal. There is an equivalent modifier called `negedge` that works as you'd expect.

2. All left-hand-side elements (elements which are assigned to in the procedure) still must be `reg` datatypes (or vectors of `reg` datatypes), but now these elements are no longer treated as wires. Instead, they're now treated as flip-flop style registers. Because the `always` procedure now only runs on positive clock edges, these `reg` types will only be assigned a new value on those edges!

3. Finally, we use a new assignment operator. In combinational logic procedures, we always assigned with the *blocking assignment* operator (`=`). Now, we use the *non-blocking assignment* operator (`<=`).

The procedure inside the `always @(posedge clk)` block will only be run when the clock transitions from zero to one. At that instant, the statements within the procedure will execute in-order, much like the `always` procedures in the chapter on combinational logic. However, the use of non-blocking assignment causes somewhat different behavior than you might expect.

3.2 Non-Blocking Assignment

Non-blocking assignment (`<=`) differs from blocking assignment (`=`) inside `always` procedures quite substantially, and it's very important to understand the differences.

As previously described, when a procedure is run, statements using blocking assignment are executed in-order. The values on the right-hand side of the equals sign are evaluated, and the result is immediately stored onto the `reg` on the left-hand side *before* moving on to the next statement.

In a similar way, non-blocking assignments execute in-order when a procedure is run. However, the registers on the left-hand side of the assignment are **not** immediately updated. Instead, the value of the expression on the right-hand side is computed, stored, and saved temporarily. Then, an update of

the left-hand side register is scheduled. Once all statements inside the procedure are executed, any scheduled updates caused by non-blocking assignments are run, storing the saved values into the appropriate registers. The order in which these updates are run is arbitrary.

Here's an example to illustrate this important but subtle difference. The following modules are identical except for their choice of assignment type within the `always @(posedge clk)` procedure.

```
1  module Blocking(
2    input clk,
3    input a,
4    output reg c
5  );
6    reg b;
7
8    always @(posedge clk) begin
9      b = a;
10     c = b;
11   end
12
13 endmodule
```

```
1  module Nonblocking(
2    input clk,
3    input a,
4    output reg c
5  );
6    reg b;
7
8    always @(posedge clk) begin
9      b <= a;
10     c <= b;
11   end
12
13 endmodule
```

Despite being almost identical, these two modules represent totally different circuits.

When the blocking procedure runs, `b` will be updated immediately with `a`'s value. Then, `c` will be updated with the *new* value of `b` (which equals `a`). Thus, the blocking procedure represents a simple rising-edge flip-flop, where `b` and `c` are always identical in value.

When the non-blocking procedure runs, `b` will be scheduled for an update with the current value of `a`, but it will *not* be updated immediately. Then, `c` will be scheduled for an update with the *current, old* value of `b` (as `b` has not yet been updated). When the procedure is finished, these updates will run. Register `b` will get the value of `a`, and register `c` will receive the *old* value of `b`. Thus, the non-blocking procedure represents a simple shift register that has a one-cycle delay between input and output.

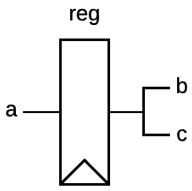


Figure 32: Blocking Circuit

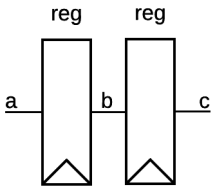


Figure 33: Non-Blocking Circuit

The behaviors of non-blocking and blocking assignment make them useful for different roles. When representing combinational logic, **always** use blocking assignment. When representing sequential logic, **always** use non-blocking assignment.

3.3 Complex Sequential Logic Blocks

All of the control-flow and conditional programming constructs available in combinational procedures (e.g. if-else blocks, case blocks) are available for use in sequential procedures.

There is one difference – you don't have to worry about implicit latching in sequential procedures. We *want* memory elements to be synthesized from sequential logic blocks! Therefore, you don't have to make sure that every variable is assigned to on every possible pass through the procedure – in fact, trying to provide default values in the code would not work with non-blocking assignment.

So, for example, the following code is a totally legitimate representation of a four-bit shift register, despite the fact that there are paths through the `always` procedure that do not assign to `out`.

```
module ShiftReg(
  input clk,
  input clr,
  input shift,
  input s_in,
  output reg [3:0] out
);

  always @(posedge clk) begin
    if (clr) begin
      out <= 4'd0;
    end
    else if (shift) begin
      out <= { out[2:0], s_in };
    end
  end

endmodule
```

Additionally, you can add more signals to the sensitivity list if needed. For example, if you want a register with an asynchronous reset, you could create it like this:

```
module ResetReg(
  input clk,
```

```
  input d_in,
  input rst,
  output reg d_out
);

  always @(posedge clk, posedge rst) begin
    if (rst) begin
      d_out <= 0;
    end
    else if (clk) begin
      d_out <= d_in;
    end
  end

endmodule
```

3.4 Chapter 3 Review Questions

3.1 In a few words, list the characteristic differences between combinational and sequential logic.

3.2 Read and understand the following Verilog code:

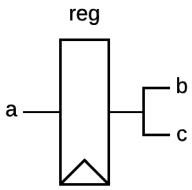
```
module DFF(
  input d,
  input clk,
  output reg q
);

  always @(posedge clk) begin
    q <= d;
  end

endmodule
```

a) What are the differences between the sensitivity lists of an `always` procedure in a sequential circuit implementation and in a combinational circuit implementation?

b) The code above represents a memory element. Is it rising- or falling-edge triggered? How would you change the code above to have the memory element triggered on the opposite clock edge?

c) Does the circuit above use *blocking* or *non-blocking* assignment? What are the differences between the two types of assignment?

3.3 Read and understand the following:



Desired Sequential Circuit

```
1  module FixMe(
2    input clk,
3    input a,
4    output reg c
5  );
6    reg b;
7
8    always @(posedge clk) begin
9      b = a;
10     c = b;
11   end
12
13 endmodule
```

Figure 34: Desired Circuit with Bad Verilog Implementation

a) Why is it good practice to use *non-blocking* assignment when representing sequential logic?

b) How would you change the code above to implement the desired sequential circuit using *non-blocking* assignment?

3.4 We have seen earlier that the `reg` datatype is coerced to behave like a wire in implementations of combinational logic. How does this differ in implementations of sequential logic?

Chapter 4

Named Constants

4.1 Local Parameters and Defines

Often, Verilog programmers will find themselves using the same numerical constants over and over again. For example, when creating a finite state machine (FSM), a wide constant is often used to uniquely identify each state. Remembering these constants can become tricky, and using them all over the code can make it difficult to understand. Thus, whenever you find yourself using a constant that has a non-trivial value over and over, it is best to name that constant using a `localparam` or a `define`.

Local parameters and define statements can both be used to define constants, although they work in slightly different ways. Local parameters must be declared within a module and can be used anywhere in that module, but they are inaccessible anywhere outside of that module. On the other hand, `define` statements are ususally made at the top of a file outside of any module. They work in the same way as C's `#define` statement – through textual substitution – and they can be used anywhere that file is included via an `include` statement. Note that constants created with a `define` statement must be referenced with a backtick in front of their names.

Here's an example of declaring constants using these two methods:

```
1 `define VALUE_1 32'd20 // Usable anywhere this file is included
2
3 module Foo(
4   input  [31:0] x,
5   input  [31:0] y,
6   output [31:0] b
7 );
8   localparam increment_val = 32'd5; // Only usable inside this module
9
```

```
10   assign b = x + y + increment_val + `VALUE1;
11
12 endmodule
```

Use a `localparam` wherever possible, reserving `define` statements for when you need to use the same set of values in several modules or files. Generally, constants are named with all uppercase letters. This is a good convention – try to stick to it.

Finally, do not use these structures to define every constant in your program. Only use them for important constants with a particular significance that is not obvious from their value alone. For example, the states in an FSM or the length of a memory's array are good candidates for using a set of local parameters, but you shouldn't name a local parameter for one and zero. Basically, just use common sense when deciding whether a constant should be named.

4.2 Module Parameters

Often, when building a module, it is useful to allow modification of certain constants within that module's definition upon instantiation. For example, let's say we have a generic module for a register file. We don't want to have an individual re-definition of that module for every possible register bit-width and file length. Instead, we'd prefer to define the module once and be able to specify the preferred width and length of the file every time we instantiate it.

This is where module `parameter` types come in. They work in much the same way as a `localparam`, but they can be modified on an instance-by-instance basis.

For example, if we were declaring our register file, we could do it as follows:

```
1  module RORegFile
2  #(
3    parameter BIT_WIDTH  = 8,
4    parameter ADDR_WIDTH = 6,
5    parameter LENGTH     = 64
6  )(
7    input                   clk,
8    input  [ADDR_WIDTH-1:0] r_addr,
9    output [BIT_WIDTH-1:0]  r_data
10 );
11
12   reg [BIT_WIDTH-1:0] rfile[LENGTH-1:0];
13
14   // Internals here
```

```
15
16 endmodule
```

When declaring modifiable module `parameter` types, just place them in a hashtagged parameter list just before the module's port declarations. Then, anywhere in the module (including port declarations), you can use the parameters as constants.

The values you provide in the parameter list are default values. If a user doesn't specify a new value for the parameters when the module is instantiated (which they can do by just decaring the module normally), then the parameters will take on their specified default values.

However, if a user wanted to instantiate this module with modified parameters, they would use a hashtagged parameter list between the module name and the instance name in their instantiation:

```
1  wire clock;
2  wire [3:0] data;
3  wire [2:0] address;
4
5  RORegFile #(
6    .BIT_WIDTH(4),
7    .ADDR_WIDTH(3),
8    .LENGTH(8)
9  ) regfile(
10   .clk(clock),
11   .r_addr(address),
12   .r_data(data)
13 );
```

This register file now has a bit-width of 4, an address width of 3, and 8 total entries.

4.3 Chapter 4 Review Questions

4.1 List characteristic similiarities and differences between the `localparam` and `define` statement?

4.2 Read and understand the following Verilog code:

```
module MyBitwiseAndGate(
  input [1:0] a,
  input [1:0] b,
  output [1:0] out
);

  assign out = a & b;

endmodule
```

a) What is a module `parameter`?

b) The above module is used to compute the bitwise AND between two 2-bit wire vectors. Assume you have a large circuit design that also needs a module to compute the bitwise AND between two 8-bit wire vectors. Update the code above with module parameters to offer this functionality without the need to create a separate module.

c) Provide an example instantiation of the module when used to perform the bitwise AND for two 8-bit wire vectors.

Chapter 5

Finite State Machines

The Finite-State Machine (FSM) is the heart and soul of any control module in digital logic design. There are many ways to design an FSM, but FSM implementation in Verilog can be made substantially easier if a few key guidelines are followed. This chapter lays out a pattern that can be used to implement any FSM in a clear, concise, and functional manner.

5.1 Components of an FSM

Any finite state machine is composed of three main parts:

1. A state register that holds the current state of the machine.
2. Combinational logic that converts the current state and the current inputs into outputs.
3. Combinational logic that converts the inputs and current state into a value for the next state, which will be loaded into the state register on the next clock edge.

A diagram of these components is shown in Figure 51.

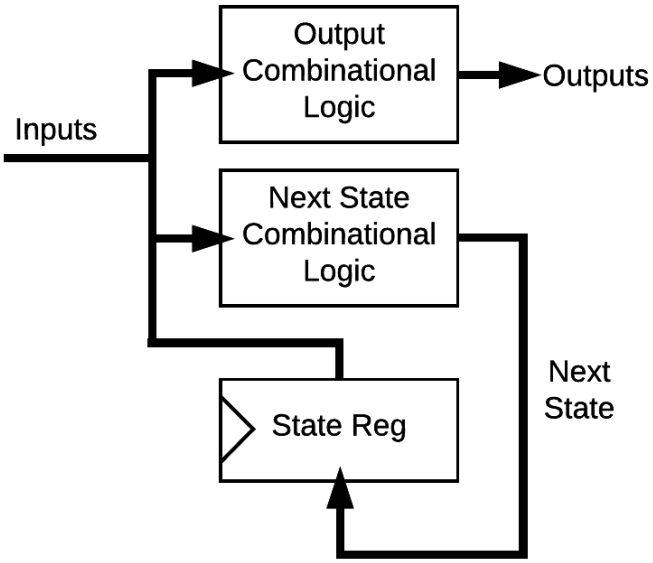


Figure 51: Components of an FSM

Splitting these three components apart makes for good, easily-maintainable design and easy-to-read code. The following sections explain in detail how to implement each of these components in clean, synthesizable Verilog.

State Register

The state register is the simplest part of the FSM. Simply put, it should only have two functions. When a reset signal is high during the relevant clock edge, the FSM should enter its inital state. Otherwise, the FSM should load the value produced by the next state logic on every relevant clock edge.

Implementing this in Verilog can be performed simply by using a sequential procedure. Note that the individual state names should be created as local parameters. Naming them with uppercase letters helps to distinguish them as constants.

```
1 reg [1:0] state;
2 reg [1:0] next_state; // really a wire type
3
4 // Local Parameters for States
5 localparam STATE_INITIAL = 2'd0;
```

```
6  localparam STATE_ONE     = 2'd1;
7  localparam STATE_TWO     = 2'd2;
8  localparam STATE_THREE   = 2'd3;
9
10 always @(posedge clk) begin
11   if (rst) begin
12     state <= STATE_INITIAL;
13   end
14   else begin
15     state <= next_state;
16   end
17 end
```

Next State Logic

The next state logic's job is to take the inputs and the current state and produce a wire that holds the next state in the FSM. The next state logic will ignore the `rst` input. If `rst` is one, the state register will automatically ignore `next_state` and load the initial state, so there's no need to worry about it in the next state logic block. If this logic is simple, it can be done with continuous assignment, but if it's complex, it's best to use a combinational logic procedure.

```
1  wire in1, in2;
2  reg [1:0] state;
3  reg [1:0] next_state;
4
5  always @( * ) begin
6    next_state = state;
7    if (in1 && state == STATE_INITIAL) begin
8      next_state = STATE_ONE;
9    end
10   else if (in2 && state == STATE_ONE) begin
11     next_state = STATE_TWO;
12   end
13 end
```

Once again, be careful to assign a default value to the `next_state` variable to avoid implicit latching. A good default value is the current state, which often makes it easier to debug the FSM (as the FSM will pause in the problematic state).

Output Logic

Finally, the output logic sets the output values based on either just the current state (a Moore machine) or a combination of the current state and the current inputs (a Mealy machine). Once again, if the logic is complex, use a combinational `always @( * )` procedure – and don't forget to use blocking assignment!

```
wire out;

always @( * ) begin
  out = 0;
  if (state == STATE_ONE || state == STATE_TWO) begin
    out = 1;
  end
end
```

5.2 Example RTL Design Problem

This section will cover the full RTL design process, including FSM and datapath generation. It should help to clarify any further questions on how to build an FSM in Verilog and how to use one as a control circuit for a datapath.

The Problem

This problem is taken from Section 5.2 of Frank Vahid's *Digital Design, 2nd Edition*.

We want to build a custom processor that captures the behavior of a simple soda machine dispenser. The machine will dispense a soda whenever enough money has been inserted. For simplicity, the machine will not give change and only dispenses one type of soda.

A block diagram of this processor is shown in Figure 52. There are three inputs and one output:

1. Signal S – An 8-bit signal that indicates the total price of a soda.
2. Signal A – An 8-bit signal that indicates the value of the most-recently-inserted coin in cents.
3. Signal C – A coin detector, which provides a 1-bit signal that is 1 for a single clock cycle whenever a coin is inserted.

4. Signal D – The output of the FSM, which should be set to 1 for a single cycle whenever a soda should be dispensed.

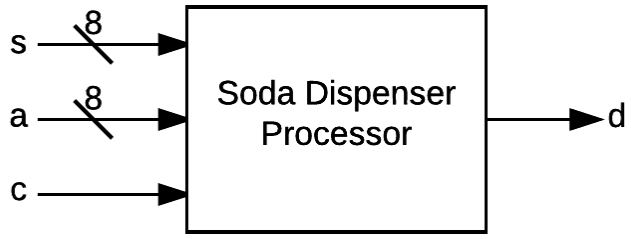


Figure 52: Block Diagram for Soda Processor

An FSM alone is not enough to solve this problem. FSMs make transitions between states based on simple one-bit boolean values, and they cannot store extra data aside from the current state or perform arithmetic on extra data. For this problem, we'll have to store the amount of money that has been deposited so that we know when to dispense a can, and we'll have to add money to that total when coins are inserted. Unless we want to have a state for every possible order of coin inputs, an FSM won't work by itself.

To solve this issue, we need to split our design into several phases:

1. **Create a High-Level State Machine (HLSM)**. This is like an FSM, except that we can label states with complex actions (like adding a number to some local storage) and can label state transitions with complex conditions (rather than simple one-bit booleans).

2. **Create a Datapath**. The datapath should handle all complex actions that an FSM cannot handle (for example, local storage, multi-bit inputs, and arithmetic operations). It should expose simple control signals that allow manipulation of the datapath's storage and arithmetic units. Finally, it should export simple boolean indicator signals that reveal the results of datapath operations (for example, a comparison of two large values).

3. **Create a Controller**. The controller should be a simple FSM. Its inputs come from external boolean signals and from the datapath's simple

one-bit indicators, and its outputs should be the control signals that manipulate the datapath appropriately.

4. **Connect the Datapath to the Controller**.

The HLSM should be fairly straighforward to create, and it can be converted into a working circuit fairly easily by splitting the datapath and controller.

First, all local storage, complex conditionals, and arithmetic manipulations should be abstracted into the datapth. The datapath should provide simple control signals that enable these operations and simple indicator signals that tell the controller the results of complex operations.

Rather than listing the complex operations that should be performed in each state, the HLSM can now simply list which control signals should be turned on for each state. Additionally, any complex conditionals on state transitions can now be replaced with the simple indicator signals from the datapath.

Once the complex operations and conditionals are replaced with simple control and indicator symbols, the HLSM becomes an FSM. Implementing an FSM is simple and straighforward. Once the FSM is complete, it can be connected to the datapath to form a fully working circuit.

A High-Level State Machine

A high-level state machine is similar to an FSM, but with additional features like local storage and arithmetic operations. Essentially, an HLSM is a high-level flowchart that describes how the circuit should function. Figure 53 shows the HLSM for our soda dispenser.

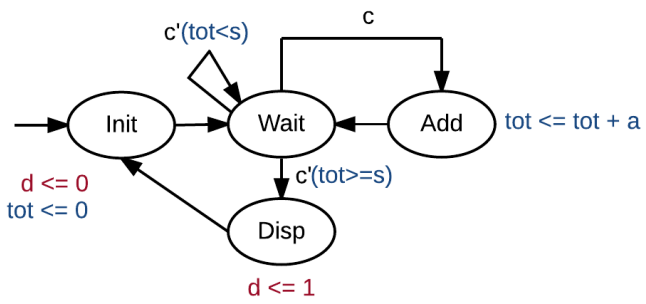


Figure 53: High-Level State Machine for Soda Processor

In this HLSM, each transition is implicitly ANDed with a rising clock edge. Output assignments are depicted in red, and local storage + arithmetic operations are depicted in blue. If an output is not explicitly depicted within a state, its value is assumed to be zero for that state.

Let's trace through what happens in this HLSM during normal operations. First, the INIT state is entered. The output, `d`, is set to zero, and the local storage that keeps track of the total amount of money currently added to the machine is reset. On the next clock edge, the WAIT state is entered. When a coin is put in, the ADD state is entered, and the value of the coin is added to the total. Then, we go back to the WAIT state. Once the value of total is greater than or equal to the value of the soda, the machine enters the DISP state for a single cycle. The dispense output (`d`) is set to one before we return to the INIT state.

Creating the Datapath

Identifying the components that the datapath needs should derive directly from the complex actions, complex conditionals, and local storage used by the HLSM.

In this example, we need the following components:

- An 8-bit register, `tot`, that holds the amount of money currently deposited. It should be able to load a new value (`tot + a`), be reset to zero, or maintain its current value.
- An 8-bit adder to add `tot` to `a`.

- An 8-bit comparator to compare `tot` to `s`. This comparator should output 1 if `tot` is greater than or equal to `s`.

A summary of this datapath is shown in Figure 54. It has two data inputs (`s` and `a`), two control inputs (`tot_ld` and `tot_clr`) that manipulate the `tot` register, and one boolean indicator (`tot_gte_s`) that is one if `tot` is greater than or equal to `s`.

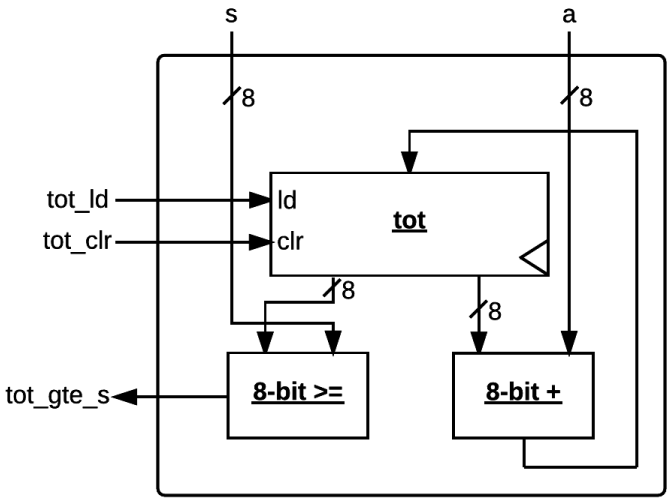


Figure 54: Datapath for Soda Processor

Now, let's implement this datapath as a module in Verilog! We have a bit of sequential logic to create the `tot` register and some combinational logic to calculate `tot_gte_s`.

```
1  module SodaDatapath(
2    input [7:0] s,          // Cost of a soda
3    input [7:0] a,          // Value of most recent coin
4    input       clk,        // Clock
5    input       tot_ld,     // Adds a to total
6    input       tot_clr,    // Resets total to zero
7    output      tot_gte_s   // Is total >= s?
8  );
9
10   reg [7:0] tot;          // Sum of money in datapath
11
```

```
12   // Sequential logic for tot register
13   always @(posedge clk) begin
14     if (tot_clr) begin
15       tot <= 8'd0;        // Clear tot register
16     end
17     else if (tot_ld) begin
18       tot <= tot + a;     // Add A to tot register
19     end
20   end
21
22   // Comparison output logic
23   assign tot_gte_s = (tot >= s);
24
25 endmodule
```

Creating the Controller

Now, let's revise the HLSM into an FSM using the datapath's control signals and indicator values. Simply replace the complex conditionals and actions (originally labelled in blue) with output control signals and boolean indicators. The result of this transformation in shown in Figure 55.

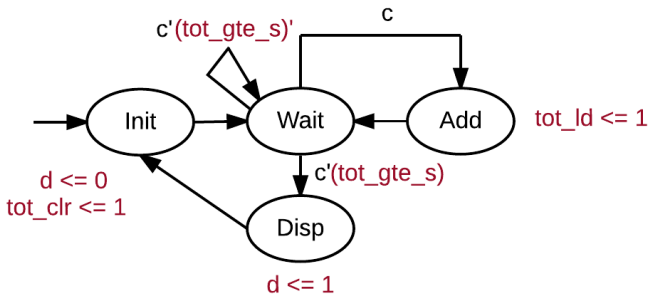


Figure 55: FSM for Soda Control Unit

Now, translation of this FSM into Verilog is straightforward.

```
1  module SodaControl(
2    input c,
3    input clk,
4    input rst,
5    input tot_gte_s,
```

```
6   output d,
7   output tot_ld,
8   output tot_clr
9 );
10
11  // State Values
12  localparam STATE_INIT = 2'd0;
13  localparam STATE_WAIT = 2'd1;
14  localparam STATE_ADD  = 2'd2;
15  localparam STATE_DISP = 2'd3;
16
17  reg [1:0] state;
18  reg [1:0] next_state;
19
20  // State Register Logic
21  always @(posedge clk) begin
22    if (rst) begin
23      state <= STATE_INIT;
24    end
25    else begin
26      state <= next_state;
27    end
28  end
29
30  // Next State Logic
31  always @( * ) begin
32    next_state = state;
33
34    case (state)
35      // Initial State
36      STATE_INIT: begin
37        next_state = STATE_WAIT;
38      end
39
40      // Waiting State
41      STATE_WAIT: begin
42        if (c) begin
43          next_state = STATE_ADD;
44        end
45        else if (tot_gte_s) begin
46          next_state = STATE_DISP;
47        end
48        else begin
49          next_state = STATE_WAIT;
50        end
51      end
52
53      // Adding State
54      STATE_ADD: begin
```

```
55          next_state = STATE_WAIT;
56        end
57
58        // Dispensing State
59        STATE_DISP: begin
60          next_state = STATE_INIT;
61        end
62      endcase
63    end
64
65    // Output Logic
66    assign d       = (state == STATE_DISP);
67    assign tot_clr = (state == STATE_INIT);
68    assign tot_ld  = (state == STATE_ADD);
69
70  endmodule
```

Note the use of the three sections of an FSM: state register update logic, next state combinational logic, and output combinational logic.

Connecting the Modules

Now that the datapath and controller modules are ready to be used, we simply need to connect them in a wrapper module:

```
1  module SodaDispenser(
2    input clk,
3    input rst,
4    input [7:0] s,
5    input [7:0] a,
6    input c,
7    output d
8  );
9
10   wire tot_ld;
11   wire tot_clr;
12   wire tot_gte_s;
13
14   SodaDatapath dpath(
15     .s         (s),
16     .a         (a),
17     .clk       (clk),
18     .tot_ld    (tot_ld),
19     .tot_clr   (tot_clr),
20     .tot_gte_s (tot_gte_s)
21   );
22
23   SodaControl ctrl(
```

```
24      .c          (c),
25      .clk        (clk),
26      .rst        (rst),
27      .tot_gte_s  (tot_gte_s),
28      .d          (d),
29      .tot_ld     (tot_ld)
30      .tot_clr    (tot_clr)
31    );
32
33  endmodule
```

This module could be hooked up to a testbench topmodule for testing or to a synthesis topmodule for programming an FPGA.

5.3 Chapter 5 Review Questions

5.1 **a)** What is the purpose of a *state register* in an FSM?

b) Is the implementation of a state register in Verilog *combinational* or *sequential*?

c) Typically, what are the two functions of a state register?

5.2 Assume that you are using a combinational logic procedure, i.e. using an `always@( * )` procedure, to implement next state logic for an FSM.

a) What is the purpose of *next state logic* in an FSM?

b) How is the next state logic affected when the FSM is reset, i.e. when the `reset` signal goes high?

c) How can you prevent *implicit latching* in your next state logic?

5.3 In a few words, describe the difference between a Moore machine and a Mealy machine.

Chapter 6

Generating Code with Loops

Sometimes it is necessary to generate a block of code automatically. For example, a large register file might need to be cleared out. One could write an individual Verilog statement to clear out each line, but that would be tedious and lead to ugly code. This is where Verilog for loops come in!

A note of caution before we describe in detail how to use for loops. Verilog's for loops should **only be used to generate static code**. This means that the bounds of the loop should be a simple integer that is either incremented or decremented each iteration, and the limit of the loop should be a constant that is known at compile time. If you find yourself writing a loop that you couldn't unroll statically by hand, **stop** and think about what you're doing.

6.1 Generate Loops

The generate loop's syntax mirrors the for loop in C, except that all generate loops are automatically unrolled into a series of Verilog statements. Here's an example of how to use a generate loop to clear out a register file:

```
1  reg [31:0] regfile [15:0];
2
3  genvar index;
4  generate
5      for (index = 0; index < 16; index = index + 1) begin
6        always @(posedge clk) begin
7          if (rst) begin
8            regfile[index] <= 32'd0;
9          end
10       end
11     end
12 endgenerate
```

This creates sixteen `always` procedures, each of which controls one entry in the register file. Avoid putting a generate loop *inside* a procedure – your code won't compile. You can even generate several copies of the same module – just place your module instantiation statement within a generate loop.

```
1 genvar index;
2 generate
3   for (index = 0; index < 64; index = index + 1) begin
4     MyModule mod(.x(x[index]), .y(y[index]));
5   end
6 endgenerate
```

Module names will be generated automatically (`mod[0]`, `mod[1]`, ...).

Use generate loops sparingly – they're only really necessary when dealing with repetitive data structures such as those you see in memory units or register files.

Chapter 7

Testbenches and Simulation

Once a module has been created, it is good practice to rigorously test it before trying to synthesize it to program an FPGA (Field-Programmable Gate Array) or manufacture an ASIC (Application-Specific Integrated Circuit). Generally, we test modules by running them through simulated unit and functional tests. These tests provide a large variety of possible inputs or common use cases for the module and then check the module's outputs to ensure that it behaves as expected.

7.1 The Top Module

All testbenches are written as a top-level module (i.e. one that has no parameters). The whole goal of this top module is to generate a sequence of inputs for the module under test. The testbench sends these inputs to the module and observes the module's outputs, verifying that the module produces correct results.

For combinational logic, testing every possible input combination is sometimes possible. When this is unfeasable, a set of common use cases and tougher corner cases usually provides a fairly thorough testing suite. Tests for these modules are usually simple unit tests – for each input, there is an immediate output that can be checked.

For more complex sequential circuits, including FSMs and processors, simply generating inputs is not enough to ensure a thorough test. In addition to unit tests, sequential circuits generally require functional tests that mimic how the circuit would be used in practice. For example, a good testing suite for a soda-machine dispenser design, such as the example RTL design in chapter five, would walk the machine through a series of coin-entry sequences with

different soda costs, ensuring that the machine correctly dispenses a can when the correct amount has been inserted.

The following subsections discuss several important components that can be used within a top module to construct a testbench.

The `initial` Procedure

There is a special, unsynthesizable procedure – the `initial` procedure – that is extrememly useful in testbenches. True to its name, the `initial` procedure runs immediately once simulation starts – and it only runs once. It's okay to have several `initial` procedures in the same top module – they'll just run in parallel, all starting at the beginning of simulation.

Basically, all of the code in a testbench aside from clock generation and variable declarations should be inside an `initial` procedure. The syntax for an initial procedure is easy:

```
1 initial begin
2   // Code Here to Generate Test Cases
3 end
```

All assignments in the `initial` procedure should be done with blocking assignment. **Do not use non-blocking (<=) assignment inside an initial block**. Also, only assign to registers.

For example:

```
1 reg x;
2
3 initial begin
4   x = 1; // NOT x <= 1;
5 end
```

Delays

Inside the `initial` procedure, we want to generate a few test cases by specifying inputs for the module and then checking outputs. However, we usually should wait for a bit between setting the inputs and checking the outputs, as results will take a bit of time to propagate through a circuit. Additionally, we want to specify some time gap between tests. The method Verilog uses to do this is *delays*.

The syntax for a delay is simple – just use a hashtag and a number. A delay statement by itself or to the left side of another statement will delay the continued exection of an initial block until the specified amount of time has passed.

For example:

```
reg x;

initial begin
  x = 0;     // Occurs at time 0
  #5;
  x = 1;     // Occurs at time 5
  #10 x = 1; // Occurs at time 15
end
```

Use delays to separate checking outputs from assigning inputs and to separate test cases from each other.

The units of delays can be set using the `timescale` directive. This directive takes the following form:

```
`timescale 10ns/10ps
```

The first unit represents the time scale. In order to get the real-time delay for a delay statement, simply multiply the number after the hashtag by the timescale. Thus, with the above timescale, #5 is a 50ns delay. The second number is the precision. Times are rounded to the nearest time unit represented by the precision. Thus, in the above timescale, we can use a delay of #1.001, but not #1.0001. Finally, both the precision and time scale must be either 1, 10, or 100 in whatever unit you use.

Please note that you do not have to define a timescale. Your Verilog compiler will define a default timescale that is usually pretty sane, so don't worry too much about it.

Clocks

Often, modules will require a clock as an input. One could manually include statements cycling the clock inside the `initial` block, but this tends to get very clunky. Instead, use an `always` procedure with no sensitivity list. This type of procedure will run continuously forever, and by including delays, one can create a periodic clock:

```
reg clk = 1;

always begin
  #5 clk = ~clk;
end
```

This clock will alternate every 5 time units and start off with a value of 1.

Inside the `initial` block, if you want to synchronize with the clock, just drop an `@(posedge clk);`. This will pause execution of the `initial` procedure and wait until a positive clock edge comes to proceed.

System Tasks and Functions

There are a number of specialized functions that are available for use in testbenches. These constructs are totally unsynthesizable, so do not attempt to use them outside of a testbench.

Dumping a VCD File for Waveform Viewers

Waveform viewers are indispensable when it comes to debugging, and it's useful to dump a file that contains waveforms for all signals in your design during your simulation. To do this, use the `$dumpfile` and `$dumpvars` functions inside an `initial` procedure:

```
initial begin
  $dumpfile("myVerilogTest.vcd");
  $dumpvars;
end
```

This will create the file `myVerilogTest.vcd`, which is a special waveform file type, and fill it with the waveform values for all signals in your module hierarchy. Later, you can open this type of file with GTKWave or a similar wave viewer for debugging.

Unfortunately, `$dumpvars` won't capture arrays (but it'll always capture multi-bit vectors – you should only be using arrays for register files or memories anyway!). You have to manually dump arrays using a for loop if you'd like to view them in a wave viewer. Let's say that you have a module, `MyModule`, that has a 4-bit wide, 16-element array named `arr`. To dump this array, use this syntax:

```
module testbench;
  MyModule m(...);

  integer idx;
  initial begin
    $dumpfile("myVerilogTest.vcd");
    $dumpvars;
    for (idx = 0; idx < 16; idx = idx + 1) begin
      $dumpvars(0, m.arr[idx]) // Use $dumpvars(0, varname) to dump
    end
  end
endmodule
```

Note that you can reach into the hierarchy to extract signals using the dot operator as shown above.

Printing Text

There are two functions for printing text out to the terminal. These are `$display` and `$write`, each of which work just like C's `printf` statement. The only difference is that `$display` adds a newline after the string it prints, while `$write` does not.

In general, simply call the function with a format string followed by any specific arguments you want to be incorporated into that string. For those unfamiliar with C's `printf` statement, the "%" sign is a special character inside the format string. The next character after a "%" indicates a type of variable, and when the format string is printed, any "%" symbols and their types will be replaced with the appropriate variable from the argument list in the order of appearance.

For example, "%d" indicates a decimal value. Thus, if the value of `wire x` is 5, then the following code block will print "The value is: 5".

```
1 $display("The value is: %d", x);
```

Here's a list of all format characters available in Verilog:

Character	Meaning
c	ASCII character
d	Decimal value
h	Hexadecimal value
o	Octal value
b	Binary value
t	Time value
s	String value
u	Unformatted Binary (0,1)
z	Unformatted Binary (0,1,Z,X)

Simulation Time

To extract the current time in a simulation, simply use `$time`. For example, to print the current simulation time, use the following code:

```
1 $display("The time is: %t", $time);
```

Ending a Simulation

To end a simulation, simply add the `$finish` directive to the end of your simulation's `initial` procedure.

```
module Testbench;

  initial begin
    $dumpvars;
    ...
    $finish;
  end
endmodule
```

If you fail to add the finish directive, your simulation will not exit!

Others

There are several other useful system functions that can manipulate files and read hex data into memories. If you'd like to learn more about these, simply refer to Vahid's *Verilog for Digital Design*, or the numerous online resources listing all of Verilog's system functions. Once again, be careful to only use them in testing!

7.2 Important Note on Equality Checking

In your testbenches, you will frequently find yourself checking for equality or inequality. For example, if you expect your module under test to be outputting the value 1, you might write this code:

```
if (out != 1) begin
  // error condition
end
```

However, if `out` is unknown (Z), then this test will actually pass! This is because the simple equality or inequality operators will return "Z" if the relation they're testing is ambiguous due to "Z" or "X" values in the operands. This could cause the error condition to fail to run!

For testbenches, **and ONLY for testbenches**, use the special `!==` and `===` operators instead. These constructs are totally unsynthesizable, so **do not** use them in code you write outside of testbenches. However, they are useful for testing because they will check for a bitwise exact match – including X and Z values! Therefore, the following code will correctly execute the error condition when `out` is "X":

```
if (out !== 1) begin
  // error condition
end
```

7.3 Macros

Testbenches will often contain very repetitive code segments, and it is often useful to factor some of this code out into an easily-reusable structure. In Verilog, the best way to do this is usually a macro "function", or a parameterized macro.

For those familiar with C's macro "functions", Verilog's work in much the same way. If you're not familiar, here's a rundown of how they work.

Macros work via textual substitution. In Verilog, a macro has three parts:

1. The `define` statement that declares a macro.
2. The macro name, plus any parameters for the macro.
3. Finally, the substitution value of the macro.

You've seen simple macros with no parameters before. For example, the following code block defines `STATE_CONST` to be equal to `15`. When the Verilog compiler runs, it will replace any uses of `` `STATE_CONST `` with `15` via textual substitution.

```
`define STATE_CONST 15
```

Having parameters for macros can often be useful in testbenches. For example, you'll frequently find yourself checking if a certain variable holds a particular value. If it doesn't, you'll want to print an error message. In order to save some typing and make your testbench more readable, it might be easiest to factor this behavior out into a parameterized macro:

```
`define ASSERT_EQ(ONE, TWO, MSG)       \
  begin                                \
    if ((ONE) !== (TWO)) begin         \
      $display("Error: %s", (MSG));    \
    end                                \
  end #0
```

Then, you can use the macro like this:

```
wire [3:0] signal;

initial begin
  // Setting some signals

  `ASSERT_EQ(signal, 4'b0010, "Signal should be 4'b0010");

  // More tests

  $finish;
end
```

A parameterized macro will expand via textual substitution, replacing uses of its parameters with the literal value passed in via the call. For example, the above call will generate this code:

```
wire [3:0] signal;

initial begin

  begin
    if ((signal) !== (4'b0010)) begin
      $display("Error: %s", ("Signal should be 4'b0010"));
    end
  end #0;

  $finish;
end
```

This example macro demonstrates a few good practices that should be followed when writing parametrized macros:

- The macro name and parameter names should be in all caps.
- Wrap usages of the parameters in parentheses, and only use them once in the macro body! This is really important – because the macro works via textual substitution, if you use one of the parameters twice and that parameter has a side effect, the side effect will be executed twice!
- Wrap the body of the macro with a `begin` statement and an `end #0` statement. The `begin` and `end` help to unify the block of statements inside the macro as one cohesive unit, and the extra `#0` at the end allows you to terminate your call to the macro with a semicolon without generating a compiler error.

- Finally, macros may be spread across several lines by terminating each line with a backslash. Make sure every line in the macro (except for the last one) ends with a backslash!

7.4 Example Testbench

In this section, we present an example testbench for the soda machine we designed in the earlier section on RTL design.

As a quick reminder, here's the soda machine's module:

```
module SodaDispenser(
  input clk,     // Clock
  input rst,     // Reset
  input [7:0] s, // Cost of a Soda
  input [7:0] a, // Value of inserted coin
  input c,       // 1 for one cycle when coin inserted
  output d       // 1 for one cycle when soda dispensed
);
```

We want to run this machine through a variety of tests with different soda and coin values. Here, we present a testbench that serves as a starting point. It performs a simple functional test, ensuring that the machine dispenses a single soda after 50 cents are inserted. Full testbenches should have many more cases, but this serves as a simple, working example.

```
module SodaTestbench;
  reg clk = 1;
  reg rst, c;
  reg [7:0] s, a;
  wire d;

  // Soda Dispenser Module (under test)
  SodaDispenser sd(
    .clk (clk),
    .rst (rst),
    .s   (s),
    .a   (a),
    .c   (c),
    .d   (d)
  );

  // Dump Variables to VCD Viewer
  initial begin
    $dumpfile("SodaTestbench.vcd");
    $dumpvars;
  end
```

```
22
23  // Generate Clock
24  always begin
25    #5 clk = ~clk;
26  end
27
28  // Main Test Logic
29  initial begin
30    //--------------------
31    // Set Initial Values
32    //--------------------
33    rst = 1;
34    c = 0;
35    a = 0;
36    s = 50;
37
38    @(posedge clk)
39    rst = 0;
40
41    //--------------------
42    // Ensure D is 0
43    //--------------------
44    if (d !== 0) begin
45      $display("D not zero after reset!");
46    end
47
48    #10
49
50    //--------------------
51    // Insert First Coin
52    //--------------------
53    @(posedge clk);
54    a = 25; // WAIT STATE
55    c = 1;
56    @(posedge clk);
57    c = 0;  // ADD STATE
58    @(posedge clk);
59      // WAIT STATE
60    @(posedge clk);
61      // WAIT STATE
62    #1
63
64    //--------------------
65    // Ensure D is 0
66    //--------------------
67    if (d !== 0) begin
68      $display("D was one before enough money inserted!");
69    end
70
```

```
71      #10
72
73      //--------------------
74      // Insert Final Coin
75      //--------------------
76      @(posedge clk);
77      a = 25; // WAIT STATE
78      c = 1;
79      @(posedge clk);
80      c = 0;  // ADD STATE
81      @(posedge clk);
82              // WAIT STATE
83      @(posedge clk);
84              // DISP STATE
85      #1
86
87      //--------------------
88      // Ensure D is 1
89      //--------------------
90      if (d !== 1) begin
91        $display("D was not 1 after final coin inserted");
92      end
93
94      @(posedge clk);
95      #1;
96
97      //--------------------
98      // Ensure D is 0
99      //--------------------
100     if (d !== 0) begin
101       $display("D did not return to 0");
102     end
103
104     $finish;
105   end
106 endmodule
```

7.5 Chapter 7 Review Questions

7.1 For software developers, testing is an important part of the development process, but even if bugs are later found in the software, developers are able to release an update or patch. Is this the same case for hardware developers? How could not rigorously testing a Verilog circuit design be financially costly, especially for hardware developers like *Intel*?

7.2 Read and understand the following Verilog code:

```
1 `timescale 10ns/10ps
```

a) How do you specify a delay in Verilog?

b) What is the `timescale` directive used for? In the above example, what is the time scale and what is the precision?

c) Why are delays useful in Verilog testbenches?

7.3 Assume you are asked to design a circuit with a clock period of 20ns.

a) What time scale would you specify for the circuit?

b) In your testbench you would need to cycle a clock. Write the code that you would use to do this. How did you determine the delay?

7.4 Read and understand the following Verilog code:

```
1 reg[3:0] x = 4'b0001;
2 reg[3:0] y = 4'b1111;
3
4 $display("Test Number %d Completed. Result is %h.", x, y);
```

a) What would be the final text printed to the terminal?

b) How would you change the code to print the test number as an octal value?

7.5 How do you properly end a simulation? How does the simulation behave if you fail to add this directive?

7.6 Why should you use `!==` and `===` when checking equality in testbenches?

7.7 Read and understand the following Verilog code:

```
1 \\ An example macro
2 `define ASSERT_EQ(ONE, TWO, MSG)       \
3   begin                                 \
4     if ((ONE) !== (TWO)) begin          \
5       $display("Error: %s", (MSG));     \
6     end                                 \
7   end #0
```

a) How can macros be useful in testbenches?

b) How could you use the above macro to check if the wire `signal` carried the expected value 14?

7.8 What does the statement `@(posedge clk)` do? Why is this useful in a testbench?