Verilog Overview

The Verilog Hardware Description Language

Things to Look For…

- The structure of a Verilog program.
- How to develop and use Verilog modules.
- The differences between gate-level, dataflow, and behavioral level models.
- How to develop combinational and sequential circuit models at all three levels.
- The types of assignment at each level of modeling.
- How to specify and model real world delays in a circuit.
- How to monitor and display dynamic circuit behavior.
- How to build a tester and test bench to evaluate a model.
A.0 Introduction

The circuits and systems that we are developing today are growing in capability and complexity every day. Yesterday, a sketch on a piece of paper and a handful of parts were sufficient to try out a design idea. Today, that is no longer possible. Today, the idea is modeled using computer based tools and languages then frequently synthesized into the desired hardware implementation. We use two key words here, model and synthesize.

We first model the design, iterating until we are satisfied then transform that design into an FPGA or ASIC. There are a number of languages that permit such a design approach. Verilog and VHDL are two of the more common. SystemC for modeling both the hardware and software components is finding its way into an increasing number of designs in the embedded world. In this text, we will use Verilog.

Verilog is a hardware design language that provides a means of specifying a digital system at a wide range of levels of abstraction. The language supports the early conceptual stages of design with its behavioral level of abstraction and later implementation stages with its structural level of abstraction. The language provides hierarchical constructs that allows the designer to efficiently and effectively manage the complexity of contemporary designs.

This appendix will introduce the Verilog language and present the important features and capabilities that are used in this book. It does not purport to be a comprehensive study of the language. In this appendix, we will begin with the basic components and organization of a Verilog program; we will then examine the gate-level or structural, dataflow, and behavioral models for combinational logic circuits, and then follow with similar models for sequential circuits.

Design is only one element of the product development, each design must also be tested to confirm that it meets specified requirements and the objectives of the modeling process. To that end, each section will also discuss how one can formulate test suites to verify the intended operation. The material on testing will lay the foundation to enable the developer to build test cases that will support testing to the desired level. It’s beyond the scope of this text to present a comprehensive treatise on testing.
A.1 An Overview of a Verilog Program

The structure of a Verilog program replicates the traditional way of designing, testing and debugging a module, subsystem or system. As engineers, we design a circuit, we build the circuit, and we take the circuit to our bench where we test it. On our electronics test bench, we have test equipment. Such equipment consists of stimulus instruments, switches, function generators, sophisticated data or pattern generators. We also have measurement equipment such as voltmeters – DVM or DMM, oscilloscopes, logic analyzers, and network analyzers. The circuit consists of electronic parts and wires. We build the modules by interconnecting the wires and electronic parts. Once the circuit is built, we connect the test equipment to the circuit with wires.

We now have a picture that looks like that in Figure A.0.

![Figure A.0: The Structure of an Electronic Test System](image)

The circuit or module is called the unit or device under test – UUT or DUT. A Verilog source program follows this model. It comprises 3 major elements

- A test bench
- A collection of stimulus and measurement modules
- A circuit or system that is being modeled

That circuit or system is made up of a number of logical components. A logical component may be an atomic device such as a logic gate or a number of components or modules. Each may, in turn, may also be made up of other logical devices. The stimulus module provides signals into the UUT; the measurement module acquires / measures the corresponding outputs of the UUT. A Verilog program, very much like the physical electronics bench, has stimulus and measurement...
equipment connected to the circuit or system being modeled. As is done on a physical bench, equipment is connected to the UUT using wires. On occasion in a Verilog program the stimulus and measurement equipment may be in the same software module.

A.2 Creating a Verilog Program

In order to perform a digital circuit simulation using Verilog we need to

- Create a Verilog source file using a text editor
- Synthesize and simulate the source file
- Debug if necessary by looking at the simulation output

The goal in writing a traditional program in a language such as C, C++, or Java is to implement and run an application. In this case, the program is the final deliverable. In contrast, a Verilog program has an initial purpose of modeling a circuit design. Once the performance of the design is satisfactory, the program is used to synthesize a hardware circuit. The HDL program is an intermediate step.

A.2.1 Some Concepts in a Verilog Source File

Before continuing, there are several important points that one should learn about the language.

Case-sensitivity
Like C, C++, or Java, Verilog is case sensitive. If an error is encountered while compiling a Verilog source file, look for case-errors.

Identifier Names
The rules for identifier names are similar to those found in C, C++, or Java. An identifier name may contain any digit or letter as well as the underscore and $ character. The first character must be a letter and the identifier cannot be a Verilog keyword.

Annotation
The Verilog language supports both the C style multiple line comment and the C++ style single line comment. The paired /* */ symbols state that all text inside the delimiter is to be interpreted as a comment. The symbol // specifies that all text on a line after the symbol is to be viewed as a comment.
**White Space**

The white space characters ‘space’ and ‘tab’ are ignored by the Verilog compiler.

**Block Delimiters**

A *sequential block* comprises one or more statements that are intended to act together. In C, C++, or Java, a block of statements is delimited by a pair of curly braces, ‘{}’. The Verilog language uses the Pascal style statements `begin` and `end`. The statements in a sequential block are executed in the order that they are specified.

**A.2.2 Modules**

The *module* is basic building block of Verilog. One can think of a module as a *black box*. To make a system consisting of modules, we link up the individual *black boxes* with *wires*. One can also think of the modules as loosely analogous to the struct in C or as a class in C++ or Java. Bear in mind that modules are *not* classes.

Like each of these data types, the module represents a user defined type. Once defined, instances of a module can be declared in the same manner as any of the intrinsic types. The concept of module permits one to build complex systems by composing or aggregating lower-level components. Like the C struct or the C++ or Java class, each module expresses a distinct local scope. All variables declared and defined therein are visible only in that scope.

The module provides a structure for the design process. As the number of modules that are defined increases, the more complex the design becomes. In such cases, it is convenient to be able to verify functionality module by module.
The code fragment in Figure A.1 gives the general syntax and structure for a Verilog module.

```verilog
module moduleName(outputsList, inputsList);
  outputs       // outputs from the module
  inputs        // inputs to the module
  reg           // local storage in the module
  wire          // conduction paths in the module
  initial       // initialize variables in block
    block
  always        // always execute statements in block
    block
  code          // your code
  ...
  ...
endmodule
```

Figure A.1
The Structure of a Verilog Module

Observe first that each line in a Verilog module, except the last, must be terminated with a semi-colon. In C, C++, or Java, the scope of a block is delimited by curly brackets, \{ \}. In Verilog, the scope of a module is delimited by the keywords `module` and `endmodule`.

**Module Name**

Analogous to a struct or class declaration, the declaration of a Verilog module begins with the keyword `module` as illustrated in the opening line of the code fragment.

```
module moduleName(outputsList, inputsList);
```

Following the module keyword is the name of the module. The module name can now be used as a type specifier.

**Coding Style:**

Always try to select a module name that conveys the purpose or function of the module.
Inputs and Outputs Declarations

Following the module name, enclosed in parentheses, are the inputs and outputs to the module. The inputsList and the outputsList are optional; however, they will be used most of the time. The inputs and outputs can be specified in any order, even commingled (not a good idea, though).

Consequently, when the module is declared, each item in the inputsList and the outputsList must be identified in an input or output declaration as follows.

Coding Style:
The standard convention in Verilog is the outputsList comes before the inputsList. Always try to select input and output names that convey the meaning of the variables

Syntax
output outputsList;
input inputsList

Nets and Variables

The Verilog language defines several different kinds of nets and variables. The net represents a class of primitive data types that are used to model a node or electrical connection in a circuit. A net cannot be assigned to; it cannot hold a value. The value on a net results from being continuously driven by the output of some logical device. If a net is not driven, it takes on the default value of ‘z’ meaning high impedance or floating. A Verilog variable, like a variable in C, C++, or Java, can be assigned a value and will hold that value until a subsequent assignment replaces the value.

A wire type is a kind of net and like real world wires is used to connect the output of one logic element to the input(s) of other logical elements. Because it is a net, the value of a wire can only be changed as the result of a gate or a behavioral statement driving it.

A reg is a kind of variable. The value of a reg or register can be changed directly by an assignment. One should not confuse the Verilog reg with the hardware register. The reg is simply an entity that can hold a value. The default value of a reg data type is ‘x’, or unknown.
The syntax for the *reg* and *wire* declarations is given as,

```
Syntax
reg regList;
wire wireList;
```

### Declaring Multi-Bit Signals

Often it’s necessary to represent multi-bit wires, for example, a 3-bit wire that can carry digital signals representing the values 0..7. The types *reg* and *wire* can also be formed into a bus such as:

```
Syntax

Big Endian
reg [msb:lsb] regList
wire [msb:lsb] wireList;

Little Endian
reg [lsb:msb] regList
wire [lsb:msb] wireList;
```

Where *msb* is the bit index of the most significant bit and *lsb* is the bit index of the least significant bit. The value of the *lsb* index must be zero since bit position 0 conventionally denotes the least-significant bit. Such statements configure a set of individual wires so that they can now be treated as a group for example,

```
wire [2:0] myWires; // a 3-bit signal (a bus)
reg [15:0] aState; // a 16-bit state holding value
```

The declaration, *myWires*, in Figure A.2 declares a 3-bit signal that has

- MSB (the 2^2’s place) as myWires[2]
- Middle bit of myWires[1].
- LSB (the 2^0’s place) as myWires[0]
The individual signals can be used just like any other binary value in Verilog. For example, we could declare,

```verilog
and a1(myWires[2], myWires[0], C);
```

This statement AND’s together C and the LSB of `myWires` and puts the result in the MSB of `myWires`.

This bus specification can be extended to input and output lists as well; that is, multi-bit signals can also be passed together to a module:

```verilog
module random(bus1, bus2);
    output [31:0] bus1;
    input [19:0] bus2;
    wire c;
    anotherRandom ar1(C, bus2, bus1);
endmodule
```

### Subsets of Multi-Bit Expressions

On occasion, it’s necessary to break apart multi-bit values. We can do that by selecting a subset of a value. For example if we have

```verilog
wire [31:0] myWires;
initial myWires[3:1] = 'b101;
```

This would set

- `myWires[3] = 1`
- `myWires[2] = 0`
myWires[1] = 1

All other bits of myWires will not be altered. One can also use the same form to take a subset of a multi-bit wire and pass it as an input to another module.

$display and $monitor statements

The $display and $monitor are standard system tasks that enable one to see the states of certain signals, in text form. The output is typically directed to the screen (or window). The difference between the two statements is that $display is only evaluated when the directive is encountered during execution. The $monitor statement is evaluated every time any of the signals that is being monitored changes state.

The syntax for the two directives is given as,

```
Syntax
$display ("formatString", variableList);
$monitor ("formatString", variableList);
```

The formatString is optional for both statements; both follow the C printf statement syntax. The formatString is a text string containing format variables that are to be instantiated, one-to-one, from the values specified in the variableList.

The more commonly used format variables are given in Table A.0.

By convention, a logic high is denoted as a 1 and a logic low is denoted as a 0. An unknown state is denoted as an x. The $display and $monitor output statements must be placed within an initial or always block.

$stop and $finish Statements

The $stop and $finish statements are system tasks that used to either stop or finish a simulation. The former directs the simulation to the interactive mode and the latter terminates the simulation.
The `$stop` is used when the designer wishes to suspend the simulation prior to exit to examine the state of signal values.

The syntax for the two directives is given as,

```
Syntax
$stop;
$finish;
```

### $time Statement

```
Syntax
$time;
```

The `$time` statement is a system function that returns the current time. The syntax is given as,

The statement can be included in a `$display` or `$monitor` statement as,

```
$display ($time, ["formatString"], variableList);
$monitor ($time, ["formatString"], variableList);
```

### A.3 Three Models – The Gate-Level, the Dataflow, and the Behavioral

With this brief introduction to some of the elements of the Verilog language, we will next look at the how the language supports the modeling process. The Verilog language supports the development of models at three different primary levels of abstraction. The gate level model gives the most detailed expression and the behavioral level the most abstract. At the gate level, modules are implemented by interconnecting the various logic gates much as one would do when working with SSI and MSI components. This is also known as a structural model. At the dataflow level, the module is implemented by specifying the movement of the data amongst the comprising hardware registers. The dataflow model is analogous to the RTL (Register Transfer Level) level used in specifying a microprocessor architecture. At the behavioral level, modeling is based upon an algorithmic description of the problem without regard for the underlying hardware.

The language does support modeling at the transistor level. However, work at that level will not be discussed in this text.
We will begin at the gate level and work up. The path that we will follow will be to use the three different levels at which the modeling process may be conducted as a means to introduce the core aspects of the language. Following the discussion of the different approaches, we will bring everything together with a discussion of developing a test module then coupling the test module with the UUT in a test bench. Because working at the gate level is the most familiar to many engineers, we will begin at that level then move up to higher levels of abstraction.

We will utilize the same combinational and sequential designs to illustrate how a model is developed at each of the different levels. The combinational circuits will be a logic block using an AND and an OR gate which are extended to implement a NAND and a NOR circuit. The sequential circuits will progress from a basic latch, to a gated latch, to a flip-flop and ultimately, a two bit binary counter.

A.3.1 The Structural / Gate Level Model

As the name suggests, at the gate level, we are working with the basic logic gates and flip-flops that one finds in any detailed digital logic diagram. These devices model the behavior of the parts that we can buy from any electronics store or that we might design into an ASIC or use in FPGA. Verilog supports the logic gates identified in Figure A.6 as predefined intrinsic modules.

The prototypes for each of the gates are given Figure A.7.
The device prototypes appear very much like those for a C or C++ function or procedure. The<br>&lt;name&gt; for a gate instance must begin with a letter and thereafter can be any combination of let-ters, numbers, the underscore ‘_’, or the ‘$’. Gates with more than two inputs are created by sim-ply including additional inputs in the declaration. Observe that the output list appears first<br>followed by the input list.

**Example A.0**

A five-input and gate is declared as

```verilog
and <name> (OUT, IN1, IN2, IN3, IN4, IN5); // 5-input AND
```

### A.3.1.1 Creating Modules

At the gate level, whether one is building a combinational or sequential logic circuit, a Verilog<br>module really is a collection of logic gates. Each time we declare and define a module, we are cre-ating that set of gates. We will look first at combinational logic models then follow with sequen-tial circuits.

**Combinational Logic**

The structural or gate level model of a combinational circuit reflects the physical gates used to<br>implement the design. To illustrate the basic process of creating a Verilog program and modeling<br>combinational logic at the gate level, we will begin with the following simple circuit.

An example of a simple module begins with the logic diagram in Figure A.8; the module requires<br>a name; we’ll call it *AndOr*.

```
// Compute the logical AND and OR of inputs A and B.
module AndOr(AandB, AorB, A, B);
    output AandB, AorB;
    input A, B;
    and myAnd (AandB, A, B);
    or myOr (AorB, A, B);
endmodule
```

**Figure A.8**

A Combinational Logic Circuit with Corresponding Structural Verilog Module
We can analyze the module line by line.

```verilog
module AndOr(AandB, AorB, A, B);
    output AandB, AorB;
    input A, B;
endmodule
```

The first line is a comment designated by the `//`. Everything on a line after a `//` is ignored. Comments can appear on separate lines or at the end of lines of code.

The top of a module begins with the keyword `module` indicating start of module, the name of the module, `AndOr`, and a list of signals connected to that module. Subsequent lines first declare that the first two binary values generated by this module are `outputs` from the module and the next two (A, B) are `inputs` to the module.

The next lines

```verilog
    and myAnd (AandB, A, B);
    or myOr (AorB, A, B);
```

Create instances of two gates: an AND gate called `myAnd` with output `AandB` and inputs `A` and `B` and an OR gate called `myOr` with output `orOut` and inputs `A` and `B`.

We declare such intrinsic components the same as we did in C, C++ or Java with int, float, or char.

The final line declares the end of the module.

```verilog
endmodule
```

All modules must end with an `endmodule` statement. Observe that the `endmodule` statement is the only one that is not terminated by a semicolon.
A.3.1.2 Using Modules

We build up a complex traditional software program by having procedures call sub procedures or by composing or aggregating classes into larger and more powerful class structures. Verilog builds up complex circuits and systems from modules using a design approach similar to composition or aggregation.

To illustrate the process, we will use the previous AndOr module to build a NandNor circuit. We begin with the logic diagram and Verilog module in Figure A.9.

The NandNor module declares an instance of the AndOr module as it would any of the intrinsic types. One can declare multiple instances of a submodule. Another instance of the AndOr module could be added to the NandNor module. Each instance of the submodule creates a new set of gates. Three instance of AndOr would create a total of $2 \times 3 = 6$ gates.

The wire statement is used to connect the outputs of the AndOr module to the two not gates. These wires comprise a net that carries the signals from the output of the AndOr module to the inverters.
A.3.1.3 Delays

In perfect world, parts are ideal and signals flow through wires and parts with no delay. In the real world, parts are not perfect. Signals are delayed by varying amounts. In Verilog, we can model how long signals take to propagate through the basic gates in a circuit using the `#` operator. The basic syntax is given as,

```
Syntax
#delay device;
```

We modify the AndOr module in Figure A.10 to incorporate delays into the design to model real world behavior.

```
// Compute the logical AND and OR of inputs A and B.
module AndOr(AandB, AorB, A, B);
    output AandB, AorB;
    input A, B;
    
    and #5 myAnd (AandB, A, B);
    or #10 myOr (AorB, A, B);
endmodule
```

The line

```
and #5 myAnd (AandB, A, B);
```

states that the AND gate takes 5 time units to propagate a change on the input to the output. While the OR gate is twice as slow, taking 10 time units.

```
or #10 myOr (AorB, A, B);
```

Note that the units of time can be whatever we want as long as we use consistent values.
In the perfect world, logic devices change state in zero time. In the real world we rarely encounter such ability. To support modeling the time required for a signal to rise or fall, Verilog also supports including device rise time, fall time. The syntax for all three is given as,

```
# (rise time, fall time, delay) device;
```

### A.3.1.4 Defining Constants

While one can use what are called *magic numbers*, a more robust design will use named or symbolic constants; variables whose value is set in one place then used throughout a piece of code. The symbolic constant in Verilog is called a *parameter*. A parameter is defined and initialized using the following syntax.

```
parameter = aValue;
```

The following code fragment illustrates the inclusion of a delay of 2 time units in a part model.

```
parameter propagationDelay = 2;
not #propagationDelay myNot(sigOut, sigIn);
```

Let’s modify the previous example to that in Figure A.11 to reflect more professional approach and also incorporate the signal rise and fall times.

```
// Compute the logical AND and OR of inputs A and B.
module AndOr(AandB, AorB, A, B);
    output AandB, AorB;
    input A, B;

    parameter delay0 = 5;
    parameter delay1 = 10;
    parameter riseTime = 3;
    parameter fallTime = 4;

    and #(riseTime, fallTime, delay0) myAnd (AandB, A, B);
    or #(riseTime, fallTime, delay1) myOr (AorB, A, B);
endmodule
```

Figure A.11
Modeling Gate Delays - Second Attempt
The modified code sets the delay of the gates to delay0 and delay1 respectively and the rise and fall times to the values specified by the remaining two parameters. To speed up either gate, one could simply change the value in the parameter lines to the desired values.

**Sequential Logic**

Sequential logic is modeled at the gate level by first developing the appropriate flip-flop module then implementing the design as a composition of instances of that module, the necessary gates, and interconnecting the components with wires. To illustrate the process we begin with the basic SR latch which is given in the logic diagram and Verilog code fragment in Figure A.12.

```verilog
// Gate Level Model   S R Latch
module srLatch(q, qnot, s, r);
    input s, r;
    output q, qnot;
    parameter delay0 = 2;

    // implement the latch
    nor #delay0 n0(q, r, qnot);
    nor #delay0 n1(qnot, s, q);
endmodule
```

**Figure A.12**

Defining an S R Latch with Corresponding Structural Verilog Module
The basic design can be extended to include an enable as an additional level of control. The logic diagram and Verilog implementation, using the *srLatch* are given in Figure A.13.

![Figure A.13](image)

Extending the S R Latch with Corresponding Structural Verilog Module

The master slave implementation using the gated latch follows in Figure A.14.

![Figure A.14](image)

The Master-Slave S R Latch with Corresponding Structural Verilog Module
We can now use the SR flip-flop to build a simple two bit synchronous binary up counter. The logic diagram and Verilog model follow in Figure A.15.

A.3.2 The Dataflow Model

Gate level modeling is an effective approach for working with smaller problems. Such an approach directly follows the typical detailed logic diagram and thus simplifies moving from design to model and simulation. Today, embedded applications are continually increasing in complexity. SSI and MSI modules of yesterday are being replaced by ASICs, FPGAs, and microprocessors. Developing a complete design at the gate level is no longer feasible. Working at the gate level today is similar to trying to write sophisticated application in assembler. While it can be done, such an approach is not practical.

Developing at a higher level is not without problems however. The farther that one moves away from the low level details and increases reliance on tools to produce those details the greater the risk that the tools will produce a less than optimum design. The ability to push the limits of a design and a technology comes from years of experience and understanding of the problem. Tools can help us to solve the majority of the design problems. They are not sufficiently advanced to solve all autonomously.
Dataflow modeling, as the name implies, views a design from the perspective of data moving through the system from source to destinations. In the digital world, such a view is often referred to as RTL or register transfer level design. Contemporary tools are able to accept a dataflow model as input and produce a low level logic gate implementation through a process called logic synthesis.

A.3.2.1 Continuous Assignment

At the dataflow level, the design is modeled as the movement of data from module to module in order to affect the application. That data moves over a net. Thus, a fundamental element of such modeling is the ability to drive a value from a source module onto the interconnecting net to the destination modules. In Verilog, such ability is expressed by continuous assignment. The continuous assignment statement is specified using the following syntax.

```
Syntax
    assign destination net = source net expression
```

The left hand side of the continuous assignment must be either a scalar or vector (multiple lines) net. The right hand side of the expression can be a net, register, or function call return and must be of the same size as the left hand side. A scalar cannot be assigned to a vector and vice versa, for example.

A continuous assignment is always active. A change on the right hand side forces evaluation of the left hand side with the resulting assignment of the right hand side value to the left hand side net.
Combinational Logic

We illustrate a combinational dataflow model using the AndOr circuit designed earlier. That model, using the continuous assignment, is expressed in the following code fragment.

![Verilog code for AndOr module]

The implementation of the function using the bitwise AND and OR operators should be familiar from earlier work with their C counterparts.

A.3.2.2 Delays

Moving up one level of abstraction from the gate level does not preclude the need to model real world effects on circuit behavior. The Verilog model for delay at the dataflow level follows naturally from that at the gate level.

The syntax is given as,
Example A.1

The model for the AndOr circuit designed earlier can include delays as seen in the following code fragment.

```verbatim
// continuous assignment
module AndOr(AandB, AorB, A, B);
    output AandB, AorB;
    input A, B;

    wire AandB, AorB;
    parameter delay0 = 10;

    assign #delay0 AandB = A&B;
    assign #delay0 AorB = A|B;
endmodule
```

The outputs of the system will now change 10 time units after either of the input signals changes as illustrated in Figure A.18.

<table>
<thead>
<tr>
<th>Time,</th>
<th>A, B,</th>
<th>AandB, AorB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 1</td>
<td>x, x</td>
</tr>
<tr>
<td>10</td>
<td>0, 1</td>
<td>1, 1</td>
</tr>
<tr>
<td>20</td>
<td>0, 0</td>
<td>0, 1</td>
</tr>
<tr>
<td>30</td>
<td>0, 1</td>
<td>0, 0</td>
</tr>
<tr>
<td>40</td>
<td>0, 1</td>
<td>0, 1</td>
</tr>
</tbody>
</table>

Rise and fall time delays are incorporated in a similar manner. The syntax for all three is given as,

```verbatim
Syntax
    assign # (rise time, fall time, delay) net;
```
Example A.2
The model for the AndOr circuit designed earlier can include all three delays as seen in the following code fragment in Figure A.19.

```verilog
// Compute the logical AND and OR of inputs A and B.
module AndOr(AandB, AorB, A, B);
  output AandB, AorB;
  input A, B;

  wire AandB, AorB;
  parameter delay0 = 10;
  parameter rise = 5;
  parameter fall = 7;

  assign #(rise, fall, delay0) AandB = A&B;
  assign #(rise, fall, delay0) AorB = A|B;
endmodule
```

The outputs of the system will now change 10 time units after either of the input signals changes and reflect the rise and fall times as well as illustrated in Figure A.20.

<table>
<thead>
<tr>
<th>Time,</th>
<th>A, B</th>
<th>AandB,</th>
<th>AorB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 1</td>
<td>x,</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>1, 1</td>
<td>1,</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0, 1</td>
<td>1,</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>0, 1</td>
<td>0,</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>0, 0</td>
<td>0,</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>0, 1</td>
<td>0,</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure A.20
System Output from the Circuit Module in Figure A.19
A.3.2.3 Operators

The syntax and operators used in Verilog at the dataflow level follow that of the C language very closely. Table A.1 gives the most commonly used operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>Divide</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>Modulus</td>
</tr>
<tr>
<td>Relational</td>
<td>&gt;</td>
<td>Greater Than</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>Less Than</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>Equal</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>Not Equal</td>
</tr>
<tr>
<td>Logical</td>
<td>!</td>
<td>Logical Negation</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitwise</td>
<td>~</td>
<td>Bitwise Negation</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt;</td>
<td>Shift Left</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
<td>Shift Right</td>
</tr>
</tbody>
</table>

Table A.1
Commonly Used Verilog Operators
Sequential Logic

The following three code modules in Figure A.21 evolve the dataflow implementations of the gated SR latch, the master-slave SR flip-flop and the two bit binary counter designed earlier at the gate level.

<table>
<thead>
<tr>
<th>// Dataflow Level Model</th>
<th>// Use two SR Latches in a master slave configuration to build a flip-flop</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Gated SR Latch</td>
<td></td>
</tr>
<tr>
<td>module gsrLatch(q, qnot, sg, rg, clr, enab);</td>
<td>module srmsff(q, qnot, s, r, clk, clr);</td>
</tr>
<tr>
<td>input sg, rg, clr, enab;</td>
<td>input s, r, clk, clr;</td>
</tr>
<tr>
<td>output q, qnot;</td>
<td>output q, qnot;</td>
</tr>
<tr>
<td>wire rL, sL;</td>
<td>gsrLatch master(qm, qmnot, s, r, clr, clk);</td>
</tr>
<tr>
<td>wire q, qnot;</td>
<td>gsrLatch slave(q, qnot, qm, qmnot, clr, ~clk);</td>
</tr>
<tr>
<td>// Build the gating logic</td>
<td>endmodule</td>
</tr>
<tr>
<td>assign rL = rg &amp; clr &amp; enab;</td>
<td></td>
</tr>
<tr>
<td>assign sL = sg &amp; clr &amp; enab;</td>
<td></td>
</tr>
<tr>
<td>// Build the basic RS latch</td>
<td></td>
</tr>
<tr>
<td>assign q = ~(rL</td>
<td>~clr</td>
</tr>
<tr>
<td>assign qnot = ~(sL</td>
<td>q);</td>
</tr>
<tr>
<td>endmodule</td>
<td></td>
</tr>
</tbody>
</table>

| // Build a synchronous two bit binary up counter |
| // using master slave SR flip-flops |
| module TwoBitCntr(qA, qB, clr, clk); |
|     input clr, clk; |
|     output qA, qB; |
|     wire sA, rA; |
|     wire qA, qAnot, qB; |
|     assign sA = qAnot & qB; |
|     assign rA = qA & qB; |
|     srmsff FFB(qB, qBnot, qBnot, qB, clk, clr); |
|     srmsff FFA(qA, qAnot, sA, rA, clk, clr); |
| endmodule |

Figure A.21
Dataflow Models of the S R Latch, Master-Slave Flip-Flop, and Two Bit Binary Up Counter
A.3.3 The Behavioral Model

The behavioral model increases the design abstraction by an additional level. Thinking about the design moves above considerations of the flow of data within the system to the algorithms that express the behavior of the system. At the behavioral level, the model begins to take on more of the guise of a C or C++ program than a digital circuit. Flow of control through the system is expressed in the familiar looping and branching constructs rather than in logic gates.

A.3.3.1 Program Structure

At the behavioral level, one of the major differences between languages such as C or C++ becomes clear. Unlike either C or C++, in which flow of control is generally sequential, flow of control in Verilog is concurrent. Statements in C or C++ execute in series; those in Verilog execute in parallel.

**always** and initial Statements

At the behavioral level, a Verilog program is structured as a collection of initial and/or always blocks. Each such block express a separate flow of control and each will finish execution independent of any other block. A module may define multiple initial and/or always blocks; however, such blocks cannot be nested. Beyond the input, output statements, and parameter declarations, all behavioral statements must be included in either one of these blocks.

The statements contained in an initial block (delimited by begin and end) are evaluated one time at the start of a simulation. The statements contained in an always block (delimited by begin and end) are evaluated continuously from the start of a simulation.

The always and initial statements are two of the many keywords in Verilog that allow one to set stimuli to a module. The syntax for the initial statement is given as,

```
Syntax
initial
begin
  Initial statements
end
```
The syntax for the always statement is given as follows,

```
always begin
    statements to always be executed
end
```

### A.3.3.2 Procedural Assignment

Assignment in the behavioral model differs from that in either the gate level or dataflow model. In the behavioral model, *procedural assignment* statements are used to update the state circuit variables. In the dataflow model, the *continuous assignment* construct continually updates the value on the net on left hand side. In contrast, in the behavioral model, a value is only updated as the result of the execution of a procedural assignment statement.

Verilog supports two kinds of procedural assignment: *blocking* and *non-blocking* and two kinds of blocks: *sequential* and *parallel*. Statements in a *sequential* block, which is delimited by a `begin` and an `end`, are executed in sequence. Statements in a *parallel* block, which is delimited by a `fork` and a `join`, are executed in parallel.

*Blocking assignment* statements are executed in the order in which they are written in a sequential block. They will block the execution of subsequent statements that appear in the *same sequential block*; they will not block the execution of statements that appear in a parallel block. A *non-blocking assignment* will not block subsequent statements in a sequential block.

Put another way. A *blocking assignment* will successively evaluate the *right hand* side then the *left hand* side of each an assignment statement in a *sequential* block. A *non-blocking assignment* will evaluate *all* of the *right hand* sides then *all* of the *left hand* sides of each statement in a *sequential* block.
The syntax for the two types of assignment is given in the following,

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Blocking</th>
<th>Non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>aVariable = aValue;</td>
<td>aVariable &lt;= aValue;</td>
</tr>
<tr>
<td></td>
<td>#d aVariable = aValue</td>
<td>#d aVariable &lt;= aValue</td>
</tr>
</tbody>
</table>

A.3.3.3 Delays

Delays may be incorporated on either side of the assignment statement according to the following syntax.

How each is interpreted can be a bit confusing.

**Blocking**

The first statement says,

Evaluate \( aValue \) then block for \( d \) time units before assigning \( aValue \) to \( aVariable \). Any subsequent use of \( aVariable \) will get the new value.

The second statement says,

Block for \( d \) time units before evaluating \( aVariable = aValue \). The variable \( aVariable \) will have the value \( aValue \) \( d \) time units in future.

**Non-blocking**

The first statement says,
Evaluate \textit{aValue}. Schedule \textit{aVariable} to be updated \(d\) time units later; however, continue processing other statements. Any other variables using the value of \textit{aVariable} within the next \(d\) time units will be assigned the old value.

The second statement says,

Wait \(d\) time units before evaluating \(aVariable = aValue\). The variable \textit{aVariable} will have the value \textit{aValue} \(d\) time units in future.

The following two examples in Figure A.22 and Figure A.24 will illustrate the behavior for each of the four cases in the same and in separate initial blocks.

\textbf{Example A.3}

```verilog
// Illustrate Procedural blocking and non-blocking assignment
// Separate initial block
module blockingNonblocking();
// declare temp registers
reg a,b,c,d,e,f,g,h,i,j,k,l;

// initialize reg variables
initial
begin
    a = 0; b = 0; c = 0; d = 0; e = 0; f = 0;
    g = 0; h = 0; i = 0; j = 0; k = 0; l = 0;
end

initial
begin
    // delay on right hand side
    // blocking
    a = #10 1;
    b = #2 1;
    c = #4 1;

    // non-blocking
    d <= #10 1;
    e <= #2 1;
    f <= #4 1;
end

Figure A.22a
Using Procedural Blocking and Non-blocking Assignment
```
Example cont.

```verilog
// Illustrate Procedural blocking and non-blocking assignment
// Separate initial block

initial
begin
    // delay on left hand side
    // blocking
    #10 g = 1;
    #2 h = 1;
    #4 i = 1;

    // non-blocking
    #10 j <= 1;
    #2 k <= 1;
    #4 l <= 1;
end

initial
begin
    $display("time, a, b, c, d, e, f, g, h, i, j, k, l");
    $monitor($time, " %b, %b, %b, %b, %b, %b, %b, %b, %b, %b, %b, %b," 
              "%b", a, b, c, d, e, f, g, h, i, j, k, l);
    #50 $finish(1);
end
endmodule

Figure A.22b
Using Procedural Blocking and Non-blocking Assignment
```
From the execution of the code fragment, we observe the output given in Figure A.23,

<table>
<thead>
<tr>
<th>time</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure A.23
System Output from the Module in Figure A.22

- The variables $a$ and $g$ from the two initial blocks change state at time 10. The variables ($b$ and $c$) and ($h$, and $i$) follow similarly according to their specified delays or 2 and 4 time units after $a$ and $g$ respectively.

- After the blocking statements have been evaluated, the non-blocking statements are evaluated.

- The variable $d$ is assigned the value 1 10 time units after the blocking statements in the first initial block, the expression $j<=1$ is evaluated 10 time units after the blocking statements in the second initial block.

- The variables $e$ and $f$ are evaluated 2 and 4 time units respectively after the blocking statements in the first initial block.

- Finally, the expressions $k<=1$ and $l<=1$ are evaluated 2 and 4 time units respectively after the blocking statements in the second initial block.
Example A.4

```verilog
// Illustrate Procedural blocking and non-blocking assignment
// Single initial block
module blockingNonblocking();
    // declare temp registers
    reg a,b,c,d,e,f,g,h,i,j,k,l;
    // initialize reg variables
    initial
        begin
            a = 0; b = 0; c = 0; d = 0; e = 0; f = 0;
            g = 0; h = 0; i = 0; j = 0; k = 0; l = 0;
        end
    initial
        begin
            // delay on right hand side
            // blocking
            a = #10 1;
            b = #2 1;
            c = #4 1;
            // non-blocking
            d <= #10 1;
            e <= #2 1;
            f <= #4 1;
            // delay on left hand side
            // blocking
            #10 g = 1;
            #2 h = 1;
            #4 i = 1;
            // non-blocking
            #10 j <= 1;
            #2 k <= 1;
            #4 l <= 1;
        end
    initial
        begin
            $display("time, 	a, 	b, 	c, 	d, 	e, 	f, 	g, 
h, 
i, 
j, 
k, 
l");
            $monitor($time, " \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b, \t%b",
                a,b,c,d,e,f,g,h,i,j,k,l);
            #50 $finish(1);
        end
endmodule
```

Figure A.24
Using Procedural Blocking and Non-blocking Assignment
The results following execution are given in Figure A.25.

<table>
<thead>
<tr>
<th>time,</th>
<th>a, b, c, d, e, f, g, h, i, j, k, l</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0</td>
</tr>
<tr>
<td>10</td>
<td>1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0</td>
</tr>
<tr>
<td>12</td>
<td>1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0</td>
</tr>
<tr>
<td>16</td>
<td>1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0</td>
</tr>
<tr>
<td>18</td>
<td>1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0</td>
</tr>
<tr>
<td>20</td>
<td>1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0</td>
</tr>
<tr>
<td>26</td>
<td>1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0</td>
</tr>
<tr>
<td>28</td>
<td>1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0</td>
</tr>
<tr>
<td>32</td>
<td>1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0</td>
</tr>
<tr>
<td>42</td>
<td>1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0</td>
</tr>
<tr>
<td>44</td>
<td>1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0</td>
</tr>
<tr>
<td>48</td>
<td>1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1</td>
</tr>
</tbody>
</table>

Figure A.25
System Output from the Module in Figure A.24

The major differences between the two implementations are reflected in the evaluation times for the variables d, e, f, g, h, and i.

**Combinational Logic**

The next example, in Figure A.26, implements the earlier NandNor combinational logic circuit using a behavioral model and utilizing both the blocking and non-blocking assignments with the right and left hand side delays.
Example A.5

module blocking_nonblocking();
    reg a,b, AandB,AorB, AnandB,AnorB;
    reg e,f, EandF,EorF, EnandF,EnorF;

    // Blocking Assignment
    initial
    begin
        a = 1; b = 1;
        // Delay on the right hand side
        AandB = #10 a&b;
        AnandB = #11 ~AandB;

        // Delay on the left hand side
        #10 AorB = a|b;
        #11 AnorB = ~AorB;
    end

    // Non-blocking Assignment
    initial
    begin
        e = 1; f = 1;
        // Delay on the right hand side
        EandF <= #10 e&f;
        EnandF <= #11 ~EandF;

        // Delay on the left hand side
        #12 EorF <= e|f;
        #13 EnorF <= ~EorF;
    end

    initial
    begin
        $display("%t time	 a, 	b, 	AnandB, 	AorB, 	AnorB, e, f, 	EandF, 	EnandF, 	EorF, 	EnorF");
        $monitor($time, "%b %b %b %b %b %b %b %b %b %b %b %b %b %b %b %b", a,b, AandB, AorB, AnorB, e, f, EandF, EnandF, EorF, EnorF);
        #50 $finish(1);
    end
endmodule

Figure A.26
A Combinational Logic Module Using a Behavioral Verilog Model
With Blocking and Non-blocking Assignment
**Example cont.**

The outputs of the circuits for each of the cases are given in Figure A.27.

<table>
<thead>
<tr>
<th>time</th>
<th>a, b, AnandB, AnorB, e, f, EnandF, EnorF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 1 x</td>
</tr>
<tr>
<td>21</td>
<td>1 1 0</td>
</tr>
<tr>
<td>25</td>
<td>1 1 0</td>
</tr>
<tr>
<td>42</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>

![Figure A.27](image)

Observe that based upon the order of evaluation of the non-blocking assignment, the NAND operand is never assigned a valid value.

### A.3.3.4 Flow of Control

The behavioral Verilog model supports many of the familiar flow of control constructs such as branches, switches, and loops. In addition, the language provides support for event based control.

#### Events

Verilog supports four different types of *event based control*. These are given as

- Regular event
- Named event
- OR event
- Level

Each is identified by the event control symbol, @. Verilog interprets an *event* as a change in the value of either a net or a register. Such a change can be used to invoke the evaluation of either a single statement or a block of statements.
The syntax for each is given as follows,

### Syntax

- **Regular Event**
  
  ```
  @(signal) action
  variable = @(signal) action
  ```

  - signal may be clock, posedge clock, negedge clock for example

- **Named Event**
  
  ```
  event anEvent // event is a keyword
  always @(anEvent) action
  ```

- **OR Event**
  
  ```
  always @(signal1 or signal2 or signal3 or...) action
  ```

- **Level**
  
  ```
  always wait(signal) action // wait is a keyword
  ```

### Branches

Like the C and C++ languages, Verilog utilizes the `if` and `if else` constructs to select alternate paths of execution based upon the value of a condition variable. Permitted combinations follow the C and C++ syntax.

```plaintext
Syntax
if (condition)
  statement;
if (condition)
  statement1;
else
  statement2;

if (condition1)
  statement1;
else if (condition2)
  statement2;
else
  statement 3;
```

If statement comprises a block of statements, the block must be delimited by the begin-end pair.
Case Statement

The *switch* or *case* statement in Verilog uses the Pascal rather than the C language syntax as shown,

```
Syntax
   case (expression)
       label0:  statement0;
       label1:  statement1;
       ...
       labeln-1:  statementn-1;
       default:  defaultStatement;
   endcase
```

If statement comprises a block of statements, the block must be delimited by the begin-end pair.

Unlike the C switch, once a statement or block of statements is evaluated, flow of control leaves the case rather than continuing through the remaining alternatives.

Loops

The Verilog language supports the four common loop constructs.

- while
- repeat
- for
- forever
The first three should be familiar from the C or C++ languages. The forever is unique to Verilog. The syntax for each is given as follows.

```
Syntax
  while(test)
  begin
    loop body
  end

  repeat(repeatcount)
  begin
    loop body
  end

  for(init; test; action)
  begin
    loop body
  end

init and action are usually assignments.

forever
  begin
    loop body
  end
```
Sequential Logic

The following three code modules in Figure A.28 evolve the behavioral implementations of the gated SR latch, the master-slave SR flip-flop and the two bit binary counter designed earlier at the gate and dataflow levels.

---

// Behavioral Level Model
// Gated SR Latch
module gsrLatch(q, qnot, s, r, clr, enab);
    input s, r, enab, clr;
    output q, qnot;
    reg q, qnot;
    always@ (~clr or enab)
        begin
            if(~clr)
                begin
                    q = 1'b0;
                    qnot = 1'b1;
                end
            else
                begin
                    if (s & ~r)
                        begin
                            q <= s;
                            qnot <= r;
                        end
                    else if (~s & r)
                        begin
                            q <= s;
                            qnot <= r;
                        end
                end
        end
endmodule

// Use two SR Latches in a master slave configuration to build a flip-flop
module srmsff(q, qnot, s, r, clk, clr);
    input s, r, clk, clr;
    output q, qnot;
    gsrLatch master(qm, qmnot, s, r, clr, clk);
    gsrLatch slave(q, qnot, qm, qmnot, clr, ~clk);
endmodule

// Build a synchronous two bit binary up counter using master slave SR flip-flops
module TwoBitCntr(qA, qB, clr, clk);
    input clr, clk;
    output qA, qB;
    reg sA, rA;
    wire qA, qAnot, qB;
    always@(posedge clk)
        begin
            sA = qAnot & qB;
            rA = qA & qB;
        end
    srmsff FFB(qB, qBnot, qBnot, qB, clk, clr);
    srmsff FFA(qA, qAnot, sA, rA, clk, clr);
endmodule

---

Figure A.28
Behavioral Models of the S R Latch, Master-Slave Flip-Flop, and Two Bit Binary Up Counter
The next code module in Figure A.29 illustrates a more commonly used approach for modeling a counting, timing, or registered types of designs. Rather than working with individual flip-flops, the design is approached algorithmically.

```verilog
// Build a synchronous two bit binary up counter
module TwoBitCntr(state, clr, clk);
  input clr, clk;
  output[1:0] state;
  reg[1:0] state;

  // Name the states
  parameter state0 = 2'b00;
  parameter state1 = 2'b01;
  parameter state2 = 2'b10;
  parameter state3 = 2'b11;

  // Build a synchronous two bit binary up counter
  always@(~clr or negedge clk)
  begin
    if(~clr)
      begin
        state = state0;
      end
    else case(state)
      state0:
        state = state1;
      state1:
        state = state2;
      state2:
        state = state3;
      state3:
        state = state0;
    endcase
  end
endmodule
```

Figure A.29
A Behavioral Model of a Two Bit Binary Up Counter Using a Case Statement

### A.4 Testing and Verifying the Circuit

Once the circuit is designed and modeled in Verilog we move into the next phase. We first need to verify that the model functions properly. The next step is to use it for its intended purpose. To that
end, we perform any functional, parametric, and stress tests on the design, through the model, that we deem necessary to confirm the design before committing to hardware.

We will illustrate the verification phase of the process using the NandNor circuit that was designed earlier. To do this we create a test bench. A test bench models the electronics workbench. It comprises the measurement and stimulus instruments and the circuit to be tested. The modules used for stimulus and measurement will go in a test module.

A high level model for the test bench has the following general structure.

The test bench plays the same role as does the main() function in C or C++ and the top level class in Java. It acts as the outermost container in the program.

A.4.1 The Circuit Module

We will use the NandNor circuit that we developed earlier as the circuit module to be tested and verified. During test, we must confirm that each path through the logic circuit is functional and that it performs according to specification. The logic diagram for the circuit is repeated in Figure A.31 for reference.

A.4.2 The Test Module

As with other modules the test module will have a set of inputs and a set of outputs. The inputs to the test module will be the outputs of the UUT and will model the measurement equipment. The outputs from the test module will be the inputs of the UUT. These will model the stimulus equipment.
The tester module for the *NandNor* combinational logic is given in the code fragment in Figure A.32.

```verilog
module Tester (X, Y, XnandYin, XnorYin);
    input     XnandYin, XnorYin;
    output    X, Y;
    reg       X, Y;
    parameter stimDelay = 10;

    initial // Stimulus
    begin
        X = 1; Y = 1;
        #stimDelay X = 0;
        #stimDelay Y = 0;
        #stimDelay X = 1;
    end

    initial // Response
    begin
        display("\t Time, \t \tX, \t Y , \t XnandYin, \t XnorYin\t");
        monitor($time, "\t \t %b, \t %b, \t %b, \t %b", X, Y, XnandYin, XnorYin);
    end
endmodule
```

The opening lines of the test module identify the sets of inputs and outputs. These signals will come from the UUT and will send stimulus vector to the UUT. The parameter, `stimDelay`, specifies the delay between the applications of successive test vectors. Next the test vectors are defined and appear as successive statements.

Four different combinations of the signals X and Y are applied to the circuit input. A delay is specified between each stimulus application. The design of the *NandNor* circuit assumes ideal parts. Had the logic gates included a delay, the `stimDelay`, between the applications of successive vectors would have provided time for the signal to propagate through the logic block.

The test vectors are written as statements within an `initial` block. Thus, the test suite is applied one time during the simulation. The circuit output in response to the set of test vectors is presented using the `$display` and `$monitor` system tasks.
The tester for the behavioral sequential *two bit binary counter* module follows the same pattern with several additions and is presented in the code module in Figure A.33.

```verilog
// Test module for two bit binary up counter
module tester(clr, clk, qA, qB);
    input qA, qB;
    output clr, clk;

    reg clk, clr;

    parameter stimDelay = 15;
    parameter clkDelay = 5;

    initial
    begin
        clk = 0;
        clr = 0;
        #stimDelay clr = ~clr;

        repeat(16)
            #clkDelay clk = ~clk;
    end

    initial
    begin
        $display("\tTime, \t\tqA, \t\tqB, \t\tclr, \t\tclk");
        $monitor($time,"\t\t%b, \t\t%b, \t\t%b, \t\t%b", qA, qB, clr, clk);
    end

endmodule

Figure A.33
A Tester Module for a Two Bit Binary Up Counter

Clocks and Resets
A synchronous sequential circuit will need a strobe, enable, or clock in order to operate. Good designs also include a reset or clear signal to establish the initial state of the circuit. Typically
these signals are supplied by the tester with a block of code such as the code fragment in Figure A.34.

```verilog
reg clk, clr;

parameter stimDelay = 15;
parameter clkDelay = 5;

initial
begin
  clk = 0;
  clr = 0;
  #stimDelay clr = ~clr;

  always
  #halfPeriod clk = ~clk;
end

Figure A.34
Building a Clock
```

**A.4.3 The Test Bench**

We’ll now bring everything together with the test bench. In the test bench, we instantiate one copy of the NandNor gate, this is the UUT, and one copy of the Tester. These are the stimulus and monitoring instruments. Finally, we connect them together using wires as illustrated in the code fragment in Figure A.35.

```verilog
module MyTest bench;
  wire XnandY, XnorY, X, Y;
  NandNor aNandNor (XnandY, XnorY, X, Y);
  Tester aTester (X, Y, XnandY, XnorY);
endmodule

Figure A.35
Building a Test Bench
A.4.4 Performing the Simulation

If the simulation is now run, the test vectors are successively applied to the input of the UUT. As the simulation executes, the $monitor system task will display the state of the input and output signals and the system time at which the samples were taken. These appear in Figure A.36.

<table>
<thead>
<tr>
<th>Time,</th>
<th>X,</th>
<th>Y,</th>
<th>XnandYin,</th>
<th>XnorYin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure A.36
Output From Running the Test Bench

If the results are satisfactory, we can move on to the real work of confirming the design.

A.5 Summary

This appendix introduced the Verilog language and presented the important features and capabilities that are used in this book. It does not purport to be a comprehensive study of the language nor of the kinds of testing necessary to confirm an embedded design.

We began with the basic components and organization of a Verilog program; then examined gate-level or structural, dataflow, and behavioral models for combinational logic circuits, and then followed with similar models for sequential circuits. Design is only one element of the product development, each design must also be tested to confirm that it meets specified requirements then the model must be used for testing and verifying the original design. To that end, the appendix concludes with a short discussion on how one can formulate a test bench and test suites to verify the proper operation. The material on testing establishes a foundation to enable the developer to build test cases that will enable testing, verifying, and stressing to the desired level.

Verilog and other hardware design languages like VHDL offer today’s designers a rich and powerful set of tools to help to attack today’s complex designs. It’s beyond the scope of this text to present a comprehensive treatise on testing or modeling using a language like Verilog. The interested reader is strongly encouraged to any of the reference material cited in the bibliography.