Introduction to Verilog

The Verilog Hardware Definition Language (Verilog HDL) was originally developed as a tool for the simulation and testing of digital systems. At the time digital design was moving to systems having 100,000+ gates and the existing methods for such design were becoming impractical. Verilog was quickly adopted by many designers and soon used also for synthesis.

Verilog was based on the syntax of the C programming language and shares many of its operators and constructions. If you are familiar with C, learning Verilog should be pretty easy. The most significant difference is that C is a procedural language used to implement sequential algorithms while Verilog is used to model hardware which operates concurrently.

A Short History

Verilog was originally developed at Gateway Design Automation in 1985. Phil Moorby developed it as as a proprietary Hardware Description Language (HDL) for the Gateway Verilog-XL digital logic simulator. Cadence Design Systems became the owner of Verilog when they acquired Gateway Design Automation in 1989.

In 1990, as a response to the growing popularity of VHDL, Cadence put Verilog in the public domain. At the same time Open Verilog International (OVI) was formed to manage the Verilog HDL. The Verilog-XL user manual was the main reference for Verilog and served as its standard, the Verilog 1.0 Reference Manual. Some minor changes to Verilog were reflected in Verilog 2.0 a few years later.

In 1993 OVI submitted a request to the IEEE to make Verilog an IEEE Standard. In 1995 this process produced IEEE 1364-1995, the first official Verilog standard based on Verilog 2.0. This version of Verilog was popularly known as Verilog-95. In 1997 work began on an updated standard which addressed short-comings in Verilog-95 and incorporated significant enhancements. In 2001 this effort resulted in IEEE 1364-2001 with this version of Verilog known as Verilog-2001.

One of the traditional shortcomings of Verilog has been the lack of constructions that are useful for systems level. Verilog has always been strong for modeling at the gate and Register Transfer Level (RTL) but weaker than VHDL in modeling at a system level. To address these short-comings SystemVerilog was developed. SystemVerilog was originally developed on top of Verilog under the auspices of Accellera starting about 2002. It became IEEE Standard 1800-2005 in 2005.

SystemVerilog was combined with the Verilog standard as IEEE 1800-2009. This standard is known as SystemVerilog. The most recent version of this standard is EEE 1800-2012, published in February, 2013.

1.1 Modeling Combinational Logic with Verilog

Combinational logic uses the current value of the inputs to determine the value of the outputs. Unlike sequential logic, combinational logic lacks memory. Combinational logic can be modeled in Verilog using gate primitives, Boolean expressions, or procedural descriptions. These correspond to structural, dataflow, and behavioral descriptions. We will first discuss modeling with structural and dataflow descriptions of logic circutis. Behavioral modeling is deferred until the end of the section.

The schematic symbol for an AND gate is shown in figure 1.1

Figure 1.1. An AND gate

The AND gate has two inputs, *a* and *b*, and one output *c*. The relationship between the two inputs and the output can be described in several ways. We could use a truth table and list, for all possible inputs, the corresponding output. Or, we could more compactly describe the relationship between inputs and outputs by means of a Boolean equation.

In Verilog a Boolean equation describing the relationship between the inputs and output can be written as a *continuous assignment statement,*

```
assign c = a&b;
```

Here a, b, and c are Verilog *nets*. In Verilog the net data type is used to model a logic signal much like the voltage on a physical wire. A net has the value with which it is driven, if a net is undriven it floats. In Verilog a scalar net is the default data type. Verilog also has a second type of data, a variable, which holds the value it is assigned until a new value is assigned. A variable is similar in this regard to a switch, it has memory; set it to a value and it retains that value until it is set to a new value.

The Boolean operator used in the continuous assignment statement should be familiar to anyone familiar with the C programming language. Most operators in Verilog are borrowed from C. The operator & indicates a bitwise AND operation on a and b. Other common operators include |, which indicates a bitwise OR operation, and ^ which indicates and XOR operation. A bit value can be complemented with ~. The Verilog keyword **assign** indicates that this is a continuous assignment statement. Unlike a procedural language such as C, this expression is evaluated continuously in simulation time. If there are any other continuous assignment statements in the model they will also be continuously evaluated at the same time, or concurrently. This concurrency is a basic difference between Verilog and procedural programming languages.

To complete our Verilog model of the AND gate in figure 1, we put the continuous assignment statement inside a Verilog *module.* You can think of a Verilog module as the package that contains the logic circuit, in this case for the AND gate. A Verilog module is similar to a C function but is used to model a hardware component. A Verilog module modeling the 2-input AND gate in figure 1.1 is the following

```
module and2 (c, b, a);
     output c;           // module outputs
     input a,b;          // module inputs
     assign c = a&b;     // continuous assignment
endmodule
```

The module begins with the **module** keyword followed by the name of the module, and2, and then the *port list*. The nets in the port list connect the module to the rest of the world much like the "wires" on the AND gate in a schematic or the pins of an integrated circuit. Inside the module we need to declare them as input or output using the corresponding Verilog keywords. The continuous assignment statement is then used to implement the AND function in this module and we use the **endmodule** keyword to indicate the end of the module. Comments are placed, as in C, between /* and */ or after //. Like C, Verilog is case sensitive so that `And2 and AND2` are not the same as `and2`.

Verilog also allows an ANSI-C style port list so that this 2-input model of the AND gate could also be written as

```
module and2 (output c,   // ANSI-C style
             input a,b); // port list
     assign c = a&b;
endmodule
```

The AND operation is used frequently enough that Verilog includes an AND gate *primitive* as part of its built in component library. Instead of having to define the module `and2` as shown above we could use for the 2-input AND gate

and (c, a, b);

The AND gate primitive, like all Verilog primitives, lists the scaler output on the left of the port list and the scalar inputs on the right. It also allows for an arbitrary number of inputs which we could add to the right side of the port list. The Verilog primitives, **or**, **xor**, **nand**, and **nor** work similarly. The primitive **not** is used for an inverter, although in this case there is a single input on the right and an arbitrary number of outputs on the left, **buf** works similarly but does not invert the input. This output and input ordering is easy to remember since assigment statements also have inputs on the right and output on the left. (Note that port lists for modules in these notes use the same convention, outputs on the left of the port list and inputs on the right.)

A Multiplexer

Now that we have seen how we can use Verilog to model basic logic components we can proceed to construct more interesting logic elements. We first consider the 2-to-1 multiplexer shown in figure 1.2 and a realization in terms of simple gates in figure 1.3.

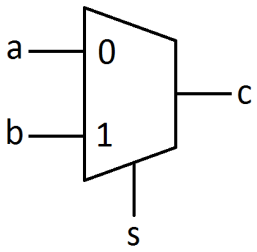

Figure 1.2. A multiplexer

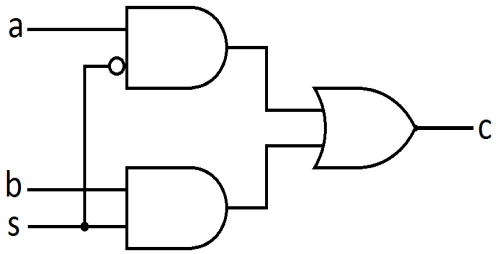

Figure 1.3. A realization of the multiplexer

Using continuous assignment statements the multiplexer can be implemented directly from the schematic in figure 1.3 using Verilog primitives. The Verilog module in this case is

```
module mux2(c,a,b,s);
     output c;      // data outputs
     input a,b;     // data inputs
     input s;       // select input

     wire snot;     // complement of s
     wire w0,w1;    // internal connections

     not G0 (snot,s);
     and G1 (w0,a,snot);
     and G2 (w1,b,s);
     or  G3 (c,w0,w1);

endmodule
```

This is a Verilog *structural model* since we have constructed the model of the multiplexer by using components, in this case Verilog *primitives*. Verilog primitives have been *instantiated* to model the inverter, the two AND gates, and the OR gate. The associated names, `G0` to `G3`, are optional but help to identify individual components. The circuit has one output, `c`, and three inputs `a`, `b`, and `s`. In our code we have chosen to separate the data inputs, `a` and `b`, from the select input `s` by using two separate input statements. All the inputs and outputs appear in the port list. While the order of variables appearing in the port list is arbitrary we have chosen to follow the same order as used with Verilog primitives putting the output on the left and the inputs on the right. Three internal nets, `snot`, `w0`, and `w1`, declared with the **wire** keyword, connect the Verilog primitives inside the module. Note that the ordering of the four gate instantiations does not matter since they operate concurrently.

Instead of using Verilog primitives we could model the gates in figure 1.3 using continuous assignment statements. The corresponding Verilog module is then

```
module mux2(c,a,b,s);
     output c;      // data outputs
     input a,b;     // data inputs
     input s;       // select input

     wire snot;     // complement of s
     wire w0,w1;    // internal connections

     assign snot = ~s;
     assign w0 = a&snot;
     assign w1 = b&s;
     assign c = w0|w1;

endmodule
```

This is an example of a Verilog *dataflow model*. Generally dataflow models describe how data is transformed as it moves between registers. Each of the gates in this module is modeled in a continuous assignment statement with the corresponding Boolean expression. Alternatively, instead of the four continuous assignment statements, we could use a single continuous assignment statement,

```
     assign c = (a&(~s))|(b&s);
```

This replaces the four continuous assignment statements and also eliminates the need for the internal connections. However, we can do better. Verilog borrows from C the conditional operator which is very useful in modeling this type of construction. Using the conditional operator we can replace the previous continuous assignment statement with

```
     assign c=s?a:b;
```

This conditional operator works as follows: If the conditional variable s = 0 then the statement will assign to c the value of the first variable while if the conditional value s = 1 then the statement will assign to c the value of the second variable b. Using the conditional operator a Verilog module which models the 2-to-1 multiplexer is the following

```
module mux2(c,a,b,s);
     output c;      // data outputs
     input a,b;     // data inputs
     input s;       // select input

     assign c = s?a:b;

endmodule
```

Let's get physical

The Verilog modules so far model functional behavior. Physical hardware however is more complex, in particular, in a physical system it takes time to move charge. Consequently, the propagation of signals through real hardware is subject to delays. To model delays when simulating hardware Verilog includes several constructions.

In the dynamic behavior of electrical circuits the physics of a device typically requires that a signal be applied to an input for a certain amount of time before it propagates to the output. Modeling such delays is an important part of Verilog coding for simulation. On the other hand Verilog coding for synthesis does not require delay modeling as the delays are due to the hardware.

Using delays we can model the Verilog circuit of the multiplexer. The logic circuit for a 2-to-1 muliplexer is shown in figure 1.4

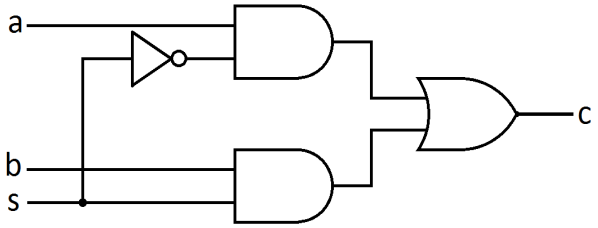

Figure 1.4. Logic circuit of a 2-to-1 multiplexer

Assume that all the gates and inverters have a delay of 2 ns. These delays can be implemented in the Verilog structural model of the muliplexer as shown in the following code

```
// structural model of a 2-input muliplexer
`timescale 1 ns / 100 ps

module mux2(c,a,b,s);

   output c;    // data output
   input  a,b;  // data inputs
   input  s;    // select input

   wire snot,w0,w1;  // internal wires

   not #2 G0(snot,s);
   and #2 G1(w0,a,snot);
   and #2 G2(w1,b,s);
   or  #2 G3(c,w0,w1);

endmodule
```

Here the 2 time unit delays are indicated by #2 in the instantiation of the inverter, and, and or gates. This operator will impose a 2 time unit delay before changes to any of the inputs appears in the output. In addition, we have used the `timescale compiler directive to indicates that in simulation the time unit is 1 ns and any delays will be rounded to the nearest 100 ps. Thus, for this Verilog module there is a 2 ns delay for each of the inverter and gates. Note that in this model the actual output delay depends on the signal path.

If we were to set both a and b to 1 and change the value of s from 1 to 0 we get the following timing diagram.

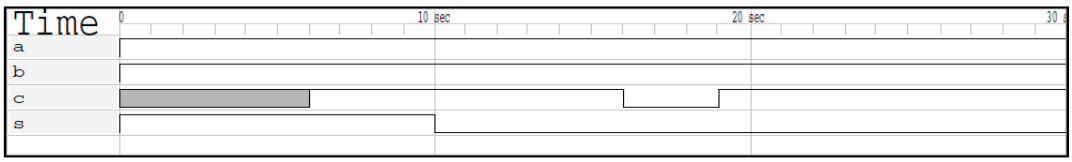

Figure 1.5. Timing for the 2-input Multiplexer

Note that the timing shows there is a glitch (due to a static 1-hazard) in the output

This code shows the simplest implementation of delays in a Verilog model. The # operator is used for delays. In actual hardware there is a time after a signal changes during which the signal must persist for any switching to take place, during this time the circuit needs sufficient time to move the output to a new value. The delays model this effect, if an input signal is applied to one of the gates for a period less than the indicated delay, in this case, 3 time units, then the gate output doesn't change. This effect can be seen by making the delay on the inverter 2 time units and looking at the response.

Delays can also be used with the dataflow description of the multiplexer. Consider the following

```
module mux2(c,a,b,s);
     output c;      // data outputs
     input a,b;     // data inputs
     input s;       // select input

     assign #5 c = s?a:b;

endmodule
```

In the continuous assignment statement the Verilog simulator will evaluate the right hand side of the expression whenever one of the variables, a, b or s, changes. The delay operator will cause the variable c to be updated 5 time units later. The single delay in this dataflow model does not depend on the path. It's an example of a *lumped delay.*

1.2 Verilog Values

The logic values 0 and 1 typically correspond to voltages in hardware. Such logic values are more complicated than simple binary values since the technology and implementation may lead to more complicated situations. For simulating hardware the use of only 0 and 1 is too restrictive.

To better model actual hardware Verilog uses a four valued logic system. In addition to 0 and 1 a logic signals can be unknown, represented by an “x” or high impedance, represented by a “z”. A signal represented by an “x” corresponds to a signal whose value is unknown, typically such values are undefined or may require more information to specify. Verilog sets any variables which are not initialized to this value. A high impedance value, “z” is typically associated with three-state devices. An undriven net, corresponding to an open circuit, defaults to z.

One consequence of having a four valued logic system is that the truth tables for the basic logic functions become more complicated. Consider for example a 2-input AND gate shown in figure 1.6 below. With the four valued logic system the truth table of this 2-input AND gate is that shown in figure 1.7.

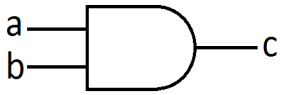

Figure 1.6. AND gate

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Figure 1.7. Truth table for AND gate

For the truth table in figure 1.7 the topmost row and the leftmost column represent the possible values of the respective inputs, a and b. The entries represent the output, c. The upper 2 by 2 block represent the outputs we would expect for an AND gate with the binary inputs, 0 and 1. The remaining entries represent the output when the inputs is x or z. The outputs are what we might expect. The output of the AND gate is always 0 if one of the inputs is 0. In the remaining cases we don't know what the output will be so we assign it a value of x. For an AND gate with more than 2 inputs the truth table is similar.

A 2-input OR gate, shown in figure 1.8, with four valued logic behaves similarly. When both inputs are 0 the output is 0. If one or both of the inputs is a 1 then the output of the OR gate is always 1. For all remaining possibilities the output is x. The truth table for the OR gate is shown in figure 1.9.

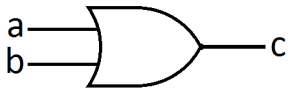

Figure 1.8. OR gate

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Figure 1.9. Truth table for OR gate

The primitive gate functions, **and** and **or**, defined in Verilog have truth tables corresponding respectively to those shown above.

The high impedance value, z, allows the modeling of three-state devices in Verilog. As an example consider and inverter and a three-state inverter. The Verilog **not** primitive models an inverter. Figure 1.10 shows the truth table of the not gate in the four valued logic system and figure 1.11 shows the primitive, **notif0**, of the not gate as a three-state device.

not	
0	1
1	0
x	x
z	x

Figure 1.10. A **not** gate

notif0	0	1	x	z
0	1	z	1 or z	1or z
1	0	z	0 or z	0 or z
x	x	z	x	x
z	x	z	x	x

Figure 1.11. A **notif0** gate

In both these tables the lefthand column, immediately under the gate name, correspond to the data input to the gate. The top row, to the right of the gate name for the **notif0** gate, indicates the control input to the gate.

Three-state buffers can be used in Verilog to connect a module to a bus. These have the truth tables shown in figures 1.12 and 1.13.

buf	
0	0
1	1
x	x
z	x

Figure 1.12. A **buf** gate

bufif0	0	1	x	z
0	0	z	0 or z	0 or z
1	1	z	1 or z	1or z
x	x	z	x	x
z	x	z	x	x

Figure 1.13. A **bufif0** gate

As an example of using three-state buffers consider the logic circuit for a 2-input multiplexer shown in figure 1.14

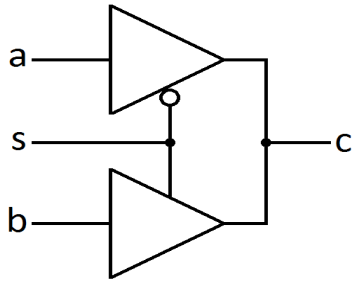

Figure 1.14. A muliplexer constructed with two three-state buffers.

A Verilog structural model of this circuit is the following module

```
module mux2tristate (c,a,b,s);

     output c;       // data output
     input a,b;      // data inputs
     input s;        // control input

     tri c;

     bufif0(c, a, s);
     bufif1(c, b, s);

endmodule
```

Note that **bufif1** is similar to **bufif0** but inverts the select signal. Also note the use of the keyword **tri**. This keyword is used to indicate a net with multiple drivers, in this example each of the buffer outputs. It works the same as **wire** but is used to indicate signals on a net can have different strengths, as for example in the case of wired logic.

Verilog Numbers

Verilog represents constants by default as decimals. Signed constants are in two's complement representation. Thus, the constant 1 is one in decimal, corresponding to the binary value 1. However, 10 corresponds to the the 4-bit unsigned binary value 1010 while – 10 corresponds to the 5-bit two's complement binary number 10110. Variables can be declared as signed by using the `signed` keyword.

Verilog numbers can also be represented as *based constants* which explicitly indicate the radix of the representation. This representation uses the b, o, d, and h to indicate respectively binary, octal, decimal, and hexidecimal. This representation is case insensitive so B, O, D, and H can also be used.

Sized constant numbers also indicated the number of bits used to represent the number. Using this representation the decimal number 255 can be represented as

```
8'b11111111
8'o377
8'd255
8'hff
```

Note that these define bit patterns, for example if a is defined as a signed integer, then `8'b11111111` represents –1. To make this more readable Verilog allows underscores to be inserted without affecting the value. So we can write, for example, `8'b11111111` as `8'b1111_1111`. Unsized constant numbers omit the explicit size, for the above

```
'b11111111
'o377
'd255
'hff
```

Signed numbers can be represented by using – before the constant, alternatively an s in the expression indicates that the bit pattern should be interpreted as a signed, two's complement number. The constant – 5 decimal for example can be represented as `-8'd5` which will put the two's complement of 5 in the 8 bits or as `8'hsfb` which will interpret the 8-bit hex number `fb` as a two's complement number.

Since Verilog has a four valued logic system we can also use `x` and `z` in constants. A 6-bit number, with the three least significant bits unknown, can be written

```
6'b101_xxx
6'o5x
```

The value 1 defaults to `32'b0000_0000_0000_0000_0000_0000_0000_0001`, a 32-bit value; if it is instead specified as `1'b1`, then it is a 1-bit value.

Frequently we would like to use multibit values, for example, a byte in memory of 8-bits or an instruction of 32-bits. In Verilog 8-bit signals, `a` and `b`, might be defined as

```
wire [7:0] a, b;
```

Here `[7:0]` indicates a *vector* that a consists of 8-bits, indexed from 7 to 0. Indexing them 7 to 0 is arbitrary, we could just as well have specified them as `[0:7]` or `[3:10]`. Our choice reflected the usual way to index bits in a byte. To select an individual bit, say bit 5 of `a`, we use `a[5]`. Braces can be used to concatenate and form larger objects. We could, for example, create a 16-bit vector with `a` the high byte and b the low byte as `{b,a}`. We can use this to easily create a 4-bit adder as follows:

```
module FourBitAdder (sum, carry_out, aop, bop);

     output [3:0] sum;
     output carry_out;
     input [3:0] aop, bop;

     wire [4:0] temp;

     assign temp = aop + bop;
     assign sum = temp[3:0];
     assign carry_out = temp[4];

endmodule
```

or more succinctly by

```
module FourBitAdder (sum, carry_out, aop, bop);

     output [3:0] sum;
     output carry_out;
     input [3:0] aop, bop;

     assign {carry_out, sum} = aop + bop;

endmodule
```

Arrays

In Verilog indexes are placed after the definition of the net or variable. For example, to define a 1KB memory we can do the following:

```
reg [7:0] memloc [0:1023];
```

Individual elements are then accessed by this index, for example the n-th byte in the array would be accessed as `memloc[n]`. It is important to distingush between the bits in the variable and the elements in the array. In this example each of the 1024 array elements is an 8-bit variable.

Elements in arrays are referenced by the index, for example, assume `data` is an 8-bit net type variable, then

```
assign data = memloc [476];
```

will drive the net variable `data` with the contents of value at location 476 in `memloc`.

Verilog Operators

Many of the operators used in Verilog expressions will be familiar to anyone who has used C or a related programming language. Additional operators have been introduced in Verilog to further facilitate the modeling and simulation of digital systems. There is a defined hierarchy for these operators which determine the order in which they are evaluated. Parenthesis can always be used to force evaluation. The following is brief discussion of some of the operators we use in these notes. For a complete discussion see section 5 of the Verilog standard.

We have already seen several of the bitwise operators. These include the bitwise operators for and, `&`, for or, `|`, and for exclusive-or, `^`. These are binary operators which perform their respective operations on corresponding operand bits of two operands. If the operands are different sized, the shorted is first filled with 0s to the left of the most significant bit. Negation, ~, is a unary operation which performs a bitwise complement of a single operand.

The bitwise operators can also be used as reduction operators. These operate on all bits in a single operand and produce a 1 bit result.

Operation	Result	Operation	Result
`0110&0111`	`0110`	`~011010`	`100101`
`0110|0111`	`0111`	`&011010`	`0`
`0110^0111`	`0001`	`|011010`	`1`
`0xx0|0111`	`01x0`	`^011010`	`1`

Arithmetic operators are the same as in C. Binary arithmetic operators support the basic operations of addition, +, subtraction, -, multiplication, *, division, /, and modulus, %. For arithmetic operations, if any bit in an operand is x, the result is x. Integer division will truncate the fractional part of a result and division by 0 produces the result of x. Arithmetic operations on a reg value or net are treated as unsigned unless they have been declared as signed.

Values of expressions are compared with relational and equality operators. These include less than, <, less than or equal, <=, greater than, >, and greater than or equal, >=. The logical equality operators are equal, ==, and not equal, !=. If the values being compared contain x or z these may produce an unknown. The case equal operator, ===, and the case not equal operator, !==, will also compare x and z bit values. Both relational and equality operators produce 1 when a comparison is a success and 0 when it fails. An ambiguous result returns x.

Operation	Result
x0110001 == x0110001	x
x0110001 === x0110001	1

Verilog includes the logical shift operators, logical shift left, <<, and logical shift right, >>, as well as the arithmetic shift operators, arithmetic shift left, <<<, and arithmetic shift right, >>>. The logic shift operators fill in bit positions with 0 for both right and left shifts. The arithmetic shift right

operator will fill in bit positions with the value of the leftmost bit.

Operation	Result
10100011<<3	00011000
10100011>>5	00000101
10100011<<<3	00011000
10100011>>>5	11111101

Two useful operators are the conditional and concatenation operators. The conditional operator, which we saw in the multiplexer model, has three operands which are Verilog expressions. The first expression represents a condition which is evaluated and compared to 0 or 1. If the expression evaluates to 1 the result is assigned the value of the second expression, if 0 then it is assigned the value of the third expression. Otherwise the result is x. Both the second and third expressions can be multi-bit vectors. For example,

```
wire [7:0] OpA, OpB, OpC

assign OpC = select?OpA:OpB
```

will update `OpC` with the 8-bits of `OpA` if `select` is 1 or `OpB` if `select` is 0. The concatenation operator will combine the bits of one or more expressions. It consists of two braces with arguments separated by commas. For example, the sum output of a 4-bit adder, can be combined with the carryout, and padded with zeros to create an 8-bit result,

```
{ 3'b000, carryout, sum[3:0]}
```

A Verilog Ripple Carry Adder

As another example we will look at the Verilog modeling of a ripple carry adder. The ripple carry adder is constructed from multiple full-adder circuits. The basic full adder can be implemented any number of ways; one method would be to implement a dataflow model based on the Boolean expressions for sum and carry. The corresponding Verilog module is

```
module FullAdder (carryout, sum, a, b, carryin);

    output carryout, sum;
    input a,b,carryin;

    assign sum = a^b^carryin;
    assign carryout = (a&b)|(carryin&a)|(carryin&b);

endmodule
```

This full adder will add the `a`, `b`, and `carryin` bits. The `sum` bit outputs the sum and if a carry is generated the `carryout` bit outputs this carry.

Suppose we wanted to add two 4-bit binary numbers, say A=(a_3, a_2, a_1, a_0) and B=(b_3, b_2, b_1, b_0), to produce a 4-bit sum, S=(s_3, s_2, s_1, s_0) and 1-bit carry out. To add flexibility to this 4-bit adder we will include a carry in. The full-adder module shown above is used as a component in this 4-bit ripple carry adder, as shown in figure 1.15.

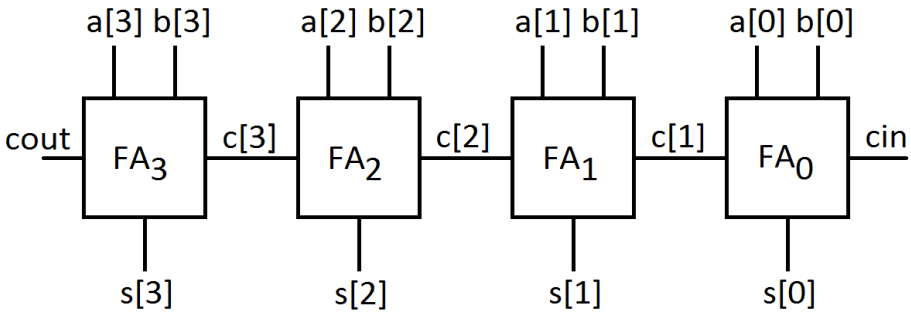

Figure 1.15. A 4-bit ripple carry adder

In the figure each full adder is represented as a box with three inputs, the two bits to be added and a carry in, and two outputs, the sum and carry. Figure 1.15. can be used as a guide to constructing the 4-bit adder. Each box corresponds to an instantiation of the full-adder module and the connections between these modules will correspond to those shown in figure 1.15. A Verilog module we will call `RippleCarryAdder4`, based on figure 1.15, is the following:

```
module RippleCarryAdder4 (s, cout, a, b, cin);

     output [3:0] s;     // vector output
     output cout;        // scalar output
     input  [3:0] a, b;  // vector inputs
     input  cin;         // vector input

     wire [3:1] c;       // internal carries

     // instantiate the full adders
     FullAdder FA0 (c[1], s[0], a[0], b[0], cin);
     FullAdder FA1 (c[2], s[1], a[1], b[1], c[1]);
     FullAdder FA2 (c[3], s[2], a[2], b[2], c[2]);
     FullAdder FA3 (cout, s[3], a[3], b[3], c[3]);

endmodule
```

The structure of the Verilog code in `RippleCarryAdder4` is straight forward. The first part of the code, the port list and declarations, define the interface to this module. In the remaining part the full-adder module, `FullAdder`, has been instantiated four times. All the nets in the port lists for the full-adder instantiations correspond directly to the labels and connections shown in figure 1.15.

Suppose instead of a 4-bit adder we wanted `RippleCarryAdder4` to code a module for an 8-bit adder. This would involve a straight-forward but tedious modification of `RippleCarryAdder4`. Each of the vectors would now have an upper range value of 7, moreover we would need to instantiate another 4 full adders with the indexes in each successive full adder increasing by one, the resulting 8-bit adder would be

```
module RippleCarryAdder8 (s, cout, a, b, cin);

     output [7:0] s;     // vector output
     output cout;        // scalar output
     input  [7:0] a, b;  // vector inputs
     input  cin;         // vector input

     wire [7:1] c;       // internal carries

     // instantiate the full adders
     FullAdder FA0 (c[1], s[0], a[0], b[0], cin);
     FullAdder FA1 (c[2], s[1], a[1], b[1], c[1]);
     FullAdder FA2 (c[3], s[2], a[2], b[2], c[2]);
     FullAdder FA3 (c[4], s[3], a[3], b[3], c[3]);
     FullAdder FA4 (c[5], s[4], a[4], b[4], c[4]);
     FullAdder FA5 (c[6], s[5], a[5], b[5], c[5]);
     FullAdder FA6 (c[7], s[6], a[6], b[6], c[6]);
```

```
    FullAdder FA7 (cout, s[7], a[7], b[7], c[7]);

endmodule
```

If we wanted a 16-bit adder (or 32-bit, or 64-bit, etc.) we could proceed in a similar fashion. As we increase the number of bits to be added the task becomes more tedious and the possibility of introducing and error increases. However, we can use Verilog to simplify our task and automate the code. In fact, we can easily construct a Verilog module for the general N-bit adder shown in figure1.16

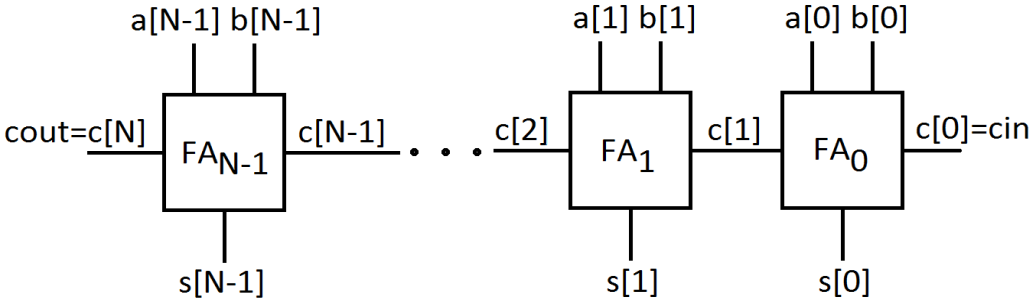

Figure 1.16. N-bit ripple carry adder

```
module RippleCarryAdderN (s, cout, a, b, cin);

    parameter N=8;              // default size of adder

    output [N-1:0] s;           // vector output
    output cout;                // scalar output
    input  [N-1:0] a, b;        // vector inputs
    input  cin;                 // vector input

    wire [N:0] c;               // internal carries

    genvar i;                   // used only in generate

    // instantiate N full adders
    generate
    for (i=0; i<N; i=i+1)
        FullAdder FA (c[i+1], s[i], a[i], b[i], c[i]);
    endgenerate

    // cin and cout
    assign c[0] = cin;
    assign cout = c[N];

endmodule
```

This module exploits the regularity of instantiating additional bits in the adder by using a **for** loop and defining the number of loops by means of **parameter**. In this code **genvar** is used to declare the index variable used in the for loop. The syntax of the for loop is the same as that used in C. It will instantiate the full adders

```
FullAdder FA (c[1], s[0], a[0], b[0], c[0]);
FullAdder FA (c[2], s[1], a[1], b[1], c[1]);
                  .
                  .
                  .
FullAdder FA (c[N], s[N-1], a[N-1], b[N-1], c[N-1]);
```

The **parameter** value can be over ridden when this module is instantiated in a higher level module. Suppose for example we want to instantiate a 16-bit adder. We would then instantiate this module as

```
RippleCarryAdderN #(16) (s, cout, a, b, cin);
```

1.3 Behavioral Modeling

Verilog behavioral models use procedural statements to determine activity. Behavioral models allow the implementation of procedural algorithms and more like coding in C than either structural or dataflow modeling. There are two types of constructions used with procedural modeling.

An **initial** construction is used for a one-pass activity flow. The activity flow for an initial construction is a sequence of procedural statements which begin activity at the start of the simulation and do not repeat. An **always** construction is used for cyclic behavior. The activity flow starts at the beginning of the simulation and repeats until the simulation ends. Data elements which are updated in procedural blocks must be defined as variable type data.

An example of an **initial** statement used to initialize three variables

```
reg a, b, c;

initial
begin
   a = 1'b0;
   b = 1'b1;
   c = 1'b0;
end
```

This initializes `a`, `b`, and `c` at the beginning of the simulation.

An example of an **always** statement used to create a clock signal,

```
reg clock;

always
   #5 clock = ~clock;
```

This always statement will execute the single statement used to update the variable clock repeatedly. The 5 unit time delay will cause it to wait 5 time units each time it updates clock with its complement. The effect is to toggle the value of clock every 5 time units which creates a 10 time unit period for clock.

A behavioral model of the 2-input multiplexer using an always statement is the

```
module mux2 (c, a, b, s);
   output c
   input a, b;
   input s;

   reg c;

   always @(a, b, s)
   if (s == 0)
      c = a;
   else
      c = b;

endmodule
```

Here the always construction is used for a behavioral model, execution of activity, in this case the if-else statement, is controlled by a *sensitivity list*. We indicate the sensitivity list by the @ symbol, when the value of an element in the sensitivity list changes the always construction proceeds to execute the if-else statement. Note that this always statement implements the Boolean function for the 2-input multiplexer.

More generally we can use an always construction to create a model of combinational logic; such behavioral models will put the inputs to the logic in the sensitivity list and use procedural statements to determine the outputs. Note that the elements in the sensitivity list may be either nets or variables. The outputs of the logic which are updated in the always construction must be declared as variables.

As another example consider a behavioral model of a full adder. Using an always construction with procedural statements we might have the following,

```
module FullAdder (carryout, sum, a, b, carryin);
```

```
      output carryout, sum;
      input a,b,carryin;

      reg carryout, sum;

      always @(a, b, carryin)
      begin
         sum = a^b^carryin;
         carryout = (a&b)|(carryin&a)|(carryin&b);
      end

endmodule
```

Procedural constuctions are usually used in the verification of Verilog modules. A *testbench* is a Verilog code which is used to test a Verilog module. You can think of a testbench as analogous to the electronic test instruments you use in the lab. In a testbench the module to be tested is instantiated, a set of test inputs, or stimulus is then applied to this instantiated module and the outputs, or response, are monitored and compared to expected results. Whenever you code a module it is a good idea to construct a testbench.

An example of a testbench which could be used to functionally test the 2-input multiplexer is the following

```
// Test Bench for the mux2
//
   `include "mux2s.v"
//
module mux2_test;

   reg in0,in1,sel;
   wire out;

// instantiate 2-to-1 multiplexer

   mux2 MUX(out,in0,in1,sel);

// generate test signals

   initial
   begin
      {in1,in0,sel}=3'b000;
      #80 $finish;
   end
```

```
        always
           #10 {in1,in0,sel}={in1,in0,sel}+1;

     // output result

        initial
           $monitor($time, " out = %b", out, " in1 in0 sel = %b",
in1, in0, sel);

     endmodule
```

For this testbench note that there is no port list, all I/O is done within the module. The module has three main parts. The first part is the instantiation of the component module to be tested. This is sometimes referred to as the unit under test. The second part is the input to the module. In this example all the possible combinations of inputs are generated and applied in turn to the module being tested. Another method we could have used would be to create a file with all possible inputs, read these into the module, and then use them as stimulus for the instantiated module. Finally output and save the response of the tests.

At the beginning of the module we use the `input compiler directive. This will read in Verilog module with the model of the multiplexer. Our testbench has no need for a port list so it is omitted. All the nets and variables used in the testbench are then declared and the multiplexer is instantiated. Two procedural constructions are used to generate the test data. The first **initial** construction is used to initialize the three variables, in0, in2, and sel, which are to be input to the multiplexer, and uses the system task **$finish** to terminate the simulation. The **always** construction is used to change the values of the variables every 10 time units. This is done by counting through the possible input values and then waiting 10 time units to let the output settle. A **$monitor** system task in the second **initial** statement outputs the values of simulation time, the inputs, and the corresponding output, whenever any of them change.

References

IEEE Standard for Verilog Hardware Description Language, IEEE Std 1364 – 2005, IEEE Computer Society, 2006

S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd Edition, Prentice-Hall, 2003

C. H. Roth, and L. L. Kinney, *Fundamentals of Logic Design,* 7th Edition, Cengage Learning, 2013.

S. Sutherland, S. Davidmann, and P. Flake, S*ystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*, 2nd Edition, Springer, 2006

S. Sutherland, *Verilog is Not Called Verilog Anymore! The Merging of the Verilog and SystemVerilog IEEE Standards*, Sutherland HDL, Inc. 25 February 2008

D. Thomas and P. Moorby, *The Verilog Hardware Description Language*, 5th Edition, Springer 2008

Xilinx, *Synthesis and Simulation Design Guide*, ug262 9 (v14.4) December 12, 2012