Verilog Reference

J Taylor and S Winberg

revised:
05 June 2020

Contents

Unless otherwise noted, this reference is based on Appendix A, S Brown and Z Vranesic, 2014, *Fundamentals of Digital Logic with Verilog Design*, 3rd ed, McGraw-Hill. This document is not intended to be a comprehensive Verilog manual, but rather a reference for the features of Verilog typically used for circuit synthesis and will hopefully help to speed up the process of learning the language while actively working on lab practical assignments.

1 Verilog Coding Style

The tendency for the novice is to write Verilog code that resembles a computer program, containing many variables and loops. It is difficult to determine what logic circuit the CAD tools will produce when synthesising such code.

In general, synthesis tools have to recognise certain structures in code. From the practical point of view, this will only work if users write code that conforms to a commonly used style. A good approach is to "write Verilog code that obviously represents the intended circuit."

2 Comments in Verilog Code

In-line documentation can be included in Verilog code by means of writing a comment. A short comment begins with the double slash, `//`, and continues to the end of the line. A long comment can span multiple lines and is contained inside the delimiters `/*` and `*/`. Examples of comments are

```
// This is a short comment
/* This is a long Verilog comment
   that spans two lines */
```

For more professional documentation, one can make use of Doxygen. Verilog support can be added by means of an extension.

3 Signals and Constants

In Verilog, a signal in a circuit is represented as a *net* or a *variable* with a specific type. The term *net* is derived from the electrical jargon, where it refers to the interconnection of two or more points in a circuit. A signal declaration has the form:

`type [range] signal_name{, signal_name};`

The square brackets indicate an optional field, and the curly brackets indicate that zero or more additional entries are permitted. The `signal_name` is an identifier, as defined in the next section. Without the `range` field the declared net or variable is scalar and represents a single-bit signal. The range is used to specify vectors that correspond to multibit signals, as explained in section 3.2.

Verilog defines a number of types of nets and variables. These types are defined by the language itself, and user-defined types are not permitted. SystemVerilog, however, supports user-defined types, including structures, enumerations and others. More about this in section 9.

3.1 Identifier Names

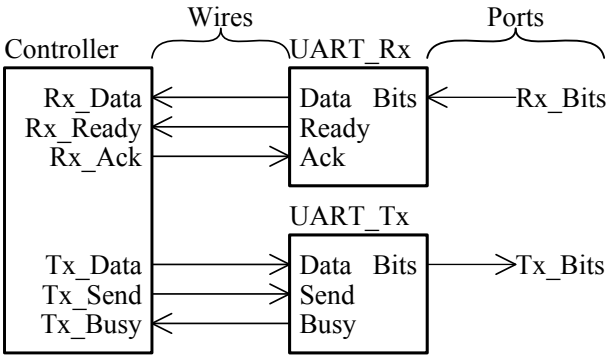
Identifiers are the names of variables and other elements in Verilog code. The rules for specifying identifiers are simple: any letter or digit may be used, as well as the _ (underscore) and $ characters. There are two caveats: an identifier must not begin with a digit and it should not be a Verilog keyword. Examples of legal identifiers are `f`, `x1`, `x_y`, and `Byte`. Some examples of illegal identifiers are `1x`, `+y`, `x*y`, and `258`. Verilog is case sensitive, hence `k` is not the same as `K`, and `BYTE` is not the same as `Byte`.

For special purposes, Verilog allows a second form of identifier, called an escaped identifier. Such identifiers begin with the (\) backslash character, which can then be followed by any printable ASCII characters except white spaces. Examples of escaped identifiers are `\123`, `\sig-name`, and `\a+b`. Escaped identifiers should not be used in normal Verilog code; they are intended for use in code produced automatically when other languages are translated into Verilog.

3.2 Nets

A net represents a node in a circuit. To distinguish between different types of circuit nodes there exist several types of nets, called *wire*, *tri*, and a number of others that are not required for synthesis.

The diagram below shows how a controller module might connect to two UART modules. Every wire has to have a name, and can connect one *output* port to many *input* ports, or many *inout* ports.



The *wire* type is employed to connect an output of one logic element in a circuit to an input of another logic element. The following are examples of scalar *wire* declarations.

```
wire UART_Tx_Send;
wire UART_Tx_Busy;
```

A vector *wire* represents multiple nodes, such as

```
wire [7:0]UART_Tx_Data;
wire [1:2]Array;
wire [3:0]S;
```

The square brackets are the syntax for specifying a vector's range. The range `[Ra:Rb]` can be either increasing or decreasing, as shown. In either case, `Ra` is the index of the most significant (leftmost) bit in the vector, and `Rb` is the index of the least-significant (rightmost) bit. The indices `Ra` and `Rb` can be either positive or negative integers.

The net `S` can be used as a four-bit quantity, or each bit can be referred to individually as `S[3]`, `S[2]`, `S[1]`, and `S[0]`. If a value is assigned to `S` such as `S = 4'b0011`, the result is `S[3] = 0`, `S[2] = 0`, `S[1] = 1`, and `S[0] = 1`.

The assignment of a single bit in a vector to another net, such as `f = S[0]`, is called a bit-select operation. A range of values from one vector can be assigned to another vector, which is called a part-select operation. The assignment `Array = S[2:1]` produces `Array[1] = S[2]` and `Array[2] = S[1]`. The index used in a bit-select operation can involve a variable, such as `S[n]`, while the indices used with a part-select operation have to be constant expressions, such as `S[2:1]`.

The *tri* type denotes circuit nodes that are connected in a tri-state fashion. These nets are treated in the same manner as the *wire* type, and they are used only to enhance the readability of code that includes tri-state gates.

3.3 Variables

Nets provide a means for interconnecting logic elements, but they do not allow a circuit to be described in terms of its behaviour. For this purpose, Verilog provides variables. A variable can be assigned a value in one Verilog statement, and it retains this value until it is overwritten in a subsequent assignment statement. There are two types of variables, *reg* and *integer*. Consider the code fragment

```
reg [2:0]Count;
integer  k;

Count = 0;
for (k = 0; k < 4; k = k+1) begin
 if (S[k]) Count = Count + 1;
end
```

The *for* and *if* statements are described in section 6. This code stores in `Count` the number of bits in `S` that have the value 1. Since it models the behaviour of a circuit, `Count` has to be declared as a variable, rather than a simple *wire*.

The keyword *reg* does not denote a storage element, or register. In Verilog

code, *reg* variables can be used to model either combinational or sequential parts of a circuit. In this example, the variable `k` serves as a loop index. Such variables are declared as type *integer*. Integer variables are useful for describing the behaviour of a module, but they do not directly correspond to nodes in a circuit.

3.4 Memories

A memory is a two-dimensional array of bits. Verilog allows such a structure to be declared as a variable (reg or integer) that is an array of vectors, such as

```
reg [7:0]R[3:0];
```

This statement defines `R` as four eight-bit variables named `R[3]`, `R[2]`, `R[1]`, and `R[0]`. Two-level indexing, such as `R[3][7]`, can be used. There is also support for higher dimension arrays. A three-dimensional array may be declared as

```
reg [7:0]R[3:0][1:0];
```

3.5 Constants

Verilog signals can have four possible values:

`0` = logic value 0
`1` = logic value 1
`Z` = tri-state (high impedance)
`X` = unknown value / don't care

The `Z` and `X` values can also be denoted by the small letters `z` and `x`. The value `X` can be used to denote a "don't-care" condition. The question-mark symbol `?` can also be used for this purpose.

Signals can be scalar (1-bit) or vector (multiple bits). Constants in Verilog take the following form:

`[size]['radix]constant`

where `size` is the number of bits in the `constant`, and `radix` is the number base. Supported radices are

`d` = decimal
`b` = binary
`h` = hexadecimal
`o` = octal

When no `radix` is specified, the default is decimal. If no `size` is specified, the default is 32. Hexadecimal digits can be lower or upper case.

If `size` specifies more bits than are required to represent the given constant, then in most cases the `constant` is padded with zeros on the most-significant side. The exception to this rule is when the first character of the constant is either `x` or `z`, in which case the padding is done using that value.

Constants can also be specified by ASCII text, which takes a different form. Some examples of constants include:

`0`	the number 0
`10`	the decimal number 10
`'b10`	the binary number $10 = (2)_{10}$
`'h10`	the hexadecimal number $10 = (16)_{10}$
`4'b100`	the binary number $100 = (4)_{10}$
`4'bx`	an unknown 4-bit value `xxxx`
`8'b1000_0011`	`_` can be inserted for readability
`8'hFX`	equivalent to `8'b1111_xxxx`
`"j"`	the 8-bit number $(6\mathrm{A})_{16} = (106)_{10}$

3.6 Parameters

A parameter associates an identifier name with a constant. Let the Verilog code include the following declarations:

```
parameter N  = 4;
parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
```

then the identifier `N` can be used in place of the number 4, the name `S0` can be substituted for the value `2'b00`, and so on. An important use of parameters is in the specification of parametrised sub-circuits, which is described in section 5.

Local constants can be defined by using the *localparam* keyword. This is especially useful when defining states for a state machine, as in the following example:

```
localparam Start    = 2'd0;
localparam Do_Stuff = 2'd1;
localparam End      = 2'd2;
```

The difference between *parameter* and *localparam* is that the former can be modified upon module instantiation, whereas the latter cannot. Put differently, a *parameter* is visible, and therefore modifiable, from outside the module it is defined in, whereas a *localparam* is only visible from within the module in which it is defined. More on modules in section 5.

4 Concurrent Statements

In any hardware description language, including Verilog, the concept of a concurrent statement means that the statement is considered in parallel with all the other concurrent statements and the ordering of these statements in the code does not matter.

4.1 Operators

Verilog has a large number of operators, as shown in table 1. The table uses operands named `A`, `B` and `C`, which may be either vectors or scalars. The syntax `~A` means that the `~` operator is applied to the variable `A`, and the syntax L(A) means that the result has the same number of bits (length) as in `A`. `N` and `M` are integer constants.

The table is in order of descending precedence. Operators with equal precedence are grouped together. The table has been adapted from https://www.utdallas.edu/~kad056000/index_files/verilog/operators.html.

Verilog has no boolean type as such. All boolean results are actually scalar signals, and can be used by any operator that accepts scaler signal input.

The bit-select, part-select and concatenate operators are all allowed on the left-hand side of an assignment as well. The operators `/`, `%`, `===` and `!==` are not synthesisable and only meaningful during simulation.

4.2 Continuous Assignments

Continuous assignments permit the description of a circuit's function. The general form of this statement is

assign `net_assignment{, net_assignment};`

The `net_assignment` can be any expression involving the operators listed in table 1. Multiple assignments can be specified in one assign statement, using commas to separate the assignments, as in

```
assign Cout = (x & y) | (x & Cin) | (y & Cin),
       s    = x ^ y ^ z;
```

It is possible to combine a continuous assignment with a wire declaration. For example, the sum, `s`, and carry-out, `c`, of a half-adder could be defined as

```
wire s = x ^ y,
     c = x & y;
```

Table 1: Verilog operators and bit lengths

Operator	**Purpose**	**Result length**
`A[N]`	Bit select	1
`A[N:M]`	Part select	|N−M|+1
`(...)`	Grouping	
`!A`	Logical not	1
`~A`	Bitwise not	L(A)
`&A`	and Reduction	1
`~&A`	nand Reduction	1
`|A`	or Reduction	1
`~|A`	nor Reduction	1
`^A`	xor Reduction	1
`^~A` or `~^A`	xnor Reduction	1
`+A`	No effect	L(A)
`-A`	2's Compliment	L(A)
`{A, ..., B}`	Concatenate	L(A)+...+L(B)
`{N{A}}`	Replicate	N×L(A)
`A * B`	Multiply	L(A)+L(B)
`A / B`	Divide	L(A)−L(B)
`A % B`	Modulus	L(B)
`A + B`	Add	Max(L(A), L(B))
`A - B`	Substract	Max(L(A), L(B))
`A << B`	Shift left	L(A)
`A >> B`	Shift right	L(A)
`A > B`	Greater than	1
`A < B`	Less than	1
`A >= B`	Greater or equal	1
`A <= B`	Less or equal	1
`A == B`	Equal	1
`A != B`	Not equal	1
`A === B`	Equivalent	1
`A !== B`	Not equivalent	1
`A & B`	Bitwise and	Max(L(A), L(B))
`A ^ B`	Bitwise xor	Max(L(A), L(B))
`A ^~ B` or `A ~^ B`	Bitwise xnor	Max(L(A), L(B))
`A | B`	Bitwise or	Max(L(A), L(B))
`A && B`	Logical and	1
`A || B`	Logical or	1
`A ? B : C`	Conditional	Max(L(B), L(C))

An example of a multibit assignment is

```
wire [1:3] A, B, C;
assign C = A & B;
```

The arithmetic assignment

```
wire [3:0] X, Y, S;
assign S = X + Y;
```

represents a four-bit adder without carry-in or carry-out. If carry-in and carry-out signals are declared,

```
wire carryin, carryout;
```

then the statement

```
assign {carryout, S} = X + Y + carryin;
```

represents the four-bit adder with carry-in and carry-out. Verilog treats the wire type as an unsigned number. Since a five-bit result is needed in `{carryout, S}`, each operand is padded with a zero. When using Verilog for synthesis, it is up to the compiler to determine, or infer, that a four-bit adder with carry-out is required and to recognize the carry-in.

Some compilers extend the result, not the inputs, so it is advisable to manually extend at least one of the input signals in order to ensure that the `carryout` signal is assigned properly:

```
assign {carryout, S} = {1'b0, X} + Y + carryin;
```

A complete example of arithmetic assignments is given below.

```
wire [3:0] X, Y,
wire [7:0] S, S2s

assign S   = X + Y,
       S2s = {{4{X[3]}}, X} + {{4{Y[3]}}, Y};
```

There are two four-bit inputs, `X` and `Y`, and two eight-bit outputs, `S` and `S2s`. To produce the eight-bit sum `S = X + Y` the Verilog compiler automatically pads `X` and `Y` with four zeros.

The assignment to `S2s` shows how a signed (2's complement) result can be generated. The assignment to `S2s` uses the concatenate and replication operators to pad `X` and `Y` with four copies of their most-significant bit, thereby performing sign extensions.

The example above specifies an adder for four-bit numbers. This code could be generalised by introducing a parameter that sets the number of bits in the adder. Below is the code for an N-bit adder. The number of bits to be added is defined with the *parameter* keyword, introduced in section 3.6. The value of `N` defines the bit widths of `X`, `Y`, `S` and `S2s`.

```
parameter N = 4;

wire [  N-1:0] X, Y;
wire [2*N-1:0] S, S2s;

assign S   = X + Y,
       S2s = {{N{X[N-1]}}, X} + {{N{Y[N-1]}}, Y};
```

5 Modules and Subcircuits

A circuit or sub-circuit described with Verilog code is called a module. The general structure of a module declaration is presented below.

```
module module_name #(
  [parameter declarations]
)(
  [input declarations]
  [output declarations]
  [inout declarations]
);
  [wire or tri declarations]
  [reg or integer declarations]
  [function or task declarations]
  [assign continuous assignments]
  [initial block]
  [always blocks]
  [gate instantiations]
  [module instantiations]
endmodule
```

The module has a name, `module_name`, which can be any valid identifier, followed by an optional list of parameters and a list of ports. The term port is adopted from the electrical jargon, in which it refers to an input or output connection in an electrical circuit. The ports can be of type *input*, *output*, or *inout* (bidirectional), and can be either scalar or vector.

An example of a module declaration is

```
module MyModule(
 input           x, y, Cin,
 input      [3:0]X, Y,
 inout      [7:0]Bus,
 output reg [3:0]S,
 output          s, Cout
);
endmodule
```

The ports `Cout`, `s`, and `Bus` are nets in this example, while `S` is a variable. Any port used as a variable must be explicitly declared as such. A module may contain any number of net (*wire* or *tri*) or variable (*reg* or *integer*) declarations, and a variety of other types of statements that are described later in this reference.

Below is the declaration for a module `FullAdd`, which represents a full-adder circuit. The input port `Cin` is the carry-in, and the bits to be added are the input ports `x` and `y`. The output ports are the sum, `s`, and the carry-out, `Cout`. The functionality of the full-adder is described with logic equations preceded by the keyword *assign*, which is discussed in section 4.

```
module FullAdd(
 input  Cin, x, y,
 output s, Cout
);
 assign s    = x ^ y ^ Cin;
 assign Cout = (x & y) | (Cin & x) | (Cin & y);
endmodule
```

There is usually more than one way to describe a given circuit using Verilog. Another version of the `FullAdd` module, in which the functionality is specified by using the concatenate and addition operators, is given below:

```
module FullAdd(
 input  Cin, x, y
 output s, Cout
);
 assign {Cout, s} = x + y + Cin;
endmodule
```

The circuits generated from the two modules above are the same.

5.1 Using Sub-circuits

A Verilog module can be included as a sub-circuit in another module. For this to work, both modules must be defined in the same file or else the Verilog compiler must be told where each module is located (the mechanism for doing this varies from one compiler to the next).

The general form of a module instantiation statement is presented below:

```
module_name [#(parameter overrides)] instance_name (
  .port_name([expression]) {, .port_name([expression])}
);
```

The `instance_name` can be any legal Verilog identifier and the port connections specify how the module is connected to the rest of the circuit. The same module can be instantiated multiple times in a given design, provided that each instance name is unique. The `#(parameter overrides)` can be used to set the values of parameters defined inside the `module_name` module. This feature is discussed in section 5.3.

Each `port_name` is the name of a port in the sub-circuit, and each expression specifies a connection to that port. The syntax `.port_name` is provided so that the order of signals listed in the instantiation statement does not have to be the same as the order of the ports given in the module statement of the sub-circuit. In Verilog jargon, this is called "named port connections". If the port connections are given in the same order as in the sub-circuit, then `.port_name` can be omitted. This format is called "ordered port connections".

An example is presented below. It gives the code of a four-bit ripple-carry adder built using four instances of the `FullAdd` sub-circuit presented in section 5.

```
module Adder4(
 input  carryin,
 input  [3:0] X, Y,
 output [3:0] S,
 output carryout
);
 wire [3:1]C;

 FullAdd Stage0(carryin, X[0], Y[0], S[0], C[1]); // Ordered connections
 FullAdd Stage1(C[1]   , X[1], Y[1], S[1], C[2]); // Ordered connections
 FullAdd Stage2(C[2]   , X[2], Y[2], S[2], C[3]); // Ordered connections
 FullAdd Stage3( // Named connections
  .Cout(carryout),
  .s   (S[3]    ),
  .y   (Y[3]    ),
  .x   (X[3]    ),
  .Cin (C[3]    )
 );
endmodule
```

The `Adder4` module instantiates four copies of the `FullAdd` sub-circuit. In the first three instantiation statements, ordered port connections are used. The last instantiation statement gives an example of named port connections. The port connections used in the instantiation statements specify how the `FullAdd` instances are interconnected to create the `Adder4` module.

5.2 The Generate Block

The module `Adder4` above instantiates four copies of the `FullAdd` sub-circuit to form a four-bit ripple-carry adder. A natural extension of this code is to add a parameter that sets the number of bits required and then use a loop to instantiate the sub-circuits. This can be achieved with the *generate* construct. The *generate* construct has the simplified form

generate
  [for `loops`**]**
  [if-else `statements`**]**
  [case `statements`**]**
  `[instantiation statements]`
endgenerate

If a *for* loop is included in the generate block, the loop index variable has to be declared of type *genvar*. A *genvar* variable is similar to an *integer* variable, but it can only have positive values and it can only be used inside *generate* blocks.

Below is the definition of a `Ripple_g` module, which instantiates `N FullAdd` modules. Each instance generated in the *for* loop will have a unique instance name, produced by the compiler, based on the *for* loop label.

```
module Ripple_g #(parameter N = 4)(
 input         carryin,
 input  [N-1:0]X, Y,
 output [N-1:0]S,
 output        carryout
);
 wire [N:0]C;

 assign C[0]     = carryin;
 assign carryout = C[N];

 genvar n;
 generate
  for(n = 0; n < N; n = n+1) begin: Addbit
   FullAdd Stage(C[n], X[n], Y[n], S[n], C[n+1]);
  end
 endgenerate
endmodule
```

5.3 Sub-circuit Parameters

By default, the `Ripple_g` module above describes the same 4-bit adder as the `Adder4` module. When creating an instance of this module, however, one can choose how many bits the adder should be. Some examples of instances of the `Ripple_g` include:

```
// 4-bit adder
Ripple_g My4BitAdder(carryin, X, Y, S, carryout);

// 15-bit adder with ordered parameter override
Ripple_g #(15) My15BitAdder(carryin, X, Y, S, carryout);

// 128-bit adder with named parameter override
Ripple_g #(.N(128)) My128BitAdder(carryin, X, Y, S, carryout);
```

6 Procedural Statements

In addition to the concurrent statements described in section 4, Verilog provides procedural statements (also called sequential statements). Whereas concurrent statements are executed in parallel, procedural statements are evaluated in the order in which they appear in the code. Note that these statements are still executed in parallel, even though they are evaluated sequentially.

Verilog syntax requires that procedural statements be contained inside an *always* block. It has the form

always `@(sensitivity_list)`
[**begin**]
  [`procedural assignment statements`]
  [**if-else** `statements`]
  [**case** `statements`]
  [**while**, **repeat**, `and` **for** `loops`]
  [**task** `and` **function** `calls`]
[**end**]

When multiple statements are included in an *always* block, the *begin* and *end* keywords are required, otherwise these keywords can be omitted. The *begin* and *end* keywords are also used with other Verilog constructs. Statements delimited by *begin* and *end* are referred to as a begin-end block.

The `sensitivity_list` is a list of signals that directly affect the output results generated by the *always* block. In essence, the *always* block will be evaluated whenever any signal in the sensitivity list changes value.

A simple example of an `always` block is

```
always @(x, y) begin
 s = x ^ y;
 c = x & y;
end
```

Since the output variables `s` and `c` depend on `x` and `y`, these signals are included in the sensitivity list, separated by a comma. When specifying a combinational circuit by using an `always` block, it is also possible to simply write

```
always @(*)
```

which indicates that all input signals used in the always block are included in the sensitivity list.

For simulation purposes, Verilog also provides the `initial` construct. The `initial` and `always` constructs have the same form, but the statements inside the `initial` construct are executed only once, at the start of a simulation. This is not meaningful for synthesis.

A Verilog module may include several `always` blocks, each representing a part of the circuit being modelled. Each entire `always` block can be considered as a concurrent statement.

Any signal assigned a value inside an `always` block has to be a variable of type `reg` or `integer`. A value is assigned to a variable with a procedural assignment statement. There are two kinds of assignments: blocking assignments, denoted by the `=` symbol, and non-blocking assignments, denoted by the `<=` symbol. The term blocking means that the assignment statement completes and updates its left-hand side before the subsequent statement is evaluated. More on these in section 6.6.

You can also find more examples on the web at http://www.asic-world.com/verilog/vbehave1.html.

6.1 The If-Else Statement

The general form of the if-else statement is given below.

```
if(expression1) begin
 // statements;
end else if(expression2) begin
 // statements;
end else begin
 // statements;
end
```

If `expression1` is true, then the first statement is evaluated. When multiple

statements are involved, they have to be included inside a begin-end block.

The *else if* and *else* clauses are optional. Verilog syntax specifies that when *else if* or *else* are included, they are paired with the most recent unfinished *if* or *else if*.

An example of an *if-else* statement used for combinational logic is

```
always @(w0, w1, s) begin
 if (s == 0) f = w0;
 else        f = w1;
end
```

which defines a 2-to-1 multiplexer with data inputs `w0` and `w1`, select input `s`, and output `f`.

6.2 Statement Ordering

Another way of describing the 2-to-1 multiplexer with an *if-else* statement is presented below.

```
always @(*) begin
 f = w0;
 if(s == 1) f = w1;
end
```

Instead of using an *else* clause, this code first makes the default assignment `f = w0` and then changes this assignment to `f = w1` if `s` has the value 1. The Verilog semantics specify that a signal assigned multiple values in an *always* construct retains the last assignment.

When the default assignment `f = w0` is removed, the compiler will infer a latch in order to remember the previous value of `f` when `s == 0`. See section 8.1 for more information regarding implied memory.

6.3 The Case Statement

The form of a *case* statement is illustrated below.

```
case(expression)
 alternative1: statement;
 alternative2: statement;
 default:      statement;
endcase
```

The bits in `expression`, called the controlling expression, are checked for a match with each alternative. The first successful match causes the associated statements to be evaluated. Each digit in each alternative is compared for an

exact match of the four values 0, 1, x, and z. A special case is the `default` clause, which takes effect if no other alternative matches.

The Verilog `case` statement differs from the equivalent C switch statement in that the alternatives do not need to be constants. It is valid to use a constant `expression` with variable alternatives, or even a variable `expression` with variable alternatives.

An example of a case statement is

```
always @(*) begin
 case (s)
  1'b0: f = w0;
  1'b1: f = w1;
 endcase
end
```

This code represents the same 2-to-1 multiplexer described in section 6.1. When using Verilog for simulation, it is necessary to give alternatives for all possible valuations of the controlling expression. A default has to be included for any valuations not explicitly covered by the listed alternatives. In this example, `s` can have the four values 0, 1, x, or z; hence, a default should be included to handle the cases where `s` is x or z. The default clause has not been included here because synthesis tools require only the bit values 0 and 1 to be considered.

The example below demonstrates the use of a `case` statement to specify truth tables. This code represents the same full-adder that is described in section 4.

```
// Full adder
module FullAdd(
 input  Cin, x, y,
 output reg s, Cout
);
 always @(*) begin
  case({Cin, x, y})
   3'b000: {Cout, s} = 2'b00;
   3'b001: {Cout, s} = 2'b01;
   3'b010: {Cout, s} = 2'b01;
   3'b011: {Cout, s} = 2'b10;
   3'b100: {Cout, s} = 2'b01;
   3'b101: {Cout, s} = 2'b10;
   3'b110: {Cout, s} = 2'b10;
   3'b111: {Cout, s} = 2'b11;
  endcase
 end
endmodule
```

The case statement is also important for representing some types of sequential circuits, such as finite state machines, which are discussed in section 8.

6.4 Casex and Casez Statements

In the `case` statement, the values `X` or `Z` in an alternative are checked for an exact match with the same values in the controlling expression. The `casez` statement adds more flexibility, by treating a `Z` digit in an alternative as a don't-care condition. The `casex` statement treats both `X` and `Z` as don't cares.

The alternatives do not have to be mutually exclusive. If they are not, then the first matching item has priority. The example below shows how `casex` can be used to describe a priority encoder with the 4-bit input `W` and the outputs `Y` and `f`.

```
module Priority(
 input      [3:0] W,
 output reg [1:0] Y,
 output f
);
 assign f = (|W);
 always @(W) begin
  casex(W)
   'b1xxx : Y = 2'd3;
   'b01xx : Y = 2'd2;
   'b001x : Y = 2'd1;
   default: Y = 2'd0;
  endcase
 end
endmodule
```

6.5 Loops in Verilog

Verilog includes four types of loop statements: `for`, `while`, `repeat` and `forever`. Synthesis tools typically support the `for` loop, which has the general form

```
for(initial_index; terminal_index; increment) statement;
```

The `initial_index` is evaluated once, before the first loop iteration, and typically performs the initialisation of the integer loop control variable, such as `k = 0`. In each loop iteration, the `statement` is performed, and then the `increment` statement is evaluated. A typical `increment` statement is `k = k + 1`. Finally, the `terminal_index` condition is checked, and if it is true (1), then another loop iteration is done. For synthesis, the `terminal_index` condition has to compare the loop index to a constant value, such as `k < 8`.

It is worth mentioning at this point that a loop in Verilog (or any HDL for that matter) is very different to a loop in a sequential programming language. The loop describes a combinational circuit, i.e. it does not cause the statements to run sequentially on different clock cycles. Loops can generate a very large circuit with little code, so care is required when using them.

An example of using a *for* loop to describe an N-bit ripple-carry adder is presented below.

```
module Ripple #(
 parameter N = 4
)(
 input              carryin,
 input      [N-1:0] X, Y,
 output reg [N-1:0] S,
 output reg         carryout
);
 reg [N:0] C;
 integer   n;

 always @(*) begin
  C[0] = carryin;
  for(n = 0; n < N; n = n+1) begin
   S[n  ] =  X[n] ^ Y[n] ^ C[n];
   C[n+1] = (X[n] & Y[n]) (C[n] & X[n]) (C[n] & Y[n]);
  end
  carryout = C[N];
 end
endmodule
```

The effect of the loop is to repeat its begin-end block for the specified values of `n`. In this example, each loop iteration defines an instance of a full-adder.

A second for-loop example is given below.

```
module BitCount #(
 parameter N     = 4,
 parameter log_N = 2
)(
 input      [N-1  :0] X,
 output reg [log_N:0] Count
);
 integer n;

 always @(*) begin
  Count = 0;
  for(n = 0; n < N; n = n+1) Count = Count + X[n];
 end
endmodule
```

This code produces a count of the number of bits in the N-bit input `X` that have the value 1. Unrolling the loop, the first two iterations give

```
Count = Count + X[0];
Count = Count + X[1];
```

The first statement produces the value `Count = 0 + X[0]`. The second assignment then gives `Count = X[0] + X[1]` and so on for the other loop iterations. At the end of the loop,

```
Count = X[0] + X[1] + ... + X[n-1]
```

is obtained, which is the expression the synthesis tool will use in generating the circuit.

The general forms of the while, repeat and forever loops are shown below. The while loop has the same structure as the corresponding statement in the C language, and the repeat loop simply specifies a number of times to repeat its begin-end block. The forever loop loops endlessly and is only meaningful in simulation.

```
while(condition) begin
 statement;
end


repeat(constant_value) begin
 statement;
end
```

6.6 Blocking versus Non-blocking Assignments for Combinational Circuits

Consider the blocking assignments

```
S = X + Y;
p = S[0];
```

The first statement sets `S` using the current values of `X` and `Y`, and then the second statement sets `p` according to this new value of `S`. Verilog also provides non-blocking assignments, specified as

```
S <= X + Y;
p <= S[0];
```

In this case, the statements are still evaluated in order, but they both use the values of variables that exist at the start of the evaluation. The first statement determines a new value for `S` based on the current values of `X` and `Y`, but `S` is not changed to this new value until all statements in the associated always block have been evaluated. Therefore, the value of `p` is based on the value of S before `X` or `Y` changed.

All the previous examples of combinational circuits used blocking assignments, which is a good way to design such circuits. A natural question is whether combinational circuits can be described using non-blocking assignments. The answer is that this would work in many cases, but if subsequent assignments depend on the results of preceding assignments, non-blocking assignments can produce nonsensical combinational circuits.

As an example, consider changing the *for* loop in the previous section to use non-blocking assignments. For simplicity, assume that `N = 3`, so that the unrolled loop is

```
Count <= Count + X[0];
Count <= Count + X[1];
Count <= Count + X[2];
```

Since non-blocking assignments are involved, each subsequent assignment statement sees the starting value of `Count`, which is 0, rather than a new `Count` value produced by the previous statements. The *for* loop thus degenerates to

```
Count <= 0 + X[0];
Count <= 0 + X[1];
Count <= 0 + X[2];
```

When there are multiple assignments to the same variable in an *always* block, Verilog semantics specify that the variable retains its last assignment. Therefore, the code produces the wrong result `Count = X[2]`, instead of the intended `Count = X[0] + X[1] + X[2]`.

7 Functions and Tasks

A Verilog function provides the means to write code in modular fashion without the need for separate modules. It has the general form:

```
function [range | integer] function_name;
  [input declarations]
  [parameter, reg, integer declarations]
  begin
    statement;
  end
endfunction
```

A function is defined within a module, and it is called either in a continuous assignment statement or in a procedural assignment statement inside that module. A function can have more than one input, but it does not have an explicit output, because the function name itself serves as the output variable.

The code below shows a BCD to 7-segment decoder for 3 digits:

```
module Group_f(
 input      [11: 0] Digits,
 output reg [ 1:21] Lights
);
 function [1:7]LEDs;
  input [3:0]BCD;
  begin
   case(BCD)        // abcdefg
    4'd0   : LEDs = 7'b1111110;
    4'd1   : LEDs = 7'b0110000;
    4'd2   : LEDs = 7'b1101101;
    4'd3   : LEDs = 7'b1111001;
    4'd4   : LEDs = 7'b0110011;
    4'd5   : LEDs = 7'b1011011;
    4'd6   : LEDs = 7'b1011111;
    4'd7   : LEDs = 7'b1110000;
    4'd8   : LEDs = 7'b1111111;
    4'd9   : LEDs = 7'b1111011;
    default: LEDs = 7'bx;
   endcase
  end
 endfunction

 always @(*)
 begin
  Lights[ 1: 7] = LEDs(Digits[ 3:0]);
  Lights[ 8:14] = LEDs(Digits[ 7:4]);
  Lights[15:21] = LEDs(Digits[11:8]);
 end
endmodule
```

Another method of writing this code appears below. This code uses a Verilog task, which is similar to a function. While a function returns a value, a task does not. It has input and output variables, like a module.

```
module Group_f(
 input      [11: 0] Digits,
 output reg [ 1:21] Lights
);
 task LEDs;
  input  [3:0]BCD;
  output [1:7]LEDs;
  begin
   case(BCD)        // abcdefg
    4'd0   : LEDs = 7'b1111110;
    4'd1   : LEDs = 7'b0110000;
    4'd2   : LEDs = 7'b1101101;
    4'd3   : LEDs = 7'b1111001;
    4'd4   : LEDs = 7'b0110011;
    4'd5   : LEDs = 7'b1011011;
    4'd6   : LEDs = 7'b1011111;
    4'd7   : LEDs = 7'b1110000;
    4'd8   : LEDs = 7'b1111111;
    4'd9   : LEDs = 7'b1111011;
    default: LEDs = 7'bx;
   endcase
  end
 endtask

 always @(*)
 begin
  LEDs(Digits[ 3:0], Lights[ 1: 7]);
  LEDs(Digits[ 7:4], Lights[ 8:14]);
  LEDs(Digits[11:8], lights[15:21]);
 end
endmodule
```

Functions and tasks are not crucial for designing Verilog code, but they facilitate the writing of modular code without using separate modules. One advantage of functions and tasks is that they can be called from an `always` block, whereas these blocks are not allowed to contain instantiation statements. These features of Verilog become increasingly important as the size of the code being developed increases.

8 Sequential Circuits

While combinational circuits can be modelled with either continuous assignment or procedural assignment statements, sequential circuits can be described only with procedural statements.

8.1 A Gated D Latch

Below is the code for a gated D latch. The *if* statement specifies that Q should be set to the value of D whenever Clock is high. There is no *else* clause in the *if* statement, which implies that Q should retain its previous value when the *if* condition is not met.

```
module Latch(input D, input Clock, output reg Q);
 always @(*) begin
  if(Clock) Q = D;
 end
endmodule
```

8.2 Register

The code below shows how registers are described in Verilog. The *always* construct uses the special sensitivity list @(*posedge* Clock). This event expression tells the Verilog compiler that any *reg* variable assigned a value in the *always* construct is the output of a register.

```
module Register(input D, input Clock, output reg Q);
 always @(posedge Clock) Q <= D;
endmodule
```

This code generates a register, with input D and output Q, that is sensitive to the positive clock edge. A negative-edge sensitive register is specified by @(*negedge* Clock).

The signal D is not specified in the sensitivity list because it does not influence when the *always* block must be evaluated. The *always* block is only evaluated on the positive Clock edge.

8.3 Multi-bit Register

One possible approach for describing a multibit register is to create an entity that instantiates multiple registers. A more convenient method is illustrated below. The code describes a four-bit register with synchronous clear.

```
module Register(
 input           Clock, nReset,
 input      [3:0]D,
 output reg [3:0]Q
);
 always @(posedge Clock) begin
  if(!nReset) Q <= 0;
  else        Q <= D;
 end
endmodule
```

The code below shows how the above module can be extended to represent an N-bit register with an enable input, `E`.

```
module RegEn #(
 parameter N = 4;
)(
 input             Clock, nReset, E,
 input      [N-1:0]D,
 output reg [N-1:0]Q
);
 always @(posedge Clock) begin
  if     (!nReset) Q <= 0;
  else if( E     ) Q <= D;
 end
endmodule
```

The number of registers is set by the parameter `N`. When the active clock edge occurs, the registers cannot change their stored values if the enable `E` is low. If `E` is high, the register responds to the active clock edge in the normal way.

8.4 Shift Registers

An example of code that defines a three-bit shift register is provided below. The lines of code are numbered for ease of reference.

```
   // A three-bit shift register
1  module Shift3(
2   input           w, Clock,
3   output reg [1:3]Q
4  );
5   always @(posedge Clock) begin
6    Q[3] <= w;
7    Q[2] <= Q[3];
8    Q[1] <= Q[2];
9   end
10 endmodule
```

The shift register has a serial input, `w`, and parallel outputs, `Q`. The right-most bit in the register is `Q[3]`, and the left-most bit is `Q[1]`. Shifting is performed in the right-to-left direction. All assignments to `Q` are synchronized to the clock edge by the `@(posedge Clock)` event, hence `Q` represents the outputs of registers. The statement in line 6 specifies that `Q[3]` is assigned the value of `w`.

The semantics of the non-blocking assignments mean that the subsequent statements do not see the new value of `Q[3]` until the next time the *always* block is evaluated (in the following clock cycle). In line 7, the previous value of `Q[3]`, before it is shifted as a result of line 6, is assigned to `Q[2]`. Line 8 completes the shift operation by assigning the previous value of `Q[2]` to `Q[1]`, before it is changed as a result of line 7.

All registers change their values at the same time, as required in a shift register. The statements in lines 6 to 8 could be written in any order without altering

the meaning of the code.

An often more convenient way to write a shift register assignment is by using concatenation. Lines 6 to 8 can be replaced with

```
Q <= {Q[2:3], w};
```

8.4.1 Blocking Assignments for Sequential Circuits

Blocking assignments should not be used for sequential circuits. As an example of the semantics involved, consider changing the shift register above to blocking assignments:

```
Q[3] = w;
Q[2] = Q[3];
Q[1] = Q[2];
```

The first assignment sets `Q[3] = w`. Since blocking assignments are involved, the next statement sees this new value of `Q[3]` and therefore produces `Q[2] = Q[3] = w`. Similarly, the final assignment gives `Q[1] = Q[2] = w`. The code does not describe the desired shift register, but rather loads all registers with the value of the input `w`.

To avoid the confusing dependence on the ordering of statements, blocking assignments should be avoided when modelling sequential circuits. Also, because they imply differing semantics, blocking and non-blocking assignments should never be mixed in a single *always* construct.

8.5 Counters

The example below presents the code for a four-bit counter with a synchronous reset input. The counter also has an enable input, `E`. On the positive clock edge, if `E` is high, the count is incremented. If `E` is low, the counter retains its previous value.

```
module Count4(
 input           Clock, nReset, E,
 output reg [3:0]Q
);
 always @(posedge Clock) begin
  if     (!nReset) Q <= 0;
  else if(E      ) Q <= Q + 1;
 end
endmodule
```

8.6 Moore-Type Finite State Machines

Verilog code for a simple Moore machine is shown below.

```
module Moore(
 input  Clock, w, nReset,
 output z,
);
 reg [1:0]y, Y;                              // State variables
 localparam A = 2'b00, B = 2'b01, C = 2'b10; // State constants

 always @(*) begin // State transitions
  case(y)
   A: if (w == 0) Y = A;
      else        Y = B;
   B: if (w == 0) Y = A;
      else        Y = C;
   C: if (w == 0) Y = A;
      else        Y = C;
   default:       Y = 2'bxx;
  endcase
 end

 always @(posedge Clock) begin
  if(!nReset) y <= A;
  else        y <= Y;
 end

 assign z = (y == C);
endmodule
```

The two-bit vector `y` represents the present state of the machine, and the state codes are defined as parameters. Some CAD synthesis systems provide a means of requesting that the state assignment be chosen automatically, but the assignments has been specified manually in this example. The present state signal `y` corresponds to the outputs of the state registers, and the signal `Y` represents the inputs to the registers, which define the next state.

The code has two *always* blocks. The top one describes a combinational circuit and uses a *case* statement to specify the values that `Y` should have for each value of `y`. The other *always* block represents a sequential circuit, which specifies that `y` is assigned the value of `Y` on the positive clock edge. The *always* block also specifies that `y` should take the value `A` when `nReset` is low, which provides the synchronous reset.

Since the machine is of the Moore type, the output `z` can be defined by using the assignment statement `z = (y == C)` that depends only on the present state of the machine. This statement is provided as a continuous assignment at the end of the code, but it could alternatively have been given inside the top *always* block that represents the combinational part of the FSM.

This assignment statement cannot be placed inside the bottom *always* block. Doing so would cause `z` to be the output of a separate register, rather than a combinational function of `y`. This circuit would set `z` to 1 one clock cycle later

than required when the machine enters state C.

An alternative version of the code for the Moore machine is given below. This code uses a single `always` block to define both the combinational and sequential parts of the finite state machine.

```
module Moore(
 input  Clock, w, nReset,
 output z
);
 reg [1:0]y;
 localparam A = 2'b00, B = 2'b01, C = 2'b10;

 always @(posedge Clock) begin
  if(!nReset) y <= A;
  else case(y)
   A: if (w == 0) y <= A;
      else        y <= B;
   B: if (w == 0) y <= A;
      else        y <= C;
   C: if (w == 0) y <= A;
      else        y <= C;
   default:       y <= 2'bxx;
  endcase
 end

 assign z = (y == C);
endmodule
```

8.7 Mealy-Type Finite State Machines

A code for a simple Mealy machine is shown below. The code has the same structure as the Moore machine above, except that the output `z` is defined within the top *always* block.

```
module Mealy(
 input      Clock, w, nReset,
 output reg z
);
 reg        y, Y;
 localparam A = 1'b0, B = 1'b1;

 always @(*) begin
  case(y)
   A: if(w == 0) begin
       Y = A;
       z = 0;
      end else begin
       Y = B;
       z = 0;
      end
   B: if(w == 0) begin
       Y = A;
       z = 0;
      end else begin
       Y = B;
       z = 1;
      end
  endcase
 end

 always @(posedge Clock) begin
  if(!nReset) y <= A;
  else        y <= Y;
 end
endmodule
```

The *case* statement specifies that, when the FSM is in state `A`, `z` should be 0, but when in state `B`, `z` should take the value of `w`. Since the top always block represents a combinational circuit, the output `z` can change value as soon as the input `w` changes, as required for the Mealy machine.

9 SystemVerilog Extensions

Thus far, this reference described Verilog-2001, which is one of the most commonly used versions of Verilog. If the compiler supports SystemVerilog, however, one can make use of some useful extensions. A small subset of these are described in the following sub-sections.

This section has been adapted from http://en.wikipedia.org/wiki/SystemVerilog and http://www.asic-world.com/systemverilog/index.html.

9.1 Array Port Definitions

Ports can be defined as any type, including arrays and user-defined types. In Verilog-2001, it is illegal to define a module as

```
module Lights_Driver(
 input      [3:0]Digits[2:0],
 output reg [1:7]Lights[2:0]
);
 // BCD to 7-segment conversion here
endmodule
```

whereas in SystemVerilog it is perfectly valid.

9.2 Data Types

SystemVerilog define the following 2-state ('0' or '1') data types:

byte	8-bit signed integer
shortint	16-bit signed integer
int	32-bit signed integer
longint	64-bit signed integer
bit	2-state version of *wire*

4-state ('0', '1', 'X' and 'Z') data types include:

logic	variable type
reg	variable type
wire	net type
integer	32-bit signed integer
time	64-bit unsigned integer

The *reg* data type is not restricted to inside *always* blocks. A *reg* variable can be assigned a value by concurrent or procedural assignments. In order to make the code more readable, the keyword *logic* can be used instead, which is synonymous to *reg*.

9.3 Assignment Operators

SystemVerilog defines additional procedural assignment operators:

```
+=  -=  *=  /=  %=  &=  |=  ^=  <<=  >>=  ++  --
```

These are all blocking operators and operate in the same way as their counterparts in C. In addition, the procedural assignment operators `<=` and `=` can operate directly on arrays.

9.4 Extended Data Types

SystemVerilog allows data type definitions of the form

```
typedef struct packed{
 bit       Sign;
 bit [14:0]Exponent;
 bit [63:0]Mantissa;
} long_double;

long_double d, f;

assign d = 80'd0; // zero
assign f.Sign     = 0,
       f.Exponent = 15'd_16_394,
       f.Mantissa = 64'h81BE_72A9_16CD_9061;
```

These new types can then be used for any net, variable or port. The *packed* keyword specifies that the structure describes a bit-vector.

Other examples include:

```
typedef enum logic [2:0]{Red, Green, Blue, Cyan, Magenta, Yellow} COLOUR;
COLOUR Colour;
assign Colour = Green;


enum reg [1:0]{Start=2'd0, Init=2'd1, Idle=2'd2, Run=2'd3} State;
always @(posedge Clk) begin
 if(Reset) State <= Start;
 else      State <= Idle;
end
```

9.5 Procedural Clarity

The `always @(*)` statement can be replaced with `always_comb` or `always_latch`, depending on whether a combinational circuit or latch is intended. Similarly, the `always @(posedge Clk)` statement can be replaced with `always_ff @(posedge Clk)`. This has no other purpose than to make the code easier to read.

9.6 Interfaces

In large designs it is often difficult to connect modules together, simply because of the number of connections required. SystemVerilog makes this easier by means of an interface:

```
interface Interface(input wire Clk);
 logic [5:0]A;
 logic [5:0]B;
 logic [7:0]C;
 modport in (input Clk, input A, output reg B);
 modport out(input Clk, input B, output reg C);
endinterface

module A(Interface.in Port);
 always_ff @(posedge Port.Clk) Port.B <= Port.A + 5'd27;
endmodule

module B(Interface.out Port);
 always_ff @(posedge Port.Clk) Port.C <= Port.B * 2'd3;
endmodule

module Top_Level(input Clk_50MHz);
  Interface i(Clk_50MHz);
  A A1(i);
  B B1(i);
 endmodule
```

The ports `A`, `B` and `C` do not need to be declared in the `Top_Level` module. Interfaces can be quite complex things – for more information, see http://www.asic-world.com/systemverilog/interface1.html#Interfaces.

9.7 Improved Loops

Loops, such as `for` loops, support `break` and `continue` statements.

10 Pre-processor

10.1 Overview

Verilog has a pre-processor very similar to that of C or C++. The `define, `ifdef, `ifndef, `elsif, `else, `endif and `include directives function similarly to #define, #ifdef, #ifndef, #elif, #else, #endif and #include in C.

One notable difference, however, is in how macros are invoked. In C, the #define MyMacro Q[7] macro can be invoked as MyMacro. In Verilog, a macro defined as `define MyMacro Q[7], must be invoked as `MyMacro.

Use Verilog macro definitions sparingly, as the macro name-space is global. If you want to define a bunch of constants, use *localparam* instead.

See http://www.veripool.org/papers/Preproc_Good_Evil_SNUGBos10_paper.pdf for more details.

10.2 Example

Different platforms have different features. One might then have different versions of a module for ASIC and FPGA implementations, or even different FPGA devices. It would then be useful to declare the code in the following fashion:

```
// Contents of Global.vh
`ifndef Global_vh
`define Global_vh

// Choose platform
`define FPGA_Cyclone
//`define FPGA_Virtex
// ASIC otherwise

// Other global user types and constants goes here

`endif
```

```
// Contents of SomeVerilog.v
`include "Global.vh"

`ifdef FPGA_Cyclone
 Cyclone_Module MyModule(Reset, Clk, Data, Result);
`else
 `ifdef FPGA_Virtex
  Virtex_Module MyModule(Reset, Clk, Data, Result);
 `else // ASIC
  ASIC_Module MyModule(Reset, Clk, Data, Result);
 `endif
`endif
```

An alternative method, especially useful with longer lists, is:

```
// Contents of Global.vh
`ifndef Global_vh
`define Global_vh

// Choose platform (one-hot)
`define FPGA_Cyclone 0
`define FPGA_Stratix 0
`define FPGA_Virtex  1
`define FPGA_Spartan 0
`define ASIC         0

// Other global user types and constants goes here

`endif
```

```
// Contents of SomeVerilog.v
`include "Global.vh"

`if    FPGA_Cyclone
 Cyclone_Module MyModule(Reset, Clk, Data, Result);
`elsif FPGA_Stratix
 Stratix_Module MyModule(Reset, Clk, Data, Result);
`elsif FPGA_Virtex
 Virtex_Module MyModule(Reset, Clk, Data, Result);
`elsif FPGA_Spartan
 Spartan_Module MyModule(Reset, Clk, Data, Result);
`elsif ASIC
 ASIC_Module MyModule(Reset, Clk, Data, Result);
`else
 `error_unsupported_platform
`endif
```