Chapter 2: Verilog Constructs

This chapter begins looking at the basic construction of a Verilog module. The chapter begins by covering the built-in features of a Verilog module including the file structure, data types, operators, and declarations. It provides a foundation of Verilog that will lead to modeling examples provided in Chap. 3. The original Verilog standard (IEEE 1364) has been updated numerous times since its creation in 1995. The most significant update occurred in 2001, which was titled IEEE 1394-2001. In 2005, minor improvements were added to the standard, which resulted in IEEE 1394-2005. The constructs described in this book reflect the functionality in the IEEE 1394-2005 standard. The functionality of Verilog (e.g., operators, signal types, functions) is defined within the Verilog standard; thus, it is not necessary to explicitly state that a design is using the IEEE 1394 package because it is inherent in the use of Verilog.

Verilog is case sensitive. Also, each Verilog assignment, definition, or declaration is terminated with a semicolon (;). As such, line wraps are allowed and do not signify the end of an assignment, definition, or declaration. Line wraps can be used to make Verilog more readable. Comments in Verilog are supported in two ways. The first way is called a *line comment* and is preceded with two slashes (i.e., *//*). Everything after the slashes is considered a comment until the end of the line. The second comment approach is called a *block comment* and begins with */** and ends with a ***/*. Everything between /* and */ is considered a comment. A block comment can span multiple lines. All user-defined names in Verilog must start with an alphabetic letter, not a number. User-defined names are not allowed to be the same as any Verilog keyword. This chapter contains many definitions of syntax in Verilog. The following notations will be used throughout the chapter when introducing new constructs.

bold	= Verilog keyword, use as is, case sensitive.
italics	= User-defined name, case sensitive.
<>	= A required characteristic such as a data type, input/output, etc.

Learning Outcomes—After completing this chapter, you will be able to:

2.1 Describe the data types provided in Verilog.
2.2 Describe the basic construction of a Verilog module.

2.1 Data Types

In Verilog, every signal, constant, variable, and function must be assigned a *data type*. The IEEE 1394-2005 standard provides a variety of predefined data types. Some data types are synthesizable, while others are only for modeling abstract behavior. The following are the most commonly used data types in the Verilog language.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-030-10552-5_2

2.1.1 Value Set

Verilog supports four basic values that a signal can take on: 0, 1, X, and Z. Most of the predefined data types in Verilog store these values. A description of each value supported is given below.

Value	Description
0	A logic zero, or false condition.
1	A logic one, or true condition.
x or **X**	Unknown or uninitialized.
z or **Z**	High impedance, tri-stated, or floating.

In Verilog, these values also have an associated *strength*. The strengths are used to resolve the value of a signal when it is driven by multiple sources. The names, syntax, and relative strengths are given below.

Strength	Description	Strength level
supply1	Supply drive for V_{CC}	7
supply0	Supply drive for V_{SS}, or GND	7
strong1	Strong drive to logic one	6
strong0	Strong drive to logic zero	6
pull1	Medium drive to logic one	5
pull0	Medium drive to logic zero	5
large	Large capacitive	4
weak1	Weak drive to logic one	3
weak0	Weak drive to logic zero	3
medium	Medium capacitive	2
small	Small capacitive	1
highz1	High impedance with weak pull-up to logic one	0
highz0	High impedance with weak pull-down to logic zero	0

When a signal is driven by multiple drivers, it will take on the value of the driver with the highest strength. If the two drivers have the same strength, the value will be *unknown*. If the strength is not specified, it will default to *strong drive*, or level 6.

2.1.2 Net Data Types

Every signal within Verilog must be associated with a data type. A *net data type* is one that models an interconnection (aka, a *net*) between components and can take on the values 0, 1, X, and Z. A signal with a net data type must be driven at all times and updates its value when the driver value changes. The most common synthesizable net data type in Verilog is the *wire*. The type wire will be used throughout this text. There are also a variety of other more advanced net data types that model complex digital systems with multiple drivers for the same net. The syntax and description for all Verilog net data types are given below.

Type	Description
wire	A simple connection between components.
wor	Wired-OR. If multiple drivers, their values are OR'd together.
wand	Wired-AND'd. If multiple drivers, their values are AND'd together.
supply0	Used to model the V_{SS}, (GND), power supply (supply strength inherent).
supply1	Used to model the V_{CC} power supply (supply strength inherent).
tri	Identical to **wire**. Used for readability for a net driven by multiple sources.
trior	Identical to **wor**. Used for readability for nets driven by multiple sources.
triand	Identical to **wand**. Used for readability for nets driven by multiple sources.
tri1	Pulls up to logic one when tri-stated.
tri0	Pulls down to logic zero when tri-stated.
trireg	Holds last value when tri-stated (capacitance strength inherent).

Each of these net types can also have an associated *drive strength*. The strength is used in determining the final value of the net when it is connected to multiple drivers.

2.1.3 Variable Data Types

Verilog also contains data types that model storage. These are called *variable data types*. A variable data type can take on the values 0, 1, X, and Z, but does not have an associated strength. Variable data types will hold the value assigned to them until their next assignment. The syntax and description for the Verilog variable data types are given below.

Type	Description
reg	A variable that models logic storage. Can take on values 0, 1, X, and Z.
integer	A 32-bit, 2's complement variable representing whole numbers between $-2{,}147{,}483{,}648_{10}$ and $+2{,}147{,}483{,}647$.
real	A 64-bit, floating point variable representing real numbers between $-(2.2 \times 10^{-308})_{10}$ and $+(2.2 \times 10^{308})_{10}$.
time	An unsigned, 64-bit variable taking on values from 0_{10} to $+(9.2 \times 10^{18})$.
realtime	Same as **time**. Just used for readability.

2.1.4 Vectors

In Verilog, a *vector* is a one-dimensional array of elements. All of the net data types, in addition to the variable type reg, can be used to form vectors. The syntax for defining a vector is as follows:

```
<type> [<MSB_index>:<LSB_index>] vector_name
```

While any range of indices can be used, it is common practice to have the LSB index start at zero.

Example:

```
wire [7:0] Sum;    // This defines an 8-bit vector called "Sum" of type wire. The
                   // MSB is given the index 7 while the LSB is given the index 0.

reg [15:0] Q;      // This defines a 16-bit vector called "Q" of type reg.
```

Individual bits within the vector can be addressed using their index. Groups of bits can be accessed using an index range.

```
Sum[0];        // This is the least significant bit of the vector "Sum" defined above.
Q[15:8];       // This is the upper 8-bits of the 16-bit vector "Q" defined above.
```

2.1.5 Arrays

An *array* is a multidimensional array of elements. This can also be thought of as a "vector of vectors." Vectors within the array all have the same dimensions. To declare an array, the element type and dimensions are defined first followed by the array name and its dimensions. It is common practice to place the start index of the array on the left side of the ":" when defining its dimensions. The syntax for the creation of an array is shown below.

```
<element_type>  [<MSB_index>:<LSB_index>]  array_name  [<array_start_index>:
<array_end_index>];
```

Example:

```
reg[7:0] Mem[0:4095];  // Defines an array of 4096, 8-bit vectors of type reg.
integer A[1:100];      // Defines an array of 100 integers.
```

When accessing an array, the name of the array is given first, followed by the index of the element. It is also possible to access an individual bit within an array by adding appending the index of element

Example:

```
Mem[2];             // This is the 3rd element within the array named "Mem".
                    // This syntax represents an 8-bit vector of type reg.

Mem[2][7];          // This is the MSB of the 3rd element within the array named "Mem".
                    // This syntax represents a single bit of type reg.

A[2];               // This is the 2nd element within the array named "A". Recall
                    //  that A was declared with a starting index of 1.
                    // This syntax represents a 32-bit, signed integer.
```

2.1.6 Expressing Numbers Using Different Bases

If a number is simply entered into Verilog without identifying syntax, it is treated as an integer. However, Verilog supports defining numbers in other bases. Verilog also supports an optional bit size and sign of a number. When defining the value of arrays, the "_" can be inserted between numerals to improve readability. The "_" is ignored by the Verilog compiler. Values of numbers can be entered in either upper or lower case (i.e., b or B, f or F). The syntax for specifying the base of a number is as follows:

```
<size_in_bits>'<base><value>
```

Note that specifying the size is optional. If it is omitted, the number will default to a 32-bit vector with leading zeros added as necessary. The supported bases are as follows:

Syntax	Description
'b	Unsigned binary.
'o	Unsigned octal.
'd	Unsigned decimal.
'h	Unsigned hexadecimal.
'sb	Signed binary.
'so	Signed octal.
'sd	Signed decimal.
'sh	Signed hexadecimal.

Example:

```
10              // This is treated as decimal 10, which is a 32-bit signed vector.
4'b1111         // A 4-bit number with the value 1111₂.
8'b1011_0000    // An 8-bit number with the value 10110000₂.
8'hFF           // An 8-bit number with the value 11111111₂.
8'hff           // An 8-bit number with the value 11111111₂.
6'hA            // A 6-bit number with the value 001010₂. Note that leading zeros
                //  were added to make the value 6-bits.
8'd7            // An 8-bit number with the value 00000111₂.
32'd0           // A 32-bit number with the value 0000_0000₁₆.
'b1111          // A 32-bit number with the value 0000_000F₁₆.
8'bZ            // An 8-bit number with the value ZZZZ_ZZZZ.
```

2.1.7 Assigning Between Different Types

Verilog is said to be a weakly typed (or loosely typed) language, meaning that it permits assignments between different data types. This is as opposed to a strongly typed language (such as VHDL) where signal assignments are only permitted between like types. The reason Verilog permits assignment between different types is because it treats all of its types as just groups of bits. When assigning between different types, Verilog will automatically truncate or add leading bits as necessary to make the assignment work. The following examples illustrate how Verilog handles a few assignments between different types. Assume that a variable called ABC_TB has been declared as type reg[2:0].

Example:

```
ABC_TB = 2'b00;    // ABC_TB will be assigned 3'b000. A leading bit is automatically
                      added.
ABC_TB = 5;        // ABC_TB will be assigned 3'b101. The integer is truncated to
                      3-bits.
ABC_TB = 8;        // ABC_TB will be assigned 3'b000. The integer is truncated to
                      3-bits.
```

CONCEPT CHECK

CC2.1 The two most commonly used data types in Verilog are *wire* and *reg*? What is the fundamental difference between these types?

(A) They are the same because they can both take on 0, 1, X, or Z.

(B) A *wire* is a net data type, meaning that it must be driven at all times. A *reg* is a variable data type, meaning that it will hold its value after it is assigned.

(C) A *wire* can only take on values of 0 and 1 while a *reg* can take on 0, 1, X, or Z.

(D) They cannot drive one other.

2.2 Verilog Module Construction

A Verilog design describes a single system in a single file. The file has the suffix *.v. Within the file, the system description is contained within a **module**. The module includes the interface to the system (i.e., the inputs and outputs) and the description of the behavior. Figure 2.1 shows a graphical depiction of a Verilog file.

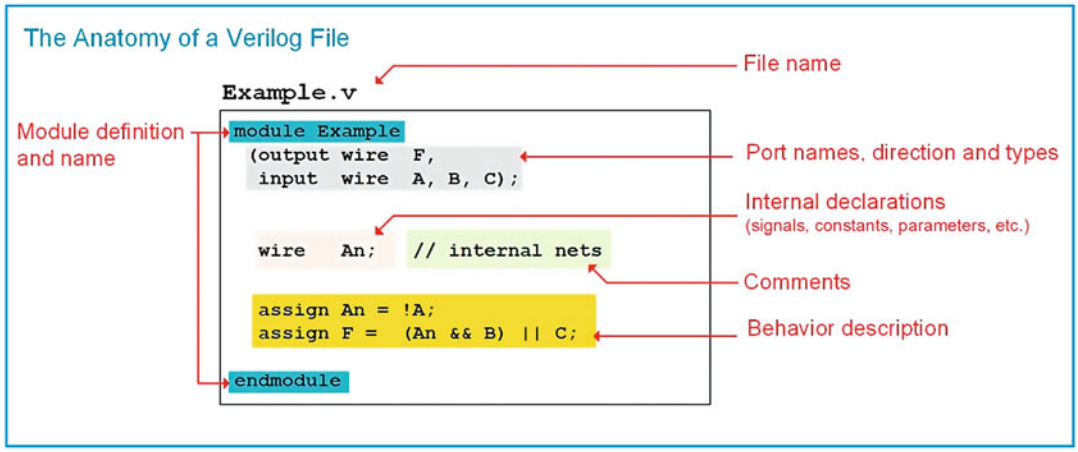


Fig. 2.1
The anatomy of a Verilog file

2.2.1 The Module

All systems in Verilog are encapsulated inside of a **module**. Modules can include instantiations of lower-level modules in order to support hierarchical designs. The keywords **module** and **endmodule** signify the beginning and end of the system description. When working on large designs, it is common practice to place each module in its own file with the same name.

```
module module_name (port_list);                          // Pre Verilog-2001
 // port_definitions
 // module_items
endmodule
```

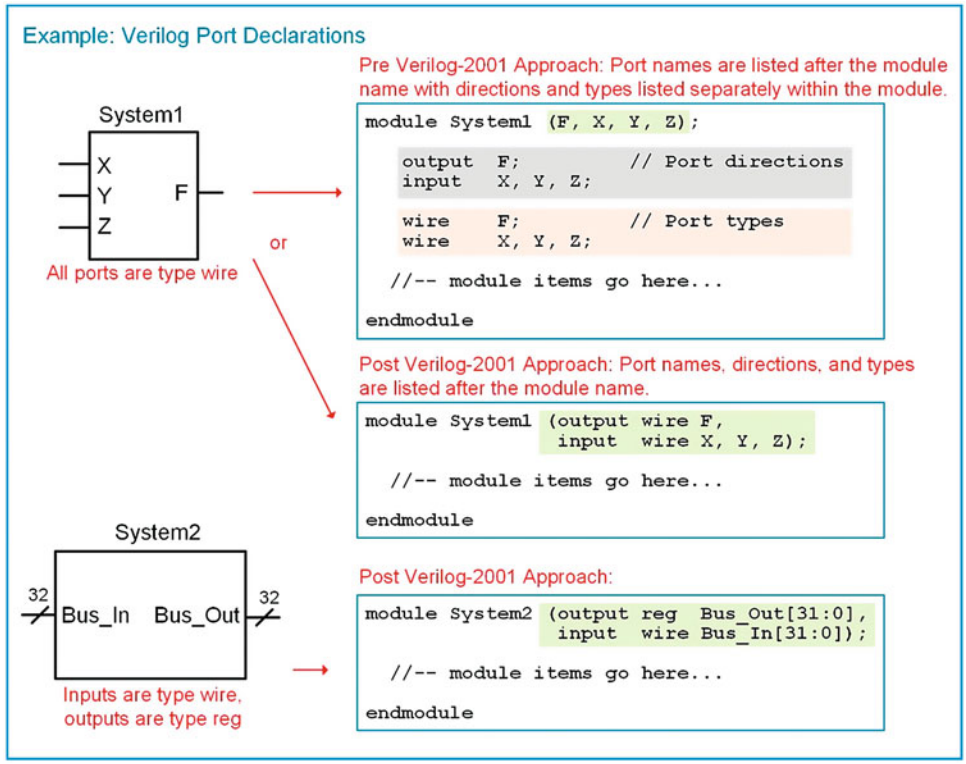
or

```
module module_name (port_list and port_definitions);     // Verilog-2001 and after
  // module_items
endmodule
```

2.2.2 Port Definitions

The first item within a module is its definition of the inputs and outputs, or ports. Each port needs to have a user-defined name, a direction, and a type. The user-defined port names are case sensitive and must begin an alphabetic character. The port directions are declared to be one of the three types: **input**, **output**, and **inout**. A port can take on any of the previously described data types, but only wires, registers, and integers are synthesizable. Port names with the same type and direction can be listed on the same line separated by commas.

There are two different port definition styles supported in Verilog. Prior to the Verilog-2001 release, the port names were listed within parentheses after the module name. Then within the module, the directionality and type of the ports were listed. Starting with the Verilog-2001 release, the port directions and types could be included alongside the port names within the parenthesis after the module name. This approach mimicked more of an ANSCI-C approach to passing inputs/outputs to a system. In this text, the newer approach to port definition will be used. Example 2.1 shows multiple approaches for defining a module and its ports.



Example 2.1
Declaring Verilog module ports

2.2.3 Signal Declarations

A signal that is used for internal connections within a system is declared within the module before its first use. Each signal must be declared by listing its type followed by a user-defined name. Signal names of like type can be declared on the same line separated with a comma. All of the legal data types described above can be used for signals; however, only types net, reg, and integer will synthesize directly. The syntax for a signal declaration is as follows:

```
<type> name;
```

Example:

```
wire    node1;          // declare a signal named "node1" of type wire
reg     Q2, Q1, Q0;     // declare three signals named "Q2", "Q1", and "Q0", all
                           of type reg
wire    [63:0] bus1;    // declare a 64-bit vector named "bus1" with all bits of type
                           wire
integer i,j;            // declare two integers called "i" and "j"
```

Verilog supports a hierarchical design approach, thus signal names can be the same within a subsystem as those at a higher level without conflict. Figure 2.2 shows an example of legal signal naming in a hierarchical design.

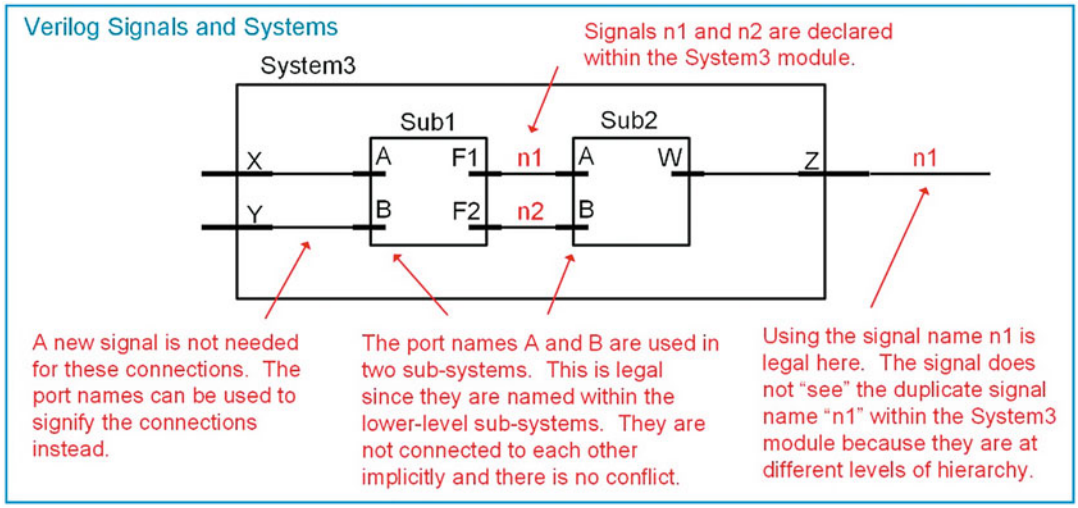


Fig. 2.2
Verilog signals and systems

2.2.4 Parameter Declarations

A parameter, or constant, is useful for representing a quantity that will be used multiple times in the architecture. The syntax for declaring a parameter is as follows:

```
parameter <type> constant_name = <value>;
```

Note that the type is optional and can only be **integer**, **time**, **real**, or **realtime**. If a type is provided, the parameter will have the same properties as a variable of the same time. If the type is excluded, the parameter will take on the type of the value assigned to it.

Example:

```
parameter BUS_WIDTH = 64;
parameter NICKEL    = 8'b0000_0101;
```

Once declared, the constant name can be used throughout the module. The following example illustrates how we can use a constant to define the size of a vector. Notice that since we defined the constant to be the actual width of the vector (i.e., 32-bits), we need to subtract one from its value when defining the indices (i.e., [31:0]).

Example:

```
wire [BUS_WIDTH-1:0] BUS_A;     // It is acceptable to add a "space" for readability
```

2.2.5 Compiler Directives

A compiler directive provides additional information to the simulation tool on how to interpret the Verilog model. A compiler directive is placed before the module definition and is preceded with a backtick (i.e., `). Note that this is not an apostrophe. A few of the most commonly used compiler directives are as follows:

Syntax	Description
`timescale <unit>, <precision>	Defines the timescale of the delay unit and its smallest precision.
`include <filename>	Includes additional files in the compilation.
`define <macroname> <value>	Declares a global constant.

Example:

```
`timescale 1ns/1ps   // Declares the unit of time is 1 ns with a precision of 1ps.
                     // The precision is the smallest amount that the time can
                     // take on. For example, with this directive the number
                     // 0.001 would be interpreted as 0.001 ns, or 1 ps.
                     // However, the number 0.0001 would be interpreted as 0 since
                     // it is smaller than the minimum precision value.
```

Concept Check

CC2.2 If a signal is declared within a module, can the same name be used in other modules within a hierarchical system?

(A) Yes. To support hierarchy, Verilog signals are only seen within their respective module. That allows other modules to use the same names.

(B) No. Once a signal name is defined, it cannot be used again.

Summary

- In a Verilog source file, all functionality is contained within a module. The first portion of the module is the port definition. The second portion contains declarations of internal signals/constants/parameters. The third portion contains the description of the behavior.
- A *port* is an input or output to a system that is defined as part of the initial module statement. A *signal*, or *net*, is an internal connection within the system that is declared inside of the module. A signal is not visible outside of the system.
- Instantiating other modules from within a higher-level module is how Verilog implements hierarchy. A lower-level module can be instantiated as many times as desired. An instance identifier is useful is keeping track of each instantiation. The ports of the component can be connected using either *explicit* or *positional port mapping*.

Exercise Problems

Section 2.1: Data Types

2.1.1 What is the name of the main design unit in Verilog?

2.1.2 What portion of the Verilog module describes the inputs and outputs.

2.1.3 What step is necessary if a system requires internal connections?

2.1.4 What are all the possible values that a Verilog net type can take on?

2.1.5 What is the highest strength that a value can take on in Verilog.

2.1.6 What is the range of decimal numbers that can be represented using the type *integer* in Verilog?

2.1.7 What is the width of the vector defined using the type *[63:0] wire*?

2.1.8 What is the syntax for indexing the most significant bit in the type *[31:0] wire*? Assume the vector is named *example*.

2.1.9 What is the syntax for indexing the least significant bit in the type *[31:0] wire*? Assume the vector is named *example*.

2.1.10 What is the difference between a *wire* and *reg* type?

2.1.11 How many bits is the type *integer* by default?

2.1.12 How many bits is the type *real* by default?

Section 2.2: Verilog Module Construction

2.2.1 What three directions can a *module port* take on?

2.2.2 What data types can a *signal* take on within a module?

2.2.3 What data types can a *parameter* take on within a module?

2.2.4 What is the purpose of a compiler directive?