MODULE-2

BASIC CONCEPTS AND MODULES AND PORTS

2.1: Objectives

- Understand the lexical conventions and define the logic value set and data type.
- Identify useful system tasks and basic compiler directives.
- Identify and understanding of components of a Verilog module definition.
- Understand the port connection rules and connection to external signals by ordered list and by name.

2.2 Lexical conventions

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog contains a stream of tokens. Tokens can be comments, delimiters, numbers, strings, identifiers, and keywords. Verilog HDL is a case-sensitive language. **All keywords are in lowercase.**

2.2.1 Whitespace

Blank spaces (\b), tabs (\t) and newlines (\n) comprise the whitespace. Whitespace is ignored by Verilog except when it separates tokens. Whitespace is not ignored in strings.

2.2.2 Comments

Comments can be inserted in the code for readability and documentation. There are two ways to write comments. A one-line comment starts with "//". Verilog skips from that point to the end of line. A multiple-line comment starts with "/*" and ends with "*/". Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

```
a = b && c; // This is a one-line comment

/* This is a multiple line comment
   */
/* This is /* an illegal */ comment */

/* This is //a legal comment */
```

2.2.3 Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

a = ~ b; // ~ is a unary operator. b is the operand

a = b && c; // && is a binary operator. b and c are operands

a = b ? c : d; // ?: is a ternary operator. b, c and d are operands

2.2.4 Number Specification

There are two types of number specification in Verilog: sized and unsized.

Sized numbers

Sized numbers are represented as <size> '<base format> <number>.

<size> is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O). The number is specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

4'b1111 // This is a 4-bit binary number

12'habc // This is a 12-bit hexadecimal number

16'd255 // This is a 16-bit decimal number

Unsized numbers

Numbers that are specified without a <base format> specification are decimal numbers by default. Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

23456 // This is a 32-bit decimal number by default

'hc3 // This is a 32-bit hexadecimal number

'o21 // This is a 32-bit octal number

X or Z values

Verilog has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by an x. A high impedance value is denoted by z.

12'h13x // This is a 12-bit hex number; 4 least significant bits unknown

6'hx // This is a 6-bit hex number

32'bz // This is a 32-bit high impedance number

An x or z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base and one bit for a number in the binary base. If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z.

This makes it easy to assign x or z to whole vector. If the most significant digit is 1, then it is also zero extended.

Negative numbers

Negative numbers can be specified by putting a minus sign before the size for a constant number. Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>. An optional signed specifier can be added for signed arithmetic.

6'd3 // 8-bit negative number stored as 2's complement of 3

-6'sd3 // Used for performing signed integer math

4'd-2 // Illegal specification

Underscore characters and question marks

An underscore character "_" is allowed anywhere in a number except the first character. Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog. A question mark "?" is the Verilog HDL alternative for z in the context of numbers. The ? is used to enhance readability in the casex and casez statements.

2.2.5 Strings

A string is a sequence of characters that are enclosed by double quotes. The restriction on a string is that it must be contained on a single line, that is, without a carriage return. It cannot be on multiple lines. Strings are treated as a sequence of one-byte ASCII values.

"Hello Verilog World" // is a string

"a / b" // is a string

2.2.6 Identifiers and Keywords

Keywords are special identifiers reserved to define the language constructs. Keywords are in lowercase. Identifiers are names given to objects so that they can be referenced in the design. Identifiers are made up of alphanumeric characters, the underscore (_), or the dollar sign ($). Identifiers are case sensitive. Identifiers start with an alphabetic character or an underscore. They cannot start with a digit or a $ sign (The $ sign as the first character is reserved for system tasks)

reg value; // reg is a keyword; value is an identifier

input clk; // input is a keyword, clk is an identifier

2.2.7 Escaped Identifiers

Escaped identifiers begin with the backslash (\ ) character and end with whitespace (space, tab, or newline). All characters between backslash and whitespace are processed literally. Any printable ASCII character can be included in escaped identifiers.

Neither the backslash nor the terminating whitespace is considered to be a part of the identifier.

\a+b-c

my_name

2.3 Data Types

This section discusses the data types used in Verilog.

2.3.1 Value Set

Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in Table 2-1.

Table 2-1. Value Levels

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits. Value levels 0 and 1 can have the strength levels listed in Table2-2.

Table 2-2. Strength Levels

Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	
pull	riving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	weakest

If two signals of unequal strengths are driven on a wire, the stronger signal prevails. For example, if two signals of strength strong1 and weak0 contend, the result is resolved as a strong1. If two signals of equal strengths are driven on a wire, the result is unknown. If two signals of strength strong1 and strong0 conflict, the result is an x.

2.3.2 Nets

Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to. In Figure 2.1 net a is connected to the output of and gate g1. Net a will continuously assume the value computed at the output of gate g1, which is b & c.

Figure 2.1. Example of Nets

Nets are declared primarily with the keyword wire. Nets are one-bit values by default unless they are declared explicitly as vectors. The terms wire and net are often used interchangeably. The default value of a net is z (except the trireg net, which defaults to x). Nets get the output value of their drivers.

If a net has no driver, it gets the value z.

wire a; // Declare net a for the above circuit

wire b,c; // Declare two wires b,c for the above circuit

wire d = 1'b0; // Net d is fixed to logic value 0 at declaration.

2.3.3 Registers

Registers represent data storage elements. Registers retain value until another value is placed onto them. In Verilog, the term register merely means a variable that can hold a value. Unlike a net, a register does not need a driver. Verilog registers do not need a clock as hardware registers do. Values of registers can be changed anytime in a simulation by assigning a new value to the register.

Register data types are commonly declared by the keyword reg.

Example 3-1 Example of Register

```
reg reset; // declare a variable reset that can hold its value
initial // keyword to specify the initial  value of reg.
reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
#100 reset = 1'b0; // after 100 time units reset is deasserted.
end
```

Example 2-2 Signed Register Declaration

```
reg signed [63:0] m; // 64 bit signed value
integer i; // 32 bit signed value
```

2.3.4 Vectors

Nets or reg data types can be declared as vectors (multiple bit widths). If bit width is not specified, the default is scalar (1-bit).

```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit   bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
reg [0:40] virtual_addr; // Vector register, virtual address 41 bits wide
```

Vectors can be declared at [high# : low#] or [low# : high#], but the left number in the squared brackets is always the most significant bit of the vector. In the example shown above, bit 0 is the most significant bit of vector virtual_addr.

Vector Part Select

For the vector declarations shown above, it is possible to address bits or parts of vectors.

busA[7] // bit # 7 of vector busA

bus[2:0] // Three least significant bits of vector bus,

// using bus[0:2] is illegal because the significant bit shouldalways be on the left of a range specification

virtual_addr[0:1] // Two most significant bits of vector virtual_addr

Variable Vector Part Select

Another ability provided in Verilog HDL is to have variable part selects of a vector. This allows part selects to be put in for loops to select various parts of the vector. There are two special part-select operators:

[<*starting_bit*>*+:width*] - part-select increments from starting bit.

[<*starting_bit*>*-:width*] - part-select decrements from starting bit.

The starting bit of the part select can be varied, but the width has to be constant. The following example shows the use of variable vector part select:

```
reg [255:0] data1; //Little endian notation
reg [0:255] data2; //Big endian notation
reg [7:0] byte;
//Using a variable part select, one can choose parts
byte = data1[31-:8]; //starting bit = 31, width =8 => data[31:24]
byte = data1[24+:8]; //starting bit = 24, width =8 => data[31:24]
byte = data2[31-:8]; //starting bit = 31, width =8 => data[24:31]
byte = data2[24+:8]; //starting bit = 24, width =8 => data[24:31]

//The starting bit can also be a variable. The width has to be constant.
//Therefore, one can use the variable part select
//in a loop to select all bytes of the vector.
 for (j=0; j<=31; j=j+1)
byte = data1[(j*8)+:8]; //Sequence is [7:0], [15:8]... [255:248]
//Can initialize a part of the vector
data1[(byteNum*8)+:8] = 8'b0; //If byteNum = 1, clear 8 bits [15:8]
```

2.3.5 Integer , Real, and Time Register Data Types

Integer, real, and time register data types are supported in Verilog.

Integer

An integer is a general purpose register data type used for manipulating quantities. Integers are declared by the keyword integer. Although it is possible to use reg as a general-purpose variable, it is more convenient to declare an integer variable for purposes such as counting. The default width for an integer is the host-machine word size, which is implementation-specific but is at least 32 bits. Registers declared as data type reg store values as unsigned quantities, whereas integers store values as signed quantities.

```
integer counter; // general purpose variable used as a counter.
initial
counter = -1; // A negative one is stored in the counter
```

Real

Real number constants and real register data types are declared with the keyword real. They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3×10^6). Real numbers cannot have a range declaration, and their default value is 0. When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

```
real delta; // Define a real variable called delta initial
begin
delta = 4e10; // delta is assigned in scientific notation

delta = 2.13; // delta is assigned a value 2.13 end
integer i; // Define an integer i
initial
i = delta; // i gets the value 2 (rounded value of 2.13)
```

Time

Verilog simulation is done with respect to simulation time. A special time register data type is used in Verilog to store simulation time. A time variable is declared with the keyword time. The width for time register data types is implementation-specific but is at least 64 bits.The system function $time is invoked to get the current simulation time.

```
time save_sim_time; // Define a time variable save_sim_time

initial
```

save_sim_time = $time; // Save the current simulation time

Arrays

Arrays are allowed in Verilog for reg, integer, time, real, realtime and vector register data types. Multi-dimensional arrays can also be declared with any number of dimensions. Arrays of nets can also be used to connect ports of generated instances. Each element of the array can be used in the same fashion as a scalar or vector net. Arrays are accessed by <array_name>[<subscript>]. For multi- dimensional arrays, indexes need to be provided for each dimension.

integer count[0:7]; // An array of 8 count variables

reg bool[31:0]; // Array of 32 one-bit boolean register variables time

chk_point[1:100]; // Array of 100 time checkpoint variables reg [4:0]

port_id[0:7]; // Array of 8 port_ids; each port_id is 5 bits wide

integer matrix[4:0][0:255]; // Two dimensional array of integers

reg [63:0] array_4d [15:0][7:0][7:0][255:0]; //Four dimensional array

wire [7:0] w_array2 [5:0]; // Declare an array of 8 bit vector wire

wire w_array1[7:0][5:0]; // Declare an array of single bit wires.

It is important not to confuse arrays with net or register vectors. A vector is a single element that is n-bits wide. On the other hand, arrays are multiple elements that are 1-bit or n-bits wide.

Examples of assignments to elements of arrays discussed above are shown below:

count[5] = 0; // Reset 5th element of array of count variables

chk_point[100] = 0; // Reset 100th time check point value

port_id[3] = 0; // Reset 3rd element (a 5-bit value) of port_id array.

matrix[1][0] = 33559; // Set value of element indexed by [1][0] to 33559

port_id = 0; // Illegal syntax - Attempt to write the entire array

matrix [1] = 0; // Illegal syntax - Attempt to write [1][0]..[1][255]

2.3.6 Memories

In digital simulation, one often needs to model register files, RAMs, and ROMs. Memories are modeled in Verilog simply as a one-dimensional array of registers. Each element of the array is known as an element or word and is addressed by a single array index. Each word can be one or more bits. It is important to differentiate between n 1-bit registers and one n-bit register. A particular word in memory is obtained by using the address as a memory array subscript.

reg mem1bit[0:1023]; // Memory mem1bit with 1K 1-bit words

reg [7:0] membyte[0:1023]; // Memory membyte with 1K 8-bit words(bytes)

membyte[511] // Fetches 1 byte word whose address is 511.

2.3.7 Parameters

Verilog allows constants to be defined in a module by the keyword parameter. Parameters cannot be used as variables. Parameter values for each module instance can be overridden individually at compile time. This allows the module instances to be customized. This aspect is discussed later. Parameter types and sizes can also be defined.

parameter port_id = 5; // Defines a constant port_id

parameter cache_line_width = 256; // Constant defines width of cache line

parameter signed [15:0] WIDTH; // Fixed sign and range for parameter WIDTH

2.3.8 Strings

Strings can be stored in reg. The width of the register variables must be large enough to hold the string. Each character in the string takes up 8 bits (1 byte). If the width of the register is greater than the size of the string, Verilog fills bits to the left of the string with zeros. If the register width is smaller than the string width, Verilog truncates the leftmost bits of the string. It is always safe to declare a string that is slightly wider than necessary.

reg [8*18:1] string_value; // Declare a variable that is 18 bytes wide initial

string_value = "Hello Verilog World"; // String can be stored in variable

Special characters serve a special purpose in displaying strings, such as newline, tabs, and displaying argument values. Special characters can be displayed in strings only when they are preceded by escape characters, as shown in Table 2-3

Table 2-3. Special Characters

Escaped Characters	Character Displayed
\n	newline
\t	tab
%%	%
\\	\
\"	"
\ooo	Character written in 1?3 octal digits

2.4 System Tasks and Compiler Directives

In this section, we introduce two special concepts used in Verilog: system tasks and compiler directives.

2.4.1 System Tasks

Verilog provides standard system tasks for certain routine operations. All system tasks appear in the form $<keyword>. Operations such as displaying on the screen, monitoring values of nets, stopping, and finishing are done by system tasks.

Displaying information

$display is the main system task for displaying values of variables or strings or expressions. This is one of the most useful tasks in Verilog.

Usage: $display(p1, p2, p3,....., pn);

p1, p2, p3,..., pn can be quoted strings or variables or expressions. The format of $display is very similar to printf in C. A $display inserts a newline at the end of the string by default. A $display without any arguments produces a newline.

Monitoring information

Verilog provides a mechanism to monitor a signal when its value changes. This facility is provided by the $monitor task.

Usage: $monitor(p1,p2,p3,....,pn);

The parameters p1, p2, ... , pn can be variables, signal names, or quoted strings. A format similar to the $display task is used in the $monitor task. $monitor continuously monitors the values of the variables or signals specified in the parameter list and displays all parameters in the list whenever the value of any one variable or signal changes. Unlike $display, $monitor needs to be invoked only once. Only one monitoring list can be active at a time.

If there is more than one $monitor statement in your simulation, the last $monitor statement will be the active statement. The earlier $monitor statements will be overridden.

Two tasks are used to switch monitoring on and off.

Usage:

$monitoron;

$monitoroff;

The $monitoron tasks enables monitoring, and the $monitoroff task disables monitoring during a simulation.

Example of Monitor Statement

```
//Monitor time and value of the signals clock and reset
//Clock toggles every 5 time units and reset goes down at 10 time units
initial
begin
$monitor ($time," Value of signals clock = %b reset = %b", clock,reset);
end
```

Partial output of the monitor statement:

```
-- 0 Value of signals clock = 0 reset = 1
-- 5 Value of signals clock = 1 reset = 1
-- 10 Value of signals clock = 0 reset = 0
```

Stopping and finishing in a simulation

The task $stop is provided to stop during a simulation.

Usage: $stop;

The $stop task puts the simulation in an interactive mode. The designer can then debug the design from the interactive mode. The $stop task is used whenever the designer wants to suspend the simulation and examine the values of signals in the design.

The $finish task terminates the simulation.

Usage: $finish;

Examples of $stop and $finish are given below

Example of Stop and Finish Tasks

```
// Stop at time 100 in the simulation and examine the results
// Finish the simulation at time 1000.
initial
begin
clock = 0;
reset = 1;
#100 $stop; // This will suspend the simulation at time = 100
#900 $finish; // This will terminate the simulation at time = 1000
end
```

2.4.2 Compiler Directives

Compiler directives are provided in Verilog. All compiler directives are defined by using the `<keyword> construct. The two most useful compiler directives are

`define

The `define directive is used to define text macros in Verilog .The Verilog compiler substitutes the text of the macro wherever it encounters a `<macro_name>. This is similar to the #define construct in C. The defined constants or text macros are used in the Verilog code by preceding them with a ` (back tick).

Example for `define Directive

```
//define a text macro that defines default word size
//Used as 'WORD_SIZE in the code
'define WORD_SIZE 32
//define an alias. A $stop will be substituted wherever 'S appears
'define S $stop;
//define a frequently used text string
'define WORD_REG reg [31:0]
```

`include

The `include directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation. This works similarly to the #include in the C programming language.

Example for `include Directive

```
// Include the file header.v, which contains declarations in themain verilog file design.v.
'include header.v
...
...
<Verilog code in file design.v>
...
...
```

2.5 Modules

Module is a basic building block in Verilog. A module definition always begins with the keyword module. The module name, port list, port declarations, and optional parameters must come first in a module definition. Port list and port declarations are present only if the module has any ports to interact with the external environment.

The five components within a module are: variable declarations, dataflow statements, instantiation of lower modules, behavioral blocks, and tasks or functions. These components can be in any order and at any place in the module definition.

The endmodule statement must always come last in a module definition. All components except module, module name, and endmodule are optional and can be mixed and matched as per design needs. Verilog allows multiple modules to be defined in a single file. The modules can be defined in any order in the file.

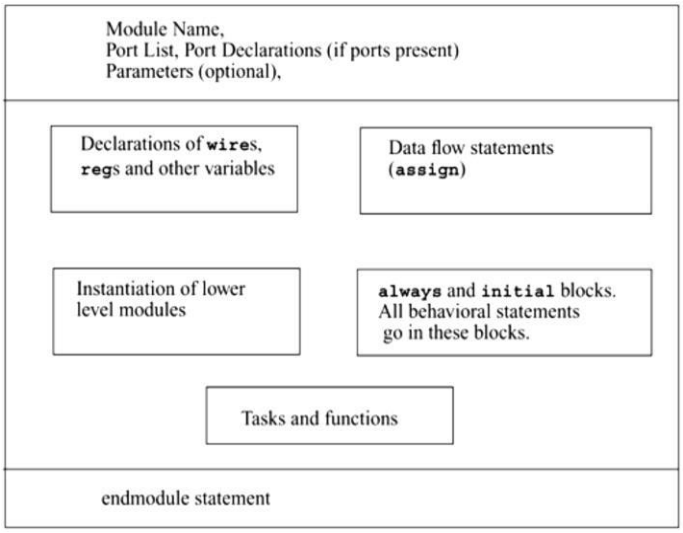

Figure 2.2.:Components of a Verilog Module

Consider a simple example of an SR latch, as shown in Figure 2.3

Figure 2-3. SR Latch

The SR latch has S and R as the input ports and Q and Qbar as the output ports. The SR latch and its stimulus can be modeled as shown in Example.

Example of Components of SR Latch

```
// This example illustrates the different components of a module
// Module name and port list
// SR_latch module
module SR_latch(Q, Qbar, Sbar, Rbar);
//Port declarations
output Q, Qbar;
input Sbar, Rbar;
// Instantiate lower-level modules
// In this case, instantiate Verilog primitive nand gates
// Note how the wires are connected in a cross-coupled fashion. nand n1(Q, Sbar, Qbar);
nand n2(Qbar, Rbar, Q);
// endmodule statement
endmodule

// Module name and port list
// Stimulus module
module Top;
// Declarations of wire, reg, and other variables
reg set, reset;
// Instantiate lower-level modules
// In this case, instantiate SR_latch Feed inverted set and reset signals to the SR latch
SR_latch m1(q, qbar, ~set, ~reset);
// Behavioral block, initial
initial
begin
$monitor($time, " set = %b, reset= %b, q= %b\n",set,reset,q);
set = 0; reset = 0;
#5 reset = 1;
```

```
#5 reset = 0;
#5 set = 1;
end
// endmodule statement
endmodule
```

From the above example following characteristics are noticed:

• In the SR latch definition above ,all components described in Figure 2-2 need not be present in a module. We do not find variable declarations, dataflow (assign) statements, or behavioral blocks (always or initial).

• However, the stimulus block for the SR latch contains module name, wire, reg, and variable declarations, instantiation of lower level modules, behavioral block (initial), and endmodule statement but does not contain port list, port declarations, and data flow (assign) statements.

• Thus, all parts except module, module name, and endmodule are optional and can be mixed and matched as per design needs.

2.6 Ports

Ports provide the interface by which a module can communicate with its environment. For example, the input/output pins of an IC chip are its ports. The environment can interact with the module only through its ports. The internals of the module are not visible to the environment. This provides a very powerful flexibility to the designer. The internals of the module can be changed without affecting the environment as long as the interface is not modified. Ports are also referred to as terminals.

2.6.1 List of Ports

A module definition contains an optional list of ports. If the module does not exchange any signals with the environment, there are no ports in the list. Consider a 4-bit full adder that is instantiated inside a top-level module Top. The diagram for the input/output ports is shown in Figure 2-4.

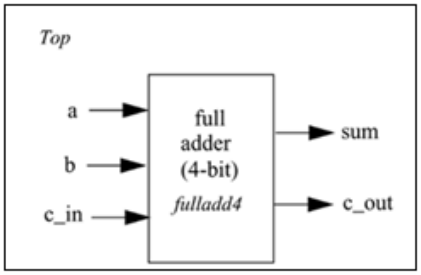

Figure 2-4. I/O Ports for Top and Full Adder

From the above figure, the module Top is a top-level module. The module fulladd4 is instantiated below Top. The module fulladd4 takes input on ports a, b, and c_in and produces an output on ports sum and c_out. Thus, module fulladd4 performs an addition for its environment. The module Top is a top-level module in the simulation and does not need to pass signals to or receive signals from the environment. Thus, it does not have a list of ports. The module names and port lists for both module declarations in Verilog are as shown in below example.

Example of List of Ports

```
module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports
module Top; // No list of ports, top-level module in simulation
```

2.6.2 Port Declaration

All ports in the list of ports must be declared in the module. Ports can be declared as follows:

input -Input port

output- Output port

inout- Bidirectional port

Each port in the port list is defined as input, output, or inout, based on the direction of the port signal. Thus, for the example of the the port declarations will be as shown in example below.

Example for Port Declarations

```
module fulladd4(sum, c_out, a, b, c_in);
//Begin port declarations section
output[3:0] sum;
output c_cout;
 input [3:0] a, b;
input c_in;
//End port declarations section
...
<module internals>
... endmodule
```

All port declarations are implicitly declared as wire in Verilog. Thus, if a port is intended to be a wire, it is sufficient to declare it as output, input, or inout. Input or inout ports are normally declared as wires.

However, if output ports hold their value, they must be declared as reg. Ports of the type input and inout cannot be declared as reg because reg variables store values and input ports should not store values but simply reflect the changes in the external signals they are connected to.

Alternate syntax for port declaration is shown in below example. This syntax avoids the duplication of naming the ports in both the module definition statement and the module port list definitions. If a port is declared but no data type is specified, then, under specific circumstances, the signal will default to a wire data type.

Example for ANSI C Style Port Declaration Syntax

```
module fulladd4(output reg [3:0] sum,
output reg c_out,
input [3:0] a, b, //wire by default
input c_in); //wire by default
...
<module internals>
...
endmodule
```

2.6.3 Port Connection Rules

A port as consisting of two units, one unit that is internal to the module and another that is external to the module. The internal and external units are connected. There are rules governing port connections when modules are instantiated within other modules. The Verilog simulator complains if any port connection rules are violated. These rules are summarized in Figure2.5

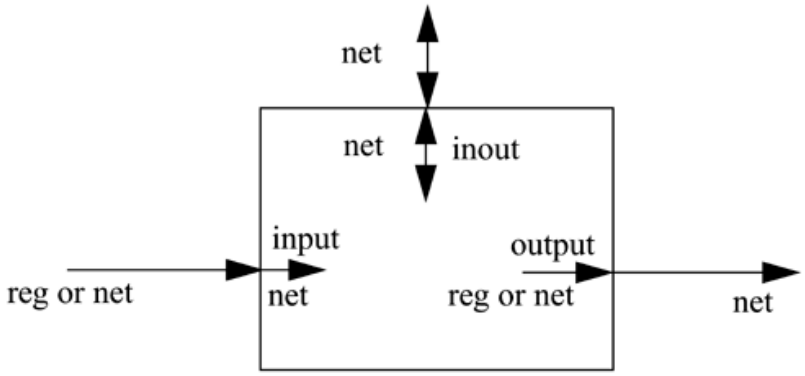

Figure 2-5. Port Connection Rules

Inputs

Internally, input ports must always be of the type net. Externally, the inputs can be connected to a variable which is a reg or a net.

Outputs

Internally, outputs ports can be of the type reg or net. Externally, outputs must always be connected to a net. They cannot be connected to a reg.

Inouts

Internally, inout ports must always be of the type net. Externally, inout ports must always be connected to a net.

Width matching

It is legal to connect internal and external items of different sizes when making intermodule port connections. However, a warning is typically issued that the widths do not match.

Unconnected ports

Verilog allows ports to remain unconnected. For example, certain output ports might be simply for debugging, and you might not be interested in connecting them to the external signals. You can let a port remain unconnected by instantiating a module as shown below

```
fulladd4  fa0 (SUM,    , A, B, C_IN); // Output port c_out is unconnected
```

Example of illegal port connection

To illustrate port connection rules, assume that the module fulladd4 Example is instantiated in the stimulus block Top. Below example shows an illegal port connection

Example 2-14 Illegal Port Connection

```
module Top;
//Declare connection variables reg
[3:0]A,B;
reg C_IN;
reg [3:0] SUM;
wire C_OUT;
//Instantiate fulladd4, call it fa0
fulladd4 fa0(SUM, C_OUT, A, B, C_IN);
//Illegal connection because output port sum in module fulladd4
//is connected to a register variable SUM in module Top.
```

.

.

<stimulus>

.

. endmodule

This problem is rectified if the variable SUM is declared as a net (wire).

2.7 Connecting Ports to External Signals

There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition. These two methods cannot be mixed. These methods are

Connecting by ordered list

The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition. Consider the module fulladd4.To connect signals in module Top by ordered list, the Verilog code is shown in below example. Notice that the external signals SUM, C_OUT, A, B, and C_IN appear in exactly the same order as the ports sum, c_out, a, b, and c_in in module definition of fulladd4.

Example 2-15 Connection by Ordered List

```
module Top;
//Declare connection variables
reg [3:0]A,B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
//Instantiate fulladd4, call it fa_ordered.
//Signals are connected to ports in order (by position)
fulladd4 fa_ordered (SUM, C_OUT, A, B, C_IN);
...
<stimulus>
... endmodule
```

```
module fulladd4(sum, c_out, a, b, c_in);
output[3:0] sum; output c_cout; input [3:0] a, b; input c_in;
...
<module internals>
... endmodule
```

Connecting ports by name

For large designs where modules have, say, 50 ports, remembering the order of the ports in the module definition is impractical and error-prone. Verilog provides the capability to connect external signals to ports by the port names, rather than by position. We could connect the ports by name in above example by instantiating the module fulladd4, as follows. Note that you can specify the port connections in any order as long as the port name in the module definition correctly matches the external signal.

```
// Instantiate module fa_byname and connect signals to ports by name
fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN), .a(A),);
```

Note that only those ports that are to be connected to external signals must be specified in port connection by name. Unconnected ports can be dropped. For example, if the port c_out were to be kept unconnected, the instantiation of fulladd4 would look as follows. The port c_out is simply dropped from the port list.

```
// Instantiate module fa_byname and connect signals to ports by
name fulladd4 fa_byname(.sum(SUM), .b(B), .c_in(C_IN), .a(A),);
```

Another advantage of connecting ports by name is that as long as the port name is not changed, the order of ports in the port list of a module can be rearranged without changing the port connections in module instantiations.

2.8 Hierarchical Names

Every module instance, signal, or variable is defined with an identifier. A particular identifier has a unique place in the design hierarchy. Hierarchical name referencing allows us to denote every identifier in the design hierarchy with a unique name. A hierarchical name is a list of identifiers separated by dots (".") for each level of hierarchy. Thus, any identifier can be addressed from any place in the design by simply specifying the complete hierarchical name of that identifier. The top-level module is called the root module because it is not instantiated anywhere. It is the starting point.

To assign a unique name to an identifier, start from the top-level module and trace the path along the design hierarchy to the desired identifier.

Consider the simulation of SR latch Example. The design hierarchy is shown in Figure 2.6.

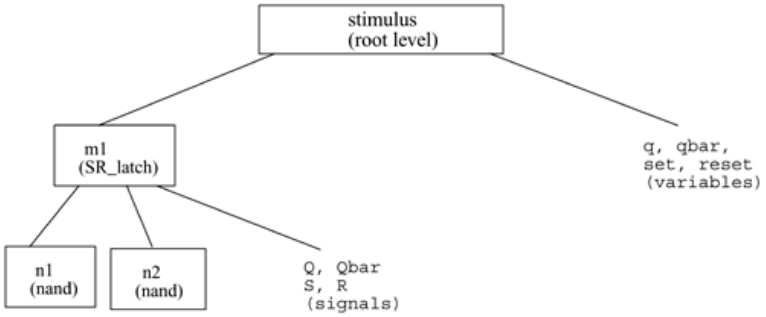

Figure 2-6. Design Hierarchy for SR Latch Simulation

For this simulation, stimulus is the top-level module. Since the top-level module is not instantiated anywhere, it is called the root module. The identifiers defined in this module are q, qbar, set, and reset. The root module instantiates m1, which is a module of type SR_latch. The module m1 instantiates nand gates n1 and n2. Q, Qbar, S, and R are port signals in instance m1. Hierarchical name referencing assigns a unique name to each identifier. To assign hierarchical names, use the module name for root module and instance names for all module instances below the root module.

Example

stimulus
stimulus.q
stimulus.qbar
timulus.set
stimulus.reset
stimulus.m1
stimulus.m1.Q
stimulus.m1.Qbar
stimulus m1.S
stimulus.m1.R
stimulus.n1
stimulus.n2

Each identifier in the design is uniquely specified by its hierarchical path name. To display the level of hierarchy, use the special character %m in the $display task.

2.9: Outcomes

After completion of the module the students are able to:

- Understand the lexical conventions and different data types of verilog.
- Identify useful system tasks such as $display and $monitor and basic compiler directives.
- Understand different components of a Verilog module definition
- Understand the port connection rules and connection to external signals by ordered list and by name

2.10: Recommended questions

1. Describe the lexical conventions used in Verilog HDL with examples.
2. Explain different data types of Verilog HDL with examples
3. What are system tasks and compiler directives?
4. What are the uses of $monitor, $display and $finish system tasks? Explain with examples.
5. Explain `define and `include compiler directives.
6. Explain the components of Verilog HDL module.
7. What are the components of SR latch? Write Verilog HDL module of SR latch.
8. Explain the different types of ports supported by Verilog HDL with examples.
9. Explain the port connection rules of Verilog HDL with examples.
10. How hierarchical names helps in addressing any identifier used in the design from any other level of hierarchy? Explain with examples.
11. What are the basic components of a module? Which components are mandatory?