

Module – 2

Basic Concepts

Lexical Conventions

- Verilog language source text files are a stream of lexical tokens.
- A token consists of one or more characters, and each single character is in exactly one token.
- The layout of tokens in a source file is free format - that is, spaces and newlines are not syntactically significant. However, spaces and newlines are very important for giving a visible structure and format to source descriptions.
- A good style of format, and consistency in that style, are an essential part of program readability.
- The types of lexical tokens in the language are:
 - i. Operator
 - ii. White Space
 - iii. Comment
 - iv. Number Specification
 - v. String
 - vi. Identifier
 - vii. Keyword

Whitespace

- White space can be
 - i. Blank spaces (\b)
 - ii. Tabs (\t)
 - iii. Newlines (\n).
- Whitespace is ignored by Verilog except when it separates tokens
- Whitespace is not ignored in strings

Comments

- Comments can be inserted in the code for readability and documentation.
- There are two ways to write comments.
 - i. A one line comment starts with “//”. Verilog skips from that point to the end of line.
 - ii. A multiple line comment starts with “/*” and ends with “*/”.

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

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
```

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> element contains digits that are legal for the specified <base_format>. The <number> element must physically follow the <base_format>, but can be separated from it by spaces. No spaces can separate the single quote and the base specifier character.

```
4'b1111          // 4-bit binary number
```

```
12'habc      // 12-bit hexadecimal number
16'd255      // 16-bit decimal number.
```

Unsize 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 i.e. simulator and machine specific (must be at least 32 bits).

```
23456        // 32-bit decimal number by default
'hc3         // 32-bit hexadecimal number
'o21         // 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      // 12-bit hex number; 4 least significant bits unknown
6'hx         // 6-bit hex number
32'bz        // 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>.

```
-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.

```
12'b1111_0000_1010    // Use of underline characters to improve readability
4'b10??                // Equivalent of a 4'b10zz
```

Strings

- A string is a sequence of characters that are enclosed by double quotes.
- 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
```

Identifiers and Keywords

- Keywords are predefined non-escaped identifiers that are used to define the language constructs written 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.

```
reg value;              // reg is a keyword; value is an identifier
input clk;              // input is a keyword, clk is an identifier
```

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**

Data Types

Value Set

- Verilog supports four values and eight strengths to model the functionality of real hardware
- The four value levels are: 0, 1, x, z
 - 0 - represents logic zero, or false condition
 - 1 - represents logic one, or true condition
 - x - represents an unknown logic value
 - z - represents high-impedance state
- Strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits.

Table 2.1: Strength Levels

Strength Level	Type	Degree
supply	Driving	
strong	Driving	
pull	Driving	
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.

Value level	HW Condition
-------------	--------------

0	Logic zero, false
1	Logic one, true
x	Unknown
z	High imp., floating

- 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.

Nets

- Nets represent physical connections between hardware elements
- Nets are declared primarily with the keyword wire
- The terms wire and net are often used interchangeably
- Nets get the output value of their drivers. The default value of a net is z
- A net does not store a value. Instead, it must be driven by a driver, such as a gate or a continuous assignment.



```

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.
  
```

Registers

- Registers represent data storage elements.
- Registers retain value until another value is placed onto them.
- Register data types are commonly declared by the keyword reg. The default value for a reg data type is x.
- In Verilog, the term register merely means a variable that can hold a value. Unlike a net, a register does not need a driver.

- Values of registers can be changed anytime in a simulation by assigning a new value to the register.

```

reg reset;           // declare a variable reset that can hold its value
initial
begin
    reset = 1'b1;      // initialize reset to 1 to reset the digital circuit.
    #10 reset = 1'b0;  // after 10 time units reset is deasserted.
end

```

- Registers can also be declared as signed variables. Such registers can be used for signed arithmetic.

```

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

```

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.

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

```

Variable Vector Part Select

- Variable part selects allows selecting 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
reg [255:0] data1;          // Little endian notation
reg [0:255] data2;          // Big endian notation
reg [7:0] byte;
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]

```

Integer

- An integer is a general purpose register data type used for manipulating quantities.
- Integers are declared by the keyword integer.
- Integers store values as signed quantities.
- The default width for an integer is the host-machine word size, which is implementation-specific but is at least 32 bits.

```

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.
- Real numbers cannot have a range declaration, and their default value is 0.
- They can be specified in decimal notation or in scientific notation.
- 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 in decimal notation

```



```

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

```

- Simulation time is measured in terms of simulation seconds. The unit is denoted by s, the same as real 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
```

- 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.

Memories

- 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.
```

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.

```
parameter msb = 7; // defines msb as a constant value 7
parameter e = 25, f = 9; // defines two constant numbers
parameter average_delay = (r + f) / 2;
parameter byte_size = 8, byte_mask = byte_size - 1;
parameter r = 5.7; //declares r as a 'real' parameter
```

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
```

System Tasks and Compiler Directives

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. The syntax of \$display is

```
$display (p1, p2, p3, ....., pn);
```

p1, p2, p3,..., pn can be quoted strings or variables or expressions.

For example:

```
// Display the string in quotes
    $display ("Hello Verilog World");
    -- Hello Verilog World

// Display value of current simulation time 230
    $display ($time);
    -- 230

// Display value of 41-bit virtual address 1fe0000001c at time 200
    reg [0:40] virtual_addr;
    $display ("At time %d virtual address is %h", $time, virtual_addr);
    -- At time 200 virtual address is 1fe0000001c
```

- A \$display inserts a newline at the end of the string by default. A \$display without any arguments produces a newline.
- Strings can be formatted using the specifications as listed below

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format (e.g., 3e10)
%f or %F	Display real number in decimal format (e.g., 2.13)
%g or %G	Display real number in scientific or decimal, whichever is shorter

Monitoring Information

- Verilog provides a mechanism to monitor a signal when its value changes. This facility is provided by the \$monitor.
- \$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 variable or signal changes.
- The syntax of \$ monitor is

\$monitor (p1, p2, p3, ..., pn);

p1, p2, ..., pn can be variables, signal names, or quoted strings.

For example:

// 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
```

- 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.


```
$monitoron;
$monitoroff;
```
- The \$monitoron task enables monitoring, and the \$monitoroff task disables monitoring during a simulation.
- Monitoring is turned on by default at the beginning of the simulation and can be controlled during the simulation with the \$monitoron and \$monitoroff tasks.

Stopping and Finishing in a Simulation

\$stop

- It is used to stop during a simulation.
- The \$stop task puts the simulation in an interactive mode to enable debugging.
- The \$stop task is used whenever the designer wants to suspend the simulation and examine the values of signals in the design.

\$finish

- It is used to terminate the simulation.


```
// 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
```

Compiler Directives

- Compiler directives are defined by using the ``<keyword>` construct.

``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).

```
// 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 32-bit register as 'WORD_REG
`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. This directive is typically used to include header files, which typically contain global or commonly used definitions

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

...

...

Modules and Ports

Modules

- A module in Verilog consists of distinct parts, as shown in fig.2.1.
- 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.

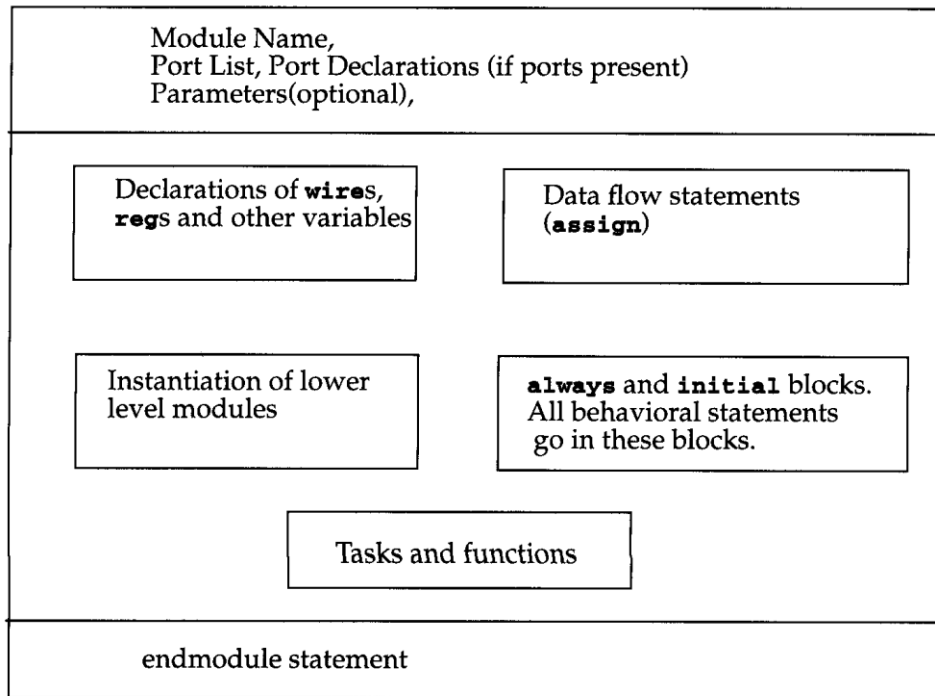


Fig.2.1: Components of a Verilog Module

- All components except module, module name, and endmodule are optional.
- Verilog allows multiple modules to be defined in a single file. The modules can be defined in any order in the file.

Verilog HDL Code to design SR latch using nand gates

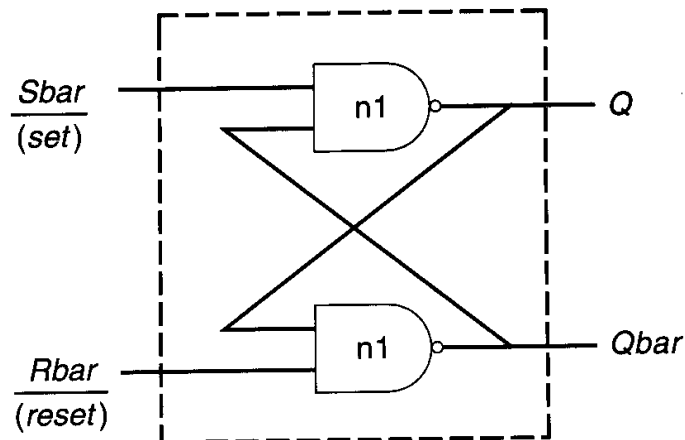


Fig.2.2: SR Latch

- The SR latch has Sbar and Rbar as the input ports and Q and Qbar as the output ports. The SR latch and its stimulus can be modeled as shown below

```
// Module name and port list
```



```
module SR_latch (Sbar, Rbar, Q, Qbar);
    // Port declarations
    input Sbar, Rbar;
    output Q, Qbar;

    // Instantiate lower-level modules (Verilog primitive nand gates)
    nand n1(Q, Sbar, Qbar);
    nand n2(Qbar, Rbar, Q);

    // endmodule statement
endmodule

// Stimulus module / Testbench
// Module name and port list
module SR_latch_tb();
    // Declarations of wire, reg, and other variables
    reg set, reset;
    wire q, qbar;

    // Instantiate lower-level modules i.e. instantiate SR_latch
    SR_latch m1(~set, ~reset, q, qbar);

    // Behavioral block, initial
    initial
    begin
        set = 0; reset = 0;
        #5 set = 0; reset = 1;
        #5 set = 1; reset = 0;
    end

    // endmodule statement
endmodule
```

- In the SR latch definition above, notice that all components described in fig.2.1 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.

Ports

- Ports also referred to as terminals, 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.
- The internals of the module can be changed without affecting the environment as long as the interface is not modified.

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 fig.2.3.

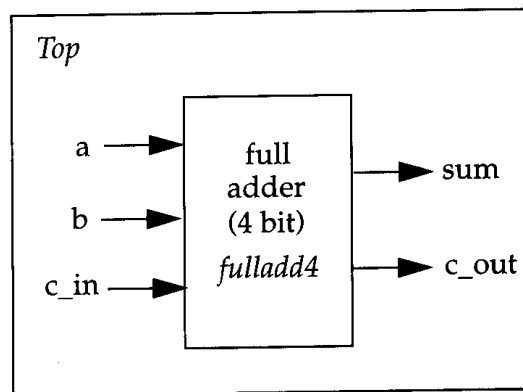


Fig.2.3: I/O Ports for Top and Full Adder

- 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.

```

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

```

Port Declaration

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

Verilog Keyword	Type of Port
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.
- 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.
- Thus, for the example of the fulladd4 in fig.2.3, the port declarations will be

```

module fulladd4(a, b, c_in, sum, c_out);
    // Begin port declarations
    input [3:0] a, b;
    input c_in;
    output[3:0] sum;

```

```
output c_cout;
    // End port declarations
...
< module internals >
...
endmodule
```

Alternative Declaration

- Alternately ports can be declared using an ANSI C style
- This syntax avoids the duplication of naming the ports in both the module definition statement and the module port list definitions.

```
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
```

Port Connection Rules

- When modules are instantiated within other modules the port connections are governed by set of rules.
- Port can be visualized as two units, one unit that is internal to the module and another that is external to the module.
- When the internal and external units are connected it is governed by set of rules.

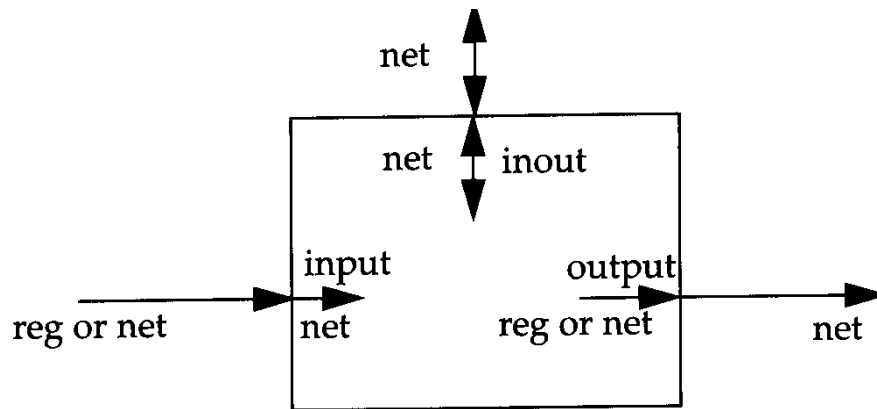


Fig.2.4: Port Connection Rules

inputs

- Internally, input ports must 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 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 inter-module port connections. However, a warning is typically issued that the widths do not match.

unconnected ports

- Verilog allows ports to remain unconnected.

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.

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 defined in fig.2.3. To connect signals in module Top by ordered list, the external signals A, B, C_IN, SUM and C_OUT appear exactly in the same order as the ports a, b, c_in, sum and c_out in module definition of fulladd4.

```
// 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, Signals are connected to ports in same order
    fulladd4 fa_ordered (A, B, C_IN, SUM, C_OUT);

    ...
    < stimulus >
    ...
endmodule


module fulladd4 (a, b, c_in, sum, c_out);
    input [3:0] a, b;
    input c_in;
    output[3:0] sum;
    output c_cout;

    ...
    < 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 for the design defined in fig.2.5 by instantiating the module fulladd4, as follows.

```
// 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 we can specify the port connections in any order as long as the port name in the module definition correctly matches the external signal.
- 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.

Verilog HDL Code to design 4-bit Ripple Carry Full Adder

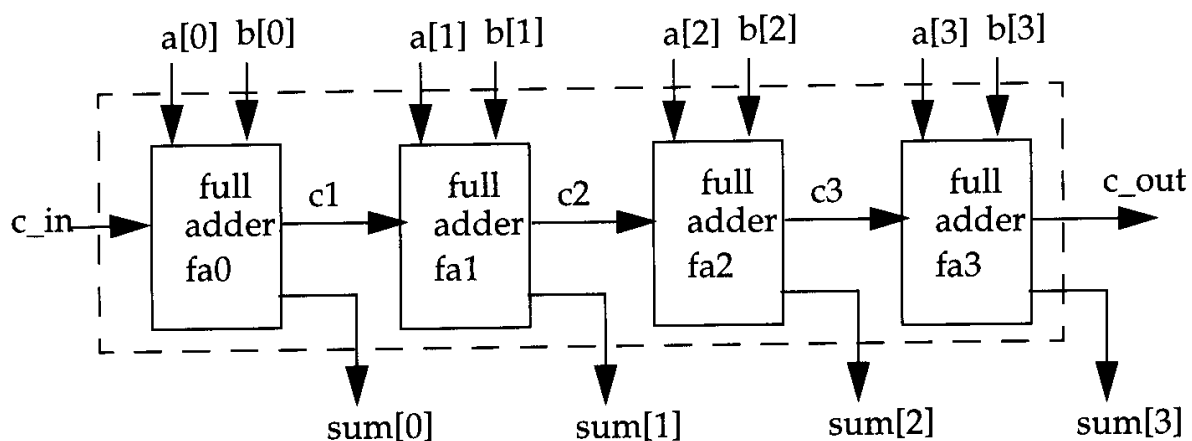


Fig.2.5: 4-bit Ripple Carry Full Adder

```
// Define a 4-bit full adder
```

```
module fulladd4 (a, b, c_in, sum, c_out);
```

```
// I/O port declarations
```

```
input [3:0] a, b;
```

```
input c_in;
```

```
output [3:0] sum;
```

```
output c_out;
```

```
// Internal nets
```

```

    wire c1, c2, c3;

    // Instantiate four 1-bit full adders.
    fulladd fa0 (a[0], b[0], c_in, sum[0], c1);
    fulladd fa1 (a[1], b[1], c1, sum[1], c2);
    fulladd fa2 (a[2], b[2], c2, sum[2], c3);
    fulladd fa3 (a[3], b[3], c3, sum[3], c_out);
endmodule

```

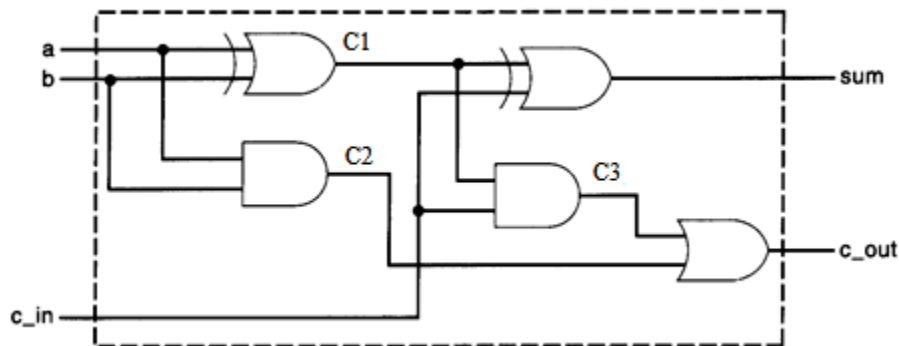


Fig.2.6: 1-bit Full Adder

```

// Define a 1-bit full adder
module fulladd (a, b, c_in, sum, c_out);

// I/O port declarations
input a, b, c_in;
output sum, c_out;

// Internal nets
wire c1, c2, c3;

// Instantiate logic gate primitives
xor (c1, a, b);
and (c2, a, b);
xor (sum, c1, c_in);
and (c3, c1, c_in);
or (c_out, c3, c2);
endmodule

```



```
// Define the stimulus / Testbench (top level module)

    module stimulus;

// Set up variables
        reg [3:0] A, B;
        reg C_IN;
        wire [3:0] SUM;
        wire C_OUT;

// Instantiate the 4-bit full adder.
        fulladd4 FA1_4 (A, B, C_IN, SUM, C_OUT);

// Stimulate inputs
        initial
        begin
            A = 4'd0; B = 4'd0; C_IN = 1'b0;
            #5 A = 4'd3; B = 4'd4;
            #5 A = 4'd2; B = 4'd5;
            #5 A = 4'd9; B = 4'd9;
            #5 A = 4'd10; B = 4'd15;
            #5 A = 4'd10; B = 4'd5; C_IN = 1'b1;
        end
    endmodule
```