

UNIT IV

Verilog as HDL – Levels for design description – Language elements – Data Types – Operators – Module structure – Gate primitives – Timing controls – Procedural and Conditional assignments – Data flow Modeling –Structural Modeling – examples.

VERILOG AS HDL

Verilog HDL is a hardware description language that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level to the switch level. The complexity of the digital system being modeled could vary from that of a simple gate to a complete electronic digital system, or anything in between. The digital system can be described hierarchically and timing can be explicitly modelled within the same description.

Typical Design Flow

A typical design flow for designing VLSI IC circuits is shown in Figure 1.1. Unshaded blocks show the level of design representation; shaded blocks show processes in the design flow.

The design flow shown in Figure 1-1 is typically used by designers who use HDLs. In any design, specifications are written first. Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. At this point, the architects do not need to think about how they will implement this circuit. A behavioral description is then created to analyse the design in terms of functionality, performance, compliance to standards, and other high-level issues. Behavioral descriptions are often written with HDLs

The behavioral description is manually converted to an RTL description in an HDL. The designer has to describe the data flow that will implement the desired digital circuit. From this point onward, the design process is done with the assistance of EDA tools.

Logic synthesis tools convert the RTL description to a gate-level netlist. A gate-level netlist is a description of the circuit in terms of gates and connections between them. Logic synthesis tools ensure that the gate-level netlist meets timing, area, and power specifications. The gate-level netlist is input to an Automatic Place and

Route tool, which creates a layout. The layout is verified and then fabricated on a chip.

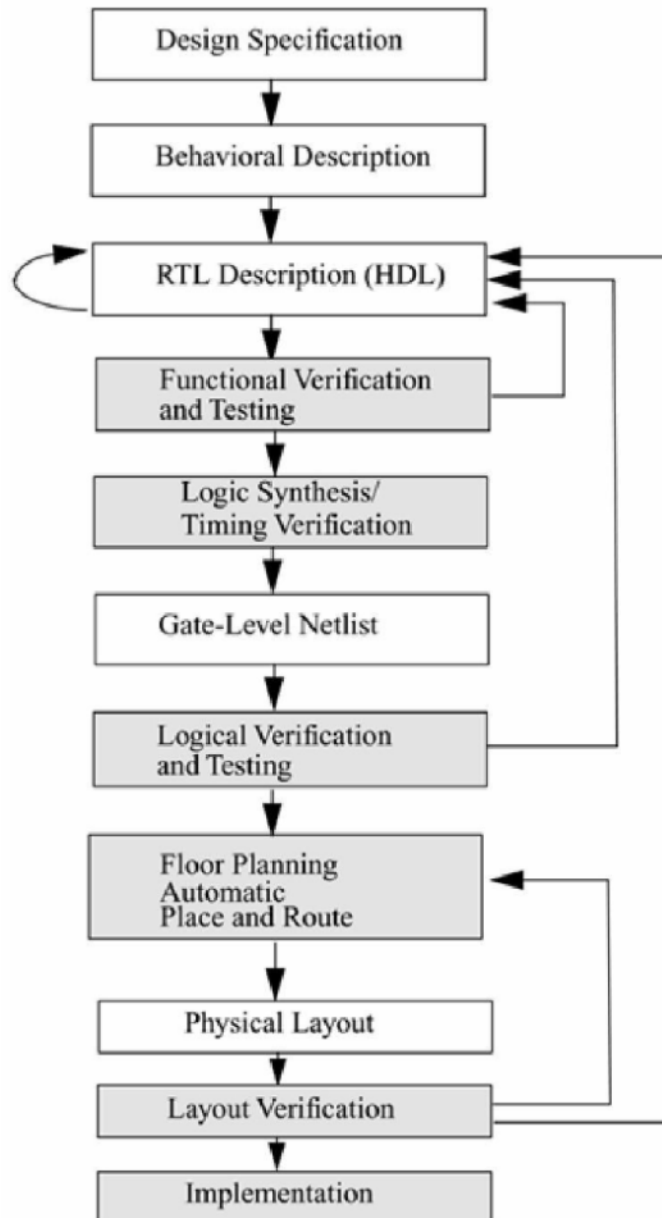


Figure 4.1: Typical Design Flow

Design Methodologies

There are two basic types of digital design methodologies: a top-down design methodology and a bottom-up design methodology. In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build the top-level block. We further subdivide the sub-blocks until we come to

leaf cells, which are the cells that cannot further be divided. Figure 1.2 shows the top-down design process.

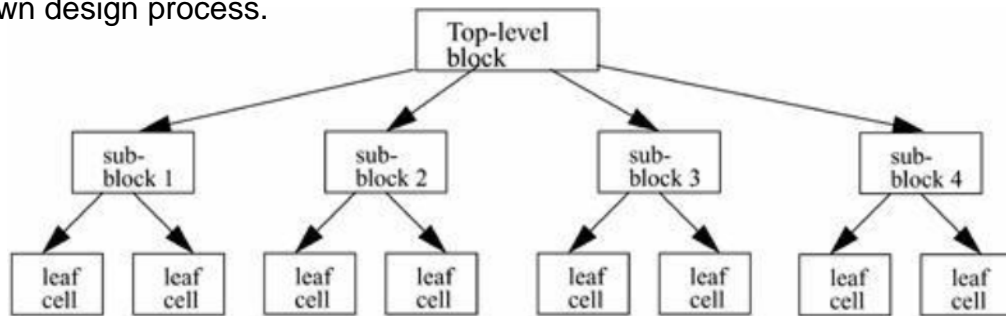


Figure 4.2: Top-down Design Methodology

In a bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells, using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design. Figure 1.3 shows the bottom-up design process

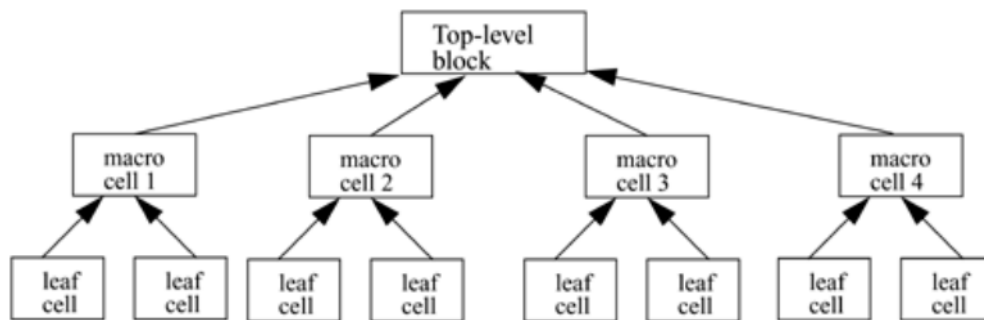


Figure 4.3: Bottom-up Design Methodology

Levels for design description

Verilog supports designing at many different levels of abstraction. Three of them are very important:

- Behavioral level
- Register-Transfer Level
- Gate Level

Behavioral Level

This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.

Register-Transfer Level

Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing bounds: operations are scheduled to occur at certain times. Modern RTL code definition is "Any code that is synthesizable is called RTL code".

Gate Level

Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z'). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.

Language Elements

Identifiers

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

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

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 */
```

Format

Verilog HDI is case sensitive. Identifiers differing only in their case are distinct. Verilog HDL, is free format, constructs may be written across multiple lines , or on one line. White space (newline, tab, and space characters) have no special significance.

System Tasks and Functions

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.

Compiler Directives

Compiler directives are provided in Verilog. All compiler directives are defined by using the '<keyword>' construct. We deal with the two most useful compiler directives.

'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
```

'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.

Example 'include Directive

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

Two other directives, 'ifdef and 'timescale, are used frequently.

Value set

Verilog supports four values and eight strengths to model the functionality of real hardware.

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

Strength 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

Data types

Verilog HDL has two groups of data types

(i) Net type

A net type represents a physical connection between structural elements. Its value is determined from the value of its drivers such as a continuous assignment or a gate output. If no driver is connected to a net, the net defaults to a value of z.

(ii) Variable type

A variable type represents an abstract data storage element. It is assigned values only within an always statement or an initial statement, and its value is saved from one assignment to the next. A variable type has a default value of x.

Net types

Here are the different kinds of nets that belong to the net data type

- wire
- tri
- wor
- trior
- wand
- triand
- trireg
- tri1
- tri0
- sup

ply0

- sup
- ply1

Variable types

There are five different kinds of variable types

- reg
- integer
- time
- real
- realti
- me

Register

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.

Example of Register

```
reg reset; // declare a variable reset that can hold its value
begin
    reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
    #100 reset = 1'b0; // after 100 time units reset is de asserted.
end
```

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
```

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
```

Expressions

An expression is formed using operands and operators. An expression can be used wherever a value is expected.

Operands

Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls.

```
integer count, final_count;
final_count = count + 1; //count is an integer operand
real a, b, c;
c = a - b; //a and b are real operands
reg [15:0] reg1,
reg2; reg [3:0]
reg_out;
reg_out = reg1[3:0] ^ reg2[3:0]; //reg1[3:0] and reg2[3:0] are
//part-select register
operands
```

```
reg ret_value;
ret_value = calculate_parity(A, B); //calculate_parity is a
//function type
operand
```

Operator Types

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. The table 1.1 shows the complete listing of operator symbols classified by category.

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
		logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two

Equality	==	equality	two
	!=	inequality case	two
	===	equality	two
	!==	case inequality	two
Bitwise	~	bitwise	one
	&	negation	two
		bitwise and	two
	^	bitwise or	two
	&	reduction and	one
	~&	reduction nand	one
Reduction		reduction or	one
	~	reduction nor	one
	>>	Right shift	Two
Shift	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Module

The basic unit of description in Verilog is the module. A module describes the functionality or structure of a design and also describes the ports through which it communicates externally with other modules. The structure of a design is described using switch-level primitives, gate-level primitives and user-defined primitives; data flow behavior of a design is described using continuous assignments; sequential behavior is described using procedural constructs. A module can also be instantiated inside another module.

```
module module_name (  
    port_list ); Declarations:  
    reg, wire,  
    parameter, input,  
    output, inout,  
    function , task, ....  
Statements :  
    Initial  
    statement  
    Always  
    statement  
    Module  
    instantiation  
    Gate  
    instantiation UDP  
    instantiation  
    Continuous  
    assignment  
    Generate statement  
endmodule
```

GATE PRIMITIVES:

In the example below, two Gate Instantiations create instances of and gate and not gate primitives.

```
module gateinst (a, b, c, d, x, y);  
input  a, b, c, d;  
output x, y;  
and and1 (x, a, b, c);  
not not1 (y, d);  
endmodule
```

In the and Gate Instantiation, the values x, a, b, and c are mapped to the corresponding terminals of the gate primitive. The connection to the output terminal is listed before the connections to the input terminals.

In the not Gate Instantiation, the connection to the output terminal is listed before the connection to the input terminal, i.e., the output terminal is mapped to y, and the input terminal is mapped to d.

TIMING CONTROL:

The procedural timing control is used to determine when statements should be executed.

Delay control:

```
#delay  
#(min:typ:max delay)
```

Event type declaration:

```
event identifier;
```

Event trigger:

```
-> event_identifier;
```

Event control:

```
@(event_identifier)
```

@(posedge identifier)

@(negedge identifier)

@(event_expression or event_expression)

Wait statement:

wait (expression) statement

Description

The Verilog HDL has two types of timing controls: delay control (*Example 1*) and event control (*Example 2*).

The delay control specifies the time between encountering and executing the statement. The delay control can be specified as a simple delay and as min:typ:max delay.

The named event (*Example 3*) is a special data type that does not hold any value. The event can be triggered using special characters -> followed by an event identifier. Any occurrence of an event trigger can be noticed using an event control statement.

An event control specifies the event that is required to resume execution of a statement. Event can be one of the following:

- Named event
- Change of a signal value
- Positive or negative edge occurred on signal (posedge, negedge)
- List of above-mentioned events (separated by or - event or operator)

A posedge is any transition from 0, x, and z to 1, and from 0 to z or x.

A negedge is any transition from 1, x, and z to 0, and from 1 to z or x.

The wait statement (*Example 4*) will suspend execution of all statements until the expression becomes true.

Examples

Example 1

#10;

The next statement will be executed after 10 time units.

#10 a = 5;

Assignment to a variable will be delayed by 10 time units.

```
#(1:2:3);
```

Delay control with min:typ:max delay value.

```
#(5:3:7) a = 5;
```

Assignment to a variable delayed by min:typ:max delay value.

Example 2

```
@ready a = a + 1;
```

The 'a' variable will be incremented by 1 when a change occurs on the 'ready' variable.

```
@(posedgeclk) q = d;
```

The 'd' variable will be assigned to 'q' on the positive edge of clk.

```
@(aorborcord)
```

```
y = (a | b) & (~c ^ d);
```

A new value will be assigned to the 'y' variable when a change occurs on any of the variables a, b, c, or d.

Example 3

```
event e;
```

Event declaration.

```
Initial
```

```
begin
```

```
#10;
```

```
->e;
```

```
end
```

Example of event triggering.

```
always @e d = 0;
```

Example of waiting for an event.

Example 4

```
wait(a);
```

```
i=i+1;
```

```
wait (!a);
```


This sequence of statements will wait until 'a' becomes 1, and then the next statement will be executed. Next, execution will be suspended until 'a' becomes 0.

PROCEDURAL AND CONDITIONAL ASSIGNMENTS

Procedural Assignments

Procedural assignments update values of reg, integer, real, or time variables. The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value.

The syntax for the simplest form of procedural assignment is shown below.

```
assignment ::= variable_lvalue = [  
                delay_or_event_control ] expression
```

The left-hand side of a procedural assignment <lvalue> can be one of the following:

- A reg, integer, real, or time register variable or a memory element
- A bit select of these variables (e.g., addr[0])
- A part select of these variables (e.g., addr[31:16])
- A concatenation of any of the above

There are two types of procedural assignment statements: blocking and nonblocking.

Blocking Assignments

Blocking assignment statements are executed in the order they are specified in a sequential block. A blocking assignment will not block execution of statements that follow in a parallel block.

The = operator is used to specify blocking assignments.

Example: Blocking

```
Statements reg x, y, z;  
reg [15:0] reg_a,  
reg_b; integer count;
```

```
//All behavioral statements must be inside an initial or always block  
initial begin
```

```
x = 0; y = 1; z = 1; //Scalar assignments count = 0;
//Assignment to integer variables
reg_a = 16'b0; reg_b = reg_a; //initialize vectors
#15 reg_a[2] = 1'b1; //Bit select assignment with delay
#10 reg_b[15:13] = {x, y, z} //Assign result of concatenation to // part
select of a vector
count = count + 1; //Assignment to an integer (increment)
end
```

In the above example, the statement $y = 1$ is executed only after $x = 0$ is executed. The behavior in a particular block is sequential in a begin-end block if blocking statements are used, because the statements can execute only in sequence. The statement $\text{count} = \text{count} + 1$ is executed last. The simulation times at which the statements are executed are as follows:

- All statements $x = 0$ through $\text{reg}_b = \text{reg}_a$ are executed at time 0
- Statement $\text{reg}_a[2] = 0$ at time = 15
- Statement $\text{reg}_b[15:13] = \{x, y, z\}$ at time = 25
- Statement $\text{count} = \text{count} + 1$ at time = 25
- Since there is a delay of 15 and 10 in the preceding statements, $\text{count} = \text{count} + 1$ will be executed at time = 25 units

Nonblocking Assignments

Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block. A \leq operator is used to specify nonblocking assignments. Note that this operator has the same symbol as a relational operator, `less_than_equal_to`. The operator \leq is interpreted as a relational operator in an expression and as an assignment operator in the context of a nonblocking assignment. To illustrate the behavior of nonblocking statements and its difference from blocking statements, let us consider the following example, where we convert some blocking assignments to nonblocking assignments, and observe the behavior.

Example : Nonblocking Assignments

```
reg x, y, z;
reg [15:0] reg_a,
reg_b; integer count;

//All behavioral statements must be inside an initial or always
block initial
begin
x = 0; y = 1; z = 1; //Scalar assignments count = 0;
//Assignment to integer variables
reg_a = 16'b0; reg_b = reg_a; //Initialize vectors
reg_a[2] <= #15 1'b1; //Bit select assignment with delay
reg_b[15:13] <= #10 {x, y, z}; //Assign result of concatenation
//to part select of a vector count <= count + 1; //Assignment to an
integer (increment)
end
```

In this example, the statements $x = 0$ through $reg_b = reg_a$ are executed sequentially at time 0. Then the three nonblocking assignments are processed at the same simulation time.

1. $reg_a[2] = 0$ is scheduled to execute after 15 units (i.e., time = 15)
2. $reg_b[15:13] = \{x, y, z\}$ is scheduled to execute after 10 time units (i.e., time = 10)
3. $count = count + 1$ is scheduled to be executed without any delay (i.e., time = 0)

Thus, the simulator schedules a nonblocking assignment statement to execute and continues to the next statement in the block without waiting for the nonblocking statement to complete execution. Typically, nonblocking assignment statements are executed last in the time step in which they are scheduled, that is, after all the blocking assignments in that time step are executed.

In the example above, we mixed blocking and nonblocking assignments to illustrate their behavior. However, it is recommended that blocking and nonblocking assignments not be mixed in the same always block.

Conditional Statements

Conditional statements are used for making decisions based upon certain conditions. These conditions are used to decide whether or not a statement should be executed. Keywords if and else are used for conditional statements. There are three types of conditional statements.

```
//Type 1 conditional statement. No else  
statement. //Statement executes or does not  
execute.
```

```
if (<expression>) true_statement ;  
//Type 2 conditional statement. One else statement  
//Either true_statement or false_statement is  
evaluated if (<expression>) true_statement ; else  
false_statement ;
```

```
//Type 3 conditional statement. Nested if-else-if.  
//Choice of multiple statements. Only one is  
executed. if (<expression1>) true_statement1 ;  
else if (<expression2>)  
true_statement2 ; else if  
(<expression3>) true_statement3 ;  
else default_statement ;
```

The <expression> is evaluated. If it is true (1 or a non-zero value), the true_statement is executed. However, if it is false (zero) or ambiguous (x), the false_statement is executed.

case Statement

The keywords case, endcase, and default are used in the case statement..

```
case (expression)  
    alternative1:  
        statement1;  
    alternative2:
```

```
    statement2;  
    alternative3:  
    statement3;  
    ...  
    ...  
default:  
    default_statement;  
endcase
```

Each of statement1, statement2 , default_statement can be a single statement or a block of multiple statements. A block of multiple statements must be grouped by keywords begin and end. The expression is compared to the alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is executed. If none of the alternatives matches, the default_statement is executed. The default_statement is optional. Placing of multiple default statements in one case statement is not allowed. The case statements can be nested.

```
//Execute statements based on the ALU control signal reg  
[1:0] alu_control;  
...  
..  
case  
(alu_control)  
2'd0 : y = x +  
z; 2'd1 : y = x -  
z; 2'd2 : y = x *  
z;  
default : $display("Invalid ALU control  
signal"); endcase
```

Example Program

Dataflow Modelling:

Dataflow modelling in Verilog allows a digital system to be designed in terms of it's function. Dataflow modelling utilizes Boolean equations, and uses a number of operators that can act on inputs to produce outputs operators like + - &&& ! ~ || | <<> {}

2 to 4 decoder in dataflow modelling would be like this

```
module decoder2_4 ( a ,b ,w ,x ,y ,z );  
output w ;  
output x ;  
output y ;  
output z ;  
input a ;  
input b ;  
assign w = (~a) & (~b);  
assign x = (~a) & b;  
assign y = a & (~b);  
assign z = a & b;  
end module;
```

Behavioral Modelling

The Behavioral modelling in Verilog is used to describe the function of a design in an algorithmic manner.

example for behavioural modelling,

```
module decoder2_4 ( din ,dout );  
output [3:0] dout ;  
reg [3:0] dout ;  
input [1:0] din ;  
wire [1:0] din ;  
always @ (din) begin  
if (din==0)
```

```
dout = 8;  
else if (din==1)  
dout = 4;  
else if (din==2)  
dout = 2;  
else  
dout = 1;  
end  
end module
```

Structural modelling:

In structural data flow modelling, digital design functions are defined using components such as an inverter, a MUX, an adder, a decoder, basic digital logic gates etc.. It is like connecting and arranging different parts of circuits available to implement

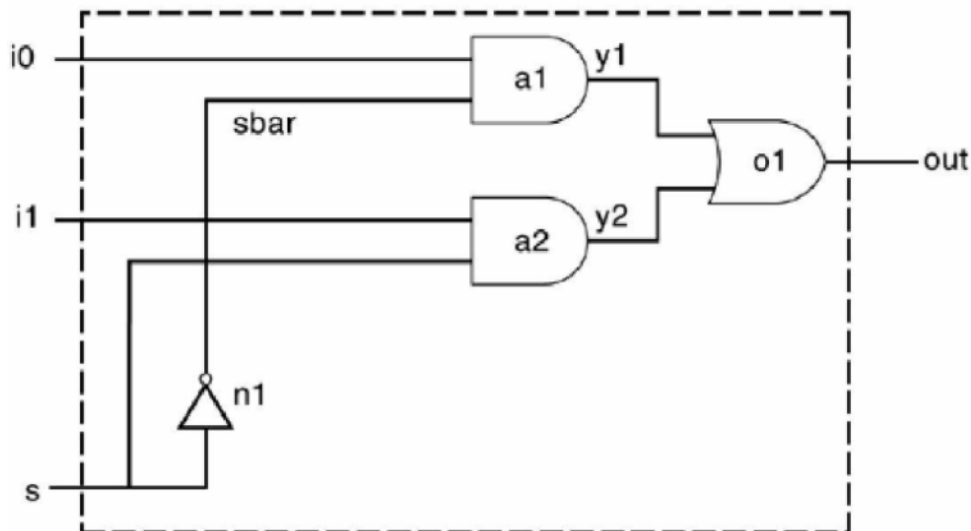


Figure 4.4: 2-to-1 Multiplexer

Verilog Description of 2-to-1 Multiplexer

```
module mux2_to_1(out, i0, i1, s);  
output out;  
input i0, i1;  
input s;  
wire sbar, y1, y2;
```

```
not n1(sbar, s);  
and a1(y1, i0,  
sbar);  
and a2(y2, i1, s);  
or o1(out, y1,  
y2);  
end module;
```

Questions For Practice

PART-A

1. List out the levels for design description in Verilog HDL.
2. Define RTL design.
3. List out the language elements are used in Verilog HDL.
4. What are the data types in Verilog HDL.
5. Define Wait Statements.
6. What are the types of operators in verilog HDL.
7. Define Gate primitives.
8. What are the types of assignment statements in verilog HDL.

PART-B

1. Write a Short note on the following.
A).Behavioral level,B).Register-Transfer Level,C).Gate Level
2. Explain in Detail about the DATA Types in Verilog HDL.
3. Explain in Detail about the Types operators in Verilog HDL
4. Explain in Detail about Procedural And Conditional Assignments.
5. Write a 4X1 Mux Verilog program Using Structural, Behavioral and Data flow model.