**SUB: DDVL**

**FACULTY NAME: D SURESH**

**---------------------------------------------------------------------------------------------------------------------**

**UNIT-I SHORT QUESTIONS**

**1. Write the difference between Verilog & VHDL.**  
 **Compilation**  
**VHDL**. Multiple design-units (entity/architecture pairs), that reside in the same system file, may be separately compiled if so desired. However, it is good design practice to keep each design unit in it's own system file in which case separate compilation should not be an issue.  
  
**Verilog**. The Verilog language is still rooted in it's native interpretative mode. Compilation is a means of speeding up simulation, but has not changed the original nature of the language. As a result care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files. Simulation results can change by simply changing the order of compilation.  
  
**Data types**  
**VHDL**. A multitude of language or user defined data types can be used. This may mean dedicated conversion functions are needed to convert objects from one type to another. The choice of which data types to use should be considered wisely, especially enumerated (abstract) data types. This will make models easier to write, clearer to read and avoid unnecessary conversion functions that can clutter the code. VHDL may be preferred because it allows a multitude of language or user defined data types to be used.  
  
**Verilog**. Compared to VHDL, Verilog data types a re very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user. There are net data types, for example wire, and a register data type called reg. A model with a signal whose type is one of the net data types has a corresponding electrical wire in the implied modeled circuit. Objects, that is signals, of type reg hold their value over simulation delta cycles and should not be confused with the modeling of a hardware register. Verilog may be preferred because of it's simplicity.  
  
**Design reusability**  
**VHDL**. Procedures and functions may be placed in a package so that they are avail able to any design-unit that wishes to use them.  
  
**Verilog**. There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module. To make functions and procedures generally accessible from different module statements the functions and procedures must be placed in a separate system file and included using the `include compiler directive

**2. Write about module.**

A module is the basic building block in verilog.It can be an element or a collection of lower level blocks. Typically elements are grouped into modules to provide common functionality that is used at many places in the design. A module provides the necessary functionality to the higher level blocks through its port interface, but hide the internal implementation. Every verilog code starts with module and ends with endmodule.

Syntax:

module <module name> (<module terminal list>);

…..

<module internals>

….

…

endmodule

**3. What are the components of a module.**

****

**4. Write about instance.**

* A module provides a template which you can create actual objects.
* When a module is invoked, Verilog creates a unique object from the template
* The process of creating a object from module template is called instantiation
* The object is called instance

module parallel adder\_4bit(a,b,ci,s,cout);

…..

……

**//instances of full adder**

Fa f1(a[0],b[0],ci,s[0],c[1]);

Fa f1(a[1],b[1],c[1],s[1],c[2]);

Fa f1(a[2],b[2],c[2],s[2],c[3]);

Fa f1(a[3],b[3],c[3],s[3],cout);

endmodule

**5. Explain the data types in Verilog.**

* Nets
* Registers
* Parameters
* ***Net* data type** represent physical connections between structural entities.

A *net* must be driven by a driver, such as a gate or a continuous assignment.

Verilog automatically propagates new values onto a *net* when the drivers change value.

wire, tri, wor, trior, wand, triand, tri0, tri1, supply0, supply1 and trireg are net data types.

* ***Registers*** represent abstract storage elements.

A *register* holds its value until a new value is assigned to it.

*Registers* are used extensively in behavior modeling and in applying stimuli.

reg, integer, time and real are register data types.

* ***Parameters*** are not variables, they are constants.

**6.What is the difference between wire and reg?**  
  
**Net types**: (wire,tri)Physical connection between structural elements. Value assigned by a continuous assignment or a gate output.

**Register type**: (reg, integer, time, real, real time) represents abstract data storage element. Assigned values only within an always statement or an initial statement.

The main difference between wire and reg is wire cannot hold (store) the value when there no connection between a and b like a->b, if there is no connection in a and b, wire loose value. But reg can hold the value even if there in no connection. Default values:wire is Z,reg is x.

**7. Explain system tasks.**

System tasks  are used to generate input and output during simulation. Their names begin with a dollar sign ($).

**$display, $strobe, $monitor**  
These commands have the same syntax, and display their values as text on the screen during simulation. They are much less convenient than waveform display tools like cwaves® or Signalscan®. $display and $strobe display once every time they are executed, whereas $monitor displays every time one of its parameters changes. The difference between $display and $strobe is that $strobe displays the parameters at the very end of the current simulation time unit. The format string is like that in C/C++, and may contain format characters. Format characters include %d (decimal), %h (hexadecimal), %b (binary), %c (character), %s (string) and %t (time). Append b, h, o to the task name to change default format to binary, octal or hexadecimal.

**$time, $stime, $realtime**  
These return the current simulation time as a 64-bit integer, a 32-bit integer, and a real number, respectively.

**$reset, $stop, $finish**  
$reset resets the simulation back to time 0; $stop halts the simulator and puts it in the interactive mode where the user can enter commands; $finish exits the simulator back to the operating system.

**$deposit**  
$deposit sets a net to a particular value.

**8. Briefly explain the compiler directives.**

All compilr directives are defined by using the ‘<keyword> construct.

`define - defines a compiler time constant or macro

`ifdef, `else, `endif - conditional compilation

`include - text inclusion

‘timescale-to specifies time unit.

**9. Briefly explain the levels of abstraction in Verilog.**

* Switch Level: Module implemented with switches and interconnects. Lowest level of Abstraction.
* Gate Level: Module implemented in terms of logic gates like (and, or, not etc...) and interconnection between gates.
* Dataflow Level: Module designed by specifying dataflow. The designer is aware of how data flows between hardware registers and how the data is processed in the design
* Behavioral Level: Module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Very similar to C programming

**10. Write about switch level modeling.**

The lowest level of abstraction for a digital HDL would be the **switch level**, which refers to the ability to describe the circuit as a netlist of transistor switches. A more detailed modeling scheme that can catch some additional electrical problems when transistors are used in this way. Now, little-used because circuits generally aren’t built this way.

**Switch Primitives:**

|  |  |
| --- | --- |
| **Unidirectional Primitives** | **Bidirectional Primitives** |
| **cmos nmos pmos pullup rcoms rnmos rpmos pulldown** | **tran tranif0 tranif1 rtran rtranif0 rtranif1** |
|  |  |

**Example 1: CMOS Inverter**

|  |  |
| --- | --- |
| http://coep.vlab.co.in/userfiles/5/image/inverter.gif | **module** cmos\_inv (In, Out); // module declaration  **input** In;  **output** Out;  **supply1**vdd;  **supply0** gnd;  **pmos** p1(Out, vdd, In); // pmos instantiation  **nmos** n1(Out, gnd, In); // nmoc instantiation  **endmodule** |
| **Fig. CMOS Inverter** | **Verilog code** |

**11. Write about Gate level modeling.**

A slightly higher level of abstraction would be the **gate level**, which refers to the ability to describe the circuit as a netlist of primitive logic gates and functions. The gates have one scalar output and multiple scalar inputs.

**Gate Primitives:**

|  |  |  |
| --- | --- | --- |
| **Gate Type** | **Description** | **Instantiation Syntax** |
| and | N-input AND gate | **and**a1(out,in1,in2) ; |
| nand | N-input NAND gate | **nand** a2(out,in1,in2) ; |
| or | N-input OR gate | **or**  a3(out,in1,in2) ; |
| nor | N-input NOR gate | **nor** a4(out,in1,in2) ; |
| xor | N-input XOR gate | **xor** x1(out,in1,in2) ; |
| xnor | N-input XNOR gate | **xnor**  x2(out,in1,in2) ; |
| not | 1-input NOT gate | **not** g1 (out , in) ; |
| buf | 1-input & N-output BUF gate | **buf**b1\_2 (out1,out2, in) ; |
| bufif1 | 1-input,1-output,1-control   BUF gate | **bufif1** b0(out,in,control) ; |
| bufif0 | 1-input,1-output,1-control BUF gate | **bufif0** b1(out,in,control) ; |
| notif1 | 1-input,1-output,1-control NOT gate | **notif1** b2(out,in,control) ; |
| notif0 | 1-input,1-output,1-control NOT gate | **notif0** b3(out,in,control) ; |

**Example 2: 2-1 Multiplexer**

We will design 2-1 Multiplexer with one select signal. **S0** is a selected signal wire. The I/O diagram and truth table for the Multiplexer are shown below.

|  |  |
| --- | --- |
| http://coep.vlab.co.in/userfiles/5/image/2-1mux.JPG | **module** mux2\_1 (o1,in1,in2,S0); //module  **output** o1;  **input** in1,in2,S0;  **and**a1 (Q, in1, S0); //and instantiation  **not** n1 (P,S0);         // not instantiation  **and**a2 (R, in2, P);  // and instantiation  **or**    o1 (o1, Q, R); // or instantiation  **endmodule** |
| **Fig. Multiplexer 2:1** | **Verilog Code** |

**12. Write about data flow modeling.**

It is the most basic statement in dateflow level, used to drive a value onto a net. It replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction.

A continuous assignment statement starts with the keyword **assign**.

**Syntax :** **assign**[ delay ] net = expression;

**Example :** **assign**sum = a ^ b;

**II.** **Implicit Continuous Assignment:**

Instead of declaring a net and then writing a continuous assignment on the net. Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration assignment per net because a net is declared only once.

**Example: wire** out= in1 & in2;

**Example 3: 2-4 Line Decoder**

|  |  |
| --- | --- |
| http://coep.vlab.co.in/userfiles/5/image/line%20decoder.png | **module** deco2\_4 (EN, A0, A1, D0, D1, D2, D3);//module  **input** EN, A0, A1;  **output** D0, D1, D2, D3;  **assign** D0 =(EN & ~A1 & ~A0);  **assign**D1 =(EN & ~A1 & A0);  **assign** D2 =(EN & A1 & ~A0);  **assign** D3 =(EN & A1 & A0);  **endmodule** |
| **Fig.  Line Decoder (2-4)** | **Verilog Code** |

**13. Write about Behavioral modeling.**

Verilog provides the designer the ability to describe the design functionality in an algorithmic manner. In other words the designer describes the ***behavior***of the circuit. The abstraction in this modeling is as simple as writing the logic in **C** language. Verilog behavioral models contain procedural statements that control the simulation and manipulate variables of the data types previously described. The activity starts at the control constructs initial and always. Each initial statement and each always statement starts a separate activity flow. All of the activity flows are concurrent, allowing the user to model the inherent concurrence of hardware.

**Procedural assignments:**

Procedural assignments are used for updating *reg*, *integer*, *time*, *real*, *realtime*, and *memory*data types.

The left hand side of a procedural assignment could be:

* *reg*, *integer*, *real*, *realtime*, or *time*data type.
* Bit-select of a *reg*, *integer*, or *time*data type, rest of the bits are untouched
* Part-select of a *reg*, *integer*, or *time*data type, rest of the bits are untouched.
* Memory word.

**Syntax:**wire out= in1 & in2;

**I.** **Blocking assignments:**

Blocking assignment statements are executed in the order they are specified in a sequential block.

**Example: initial**

**begin**

                a = 1 ;

                b = #35 ;

**end**

**II.** **Non-blocking assignments:**

The nonblocking assignment allows assignment scheduling without blocking the procedural flow.

**Syntax:  initial**

**begin**

a<=1 ;

b<= # 35 ;

**end**

* **Conditional (if-else) statement:**

        The condition (if-else) statement is used to make a decision whether a statement is executed or not. The keywords *if*and *else*are used to make conditional statement.

**Syntax:  if (condition\_1)**

**statement 1;**

**else if (condition\_2)**

**begin**

**statement 2;**

end

else

**statement 3;**

* **Case statement:**

        The case statement is a multi-way decision statement that tests whether an expression matches one of the expressions and branches accordingly. Keywords *case*and *endcase*are used to make a case statement. The case statement syntax is as follows.

**Syntax:  case (expression)**

**case 1: statement\_1;**

**case 2: statement\_2;**

**case 3: statement\_3;**

**default: default\_statement;**

**endcase**

* **Loop statement:**

 There are four types of looping statements in Verilog:

* for loop
* while
* forever
* Repeat

**14. Write about stimulus block (Test bench).**

Once the design block is completed, it must be tested. The functionality of the design block can be tested by using stimulus block. The stimulus block is also called as **test bench.**

Two styles of stimulus application are possible. In the first style, the stimulus block instantiate the design block and directly drives the signals in the design block. The second style of applying stimulus is to instantiate both the stimulus and design blocks in a top-level dummy module.

**15. Write the syntax of if-else statement.**

**Syntax:  if (condition\_1)**

**statement 1;**

**else if (condition\_2)**

**begin**

**statement 2;**

end

else

**statement 3;**

**16. Write the syntax of case statement.**

**Syntax:  case (expression)**

**case 1: statement\_1;**

**case 2: statement\_2;**

**case 3: statement\_3;**

**default: default\_statement;**

**endcase**

**17.Write about the blocking assignment in behavioral modeling.**

Blocking assignment statements are executed in the order they are specified in a sequential block.

**Example: initial**

**begin**

                a = 1 ;

                b = #35 ;

**end**

**18.Write about the Non blocking assignment in behavioral modeling.**

The nonblocking assignment allows assignment scheduling without blocking the procedural flow.

**Syntax:  initial**

**begin**

a<=1 ;

b<= # 35 ;

**end**

**19. Explain Bitwise operators in verilog.**

Bitwise operators:

Logical bit-wise operators take two single or multiple operands on either side of the operator and return a single bit result. The only exception is the **NOT** operator, which negates the single operand that follows. Verilog does not have the equivalent of **NAND** or **NOR** operator, their funstion is implemented by negating the **AND** and **OR** operators.

**module**Bitwise (A, B, Y);

**input**[6:0] A;

**input**[5:0] B;

**output**[6:0] Y;

**reg**[6:0] Y;

**always**@(A **or**B)

**begin**

                                  Y(0)=A(0)&B(0); //binary AND

                                  Y(1)=A(1)|B(1); //binary OR

                                  Y(2)=!(A(2)&B(2)); //negated AND

                                  Y(3)=!(A(3)|B(3)); //negated OR

                                  Y(4)=A(4)^B(4); //binary XOR

                                  Y(5)=A(5)~^B(5); //binary XNOR

                                  Y(6)=!A(6); //unary negation

**end**

**endmodule**

**20. Explain the Reduction operators**:

Verilog has six reduction operators, these operators accept a single vectored (multiple bit) operand, performs the appropriate bit-wise reduction on all bits of the operand, and returns a single bit result. For example, the four bits of A are **AND**ed together to produce Y1.

**module** Reduction (A, Y1, Y2, Y3, Y4, Y5, Y6);

**input**[3:0] A;

**output**Y1, Y2, Y3, Y4, Y5, Y6;

**reg**Y1, Y2, Y3, Y4, Y5, Y6;

**always**@(A)

**begin**

                                  Y1=&A; //reduction AND

                                  Y2=|A; //reduction OR

                                  Y3=~&A; //reduction NAND

                                  Y4=~|A; //reduction NOR

                                  Y5=^A; //reduction XOR

                                  Y6=~^A; //reduction XNOR

**end**

**endmodule**

**UNIT-II SHORT QUESTIONS**

### 1. Write the difference between continuous and procedural assignments.

|  |  |
| --- | --- |
| **Continuous assignment** | **Procedural assignment** |
| 1. It is used in dataflow modeling. | 1. It is used in Behavioural modelling. |
| 2. The continuous assignment statement is used to infer combinatorial logic. | 2. The procedural assignment statement is used to infer the combinatorial as well as the sequential logic including flip flops and latches. |
| 3. The continuous assignment statement assigns value primarily to nets. | 3. The procedural assignment statement assigns values primarily to reg element. |
| 4. The variables are driven to the output continuously to the output in the continuous assignment statement. | 4. In the procedural assignment statement, the results of the calculation are stores in a variable. |

**2. Explain the structured procedural statements.**

Structured procedures provide a means of modeling blocks of procedural statements.

## Simplified Syntax

**always** statement

**initial**statement

**function**

**task**

## Description

Functions and tasks are described in the section: Task and Functions.

The **initial** statement (*Example 1*) is executed only during a simulation run. The always procedural block statement (*Example 2*) is executed continuously during simulation, i.e. when the flow of program reaches the last statement in the block, the flow continues with the first statement in the block.

The **always** statement should contain at least one procedural timing control because otherwise it may hang the simulation.

Module definition can contain more than one **initial** or **always** statement.

Care must be taken when same reg type variables are used in multiple procedural blocks, initial or always. This is because these blocks run in parallel and changing or assigning to one variable affects the same variable in another parallel block.

**3. Write about initial statement**

An initial block, as the name suggests, is executed only once when simulation starts. This is useful in writing test benches. If we have multiple initial blocks, then all of them are executed at the beginning of simulation.

**initial begin**

**clk = 0;**

**reset = 0;**

**req\_0 = 0;**

**req\_1 = 0;**

**end**

In the above example, at the beginning of simulation, (i.e. when time = 0), all the variables inside the begin and end block are driven zero.

**4. Write about always statement.**

As the name suggests, an always block executes always, unlike initial blocks which execute only once (at the beginning of simulation). A second difference is that an always block should have a sensitive list or a delay associated with it.

The sensitive list is the one which tells the always block when to execute the block of code, as shown in the figure below. The @ symbol after reserved word ' always', indicates that the block will be triggered "at" the condition in parenthesis after symbol @.

One important note about always block: it cannot drive wire data type, but can drive reg and integer data types.

**always @ (a or b or sel)**

**begin**

**y = 0;**

**if (sel == 0) begin**

**y = a;**

**end else begin**

**y = b;**

**end**

**end**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

**5. Write the difference between blocking and non-blocking assignments.**

|  |  |
| --- | --- |
| **Blocking assignments** | **Non-blocking assignments** |
| |  | | --- | |  | | |  | | --- | |  | | 1. A blocking statement must be executed before the execution of the statements that follow it in a sequential block. In the example below the first time statement to get executed is a = b followed by   ../images/tidbits/blocki11.gif  **// Blocking assignments**  **initialbegin**  **a = #10 1'b1;// The simulator assigns 1 to a at time 10**  **b = #20 1'b0;// The simulator assigns 0 to b at time 30**  **c = #40 1'b1;// The simulator assigns 1 to c at time 70**  **end** | |  |  |  | |  |  | |  |  |  | | 1. Nonblocking statements allow you to schedule assignments without blocking the procedural flow. You can use the nonblocking procedural statement whenever you want to make several register assignments within the same time step without regard to order or dependence upon each other. It means that nonblocking statements resemble actual hardware more than blocking assignments.   **// Nonblocking assignments**  **initialbegin**  **d <= #10 1'b1;// The simulator assigns 1 to d at time 10**  **e <= #20 1'b0;// The simulator assigns 0 to e at time 20**  **f <= #40 1'b1;// The simulator assigns 1 to f at time 40**  **end** |

**6. Explain timing controls in verilog.**

Timing control statements are required in simulation to advance time. The time at which procedural statements will get executed shall be specified using timing controls. Delay based, event based and level sensitive timing controls are available in Verilog. Each of these are discussed below.

### 1.Delay Based Timing Control

In this, timing control is achieved by specifying waiting time to execution, when the statement is encountered. The symbol “#” is used to specify the delay. There are 3 ways, delay based timing control can be specified.

#### Regular Delay Control

In this delay will be specified on the left of the procedural assignment as a non-zero number. In the below code, execution of line 11 to 14 statements are controlled by some delays. The statement b = 1 will be executed at time 10 and others at time 30,35 and 45. Refer simulation result for more details.

     //Regular Delay Control

     #10 b = 1;

     #20 c = a + b;

     #5 a = c;

#### Intra-assignment Delay Control

In this case, delays will be specified on the right hand side of the assignment operation. The RHS expression will be evaluated at the current time and the assignment will be occurred only after the delay. The statement a = #5 (b + c) and rval = #20 L\_DELAY + a – 4 are intra-assignment delay control statements.  
Value of b and c are evaluated at time 45, but assignment to ‘a’ will happen only at time 50. The same way L\_DELAY + a – 4 will be calculated at time 50 and assignment to rval will happen at time 70.

     //Intra assignment

     a=#5 (b + c);

     rval=#20 L\_DELAY + a - 4;

#### Zero Delay Control

Zero delay control statement specifies zero delay value to LHS of procedural assignment. This is a method to ensure that the statement is executed at the end of that simulation time. That means, zero delay control statement is executed only after all other statements in that simulation time are executed. Here, a,b,c are executed different values in two initial block, one using normal assignment and other using zero delay. This example shows that how assignments will happen in the simulation

     #0 a = 1;

     #0 b = 1;

     #0 c = 1;

### 2.Event Based Timing Control

In this, execution of a statement or a block of a statement is controlled by an event. Regular, named and event OR control are different types of event based controls.

#### Regular event control

Execution of statement will happen on changes in signal or at a positive or negative transitions of signals. For example posedge of clock, negedge of reset etc.

eg:

@(posedge clock)out=in;

@(clock)z=n<<2;

#### Named Event Control

Event will be declared using keyword ‘event’ and is triggered by using the symbol ‘ -> ‘. In the below example the event ‘received’ is declared and is triggered when pkt\_done is high. Once event is triggered, always block at line 23 will get executed

module event\_control(in,ctrl,clock,out);

input[31:0]in;

input[2:0]ctrl;

input clock;

output[31:0]out;

reg[31:0]out;

wire pkt\_done;

 event received;

  assign pkt\_done=^ctrl;

   always@(posedge clock)

begin

    if(pkt\_done)begin

      $display("Event : Triggered");

       ->received;

    end

end

#### Event OR control

The transitions of signal or event can trigger the execution of statements as shown below

always@(clock orin)//Wait for clock or in to change

 OR

 always@(clock,in)

### 3.Level Sensitive Timing Control

This waits for a certain condition to be true. In the below example, it waits for xor-ctrl bits to be 1, and then do the assignment to out.

module level\_control(in,ctrl,out);

 input[31:0]in;

input[2:0]ctrl;

output[31:0]out;

 reg[31:0]out;

 always begin

   wait(^ctrl)#2 out = in + 100;

end

endmodule

**7. Write about Delay based timing controls.**

In this, timing control is achieved by specifying waiting time to execution, when the statement is encountered. The symbol “#” is used to specify the delay. There are 3 ways, delay based timing control can be specified.

#### Regular Delay Control

In this delay will be specified on the left of the procedural assignment as a non-zero number. In the below code, execution of line 11 to 14 statements are controlled by some delays. The statement b = 1 will be executed at time 10 and others at time 30,35 and 45. Refer simulation result for more details.

     //Regular Delay Control

     #10 b = 1;

     #20 c = a + b;

     #5 a = c;

#### Intra-assignment Delay Control

In this case, delays will be specified on the right hand side of the assignment operation. The RHS expression will be evaluated at the current time and the assignment will be occurred only after the delay. The statement a = #5 (b + c) and rval = #20 L\_DELAY + a – 4 are intra-assignment delay control statements.  
Value of b and c are evaluated at time 45, but assignment to ‘a’ will happen only at time 50. The same way L\_DELAY + a – 4 will be calculated at time 50 and assignment to rval will happen at time 70.

     //Intra assignment

     a=#5 (b + c);

     rval=#20 L\_DELAY + a - 4;

#### Zero Delay Control

Zero delay control statement specifies zero delay value to LHS of procedural assignment. This is a method to ensure that the statement is executed at the end of that simulation time. That means, zero delay control statement is executed only after all other statements in that simulation time are executed. Here, a,b,c are executed different values in two initial block, one using normal assignment and other using zero delay. This example shows that how assignments will happen in the simulation

     #0 a = 1;

     #0 b = 1;

     #0 c = 1;

**8. Write about Event based timing controls.**

### Event Based Timing Control

In this, execution of a statement or a block of a statement is controlled by an event. Regular, named and event OR control are different types of event based controls.

#### Regular event control

Execution of statement will happen on changes in signal or at a positive or negative transitions of signals. For example posedge of clock, negedge of reset etc.

eg:

@(posedge clock)out=in;

@(clock)z=n<<2;

#### Named Event Control

Event will be declared using keyword ‘event’ and is triggered by using the symbol ‘ -> ‘. In the below example the event ‘received’ is declared and is triggered when pkt\_done is high. Once event is triggered, always block at line 23 will get executed

module event\_control(in,ctrl,clock,out);

 input[31:0]in;

input[2:0]ctrl;

input clock;

output[31:0]out;

 reg[31:0]out;

wire pkt\_done;

 event received;

  assign pkt\_done=^ctrl;

   always@(posedge clock)

begin

    if(pkt\_done)begin

      $display("Event : Triggered");

       ->received;

    end

end

always@(received)begin

   $display("Event : Received");

   out=in+100;

end

endmodule

#### Event OR control

The transitions of signal or event can trigger the execution of statements as shown below

always@(clock orin)//Wait for clock or in to change

 OR

 always@(clock,in)

**9. Write about level sensitive timing controls.**

### Level Sensitive Timing Control

This waits for a certain condition to be true. In the below example, it waits for xor-ctrl bits to be 1, and then do the assignment to out.

module level\_control(in,ctrl,out);

 input[31:0]in;

input[2:0]ctrl;

output[31:0]out;

 reg[31:0]out;

 always begin

   wait(^ctrl)#2 out = in + 100;

end

endmodule

**10. Explain the conditional statements in verilog.**

if-else Statements

if statements allows the tool to decide a statement is to be executed or not, depending on the conditions specified. General syntax is as follows:

**if**( *condition* )   
*statement;*

If the condition or conditional expression is true, then statement will be executed, otherwise not.

Consider the example

**if**( *hold == 0* )   
*counter = counter + 1;*

If reset is not zero, counter will be incremented.

**else** statement can be used with if optionally. If condition specified in if statement is false, statement after else will be executed. See this example, if reset is nonzero counter will become zero, othewise it will be incremented.

**if**( *reset* )   
*counter = 0;***else** *counter = counter + 1;*

 If there are more than one statements within an if block, we can combine them using begin -- end. We can also nest if-else statements as in these examples.

**if**( *reset* )   
**begin**  
  *counter <= 0;   
   over\_flow <= 0;***end****else if***( counter == 15 )***begin**  
  *counter <= 0;   
   over\_flow <= 1;***end   
else   
begin***counter <= counter + 1;   
   over\_flow <= 0;* **end**

if-else statements should be used inside initial or always blocks. Generally if-else statements generates multiplexers while synthesizing.

Here is a full Verilog code example using if else statements. This is a adder/ subtracter with 'addnsub' signal to control addition and subtraction.

|  |
| --- |
| **module** addsub (a, b, addnsub, result);  **input**[7:0] a;  **input**[7:0] b;  **input** addnsub;  **output**[8:0] result;  **reg**[8:0] result;  **always**@(a **or** b **or** addnsub)  **begin**  **if**(addnsub)  result = a + b;  **else**  result = a - b;  **end**  **endmodule** |

If *addnsub* is true( nonzero ), result will be a+b, otherwise result will be a-b

**11. Explain about Case statement in verilog.**

The case statement starts with a **case** or **casex** or **casez** keyword followed by the case expression (in parenthesis) and case items or **default** statement. It ends with the **endcase** keyword. The default statement is optional and should be used only once. A case item contains a list of one or more case item expressions, separated by comma, and the case item statement. The case item expression and the case item statement should be separated by a colon.

During the evaluation of the case statement, all case item expressions are evaluated and compared in the order in which they are given. If the first case item expression matches the case expression, then the statement which is associated with that expression is executed and the execution of the case statement is terminated. If comparison fails, then the next case item expression is evaluated and compared with the case expression. If all comparisons fail and the **default** section is given, then its statements are executed. Otherwise none of the case items will be executed.

Both case expression and case item expressions should have the same bit length. None of the expressions are required to be a constant expression.

The case expression comparison is effective when all compared bits are identical. Therefore, special types of case statement are provided, which can contain don't-care values in the case expression and in the case item expression. These statements can be used in the same way as the case statement, but they begin with the keywords**casex** and **casez**.

The **casez** statement treats high-impedance (z) values as don't-care values and the **casex** statement treats high-impedance and unknown (x) values as don't care values. If any of the bits in the case expression or case item expression is a don't-care value then that bit position will be ignored.

The don't-care value can be also specified by the question mark (?), which is equal to z value.

**12. Explain loops in verilog.**

Loop statements provide a means of modeling blocks of procedural statements.

## Simplified Syntax

* **forever** statement;
* **repeat**(expression) statement;
* **while**(expression) statement;
* **for**(assignment; expression; assignment) statement;

## Description

There are four types of loop statements: forever, repeat, while, and for statements.

The **forever** instruction (Example 1) continuously repeats the statement that follows it. Therefore, it should be used with procedural timing controls (otherwise it hangs the simulation).

The **repeat** instruction (Example 2) executes a given statement a fixed number of times. The number of executions is set by the expression, which follows the repeat keyword. If the expression evaluates to unknown, high-impedance, or a zero value, then no statement will be executed.

The **while** instruction (Example 3) executes a given statement until the expression is true. If a **while** statement starts with a false value, then no statement will be executed.

The **for** instruction (Example 4) executes a given statement until the expression is true. At the initial step, the first assignment will be executed. At the second step, the expression will be evaluated. If the expression evaluates to an unknown, high-impedance, or zero value, then the **for** statement will be terminated. Otherwise, the statement and second assignment will be executed. After that, the second step is repeated.

**13. Write the difference between sequential & parallel block.**

|  |  |
| --- | --- |
| **Sequential block** | **Parallel block** |
| In the sequential, blocks, **begin** and **end** keywords are used to group the statements, All the statement in this group executes sequentially. ( this rule is not applicable for nonblocking assignments). If the statements are given with some timing/delays then the given delays get added into.   It would be clearer with following examples.  Example -1 –  reg a,b,c; initial begin      a = 1′b1;      b = 1′b0;      c = 1′b1; end The Example -1 is showing the sequential block without delays, All the statements written inside the begin-end will execute sequentially and after the execution of initial block, final values are a=1, b=0 and c=1  Example -2 –  reg a,b,c; initial begin      #5 a = 1′b1;      #10 b = 1′b0;      #15 c = 1′b1; end  The Example -2 is showing the sequential block with delays, In this case, the same statements are given with some delays, Since All the statements execute sequentially, the a will get value 1 after 5 time unit, b gets value after 15 time unit and c will take value 1 after 30 time unit | The statements written inside the parallel block, execute parallel, If the sequencing is required then it can be given by providing some delays before the statements. In parallel blocks, all the statements occur within **fork and join**  Example -3 –  reg a,b,c; initial fork      #5 a = 1′b1;      #10 b = 1′b0;      #15 c = 1′b1; join  Form Example -3, all the statements written inside the fork and join, executes parallel, it means the c with have value ‘1′ after 15 time unit, in case of sequential blocks it was 30 time unit ( example 2)  The fork and join statements can be nested with begin-end  Example -4 ( Nested block)  reg a,b,c,d; initial begin fork      #5 a = 1′b1;      #10 b = 1′b0;      #15 c = 1′b1; join   1. 30 d = 1′b0;   end  From Example -4, the initial block contains begin-end and fork-join both. In this case c takes value after 15 time unit, and d takes the value after 30 time unit |

**14. Explain the difference between a task & a function.**

|  |  |
| --- | --- |
| **Functions** | **Tasks** |
| 1. A function can enable other function but not other task. 2. Functions always execute in 0 simulation time. 3. Functions must not contain any delay, event or timing control statements. 4. A function must have at least one input argument. 5. A function returns a single value | 1. A task can enable other tasks and functions. 2. Tasks may execute in non- zero simulation time. 3. Tasks may contain any delay, event or timing control statements. 4. A task can have zero or more arguments of any type. 5. A task does not return a value, but can pass multiple values through **output** and **inout** arguments |

**15. Explain the Logical operators**

Logical comparison operators are used in conjuction with relational and equality operators as described in the relational operators section and equality and inequality operators section. They provide a means to perform multiple comparisons within a a single expression.

**module** Logical (A, B, C, D, E, F, Y);

**input**[2:0] A, B, C, D, E, F;

**output**Y;

**reg**Y;

**always**@(A **or**B **or**C **or**D **or**E **or**F)

**begin**

**if**((A==B) && ((C>D) || !(E<F)))

                                                   Y=1;

**else**

                                                   Y=0;

**end**

**endmodule**

**16. Explain the Concatenation and Replication operator.**

The concatenation operator "{ , }" combines (concatenates) the bits of two or more data objects. The objects may be scalar (single bit) or vectored (muliple bit). Mutiple concatenations may be performed with a constant prefix and is known as replication.

**module**Concatenation (A, B, Y);

**input**[2:0] A, B;

**output**[14:0] Y;

**parameter**C=3'b011;

**reg**[14:0] Y;

**always**@(A or B)

**begin**

                                  Y={A, B, (2{C}}, 3'b110};

**end**

**endmodule**

**17. Explain the Arithmetic operators.**

There are five arithmetic operators in Verilog.

**module** Arithmetic (A, B, Y1, Y2, Y3, Y4, Y5);

**input** [2:0] A, B;

**output** [3:0] Y1;

**output**[4:0] Y3;

**output**[2:0] Y2, Y4, Y5;

**reg**[3:0] Y1;

**reg** [4:0] Y3;

**reg** [2:0] Y2, Y4, Y5;

**always** @(A **or** B)

**begin**

                                  Y1=A+B;//addition

                                  Y2=A-B;//subtraction

                                  Y3=A\*B;//multiplication

                                  Y4=A/B;//division

                                  Y5=A%B;//modulus of A divided by B

**end**

**endmodule**

**18. Explain the Conditional operators.**

An expression using conditional operator evaluates the logical expression before the "?". If the expression is true then the expression before the colon (:) is evaluated and assigned to the output. If the logical expression is false then the expression after the colon is evaluated and assigned to the output.

**module**Conditional (Time, Y);

**input**[2:0] Time;

**output**[2:0] Y;

**reg**[2:0] Y;

**parameter**Zero =3b'000;

**parameter**TimeOut = 3b'110;

**always**@(Time)

**begin**

                                  Y=(Time!=TimeOut) ? Time +1 : Zero;

**end**

**endmodule**

**19. Explain the Shifting operators**

Shift operators require two operands. The operand before the operator contains data to be shifted and the operand after the operator contains the number of single bit shift operations to be performed. **0** is being used to fill the blank positions.

**module**Shift (A, Y1, Y2);

**input**[7:0] A;

**output**[7:0] Y1, Y2;

**parameter**B=3; reg [7:0] Y1, Y2;

**always**@(A)

**begin**

                                  Y1=A<<B; //logical shift left

                                  Y2=A>>B; //logical shift right

**end**

**endmodule**

**20. Explain the Equality operators.**

There are two types of Equality operators. Case Equality and Logical Equality.

|  |  |
| --- | --- |
| a === b | a equal to b, including x and z (Case equality) |
| a !== b | a not equal to b, including x and z (Case inequality) |
| a == b | a equal to b, result may be unknown (logical equality) |
| a != b | a not equal to b, result may be unknown (logical equality) |

|  |  |
| --- | --- |
|  | * Operands are compared bit by bit, with zero filling if the two operands do not have the same length * Result is 0 (false) or 1 (true) * For the == and != operators, the result is x, if either operand contains an x or a z * For the === and !== operators, bits with x and z are included in the comparison and must match for the result to be true   **Note:**The result is always 0 or 1. |

**UNIT-III SHORT QUESTIONS.**

**1. Explain the Relational operators.**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| a < b | a less than b |
| a > b | a greater than b |
| a <= b | a less than or equal to b |
| a >= b | a greater than or equal to b |

|  |
| --- |
| * The result is a scalar value (example a < b) * 0 if the relation is false (a is bigger than b) * 1 if the relation is true ( a is smaller than b) * x if any of the operands has unknown x bits (if a or b contains X) |
|  |  | space.gif |

**Note:**If any operand is x or z, then the result of that test is treated as false (0)

**2. Write the operator precedence in verilog.**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Symbols** | **Precedence** |
| Unary,  Multiply, Divide, Modulus | +,!, ~,  \*, /, % | **Highest precedence** |
| Add, Subtract,  Shift | +, - ,  <<, >> |  |
| Relation,  Equality | <, >, <= ,>=,  = =, !=, ===, !== |  |
| Reduction | &, !&,^,^~,|,~| |  |
| Logic | &&, || |  |
| Conditional | ? : | **Lowest precedence** |

**3. Write the difference between flip flop and latch.**

|  |  |
| --- | --- |
| **Flip-flop** | **Latch** |
| A flip-flop samples the inputs only at a clock event (rising edge, etc.) | A Latch samples the inputs continuously whenever it is enabled, that is, only when the enable signal is on. (or otherwise, it would be a wire, not a latch). |
| Flip-Flop are edge sensitive. | Latches are level sensitive. |
| Flip flop is sensitive to signal change and not on level. They can transfer data only at the single instant and data cannot be changed until next signal change. | Latch is sensitive to duration of pulse and can send or receive the data when the switch is on. |
| A flip-flop continuously checks its inputs and correspondingly changes its output only at times determined by clocking signal. | Latch is a device which continuously checks all its input and correspondingly changes its output, independent of the time determined by clocking signal. |
| It work’s on the basis of clock pulses. | It is based on enable function input |
| It is a edge triggered , it mean that the output and the next state input changes when there is a change in clock pulse whether it may a +ve or -ve clock pulse. | It is a level triggered, it mean that the output of present state and input of the next state depends on the level that is binary input 1 or 0. |
|  |  |

**4.Write a verilog code for D flip flop in behavioral modeling.**

module dff(d,clk,rst,q,qb);

Input d,clk,rst;

Output q,qb;

Reg q;

always@(posedge clk)

begin

if(rst)

q=0;

else

q=d;

end

assign qb=~q;

endmodule

**5.Write a verilog code for T flip flop in behavioral modeling.**

module tff(t,clk,rst,q,qb);

Input t,clk,rst;

Output q,qb;

Reg q;

always@(posedge clk)

begin

if(rst)

q=0;

else if(t)

q=~q;

else

q=q;

end

assign qb=~q;

endmodule

**6.Write a verilog code for JK- flip flop in behavioral modeling.**

module jk\_ff(j,k,clk,rst,q,qb);

input j,k,clk,rst;

output q,qb;

reg q;

always@(posedge clk)

begin

if(rst)

q=0;

else

case({j,k})

2’b00:q=q;

2’b01:q=0;

2’b10:q=1;

2’b11:q=~q;

endcase

end

assign qb=~q;

endmodule

**7. Write a verilog code for SR- flip flop in behavioral modeling.**

module jk\_ff(s,r,clk,rst,q,qb);

input s,r,clk,rst;

output q,qb;

reg q;

always@(posedge clk)

begin

if(rst)

q=0;

else

case({s,r})

2’b00:q=q;

2’b01:q=0;

2’b10:q=1;

2’b11:q=~q;

endcase

end

assign qb=~q;

endmodule

**8. Write about the Finite state machine.**

A **finite-state machine** (**FSM**) or **finite-state automaton** (**FSA**) is a mathematical [model of computation](https://en.wikipedia.org/wiki/Model_of_computation).  It is an [abstract machine](https://en.wikipedia.org/wiki/Abstract_machine) that can be in exactly one of a finite number of [*states*](https://en.wikipedia.org/wiki/State_(computer_science)) at any given time. The FSM can change from one state to another in response to some external [inputs](https://en.wikipedia.org/wiki/Input_(computer_science)); the change from one state to another is called a *transition*. An FSM is defined by a list of its states, its initial state, and the conditions for each transition.

FSM is classified into two types

(i).Moore FSM. (ii).Mealy FSM

**9. Write the difference between Synchronous sequential circuits and Asynchronous sequential circuits.**

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
| **Synchronous sequential circuits** | | **Asynchronous sequential circuits** |
| * These are easy to design. | | * These are difficult to design. |
| * A clocked flip flop acts as memory element. | | * An un clocked flip flop or time delay is used as memory element. |
| * They are slower as clock is involved. | | * They are comparatively faster as no clock is used here. |
| * The states of memory element is affected only at active edge of clock, if input is changed. | | * The states of memory element will change any time as soon as input is changed. |

**10. Write about Sequence detector.**

A sequence detector accepts as input a string of bits: either 0 or 1. Its output goes to 1 when a target sequence has been detected.

There are two basic types: overlap and non-overlap.

In an sequence detector that allows overlap, the final bits of one sequence can be the start of another sequence.

Example Sequence detector 11011

X (input) 1 1 0 1 1 0 1 1 0 1 1

Z (Output with overlap 0 0 0 0 1 0 0 1 0 0 1

Z (Output with no overlap) 0 0 0 0 1 0 0 0 0 0 1

**11. Write the difference between Moore and Mealy FSM.**

|  |  |
| --- | --- |
| **Mealy Machine** | **Moore Machine** |
| Output depends both upon present state and present input. | Output depends only upon the present state. |
| Generally, it has fewer states than Moore Machine. | Generally, it has more states than Mealy Machine. |
| Output changes at the clock edges. | Input change can cause change in output change as soon as logic is done. |
| Mealy machines react faster to inputs | In Moore machines, more logic is needed to decode the outputs since it has more circuit delays. |

**12. Write about one hot encoding.**

In [digital circuits](https://en.wikipedia.org/wiki/Digital_circuits), **one-hot** is a group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0).

|  |  |  |
| --- | --- | --- |
| **Binary** | **Gray code** | **One-hot** |
| 000 | 000 | 00000001 |
| 001 | 001 | 00000010 |
| 010 | 011 | 00000100 |
| 011 | 010 | 00001000 |
| 100 | 110 | 00010000 |
| 101 | 111 | 00100000 |
| 110 | 101 | 01000000 |
| 111 | 100 | 10000000 |

One-hot encoding is often used for indicating the state of a [state machine](https://en.wikipedia.org/wiki/State_machine). When using [binary](https://en.wikipedia.org/wiki/Binary_number) or [Gray code](https://en.wikipedia.org/wiki/Gray_code), a [decoder](https://en.wikipedia.org/wiki/Binary_decoder) is needed to determine the state. A one-hot state machine, however, does not need a decoder as the state machine is in the *n*th state if and only if the *n*th bit is high.

**13. Write the advantages and disadvantages of one-hot encoding.**

**Advantages**

* Determining the state has a low and constant cost of accessing one [flip-flop](https://en.wikipedia.org/wiki/Flip-flop_(electronics))
* Changing the state has the constant cost of accessing two flip-flops
* Easy to design and modify
* Easy to detect illegal states
* Takes advantage of an [FPGA](https://en.wikipedia.org/wiki/Field-programmable_gate_array)'s abundant flip-flops

Using a one-hot implementation typically allows a state machine to run at a faster clock rate than any other encoding of that state machine.[[2]](https://en.wikipedia.org/wiki/One-hot#cite_note-2)

**Disadvantages**

* Requires more flip-flops than other encodings, making it impractical for [PAL](https://en.wikipedia.org/wiki/Programmable_Array_Logic) devices

Many of the states are illegal

**UNIT-IV SHORT QUESTIONS**

**1.What is flow chart**

A flow chart is a convenient way to specify the sequence of procedural steps and decision paths for an algorithm. A flow chart for a hardware algorithm translates the word statement to an information diagram that enumerates the sequence of operations together with the conditions necessary for their execution.

**2.Define ASM Chart.**

A special flow chart that has been developed specifically to define digital hardware algorithms.

**3. What are the elements of ASM chat.**

(a).State box.

(b). Decision box

(c). Conditional output box.

An ASM chart has an entry point and is constructed with blocks.  A block is constucted with the following type of symbols.

|  |  |
| --- | --- |
| http://uhaweb.hartford.edu/kmhill/suppnotes/AsmChart/box1.png | One state box.  The state box has a name and lists outputs that are asserted when the system is in that state.  These outputs are called synchronous or *Moore* type outputs. |
| http://uhaweb.hartford.edu/kmhill/suppnotes/AsmChart/decision1.png | Optional decision box(es).  A decision box may be conditioned on a signal or a test of some kind. |
| http://uhaweb.hartford.edu/kmhill/suppnotes/AsmChart/output1.png | Optional conditional output box(es).  Such an ouput box indicates outputs that are conditionally asserted.  These outputs are called asynchrous or *Mealy* outputs. |

**4.Define State box.**

State name state code

A rectangle describes one state of the synchronous sequential digital system. It is similar to a circle representing a state of a state diagram. The main difference is that it only has one output transition (exit path). The state block symbol contains a listing of all unconditional actions and (Moore) outputs associated with that state. The outputs are updated concurrently when the state is entered after an active clock edge.

**5.Define decision box**.

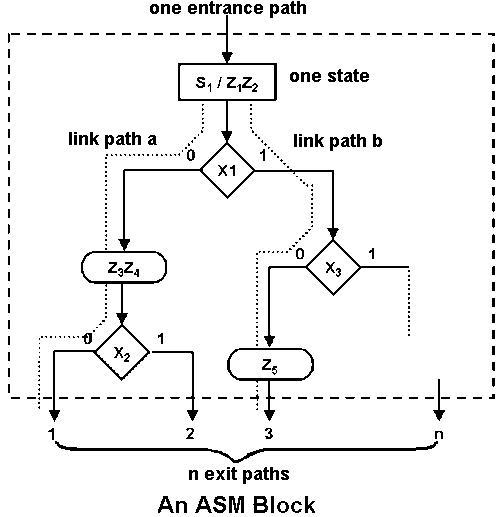
**False True**

A diamond shape contains the input condition on which depends the branching from a given state. It has one entry point and two exit path. Condition symbols can be concatenated.

**6.Define conditional box**.

An oval or rectangular with rounded edges represents an action, i.e. signal assignment or calculation, that is taken if an input condition is fulfilled. Conditional output boxes are only used to depict Mealy-type outputs. The entry path to the symbol is always from a decision symbol, but its exit path can be either to a state box or to another decision symbol.

**7.ASM Block**  
  
 An ASM block is a structure consisting of one state box and all the decision and conditional boxes connected to its exit path. An ASM block has one entrance path and one or more exit paths.



Write about Asynchronous sequential circuits.

**8.**  **What is state table?**

The state table representation of a sequential circuit consists of three sections labelled *present state*, *next state* and *output*. The present state designates the state of flip-flops before the occurrence of a clock pulse. The next state shows the states of flip-flops after the clock pulse, and the output section lists the value of the output variables during the present state.

**9.What is transition table?**

Transition table is useful to analyze an asynchronous circuit from the circuit diagram  Procedure to obtain transition table: 1. Determine all feedback loops in the circuits 2. Mark the input (yi ) and output (Yi ) of each feedback loop 3. Derive the Boolean functions of all Y’s 4. Plot each Y function in a map and combine all maps into one table 5. Circle those values of Y in each square that are equal to the value of y in the same row

**10.What is flow table.**

Similar to a transition table except the states are represented by letter symbols  Can also include the output values  Suitable to obtain the logic diagram from it

**11.Primitive flow table**

In a flow table if each row consists of only one stable state then it is called as Primitive flow table

**12.What are hazard free digital circuits?**

A circuit which has no hazard like static-0-hazard and static-1-hazard is called hazard free digital circuit.

**13.**           **What are Hazards?**

The unwanted switching transients (glitches) that may appear at the output of a circuit are called Hazards.

**14.**           **Distinguish between a flowchart and an ASM chart.**

   A conventional flow chart describes the square of procedural steps and decision paths for an algorithm without concern for their time relationship.

   The ASM chart describes the sequence of event as well as timing relationship between the states of a sequential controller and the events that occur while going from one state to the next.

**15.** **What is a state diagram? Give an example.**

A state diagram is a type of diagram used in computer science and related fields to describe the behavior of systems. State diagrams require that the system described is composed of a finite number of states; sometimes, this is indeed the case, while at other times this is a reasonable abstraction. Many forms of state diagrams exist, which differ slightly and have different semantics.

**UNIT-V SHORT QUESTIONS**

**1.**           **What are the advantages of static RAM and Dynamic Ram?**

**Static RAM:**

              Access time is less.

              Fast operation.

**Dynamic Ram**

              It consumes less power.

              Cost is low.

**2.**           **What is difference between PAL and PLA?**

PLA:

* Both AND and OR arrays are programmable and Complex
* Costlier than PAL



PAL:

* AND arrays are programmable OR arrays are fixed
* Cheaper and Simpler

**.**           **Compare Dynamic RAM with Static RAM.**

              Static Ram is very costly.

              Dynamic Ram is cheaper.

              Static Ram contains Transistors.

              Dynamic Ram contains Capacitors.

              Static Ram is used in L1 and L2 cache.

              Dynamic Ram is used in system RAM.

**4.**           **What is meant by memory Expansion? Mention its limit.**

The memory expansion can be achieved in two ways: by expanding word size and expanding memory capacity.

**Limitations:**

* Memory capacity upto 16Mbytes.
* 24 address lines and 16 data lines.

**5.**           **Mention few applications of PLA and PAL.**

              Implement combinational circuits

              Implement sequential circuits

              Code converters

              Microprocessor based systems

**6.**           **What are the different types of programmable logic devices?**

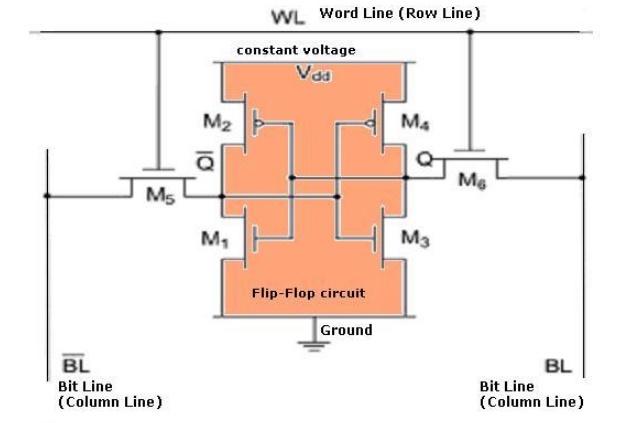
(a).SPLDs (Simple Programmable Logic Devices)

* ROM (Read-Only Memory)
* PLA (Programmable Logic Array)
* PAL (Programmable Array Logic)
* GAL (Generic Array Logic)

(b).CPLD (Complex Programmable Logic Device)

(c).FPGA (Field-Programmable Gate Array**)**

**7.**           **Draw the structure of a static RAM cell.**



**8.**           **List the advantages of PLDs.**

              low and fixed (two gate) propagation delays (typically down to 5 ns),

              simple,

              low-cost (free),

              design tools.

**9.**      **What is PAL?**

PAL is programmable array logic, PAL consists of a programmable AND array and a fixed OR array with output logic**.**

**10.**      **What is access time and cycle time of a memory?**

Access time is the maximum specified time within which a valid new data is put on the data bus after an address is applied.

Cycle time is the minimum time for which an address must be held stable on the address bus in read cycle.

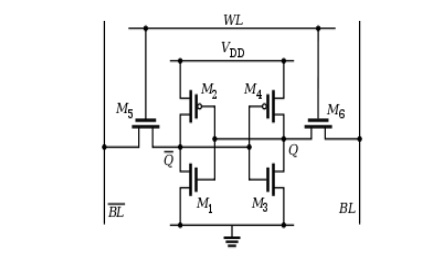
**11.**      **How the memories are classified?**

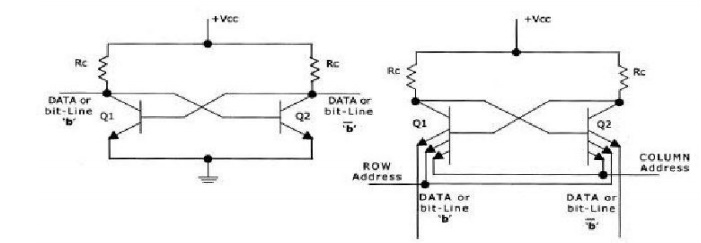
It is classified into two types:

              volatile

              non-volatile memory

**12.**      **Draw the logic diagram of a static RAM cell and Bipolar cell.**





**13.**    **What is volatile and non-volatile memory?**

* The memory which cannot hold the data when power is turned off is known as volatile memory.
* The memory which can hold the data when power is turned off is known as non-volatile memory

**14.**    **Give the advantages of RAM.**

              Read and write the data.

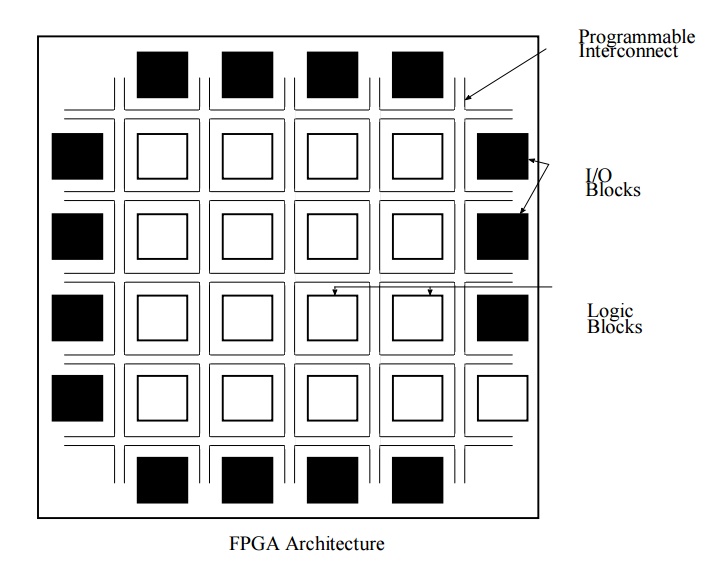
              Data is accessed by using address of the memory location.

              Higher speed.

**15. What is FPGA.**

A *field-programmable gate array* (FPGA) uses an array of logic blocks, which can be configured by the user. The term ‘field-programmable’ here signifies that the device is programmable outside the factory where it is manufactured. The internal architecture of an FPGA device has three main parts, namely the array of logic blocks, the programmable interconnects and the I/O blocks.

Each of the I/O blocks provides an individually selectable input, output or bidirectional access to one of the general-purpose I/O pins on the FPGA package. The logic blocks in an FPGA are no more complex than a couple of logic gates or a look-up table feeding a flip-flop. The programmable interconnects connect logic blocks to logic blocks and also I/O blocks to logic blocks.



**16. What is an ASIC.**

ASICs are silicon chips that have been designed for a specific application. Putting

in other words, it is a chip designed to perform a particular operation as opposed to

general purpose integrated circuits:

• An ASIC is NOT software programmable to perform different tasks.

• ICs that are not ASICs are :

– DRAM

– SRAM

– 74xx series ICs

• ICs which are ASICs:

– Baseband processor in mobile phone

– Chipsets in PCs

– MPEG encoders/ decoders

– DSP functions in hardware, e.g. FFT

**17.Difference between ASIC and FPGA.**

* An ASIC is a unique type of integrated circuit meant for a specific application while an FPGA is a reprogrammable integrated circuit.
* An ASIC can no longer be altered once created while an FPGA can.
* It is common practice to design and test on an FPGA before implementing on an ASIC.
* An ASIC wastes very little material compared to an FPGA and the recurring costs are low.

FPGA is better than an ASIC when building low volume production circuits.