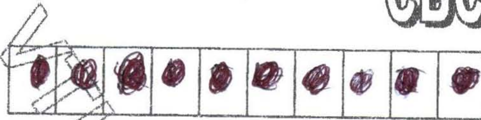


USN 15EC53

Fifth Semester B.E. Degree Examination, Dec.2017/Jan.2018
Verilog HDL

Time: 3 hrs. Max. Marks: 80

Note: Answer any FIVE full questions, choosing one full question from each module.

Important Note : 1. On completing your answers, compulsorily draw diagonal cross lines on the remaining blank pages.
2. Any revealing of identification, appeal to evaluator and /or equations written eg, 42+8 = 50, will be treated as malpractice.

VTU EXAMS DEC-JAN 2018 23/12/2017 12:55:11

Module-1

1 a. Explain a typical design flow for designing VLSI IC circuit using the block diagram. **(06 Marks)**

b. Explain top down design methodology and bottom up design methodology. **(10 Marks)**

OR

2 a. With a block diagram of 4-bit Ripple carry counter, explain the design hierarchy. **(10 Marks)**

b. Explain the trends in Hardware Description Languages (HDLs). **(06 Marks)**

Module-2

3 a. With a neat block diagram, explain the components of verilog module. **(06 Marks)**

b. Explain the following data types with an example in verilog:
(i) Nets (ii) Register (iii) Integers (iv) Real (v) Time Register. **(10 Marks)**

OR

4 a. Explain the port connection rules. **(06 Marks)**

b. Explain the two methods of connecting ports to external signals with an example. **(10 Marks)**

Module-3

5 a. What are Rise, Fall and Turn-off delays? How they are specified in verilog? **(06 Marks)**

b. Design a 2-to-1 multiplexer using bufifo and bufifl gates. The delay specification for these gates are as follows:

Delay	Min	Typ	Max
Rise	1	2	3
Fall	3	4	5
Turn-off	5	6	7

Write gate level description and stimulus in verilog. **(10 Marks)**

OR

6 a. Write a verilog dataflow level of abstraction for 4-to-1 multiplexer using conditional operator. **(06 Marks)**

b. Write a verilog dataflow description for 4-bit Full adder with carry lookahead. **(10 Marks)**

Module-4

7 a. Explain the blocking assignment statements and non-blocking assignment statements with relevant examples. **(08 Marks)**

b. Write a note on the following loop statements:
(i) While loop (ii) forever loop. **(08 Marks)**

OR

8 a. Explain sequential and parallel blocks with examples. (08 Marks)
b. Write a verilog program for 8-to-1 multiplexer using case statement. (08 Marks)

Module-5

9 a. Explain the synthesis process with a block diagram. (08 Marks)
b. Write a VHDL program for two 4-bit comparator using data flow description. (08 Marks)

OR

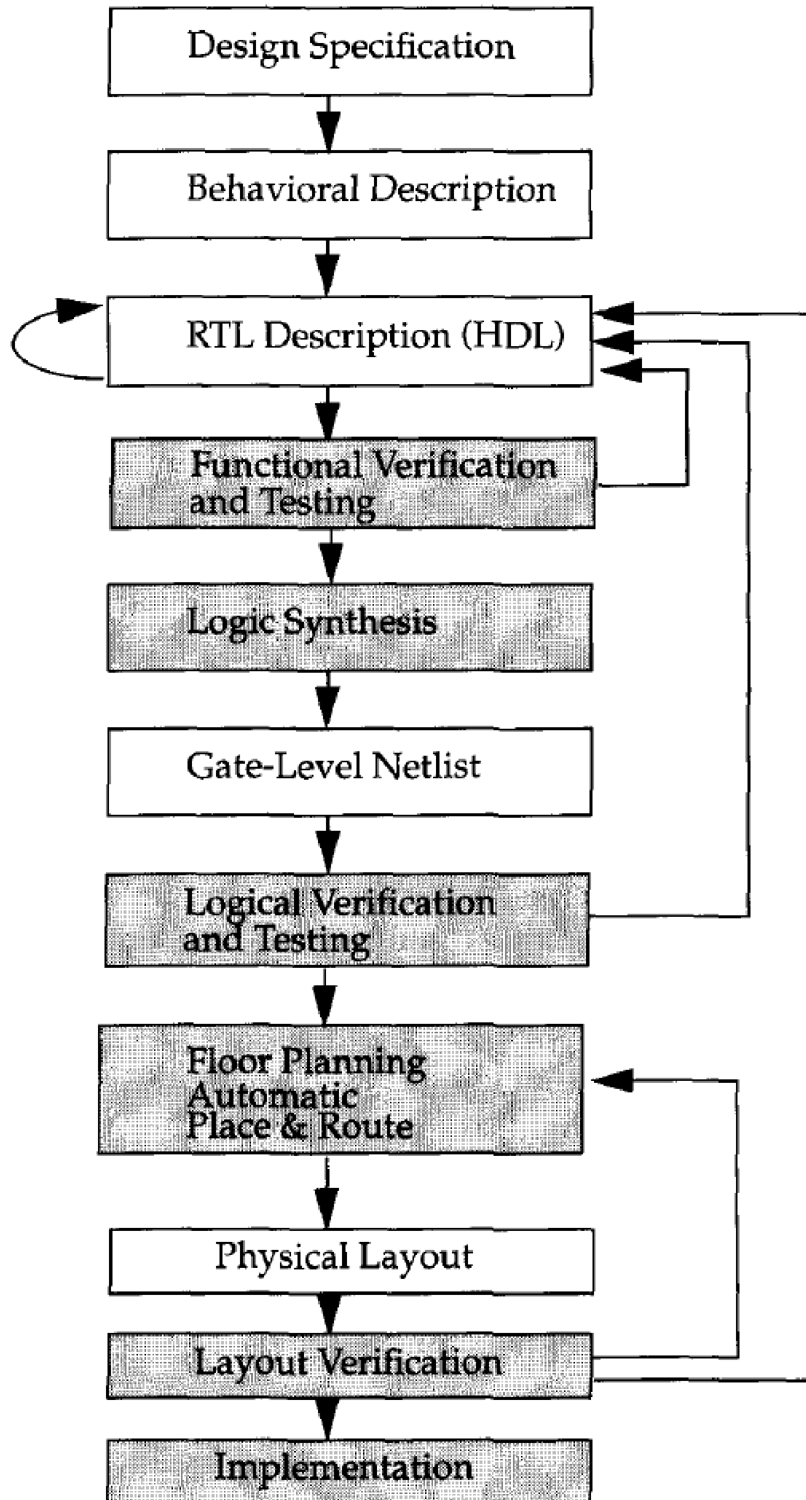
10 a. Explain the declaration of constant, variable and signal in VHDL with example. (08 Marks)
b. Write a VHDL program for half adder in behavioral description. (08 Marks)

* * * * *

VTU EXAMS DEC-JAN-18 23/12/2017 12:55:11
VTU EXAMS DEC-JAN-18 23/12/2017 12:55:11

1 a) Explain a typical design flow for designing VLSI IC circuits using the block diagram. 6 Marks

A typical design flow for designing VLSI IC circuits is shown in Figure below



03 marks

In any design, specifications are written first. Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. A behavioral description is then created to nalyse the design in terms of functionality, performance, compliance to standards, and other high-level issues. They are written using

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. 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. 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 chip. Thus, most digital design activity is concentrated on manually optimizing the RTL description of the circuit. Behavioral synthesis tools have begun to emerge recently. These tools can create RTL descriptions from a behavioral or algorithmic description of the circuit.

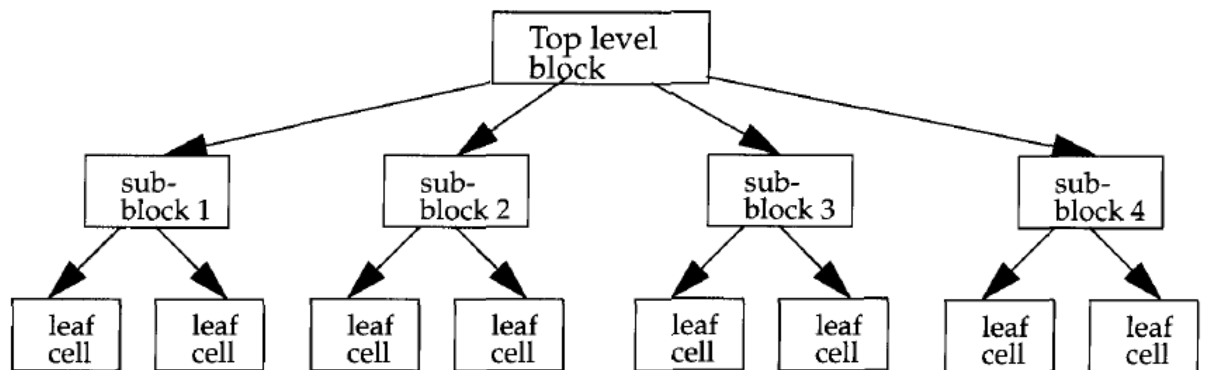
03 marks

2. b) Explain top down design methodology and bottom up Design methodology. 10 Marks

There are two basic types of digital design methodologies: a top-down design methodology and a bottom-up design methodology.

Top-Down design methodology:

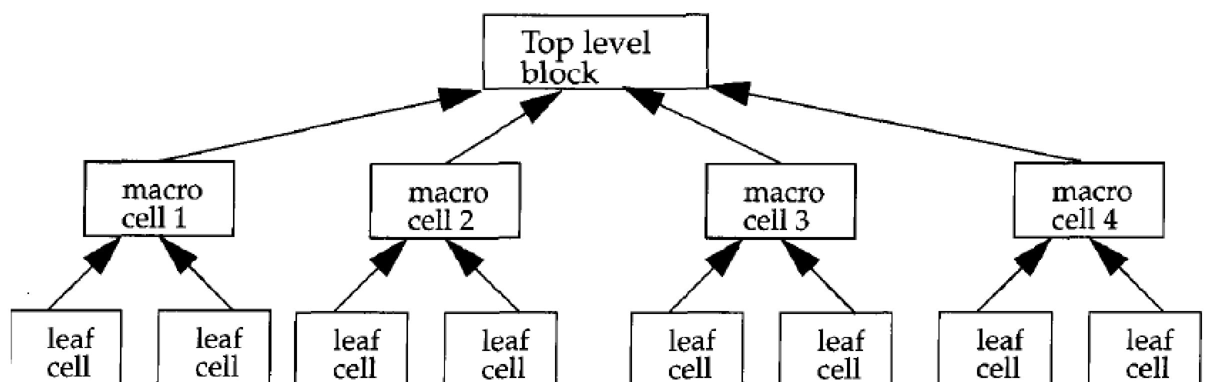
In a top-down design methodology, define the top-level block and identify the sub-blocks necessary to build the top-level block. Further subdivide the sub-blocks until leaf cells are available, which are the cells that cannot further be divided. Following figure shows the top-down design process:



04 marks

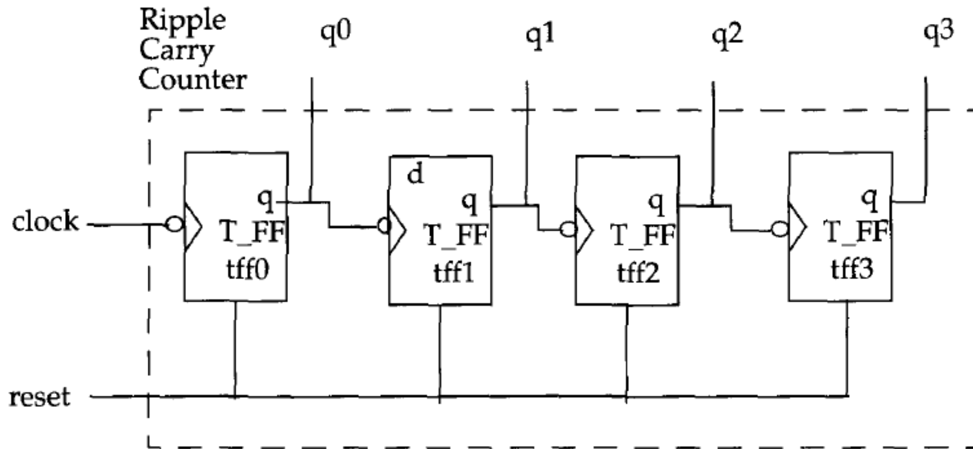
Bottom-up design methodology:

First identify the building blocks that are available. Then build bigger cells, using these building blocks. These cells are then used for higher-level blocks until the top-level block in the design is built. Following figure shows the bottom-up design process. Following figure shows the bottom-up design process:



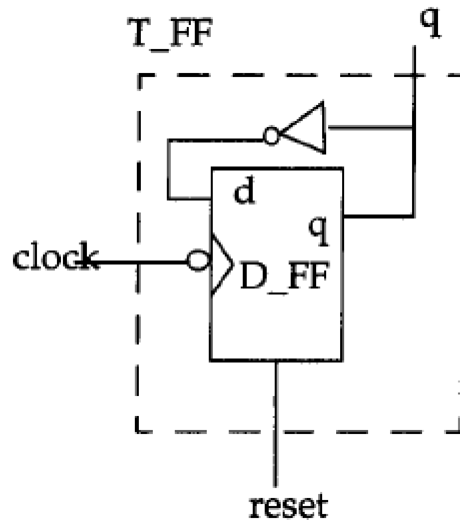
04 marks

The ripple carry counter shown in following figure:

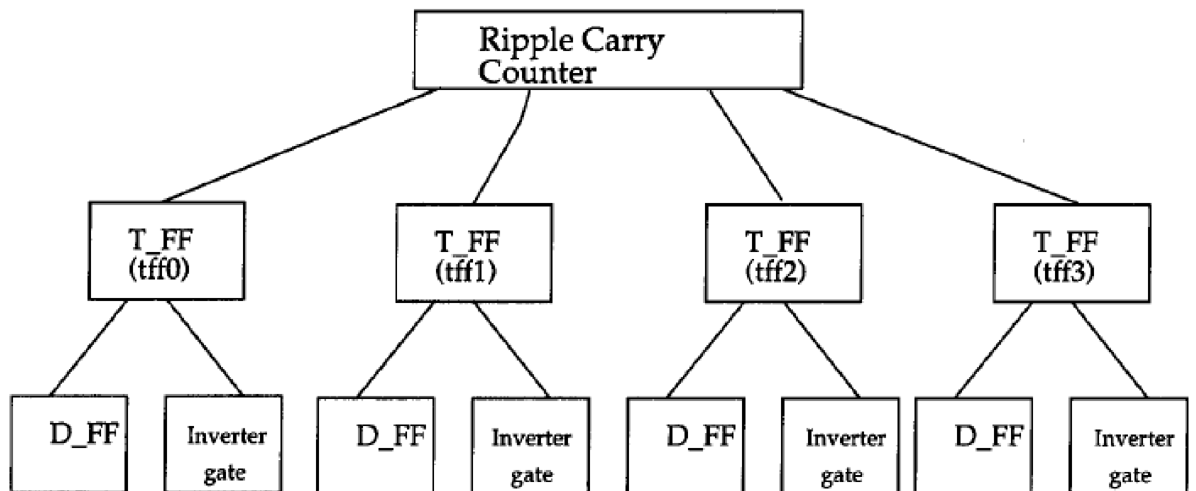


It is made up of negative edge triggered toggle flip-flops (T_FF). Each of the T-FFs can be made up from negative edge-triggered D-flipflops (D_FF) and inverters as shown in the following figure:

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0



Thus, the ripple carry counter is built in a hierarchical fashion by using building blocks. The diagram for the design hierarchy is:



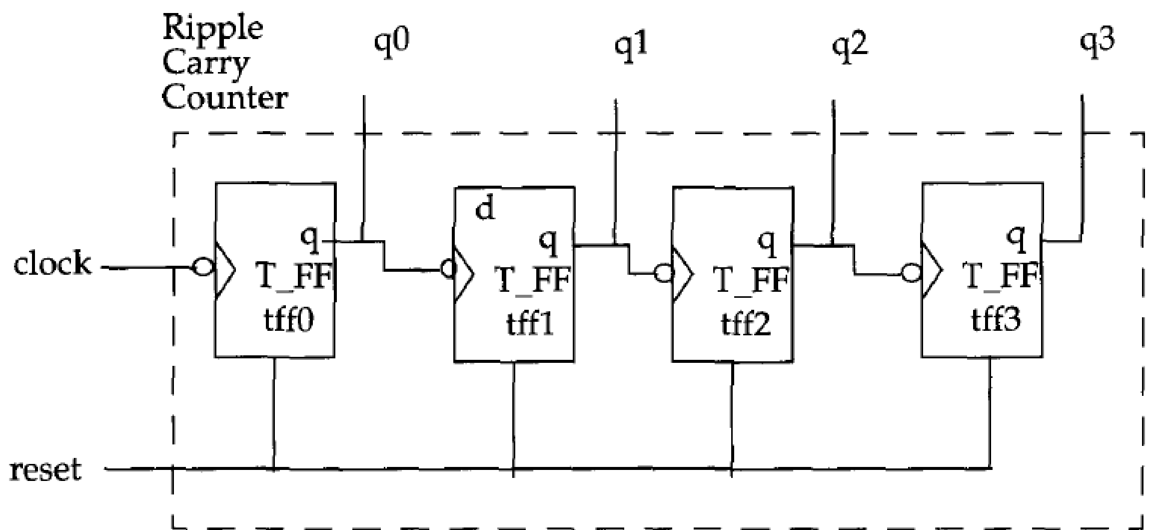
02 marks

2 a) With a block diagram of 4-bit ripple carry counter explain the design hierarchy. 10 Marks

Here top level module is 4-bit Ripple carry counter.
Ripple carry counter is constructed using T-flipflop.
Hence in the next level of hierarchy it consists of T-flipflop.
T-flipflop is constructed using D flipflop and an inverter.
Hence in the third level hierarchy consists of only
D-flipflops and an inverter. Dflipflop can be implemented
using ~~only~~ gates like and and inverter or it may
be designed using only switches. ~~The~~ If D-ff is implem
ented using only switches then D-ff is ~~the leaf~~
one of the leaf cell. Inverter is implemented using
switches. Hence the leaf cells of Ripple carry
counter is ~~only~~ ~~DFF~~ D-ff and an inverter.

02 marks

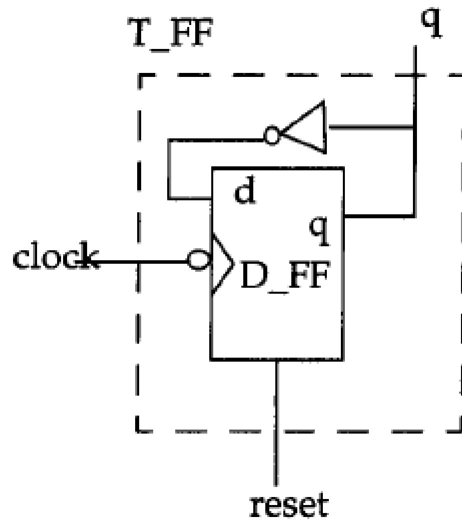
The ripple carry counter shown in following figure:



02 marks

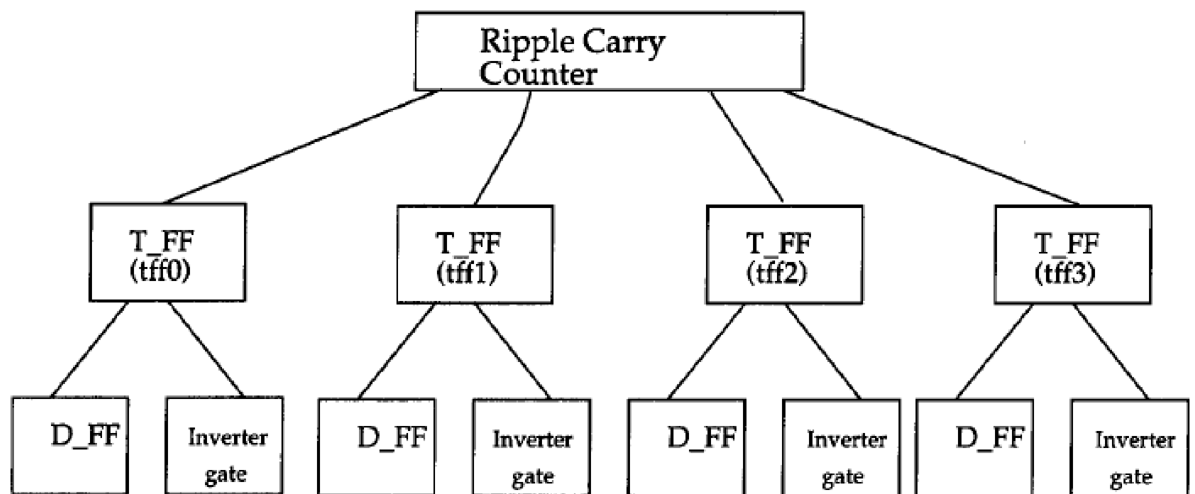
It is made up of negative edge triggered toggle flip-flops (T_FF). Each of the T-FFs can be made up from
negative edge-triggered D-flipflops (D_FF) and inverters as shown in the following figure:

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0



02 marks

Thus, the ripple carry counter is built in a hierarchical fashion by using building blocks. The diagram for the design hierarchy in top-down design methodology is:



04 marks

```
// Ripple Carry Counter Top Block
module ripple-carry-counter(q, clk, reset) ;
output [3:01 q;
input clk, reset;
T-FF tffO(q[O] ,clk, reset)
T-FF tffl(q[l] ,q[O], reset
T-FF tff2 (q[2] ,q[l], reset
T-FF tff3 (q[3l ,q[2], reset
Endmodule
//Flip-flop T-FF
module T-FF (q, clk, reset) ;
output q;
input clk, reset;
wire d;
D-FF dff0 (q, d, clk, reset) ;
not nl(d, q) ; / / not is a Verilog-provided primitive. case
sensitive
```

```
endmodule

//module D-FF with synchronous reset
module D-FF(q, d, clk, reset) ;
output q;
input d, clk, reset;
reg q;
always @(posedge reset or negedge clk)
if (reset)
q = l'bO;
else
q = d;
endmodule
```

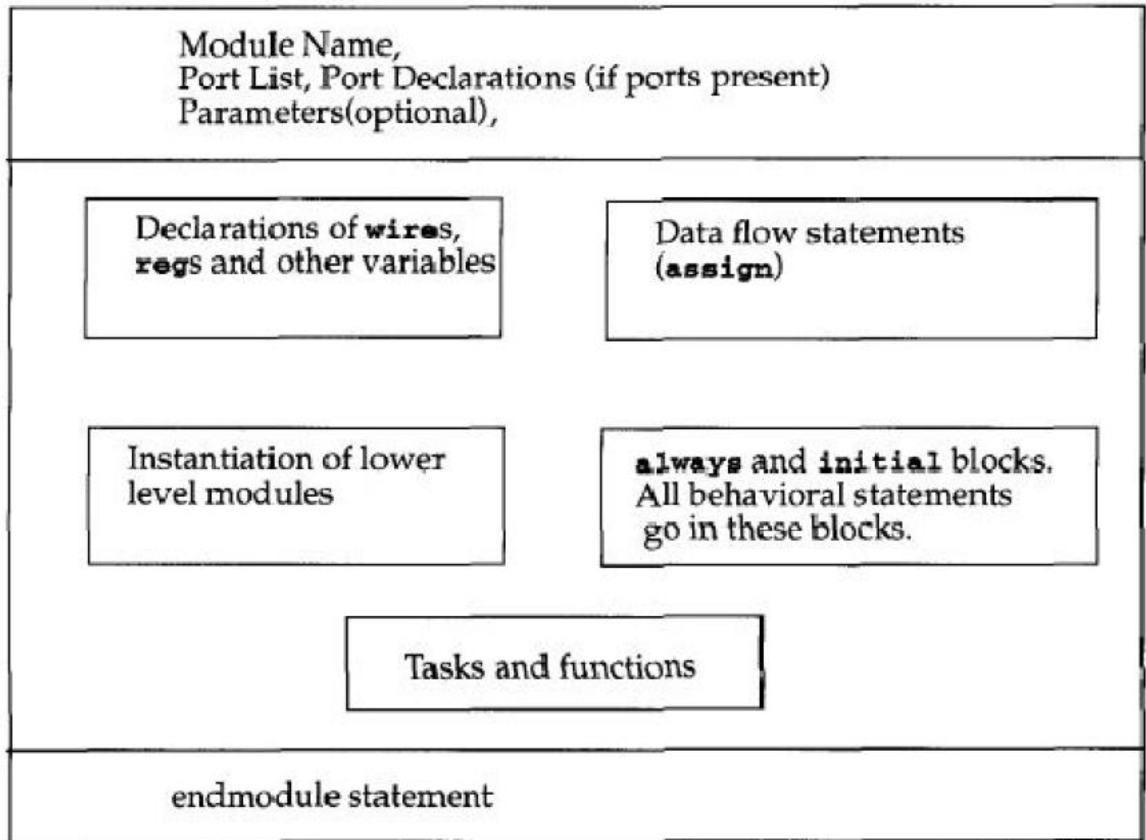
b) Explain the trends in Hardware Description Languages (HDLs) 06 Marks

1. Start design of digital circuits using HDL at an RTL level, because logic synthesis tools can create gate-level netlists from RTL level des
2. Behavioral synthesis helps designers to design directly in terms of algorithms and the behavior of the circuit, and then use CAD tools to do the translation and optimization in each phase of the design.
3. *Formal verification* techniques are also appearing on the horizon. Formal verification applies formal mathematical techniques to verify the correctness of Verilog HDL descriptions and to establish equivalency between RTL and gate-level netlists.
4. Designers can mix gate-level description directly into the RTL description to achieve optimum results.
5. System-level design can be a mixed bottom-up methodology where the designers use either existing Verilog HDL modules, basic building blocks, or vendor-supplied core blocks to quickly bring up their system simulation. This is done to reduce development costs and compress design schedules.

06 marks

3 a) With a neat block diagram, explain the components of Verilog module. 06 Marks

The components of Verilog HDL program is as shown below:



03 marks

A module definition always begins with the keyword **module.** The *module name, port list, port declarations,* and optional *parameters* must come first in a module definition. *Port list* and *port declarations* are present only if the module has any ports to interact with the external environment. The five components within a module are - *variable declarations, dataflow statements, instantiation of lower modules, behavioral blocks,* and *tasks* or *functions.* These components can be in any order and at any place in the module definition. The **endmodule** statement must always come last in a module definition. All components except **module,** *module name,* and **endmodule** are optional and can be mixed and matched as per design needs. Verilog allows multiple modules to be defined in a single file. The modules can be defined in any order in the file.

03 marks

b) Explain the following data types with an example in erilog:
(i) Nets (ii) Register (iii) Integers (iv) Real (v) Time Register. 10 Marks

(i) Nets: Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to. Nets are declared primarily with the keyword wire.
Example:
wire a; //Declare net a for the above circuit
wire b,c; //Declare two wires b,c for the above circuit
wire d = l’b0; //Net d is fixed to logic value 0 at declaration.

02 marks

(ii) Register: *Registers* represent data storage elements. Registers retain value until another value is placed onto them. Unlike a net, a register does not need a driver. Verilog registers do not need a clock as hardware
registers do. Values of registers can be changed anytime in a simulation by assigning a new value to the register. Register data types are commonly declared by the keyword reg. The default value for a reg data type is ‘x’.
Example:

```
reg reset; //declare a variable reset that can hold its value
initial //this construct will be discussed later
begin
reset = l’b1; //initialize reset to 1 to reset the digital circuit.
#100 reset = l’b0; //after 100 time units reset is deasserted.
End
```

02 marks

(iii) Integers:
An integer is a general purpose register data type used for manipulating quantities. Integers are declared by the keyword integer. 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.
Example:

```
integer counter; //general purpose variable used as a counter.
Initial
counter = -1;
```

02 marks

(iv) 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)
Example:

```
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)
```

02 marks

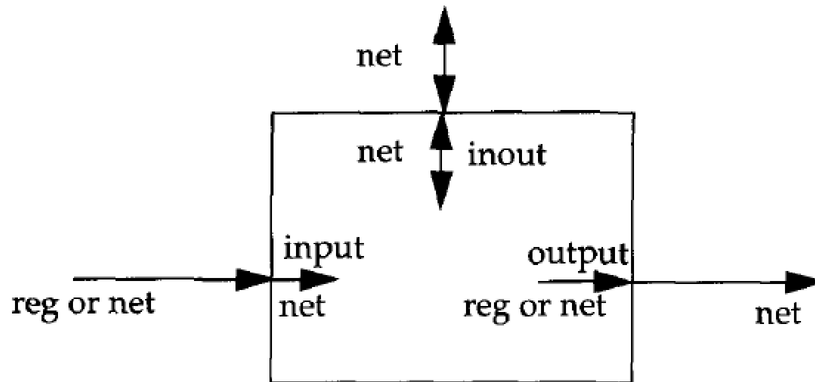
(v) Time Register:
Verilog simulation is done with respect to *simulation time.* A special time register data type is used in Verilog to store simulation time. A time variable is declared with the keyword time. The width for time register data types is implementation specific but is at least 64 bits.

```
time save_sim_time; //Define a time variable save_sim_time
initial
save_sim_time = $time; // Save the current simulation time
```

02 marks

4 a) Explain the port connection rules. 06 Marks

A port consisting of two units, one unit that is *internal* to the module another that is *external* to the module. The internal and external units are connected.



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

02 marks

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.

02 marks

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

02 marks

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.

Verilog allows ports to remain unconnected.

b) Explain the two methods of connecting ports to external signals with an example. 10 Marks

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.

Example:

```
module Top;
//Declare connection variables
reg [3:0]A,B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
fulladd4 fa_ordered (SUM, C_OUT, A, B, C_IN);
….
<stimulus>
…
endmodule

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

```
output c_cout;
input [3:0] a, b;
input c_in;
<module internals>
endmodule
```

The external signals *SUM, C_OUT, A,* B, and *C_IN* appear in exactly the same order as the ports *sum, c_out, a, b,* and *cin* in module definition of *fulladd4.*

05 marks

Connecting ports by name:

Verilog provides the capability to connect external signals to ports by the port names, rather than by position. The port connections can be specified in any order as long as the port name in the module definition correctly matches the external signal.

Example:

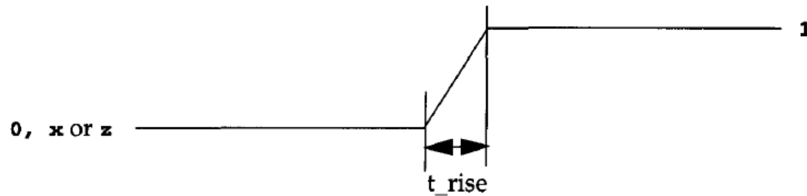
```
module Top;
//Declare connection variables
reg [3:0]A,B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
fulladd4 fa_byname(.c_out(C_OUT), . sum (SUM) , .b(B), .c_in(C_IN), .a(A),) ;
// Ports are connected by name.
<stimulus>
…
endmodule

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

The external signals *SUM, C_OUT, A,* B, and *C_IN* appear the different order and both internal and external signals are present in the connection list.

05 marks

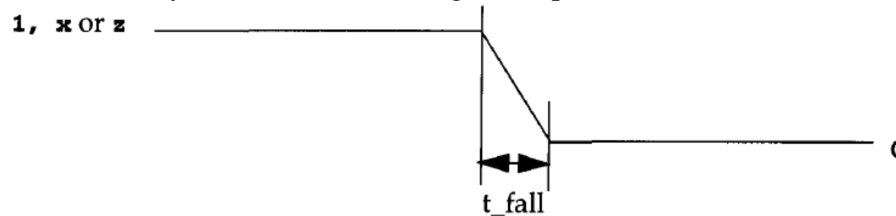
5 a) What are Rise, Fall and turn-off delays? How they are specified in Verilog? 06 Marks

Rise delay:
The rise delay is associated with a gate output transition to a 1 from another value.



01 marks

Fall delay
The fall delay is associated with a gate output transition to a 0 from another value.



01 marks

Turn-off delay
The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value. If the value changes to x, the minimum of the three delays is considered.

01 marks

Three types of delay specifications are allowed.
If only *one* delay is specified, this value is used for all transitions.

```
// Delay of delay_time for all transitions
and #(delay_time) a1(out, i1, i2);
```

and #(5) al(out, il, i2);

If *two* delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays.

```
// Rise and Fall Delay Specification.
and #(rise_val, fall_val) a2(out, i1, i2);
```

and #(4,6) a2(out, il, i2);
If all *three* delays are specified, they refer to rise, fall, and turn-off delay values. If no delays are specified, the default value is zero.

```
// Rise, Fall, and Turn-off Delay Specification
bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Bufif0 # (3,4,5) bl (out, in, control) ;

03 marks

b) Design a 2-to-l multiplexer using bufif0 and bufif1 gates. The delay specification for these gates are as follows:

	Min	Typ	Max
Rise	1	2	3
Fall	3	4	5
Turnoff	5	6	7

10 Marks

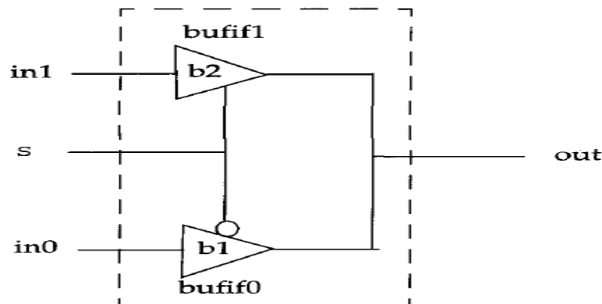
Write gate-level description and stimulus in verilog.

Truth table:

S	Out
0	in0
1	in1

02 marks

Logic diagram:



02 marks

Design:

```
module bufnot(
input in1, in0, s,
output out
);
bufif1 #(1:2:3,3:4:5,5:6:7) b2(out,in1,s);
bufif0 #(1:2:3,3:4:5,5:6:7) b1(out,in0,s);
endmodule
```

03 marks

Stimulus:

```
module bufnot_tb;
reg in1, in0, s;
wire out;
bufnot uut (.in1(in1), .in0(in0), .s(s), .out(out));
initial begin
in1 = 0; in0 = 0; s = 0;#100;
in1 = 1; in0 = 0; s = 0;#100;
in1 = 0; in0 = 1; s = 0;#100;
in1 = 1; in0 = 1; s = 0;#100;
in1 = 0; in0 = 0; s = 1;#100;
in1 = 1; in0 = 0; s = 1;#100;
in1 = 0; in0 = 1; s = 1;#100;
in1 = 1; in0 = 1; s = 1;#100;
end
endmodule
```

03 marks

6 a) Write a Verilog dataflow level of abstraction for 4-to-l multiplexer using conditional operator. 06 Marks

Design:

```
module mux_41( s1, s0, i0, i1, i2, i3, y);
input  s1, s0, i0, i1, i2, i3;
output y;
assign y = s1 ? (s0 ? i3: i2) : (s0 ? i1: i0);
endmodule
```

06 marks

b) Write a Verilog dataflow description for 4-bit Full adder with carry lookahead. 10 Marks

```
module fulladd4(sum, c_out, a, b, c_in);
output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;
wire p0,p1, p2,p3,g0,g1, g2,g3;
wire c4, c3, c2, c1;
```

02 marks

```
assign p0 = a[0] ^ b[0],
p1 = a[l] ^ b[1],
p2 = a[2] ^ b[2],
p3 = a[3] ^ b[3];
```

02 marks

```
assign g0 = a[0] & b[0],
g1 = a[l] & b[1],
g2 = a[2] & b[2],
g3 = a[3] & b[3];
```

02 marks

```
assign c1= g0 | (p0 & c_in),
c2= g1 | (p1 & g0) | (p1 & p0 & c_in)),
c3= g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in)),
c4= g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) | (p2 & p1 & p0 & c_in));
```

02 marks

```
assign s[0]=a[0] ^ b[0] ^ c_in;
assign s[1]=a[1] ^ b[1] ^ c1;
assign s[2]=a[2] ^ b[2] ^ c2;
assign s[3]=a[3] ^ b[3] ^ c3;
endmodule
```

02 marks

7 a) Explain the blocking assignment statements and non-blocking assignment statement with relevant examples. 08 Marks

Blocking assignment statements:
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.

Example:
```
reg x, y, z ;
reg r[15:0] reg_a, reg_b;
integer count;
initial
begin
x = 0 ; y = l ; z = 1; //Scalar assignments
count = 0; //Assignment to integer variables
reg_a = 16'b0; reg_b = reg_a; //initialize vectors
#l5 reg_a[2] = l'bl; //Bit select assignment with delay
#l0 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
```

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 count = count + 1 is executed last.
The simulation times at which the statements are executed are as follows:
1. All statements ***X*** = *0* through reg_b = reg_a are executed at time 0
2. Statement reg_a[2] = *0* at time = 15
3. Statement reg_b[15:13] = **{X,** y, *z}* at time = 25
4. Statement count = count + 1 at time = 25
5. Since there is a delay of 15 and 10 in the preceding statements, count = count+ 1 will be executed at time = 25 units

04 marks

Nonblocking Assignments:
Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block. A <= operator is used to specify nonblocking assignments.

Example:

```
reg x, y, z ;
reg r[15:0] reg_a, reg_b;
integer count;
initial
begin
x = 0 ; y = l ; z = 1; //Scalar assignments
count = 0; //Assignment to integer variables
reg_a = 16'b0; reg_b = reg_a; //initialize vectors
#l5 reg_a[2] <= l'bl; //Bit select assignment with delay
```

```
#l0 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 this example the statements *x* = 0 through *reg_b* = *reg_a* are executed sequentially at time O. 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)

04 marks

b) Write a note on the following loop statements:
(i) While loop (ii) forever loop. 08 Marks

(i) While loop:
The while loop executes until the while-expression becomes false. If the loop is entered when the while-expression is false, the loop is not executed at all. Any logical expression can be specified with these operators. If multiple statements are to be executed in the loop, they must be grouped typically using keywords begin and end.
Example:

```
integer count;
initial
begin
count = 0;
while (count < 128) //Execute loop till count is 127.
//exit at count 128
begin
end
$display("Count = %d", count);
count = count + 1;
end
```

04 marks

(ii) forever loop:
The keyword forever is used to express this loop. The loop does not contain any expression and executes forever until the $finish task is encountered. The loop is equivalent to a while loop with an expression that always evaluates to true,
e.g., while (1).
A forever loop is typically used in conjunction with timing control constructs.
Example:

```
reg clock;
initial
begin
clock = l'b0;
forever #10 clock =¬clock; //Clock with period of 20 units
end
```

04 marks

8 a) Explain the sequential and parallel blocks with examples. 08 Marks

Sequential blocks

The keywords **begin** and **end** are used to group statements into sequential blocks. Sequential blocks have the following characteristics:

1. The statements in a sequential block are processed in the order they are specified. A statement is executed only after its preceding statement completes execution (except for nonblocking assignments with intra-assignment timing control).
2. If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.

Example1:

```
reg X, Y;
reg [1:0] z, w;
initial
begin
X = l'bO;
y = l'bl;
z = {x, y};
w = {y , x};
end
```

The final values are **X** = 0, y= 1, z = 1, **W** = 2 at simulation time 0.

Example2:

```
reg X, y;
reg [1:0] z, w;
initial
begin
X = l'bO;
#5 y = l'bl;
#10  z = {x, y};
#20 w = {y , x};
end
```

The final values are the same except that the simulation time is 35 at the end of the block.

04 marks

Parallel blocks

Parallel blocks, specified by keywords fork and join, provide interesting simulation features. Parallel blocks have the following characteristics.

1. Statements in a parallel block are executed concurrently.
2. Ordering of statements is controlled by the delay or event control assigned to each statement.
3. If delay or event control is specified, it is relative to the time the block was entered.

Example

```
reg x, y;
reg [1 : 0] z, w;
initial
fork
x = l'b0; //completes at simulation time 0
#5 y = l'b1; //completes at simulation time 5
#10 z {x, y}; //completes at simulation time 10
#20 w = {y, x};
```

join 04 marks

b) Write a Verilog program for 8-to-1 multiplexer using case statement. 08 marks

```
module mux8_1 ( sel, I, out);
input [2:0] sel;
input [7:0] I;
output out;
reg out;
```

03 marks

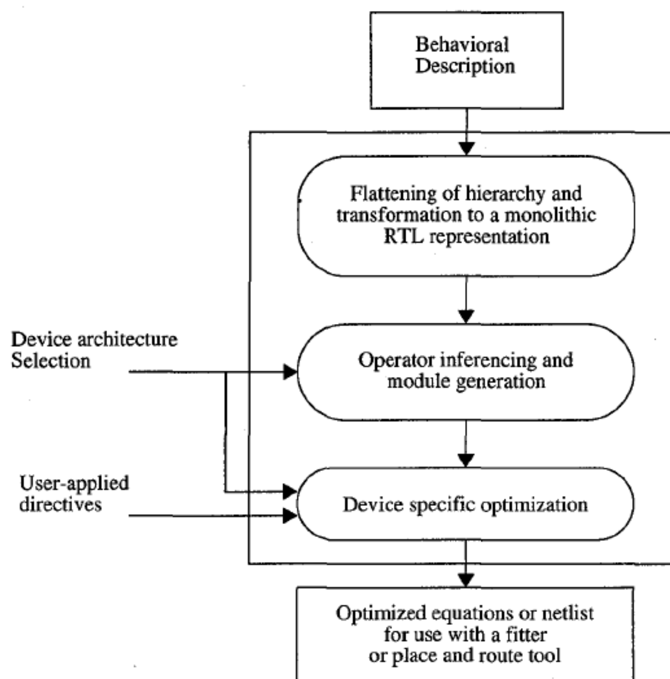
```
always @(sel, I)
case (sel)
3'b000: out = I[0];
3'b001: out = I[1];
3'b010: out = I[2];
3'b011: out = I[3];
3'b100: out = I[4];
3'b101: out = I[5];
3'b110: out = I[6];
3'b111: out = I[7];
default: out = 1'bx;
endcase
endmodule
```

05 marks

9 a) Explain the synthesis process with a block diagram. 08 Marks

Synthesis is the realization of design descriptions into circuits. In other words, synthesis is the process by which logic circuits are created from design descriptions. VHDL synthesis software tools convert VHDL descriptions to technology-specific netlists or sets of equations. Synthesis tools allow designers to design logic circuits by creating design descriptions without having to perform all of the Boolean algebra or create technology-specific, optimized netlists. Synthesis should be technology specific. Figure shown below illustrates the synthesis and optimization processes. The synthesis process then converts the design to internal data structures, allowing the "behavior" of a design to be translated to a register transfer level (RTL) description. RTL descriptions specify registers, signal inputs, signal outputs, and the combinational logic between them. At this point, the combinational logic is still represented by internal data structures. The synthesis process converts the design to internal data structures, allowing the "behavior" of a design to be translated to a register transfer level (RTL) description. RTL descriptions specify registers, signal inputs, signal outputs, and the combinational logic between them. Other RTL elements depend on the device-specific library. Some synthesis tools will search the data structures for identifiable operators and their operands, replacing these portions of logic with technology-specific, optimized components. Other portions of logic that are not identified are then converted to Boolean expressions that are not yet optimized.

05 marks



03 marks

b) Write a VHDL program for two 4-bit comparator using dataflow description. 08 Marks

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity magcomp is
```

```
    Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
           b : in  STD_LOGIC_VECTOR (3 downto 0);
           aeqb : out  STD_LOGIC;
           altb : out  STD_LOGIC;
           agtb : out  STD_LOGIC);
end magcomp;
```

03 marks

```
architecture dataflow of magcomp is

begin
aeqb <= '1' when (a = b) else '0';
agtb <= '1' when (a > b) else '0';
altb <= '1' when (a < b) else '0';

end dataflow;
```

05 marks

10 a) Explain the declaration of constant, variable and signal in VHDL with example. 08 Marks

Constants

A constant holds a specific value of a type that cannot be changed within the design description, and therefore is usually assigned upon declaration. Constants are generally used to improve the readability of code.

Example:

constant width: integer := 8;

Constants must be declared in a declarative region such as the package, entity, architecture, or process declarative region. A constant defined in a process declarative region is only visible to that process; one defined in an architecture is visible only to that architecture; one defined in an entity can be referenced by any architecture of that entity; one defined in a package can be referenced by any entity or architecture for which the package is used.

03 marks

Signals

Signals can represent wires, and they can therefore interconnect components. As wires, signals can be inputs or outputs of logic gates. Signals can also represent the state of memory elements.

Example:

signal count: bit_vector(3 downto 0);

Signals can be initialized as follows

signal count: bit_vector(3 downto 0) := "0101";

03 marks

Variables

Variables are used only in processes and subprograms and therefore they are declared in the declarative region of a process or subprogram. Variables should be initialized before being used. For the value of a variable to be used outside of a process, it must be assigned to a signal of the same type.

Example:

variable result: integer := ‘0’;

Variable assignments are immediate, not scheduled, as with signal assignments.

02 marks

b) Write a VHDL program for half adder in behavioral description. 08 Marks

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity halfadder is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          sum : out  STD_LOGIC;
          cout : out  STD_LOGIC);
end halfadder;
```

03 marks

```
architecture Behavioral of halfadder is

begin
process(a,b)
```

```
begin

if(a = '0' and b = '1') then
        sum <= '1';
        cout <= '0';
elsif(a = '1' and b = '0') then
        sum <= '1';
        cout <= '0';
elsif(a = '1' and b = '1') then
        sum <= '0';
        cout <= '1';
else
        sum <= '0';
        cout <= '0';
end if;
end process;
end Behavioral;
```

05 marks