* Introduction:

- There are two types of Hardware Description Languages (HDL):
  - Verilog HDL
  - VHDL: Very High Speed Integrated circuit HDL

- Verilog HDL

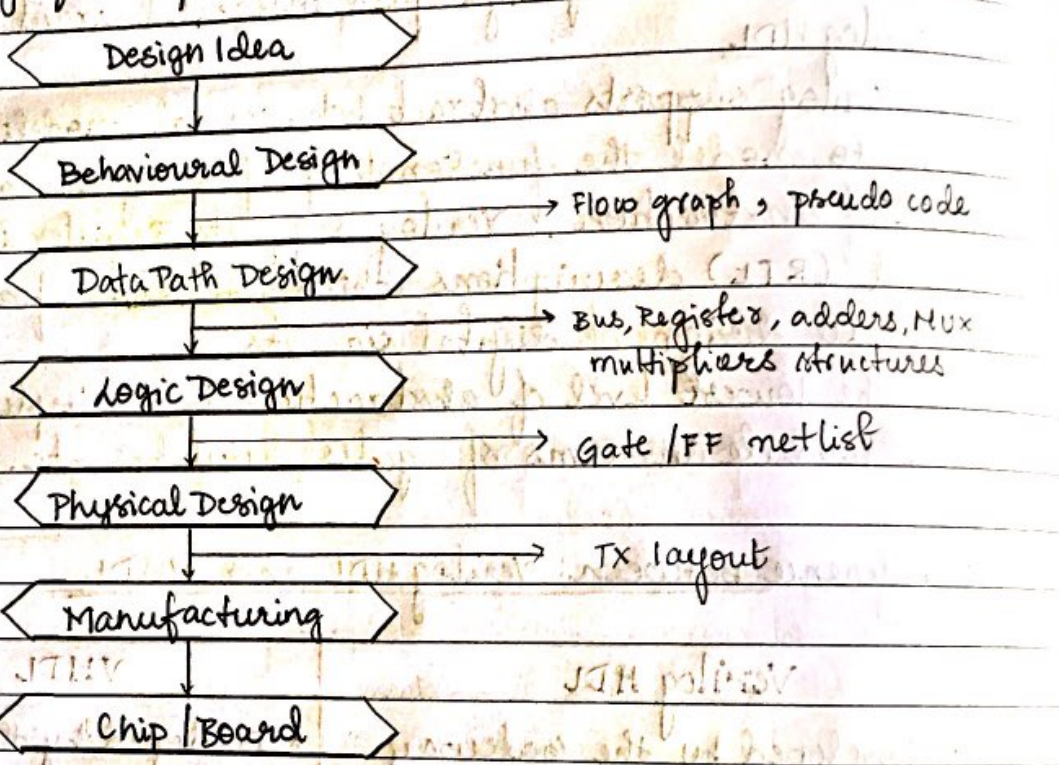
Verilog supports abstract behavioural modeling so can be used to model the functionality of a system at a high level of abstraction. Verilog supports Register Transfer Level (RTL) descriptions which are used for the detailed design of digital circuits.

The lower level of abstraction is the implementation of a module in terms of gates, switches etc.

- Difference between Verilog HDL and VHDL

Verilog HDL	VHDL
• Developed by the Gateway design Automation in year 1985. IEEE standard : 1995	• Developed by the US defense in the year 1981.
• C-like concise syntax.	• ADA like verbose syntax
• simpler code.	• Bulky code
• Design is composed of modules which have just one implementation.	• Design is composed of entities each of which can have multiple architectures.
• At switch and gate level abstraction.	• Suitable for behavioural level abstraction.
• case sensitive	• Not case sensitive
• Commercially developed.	• Government developed.

- Advantages of verilog HDL
  - Multiple level of abstraction
  - Easy to learn as it has c like syntax.
- Design flows of verilog HDL



Behavioural Design: specifies functionality of the block in terms of its behaviour. It can be expressed in terms of Boolean expression, truth table, FSM (timing diagram): sequen

Behavioural design → Data Path design: Synthesis.

Data Path Design: Netlist is the interconnection of blocks (structural design).

Logical Design:
optimizing the design by
- minimizing the number of gates
- minimizing the delay (gate levels)
- minimizing the signal transitions.

Physical Design: A physical layout is made which will be varified and then sent to fabrication.

Keerthana Ashok

UNIT - 01

Introduction to Verilog

* **Module concept:**

A module is the basic building block. It may be an element or a collection of lower level design blocks.

* **Syntax of Verilog HDL:**

```
Module Module_name (port list);
Port declaration;
Data type Port declaration;
Body of the program;
endModule
```

```
Module Module_name (port list).
Port declaration (if any)
Declaration of wires, registers and other variables
Instantiation of lower level modules
Dataflow statements (assign)
Always and initial blocks (all behavioural statements)
Tasks and functions
endModule
```

(Core circuit Body)

A module consists of two parts:

- core circuit : body of the program
- interface : port

Key words are case sensitive.

Ex: Module, always, initial, endModule, assign etc.

Ex: Module adder (a, b, cin, sum, cout)
input [3:0] a, b;
input cin;
output [3:0] sum;
output cout;
endModule

The body of the program has five components:
- variable declaration
- dataflow statements
- instantiation of lower modules
- behavioural blocks
- tasks and functions

* Lexical conventions:

1. Comments:

There are two ways to include comments:
- a single line comment
  Ex: // This is a single line comment
- a multiline coment: It cannot be nested but can include a single line comment
  Ex: 1. /* This a block comment which can include any ASCII character */
  2. /* This is /* invalid */ comment */
  3. /* This is // valid comment */

2. Number Specification:

<size>'<base format><number>
- size: optional
- base format: binary, octal, decimal, hexadecimal
- number: ↳ contains digits which are legal for the <base format>

- sized: Ex: 4'b1111 // 4 bit binary number
- unsized: Ex: d 255

3. Keywords: keywords are reserved identifiers. They are defined in lower case only.
   Ex: and, xor, nor, nand, module, endmodule, assign, initial, always, input, output, inout etc.

4. Constants:
   constant numbers can be specified as integer or a real constants only.
   - integer numbers: decimal, octal, binary, hexadecimal
     It can be represented in:
     - simple decimal form
     - base format notion.
   - real numbers:
     They can be specified in:
     - decimal notion Ex: 2.3
     - scientific notion Ex: 4.5e1, 35.6E-8

5. String: strings are sequences of characters enclosed by double quotes and contained on one line.
   Ex:
```
module example
begin
display ("Hello \n");
end
endmodule
```

6. Identifiers:
   Identifiers give objects unique names for reference and can consists of any sequence of letters, digits and the characters: $ and _ (underscore).
   The first character of an identifier can be a letter or underscore but not $ character or digit.
   They are case sensitive.

Date:

* Data types:

Data types in verilog are devided into NETS and Registers. these data types differ in the way that they are assigned and hold values. They also represent different hardware structures.

- 1. NEt Data type:

The net variables represent the physical connection between structural entities. These variables do not store values. It must be continuously driven.

Some of the net data types are:

wire, tri, wor, trior, wand, triand, tri0, tri1, supply0, supply1, trireg.

net_type [signed] [[MSB:LSB]] net1, net2 -- netn;

keyword — default range : 1 bit : scalar

specifying range > 1 bit : vector.

wire net datatype

Ex: wire a, b, c_in ; // a, b, c_in are 1 bit wires

wire [7:0] data_a; // data_a is 8 bit wire

wire [0:7] data_b; // data_b is 8 bit wire

wire signed [7:0] d; // d is a 8 bit signed wire.

- 2. Variable data type:

A variable is used as a data storage element.

Some of the variable data types are:

reg, integer, time, real, realtime.

Reg variable datatype

reg [signed] [[MSB:LSB]] reg1, reg2, --- regn;

* Primitives:

- Verilog has a number of builtin primitives that model gates and switches.
  - 12 gate primitives : for gate level modeling
  - 16 switch primitives : for switch level modeling

- **User level Primitives:**
  There are two types of user level primitives.
  a. combinational UDPs:
  b. sequential UDPs:

* **Attributes:**
An attribute specifies special properties of a verilog object or statement for use by specific software tools such as synthesis. An attribute can appear as a prefix to a declaration, module items, statements or port connections.

* **Modelling Styles:**

1. **Behavioural Modelling style:** (also known as algorithm program)
It is the highest level of abstraction which completely depends on the circuit behaviour.

Ex: AND gate

```
module and_1(c,a,b);
input a,b;
output c;
reg c;
always @(a,b)
begin
c = a & b;
end
endModule
```

2. **Data flow Modelling Style:**
It is the medium level abstraction which is achieved by defining the data flows of the module between register gates etc.

Ex: AND gate

```
module and_2(c,a,b);
input a,b;
```

output c;
assign c=a+b;
endModule

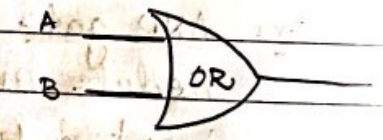
3. Structural Level Modelling Style:
It is the lowest level of abstraction obtained using logic gates.
Ex: AND gate.

```
module and_3(c,a,b);
input a,b;
output c;
and and_3(c,a,b);
endModule
```

Q: write a verilog code at all three levels for OR gate.

Sol: Behavioural level:

```
module or_1(c,a,b);
input a,b;
output c;
reg c;
always @(a,b)
begin
c = a|b;
end
endModule
```



A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Dataflow level

```
module or_2(c,a,b);
input a,b;
output c;
assign c = a|b;
endModule
```

Structural level

```
module or_3(c,a,b);
input a,b;
output c;
or or_3(c,a,b);
endModule
```

Page No.:
Date:
YOUVA

Q: Write a verilog code for all logic gates at all levels.

Sol:

		OR	AND	XOR	XNOR	NOR	NAND	NOT ($\bar{A}$)
a	b	c	d	e	f	g	h	i
0	0	0	0	0	1	1	1	1
0	1	1	0	1	0	0	1	1
1	0	1	0	1	0	0	1	0
1	1	1	1	0	1	0	0	0
		a|b	a&b	a^b	~(a^b)	~(a|b)	~(a&b)	~a

Data flow level

```
module logic_gates (c, d, e, f, g, h, i, a, b);
    input a;
    input b;
    output c;
    output d;
    output e;
    output f;
    output g;
    output h;
    output i;
    assign c = a|b;
    assign d = a&b;
    assign e = a^b;
    assign f = ~(a^b);
    assign g = ~(a|b);
    assign h = ~(a&b);
    assign i = ~a;
endmodule
```

Page No.:
Date:

Behavioural level

```
module logic_gates(c,d,e,f,g,h,i,a,b);
input a,b;
output c,d,e,f,g,h,i;
reg c,d,e,f,g,h,i;
always @(a,b)
begin
  c = a|b;
  d = a&b;
  e = a^b;
  f = ~(a^b);
  g = ~(a|b);
  h = ~(a&b);
  i = ~a;
end
endModule.
```

Structural level

```
module logic_gates(c,d,e,f,g,h,i,a,b);
input a,b;
output c,d,e,f,g,h,i;
or logic_gates (c,a,b);
and logic_gates (d,a,b);
xor logic_gates (e,a,b);
xnor logic_gates (f,a,b);
nor logic_gates (g,a,b);
nand logic_gates (h,a,b);
not logic_gates (i,a);
endModule
```

Page No.:
Date:
youva

Q: Write a verilog code for Half adder.

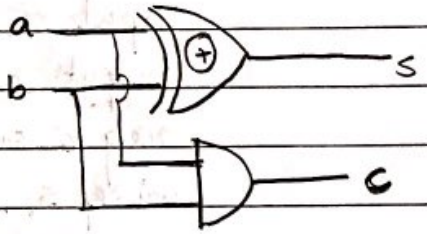
Sol:

inputs		outputs	
a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Data flow level :

```
module half_adder (s,c,a,b);
input a,b;
output s,c;
assign s = a^b;
assign c = a&b;
endModule
```

Behavioural level :




```
module half_adder (s,c,a,b);
input a,b;
output s,c;
reg s,c;
always @(a,b)
begin
  s = a^b;
  c = a&b;
end
end Module
```

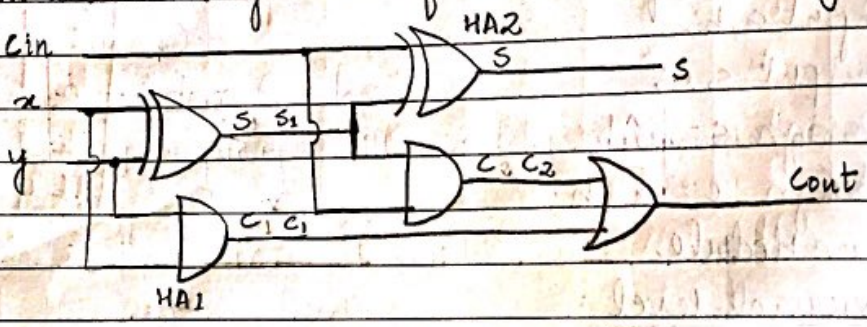
Structural level :

```
module half_adder (s,c,a,b);
input a,b;
output s,c;
xor half_adder (s,a,b);
and half_adder (c,a,b);
end Module
```

4. Mixed Style:

Ex: Full Adder

Full adder using two half adders and an OR gate.

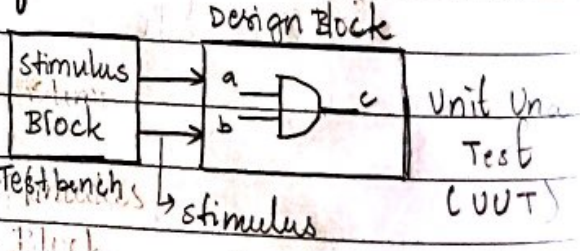



```
module full_adder_mixed_style(x,y,c_in,s,c_out);
input x,y,c_in;
output s,c_out;
reg c_out;
wire s1,c1,c2;
xor full_adder_mixed_style1(s,x,y);
and full_adder_mixed_style2(c_in,x,y);
assign s = c_in ^ s1;
assign c2 = c_in & s1;
always @ (c1,c2)
c_out = c1 | c2;
endmodule
```

* Basics of Simulation:

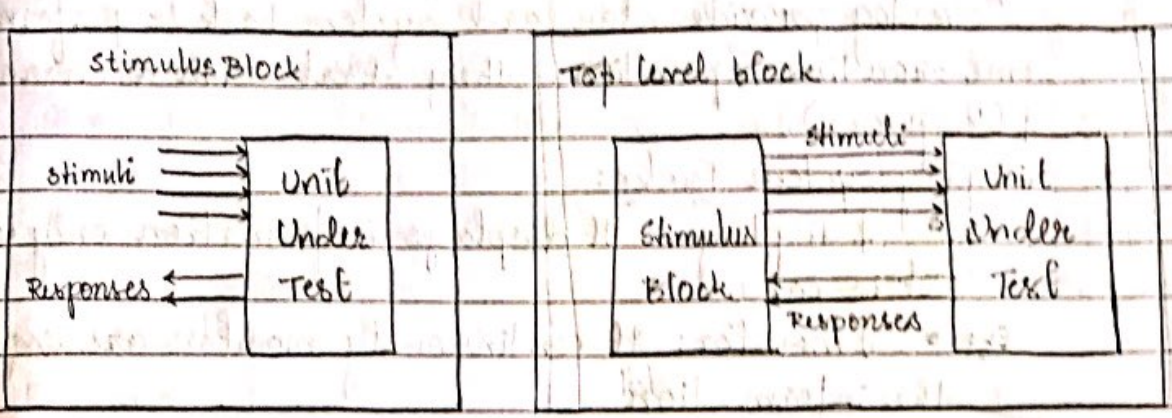
The input applied to a design block for testing is called as stimulus which is generated and controlled by the stimulus block.

The design block to be tested is called as Unit under Test (UUT)



Test Bench: It is used for analyse the system by simulation. It contains both the UUT as well as stimuli for the simulation.

The two basic simulation constructs are:-



To check the functionality of the main design we require certain value of input signal. To model these input value we will use the stimulus block. This is same as test bench.

* compiler directive:
  - Time Scale compiler directive:

To define the simulation time unit we need a reference time unit. 'timescale is used for specifying the reference time unit for the simulator.

'timescale ref_time_unit / time_precision

Ex: 'timescale 10ns/1ns

```
# 3.55 a=b+1; // corresponds to 36ns
# 3.54 b=c+1; // corresponds to 35ns
```

delays are rounded off to real numbers.

'timescale 1ns/10ns

```
# 3.55 a=b+1; // corresponds to 3.6 ns
# 3.54 b=c+1; // corresponds to 3.5 ns
```

Reference time unit: all the signals to pass the data at the given time are multiplied by the time unit.

Precision: It is the minimum value up to which the precision can be measured.

Page No.:
Date:

* System Tasks:
Verilog provide standard system task to perform some routine operation. They start with a character $(keyword).

1. Display System Tasks:
   - $display : It displays information only when it is called.
   - $monitor : It continuously monitors or displays the information
     - $monitoron
     - $monitoroff

     To enable or disable the $monitor system task

Syntax: task_name [(arguments)];

Ex: $display ($time, "ns %d %d %h %h", x, y, cin, xcout, sumy);
$monitor ($realtime, "ns %d %d %h %h", x, y, cin, xcout, sumy);

2. Simulation time system function:
   - $time : rounds off to integer values
   - $realtime : rounds off to real values.

Ex:
```
`timescale 10ns/1ns
module time_usage;
reg a;
initial begin $realtime
  $monitor ($time, " a = %b", a);
  #3.55 a = 0;
  #3.55 a = 1;
end
endmodule
```

NOTE: # indicates delay

For $time
#0 a = x
#4 a = 0
#8 a = 1

For $realtime
#0 a = x
#3.6 a = 0
#7.2 a = 1

3. Simulation control system Task:

- $stop : It will stop the simulation at the given simulation time and provide control to user.
- $finish : It will terminate the simulation at the given simulation time.

Ex: #200 $stop // stop the simulation at 200 time units
#200 $finish // terminates the simulation at 200 time units

* Test bench:

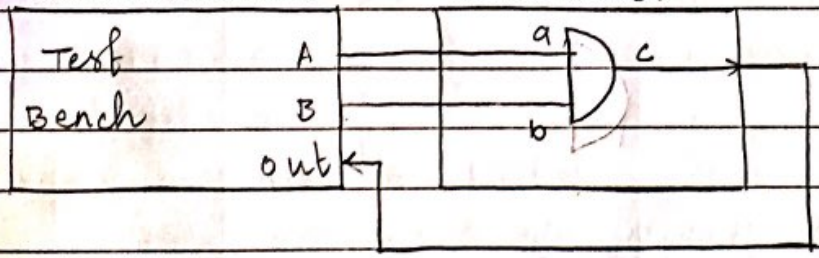
AND gate: dataflow level

```
module and_2in(c,a,b);
input a,b;
output c;
assign c= a&b;
endmodule
```

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

a 0 0 1 1
b 0 1 0 1
c 0 0 0 1

Test bench file: and_2in_test.v
Design file: and_2in.v




```
include "and_2in.v"
`timescale 1ns/100ps
module and_2in_test;
reg A,B;
wire out;
and_2in U1 // calling design module (instantiation)
(
  .a(A);
  .b(B);
  .c(out);
);
```

transfer

Page No.:
Date:

```
initial begin
A=0 ; B=0;
#20 A=0; B=1;
#20 A=1; B=0;
#20 A=1; B=1;
#40
$stop
$finish
end
endmodule
```

UNIT - 02

Structural Modelling

In structural style, a module is designed as a set of interconnected components.

NOTE:

- There are 12 gate primitives and they are synthesizable.
- There are 16 switch primitives and are not synthesizable.

The components can be:

- modules
- gate primitives
- switch primitives
- user defined primitives.

* Gate Level Modelling:

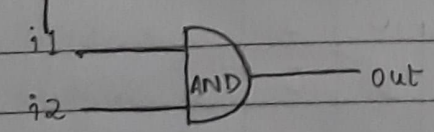
Gate level modelling describes a design which only uses gate primitives.

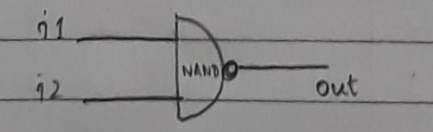
- GATE PRIMITIVES:

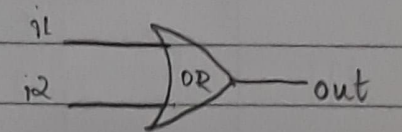
There are 12 predefined gate primitives. They are of two types:

1. and/or gates: (and, nand, or, nor, xor and xnor)

They have one scalar output and multiple scalar inputs.



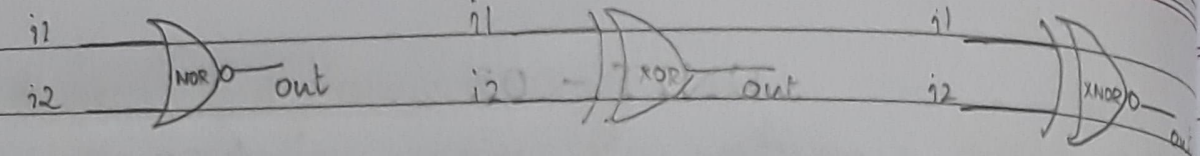




AND		i2			
		0	1	x	z
i1	0	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

NAND		i2			
		0	1	x	z
i1	0	1	1	1	1
	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x

OR		i2			
		0	1	x	z
i1	0	0	1	x	x
	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x



NOR	i2 = 0	1	x	z
i1 = 0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

XOR	i2 = 0	1	x	z
i1 = 0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

XNOR	i2 = 0	1	x	z
i1 = 0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

syntax

gate_name [instance_name] (output, input1, input2, ...);

where gate_name is of and, nand, or, nor, xor, xnor.

For multiple instances of same gate

gate_name

[instance_name1] (output, input1, input2, ..., inputn),

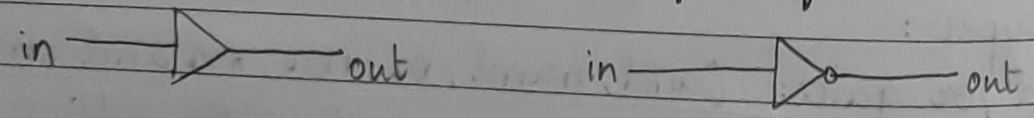
[instance_name2] (output, input1, input2, ..., inputn),

.....

[instance_namen] (output, input1, input2, ..., inputn);

2. buf/not gates: (buf, not, bufif0, bufif1, notif0 and notif1)

They have one scalar input and one or multiple scalar outputs. They are used to realise the NOT operation or to buffer the output of an AND/OR gate.



in	out
0	0
1	1
x	x
z	x

buffer

in	out
0	1
1	0
x	x
z	x

not

syntax

buf_not [instance_name] (output1, output2, ..., outputn, input

Page No.: YOUVA
Date:

For multiple instances of same gate

buf_not

[instance_name1] (output1, output2, ..., outputn, input);
[instance_name2] (output2, output2, ..., outputn, input);

[instance_namen] (output1, output2, ..., outputn, input);

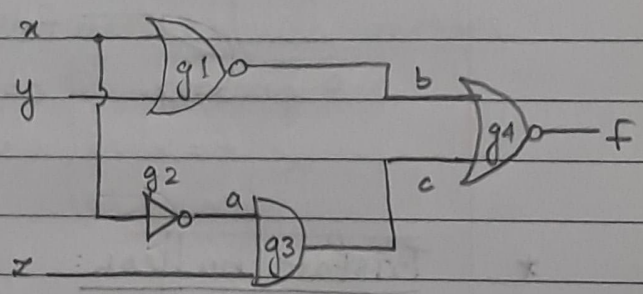
NOTE

An array of and/or gates.

gate_name [instance_name [range]] (output, input1, ..., inputn);

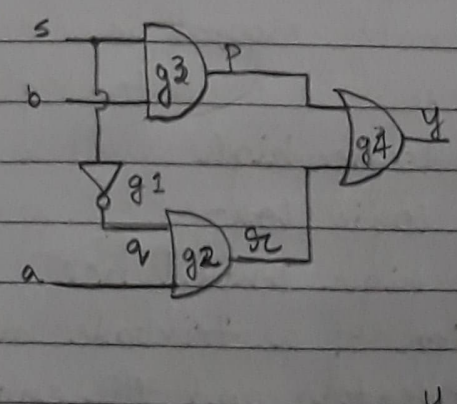
Q: Write a program for the following simple application of basic gates.

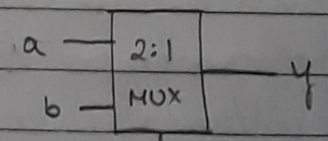
— module basic_gates (x, y, z, f);
input x, y, z;
output f;
wire a, b, c;
nor g1 (b, x, y);
not g2 (a, x);
and g3 (c, a, z);
nor g4 (f, b, c);
endmodule



Q: Write a code for 2:1 MUX.

— module mux(s, b, a, y)
input s, a, b;
output y;
wire p, q, r;
not g1 (q, s);
and g2 (r, q, a);
and g3 (p, s, b);
or g4 (y, p, r);
endmodule

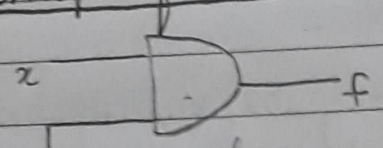




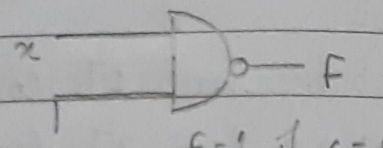
s	y
0	a
1	b

$y = \bar{s}a + sb$

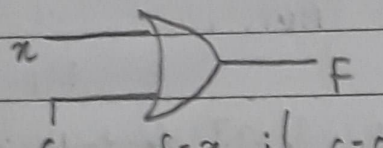
- **concept of controlled gates:**



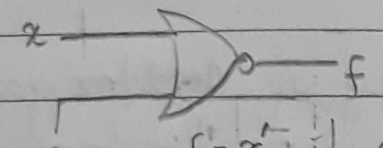
f=0 if c=0
f=x if c=1



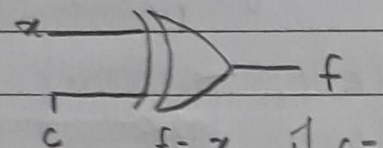
f=1 if c=0
f=x' if c=1



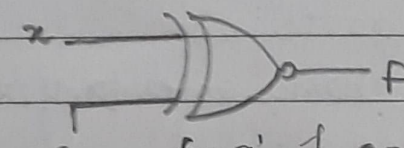
f=x if c=0
f=1 if c=1



f=x' if c=0
f=0 if c=1



f=x if c=0
f=x' if c=1



f=x' if c=0
f=x if c=1

* **Tristate buffers:**

When an output of a logic circuit can also be set to an extra state, high impedance along with two normal states 0 and 1, the logic circuit is said to be a tristate logic circuit.

The tristate output is controlled by an enable input. When the enable signal is activated, the logic circuit operates in its two normal output states 0 and 1, but when the enable signal is deactivated, the output of the logic circuit is set to high-impedence.

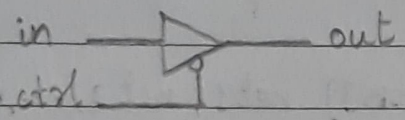
1: logic high
0: logic low
z: high impedence.

The function of a tristate may be associated with buffers or inverters and the enable signal can be activated at low-logic level or high logic level.

There are four kinds of tristate buffers or inverters:

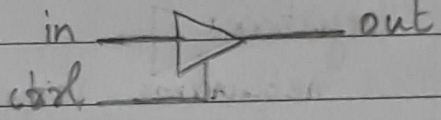
i. active-low buffer

bufif0



ii. active-high buffer

bufif1

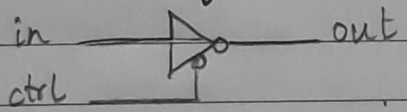


bufif0		ctrl			
		0	1	x	z
in	0	0	z	L	L → 0 or z
	1	1	z	H	H → 1 or z
	x	x	z	x	x
	z	x	z	x	x

bufif1		ctrl			
		0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

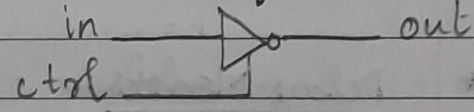
iii. active-low inverter

notif0



iv. active-high inverter

notif1



notif0		ctrl			
		0	1	x	z
in	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

notif1		ctrl			
		0	1	x	z
in	0	z	1	H	H
	1	x	0	L	L
	x	z	x	x	x
	z	z	x	x	x

To instantiate a buffer from this group, the following syntax is used:

```
buf_name [instance_name] (output, input, control);
```

To instantiate multiple instances of the same tristate buffer, the following syntax is used:

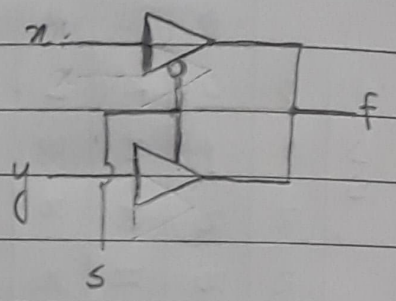
```
buf_name
    [instance_name1]  (output, input, control),
    ------
    [instance_namen] (output, input, control);
```

- wire and tri nets : connect elements
  a wire net - driven by a single driver
  a tri net - driven by multiple drivers.

syntax:
net-name [signed] [[msB:LSB]] net1, net2, ---, netn;

Q: A 2:1 MUX from tristate buffers:

```
module mux (x, y, s, f);
input x, y, s;
output f;
tri f;
bufif0 b1 (f, x, s);
bufif1 b2 (f, y, s);
endmodule
```



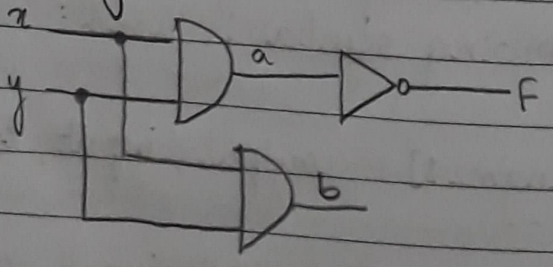
* Delay Models:

Due to existance of R and C in logic circuits, all of the actual logic gates have finite propagation delays. In addition, in every interconnection between two circuits there always exists an RC delay called the transport delay.

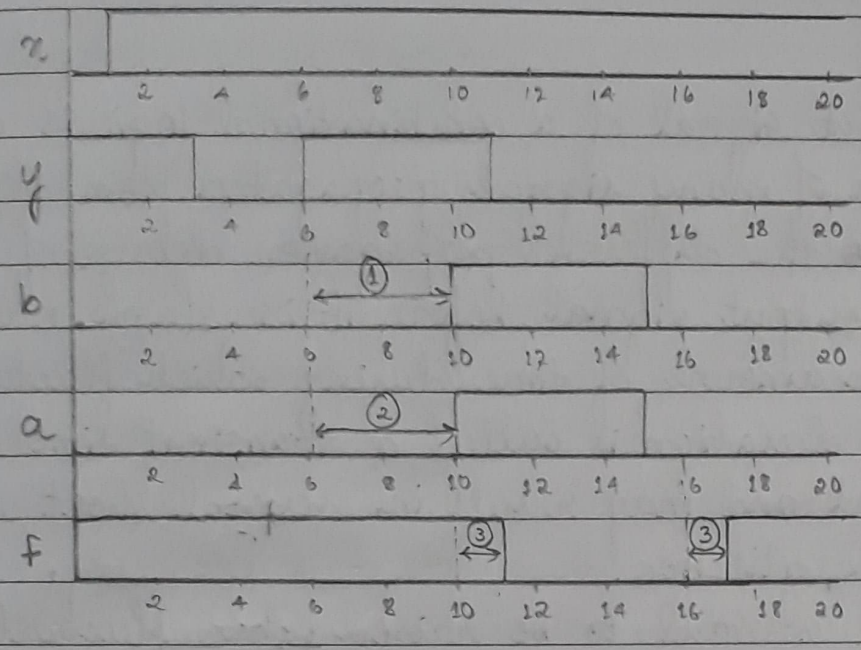
- Intertial Delay Model

It takes certain amount of time and energy for the output of a gate to respond to change in the input. This implies that the signals that do not persist long enough, will be filtered out and not propagated to the output of the gates.

Ex:



Assume propagation delays of both AND gates are 4 time units and of inverter is 1



↔ inertia delay
① inertial delay of 2nd AND gate
② inertial delay of 1st AND gate
③ inertial delay of NOT gate

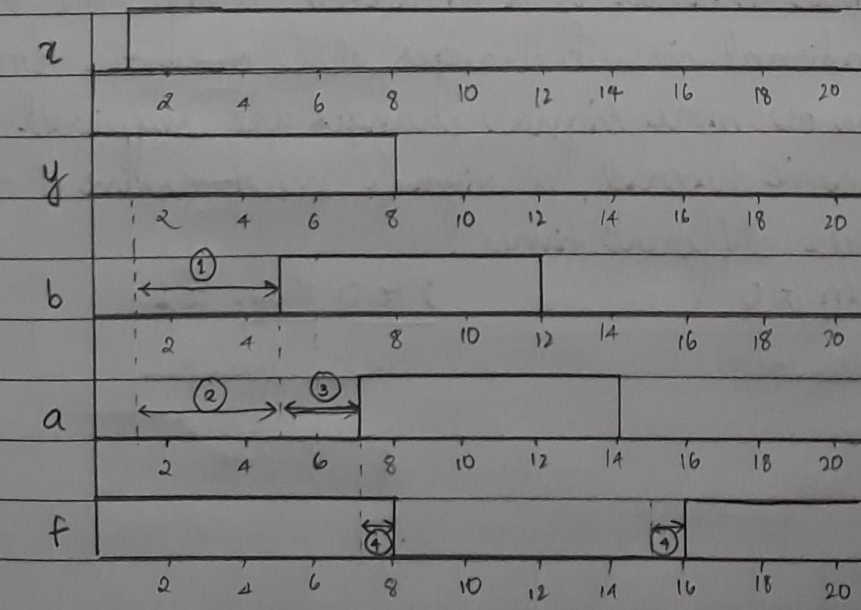
NOTE: 0 to 3 input y is filtered out as it is less than the propagation delay of the AND gate.

```
wire a, b;
and #4 (b, x, y); // inertial delay
and #4 (a, x, y);
not #1 (f, a);
```

- Transport Delay Model

The transport delay model is usually used to model net (wire) delays, i.e., the time of flight of a signal passing through a wire.

Ex: Above example with wire a having 2 time units of transport delay.



↔ inertial delay
⟺ transport delay
① inertial delay due to 2nd AND gate
② inertial delay due to 1st AND gate
③ transport delay due to wire a
④ inertial delay due to inverter.

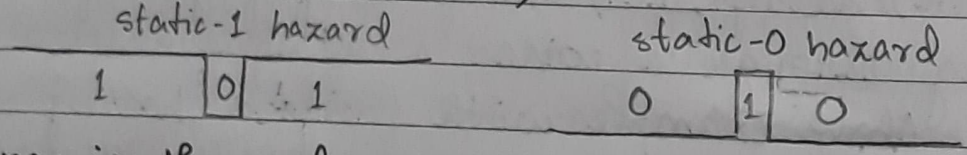
* Hazards:

The output signal of a combinational logic is a combination of many signals propagated from different paths. Due to the different propagation delays of these paths, the output signal before it is stable must experience an amount of time during which fluctuations occur. This duration is called a transient time of the output signal and may result in several short undesired pulses called glitches.

A hazard is said to be raised when fluctuation occur during the transient time. There are two types of hazards:

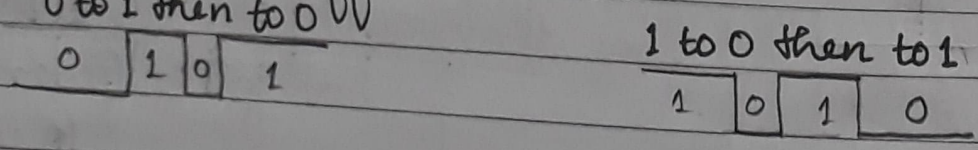
1. **Static Hazards:**

A static hazard is a situation where the output produces a '0' glitch when its stable value is 1 (static-1 hazard) and a '1' glitch when its stable value is 0 (static-0 hazard)



2. **Dynamic Hazards:**

A dynamic hazard is a situation where the output of a combinational circuit changes three or more times. Because three or more signal changes are required to have a dynamic hazard, a signal must arrive at the output at three different times.



★ SWITCH LEVEL MODELLING:

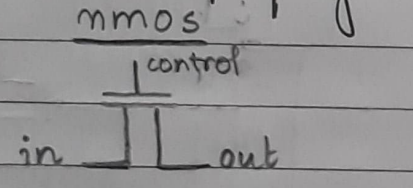
• MOS Switches:

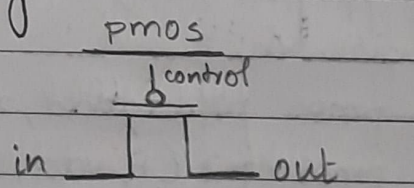
- There are two types of switches:

1. Ideal Switches: there will be zero resistance when the switch is closed. Ex: nmos, pmos, cmos.
2. Resistive Switches: there will be certain amount of resistance is present when switch is closed. Ex: rnmos, rpmos, rcmos.

- There are two MOS switches: nmos/rnmos and pmos/rpmos. These switches are usually used to model unidirectional switches through which data can be allowed to pass from input to output or be blocked by appropriately setting the control input.

The nmos and pmos switches pass signals from their inputs to their outputs without degradation whereas the rnmos and rpmos switches reduce the strength of the signals that propagate through them.





nmos		control			
		0	1	X	Z
i/p	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	Z	Z	Z

pmos		control			
		0	1	X	Z
i/p	0	0	Z	L	L
	1	1	Z	H	H
	X	X	Z	X	X
	Z	Z	Z	Z	Z

syntax:

switch_name [instance_name] (output, input, control);

- supply0 and supply1 nets:

The supply0 and supply1 nets are used to model ground and power nets respectively. The logic value of

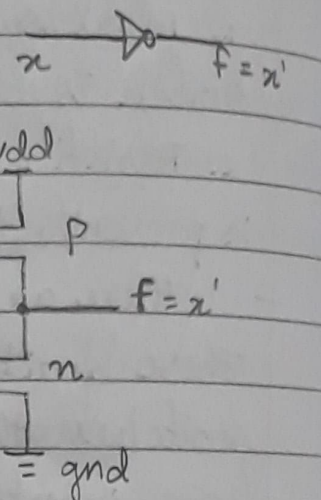
supply 0 is 0 and the logic value of supply 1 is 1.

syntax:

supply0/supply1 [[msb:lsb]] net1, net2, ---, netn;

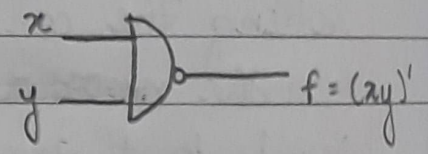
Q:

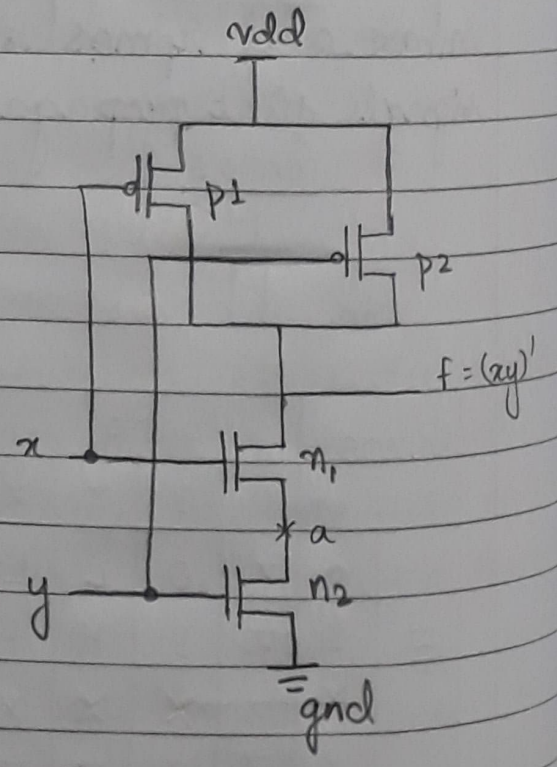
1. A CMOS inverter.




```
module not_1(input x, output f);
supply1 vdd;
supply0 gnd;
  pmos p1(f, vdd, x);
  nmos n1(f, gnd, x);
endmodule
```

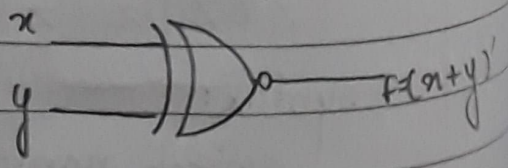
2. A two input CMOS NAND gate





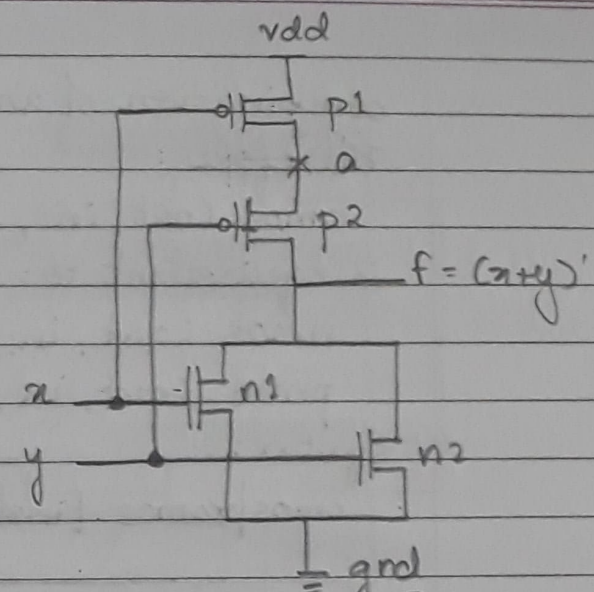

```
module nand_1(
  input x, y,
  output f);
supply1 vdd;
supply0 gnd;
wire a;
  pmos p1(f, vdd, x);
  pmos p2(f, vdd, y);
  nmos n1(f, a, x);
  nmos n2(f, gnd, y);
endmodule
```

3. A two-input CMOS NOR gate




```
module nor_1(
  input x, y,
  output f);
supply1 vdd;
supply0 gnd;
wire a;
```

```
pmos p1(a, vdd, x);
pmos p2(f, a, y);
nmos n1(f, gnd, x);
nmos n2(f, gnd, y);
endmodule
```



- pullup and pulldown sources:

The pullup and pulldown sources separately place logic values 1 and 0 on the nets connected in its terminal list. No delay specification can be applied to these sources.

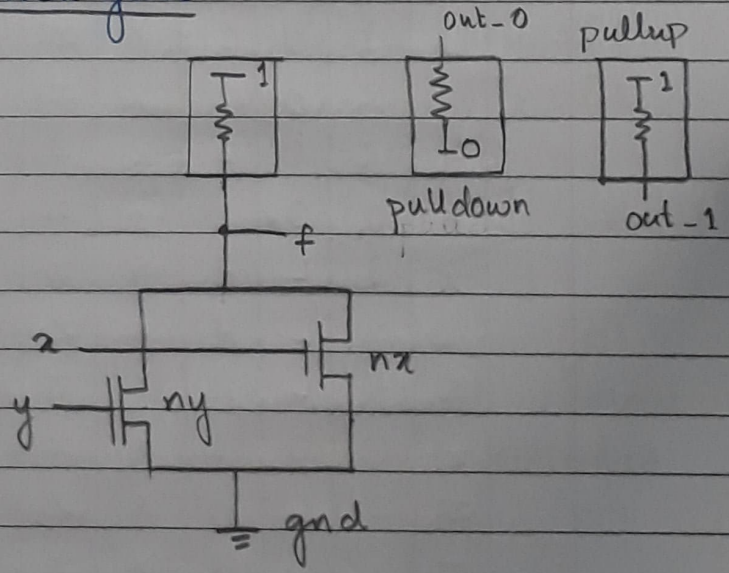
syntax:

pullup | pulldown [(strength)] [instance_name] (net_name);

Ex: pullup (strong1) p1 (net a), p2 (net b);

Q: A two-input pseudo-NMOS NOR gate.

```
- module pseudo_nor(
    input x, y,
    output f);
  supply0 gnd;
    nmos nx(f, gnd, x);
    nmos ny(f, gnd, y);
    pullup a(f);
  endmodule
```



* CMOS switches:

The cmos and nmos switches have a data input, a data output and two control inputs.

The cmos gate passes signals without reduction whereas the nmos gate reduces the strength of signals passing through it. The cmos switch is virtually a combination of a pmos switch and an nmos switch. The nmos switch is a

combination of an rpmos switch and an nmos switch
Therefore:

```
cmos (out, in, ncontrol, pcontrol);
```

is equivalent to:

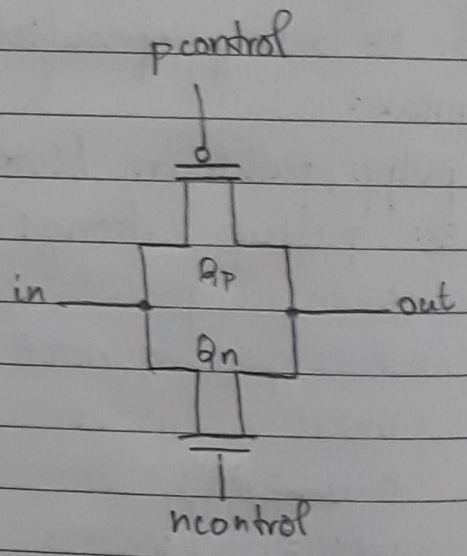
```
nmos (out, in, ncontrol);
pmos (out, in, pcontrol);
```

syntax:

```
cmos|rcmos [instance_name] (output, input, ncontrol, pcontrol);
```

control		data			
n	p	0	1	x	z
0	0	0	1	x	z
0	1	z	z	z	z
0	x	L	H	x	z
0	z	L	H	x	z
1	0	0	1	x	z
1	1	0	1	x	z
1	x	0	1	x	z
1	z	0	1	x	z
x	0	0	1	x	z
x	1	L	H	x	z
x	x	L	H	x	z
x	z	L	H	x	z
z	0	0	1	x	z
z	1	L	H	x	z
z	x	L	H	x	z
z	z	L	H	x	z

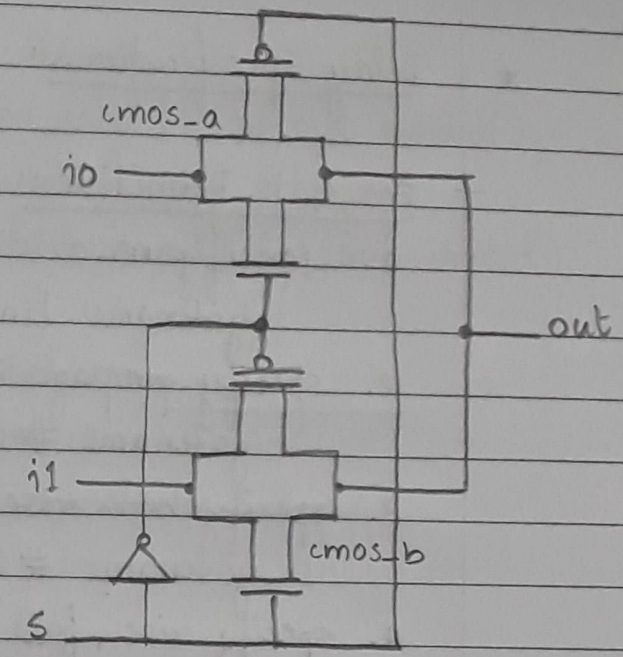
CMOS switch



Q: A 2:1 MUX

```
module mux (out, s, i0, i1);
output out;
input s, i0, i1;
wire sbar
```

```
not (sbar, s);
cmos (out, i0, sbar, s);
cmos (out, i1, s, sbar);
endmodule
```

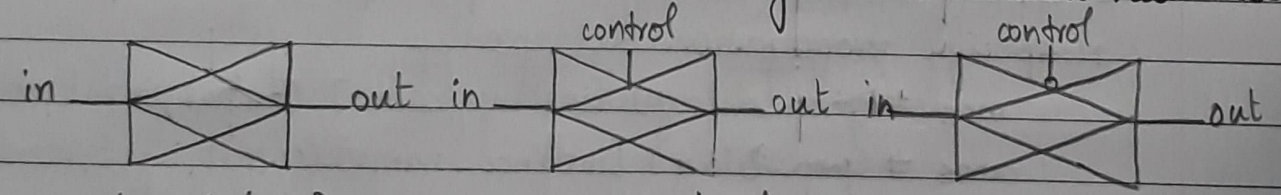


★ **Bidirectional switches:**

There are six bidirectional switches:
tran, tranif0, tranif1, rtran, rtranif0 and rtranif1.

These switches are usually used to model bidirectional switches through which data can be allowed to flow both ways by appropriately setting the control input.

tranif0 and rtranif0 : active-low bidirectional switches
tranif1 and rtranif1 : active-high bidirectional switches



tran (rtran) tranif1 (rtranif1) tranif0 (rtranif0)

The tran, tranif0 and tranif1 switches pass signals without reduction whereas the rtran, rtranif0 and rtranif1 switches reduce the signals passing through them.

syntax:

```
tran/rtran [instance_name] (in, out);
tranif0/rtranif0 [instance_name] (in, out, control);
tranif1/rtranif1 [instance_name] (in, out, control);
```

tran and rtran switches cannot be turned off but the other four switches can be turned off through appropriately setting the values of control inputs.

★ Delay Specifications:

– For Gate Primitives:

1. specify no delay
   gatename [instance_name] (output, in1, in2, ...);
2. specify propagation delay only
   gatename #(prop-delay) [instance_name] (output, in1, in2,
3. specify both rise and fall times:
   gatename #(t_rise, t_fall) [instance_name] (output, in1, in2,
4. specify rise, fall and turn-off times:
   gatename #(t_rise, t_fall, t_off) [instance_name] (output, in1,...

delay specifiers:

(minimum : typical : maximum)

Ex: and #(10:20:15) and_1 (output, input1, input2);

– For MOS and CMOS switches:

1. specify no delay
   mos [instance_name] (output, input, control);
   cmos [instance_name] (output, input, ncontrol, pcontrol);
2. specify propagation delay only
   mos #(prop_delay) [instance_name] (output, input, control);
   cmos #(prop-delay) [instance_name] (output, input, ncontrol, pcontrol);
3. specify both rise and fall times:
   mos #(t_rise, t_fall) [instance_name] (output, input, control);
   cmos #(t_rise, t_fall) [instance_name] (output, input, ncontrol, pcontrol);
4. specify rise, fall and turn-off times:
   mos #(t_rise, t_fall, t_off) [instance_name] (output, input, control);
   cmos #(t_rise, t_fall, t_off) [instance_name] (output, input, ncontrol, pcontrol);

– For bidirectional Switches:

1. Specify no delay
   bdsw_name [instance_name] (in, out, control);

2. Specify turn-on and turn-off delay
   bdsw-name #(t_on, t_off) [instance_name] (in, out, control);
3. Specify separately turn on and turn off delays:
   bdsw-name #(t_on, t_off) [instance_name] (in, out, control);

note:

t-rise: transition to the 1 value
t-fall: transition to the 0 value
t-off: transition to a high-impedance value.

* Signal Strength:

There are two kinds of signal strengths that can be specified to a scalar net:

1. Driving strengths: signals propagate from gate outputs and continuous assignment outputs. There are four driving strengths: supply, strong, pull, weak.
2. Charge storage strengths: Signals originate in the trireg net type. There are three charge storage strengths: large, medium and small.

signal strength represents the ability of the source device to supply energy to drive the signal.

- Strength Hierarchy:

supply → strong → pull → large → weak → medium → small → highz.

signal strength can be reduced due to issues in circuit. When reduced which level it switches to is given by the above strength hierarchy.

- Signal Contention:

When multiple drivers drive a net at the same time, a contention occurs on the net. There are rules to resolve the contention:

- Combined signals with the same value and unequal strengths
  If two signals with the same known values but different

strength drive the same net, the stronger signal dominates.
- combined signal with an opposite value and equal strength
If two signals with the opposite known values but equal strengths drive the same net, the result is an unknown value

★ trireg Net:

The trireg net stores a value and is used to model charge storage nodes. It can be in one of two states.

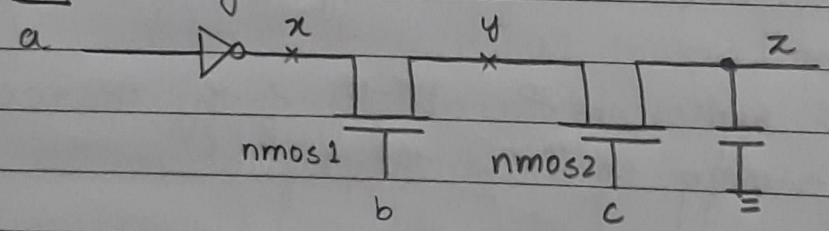
1. Driven state.

When inputs are 1, 0 or x then net takes the stregth of the driver. The strength can be supply, strong, pull or weak.

2. Capacitive State

When all drivers to a trireg net are at high impedance state the net retains its last driven state. The strength can be small, medium (default) or large.

— Ex: trireg net



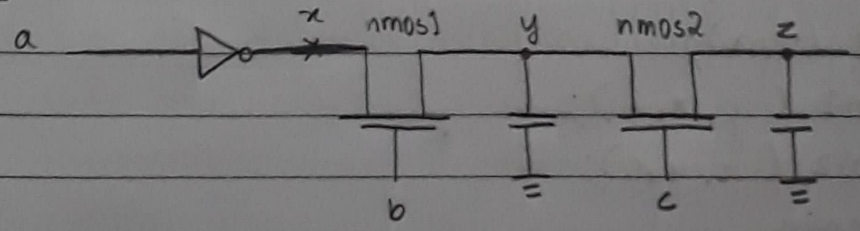
In simulation time 0
→ a, b, c are set to 1
↳ output x of inverter is 0
↳ then net y changes it value to 0.
↳ trireg net z enters driven state and discharges to strong0.
At simulation time 10
→ b is cleared
↳ net y changes to high impedance.
↳ net z enters the capacitive state and stores its last driven value 0 with medium strength

```
module trireg_ex;
reg a, b, c;
wire x, y;
trireg (medium) z;
not not1 (x, a);
nmos nmos1 (y, x, b);
nmos nmos2 (z, y, c);
initial begin
  $monitor ("%0d a=%v b=%v c=%v x=%v y=%v z=%v",
            $time a, b, c, x, y, z);
    // %v displays signal strength
  a = 1;
  b = 1;
  c = 1;
  #10 b = 0;
  #30 b = 1;
  #10 b = 0;
  #100 $finish;
end
endmodule
```

Result:

```
#0   a=st1  b=st1  c=st1  x=st0  y=st0  z=st0
#10  a=st1  b=st0  c=st1  x=st0  y=Hiz  z=Me0
#40  a=st1  b=st1  c=st1  x=st0  y=st0  z=st0
#50  a=st1  b=st0  c=st1  x=st0  y=Hiz  z=Me0
```

Ex:



Charge sharing problem

At simulation time 0
→ a=0 and b=c=1
↳ thus nodes x, y, z are driven to strong 1

At simulation time 10
→ a=b=0 and c=1
↳ y enters capacitive state and stores its last driven values 1 with large strength.
z still in driven state and is driven to value 1 with large strength

At simulation time 20
→ a=b=c=0
↳ net z enters capacitive state and stores a value 1 of small strength.

At simulation time 30
→ a=b=0 and c=1
↳ two trireg nets y and z are connected and share the same charge.

At simulation time 40
→ a=b=c=0
↳ net z enters into capacitive state and store a value 1 of small strength.

```
module trireg_charge_sharing;
reg a, b, c;
wire x;
trireg (large) y;
trireg (small) z;
not not1 (x, a);
nmos nmos1 (y, x, b);
nmos nmos2 (z, y, c);
initial begin
  $monitor ("%0d a=%v b=%v c=%v x=%v
    y=%v z=%v", $time, a, b, c, x, y, z);
```

Keerthana Ashok

```
a=0 ; b=1; c=1;
#10 b=0;
#10 c=0;
#10 c=1;
#10 c=0;
#100 $finish;
end
endmodule
```

Result:

#0	a=st0	b=st1	c=st1	x=st1	y=st1	z=st1
#10	a=st0	b=st0	c=st1	x=st1	y=La1	z=La1
#20	a=st0	b=st0	c=st0	x=st1	y=La1	z=Sm1
#30	a=st0	b=st0	c=st1	x=st1	y=La1	z=La1
#40	a=st0	b=st0	c=st0	x=st1	y=La1	z=Sm1

- trireg Net charge delay

#(t_rise, t_fall, t_decay)

The t_decay specifies the charge decay time of a trireg net, which is the time between when its drivers turn off and the point that its stored charge can no longer be determined.

The charge decay process begins when the drivers turn off and the trireg net starts to hold charge. It ends whenever either one of the following two conditions is satisfied:

- the charge decay time elapses and the net makes a transition to x
- the drivers turn on and propagate a 1, 0 or x into the net.

Date:

* **HIERARCHIAL STRUCTURAL MODELLING:**

The three closely related issues of hierarchial structural modeling are:

instantiations, generate statements, configurations.

* **Module:**

The basic units of verilog HDL are modules. Each module has two major parts:

- interface: way through the module communicates with outside world
- body: defines the functionality of the module.

A module is defined by using the keywords module and endmodule and can be one of the following two forms:

```
// port list style
    module module_name [#(parameter_declarations)]
     [port_list];
     parameter_declarations;
     port_declarations;
     other_declarations;
     statements;
    endmodule
// port list declaration style
    module module_name [#(parameter_declarations)]
    [port_declarations];
    parameter_declarations;
    other_declarations;
    statements;
    endmodule
```

- Port Declaration

1. The input Port Declaration

Used to declare a group of signals as input ports.

input [net_type] [signed] [range] port_names;

2. The output Port Declaration

used to declare a group of signals as output ports.

output [net_type] [signed] [range] port_names;

3. The inout Port Declaration

Used to dedare a group of signals as bidirectional ports.

inout [net_type] [signed] [range] port_names;

Ex: Adder

```
module adder (x, y, c_in, sum, c_out);
input [3:0] x, y;
input c_in;
output reg [3:0] sum;
output reg c_out;
```

* Parameters:

Parameters are constants, which can be used throughout the module defining them and are often used to specify delays and widths of variables.

- Parameter declaration:

The parameter is most commonly used to define module parameters that can be overridden by defparam or module instance parameter value assignment

syntax: parameter [signed] [range] param_assignments
parameter var_types param_assignments

Ex:
```
parameter SIZE = 7;
parameter WIDTH_BUSA = 24, WIDTH_BUSB = 8;
parameter signed [3:0] mux_selector = 4'b0;
```

- localparam Declaration

The localparam is used to define parameters local to a model. It is identical to parameter except that it cannot be overriden by the defparam statement or by module instance parameter value assignment.

Ex: localparam SIZE = 7;
localparam WIDTH_BUSA = 24, WIDTH_BUSB = 8;
localparam signed [3:0] mux_selector = 4'b0;

Thus, the localparam is used to define parameters when their values are not changed whereas the parameter is used to define parameters when their values may be changed at module instantiation or by using the defparam statement.

Parameter Dependence

Ex: parameter word_size = 16,
localparam memory_size = word_size * 512;

since memory_size depends on the value of word_size, a modification of word_size changes the value of memory_size accordingly.

- Parameter Ports:

When a parameter is placed between module name and port list / port list declarations, it is called a parameter port.

```
module module_name
#( parameter [signed] [range] param_assignments
   parameter var_types param_assignments
)
( port list or port list declarations)
 ...
endmodule
```

Page No.:
Date:

Ex: module module_name
#(parameter SIZE = 7,
parameter WIDTH_BUSA = 24, WIDTH_BUSB = 8,
parameter signed [3:0] mux_selector = 4'b0
)
(port list or port list declarations)
....
endmodule

* Module Instantiation:

A module can incorporate a copy (called an instance) of another module into itself through instantiation. To instantiate an instance of a module

module_name [#(parameters)] instance_name [range]
([ports]);
module_name [#(parameters)] instance_name
[x, instance_name y] ([ports]);

- Port connection Rules:

Connecting ports to external signals can be done by the following two methods:

- Named Association: The ports are connected by listing their names. The port identifiers and their associated port expressions are explicitly specified.

.port_id1 (port_expr1), ..., .port_idn (port_exprn)

An unconnected port is just skipped or places are empty such as .port_id().

- Positional Association: The ports are connected by the ordered list of ports, each corresponding to a port.

port_expr1, ..., port_exprn

An unconnected port is just skipped such as x, , y where a port is skipped between x and y.

NOTE:

- Unconnected ports are driven to the x state, unconnected outputs are not used.
- The real datatype is not permitted to pass directly to a port. Both $realtobits and $bitstoreal system functions are used to pass the bit pattern across module ports.
- Named association is usually preferred as synthesis tool may change the order of ports in the port list of the module they generate.

Q A parameterized module

```
module adder_nbit (x, y, c_in, sum, c_out);
parameter N=4;
input [N-1:0] x, y;
input c_in;
output [N-1:0] sum;
output c_out;
assign {c_out, sum} = x+y+c_in;
endmodule
```

★ Module Parameter Values:

When a module instantiates other modules, the features and operations of the module can be changed by modifying the parameters defined within it. This operation is called parameter override. The two ways to override parameter values are:

– Using the defparam statement:

A defparam statement is used to redefine the parameter values defined by the keyword parameter in any module instance throughout the design using the hierarchical path name of the parameter.

syntax:

```
defparam hierarchial_pathname1 = value1,
         hierarchial_pathname2 = value2,
         .....
         hierarchial_pathnamen = valuen,
```

Q: Ex: counter_nbits: default value of N = 4
into Two instances of counter_nbits with bits of 4 and 8 resp.

```
module two_counters (clock, clear, qout4b, qout8b);
input clock, clear;
output [3:0] qout4b;
output [7:0] qout8b;
defparam cnt_4b.N=4, cnt_8b.N=8;
counter_nbits cnt_4b (clock, clear, qout4b);
counter_nbits cnt_8b (clock, clear, qout8b);
endmodule
module counter_nbits (clock, clear, qout);
parameter N=4;
input clock, clear;
output reg [N-1:0] qout;
always @(posedge clear or negedge clock)
begin
   if (clear) qout <= {N{1'b0}};
   else       qout <= {qout+1};
end
endmodule
```

- Module Instance Parameter Value Assignment:

The parameters defined by using the keyword parameter within a module are overridden by the parameters passed through parameter ports whenever the module is instantiated.

- Positional association
- Named association

Q: Module instance parameter value Assignment : Positional association (For previous example)

```
— module two_counters (clock, clear, qout4b, qout8b);
    input clock, clear;
    output [3:0] qout4b;
    output [7:0] qout8b;
    counter_nbits #(4) cnt_4b (clock, clear, qout4b);
    counter_nbits #(8) cnt_8b (clock, clear, qout8b);
  endmodule
  module counter_nbits (clock, clear, qout);
    parameter N = 4;
    input clock, clear;
    output reg [N-1:0] qout;
    always @(posedge clear or negedge clock)
    begin
      if (clear) qout <= {N{1'b0}};
      else       qout <= qout + 1;
    end
  endmodule
```

Q: Module instance parameter value assignment : Named Association

```
— module two_counters (clock, clear, qout4b, qout8b);
    input clock, clear;
    output [3:0] qout4b;
    output [7:0] qout8b;
    counter_nbits #(.N(4)) cnt_4b (clock, clear, qout4b);
    counter_nbits #(.N(8)) cnt_8b (clock, clear, qout8b);
  endmodule
  module counter_nbits (clock, clear, qout);
    parameter N=4;
    input clock, clear;
    output reg [N-1:0] qout;
```

```
always @(posedge clear or negedge clock)
begin
    if(clear) qout <= {N{1'b0}};
    else      qout <= qout + 1;
end
endmodule
```

* **Hierarchial Path Names:**

An identifier can be defined within one of the following four elements:

- modules
- tasks
- functions
- named blocks

Within each element, an identifier must declare one item. Any identifier can be accessed directly within the elements defining it. To refer an identifier defined within the other elements, a hierarchial path name must be used.

Ex:
```
4bit_adder                      // top level
4bit_adder.fa_1                 // fa1 within 4bit_adder
4bit_adder.fa_1.ha_1            // ha_1 within fa_1
4bit_adder.fa_1.ha_1.xor1       // xor1 within ha_1
4bit_adder.fa_1.ha_1.xor1.s     // net s within xor1
```

* **Generate statement**

A generate statement allows selection and replication of some statements during elaboration time. The elaboration time is the time after a design has been parsed but before simulation begins.

The power of generate statements is that they can conditionally generate declarations and instantiations into a design.

General Form:

```
generate
    generate-declarations
    generate-loop statements
    generate-conditional statements
    generate-case statements
    generate-block
    nested generate statements
endgenerate
```

The generate statements cannot include parameters, localparameters and port declarations (input, output, inout).

There are three kinds of generate statements.

1. Generate-loop statement

A generate-loop statements is formed by using a for statement within a generate statement. The generate-loop allows statements to be duplicated at elaboration time.

```
for (init_expr ; condition_expr ; update_expr)
    begin : block_name
        generate_statements
    end
```

To use the for statement within a generate statement, it is necessary to declare an index variable genvar which is an integer. It is used only in the evaluation of a generate-loop, thus cannot have a negative value.

```
genvar genvar_id1, ...., genvar_idn;
```

Q: The generate-loop and assign

```
// converting Gray code to binary code
module gray2bin1(gray, bin);
  parameter SIZE = 8;
  input [SIZE-1:0] gray;
  output [SIZE-1:0] bin;
  genvar i;
```

```
generate for (i=0; i<SIZE; i=i+1)
  begin : bit
    assign bin[i] = ^gray[SIZE-1:i];
  end
endgenerate
endmodule
```

↳
```
assign bin[0] = ^gray[SIZE-1:0];
assign bin[1] = ^gray[SIZE-1:1];
- - - - -
assign bin[7] = ^gray[SIZE-1:7];
```

Q: The generate-loop and always

```
module gray2bin2 (gray, bin);
parameter SIZE = 8;
input [SIZE-1:0] gray;
output [SIZE-1:0] bin;
reg [SIZE-1:0] bin;
genvar i;
generate for (i=0; i<SIZE; i=i+1)
    begin : bit
      always @(*)
        bin[i] = ^gray[SIZE-1:i];
    end
endgenerate
endmodule
```

2. Generate-conditional statement

The generate conditional statement can be used alone or inside a generate loop.

→ if (condition) generate_statements
[else generate_statements]

→ if (condition1) generate_statements
[else if (condition2) generate_statements
[else generate_statements.

Q: A parameterized n-bit ripple-carry adder

```
// using module instantiations inside generate block:
module adder_nbit (x, y, c_in, sum, c_out);
  parameter N = 4;
  input [N-1:0] x, y;
  input c_in;
  output [N-1:0] sum;
  output c_out;
  genvar i;
  wire [N-2:0] c;
  generate
    for (i=0; i<N; i=i+1)
      begin: adder
        if (i==0)
          full_adder fa [x[i], y[i], c_in, sum[i], c[i]);
        else if (i==N-1)
          full_adder fa (x[i], y[i], c[i-1], sum[i], c_out);
        else
          full_adder fa (x[i], y[i], c[i-1], sum[i], c[i]);
      end
  endgenerate
endmodule
// full adder module
module full_adder (x, y, c_in, sum, c_out);
output sum, c_out;
input x, y, c_in;
assign {c_out, sum} = x+y+c_in;
endmodule
```

- // using continuous assignments inside generate block

```
module adder_nbit (x, y, c_in, sum, c_out);
  parameter N=4;
  input [N-1:0] x, y;
  input c_in;
  output reg [N-1:0] sum;
  output reg c_out;
  genvar i;
  reg wire [N-2:0] c;
  generate
    for (i=0; i<N; i=i+1)
      begin: adder
        if (i==0)
  always @(*) assign {c[i], sum[i]} = x[i]+y[i]+c_in;
        else if (i==N-1)
  always @(*) assign {c_out, sum[i]} = x[i]+y[i]+c[i-1];
        else
  always @(*) assign {c[i], sum[i]} = x[i]+y[i]+c[i-1];
      end
  endgenerate
endmodule
```

* using always block inside generate block

3. Generate-case statement:

The generate-case statements can be used alone or inside a generate loop:

```
case (case_expr)
case_item1: generate_statements
- - - .
case_itemn: generate_statements
  [default: generate_statements]
```

Q: A n-bit adder:

```
module adder_nbit (x, y, c_in, sum, c_out);
  parameter N=4;
  input [N-1:0] x, y;
  input c_in;
  output [N-1:0] sum;
  output c_out;
  genvar i;
  wire [N-2:0] c;
  generate
    for (i=0; i<N; i=i++)
      begin: adder
        case(i)
          0: assign {c[i], sum[i]} = x[i] + y[i] + c_in;
        N-1: assign {c_out, sum[i]} = x[i] + y[i] + c[i-1];
    default: assign {c[i], sum[i]} = x[i] + y[i] + c[i-1];
        endcase
      end
  endgenerate
endmodule
```

Keerthana Ashok

UNIT - 03

Dataflow Modelling

• Data Flow Modelling:

* Continuous Assignment:

A continuous assignment is the most basic statement of dataflow modeling. It continuously drives a value onto a net. The assignment begins with the keyword assign.

syntax: assign net_value = expression;

assign net1 = expr1, net2 = expr2, ..., netn = exprn;

Ex: assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;

(carries out a 4 bit addition with carry)

* Net Declaration Assignment:

A continuous assignment can be replaced on a net when it is declared it is called net declaration assignment.

```
Ex: wire out;                    // normal continuous assignment
    assign out = in1 & in2;
```

which is equivalent to

```
wire out = in1 & in2; // net declaration assignment
```

* Implicit Net Declaration:

A implicit net declaration will be inferred for a signal name when it is used on the left hand side of a continuous assignment.

```
Ex: wire in1, in2;
    assign out = in1 & in2;
```

Here out is not declared as a wire but an implicit wire declaration for out is done by the simulator.

Keerthana Ashok

* **Expressions:**

Expressions = Operands + operators.

Expressions consist of sub expressions, operators and operands. The operands in an expression can be constant, variables, net, reg, integer, time, real, real time, wire.

* **The precedence of operators in verilog HDL**

Operators	symbols	Precedence
Unary	+ ! ~	highest ↑
Exponent	**	
Multiply, devide, modulus	* / %	
Add, Subtract	+ −	
shift	<< >> <<< >>>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	& ~&	
	^ ^~	
	| ~|	
Logical	&&	
	||	
conditional	? :	lowest

when operators in an expression differ in precedence the operators with the higher precedence associate first. Parentheses can be used to change the operator precedence.

Page No.:
Date:
YOUVA

* **Delays:**

The time between when any operand in the right hand side changes and when the new value of the right hand side is assigned to the left hand side is controlled by the delay value.

→ The delay value of a continuous assignment:

```
assign #delay net-value = expression;
assign #delay1 net1 = expr1,
       #delay2 net2 = expr2,
       ----,
       #delayn netn = exprn;
```

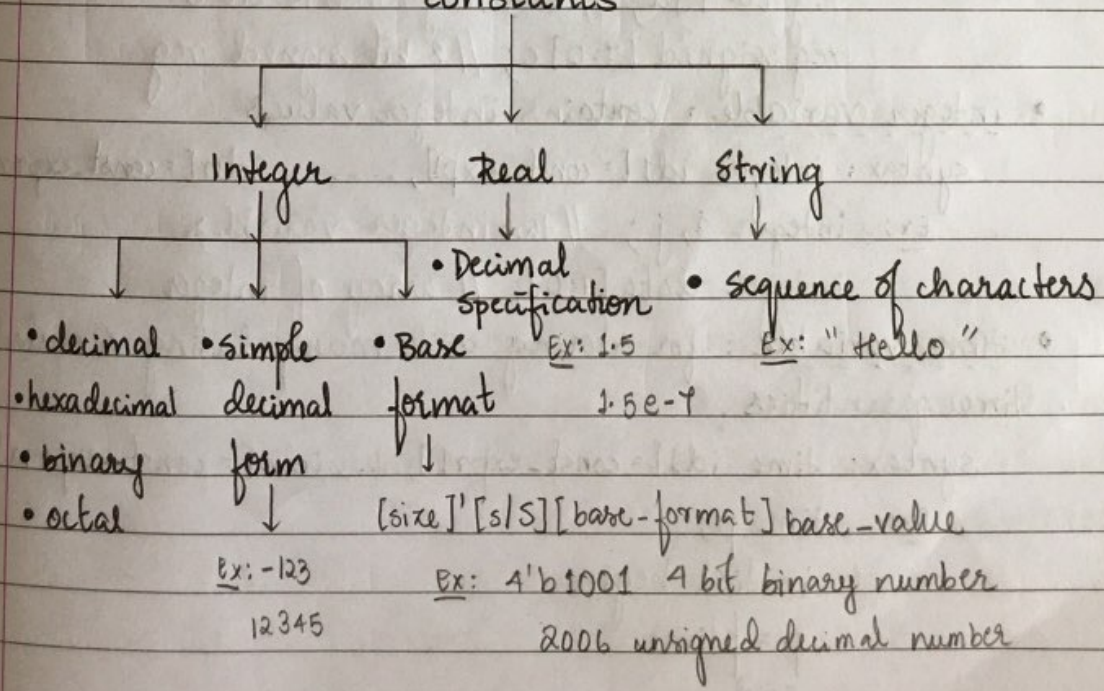
Ex:
```
wire in1, in2, out;
assign #10 out = in1 & in2;
```

→ The delay value of a net declaration assignment:

Ex:
```
wire #10 out = in1 & in2;
```

* **Operands:**

Operands in an expression can be constants, parameters, nets, variables (reg, integer, time, real, realtime).



Ex: string manipulation

```
module string_test;
reg [25*8:1] str;
initial
begin
str = "Hello!";
$display("%s\n", str);
end
endmodule
```

* Data Types

Verilog HDL has two classes of data types:

a. Variable Data Type

A variable represents a data storage element. The variable data type includes five different kinds:

- reg variable : holds a value between assignments

  syntax: reg [signed] [[msb:lsb]] reg1, reg2 ... regn.

  Ex: reg a, b, c_in; // one bit reg variables
  reg [7:0] a; // 8 bit reg; msb is bit 7
  reg [0:7] a; // 8 bit reg; msb is bit 0
  reg signed [7:0] c; // 8 bit signed reg

- integer variable : contains integer values

  syntax: integer id1 [= const_exp1], ..., idn [= const_exprn

  Ex: integer i, j; // two integer variables
  integer data [7:0]; // array of integer

- time variable : for storing and manipulating simulation time quantities.

  syntax: time id1 [= const_expr1], ..., idn [= const-exprn];

  Ex: time p;
  initial begin
  p = $time
  end.

Page No.: Date: YOUVA

- real variable:

  syntax: real identifier1,....,identifiern;

  Ex: real count; // declare a real variable

  initial begin

  count = 4e10;

  count = 2.14;

  end

- realtime variable:

  syntax: realtime identifier1,...., identifiern;

  Ex: realtime current_time // hold current time as real

* Array Elements:

Array can be used to group elements into multidimensional objects. Although only nets and the reg variable can be declared as vectors, all net and variable data types are allowed to be declared as multidimensional array.

Ex: wire a[3:0]; // scalar wire array of 4 elements

reg d[7:0]; // scalar reg array of 8 elements

wire [7:0] x[3:0]; // 8 bit wire array of 4 elements

reg [31:0] y[15:0]; // 32 bit reg array of 16 elements

integer states[3:0]; // integer array of 4 elements

time current[5:0]; // time array of 6 elements

To access an element from an array

array_name[addr_expr][addr_expr]

To assign a value to an element of an array, it needs to specify an index for every dimension.

states[3] = 33559; // assign decimal number to integer in array

current[t_index] = $time; // assign current simulation time to element indexed by integer index

* Memory Elements:

In Verilog HDL Memory is declared as a one-dimensional reg array.

To access a memory word

mem_name [addr-expr]

where addr-expr can be any expression.

Ex: reg [7:0] mema [7:0]; // one dimensional array of 8-bit vector.

reg [7:0] memb [3:0][3:0]; // two dimensional arra of 8 bit vector.

wire sum [7:0][3:0]; // two dimensional array of scalar wire

* Parameter:

Parameters can be used to determine the width of inputs and outputs.

Ex: parameter id = 9;

parameter addr = 16;

* String:

Ex: reg [8*5 : 1] str;

initial begin

str = "hello";

end

* Operators:

- Bitwise Operators:

Bitwise operators perform a bit-by-bit operation on two operands and produce a vector result.

~ : Bitwise negation

& : Bitwise and

| : Bitwise or

^ : Bitwise Exclusive or

~^, ^~ : Bitwise Exclusive nor

Ex: 4:1 Multiplexer

```
module mux1( i0, i1, i2, i3, s1, s0, out);
input i0, i1, i2, i3, s1, s0;
output out;
assign out=(~s1 & ~s0 & i0) | (~s1 & s0 & i1) |
                    (s1 & ~s0 & i2) | (s1 & s0 & i3);
endmodule
```

- Arithmetic Operators:

+ : addition

- : subtraction

* : multiplication

/ : division

% : modulus

** : exponent

Ex: to multiply 8 bit data and add the result with operand z and store the result in y.

```
module xyz(m, n, z, y);
input [7:0] m, n, z;
output [5:0] y;
assign y = m*n + z;
endmodule
```

Ex: Unsigned divider

```
module divider(x, y, result);
input [7:0] x, y;
output [7:0] result;
assign result = x/y;
endmodule
```

- concatenation and replication operators:
A concatenation operator is expressed by {, }

Ex: y = {a, b[0], c[1]};

{, }: concatenation

{const_exp{}}: Replication

Ex: 4 bit adder

```
module adder (x, y, c_in, sum, c_out);
input [3:0] x, y;
input c_in;
output [3:0] sum;
output c_out;
assign {c_out, sum} = x+y+c_in;
endmodule
```

Replication

Ex: y = {a, {4{b[0]}}, c[1]};

- Reduction Operators:
The set of unary operators carries out a bit-wise operation on a single vector operand and yields a 1 bit result. Reduction operators only perform on one vector operand and work in a bit by-bit way from right to left

&: Reduction and

~&: Reduction nand

|: Reduction or

~|: Reduction nor

^: Reduction exclusive or

~^, ^~: Reduction exclusive nor

It is performed on a single operand

Ex: assign out = &x;

- Logical operators
They operate on the logical values 0 and 1 and produce a 1-bit value 0, 1 or x.

Page No.:
Date:
YOUVA

! : logical negation
&& : logical and
|| : logical or

Ex: reg c = 123 ; d = 0
reg a, b;
reg [7:0] c, d;
a = c || d ; // a is set to 1
b = c && d; // b is set to 0

- Relational Operators:

Relational operators return the logic value 1 if the expression is true and 0 if the expression is false. The expression results in a value x if there are any x or z bits in the operands.

> : greater than
< : lesser than
>= : greater than or equal
<= : lesser than or equal

- Equality Operators:

Equality operators return the logical value 1 if the expression is true and 0 if the expression is false. These operators compare the two operands bit by bit, zero-extended if the operands are not of equal length.

== : logical equality
!= : logical inequality
=== : case equality
!== : case inequality

- Shift Operator:

The shift operation shifts the left operand by the number specified by the right operand. The vacant bit positions are filled with zeros for logical and arithmetic left-shift operations and filled with the MSBs (sign bits) for arithmetic right shift operation.

>> : Logical right shift
<< : Logical left shift
>>> : Arithmetic right shift
<<< : Arithmetic left shift

- Conditional Operator:
The conditional operator selects an expression based on the value of the condition expression.
condition_expr ? true_statement : false_statement

• Behavioural Modeling:

* Procedural constructs:
An initial or an always statement is usually called an initial or always block facilitates continuity and parallelism. Initial blocks are used to initialize variables and set values into variables or nets and always statements are used to model the continuous operations required in the hardware modules. All other behavioural statements must be within an initial or always block.
All initial and always statements execute concurrently with respect to each other (relative order in module is not important). Each of the initial and always statements represents a separate activity flow and each activity starts at the simulation time 0.

- initial block:
An initial block is composed of all statements inside an initial statement. It starts at simulation time 0 and executes exactly once during simulation.
syntax: initial [timing-control] procedural_statement
Ex: reg clock;
initial clock = 1'b0;

- always block:

An always block consists of all behavioural statements within an always statement. It starts at simulation time 0 and repeatedly executes the statements within it in a loop fashion during simulation.

syntax: always [timing_control] procedural_statement

Ex: reg clock; initial clock = 1'b0;
always #5 clock = ~clock; // period = 10

* Procedural Assignments:

Procedural assignments are placed inside initial or always statements. They update the values of variable data types, reg, integer, time, real, realtime, or array elements.

syntax: variable_value = [timing_control] expression
[timing_control] variable_value = expression.

NOTE: Difference between procedural and continuous assignment

continuous assignments drive nets, which are evaluated and updated whenever any input operand changes its value. The procedural assignments update the values of variables under the control of the procedural flow constructs that surround them. Thus the LHS of continuous assigments are nets but the LHS of procedural assigments are variables. Procedural assigments are used within always and initial block only.

There are two types of procedural assignments:

- Blocking Assignments:

Blocking assignment statements use the operand "=" and are executed in the order that they are specified. It is executed before the execution of statements that follows it in a sequential block (statement grouped with begin and end keyword).

Thus blocking assignment statements are executed sequentially.

Keerthana Ashok

Ex: module blocking;
reg x, y, z;
initial begin
x = #5 1'b0; // x=0 at time 5
y = #3 1'b1; // y=1 at time 8
z = #6 1'b0; // z=0 at time 14

- Non blocking assignment

Non blocking assignment use the <= operator and are executed without blocking the other statements in a sequential block. They provide a method to transfer data concurrently after a common event i.e., several variables assignments are made within the same time regardless of the order or the dependency of each other.

Ex: module nonblocking;
reg x, y, z;
initial begin
x <= #5 1'b0; // x=0 at time 5
y <= #3 1'b1; // y=1 at time 3
z <= #6 1'b0; // z=0 at time 6

Mixed use of blocking and nonblocking assignments

Ex: module mix;
reg a, b, c, d;
initial begin
a <= #5 1'b0; // a=0 at time 5
b = #3 1'b1; // b=1 at time 3
c = #6 1'b0; // c=0 at time 9
d = #7 1'b1; // d=1 at time 10
end
endModule

Executing nonblocking statements during simulation

→ Read: read the values of all RHS variables
→ Evaluation: evaluate the RHS expressions and store

temporary variables that are scheduled to assign to the LHS variable later.

→ Assignment: assign the values stored in temporary variables to the LHS variables.

NOTE:

- Do not mix blocking and non blocking assignments in the same always block.
- In the always block it is good to use non blocking operators (<=) for sequential logic and blocking operators (=) when it is combinational logic.

* Timing control:

Timing controls provide a way to specify the simulation time at which procedural statements will execute.

There are two timing control methods provided in Verilog HDL:

- Delay Timing Control:

Delay timing control in an expression specifies the time duration between when the statement is encountered and when the statement is executed.

- # delay_value
- #min:typ:max_expression.

His further divided into two types according to the position of the delay specifier in a statement

a. Regular Delay control

It defers the execution of the entire statement by a specified number of time units. It is specified by putting a non-zero delay to the left of a procedural statement.

delay procedural_statement

Ex: #25 x = a+b // wait 25 time units and execute

The regular delay control defers the execution of the entire statement so there is no difference whichever "<=" or "=" is used.

b. Intra-assignment delay control:

Intra-assignment delay control defer the assignment to the LHS variable by a specified number of time units but the RHS expression is evaluated at the current simulation time.

variable_value = #delay expression

variable_value <= #delay expression

Ex: → y = #25 x; // assign to y at time 25
count = #15 count + 1; // assign to count at time 15
but both statements are evaluated at time 0

→ y <= #25 x; // assign to y at time 25
count <= #15 count+1; // assign to count at time 15
but both statements are evaluated at time 0

- Event Timing Control:

An event control with a procedural statement defers the execution of the statement until the occurrence of the specified event.

a. Named Event control:

A new data type called event can be declared in addition to nets and variables. This provides users a capability to declare an event and then to trigger and recognise it. An identifier declared with an event is called a named event. The event doesnot hold any data and has no time duration. A named event is triggered explicitly by using the symbol -> and the triggering of the event is recognized by using the symbol @.

Ex: // step 1: declare an event
event received_data;
// step 2: trigger the event received_data
always @ (posedge clock)
if (last_byte) -> received_data;
// step 3: recognize the event

always @ (received_data)
begin
// step 4: put the required operations
end

b. Event or control:

In many applications any one of multiple signals or events can trigger the execution of a procedural statement or a block of procedural statements. The signals or events represented as logical or are also called a sensitivity list or event list. The keyword or or a comma (,) is used to specify multiple triggers in a sensitivity list.

Ex: always @ (a or b or c_in)
{c_out, sum} = a + b + c_in;

is equivalent to

always @ (a, b, c_in)
{c_out, sum} = a+b+c_in;

* Selection statements:

selection statements are used to make a selection according to a given condition.

- if-else statement:

syntax: • if (<condition>) true_statement
• if (<condition>) true_statement
else false_statement
• if (<condition1>) true_statement1
else if (<condition2>) true_statement2
else false_statement.

Ex: Upcounter

module upc (c, r, q);
input c, r;
output reg [3:0] q;
initial

Keerthana Ashok

```
always @(c)
begin
  q = 4'b0000;
  if (r==1)
    q=4'b0000;
  else
    q = q+1;
end
endmodule
```

- **case statement:**

A case statement is used to perform a multiway selection according to a given input condition and equivalent to nested if-else statement.

```
syntax: case (case_expression)
          case_item1: statement1;
          case_item2: statement2;
            ⋮
          case_itemn: statementn;
          default: default_statement;
          endcase
```

The case_expression is evaluated first. Then the case_item expressions are evaluated and compared in order given. The statement associated with the case_item that first matches the case_expression is executed. When none of the case_item expression matches the case_expression, default statement is executed if it exists as it is optional.

Ex: Calculator

```
module calci(a,b,c,d,e,f);
input [31:0] a,b,s;
output [31:0] c,d,e,f;
always @(*)
begin
```

```
case (s)
  2'b00 : c = a+b;
  2'b01 : d = a-b;
  2'b10 : e = a*b;
  2'b11 : f = a/b;
endcase
end
endmodule
```

- casex and casez statements:

Both casex and casez statements are used to perform a multiway selection like that of case statement except that the casez statement treats all z values as don't cares while the casex treates all x and z values as don't cares. These two statements only compare non-x or z positions in the case_expr and the case_item expressions.

Ex: counting trailing zeros in a nibble

```
module trailing_zeros(data, out);
input [3:0]data;
output reg [2:0]out;
always @(data)
  casex (data)
    4'bxxx1 : out = 0;
    4'bxx10 : out = 1;
    4'bx100 : out = 2;
    4'b1000 : out = 3;
    4'b0000 : out = 4;
    default : out = 3'b111;
  endcase
endmodule
```

Page No.:
Date:

* Iterative (loop) statements:

Loop statements provide a means of controlling the execution of a statement or a block of statements zero, one or more times. These include four types of loop statements:

- while statement

while loop statement executes the procedural statement until the given condition becomes false. If the calculated value of the expression is z or x it is treated as false

syntax: while (condition_expr) procedural_statement

Ex:
```
initial
begin
a=20;
i=0;
while(i<a)
begin
$display("%d", i);
i=i+1;
a=a-1;
end
end
```

- For loop statement

The for loop is usually used to perform a counting loop which is a loop that is repeated with a fixed number of times. A for loop statement contains three parts:

- an initial condition: executes an assignment to initialize a variable that contains the number of times block is executed.
- termination condition: evaluates an expression, if result is false or z or x, the for loop terminates and if the for loop shall continue the execution.
- control variable updating: executes an assignment,

normally used to modify the value of the loop control variable and then repeats termination condition checking.

syntax: for (init_expr; condition_expr; update_exp)
procedural statement

Ex:
```
initial
begin
a=20
for (i=0; i<a; i=i+1, a=a-1)
$display ("%d", a);
end
```

- Repeat loop statement

Repeat loop statement continuously executes a block for a given number of time which can be mentioned by using a constant or an expression. This expression value is calculated only once, i.e., before the start of the loop. If the expression value turns out to be x or z, then it is treated as 0, hence the block is not executed at all.

syntax: repeat (counter-expr) procedural statement

Ex:
```
initial
begin
a=10;
b=5;
b<= #10 10;
i=0;
repeat (a*b)
begin
$display ("Repeat in progress");
#1 i=i+1;
end
end
```

Page No.
Date:

— Forever loop statement:

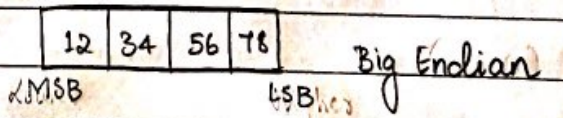
The forever loop continuously performs a procedural statement until the $finish system task is encountered.

syntax: forever procedural-statement.

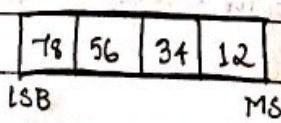
Ex:
```
initial
begin
  clock <= 0;
  forever
  begin
    #clock <= ~clock;
  end
end
```

Q: Write a verilog code for converting Big Endian to Little Endian and vice versa.

—
```
module swap_byte(in, out);
input [31:0] in;
output [31:0] out;
assign out[31-:8] = in[0+:8];
       out[23-:8] = in[8+:8];
       out[15-:8] = in[16+:8];
       out[7-:8] = in[24+:8];
endmodule
```



Big Endian



Little Endian