Introduction to Verilog: Design Methodology :-

Saturday, February 13, 2021 12:56 PM

Books:- Verilog HDL by Samir Palnitkar

Advanced Digital System Design Using Verilog HDL } by Cilicti

Verilog History:- 1983

Phil Moorby and Prabhu Goel } Gateway design Automation

Proprietary Hardware - description language.

1989 :- Cadence bought Gateway design.

In December 1995 Verilog HDL became IEEE 1364 - 1995.

Evolution of Computer - Aided Digital Design :-

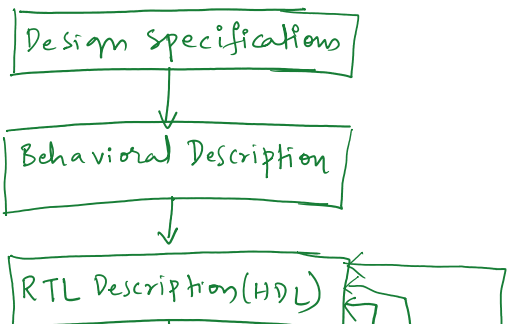
Vacuum Tubes & transistors

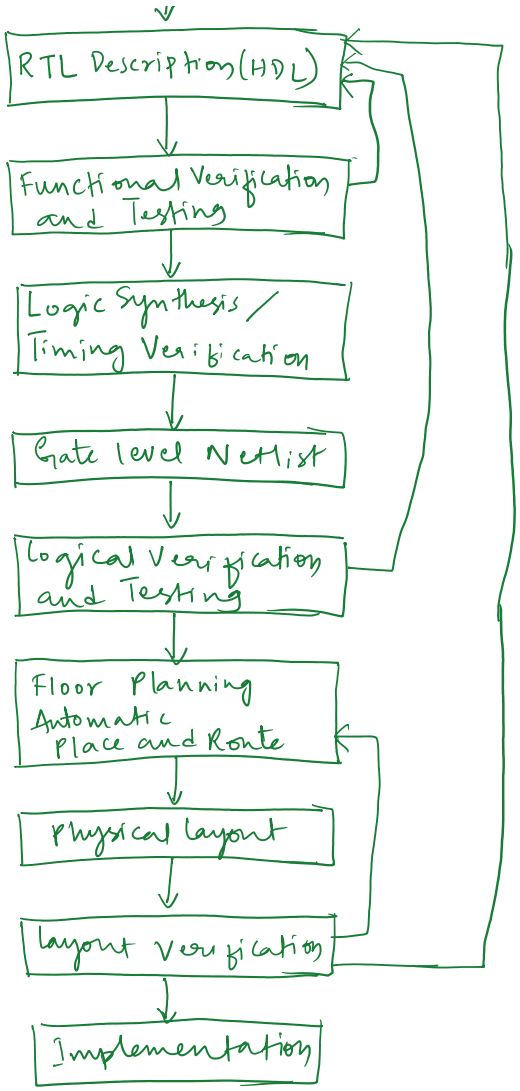
SSI IC's → gate count was small

MSI IC's :- → Hundreds

LSI IC's → Thousands of gates on a chip

Typical Design Flow :-





Importance of HDLs :-

1) Designs can be described at a very abstract level by the use of HDL.

2) By describing designs in HDLs, functional Verification of the design can be done early in the design cycle

3) Designing with HDLs is analogus to Computer Programming.

Popularity of Verilog HDL :-

1) Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use.

2) Verilog HDL allows different levels of abstraction to

2) Verilog HDL allows different levels of abstraction to be mixed in the same model.

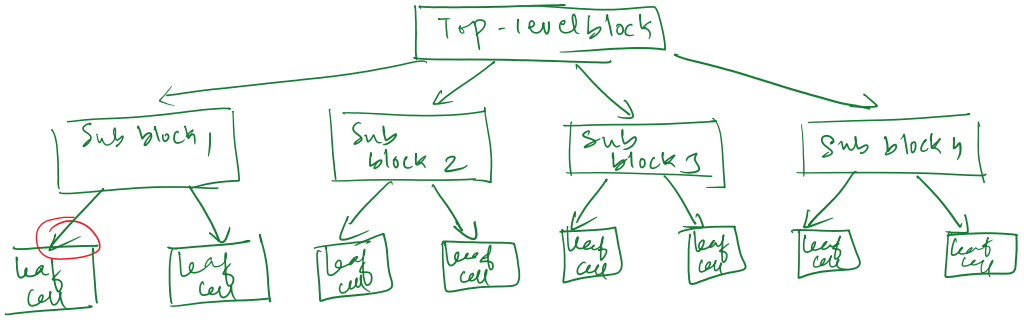
3) Almost all logic synthesis tools support Verilog HDL.

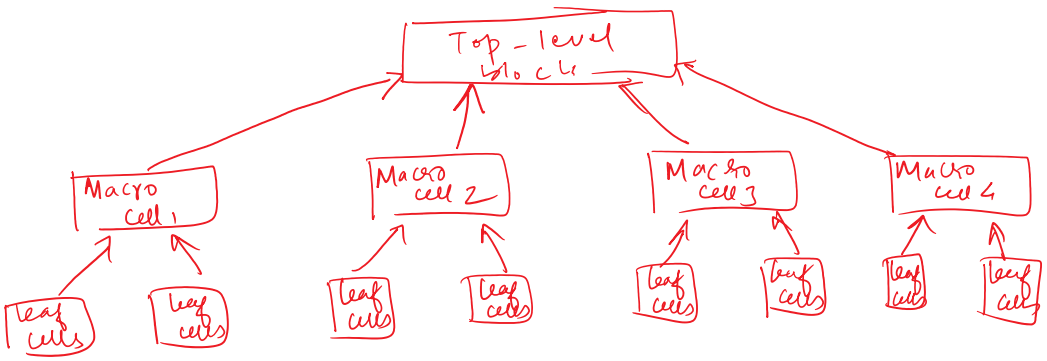
4) All fabrication Vendors provide Verilog HDL libraries for post logic synthesis simulation

5) PLI (Programing Language Interface) C Code

Design Methodologies :-

Top down :- Define the top level blocks and Identify the Sub-blocks necessary to build the top-level block. We further Sub-divide the Sub-blocks until we come to leaf cells, which are the cells that cannot be further divided.





Modules :- A Module is a basic building block in Verilog. A module is declared by the keyword module. A corresponding keyword endmodule must appear at the end of the module

T_FF (q, clock, reset)

Module <moduleName> (<Module Terminal_list>);

=

general

module <module name> (<module terminal_list>);

Module <internals <functionality of T-flip-flop

endmodule

general structure

→ There are 4 levels of abstraction in which Module can be described

1→ Behavioral or Algorithmic level :-

This is highest level of abstraction given by Verilog HDL. Module is implemented interms of desired design Algorithm without concern for the hardware implementation details. It is similar to C Program

2→ Dataflow level :- At this level module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers & how the data is processed in the design.

3→ Gate level :- Module is implemented in terms of logic gates and interconnections bet'n these gates.

4→ Switch level :- lowest level of abstraction in Verilog Module is implemented in terms of switches, storage nodes & interconnections between them.

Instances :-

module ——— ———
{
endmodule
module ——— ———
:
endmodule

ONE Module definition cannot contain another module definition within the module and endmodule statement.

Lexical Conventions :- Verilog HDL is a case-sensitive language

All keywords are in lower case.

i) whitespace :-

ii) Comments :- Two ways to write comments

i) whitespace :-

ii) Comments :- Two ways to write Comments

One-line comment starts "//"

Multiple line "/*" & ends with "*/"

```
a = b && c; // This is a one line comment
/* This is a multiple line comment */
/* this is /* an illegal block */ comm */ ] illegal.
```

3) Operators :- There are 3 types of operators

i) Unary operator ii) Binary iii) Ternary

Unary operators precedes the operand, Binary operators appear between two operands. Ternary operators have two seperate operators that seperates 3 operands

✓ `a = ~b;` // ~ is a unary operator, b is the operand

✓ `a = b && c;` // && is a binary operator & b and c are operands

✓ `a = b ? c : d;` // ?: is a ternary operator, b c, d are operands

Number Specifications :- 2 types of no. specifications i) Sized ✓
ii) Unsized ✓

Sized numbers :- These are represented as

<size>'<base format><number>

<size> specifies the no. of bits in the number

<base format> (d, or D) (h or H) (b or B) (o or O)

<number> 0, 1, 2, 3, ... 9, a, b, c, d, e, f

Ex :- `4'b1111` // This is a 4-bit binary no

`12'habc` // This is a 12-bit hexadecimal no

`16'd255` // This is a 16-bit decimal no

16'd255 // This is a 16-bit decimal no

Unsized No.s :- Numbers that are specified without <base format> specification are decimal no.s by default. Numbers that are written without <size> specification have a default number of bits that simulator and machine specific.

23456 // Decimal no. by default

'hC3 // hexadecimal no.

X or Z values :- Verilog has two symbols for unknown and high impedence value.

00010011xxxx

12'h13x // This is a 12-bit hex number; 4 Least significant bits unknown.

6'hx // This a a 6-bit hex no. which is unknown

32'bz // This is a 32-bit high impedence no

underscore character :- An underscore character is allowed anywhere in a no. except the first character.

12'b1111_0000_1010] To improve the readability of no.s are ignored by verilog.

Strings :- "Hello Verilog word"

Identifiers and keywords: keywords are special Identifiers to define the language constructs. keywords are in lowercase. Identifiers are the names given to objects so that they can be referenced in the design.

reg value; // reg is keyword ; value is identifier

input clk ; // input is keyword, clk is identifier.

Data types:-

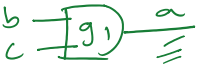
1) Value set :- Supports 4 values & 8 strengths to model the functionality of real hardware

1) Value set :- Supports 4 values & 8 strengths to model the functionality of real hardware

Value Level	Condition in Hardware circuit
0	Logic zero, false condition
1	Logic one, True condition
x	Unknown logic value
z	High Impedence, floating state

strength level	Degree
supply	Strongest
strong	
pull	
large	
weak	
Medium	
small	
highz	Weakest

2) Nets :- They represent connections between Hardware Elements.



] Nets are declared primarily using the keyword wire

```
wire a
wire b, c;
```

default value of net is z [tri reg net x]

3) Registers :- Registers represent data storage elements. Unlike a net, a register does not need a driver. They are declared by keyword reg. The default value of reg data type is x.

```
reg reset ;
initial
  begin
    ..
```

```
initial
  begin
  reset = 1'b1;
#100  reset = 1'b0;
  end
```

Vectors :- Nets or Registers can be declared as Vector (multiple bit widths).

```
wire a //
wire [7:0] bus ; //  8-bit bus
wire [31:0] busA, busB, busC ;// 3 buses of 32-bit width
reg [0:40] Virtual_addr //
```

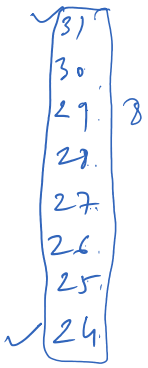
[high# : low #] or [low# : high#]

Variable Vector Part select :-

[<starting bit>+ : width] — part-select increments from Startingbit

[<starting bit> - : width] - " " " decrement " " "

```
reg [255 : 0] data1; // Little endian notation
reg [0 : 255] data2; // Big endian notation
reg [7:0] byte;
byte = data1 [31-:8]; // starting bit = 31, width = 8
                      => data [31:24]
byte = data1 [24+:8]; // Starting bit = 24   width = 8
                      => data [31:24]
byte = data2 [31-:8]; // Starting bit = 31, width = 8
                      => data [24:31]
```



Integers, Real and Time Register data Types

Integers, Real and Time Register data Types

Integer : keyword **integer**

```
integer counter
initial
    counter = -1 ; //
```

Real : real , default value is zero

Time :- Verilog simulation is done wrt simulation time

time { keyword }

$time [the system function $time is invoked to get current simulation time

Arrays :- Arrays are allowed in Verilog reg, integer, time, real & for vector <Array Name> [<subscript>]

```
integer count [0:7]; // An Array of 8 count variables
real bool [31:0]; // Array of 32 one bit boolean register variables
reg [4:0] port_id [0:7]; // Array of 8 port_ids ; each port id is of 5 bits wide
```

System Tasks and Compiler directives:-

All the system tasks appear in the form $<keyword>

Displaying Information:-

$display { It displays the values of variables or strings of expression }

$monitor → Monitor a signal when its value changes
* it should be invoked only once.

I . . . stops during simulation [Debug]

* If [illegible]

$stop → stops during simulations [Debug]

$finish → Terminates the simulation

Compiler directives :- These are defined using the

`< keyword> construct

`define → directive used to define text macros

`include → Allows you to include entire contents of a a verilog source file in another verilog file during compilation.

Modules and Ports :-

Module Name
Port List, Port Declaration [if ports are present]
Parameters (optional)

Declarations of wires, registers & other variables

Dataflow assignment

Instantiation of lower level Modules
Tasks & functions

always & initial blocks

End module statement

Dataflow Modelling :- We describe continuous assignment statement in verilog code. (assign is the keyword used)

1) Continuous Assignment :- Is the most basic statement of data flow modeling,

i) The left hand side of an assignment must always be a scalar or vector net or concatenation of scalar & vector nets. It cannot be a scalar or vector register.

2) Continuous assignments are always active. The assignment

vector nets. It cannot be a scalar or vector register.

2) Continuous assignments are always active. The assignment expression is evaluated as soon as one

3) Delay value can be specified for assignments in terms of time units

4) The operands on the RHS can be registers or nets or function calls

Ex: assign out = i1 & i2;

assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

assign {C_out, Sum[3:0]} = a[3:0] + b[3:0] + C_in;

II Implicit Continuous Assignment:-

wire out;
assign out = in1 & in2;

wire out = in1 & in2;

III Implicit Net Declarations

wire i1, i2;
assign out = i1 & i2;

Delays :- 1) Regular Assignment delay

2) implicit continuous Assignment delay

3) Net declaration delay.

[assign #10 out = in1 & in2; // delay in a continuous ass]

[wire #10 out = in1 & in2;]

[wire #10 out;
assign out = in1 & in2;]

Operators :-

Operators :-

1) Arithmetic Operators :- [Binary & Unary]

Binary operators

A = 4'b0011 ; B = 4'b0100

D = 6 ; E = 4 , F = 2

A * B // 4'b1100

A + B // 4'b 0111

B - A // 4'b0001

D/E // Evaluates 1

But any operand bit has a value x, then the result of entire expression is x

in1 = 4'b101x;

in2 = -4'b1010;

-10 / 5 : -2

Sum = in1 + in2 ; // 4'bx

2) Logical Operators :-

Logical and (&&), logical or (||), logical Not (!)

→ They always evaluate to a 1-bit value 0 (false), 1 (true) or x (ambiguous)

→

A = 3, B = 0

A && B // Evaluates 0

A || B // Evaluates 1

!A // Evaluates to 0

!B // Evaluates to 1

A = 2'b0x : B = 2'b10

A = 2'b0x ; B = 2'b10

A && B // Evaluates to x

3) Relational operators :-

Greater-than (>), greater-than-or-equal-to (>=)

less-than (<), less-than-or-equal-to (<=)

// A = 4, B = 3

// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A <= B // Evaluates to a logical 0

A > B // " " " 1

Y < Z // " x

Equality operators :-

i) Logical Equality (==)
ii) logical inequality (!=)
} → 0, 1, x (false, true, unknown)

iii) case equality (===)
iv) case inequality (!==)
} → 0, 1 { case equality operator never result in x }

A = 4, B = 3

N = 4'b1xxx

X = 4'b1010, Y = 4'b1101, Z = 4'b1xxz, M = 4'b1xxz,

A == B // It results in logical 0

X != Y // It results in logical 1

X === Z // It results in x

Z === M // It Result in logical 1

Bitwise operators :- Performs bit-by-bit operation on operands

negation (~) and (&), or (|), xor (^), xnor (^~ or ~^)

Bitwise operators

negation (~), and (&), or (|), xor (^), xnor (^~ or ~^)

* (z is treated as an x in a bitwise operation)

bitwise and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitwise or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bitwise xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitwise xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

bitwise negation	result
0	1
1	0
x	x

x = 4'b1010 ; x = 4'b1010

z = 4'b10x1

x & y = 4'b1000

~x = 4'b0101

x & z = 4'b10x0

Reduction operators :- [1-bit Result] xnor(~^, ^~)

and (&) nand (~&), or (|), nor (~|), xor (^) &

* Reduction operators take only one operand.

x = 4'b1010

&x // Equivalent to 1 & 0 & 1 & 0, Results in 1'b0

|x // Equivalent to 1 | 0 | 1 | 0, Results in 1'b1

^x // Equivalent to 1 ^ 0 ^ 1 ^ 0, Results in 1'b0

^x // Equivalent to 1^0^1^0, Results in 1'b0

Shift operators :-

right shift (>>), Arithmetic left shift (<<<)

left shift (<<), Arithmetic Right shift (>>>)

x = 4'b1101  Result

y = x >> 1; y = 4'b0110

y = x << 1; y = 4'b1010

y = x <<< 2; y = 4'b0100

Concatenation operator :- ({,}) It provides a mechanism to append multiple operands. Operands must be sized.

A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110

Y = {B, C} // Result of Y is 4'b0010

Y = {A, B, C, D, 3'b001} // Result Y is 11'b10010111001

Y = {A, B[0], C[1]} // Result Y is 3'b101

Replication operators :- ({{}}) ← Replicate ({,}) ← Count

A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110

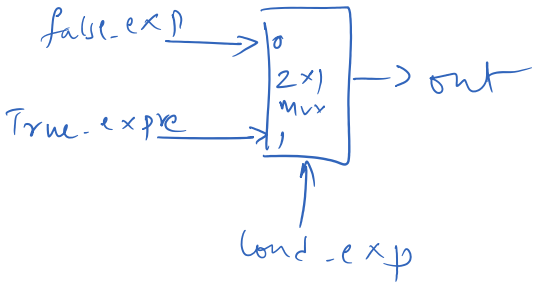
Y = {4{A}} // Y = 4'b1111

X = {4{A}, 2{B}} Y = 8'b11110000

Conditional operator :- (?:)

Conditional operator : (? :)

usage: condition_expression ? true_exp : false_expr ;



Unary, Multiply, Divide, Modulus — Highest Precedence

Add, Substract, Shift

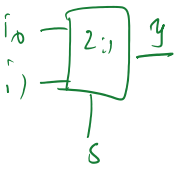
Relational, Equality

Reduction, Logical

Conditional — Lowest precedence

1) Design a 2:1 MUX using CAS in Verilog

```
Module mux2to1 (i,s,y);
input [1:0] i;
input s;
output y;
assign y = (i[0] & (~s)) | (i[1] & (s));
```



or

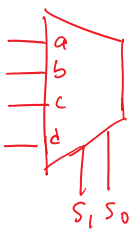
```
assign y = s ? i[0] : i[1];
```

or

```
assign y = (s==1) ? i[1] : i[0];
Endmodule
```

2) Design a 4:1 Multiplexer using assignment stmt &

2) Design a 4:1 Multiplexer using assignment state & Conditional operator.



S_1	S_0	y
0	0	a
0	1	b
1	0	c
1	1	d

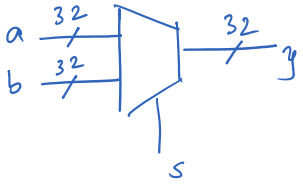
```
module  mux4to1 (a,b,c,d,s1,s0,y);
input a,b,c,d;
output y;
input s1, s0;
assign y = ( (~s0)&(~s1)& a) | ((~s0) & (s1) & b) |
           ((s0) & (~s1) & c) | (s0 & s1 & d) ;
```

With Conditional Operator :-

```
assign y = s1 ? s0 ? d : c : s0 ? b : a ;
                  1            0
```

Endmodule

3) Write a Verilog code for following design

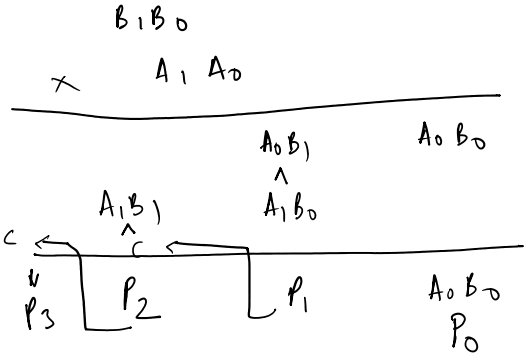



```
module design1 (a,b,s,y)
input [31:0] a,b;
output [31:0] y
input s;
assign  y = s ? a : b;
Endmodule
```

4) Write a Verilog code for 2-bit Magnitude comparator using CAS { Home work }

5) Write a Verilog code for 2x2 Array Multiplier

$B_1B_0 \times A_1A_0$



$P_0 = A_0B_0$

$P_1 = A_0B_1 \wedge A_1B_0$

$P_2 = A_1B_1 \wedge C$

$P_3 = A_1B_1 \& C$

$C = (A_0B_1) \& (A_1B_0)$

```
... assignment ( input [1:0] a,1   output [3:0] P);
```

P_3 P_2 P_1 P_0

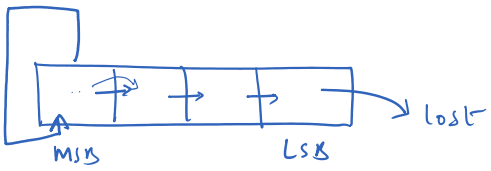
```
module arraymul (input [1:0] a,b, output [3:0] p);
wire c;
assign p[0] = A[0] & B[0];
assign p[1] = (A[0] & B[1]) ^ (A[1] & B[0]);
assign c = (A[0] & B[1]) & (A[1] & B[0]);
assign p[2] = (A[1] & B[1]) ^ c
assign p[3] = A[1] & B[1] & c;
endmodule
```

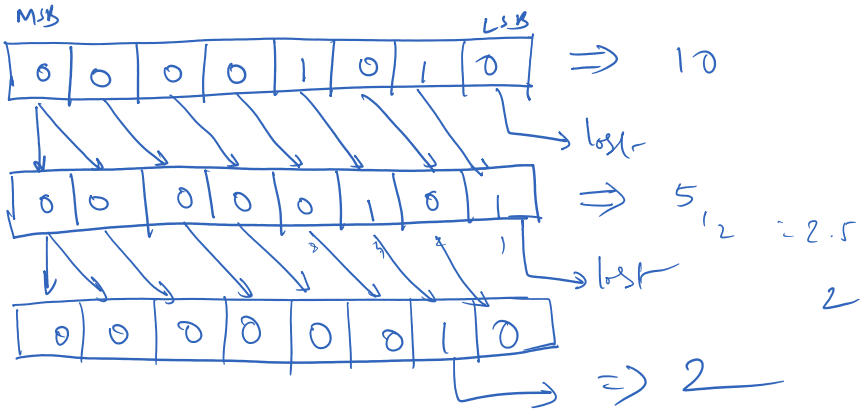
Arithmetic shift Right & shift left :-

→ Arithmetic shift left { Multiplying a no. by 2 }

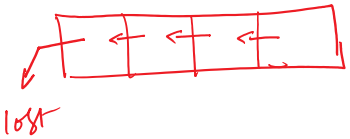
→ Arithmetic shift Right { Divide a no. by 2 }

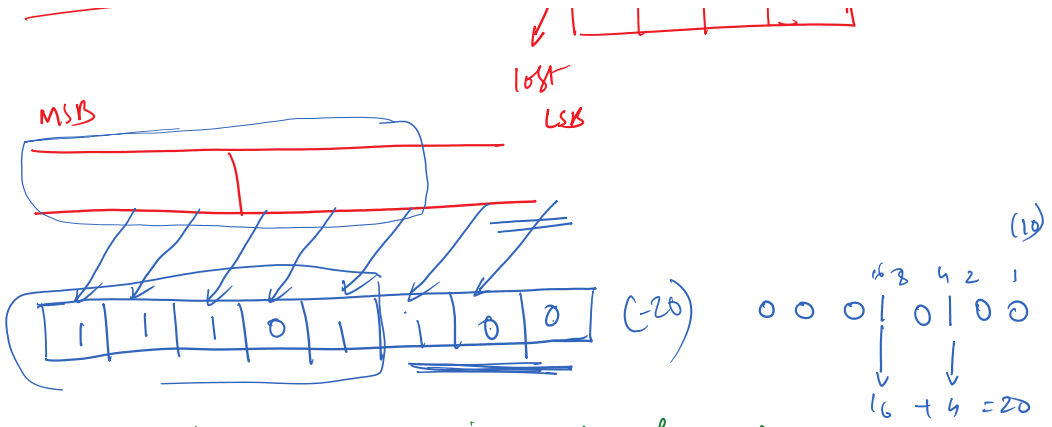
→ Arithmetic shift Right :-





Arithmetic left shift :- Multiply by 2



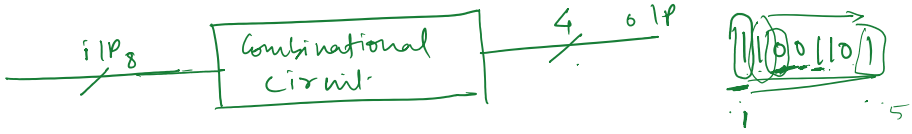


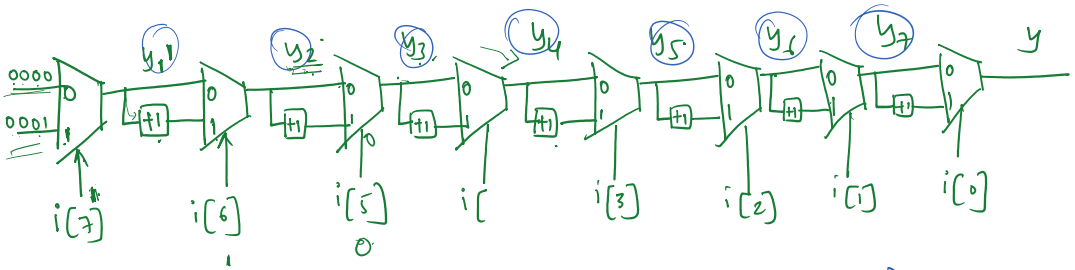
5) Write a Verilog code using CAS for 2:4 decoder.

```
assign y[0] = En & (~s[0]) & (~s[1]);
assign y[1] = En & (~s[0]) & s[1];
assign y[2] = En & s[0] & (~s[1]);
assign y[3] = En & s[0] & s[1];

assign y = s[1] ? (s[0] ? 4'b1000 : 4'b0100) :
           (s[0] ? 4'b0010 : 4'b0001);
```

6) Write a Verilog code that produces a 4-bit output indicating number of 1's in a 8 bit no. word.






```
module count8 (input [7:0] a, output [3:0] y);
wire [3:0] y1, y2, y3, y4, y5, y6, y7;
assign y1 = i[7] ? 0001 : 0000;
assign y2 = i[6] ? y1+1 : y1;
```

```
assign y3 = i[5] ? y2+1 : y2;
assign y4 = i[4] ? y3+1 : y3;
assign y5 = i[3] ? y4+1 : y4;
assign y6 = i[2] ? y5+1 : y5;
assign y7 = i[1] ? y6+1 : y6;
assign y  = i[0] ? y7+1 : y7;
endmodule
```

D-latch :-

En	Qout
1	Data
0	hold

```
module d-latch (en, data, q)
 input en, data;
 output q;
assign q = en ? data : q
endmodule
```

Verilog Primitives :- AND, OR, NAND, NOR, XOR, XNOR, NOT, buf

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

OR	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

NOR	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

XOR | 0 1 X Z

XNOR | 0 1 X Z

XOR	0	1	X	Z
0	0	1	x	x
1	1	0	x	x
X	x	x	x	x
Z	x	x	x	x

XNOR	0	1	X	Z
0	1	0	x	x
1	0	1	x	x
X	x	x	x	x
Z	x	x	x	x

Not	o/p
0	1
1	0
X	X
Z	X

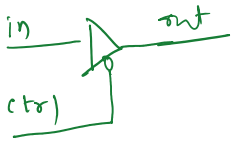
Buf	o/p
0	0
1	1
X	x
Z	Z

Bufif1, Bufif0, notif1, notif0 gates:-

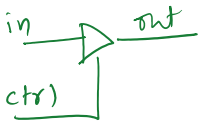
These are the gates with additional control signals. These gates have 3 ports. The first is o/p port, 2nd is data port & third is control port.

These gates propagate only if their control signal is asserted. & They propagate Z if their control signal is deasserted.

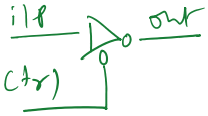
Bufif0



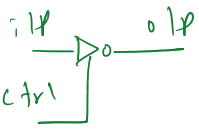
Bufif1



notif0



notif1



bufif0	0	1	X	Z
0	0	Z	L	L
1	1	Z	H	H
X	X	Z	X	X
Z	X	Z	X	X

(control signals; data i/p)

L ⇒ weak zero & Z

H ⇒ weak one

notif0	0	1	X	Z
0	1	Z	H	H

(control)

	0	1	X	Z
0	Z	0	L	L
1	Z	1	H	H
X	Z	X	X	X
Z	Z	X	X	X

control

buf if1 — data i/p

	0	1	X	Z
0	1	Z	H	H
1	0	Z	L	L
X	X	Z	X	X
Z	X	Z	X	X

notif0 — data i/p

	0	1	X	Z
0	Z	1	H	H
1	Z	0	L	L
X	Z	X	X	X
Z	Z	X	X	X

notif1