Unit-II

By Smitashree Mohapatra

Switch Level Modelling

- Verilog provides the ability to design at a MOS-transistor level.
- Design at this level is becoming rare with the increasing complexity of circuits (millions of transistors) and with the availability of sophisticated CAD tools.

Switch-Modeling Elements

- Verilog provides various constructs to model switch-level circuits. Digital circuits at MOS-transistor level are described using these elements.

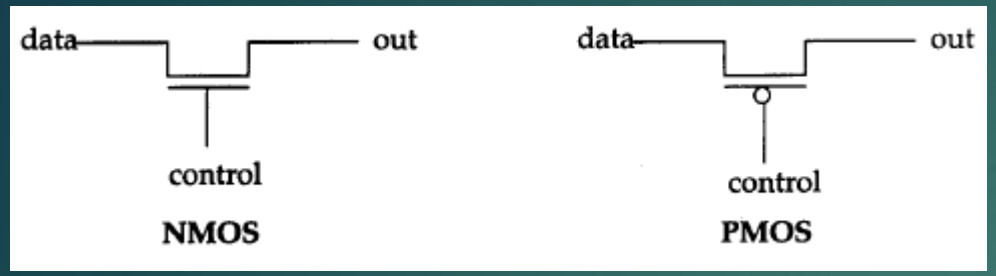
MOS Switches

- Two types of **MOS** switches can be defined with the keywords, **nmos** and pmos.

```
//MOS switch keywords
  nmos          pmos
```

- Keyword **nmos** is used to model ***NMOS*** transistors; keyword pmos is used to model **PMos** transistors

- The symbols for **nmos** d **pmos** switches are shown in the below Figure



- ***Instantiation of NMOS and PMOS Switches***

  nmos nl (out, data, control) ; **//instantiate a nmos switch**

  pmos pl(out, data, control); //instantiate a pmos switch

- Since switches are Verilog primitives, like logic gates, the name of the instance is optional. Therefore, it is acceptable to instantiate a switch without assigning an instance name.

  nmos (out, data, control); //instantiate an nmos switch;no instance name

  pmos (out, data, control) ; **//instantiate a pmos switch;no instance name**

- Value of the *out* signal is determined from the values of *data* and *control* signals.
- The symbol L stands for 0 or **z;** H stands for **1** or **z.**

Logic Tables for *nmos and pmos*

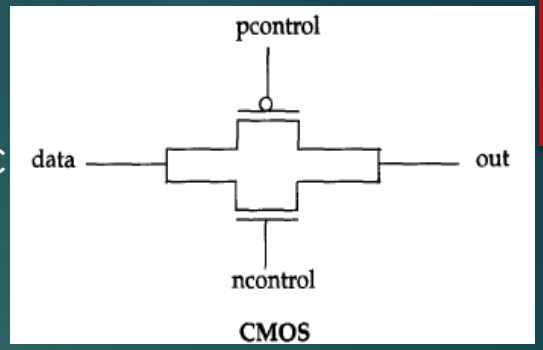
		control			
nmos		0	1	x	z
data	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	z	z	z

		control			
pmos		0	1	x	z
data	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	z	z	z	z

- The **nmos** switch conducts when its *control* signal is **1.** If *control* signal is 0,the output assumes a high impedance value.
- Similarly, a pmos switch conducts if the *control* signal is 0.

CMOS Switches

- ***CMOS*** switches are declared with the keyword **cmos.**
- **A cmos** device can be modeled with a **nmos** and a **pmos** devic



//Instantiate of cmos Switch

cmos cl(out, data, ncontrol, pcontrol);//instantiate cmos gate.

or

cmos (out, data, ncontrol, pcontrol); //no instance name given.

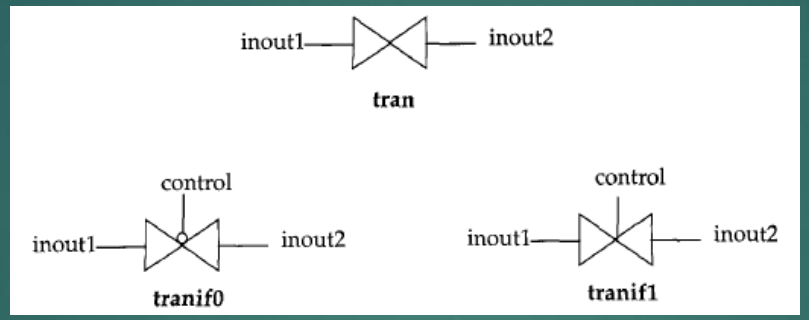
- The *ncontrol* and *pcontrol* are normally complements of each other. When the *ncontrol* signal is 1 and *pcontrol* signal is 0, the switch conducts. If *ncontrol* signal is 0 and *pcontrol* is 1, the output of the switch is high impedance value.
- The **cmos** gate is essentially a combination of two gates: one **nmos** and one **pmos.** Thus the **cmos** instantiation shown above is equivalent to the following.

nmos (out, data, ncontrol); //instantiate a nmos switch

pmos (out, data, pcontrol); //instantiate a pmos switch

Bidirectional Switches

- NMOS, PMOS and CMOS gates conduct from drain to source.
- It is important to have devices that conduct in both directions. In such cases, signals on either side of the device can be the driver signal. Bidirectional switches are provided for this purpose. Three keywords are used to define bidirectional switches: tran,tranif0, and tranif1.



- The **tran** switch acts as a buffer between the two signals *inout1* and *inout2*. Either *inout1* or *inout2* can be the driver signal.
- The *tranif0* switch connects the two signals *inout1* and *inout2* only if the *control* signal is logical **0.** If the *control* signal is a logical **1,** the nondriver signal gets a high impedance value **z.** The driver signal retains value from its driver.
- The **tranif1** switch conducts if the *control* signal is a logical **1.**

```
tran t1(inout1, inout2); //instance name t1 is optional
tranif0 (inout1, inout2, control); //instance name is not specified
tranif1 (inout1, inout2, control); //instance name is not specified
```

Power and Ground

- The power (Vdd, *logic* **1)** and Ground (Vss, *logic* **0)** sources are needed when transistor-level circuits are designed. Power and ground sources are defined with keywords **supply1** and **supply0.**
- Sources of type **supply1** are equivalent to Vdd in circuits and place a logical **1** on a net. Sources of the type **supply0** are equivalent to *ground* or Vss and place a logical **0** on a net.

```
supply1 vdd;
supply0 gnd;
assign a = vdd; //Connect a to vdd
assign b = gnd; //Connect b to gnd
```

Example1

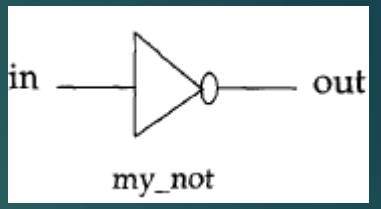
CMOS Inverter:

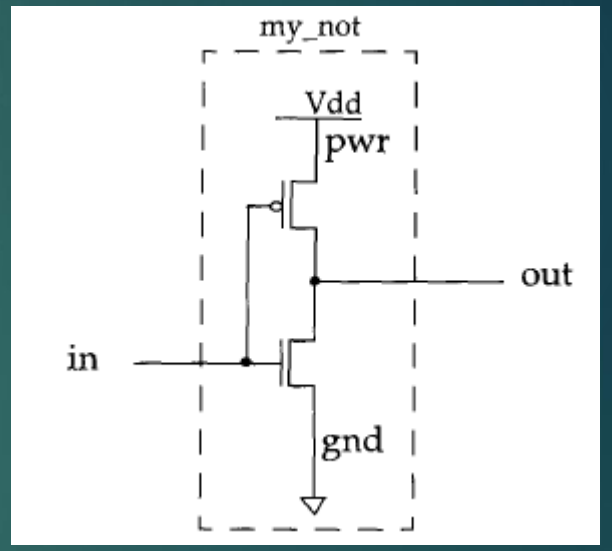
Design module

```
//Define an inverter using MOS switches
module my_not(out, in);
output out;
input in;
//declare power and ground
supply1 pwr;
supply0 gnd;
//instantiate nmos and pmos switches
pmos (out, pwr, in) ;
nmos (out, gnd, in);
endmodule
```

Testbench

```
//stimulus to test the gate
`timescale 1ns/1ps
module stimulus;
reg in;
wire out;
//instantiate the my_not module
my_not nl(out,in);
//Apply stimulus
initial
begin
//test all possible combinations
in= 1'b0;
#5 in= 1'b1;
#5 $finish;
end
endmodule
```





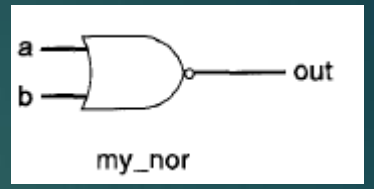
Example2

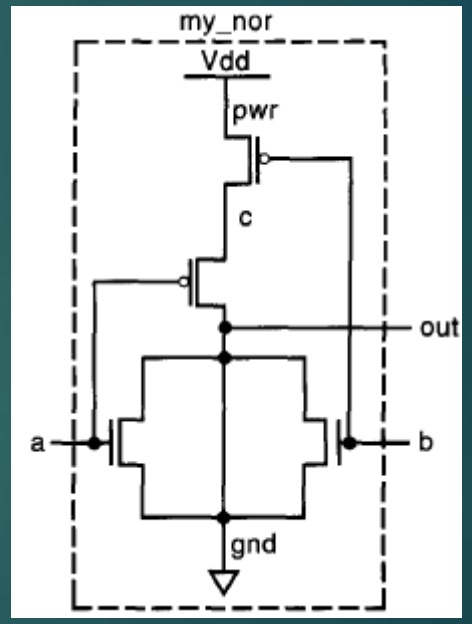
CMOS NOR Gate

```
module my_nor (out, a, b) ;
output out;
input a, b;
wire c;
supply1 pwr;
supply0 gnd ;
pmos (c, pwr, b);
pmos (out, c, a);
nmos (out, gnd, a);
nmos (out, gnd, b);
endmodule
```

Testbench:

```
`timescale 1ns/1ps
module stimulus;
reg a,b;
wire out;
my_nor nl(out,a,b);
initial
begin
a = 1'b0; b = 1'b0;
#5 a = 1'b0; b = 1'b1;
#5 a = 1'b1; b = 1'b0;
#5 a = 1'b1; b = 1'b1;
#5 $finish;
end
endmodule
```





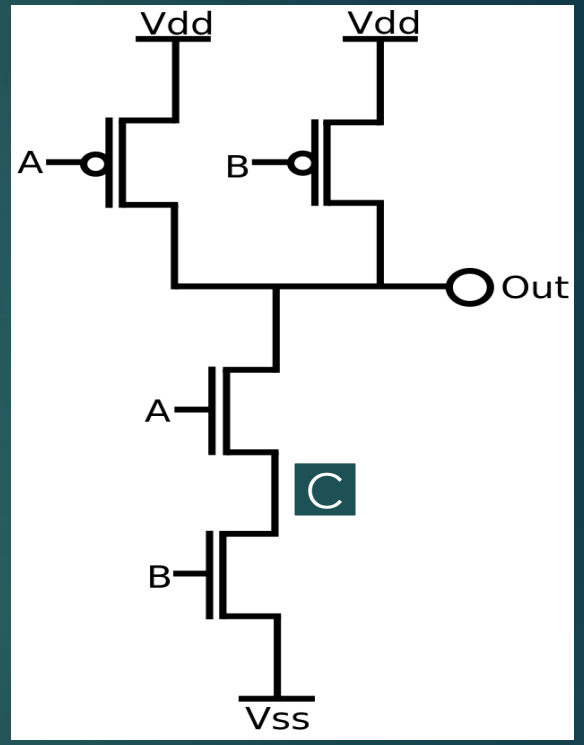
Example3

CMOS NAND Gate:

```
module my_nand (Out,A,B) ;
output Out;
input A, B;
Wire C;
supply1 Vdd;
supply0 Vss;
pmos (Out, Vdd, A);
pmos (Out, Vdd, B);
nmos (Out, C, A);
nmos (C, Vss, B);
endmodule
```

Testbench:

```
`timescale 1ns/1ps
module stimulus;
reg A,B;
wire Out;
my_nand nl(Out,A,B);
initial
begin
A = 1'b0; B = 1'b0;
#5 A = 1'b0; B = 1'b1;
#5 A = 1'b1; B = 1'b0;
#5 A = 1'b1; B = 1'b1;
#5 $finish;
end
endmodule
```



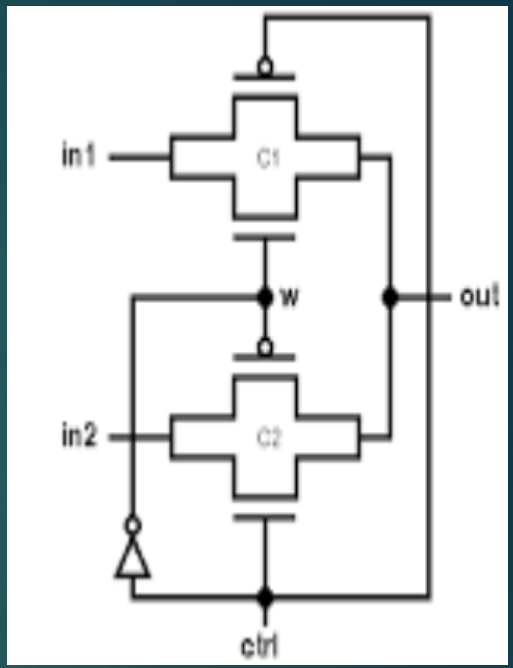
Example4

2 to 1 Multiplexer

```
module my_mux (out,ctrl,in1,in2);
output out;
input ctrl,in1,in2;
wire w;
my_not n1(w,ctrl); //Instantiating cmos not
cmos (out, in1, w, ctrl);
cmos (out, in2, ctrl, w);
endmodule
```

Testbench

```
`timescale 1ns/1ps
module mux_tb;
wire out;
reg ctrl,in1,in2;
my_mux  uut(out,ctrl,in1,in2);
initial
begin
in1=1'b0;in2=1'b1;
ctrl=1'b0;
#50 ctrl=1'b1;
#50 $finish;
end
endmodule
```



Behavioral Modelling

Behavioral Modeling:

- Verilog provides designers the ability to describe design functionality in an algorithmic manner. In other words, the designer describes the behavior of the circuit.
- Thus, behavioral modeling represents the circuit at a very high level of abstraction. Design at this level resembles C programming more than it resembles digital circuit design.

Structured Procedures:

- There are two structured procedure statements in Verilog:

  always and initial

- These statements are the two most basic statements in behavioral modeling.
- All other behavioral statements can appear only inside these structured procedure statements.
- Each always and initial statement represents a , separate activity flow in Verilog. Each activity flow starts at simulation time 0.
- The statements always and initial cannot be nested.

initial Statement

- All statements inside an initial statement constitute an initial block.
- An initial block starts at time 0, executes exactly once during a simulation, and then does not execute again.
- If there are multiple initial blocks, each block starts to execute concurrently at time 0. Each block finishes execution independently of other blocks.
- Multiple behavioral statements must be grouped,typically using the keywords begin and end. If there is only one behavioral statement, grouping is not necessary.

Example:

```
module stimulus;
reg x,y, a,b, m;
initial
m = 1'b0;
initial
begin
#5 a = 1'b1;
#25 b = 1'b1;
end
initial
begin
#I0 x= 1'b0;
#25 y = 1'b1;
end
initial
#50 $finish;
endmodule
```

Result:

time	statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
3 0	b = 1'b1;
3 5	y = 1'b1;
5 0	$finish;

- In the example, the three initial statements start to execute in parallel at time 0.
- If a delay *#<delay> is seen before a statement, the statement is executed* <delay> time units after the current simulation time.
- The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run.

always statement

- All behavioral statements inside an always statement constitute an always block.
- The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion.

Example:

```
module clock_gen;
reg clock;
// Initialize clock at time zero
initial
clock = 1'b0;
// Toggle clock every half-cycle (time period =20)
always
#10 clock = ~clock;
initial
#1000 $finish;
endmodule
```

•the always statement starts at time 0 and executes the statement *clock = ~clock every 10 time units.*
•*the initialization of clock has to be* done inside a separate **initial** statement.
• If we put the initialization of clock inside the always block, *clock will be initialized every time the always is entered.*
•Also, the simulation must be halted inside an initial statement. If there is no $stop or $finish statement to halt the simulation, the clock generator will run forever.

Procedural Assignments

- Procedural assignments update values of reg, integer or **real** variables.
- The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value.
- The left-hand side of a procedural assignment can be one of the following:

1. **A** reg, integer or real register variable
2. A bit select of these variables (e.g., addr[0])
3. A part select of these variables (e.g., addr[31:161)
4. **A** concatenation of any of the above

- The right-hand side can be any expression that evaluates to a value. In behavioral modeling all operators listed in the data-flow modelling can be used in behavioral expressions.

Continued Procedural Assignment

- There are two types of procedural assignment statements:

1. **blocking assignment(=)**
2. **Nonblocking assignment(<=)**

Blocking assignments

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

Example:

```
reg x ;
integer count;
initial
begin
x = 0;
count = 0; //Assignment to integer variables
#10 count = count + 1;
end
```

First x will be executed then count=count+1 will be executed last.

Nonblocking Assignments

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

Example:

```
reg x;
reg [15:0] reg_a;
integer count;
initial
begin
x= 0; //Scalar assignments
count = 0; //Assignment to integer variables
reg_a = 16'b0; //Initialize vectors
 reg_a[21 <=#15 1'b1; //Bit select assignment with delay
count <= count + 1; //Assignment to an integer (increment)
end
```

- In this example the statements **x**= *0* through *reg_a=16'b0* are executed sequentially at time 0.
- *Reg_a[2]* =1'b1 is scheduled to execute after 15 units
- *The non-blocking statement count = count+1* is scheduled to be executed without any delay (i.e., time =0)

Application of nonblockling assignment (or advantages over blocking assignment)

- Used to model concurrent data transfers
- Eliminate the race condition

Example1:

```
//Two concurrent always block with blocking assignment
      always @(posedge clock)
        b = a;
      always @(posedge clock)
        a = b;
```

Example2:For swapping two number

```
//Two concurrent always block with blocking assignment
      always @(posedge clock)
        b <= a;
      always @(posedge clock)
        a <= b;
```

```
always @(posedge clock)
begin
//read operation
//storevalues of right-hand-side expressions in temporary variable:
temp_a = a ;
temp_b = b;
//Write operation
//Assign values of temporary variables to left-hand-side variables
a = temp_b;
b = temp_a ;
end
```

Disadvantage:

- Higher memory usage in the simulator

Behavioral Modeling

TIMING CONTROLS

Introduction

No timing controls ⇒ No advance in simulation time

Three methods of timing control

1. delay-based
2. event-based
3. level-sensitive

1.Delay-based Timing Controls

- ***Delay*** = Duration between encountering and executing a statement
- Delay symbol: #
- Delay-based timing control can be specified by a number, identifier, or a mintypemax_expression.
- There are three types of delay control for procedural assignments:

a) regular delay control
b) intra-assignment delay control
c) zero delay control

Regular Delay Control

- Symbol: non-zero delay before a procedural assignment
- Used in most of our previous examples

Example:

```
reg x,y;
initial
begin
x= 0; // no delay control
#10 y = 1; // delay control with a number. Delay execution of y = 1 by 10 units
end
```

Intra-assignment Delay Control

- Symbol: non-zero delay to the right of the assignment operator
- Operation sequence:
  1. Compute the right-hand-side expression at the current time.
  2. Defer the assignment of the above computed value to the LHS by the specified delay.

Intra-assignment delay examples

```
//define register variables
reg x, y, z;

//intra assignment delays
initial
begin
        x = 0; z = 0;
        y = #5 x + z; //Take value of x and z at the time=0, evaluate
                      //x + z and then wait 5 time units to assign value
                     //to y.

end

//Equivalent method with temporary variables and regular delay control
initial
begin
        x = 0; z = 0;
        temp_xz = x + z;
       #5 y = temp_xz; //Take value of x + z at the current time and
                       //store it in a temporary variable. Even though x and z
                          //might change between 0 and 5,
                          //the value assigned to y at time 5 is unaffected.
end
```

Zero delay control

- Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed. This is used to eliminate race conditions.
- However, if there are multiple zero delay statements, the order between them is nondeterministic.

Example

```
initial
begin
x=1'b0;
y=1'b0;
end
initial
begin
#0 x=1'b1; //zero delay control
#0 y=1'b1;
end
```

2.Event based timing control

- Event
  - Change in the value of a register or net
  - Used to trigger execution of a statement or block (reactive behavior/reactivity)
- Types of Event-based timing control
  a. Regular event control
  b. Named event control
  c. Event OR control
  d. Level-sensitive timing control

Regular event control

- The **@** symbol is used to specify an event control.
- Statements can be executed on changes in signal value or at a *positive* or *negative* transition of the signal value.
- The keyword **posedge** is used for a positive transition and negedge is used for negative transition.

Example

always@(clock)

q = d; //q = d is executed whenever signal clock changes value

always@(posedge clock)

q = d; //q = d is executed whenever signal clock does a positive transition (0 to 1,x or z to 1)

always@(negedge clock)

Named event control

- Verilog provides the capability to *declare* an event and then *trigger* and *recognize* the occurrence of that event . The event does not hold any data.
- A named event is declared by the keyword **event.** An event is triggered by the symbol ->. The triggering of the event is recognized by the symbol @.

Example

```
//This is an example of a data buffer storing data after the
//last packet of data has arrived.

event received_data; //Define an event called received_data

always @(posedge clock) //check at each positive clock edge
begin
        if(last_data_packet) //If this is the last data packet
                ->received_data; //trigger the event received_data
end
```

Event OR control

- Sometimes a transition on any one of multiple signals or events can trigger the execution of a statement or a block of statements.
- This is expressed as an **or** of events or signals.
- The keyword **or** is used to specify multiple triggers.

Example

```
//A level-sensitive latch with asynchronous reset
always @( reset or clock or d)
                              //Wait for reset or clock or d to change
begin
        if (reset)              //if reset signal is high, set q to 0.
                q = 1'b0;
        else    if(clock)       //if clock is high, latch input
                q = d;
end
```

Level-Sensitive Timing Control

- Verilog allows level-sensitive timing control, that is, the ability to wait for a certain condition to be true before a statement or a block of statements is executed.
- The keyword **wait** is used for level-sensitive constructs.

```
always
wait (count-enable) #20 count = count + 1;
```

- In the above example, the value of *count-enable* is monitored continuously.
- If *count-enable* is **0,** the statement is not entered.
- If it is logical **1,** the statement *count= count + 1* is executed after 20 time units.
- If *count-enable* stays at 1, count will be incremented every 20 time units.

Conditional Statements

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

```
//Type 1 conditional statement. No else statement.
//Statement executes or does not execute.
if (<expression>) true_statement ;

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

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

Multiway Branching

- The nested if-else-if can become unwieldy if there are too many alternatives.
- A shortcut to achieve the same result is to use the case statement.

case Statement

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

```
case ( expression)
alternative1: statement1;
alternative2: statement2;
alternative3: statement3;
...
default: default_statement;
endcase
```

- Each of *statementl, statement2 ..., default-statement* can be a single statement or a block of multiple statements.
- **A** block of multiple statements must be grouped by keywords **begin** and **end.**
- The expression is compared to the alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is executed.
- If none of the alternatives match, the *default-statement* is executed. The *default-statement* is optional.

casex, casez Keywords

- There are two variations of the **case** statement. They are denoted by keywords, **casex** and **casez.**
- **casez** treats all **z** values in the case alternatives or the case expression as don't cares. All bit positions with **z** can also represented by ? in that position.
- **casex** treats all **x** and **z** values in the case item or the case expression as don't cares.
- Only one bit is considered to determine the next state and the other bits are ignored.

Example

```
reg [3:0] encoding;
integer state;

casex (encoding) //logic value x represents a don't care bit.
4'b1xxx : next_state = 3;
4'bx1xx : next_state = 2;
4'bxx1x : next_state = 1;
4'bxxx1 : next_state = 0;
default : next_state = 0;
endcase
```

- An input encoding = 4'b10xz would cause next_state = **3** to be executed.

Loops

- There are four types of looping statements in Verilog:

1. while
2. For
3. Repeat
4. forever

- The syntax of these loops is very similar to the syntax of loops in the C programming language.
- All looping statements can appear only inside an **initial** or **always** block.

While Loop

- The keyword **while** is used to specify this loop.
- The **while** loop executes until the while-expression becomes false.
- If the loop is entered when the while-expression is false, the loop is not executed at all.
- If multiple statements are to be executed in the loop, they must be grouped typically using keywords **begin** and **end.**

Example : Increment count from 0 to 127. Exit at count 128.

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

For Loop

- The keyword **for** is used to specify this loop.
- The **for** loop contains three parts:

1. An initial condition
2. A check to see if the terminating condition is true
3. A procedural assignment to change value of the control variable

- The initialization condition and the incrementing procedural assignment are included in the **for** loop and do not need to be specified separately. Thus, the **for** loop provides a more compact loop structure than the **while** loop.

Example:

```
integer count;
initial
for ( count=0; count < 128; count = count + 1)
$display("count = %d, count);
```

Repeat Loop

- The keyword **repeat** is used for this loop.
- The **repeat** construct executes the loop a fixed number of times.
- A **repeat** construct cannot be used to loop on a general logical expression. A **while** loop is used for that purpose.
- A **repeat** construct must contain a number, which can be a constant, a variable or a signal value.
- However, if the number is a variable or signal value, it is evaluated only when the loop starts and not during the loop execution.
- **Example:** Increment and display count from 0 to 127

```
integer count;
initial
begin
count = 0;
repeat (128)
begin
$display("Count = %d", count);
count = count + 1;
end
end
```

Forever loop

- The keyword **forever** is used to express this loop.
- The loop does not contain any expression and executes forever until the **$finish** task is encountered.
- The loop is equivalent to a **while** loop with an expression that always evaluates to true.
- A forever loop can be exited by use of the **disable** statement.
- Example: Clock generation

```
//Use forever loop instead of always block
reg clock;
initial
begin
clock = 1'b0;
forever #10 clock = ~clock; //Clock with period of 20 units
end
```

Block Types

- There are two types of blocks:

1.sequential blocks

2. parallel blocks.

Sequential blocks

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

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

Example of Sequential block

```
reg x,y;
reg [1:0]z;
initial
begin
X=1'b0; //completes at simulation time 0
#5 y =1'b1//completes at simulation time 5
#10 z = {x, y}; //completes at simulation time 15
end
```

Parallel Block

- *Parallel blocks, specified by keywords fork and join .*
- Statements in a parallel block are executed concurrently.
- Ordering of statements is controlled by the delay or event control assigned to each statement.
- If delay or event control is specified, it is relative to the time the block was entered.

Example of Parellel block

```
reg x,y;
reg [1:0]z;
initial
begin
X=1'b0; //completes at simulation time 0
#5 y =1'b1//completes at simulation time 5
#10 z = {x, y}; //completes at simulation time 10
end
```

Three special features available with block statements:

- *nested blocks*
- *named blocks*
- *disabling of named blocks.*

nested block

```
//Nested blocks
initial
begin
x= 1'b0;
fork
#5 y = 1'b1;
#I0 z = {x, y};
join
#20 W = {x,y);
end
```

Named blocks

- Blocks can be given names.
- Named blocks are a part of the design hierarchy. Variables in a named block can be accessed by using hierarchical name referencing.

Example of Named block

```
initial
begin: block1  //sequential block named block1
...
...
end
```

Disabling named blocks

- The keyword **disable** provides a way to terminate the execution of a block.
- disable can be used to get out of loops, handle error conditions, or control execution of pieces of code, based on a control signal.
- Example:

```
initial
begin: block1  //sequential block named block1
...
...
disable block1;
end
```

Tasks and Functions

- Verilog provides tasks and functions to break up large behavioral designs into smaller pieces.
- Tasks and functions allow the designer to abstract Verilog code that is used at many places in the design.
- Tasks have input, output, and inout arguments; functions have input arguments.
- In addition, they can have local variables, registers, time variables, integers, real, or events.
- Tasks or functions cannot have wires.
- Tasks and functions contain behavioral statements only. Tasks and functions do not contain always or initial statements but are called from always blocks, initial blocks, or other tasks and functions.
- Tasks are declared with the keywords **task** and **endtask**.
- Functions are declared with the keywords **function** and **endfunction.**

Differentiate between task and function

Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one **input** argument. They can have more than one **input**.	Tasks may have zero or more arguments of type **input, output or inout**.
Functions always return a single value. They cannot have **output** or **inout** arguments.	Tasks do not return with a value but can pass multiple values through **output** and **inout** arguments.

Example of Task for 4to 1 multiplexer

```
module mux4_1(y,d,s);
output reg y;
input [3:0]d;
input [1:0]s;
always@(*)
mux (y,d,s);//Calling task
task mux;//Declaration of task mux
output out;
input [3:0]a;
input [1:0]select;
begin
case(s)
2'b00:out=a[0];
2'b01:out=a[1];
2'b10:out=a[2];
2'b11:out=a[3];
default:out=0;
endcase
end
endtask
endmodule
```

Example of Function for 4to 1 multiplexer

```
module mux4_1(y,d,s);
output reg y;
input [3:0]d;
input [1:0]s;
always@(*)
y=mux (d,s);//Calling function
function mux;//Declaration of function mux
 input [3:0]a;
input [1:0]select;
begin
case(s)
2'b00:mux=a[0];
2'b01:mux=a[1];
2'b10:mux=a[2];
2'b11:mux=a[3];
default:mux=0;
endcase
end
endfunction
endmodule
```

Automatic (Re-entrant) Task

- Task are normally static in nature.
- All declared items are statically allocated and they are shared across all uses of task executing concurrently.
- If a task is called concurrently from two places in the code, these task will operate on the same variables. So the result of such operation will be incorrect.
- To avoid that problem, a keyword automatic is added in front of task keyword to make the task re-entrant. Such tasks are called automatic tasks.
- All items declared inside automatic tasks are allocated dynamically for each invocation.
- e.g: task automatic mux;

Generate Block

- It is a alternative to structural modelling.
- The statements inside generate block are executed concurrently.
- It is useful to replicate identical statements in Verilog.
- It provides a compact description of regular structures.
- There are three forms of generate statements:

1. generate for
2. generate case
3. generate conditional

Generate for

Syntax of for loop generate statement:

generate for(initial value;final value;increment)

begin:generate label

//logic of the program

end

endgenerate

Example:

```
module generateand(z,x,y);
parameter n=16;
output [n-1:0]z;
input [n-1:0]x,y;
genvar i;
generate for(i=0;i<n;i=i+1)
begin:and_loop
and(z[i],x[i],y[i]);
end
endgenerate
endmodule
```

Generate case

Syntax for generate case:

```
generate
case(n)
1://statement1
2://statement2
....
n://statement
default://default statement
endcase
endgenerate
endmodule
```

- Example:

```
module generate_case(z,x,y);
parameter n=1;
output z;
input x,y;
generate
case(n)
1:and(z,x,y);
2:or(z,x,y);
default:xor(z,x,y);
endcase
endgenerate
endmodule
```

Generate Conditional

Syntax:

```
generate
if(condition)
//statement
else
//statement
endgenerate
```

Example

```
module generateconditional(z,x,y);
parameter n=9;
parameter m=9;
parameter out=9;
output [out-1:0]z;
input [m-1:0]x;
input [n-1:0]y;
generate
if((m<8) | | (n<8))
and(z,x,y);
else
or(z,x,y);
endgenerate
endmodule
```

Verilog program for 4 to 1 Mux

```
module mux4(y,d,s);
output y;
input [3:0]d;
input [1:0]s;
reg y;
always @(*)
case (s)
2'b00:y=d[0];
2'b01:y=d[1];
2'b10:y=d[2];
2'b11:y=d[3];
default:y=0;
endcase
endmodule
```

Testbench

```
`timescale 1ns/1ps
module mux_tb;
wire y;
reg [3:0]d;
reg [1:0]s;
mux4  uut(y,d,s);
initial
begin
d[0]=1’b0;d[1]=1’b1;d[2]=1’b0;d[3]=1’b1;
s=2’b00;
#50 s=2’b01;
#50 s=2’b10;
#50 s=2’b11;
#50 $finish;
end
endmodule
```

Verilog program for 8 to 1 multiplexer

```
module mux8(y,d,s);
output reg y;
input [7:0]d;
input [2:0]s;
always @(*)
case (s)
3'b000:y=d[0];
3'b001:y=d[1];
3'b010:y=d[2];
3'b011:y=d[3];
3'b100:y=d[4];
3'b101:y=d[5];
3'b110:y=d[6];
3'b111:y=d[7];
default:y=0;
endcase
endmodule
```

Test bench

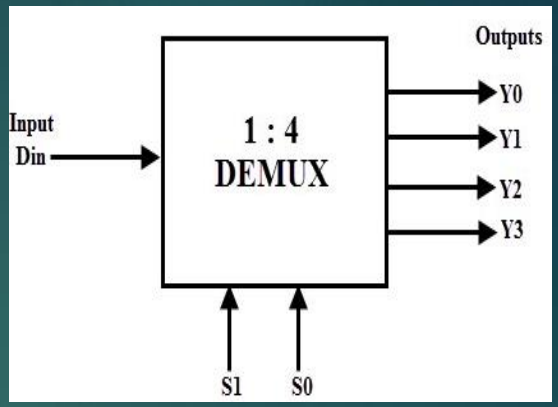
```
`timescale 1ns/1ps
module mux8_tb;
wire y;
reg [2:0]s;
reg [7:0]d;
mux8 uut (y,d,s);
initial
begin
d=8'b10101101;
s=3'b000;
#10 s=3'b001;
#10 s=3'b010;
#10 s=3'b011;
#10 s=3'b100;
#10 s=3'b101;
#10 s=3'b110;
#10 s=3'b111;
#10 $finish;
end
endmodule
```

1 to 4 demultiplexer in dataflow modelling

```
module demux(y,d,s);
output [3:0]y;
input d;
input [1:0]s;
assign y[0]=~s[1]&~s[0]&d;
assign y[1]=~s[1]&s[0]&d;
assign y[2]=s[1]&~s[0]&d;
assign y[3]=s[1]&s[0]&d;
endmodule
```

Testbench

```
`timescale 1ns/1ps
module demux_test;
wire [3:0]y;
reg d;
reg [1:0]s;
decoder4 uut(y,d,s);
Initial
begin
d=1'b1;
s=2'b00;
#50 s=2'b01;
#50 s=2'b10;
#50 s=2'b11;
#50 $finish;
end
endmodule
```



$Y0 = \overline{S1}\ \overline{S0}\ D$

$Y1 = \overline{S1}\ S0\ D$

$Y2 = S1\ \overline{S0}\ D$

$Y3 = S1\ S0\ D$

Data Input	Select Inputs		Outputs			
D	S_1	S_0	Y_3	Y_2	Y_1	Y_0
D	0	0	0	0	0	D
D	0	1	0	0	D	0
D	1	0	0	D	0	0
D	1	1	D	0	0	0

1 to 4 demultiplexer in behavioral modelling

```
module demux(y,d,s);
output reg [3:0]y;
input d;
input [1:0]s;
always @(*)
begin
y=4'b0000;
case (s)
2'b00:y[0]=d;
2'b01:y[1]=d;
2'b10:y[2]=d;
2'b11:y[3]=d;
default:y=0;
endcase
end
endmodule
```

Testbench

```
`timescale 1ns/1ps
module demux_test;
wire [3:0]y;
reg d;
reg [1:0]s;
demux uut(y,d,s);
initial
begin
d=1'b1;
s=2'b00;
#50 s=2'b01;
#50 s=2'b10;
#50 s=2'b11;
#50 $finish;
end
endmodule
```

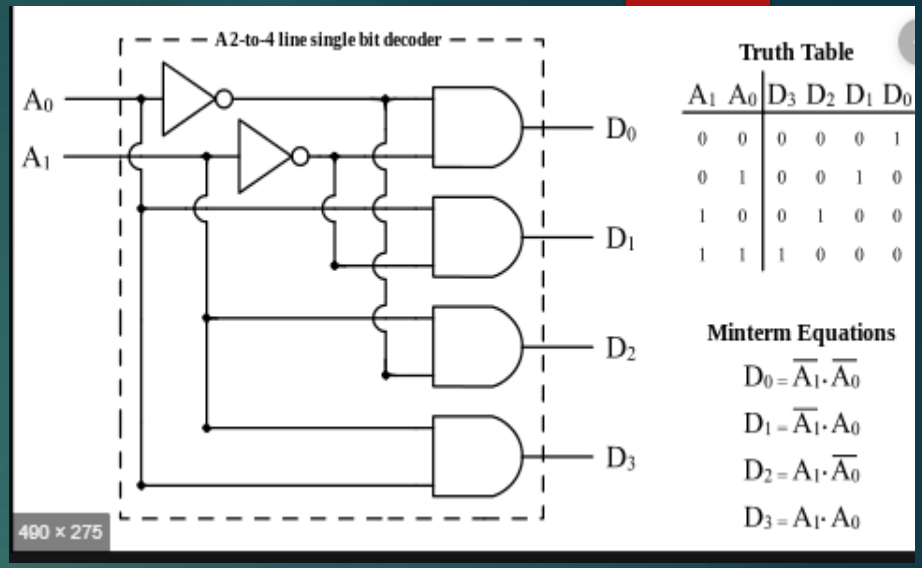
Data Input	Select Inputs		Outputs			
D	S_1	S_0	Y_3	Y_2	Y_1	Y_0
D	0	0	0	0	0	D
D	0	1	0	0	D	0
D	1	0	0	D	0	0
D	1	1	D	0	0	0

Verilog program for 2 to 4 decoder in dataflow modelling

```
module decoder4(d,a);
output [3:0]d;
input [1:0]a;
assign d[0]=~a[1]&~a[0];
assign d[1]=~a[1]&a[0];
assign d[2]=a[1]&~a[0];
assign d[3]=a[1]&a[0];
endmodule
```

Testbench

```
`timescale 1ns/1ps
module decoder4_test;
wire [3:0]d;
reg [1:0]a;
decoder4 uut(d,a);
Initial
begin
a=2'b00;
#50 a=2'b01;
#50 a=2'b10;
#50 a=2'b11;
#50 $finish;
end
endmodule
```



3 to 8 decoder in behavioral modelling

```
module decoder8(d,a);
output reg [7:0]d;
input [2:0]a;
always @(a)
begin
d=8'b00000000;
case(a)
3'b000:d[0]=1;
3'b001:d[1]=1;
3'b010:d[2]=1;
3'b011:d[3]=1;
3'b100:d[4]=1;
3'b101:d[5]=1;
3'b110:d[6]=1;
3'b111:d[7]=1;
default:d=0;
endcase
end
endmodule
```

Testbench

```
`timescale 1ns/1ps
module decoder8_tb;
wire [7:0]d;
reg [2:0]a;
decoder8 uut(d,a);
initial
begin
a=3'b000;
#50 a=3'b001;
#50 a=3'b010;
#50 a=3'b011;
#50 a=3'b100;
#50 a=3'b101;
#50 a=3'b110;
#50 a=3'b111;
#50 $finish;
end
endmodule
```

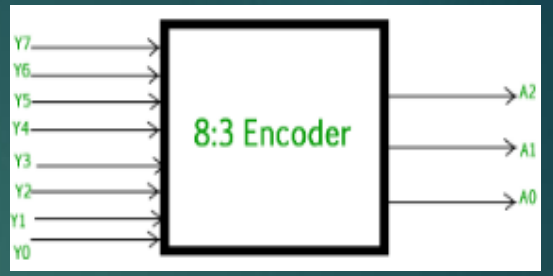
A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

8 to 3 encoder in dataflow modelling

```
module encoder8(a,y);
output reg [2:0]a;
input [7:0]y;
assign a[0]=y[7] | y[5] | y[3] | y[1];
assign a[1]=y[7] | y[6] | y[3] | y[2];
assign a[2]=y[7] | y[6] | y[5] | y[4];
endmodule
```

Testbench

```
`timescale 1ns/1ps
module encoder8_tb;
wire [2:0]a;
reg [7:0]y;
encoder8 uut(a,y);
initial
begin
y=8'b00000001;
#50 y=8'b00000010;
#50 y=8'b00000100;
#50 y=8'b00001000;
#50 y=8'b00010000;
#50 y=8'b00100000;
#50 y=8'b01000000;
#50 y=8'b10000000;
#50 $finish;
end
endmodule
```

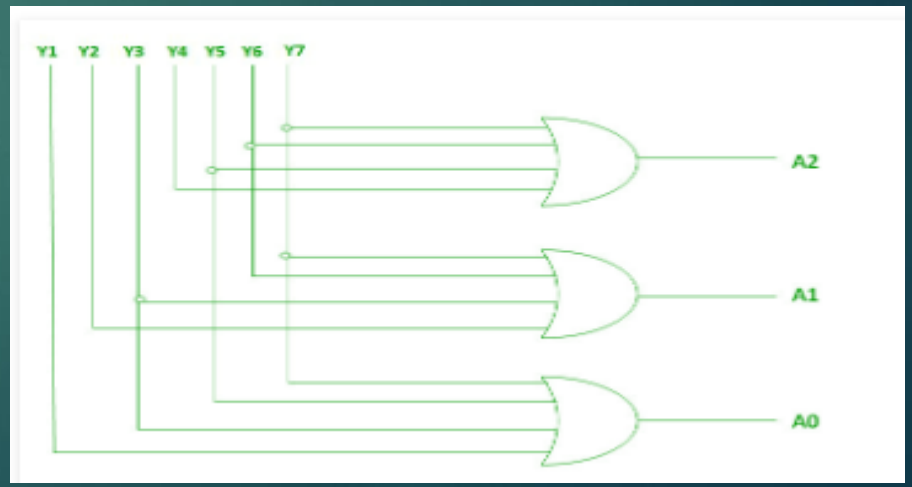


$$A2 = Y7 + Y6 + Y5 + Y4$$

$$A1 = Y7 + Y6 + Y3 + Y2$$

$$A0 = Y7 + Y5 + Y3 + Y1$$

INPUTS								OUTPUTS		
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	A2	A1	A0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1



8 to 3 encoder in behavioral modelling

```
module encoder8(out,d);
output reg [2:0]out;
input [7:0]d;
always @(d)
case(d)
8'b00000001:out=3'd0;
8'b00000010:out=3'd1;
8'b00000100:out=3'd2;
8'b00001000:out=3'd3;
8'b00010000:out=3'd4;
8'b00100000:out=3'd5;
8'b01000000:out=3'd6;
8'b10000000:out=3'd7;
default:out=0;
endcase
endmodule
```

Testbench

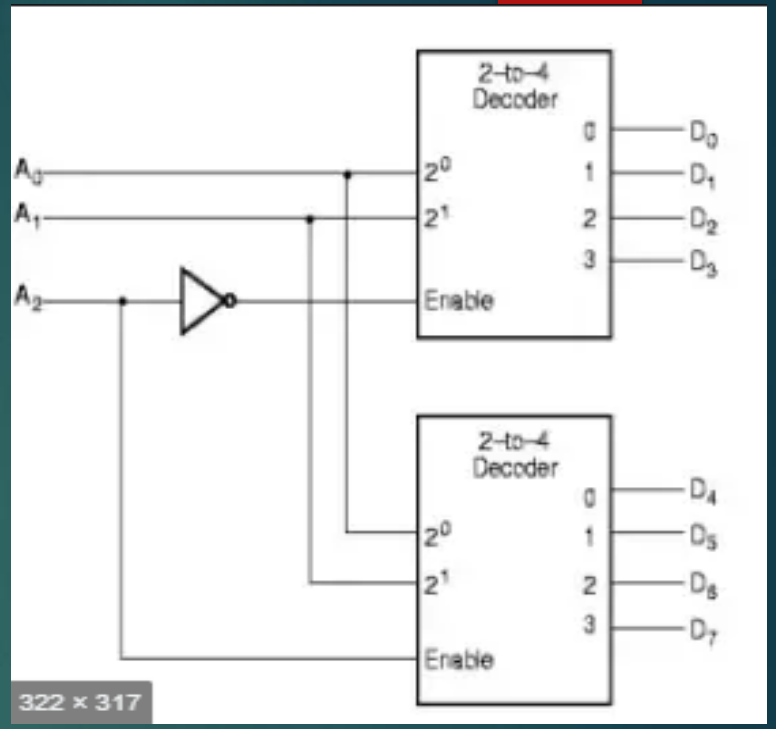
```
`timescale 1ns/1ps
module encoder8_tb;
wire [2:0]out;
reg [7:0]d;
encoder8 uut(out,d);
initial
begin
d=8'b00000001;
#50 d=8'b00000010;
#50 d=8'b00000100;
#50 d=8'b00001000;
#50 d=8'b00010000;
#50 d=8'b00100000;
#50 d=8'b01000000;
#50 d=8'b10000000;
#50 $finish;
end
endmodule
```

3 to 8 decoder using 2 to 4 decoder

```
//3 to 8 decoder
module decoder8(d,a,e);
output [7:0]d;
input [1:0]a;
input e;
wire en;
not (en,e);
decoder4 u1(d[3:0],a,en);
decoder4 u2(d[7:4],a,e);
endmodule
//2 to 4 decoder
module decoder4(d,a,e);
output [3:0]d;
input [1:0]a;
input e;
assign d[0]=~a[1]&~a[0]&e;
assign d[1]=~a[1]&a[0]&e;
assign d[2]=a[1]&~a[0]&e;
assign d[3]=a[1]&a[0]&e;
endmodule
```

Testbench

```
`timescale 1ns/1ps
module decoder8_tb;
wire [7:0]d;
reg [1:0]a;
reg e;
decoder8 uut(d,a,e);
initial
begin
a=2'b00;e=1'b0;
#50 a=2'b01;
#50 a=2'b10;
#50 a=2'b11;
#50 a=2'b00;e=1'b1;
#50 a=2'b01;
#50 a=2'b10;
#50 a=2'b11;
#50 $finish;
end
endmodule
```

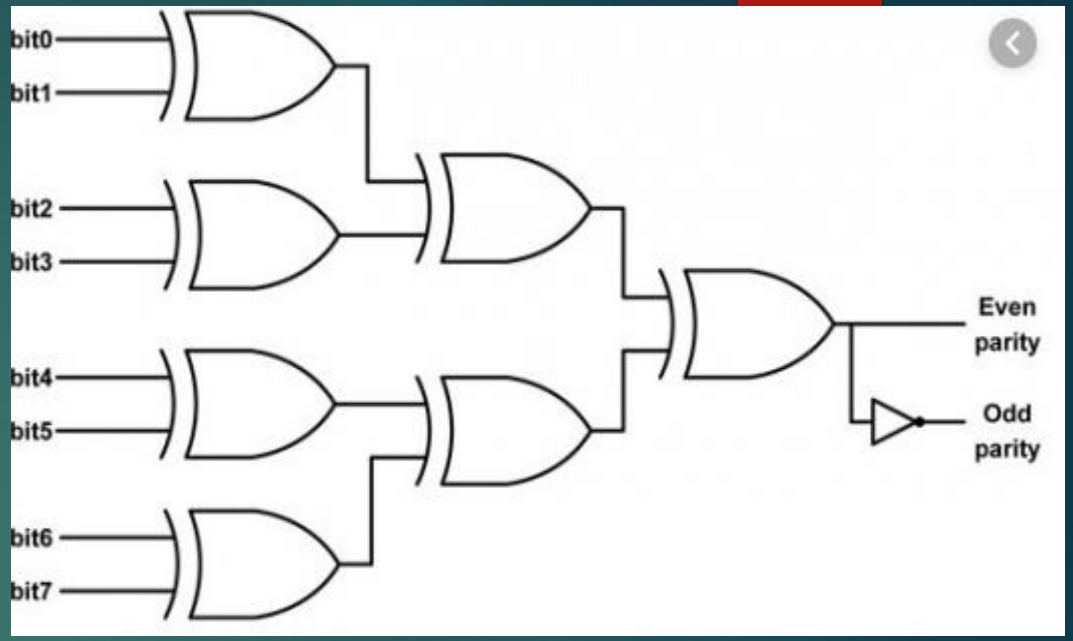


Parity Generator and Checker

```
module pgc(ep,op,bit);
output reg ep,op;
input [7:0]bit;
always @(bit)
begin
ep=^bit;
op=~ep;
end
endmodule
```

Test Bench

```
`timescale 1ns/1ps
module pgc_tb;
wire ep,op;
reg [7:0]bit;
pgc uut(ep,op,bit);
initial
begin
bit=8'b00000000;
#10 bit=8'b00000001;
#10 bit=8'b11001100;
#10 bit=8'b11001100;
#10 bit=8'b11001110;
#10 $finish;
end
endmodule
```



Verilog program for ALU(Arithmetic Logic Unit) with 8 instructions

```
module alu(f,s,a,b);
output reg [3:0]f;
input [2:0]s;
input [3:0]a,b;
always @(s,a,b)
case(s)
3'b000:f=a+b;
3'b001:f=a-b;
3'b010:f=a*b;
3'b011:f=a+1;
3'b100:f=a-1;
3'b101:f=a^b;
3'b110:f=a|b;
3'b111:f=a&b;
default:$display("Inavalid Display");
endcase
endmodule
```

Testbench

```
`timescale 1ns/1ps
module alu_tb;
wire [3:0]f;
reg [2:0]s;
reg [3:0]a,b;
alu uut(f,s,a,b);
initial
begin
s=3'b000;
a=4'b0011;b=4'b0011;
#10 s=3'b001;
#10 s=3'b010;
#10 s=3'b011;
#10 s=3'b100;
#10 s=3'b101;
#10 s=3'b110;
#10 s=3'b111;
#10 $finish;
end
endmodule
```

Bus Structure:

- A typical digital computer has many registers and paths must be provided to transfer information from one register to another.
- A more efficient scheme for transferring information between registers in a multiple register configuration is a common bus structure.
- A bus structure consists of a set of common lines. One for each bit of registers, through which binary information is transferred one at a time.
- Control signal determines which register is selected by the bus during each particular register transfer.
- One way of constructing a common bus system is with multiplexers. The multiplexer select the source register whose binary information is then placed on the bus.
- Here we are taking example of bus system of 4 registers. Each register has four bits.
- Bus consists of 4 to 1 multiplexers each having four data inputs 0 through 3 and two select lines s1 and s0.

Function Table:

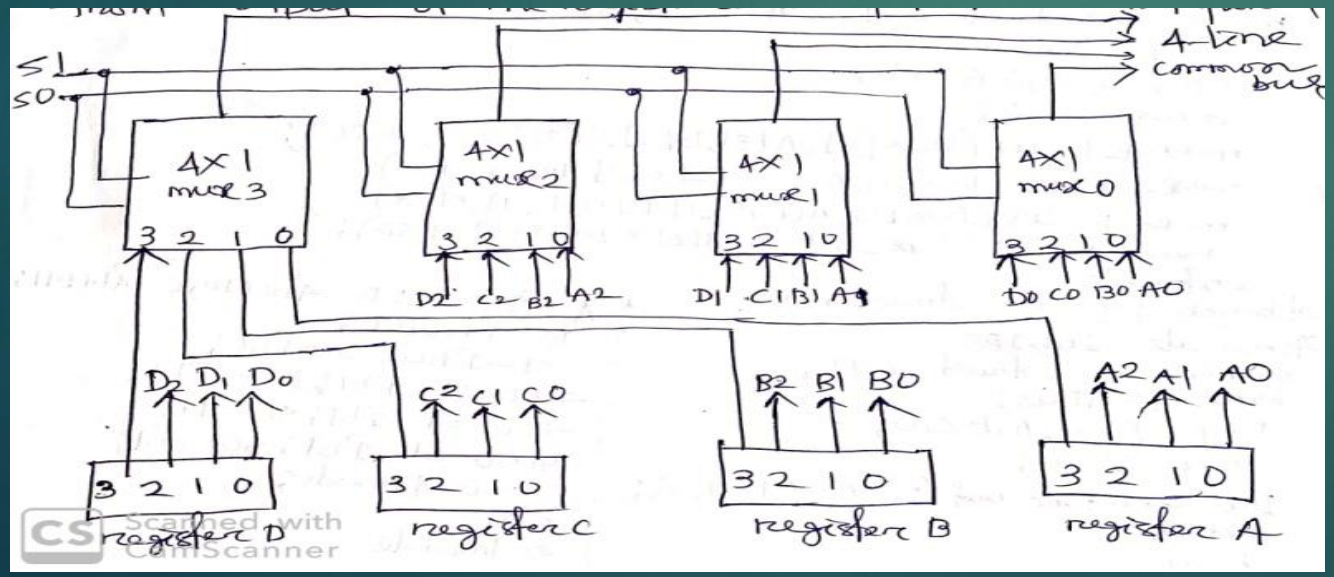
S1	S0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

- In general, a bus system will multiplex k registers of n bits each to produce an n-line common bus. The number of multiplexer are needed to construct the bus is equal to n,the number of bits in each register. The size of each multiplexer must be k*1 since it multiplies k data lines.
- Verilog program for 4 to 1 mux in dataflow modeling:

```
module mux4_1(Bus,A,B,C,D,S1,S0);
output Bus;
input A,B,C,D;
input S1,S0;
assign Bus=(A&~S1&~S0) | (A&~S1&S0) | (A&S1&~S0) | (A&S1&S0);
endmodule
```

//Verilog program for 4-bit 4 register Bus structure

```
module Bus_structure(Bus,A,B,C,D,S1,S0);
output [3:0]Bus;
input [3:0]A,B,C,D;
input S1,S0;
mux4_1 u1(Bus[3],A[3],B[3],C[3],D[3],S1,S0);
mux4_1 u2(Bus[2],A[2],B[2],C[2],D[2],S1,S0);
mux4_1 u3(Bus[1],A[1],B[1],C[1],D[1],S1,S0);
mux4_1 u4(Bus[0],A[0],B[0],C[0],D[0],S1,S0);
endmodule
```



Testbench

```
‘timescale 1ns/1ps
module Bus_structure_tb;
wire [3:0]Bus;
reg [3:0]A,B,C,D;
reg S1,S0;
Bus_structure uut(Bus,A,B,C,D,S1,S0);
initial
begin
A=4’b0001;B=4’b0010;C=4’b0011;
D=4’b0100;S1=1’b0;S0=1’b0;
#50 S1=1’b0;S0=1’b0;
#50 S1=1’b0;S0=1’b1;
#50 S1=1’b1;S0=1’b0;
#50 S1=1’b1;S0=1’b1;
#50 $finish;
end
endmodule
```

Static Timing Analysis(STA)

- The timing simulator employs timing analysis that analyzes logic in a static manner computing the delay times for each path. This is called Static Timing Analysis because it doesn't require the creation of a set of a test(or stimulus).
- Timing analysis works best with synchronous systems whose maximum operating frequency is determined by the longest path delay between the successive flip-flops.
- The path with the longest delay is the Critical Path .
- Only two kinds of Timing errors are possible in synchronous systems

1. A setup time violation , when the signal arrives too late, or misses the time when it should advance.

2. A hold time violation, when an input signal changes too soon after the clock's active transition .

- STA is capable of verifying every path, it can detect other problems like glitches , slow paths and clock skew.

The way STA is performed on a given circuit, To check a design for violations or say to perform STA there are 3 main steps:

1) Design is broken down into sets of timing paths.
2) Calculates the signal propagation delay along each path
3) And checks for violation of timing constraints.

To start STA we should know a few key concepts,

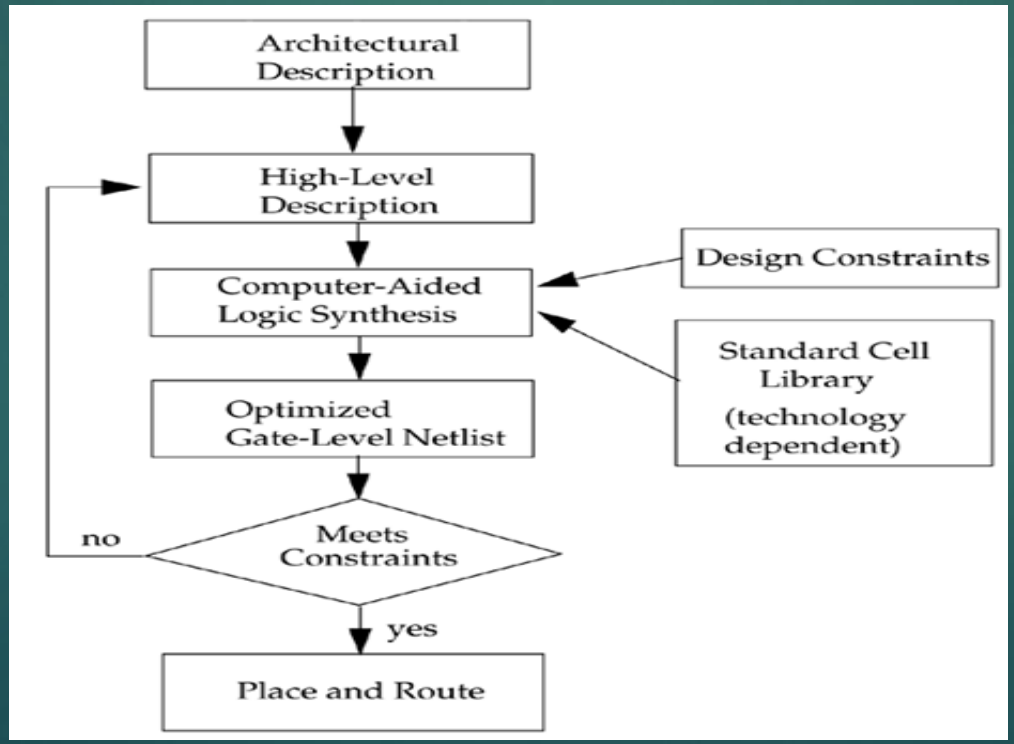
1. Timing Path
2. Arrival Time
3. Required time
4. Slack
5. Critical Path

- Timing path has a start point and an end point, it is divided into data path & clock path depending on the types of signal.
- The arrival time of a signal is the time elapsed for a signal to arrive at a certain point.
- Required time is the latest time at which a signal can arrive without making the clock cycle longer than desired.
- Slack is the difference between the required time and arrival time.
- The critical path is defined is defined as the path between an input and an output with the maximum delay.

Logic Synthesis

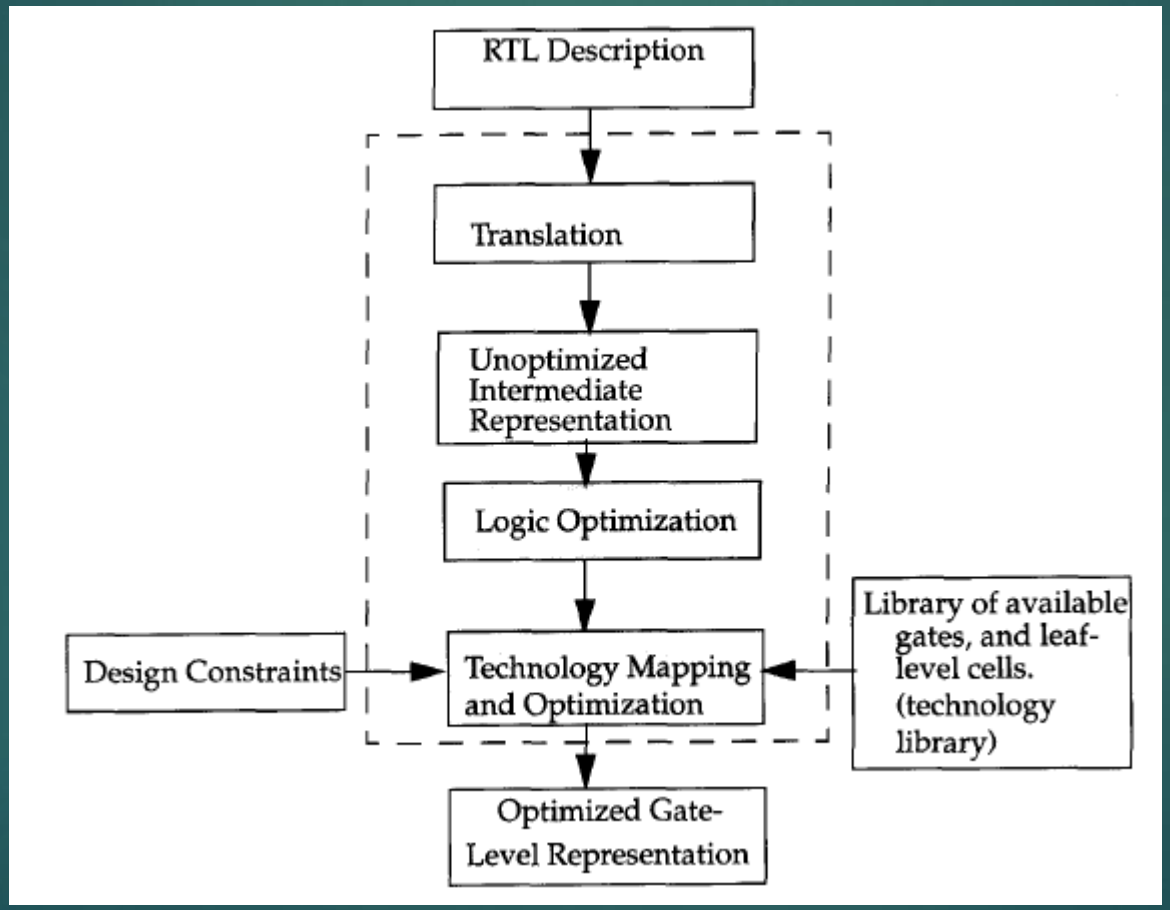
- *logic synthesis* is the process of converting a high-level description of the design into an optimized gate-level representation, given a *standard cell library* and certain design constraints.
- A standard cell library can have simple cells, such as basic logic gates like **and**, **or**, and **nor**, or macro cells, such as adders, muxes, and special flip-flops.
- A standard cell library is also known as the technology library.

Traditional Logic Design Flow



Verilog HDL Synthesis

- let us now discuss the synthesis design flow from an RTL(Register Transfer Level) description to an optimized gate-level description.



Logic Synthesis Flow from RTL to Gates

RTL description

- The designer describes the design at a high level by using RTL constructs.
- The designer spends time in functional verification to ensure that the RTL description functions correctly.
- After the functionality is verified, the RTL description is input to the logic synthesis tool.

Translation

- The RTL description is converted by the logic synthesis tool to an unoptimized, intermediate, internal representation. This process is called *translation*.
- The translator understands the basic primitives and operators in the Verilog RTL description.
- Design constraints such as area, timing, and power are not considered in the translation process.

Unoptimized intermediate representation

- The translation process yields an *unoptimized intermediate representation* of the design.
- The design is represented internally by the logic synthesis tool in terms of internal data structures.

Logic optimization

- The logic is now optimized to remove redundant logic.
- This process is called *logic optimization*. It is a very important step in logic synthesis, and it yields an *optimized internal representation* of the design.

Technology mapping and optimization

- The design description is independent of a specific *target technology*.
- In this step, the synthesis tool takes the internal representation and implements the representation in gates, using the cells provided in the technology library.
- In other words, the design is *mapped* to the desired *target technology*.

Optimized gate-level description

- After the technology mapping is complete, an optimized gate-level netlist described in terms of target technology components is produced.

Verilog Constructs

- Not all constructs can be used when writing a description for a logic synthesis tool.

Construct Type	Keyword or Description	Notes
ports	input, inout, output	
parameters	parameter	
module definition	module	
signals and variables	wire, reg, tri	Vectors are allowed
instantiation	module instances, primitive gate instances	E.g., mymux m1(out, i0, i1, s); E.g., nand (out, a, b);
functions and tasks	function, task	Timing constructs ignored
procedural	always, if, then, else, case, casex, casez	initial is not supported

Construct Type	Keyword or Description	Notes
procedural blocks	begin, end, named blocks, disable	Disabling of named blocks allowed
data flow	assign	Delay information is ignored
loops	for, while, forever,	while and forever loops must contain @(posedge clk) or @(negedge clk)

Verilog HDL Constructs for Logic Synthesis