Introduction :

Verilog HDL is a hardware description language that can be used to model a digital system at many levels of abstraction ranging from a algorithm level to the switch level.

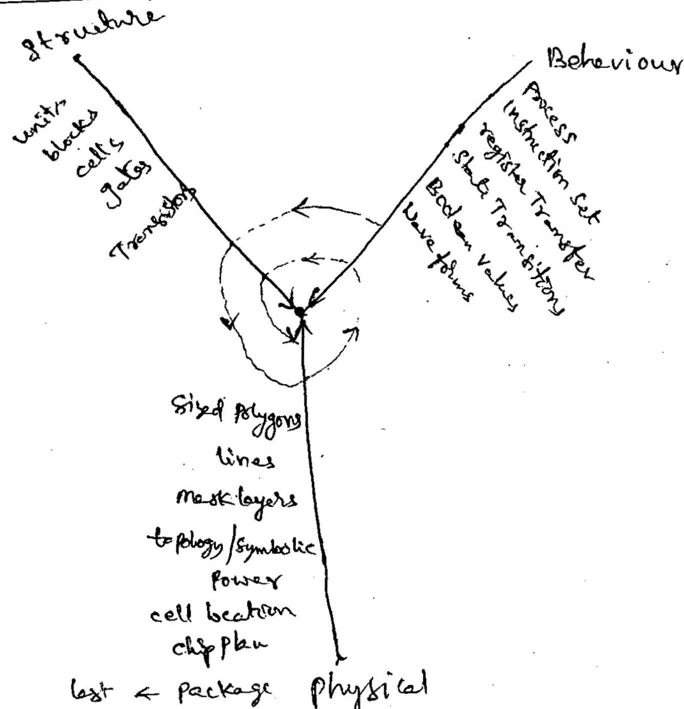
* The Verilog HDL language was first developed by Gateway Design Automation (its accuried by Cadance Design Systems) in 1983 as a hardware modelling language for their Simulator product.

* OVI → open verilog internation → IEEE standard.
  Now the standard is called IEEE Std 1364-1995.

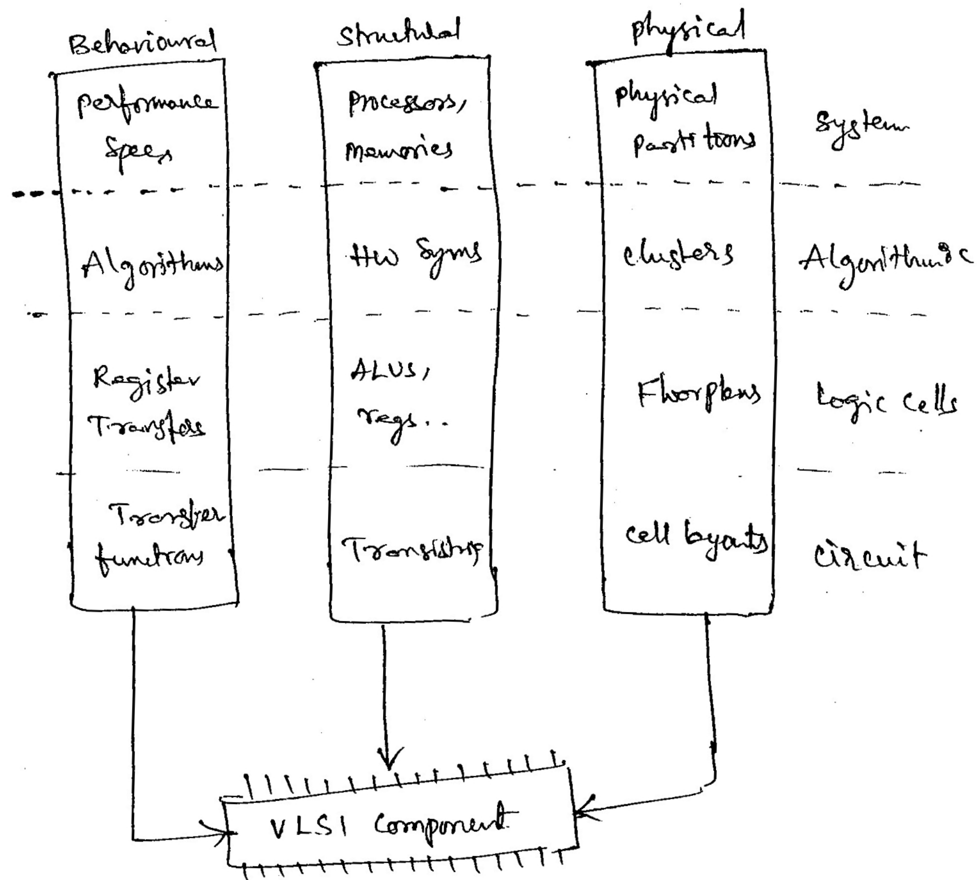
Levels of Design Description

- Algorithmic level (much-like C-code)
- RTL level (Boolean eqn).
- Gate level (with Basic gates)
- Switch level (with CMOS transistors).

GaJaski Y-Diagram



Domains of Description and Abstraction levels



System level :-

A behaviour of a system is described by a set of performance specification, which defines the required operational characteristics for the system.

* The corresponding structural description contains the components which are required to realize the system.

  Ex: processors, Memories, Controllers Buses.

* In physical domain, the physical partitions of the system are defined.

  Eg : Cabinet, rack, PCB and chip partitions.

Algorithmic level

* A behavioural description would define the process to be executed concurrently by the system – this would include the algorithm performed by each process, together with its associated data structures and procedures.

* In structural domain, HW sub syms would represent the individual process.

* The physical description would contain clusters of functionality related HW sub-syms.

Micro Architecture level (Register Transfer level)

* A behavioural description defines a set of data manipulation operations, and data transfer b/n registers – the data path – together with the ordering of the operations and transfers – the control path.

* In structural description defines the abstract implementation with a set of functional components.
  Ex: ALUs, adders, MUXs, PLAs, ROMs, and registers.
  - Seperate structural descriptions would be given for the data path and their ~~controlling~~ corresponding control path.

* The physical description would depend on the target implementations.
  Ex: gate arrays (or) standard cell.
  - It may be possible to implement the structural descriptions, (or) part of it, directly in silicon using library cells PD module generators.

Logic level

* A behavioural description would defines switching ckts, expressed interms of combinatorial logic functions, together with finite state m/c.

* A structural description would consists of a netlist of gates, flipflops, and registers.

* In the physical domain, the structural description for an ASIC would be realised directly in silicon by predefined library cells.
  - In addition, the chip floorplan – a geometrical arrangements of interconnected cells – would be derived.

Circuit level

* In behavioural domain, the behaviour of a library cells would be given in terms of its d.c and a.c electrical characteristics.

* In structural domain, transistor n/w for each cell, specific to the implementation technology, would be defined.

* The physical description would define cell layouts in terms of their physical geometry.

NOTE :- ASIC designers are not normally concerned with this level – they stop at the logic level.

Points

* The behavioural domain describes the functionality of a sym in implementation independent manner.
* The structural domain defines a n/w of abstract components that realise the specified behaviour in implementation independent manner.
* The physical domain describes the actual realisation of the sym in independent implementation manner.

Verilog HDL levels of Design Description

~~Four level~~

The HDL languages contains four levels of abstraction.

1) Behavioral (or) algorithmic level.
2) Data flow
3) Gate level
4) switch level.

① **Behavioral level:-**

* This is highest level of abstraction provided by Verilog HDL.
* A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details.
* Designing at this level is very similar to C language programming.

② **Data flow level :-**

* At this level the module is designed by specifying the data flow.
* The designers is aware of how data flows b/n HW registers and how the data is processed in the design.

③ **Gate level :-**

* The module is implemented interms of logic gates and interconnects b/n these gates.
* Design at this level is similar to describing a design in terms of a gate-level logic Diagram.

④ **Switch level :-**

* This is the lowest level of abstraction provided by verilog.
* A module can be implemented interms of switches, storage nodes, and the interconnections b/n them.
* Design at this level requires knowlege of switch level implementation details.

→ Verilog allows the designers to mix and match all four levels of abstraction in a design.

→ In the digital design community, the term RTL (register-transfer level) is frequently used for a Verilog Description that uses a combination of behavioural and dataflow constructs and is acceptable to logic synthesis tools.

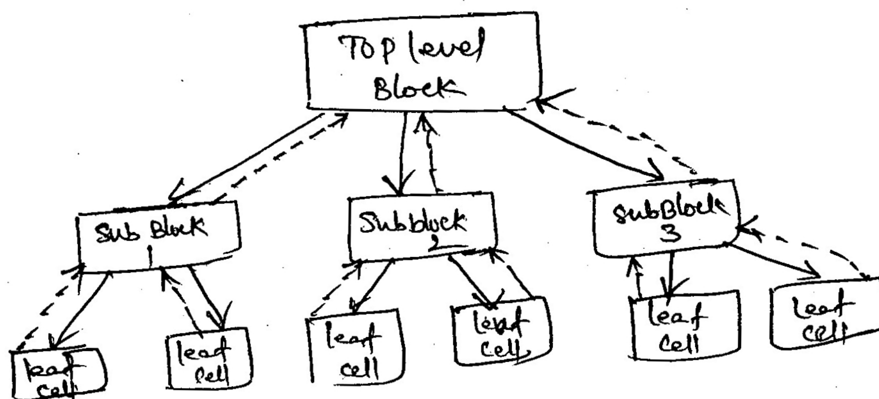
→* The Design Engineers, most modules are replaced with the gate-level implementations.

→* The higher level of abstraction, is more flexible and technology Independent the design.

→* The switch level design, becomes technology dependent and flexible.

Design Methodologies

1) top-down design
2) Bottom-up design.



"⟶" represents TOP-dow design flow.

"- - - →" represents Bottom up design flow.

Test bench

Once a design block is completed, it must be tested. The functionality of the design block can be tested by applying stimulus and checking results. We call such a block the stimulus block.

* It is good practice to keep the stimulus and design blocks separate.
* The stimulus block can be written in Verilog.
* A seperate language is not required to describe stimulus. The stimulus block is also commonly called a test bench.

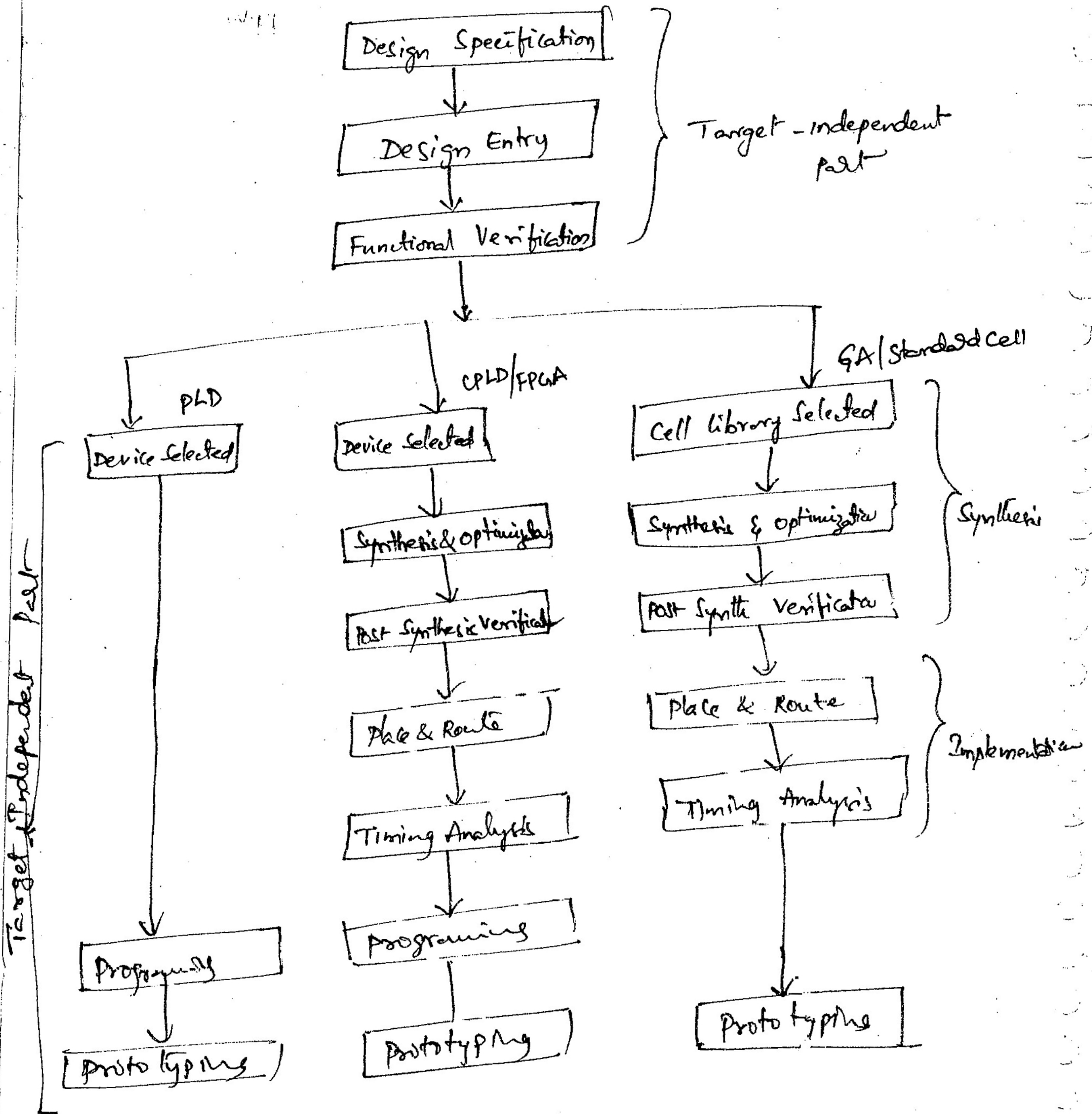
~~Structures~~

Programming Language Interface (PLI)

Verilog provides the set of standard system tasks and functions. However, designers frequently need to customize the capability of the verilog language by defining their own system tasks and functions.

* To do this, the designers need to interact with the internal representation of the design and the simulation environment in the verilog simulator.
* The PLI provides a set of interface routines to read internal data representation, write to internal data representation, and extract information about the simulation.
* User-defined system tasks and functions can be created with this predefined set of PLI interface routines.
* The PLI interface allows the user to do the following.
  - Read internal data structures
  - Modify internal data structures
  - Access simulation environment.

HDL Based Design



More Detailed ↓		More Abstract ↑
	System	
	Architectural	
	Behavioural	
	Algorithmic	
	RTL	
	Boolean Eqn	
	Structural	
	Gates	
	Switches	
	Transistors	
	Polygons	
	Masks	

Types of Simulation

1. Discrete (event driven) (Verilog)
2. Continuous (SPICE)

(I) Lexical Tokens

1) White space:

* White spaces seperate words and can contain spaces (\b) (or) Blank spaces, tabs (\t), new-lines (\n) and form feeds.
* White spaces is ignored by verilog except when it seperates tokens.
* White space is not ignored in strings.

2) Comments:

* Comments can be specified in two ways (exactly the same way as C/C++);

  i) — Begin the comment with double slashes (//). All text between these characters and the end of the line will be ignored by the compiler.

  ii) Enclose comments b/w the characters /* and */. Using this method allows you to continue comments on more than one line. This is good "commenting out" many lines code, (or) for very brief in-line comments.

Ex:

```
a = c+d // this is simple comment.

/* Design Name :
   Project Head Name :
   Company Name :
   . . . .  . .      */.
```

3) Numbers

Syntax: <size>'<base format><number>

* Number storage is defined as no. of bits, but values can be specified in binary, octal, decimal (or) Hexa decimal.

Ex:

```
4'b1111    // This is a 4-bit binary number
12'habc    //   "    "  12-bit Hexa decimal number
16'd255    //   "    "  16-bit decimal   "
10'o237    //   "    "  10-bit Octal     "
```

```
8'bx1 = xxxx_xxx1      8'b1  = 0000-0001     | 8'hx  = xxxx xxxx
8'bx0 = xxxxxxx0       8'b0  = 0000-0000     | 8'hzx = zzzz xxxx
                       8'bx  = xxxx-xxxx     | 8'hz1 = zzzz_0001
                       8'b1x = 0000-001x     | 8'hz  = zzzzzzzz
                       8'b0x = 0000-000x     |
```

Unsized numbers

* Number that are specified without a <baseformat> specification are decimal numbers by default. 8'h0z = 0000_zzzz

* Number that are written without <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

Ex

```
23456.  // This is a 32-bit decimal number by default.
'hc3    //   "       32-bit Hexadecimal number.
'o21    //   "       32-bit octal number.
```

X and Z values:

* Verilog has two symbols — 1) X — unknown
  2) Z — high impedance.

* These values are very important for modelling real ckts.

Ex:

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

6'hx // This is a 6-bit hex number
32'bz // This is 32-bit high impedance number.
```

Negative Numbers:

* Negative numbers can be specified by putting a minus sign before the size ~~for~~ a constant number.
* Size constants are always positive.
* It is illegal to have a minus sign between <base format> and <number>.

Ex:

-6'd3 // 8-bit 've' number stored as 2's complement of 3.

4'd-2 // Illegal specification.

Underscore characters and Question marks

* An underscore character "_" is allowed in a number except the first character.
* Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog.
* A question mark "?" is the Verilog HDL alternative for z in the context of numbers.
* The ? is used to enhance readability in the case x and case z statements.

Ex: 12'b1111_0000_1010 // readable.

4'b10?? // equivalent of a 4'b10zz.

4) Strings:

* A string is a sequence of characters that are enclosed by double quotes (" ").
* The restriction on a string is that it must be contained on a single line, that is, without carriage return.
* It can not be on multiple lines.
* Strings are treated as a sequence of one byte ASCII values.

Ex: "Hellow world" // is a string

"a/b" // is a string.

5) Identifiers and Keywords

* Keywords are special identifiers reserved to define the language constructs.
* Keywords are in lowercase.

  Ex: assign, case, while, wire, reg, and, or, nand and module.

* They should not be used as identifier.
* Verilog keywords also includes Compiler Directives and System Tasks.
* These key words that have special meaning in Verilog.

Identifiers:

* Identifiers are made up of alphanumeric characters, the underscore (_) and the doller sign ($) and are case sensitive.
* Identifiers start with an alphanumeric character (or) an underscore (or) with letter (not with a number (or) $)
* Identifiers are user-defined words for variables, function names, module names, block names, and instance names.

Ex:

```
reg value;  // reg -> keyword ; value -> identifier
input clk;  // input -> keyword ; clk -> identifier.
```

Note:-

"$" sign as the first character is reserved for system tasks.

Encaped Identifiers

* Its begin with backslash (\) character and end with white space (space, tab, or new line)
* All characters b/n backslash and white space processed literall
* The backslash (or) whitspace is not considered a part of the iden'

II. Data Types → Two groups < Net type / Register Type.

* The Verilog support four values and Eight strengths to model the functionality of real hardware.

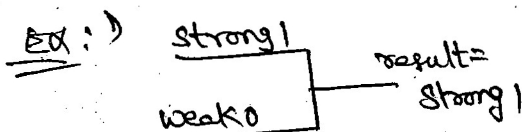
1) Value set

Value level	Condition in Hardware ckt
0	Logic zero (or) false condition
1	Logic one (or) True condition
X	Unknown
Z	High impedance, floating state.

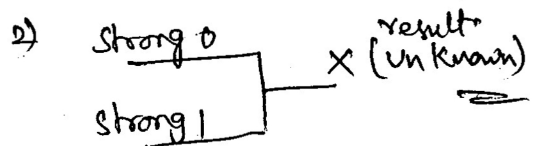
Strength level	Type	Degree.
supply	Driving	Strongest
strong	Driving	↑
pull	Driving	
large	storage	
weak	Driving	
medium	Storage	
Small	Storage	
high z	high impedance	weakest

* If two signals of unequal strengths are driven on a wire, the stronger signal prevails.
* If two signals of equal strengths are driven on a wire, the result is unknown.

Ex: 1)



2)



2) Nets : wire / assign

* Nets represent connection b/n hardware element.
* Nets are declared primarily with the keyword wire.
* Nets are one-bit value by default unless they are declared explicitly as vectors.
* The terms wire and net are often used interchangeably.
* The default value of net is Z. (except the trireg net, which defaults to 'x').
* Nets get the o/p value of their drivers. If a net has no driver, it gets the value z.

- Net data type

  - wire
  - tri
  - wor
  - trior
  - wand
  - triand
  - trireg
  - tri1
  - tri0
  - supply 0
  - supply 1

Ex

```
wire a; //declare net a.
wire b, c; // declared two wire
```

3) Registers

1) reg
2) integer
3) time
4) real
5) real time.

(see text book by J. Bhasker.)

* Registers represent data storage elements.
* Registers retain value until another value is placed onto them.
* It's declaration is reg.
* default value for a reg data type is X.
* Verilog registers do not need a clock as hardware registers do.
* Value of registers can be changed any time in a simulation by assigning a new value to the register.

Ex

```
reg a; // single 1-bit register variable.
reg [7:0] tom; // an 8-bit vector, a bank of 8 registers.
reg [5:0] b, c; // two 6-bit variables.
```

(4) Vectors & scalar

* Nets (or) reg data types can be declared as vectors (multiple bit width).
* If bit width is not specified, the default is scalar (1-bit).

Ex:

```
wire a; // scalar
wire [7:0] bus; // 8-bit bus
wire [31:0] busA, busB, busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default.
```

→ busA[7] // bit #7 of vector busA.

(5) Integer, Real, and Time Register Data Types

* Integer, real and time register data types are supported in Verilog.

Integer:

* Keyword : integer
* Although it is possible to use reg as general purpose variable, it is more convenient to declare an integer variable for purposes such as counting.
* The default :- the host m/c word size.
* reg type store values as unsigned quantities
* integer values store as signed quantities

Ex:

Integer count; // general purpose variable used as a count.
initial
    count = -1; // A negative one is stored in the counter.

Real:

* Key word : real
* They can be specified in decimal notation (or) scientific notation.

  Ex: decimal : 3.14
      scientific : 3e6 ⇒ 3×10^6

* Real numbers can not have a range declaration.
* Default value is '0'.
* When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

Time:

* Verilog simulation is done with respect to simulation time.
* A special time register data type is used in verilog to store simulation time.
* Key word : time
* Width for time register data types is implementation specific but is at least 64 bits.
* The system function $time is invoked to get the current simulation time.
* Simulation time is measured in terms of simulation seconds (units 'S')

Ex: time save_sim_time; // Define a time variable save_sim_time
    initial
        save_sim_time = $time; // save the current simulation time.

6). Arrays

* Arrays are allowed in Verilog for reg, integer, time and vector register data type.
* Arrays are not allowed for real variables.
* Multi Dimensional arrays are not permitted in Verilog.
* It is important not to confuse arrays with net (or) register vectors.

Ex:
```
integer count [7:0] ; // An array 8 count variables.
reg bool [31:0] ; // Array of 32 one-bit boolean register variables.
time chk_point [1:100] ; // Array of 100 chk_point variables.
count [5] // 5th element of array of count variables.
chk_point [100] // 100th time chk point value.
```

→ `integer matrix [4:0] [4:0] ; // Illegal declaration.`

7) Memories

* register files, RAMs and ROMs often used.
* Memories are modelled in Verilog simply as an array of registers.
* Each element of the array is known as a word.
* Each word can be one (or) more bits.
* A particular memory is obtained by using the address as a memory array subscript.

Ex:
```
reg mem1bit [0 : 1023] ; // memory mem1bit with 1K 1-bit words.
reg [7:0] membyte [0:1023]; // Memory membyte with 1K 8-bit words (bytes)
```

→ `membyte [511] // fetches 1 byte word whose address is 511.`

8) Parameters :

* Verilog allows constant to be defined in a module by the keyword Parameter.
* Parameter cannot be used as Variables
* Parameter values for each module instance can be override individually at compile time.
* This allows the module instances to be customized.
* Parameter can be changed at modules instantiation (or) by using the defparam statement.

Ex: Parameter Port_id = 5; // Defines constant port_id.

9) Strings

* strings can be stored in reg.
* The widths of the register variables must be large enough to hold the string.
* Each character in the string takes up 8 bits (1 byte).
* If the width of the register is greater than the size of the string, Verilog fills bits to the left of the string with zeros.
* If the width is smaller than the string width, Verilog truncate the left most bits of the string.

```
reg [8*18 :1] string_value; // Declare a variable that is 18 bytes wide.
initial
    string_value = " Hellow verilog world"; // string can be store in variable.
```

Escaped characters	characters Displayed
\n	newline
\t	tab
%%	%
\\	\
\"	"
\ooo	characters written in 1-3 digits.

Operators

* Operators in Verilog HDL are classified into the following categories.

S.No	Operator Type	Operator Symbol	Operation Performed	Number of Operands
1)	Arithmatic	*	Multiply	TWO
		/	Divide	"
		+	Add	"
		−	Minus (substract).	"
		%	Modulus	"
2)	Logical (T|F)	!	logical Negation	one
		&&	logical AND	TWO
		||	logical OR	TWO
3)	Relational (T|F)	>	greater than	TWO
		<	less than	"
		>=	greater than or equal	"
		<=	less than or equal	"
4)	Equivality (T|F)	==	equivality	TWO
		!=	inequivality	"
		===	Case equality	"
		!==	Case inequality	"
5)	Bitwise operator/ (T/F)	~	bitwise negation	one
		&	bitwise AND	two
		|	bitwise OR	two
		^	bitwise EX-OR	"
		^~ (or) ~^	bitwise XNOR	"
6)	Reduction	&	reduction AND	one
		~&	reduction NAND	"
		|	reduction OR	"
		~|	reduction NOR	"
		^	reduction EX-OR	"
		~^ (or) ^~	reduction XNOR	"

7)	Shift	>> <<	Right shift Left shift	Two Two
8)	Conditional (T/F)	? :	Conditional (Ternary operator)	three
9)	Concatenation	{ }	Concatenation	any number
	Replication (both T/F)	{ { } }	Replication	any number

* All operators associate left to right except for the conditional operator that associate right to left.

	Expression	Evaluated	Comment
1)	A + B − C ⇒	(A+B) − C	// Left to right
2)	A ? B : C ? D : F ⇒	A ? B : (C ? D : F)	// Right to Left.

If we use parenthesis, we can change the order of precedence

⇒ (A ? B : C) ? D : F.

* If any bit of an operand in arithmetic is an x (or) z, the result is an x.

Ex : 'b10x1 + 'b01111 = 'bxxxxx

Verilog operators & their Precedence.

Operator Symbol	function	Precedence
+ − ! ~ (unary)	Sign, complement	↓
* / %	MUL, DIV, modulus	
+ − (binary)	Add, Sub	
<<, >>	Shift	
<, <=, >, >=	Relational	
==, !=, ===, !==	Reduction	
	~~Logical~~ Conditional	

4) Equivality operators

== (or) != (Equivality)	=== (or) !== (Identity)
* These operators can be used in a synthesizable code	* Can not be used synthesizable code.
* If either of the operands have x or z value, the result is unknown	* The operands will be compared, even if they have x and z values in the bits.
* If any of the operators is x or z, the logical result of comparision is always FALSE.	* The x and z bits will be used in comparision, and the logical result will be a TRUE, (or) FALSE, based on actual comparision.
* Since the operands contain x and z, the result will be an x, Hence, the comparision can be non-deterministic	* Since x and z are also used in comparision, the result of comparision will be Boolean 1 or 0. Hence the comparision can be deterministic.

* Ex:
 if (a==b)
 out1 = a & b ;
 else
 out1 = a | b ;
	* Ex:
	 if (a===b)
	 out1 = a & b ;
	 else
	 out1 = a | b ;

If either a or b becomes x or z, the else clause will be executed and out1 will be driven by OR gate	* If a and b are identical even if they becomes x or z, the if clause will be executed and out1 will be driven by AND gate

1) <u>Arithmatic Operators</u> (*, /, +, - and %)

* Binary operators take two operands.

* If any operand bit has a value x, then the result of the entire expression is x.

2) <u>Logical operators</u> (&&, ||, !) ⇒ T/F

* Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), (or) x (ambiguous).

* If an operand is not equal to zero, it is equivalent to a logical 1 (true condition).

* If an operand is equal to zero, it is equivalent to a logical 0 (False condition).

* If any operand bit is x or z, it is equivalent to x (ambigoues condition) and normally treated by simulators as a false condition.

* Logical operators take variables (or) expressions as operands.

3) <u>Relational operators</u> (>, <, >=, <=) ⇒ T/F

* If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false.

* If there are any unknown or z in the operands, the expression ~~is false~~ takes a value x.

5) Bit wise Operators (~, &, |, ^) ⇒ T/F

* Bitwise operators perform a bit-by-bit operation on two operands.
* They take each bit in one operand and perform the operation with the corresponding bit in the other operand.
* If one operand is shorter than the other, it will be bit extended with zeros to match the length of the longer operand.
* The exception is the unary negation operator (~), which takes only one operand and operates on the bits of single operand.

(bitwise 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

bitwise 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

bitwise EXOR (^)	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

bitwise 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

bitwise negation (~)	0	1	X	Z
	1	0	X	X

8374043583

6) <u>Reduction operator</u> (&, ~&, |, ~|, ^, ~^ (or ^~))

* Reduction operator takes only one operand.
* Reduction operators perform a bitwise operation on a single vector operator and yield a 1-bit result.
* logic tables same as before bitwise operators.
* The difference is that bitwise operation are on bits from two different operands, whereas reduction operators work bit by bit from right to left.
* Reduction NAND, reduction NOR, and reduction XNOR are computed by inverting the result of the reduction AND, reduction OR and reduction XOR respectively.

7) <u>Shift operators</u> (>>, <<)

* These operators shift a vector operand to the right or the left by a specified no. of bits.
* The operands are the vector and the no. of bits to shift. When the bits are shifted, the vacant bit positions are filled with zeros.
* Shift operations do not wrap around.
* If the right operand evaluates to an X or a Z, the result of the shift operation is an X.

<u>Application</u>: Shift-and-add operation (algorithms) for multiplication.

8) <u>Conditional operator</u> (cond-exp ? exp1 : exp2)

* First cond-exp is evaluated.
  - If result is True (logic 1) then exp1 is evaluated.
  - If " False (logic 0) " exp2 "
  - If " ambiguous (logic X) " both (exp1 AND exp2) is evaluated.
  - If " Z then both (exp1 AND exp2) is evaluated.

(bitwise operation AND)

for logic x and logic z

⇓

result = exp1 AND exp2; // evaluates bitwise AND.

If exp1 AND exp2 = 0 AND 0 ⇒ 0

exp1 AND exp2 = 1 AND 1 ⇒ 1

exp1 AND exp2 = 1 AND X ⇒ X

= Z AND 1 ⇒ X

= 0 AND X ⇒ 0

= 0 AND Z ⇒ 0

9) i) <u>Concatenation Operator</u> ({ , })

* It provides a mechanism to append multiple operands.
* The operands must be sized.
* Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.
* Concatnations are expressed as operands within braces, with commas seperating the operands.
* Operands can be scalar nets (or) registers, vector nets or registers bit-select, part-select, or sized constands.

Ex: A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110.

Y = {B, C} ⇒ Y = 4'b0010

Y = {A, B, C, D, 3'b001} ⇒ Y = 11'b10010110001

Y = {A, B[0], C[1]} ⇒ Y = 3'b101

ii) Replication operator

* Repetitive concatenation of the same number can be expressed by using a replication constant.

* A replication constant specifies how many times to replicate the number inside the brackets ({ }).

Ex: A = 4'b1011

Y = {3{4'b1011}}; ⇒ Y = 12'b1011_1011_1011

Prt → bit vector net
Bbq → scalar net.

~~Miscellanies~~ Miscellaneous

1) Bit select

* A bit select extracts a particular bit from a vector.

Ex: State[1] && State[4] // Register bit-select
Prt[0] | Bbq // Net bit-select.

* If the select expression evaluates to an X or Z (or) if it is out of bounds, the value of the bit-select is an X.

Ex: State[X] is an X.

2) Part Select

* In a part-select, a contiguous sequence of bits of vector is selected. where the range must be constant expressions.

Ex State [1:4] // Register part-select.
Prt [1:3] // Net part-select.

* If either of the range index is out of bounds (or) evaluates to an X (or) Z, the part-select value is X.

3) Memory Element

* A memory element select one word of a memory.

Ex
```
reg [1:8] A ;
reg [1:8] Ack, Dram [0:63] ;
. . . .
. . . .
Ack = Dram[60] ; // 60th element of memory.
```

* No part-select (or) bit select of a memory is allowed.

Ex:
```
Dram [60] [2]    is not allowed.
Dram [60] [2:4]   "     "    .
```

* One approach to read a bit-select (or) Part-select of a word in memory is to assign the memory element to a register and then use a part-select (or) a bit-select of this register.

Ex
```
Ack [2] and Ack [2:4] ; // Legal expressions.
```

wire — wire net

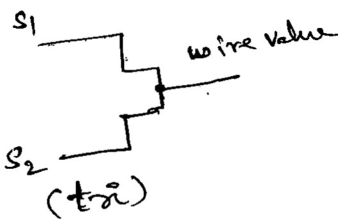
S2 \ S1	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

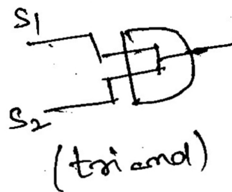
wand — wire and

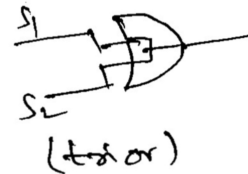
S2 \ S1	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

wor — wired or

S2 \ S1	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z







trio (tri1)	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	0 (1)

Unary Operators

* Unary operators do an operation on a single operand and assign the the result to the specified net.
* All unary operators get precedence over binary and ternary operators.
* The operators "+" and "-" preceding an integer or real number change its sign.

Operator Type	Symbol	Remarks
Logical negation	!	only for scalars
Bitwise negation	~	For scalars and vectors
Reduction AND	&	For vectors - yields a single bit o/p
Reduction NAND	~&	
Reduction OR	|	
Reduction NOR	~|	
Reduction XOR	^	
Reduction XNOR	~^ (or ^~)	

Binary operators

* Most operators available are of the binary type.
* A binary operator takes on two operands; the operator comes in b/w the two operands in the assignment.
* The binary operators are grouped into type categories and discussed separately.
  - The arithmatic operators treat both the operands as numbers and return the result as a number.
  - All net and reg operands values are treated as unsigned numbers.
  - Real and integer operands may be signed quantities.
  - If either of the operand values has a zero value, the entire result has to zero value (?).

NOTES

I) Number representation.

① 33 (or) 'd33 = default 32-bit ie, 0000 0000 0000 0000
0000 0000 0010 0001

② 9'd439
9'd4_39
9'D439
} all are same syntax.

d (or) D
b (or) B
h (or) H
x (or) X
z (or) Z
o (or) O

③ 9'b1_1011_1x01 = 9'b11011 1x01

④ 9'o1X3 ⇒ 001 xxx 011
9'o12z ⇒ 001 010 zzz

⑤ 9'hza ⇒ zzzzz1010

⑥ 5'hza ⇒ z1010

⑦ 5'h?a ⇒ z1010

⑥ -5'h1a ⇒ 11010

00101
+1
00110 = (6)

Ans: (-6)

0111
1000
+1
1001
(9)

⑦ -4'd7 = -(16-7) = -9.

(or)

0111
1000
+1
1001 = (9)

So, sign -ve
Ans: -9.

Examples

1) 'b10x1 + 'b01111 = 'bxxxxx

2) -7 % 2 = -1 // takes sign of the first operand

3) 7 % -2 = +1 // " "

4) An unsigned values stored in

- net
- reg register
- an integer in base format.

An signed value is stored in

- an integer register
- an integer decimal.

4'shf = -1
8'sb1111_1111 = -1
8'sb0111_1111 = 127
-4'sb1111 = 1 = -(-1)

Integer I
I = -12/3 = -4
I = -'d12/3 = 1431655761
I = -'sd12/3 = -4
I = -4'sd12/3 = -4

5) -'d10/5

= (2's compliment of 10)/5

= $(2^{32} - 10)/5$ // default 32 bit machine width

=

-12/3 = -4 (N)
= 1431655761

6) reg [0:5] Bar;

integer Tab;

case ①
Bar = -4'd12; // Reg Bar has the decimal value 52 (110100)

Tab = -4'd12; // Integer Tab has the value -12.

case ②
Bar = -4'd12/4; // = -3 (2's comp. value = 111101)
Bar is 6 bit.
Tab = -4'd12/4; // = 1073741821 (2's comp of '-3')
32-bit default.

Case ③

Bar = 4-6 // (= 2's complement of -2) = 62 result

Tab = 4-6 // result = -2 (Take 6'bit 111110)

Bar gets = 62

Tab gets = -2.

Case ④

Bar = -2 + (-4); // 58 = 111010 (bit vector).

Tab = -2 + (-4); // -6 = 111010 (bit vector).

Bar gets = 58

Tab gets = -6

7) Logical operators (&&, ||, !) ⇒ T|F

i) A = 4'b100 B = 4'b 0001	A = 4'b 1001 B = 4'b0000	A = 'b0110 B = 'b 0100
A && B = 1'b1	A&&B = 1'b0	A&&B = 1'b1
ii) A || B = 1'b1	A||B = 1'b1	A||B = 1'b1
iii) !A = !(4'b1001) = 1'b0	!A = 1'b0	!A = 1'b0
!B = 1'b0	!B = 1'b1	!B = 1'b0

Expression: (a==2)&&(b==3) // Both are true so, result is true.

8) Relational Operators (>, <, >=, <=) ⇒ T|F

A = 4, B = 3, X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A<=B // False (0)

A>B // True (1)

Y>=X // True (1)

Y<Z // unknown (x).

9) <u>Equivality operators</u> ⇒ T/F (==, !=, ===, !==)

i) Equivalent (==)

(4'b1001) == (4'b0001) ⇒ 1'b0

(4'b1001) == (4'b1001) ⇒ 1'b1

(4'b1001) == (4'b000x) ⇒ 1'bx

(4'b1001) == (4'b000z) ⇒ 1'bx

ii) Non equivalent (!=)

(4'b1001) != (4'b0001) ⇒ 1'b0

(4'b1001) != (4'b1001) ⇒ 1'b1

(4'b1001) != (4'b000x) ⇒ 1'bx

(4'b1001) != (4'b000z) ⇒ 1'bx

(iii) Identical (===)

(4'b1001) === (4'b0001) ⇒ 1'b0

(4'b1z01) === (4'b1z01) ⇒ 1'b1

(4'b1001) === (4'b000x) ⇒ 1'b0

(4'b100x) === (4'b000x) ⇒ 1'b1

10) <u>Bitwise operators</u> ⇒ T/F

x = 4'b1010, Y = 4'b1101, z = 4'b10x1

i) ~x = 4'b0101

ii) x & Y = 4'b1000

iii) x | Y = 4'b1111

iv) x ^ y = 4'b01111

v) x ^~ y = 4'b1000

vi) x & z = 4'b10x0

NOTE.

x | Y = 4'b1010 (bitwise)

x || Y = 1 || 1 = 1 (logical)

(T || T) = T

11) Reduction Operators

x = 4'b1010

&x = 1&0&1&0 = 1'b0 (= reduction and)

|x = 1|0|1|0 = 1'b1 (= reduction OR)

^x = 1^0^1^0 = 1'b0 (= reduction XOR).

~&x = 1'b1

~|x = 1'b0

~^x = ^~x = 1'b1

12) Shift operators

0101

x = 4'b1010

y = x>>1; ⇐ y value = 0101

y = x<<1; ⇐ y value = 0100

y = x<<2; ⇐ y value = 1000

13) Conditional operator (? :)

out = s ? a : b;

out = a if s = 1'b1;

out = b if s = 1'b0;

14) i) Concatenation operator { }

A = 2'b10; B = 3'b101

y = {A, B} ⇒ y = 5'b10101

ii) Replication operator { { } }

```
reg A;
reg [1:0] B, C;
reg [2:0] D;
A = 1'b1; B = 2'b00 ; C = 2'b10; D = 3'b110;
```

i) Y = { 4{A} } ⇒ Y = 4'b1111

ii) Y = {4{A}, 2{B} } ⇒ Y = 8'b1111_0000
= 8'b11110000

iii) Y = { 4{A}, 2{B}, C} ⇒ Y = 10'b1111000010

NOTE EX: Parameter n = 5, m = 5;

assign x = {(n-m) {a} }

= { 0{a} } ⇐ // Synthesis is not supported.

Truth Tables

1) a == b

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

2) a === b

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

3) a != b

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

4) a !== b

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

5) $a < b$

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

(6) $a <= b$

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

(7) $a > b$

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

6) $a >= b$

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

Modules and Ports

* A module definition always begins with the keyword <u>module</u>.
* The <u>endmodule</u> statement must always come last in a module definition.
* Verilog allows multiple modules to be defined in a single file. The module can be defined in any order in the file.

<u>Syntax</u>:

```
module module_name (Port_list);
    Declarations:
        reg, wire, Parameter,
        input, output, inout,
        function, task, ....

    Statements:
        Initial statement
        Always statement
        Module instantiation
        Gate instantiation
        UDP instantiation
        Continuous assignment
endmodule
```

* Declarations are used to define the various items, such as registers and parameters, used with in the module.
* Statements are used to define the functionality (or) structure of the design.
* Declaration and statements can be ~~interspersed~~ interspersed with in a module.

Design styles

1) Dataflow style.
2) Behavioral style
3) Structural style
4) Any mix of above.

Module instantiations

* Module instantiation statement can be written with in a module declaration
* Module declarations are templates from which one creates actual objects (instantiations).
* The instantiated module's ports must be matched to those defined in template.
  i) by name, using a dot (.) ".template_port_name(name_of_wire_connected_to_port)". (or)
  ii) by position, placing the ports in exactly the same positions in the portlists of both the template and the instance.

Syntax:

```
module_name
    instance_name_1 (port_connections_list),
    instance_name_2 (port_connections_list),
    . . . .
    instance_name_n (port_connections_list);
```

Ex

```
Module and4 (a, b, c)
  input [3:0] a, b;
  output [3:0] c;
  assign c = a & b;
endmodule
```

```
// Instantiations.
wire [3:0] in1, in2;
wire [3:0] o1, o2;
and4 c1 (in1, in2, o1)
and4 c2 (.c(o2), .a(in1), .b(in2));
```

What are the different approaches of connecting ports in a hierarchical design? What are the pros and cons?

Ans: While instantiating the submodules in a given hierarchy, the port connections to those modules can be done in one of ~~three~~ two ways.

1) Ordered port connection

In this method, the port expressions listed for module instance shall be in the same order as the port listed in the module declaration, ie, the first element in the list is connected to the first port declared, the second ~~port~~ element to the second port and ~~so~~ so on.

EX: The upper module instantiates a lower module. The ports are implicitly connected ie, the connection is based on order and position.

```
module lower (addr, data);
    input [width-1 : 0] addr;
    inout [depth-1 : 0] data;
endmodule // lower

module upper (in1, out1);
   input [width-1 : 0] in1;
   output [depth-1 : 0] out1;
   lower U0 (in1, out1); // implicit connection
                           in1 to addr ; out1 to data
endmodule /upper.
```

2) Named port connection.

In this method, the connection b/w the ports can be done explicitly by linking the two names for each side of the connection, ie, the port declaration name from the module declaration can be linked to the name used in the instatiating module.

EX:

```
lower U1 (
      .data(out1) ,  // order is changed
      .addr (in1)    // connection is by name only and not
                                                   position.
);
```

Note :- The order of port connection is changed, However, it is recommended to keep the same order for reusability and readability.

Adv :-

i) Improve readability of connections without having to refer to the port list of the instead module as the names from both sides are explicitly specified.

ii) The order of port connections is not relevant anymore since they are explicitly connected.

NOTE :- Implicit & explicit link types of module port connections cannot be mixed.

* Follow the connections to the ports of particular module instance shall be either by order (or) by name.

Note: `lower U_wrong (in1, .addr(out1)); //error`

→ In System Verilog, the port connecting ways are 3 types.

1) Implicit ".*" port connection
2) Implicit .name port connection
3) Interface port connection.

Can there be full (or) partial no-connects to a multi-bit port of a module during its instantiation?

Ans: No. There cannot be full (or) partial no-connects to a multi bit port of a module during instantiation.

Ex:

* The following instantiation with an intermediate bit left to float is illegal, and gives syntax error.

```
// Instantiating lower with some ports bits unconnected
lower U1 (
        .in1 (v_in1),
        .in2 ({u_in2[7], u_in2[5:0]}), //error_1
        .out1 ({u_out1[7:6], , u_out1[2:0]}), //error2
        .out2 (u_out2)
    );
```

→ for error 1, the bit 6 for in2 is floating in 8 bit in 2.

→ for error 2, bits [5:3] for out1 are unconnected in 8-bit out1.

Imp
* In the case where there is a genuine situation to not connect a particular o/p, then it must be connected to an unused <u>wire</u>, and

NOTE: A module is declared by writing
1) its functionality
2) its ports to / from the environment.
3) its timing and other attributes of design (Ex: silicon area, timing, Technology).

Module Contents

- Module The_DESIGN
    - Port Declaration
    - Parameter declaration and override
    - Type declaration
    - Event declaration
    - Continuous assignment
    - Primitive instantiation
    - Module instantiation
    - Specify block
    - Task declaration
    - Function declaration
    - Behavioural statements

    initial

    always

    Procedural assignment
    Non blocking assignment
    Procedural continuous assignment
    loops (for, repeat, while, forever)
    Flow control (if, conditional, case, wait, disable)
    system tasks and functions
    Event trigger
    Task calls
    Function calls

- End module.

GATE LEVEL MODELING UNIT- II

* Most digital design is now done at gate level (or) higher levels of abstraction.
* This gate level, the ckt is described interms of gates (eg: AND, EXOR)

1) The Built-in Primitive Gates :- The given following gate types are:

i) Multiple-input gates:
and, nand, or, nor, xor, xnor ⇐ And/Or gates

ii) Multiple-output gates:
buf, not ⇐ buf/not gates → basic Gate types

iii) Tristate gate
bufif0, bufif1, notif0, notif1

iv) Pull gates
pullup, pulldown

v) Mos switches
cmos, nmos, pmos, rcmos, rnmos, rpmos

vi) Bidirectional switches.
tran, tranif0, tranif1, rtran, rtranif0, rtranif1

* Two properties can be specified, drive_strength and delay.

Drive_strength :- It specifies the strength at the gate o/p. The strongest o/p is a direct connection to a source, next comes a connection through a conducting transistor, then a resistive pull-up/down.

* The Drive strength is usually not specified, in which case the strength default to strong1 and strong0.

Delays :- If no delay is specified, then the gate has no propagation delay; if delays are specified (rise delay, fall delay); If only one delay specified, the the rise and fall delays are equal.

* Delays are ignored in synthesis.

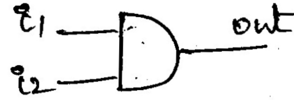
Truth Tables

(i) Multiple-input gates

1)

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

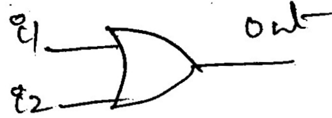
(columns: i1; rows: i2)



2)

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

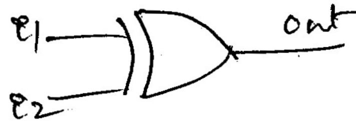
(columns: i1; rows: i2)



3)

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

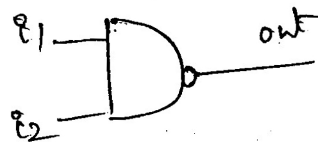
(columns: i1; rows: i2)



4)

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

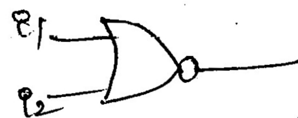
(columns: i1; rows: i2)



5)

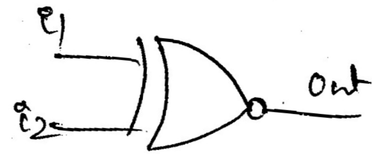
NOR	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

(columns: i1; rows: i2)



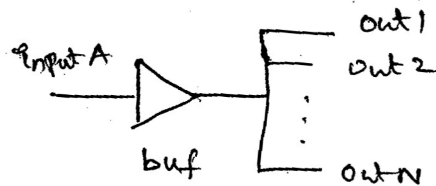
6)

XNOR (i_2 \ i_1)	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

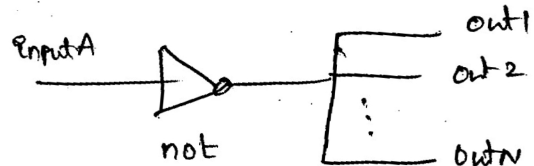


ii) multiple o/p gates (buf, not)

buf	i/p 0	1	x	z
o/p	0	1	x	x

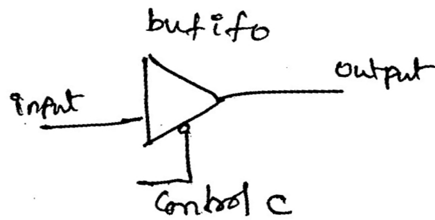


not	i/p 0	1	x	z
o/p	1	0	x	x

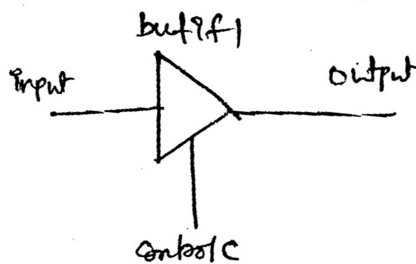


iii) Tri-state gates

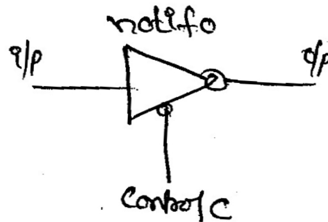
bufif0 (Data \ Control)	0	1	x	z
0	0	z	0/z	0/z
1	1	z	1/z	1/z
x	x	z	x	x
z	x	z	x	x



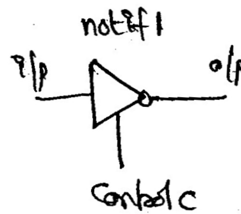
bufif1 (Data \ Control)	0	1	x	z
0	z	0	0/z	0/z
1	z	1	1/z	1/z
x	z	x	x	x
z	z	x	x	x



notif0		Control 0	1	x	z
Data	0	1	z	1/z	1/z
	1	0	z	0/z	0/z
	x	x	z	x	x
	z	x	z	x	x



notif1		Control 0	1	x	z
Data	0	z	1	1/z	1/z
	1	z	0	0/z	0/z
	x	z	x	x	x
	z	z	x	x	x



(v) **Pull gates** (Pull up, pull down)

* The Pullup and Pulldown gates have only one o/p with no i/ps.
* The pullup gate places a 1 on its o/p.
* The pulldown gates places a 0 on its o/p.

Ex: pullup PUP (pwr); // PUP → instance name.
o/p pwr tied to 1

* Verilog's pullup and pullDown primitives can be used to model pull-up and pull-Down devices in electrostatic-discharge circuitry tied to unused inputs on flip-flops.

Gate and Switch Level Primitives

n-input gates	n-output gates	tristate gates	Pull gates	mos switches	bidirectional switches
and	buf	bufif0	pullup	nmos	tran
nand	not	bufif1	pulldown	pmos	tranif0
nor		notif0		cmos	tranif1
or		notif1		rnmos	rtran
xor				rpmos	rtranif0
xnor				rcoms	rtranif1

vi) <u>MOS switches</u> (cmos, pmos, nmos, rcmos, rpmos, rnmos).

* These gates model unidirectional switches, i.e, data flow from input to o/p.

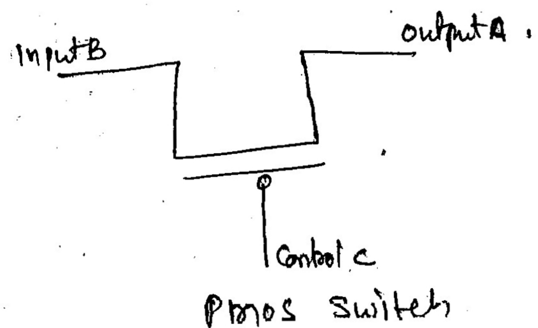
* The data flow can be turned off by appropriately setting the control i/p(s).

* The pmos, nmos, rnmos, rpmos ('r' means resistive) switches have one o/p, one i/p and one control i/p.

<u>Syntax</u>:

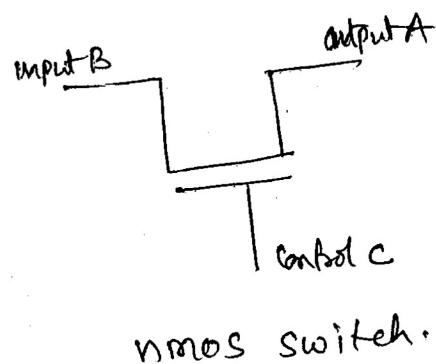
gate-type [instance_name] (output A, input B, control c);

pmos / rpmos		Control 0	1	x	z
Data	0	0	z	0/z	0/z
	1	1	z	1/z	1/z
	x	x	z	x	x
	z	z	z	z	z



pmos switch

nmos / rnmos		0	1	x	z
Data	0	z	0	0/z	0/z
	1	z	1	1/z	1/z
	x	z	x	x	x
	z	z	z	z	z



nmos switch.

* The cmos (or) rcmos switches have one data o/p, one data i/p and two control i/ps.

Syntax: (r)cmos [instance_name] (output A, input B, NControl, PControl);

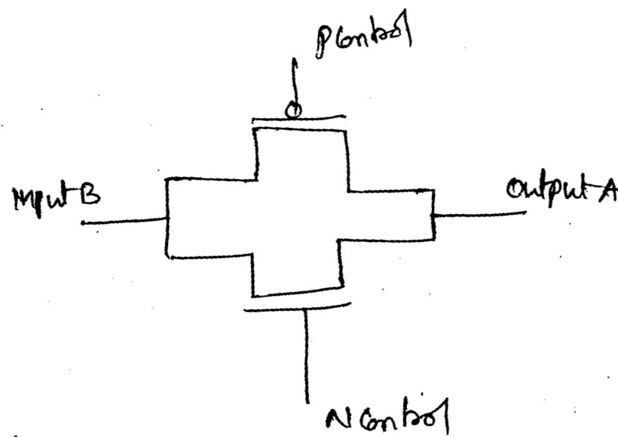


fig: (x) cmos switch.

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

vii) Bidirectional Switches (tran, rtran, tranif0, tranif1, rtranif0, rtranif1).

* The default delay is 0.
* tran → Bidirectional switch.
* rtran → Resistive bidirectional switch.
* tranif0, tranif1 } → Three state bidirectional switch.
* rtranif0, rtranif1 } → Three state resistive bidirectional switch.

* A signal passing through a bidirectional switches is not delayed. (i.e., o/p transitions follow input transitions without delay).

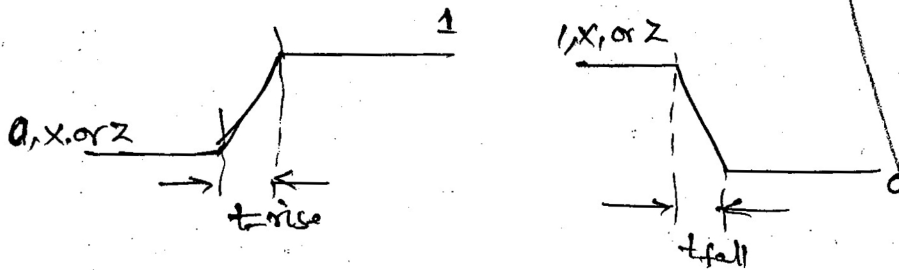
* NOTE:- * The tran and rtran primitive model bidirectional pass gates, and may not have a delay specification.

* The tranif0, tranif1, rtranif0, rtranif1 switches are accompanied by a delay specification, which specifies the turn-on and turn-off delays of the switch; the signal passing through the switch has no delay.

Delays

Rise Delay :- The rise delay is associated with a gate o/p transition to a 1 from another value.

Fall Delay :- The fall delay is associated with a gate o/p transition to a 0 from another value.



Turn-off Delay :- It is associated with a gate o/p transition to the high impedence value (z) from another value.

* If the value changes to x, the minimum of the three delays is considered.

Note :-

1. If only one delay is specified, this value is used for all transitions.
2. If two delays " , they refer to rise and fall delay values.
3. The turn off delay is the minimum of the two delays.
4. If all three delays is specified, they refer to rise, fall and turn-off delay values.
5. If no delay are specified, the default value is zero.

Syntax :

gate_type [delay] [instance name] (list of terminal_names) ;

* Multiple-i/p gates, such as and and or, and multiple o/p gates (buf and not) can have only upto two delays specified (since o/p never goes to z).
* Tristate can have upto three delays and pull gates can not have any delays.

Minimum / Typical / Maximum Values

* For each type of delay (rise, fall and turn off) contain three values, min, typ and max specified.

- These three delays are used for fabrication process variations.

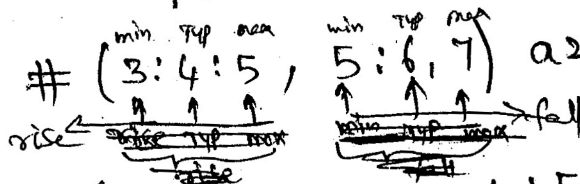
<u>Min Value:</u> The minimum value is the minimum delay value that the designers expect the gate to have.

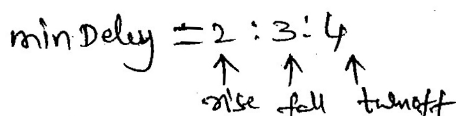
<u>Typ value:</u> The Typ is typical delay value that the designers expect the gate to have.

<u>Max Value:</u> The max value is the maximum delay value that the designers expect the gate to have.

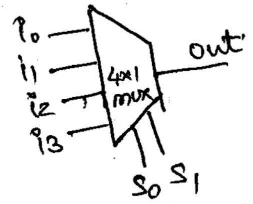
<u>Eg:</u>

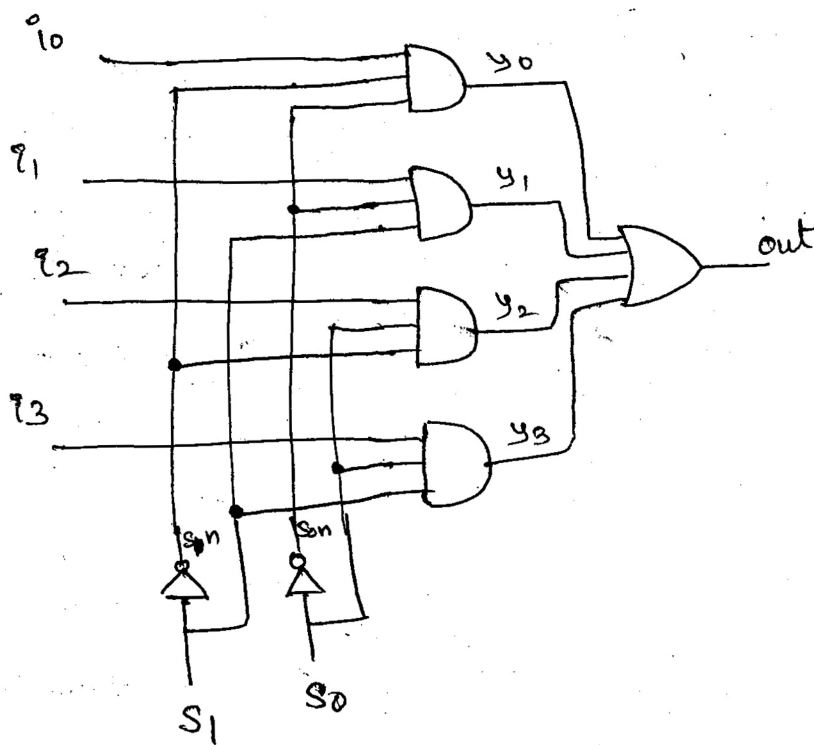
```
// if min delays, delay = 4 ; Typ delay = 5, max delay = 6)
→ one Delay
1) and #(4:5:6) a1 (out, i1, i2);

// Two Delays
// if min delays, rise = 3, fall = 5, turnoff = min(3,5)
// if Typ delays, rise = 4, fall = 6, turnoff = min(4,6)
// if Max delays, rise = 5, fall = 7, turnoff = min(5,7)
```





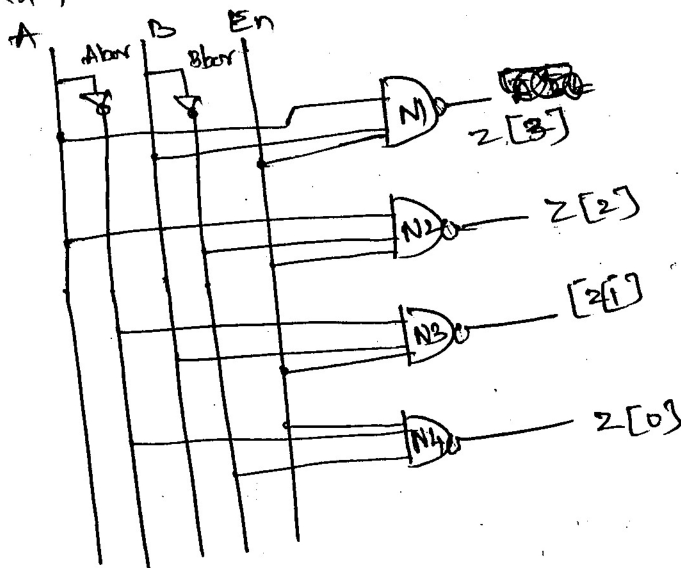
4 x 1 MUX Design using Gatelevel model.



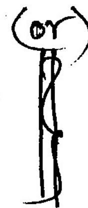



```
module mux4-to-1 (out, i0, i1, i2, i3, s1, s0);

  output out;
  input i0, i1, i2, i3;
  input s0, s1;

  wire s1n, s0n;
  wire y0, y1, y2, y3;

 // Create s1n and s0n signals.

  not (s1n, s1);
  not (s0n, s0);

  and (y0, i0, s1n, s0n);   // 3-i/p and
  and (y1, i1, s1n, s0);
  and (y2, i2, s1, s0n);
  and (y3, i3, s1, s0);
  OR (out, y0, y1, y2, y3);  // 4 i/p OR.
endmodule.
```

2x4 Decoder




```
module DEC2x4 (A,B, En, Z);
    input A, B, En;
    output [0:3] Z;
    wire Abar, Bbar;
    not #(1,2) V0 (Abar, A);
    not # (1,2) V1(Bbar, B);
```

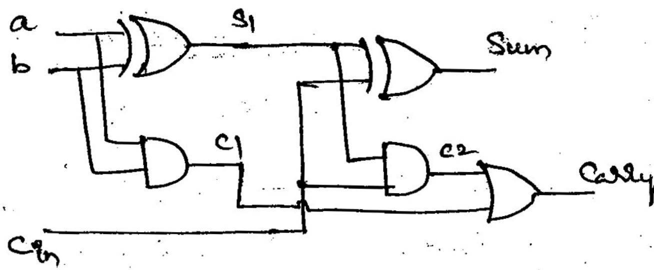


(or)

```
not#(1,2)
    V0 (Abar, A),
    V1 (Bbar, B);
```

```
    nand #(4,3)
        N0 (Z[0], En, Abar, Bbar),
        N1 (Z[1], En, Abar, B),
        N2 (Z[2], En, A, Bbar),
        N3 (Z[3], En, A, B);

endmodule
```

1-bit Adder using Gatelevel model.



Sum = a ⊕ b ⊕ c

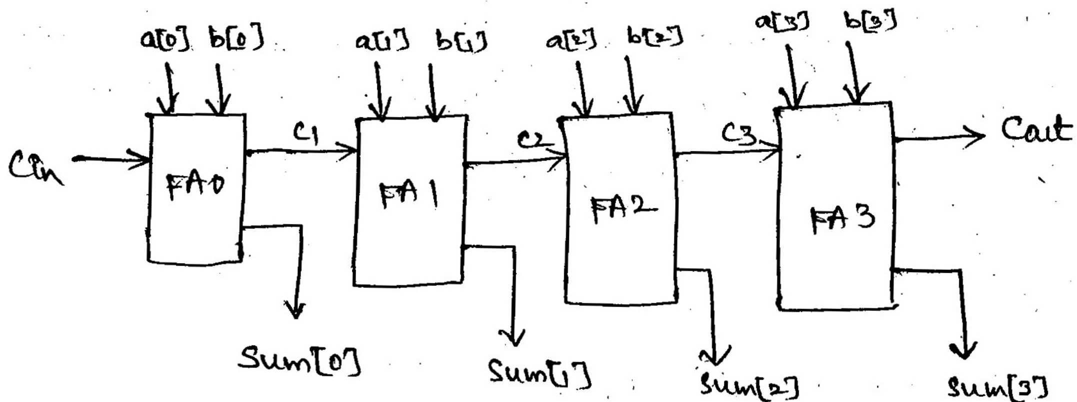
Carry = ab + bc + ca

(or)

= (a+b)c + ab

```
module fulladd(sum, carry, a, b, cin);
   output sum, carry;
   input a, b, cin;
   wire s1, c1, c2; // internal nets.
   // instantiate logic gate primitive
   xor (s1, a, b);
   and (c1, a, b);
   xor (sum, s1, cin);
   and (c2, s1, cin);
   or (carry, c2, c1);
endmodule
```

4-bit full adder.



```
module fulladd4 (sum, cout, a, b, cin);
    output [3:0] sum;
    output cout;
    input [3:0] a, b;
    input cin;
    wire c1, c2, c3; // internal nets
    // instantiate four 1-bit full adders.

    fulladd fa0 ( sum[0], c1, a[0], b[0], cin);
    fulladd fa1 ( sum[1], c2, a[1], b[1], c1);
    fulladd fa2 ( sum[2], c3, a[2], b[2], c2);
    fulladd fa3 ( sum[3], cout, a[3], b[3], c3);

endmodule
```

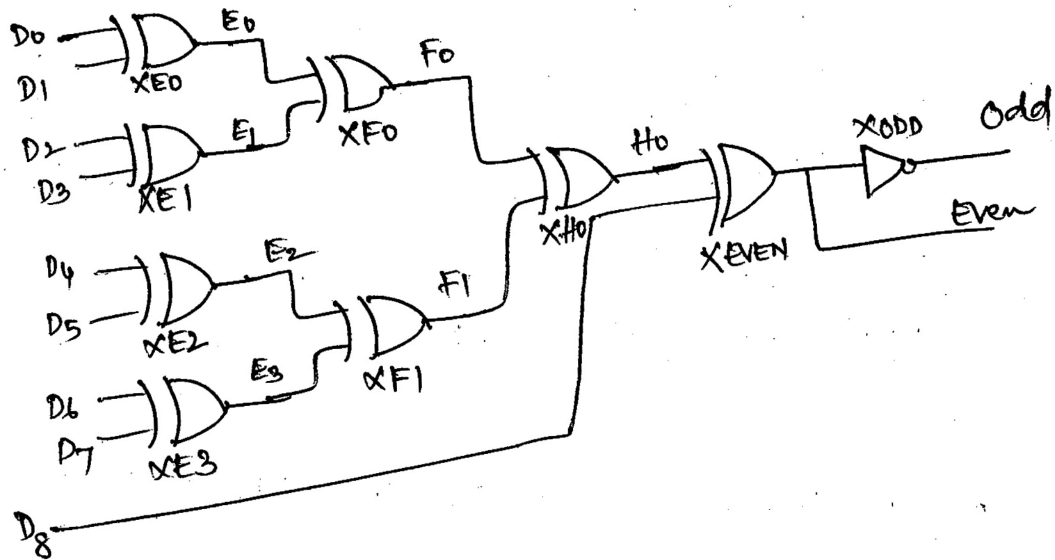
SIMULATION

```
module stimulus;
// setup variables
reg [3:0] A, B;
reg Cin;
wire [3:0] Sum;
wire Cout;
// instantiate the 4-bit fulladder
fulladd4 FA1_4 (SUM, Cout, A, B, Cin);
// setup the monitoring for the signal values.
initial
begin
    $monitor ( $time, " A = %b , B = %b, Cin = %b, Cout = %b,
                 sum = %b\n", A, B, Cin, Cout, Sum);
end

// stimulate inputs
initial
begin
    A = 4'd0; B = 4'd0; Cin = 1'b0
    #5 A = 4'd3 ; B = 4'd4;
    #5 A = 4'd2 ; B = 4'd5;
    #5 A = 4'd9 ; B = 4'd9;
    #5 A = 4'd10; B = 4'd5;
    #5 A = 4'd10 ; B = 4'd5 ;
                   Cin = 1'b1;
end
endmodule
```

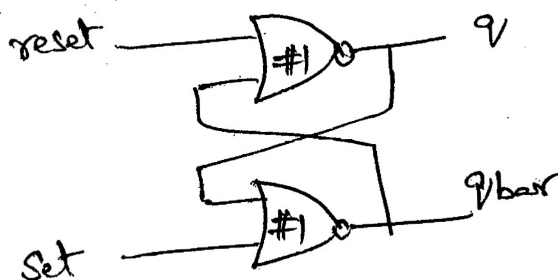
o/p of the simulation

0 A = 0000 , B = 0000 , Cin=0, ... Cout = 0, Sum = 0000
5 A = 0011 , B = 0100 , Cin=0, ... Cout=0 , sum = 0111
10 A = 0010 , B = 0101, Cin=0 ,... Cout=0 , Sum = 0111
15 A = 1000 , B = 1001 , Cin=0 ,... Cout=1 , Sum = 0010
20 A = 1010, B = 1111, Cin=0 ,... Cout=1 , Sum = 1001
25 A = 1010, B = 0101, Cin=1 ,... Cout=1 , Sum = 0000

Parity circuit.



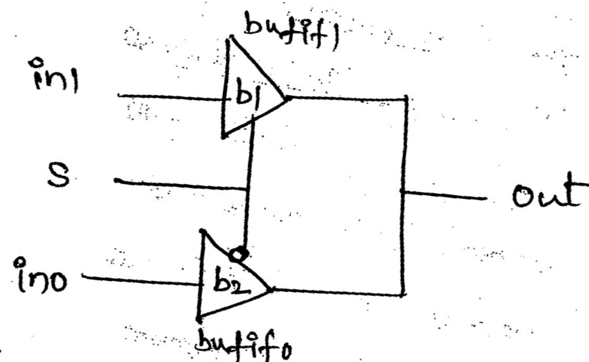
RS latch



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

9502029107

2x1 mux using bufif0 and bufif1 gates.



The delay specification for b1 and b2 are follows.

	min	Typ	max
Rise	1	2	3
Fall	3	4	5
Turnoff	5	6	7

Apply the stimulus and test the o/p values.

Note:

```
or gg[3:1] (a[3:1], b[4:2], c);
```

→ The above statement is equivalent to the combination of instantiations

```
or gg[3] (a[3], b[4], c[2]),
   gg[2] (a[2], b[3], c[1]),
   gg[1] (a[1], b[2], c[0]);
```

```
module trimux2_1 (out, in1, in0, sel);
  input in1, in0, sel;
  output out;
  bufif0  b1 (out, in0, sel);
  bufif1  b2 (out, in1, sel);
endmodule
```

(OR) for delay insertion.

(or)

```
module trimux2x1 (out, in1, in0, sel);
  input in1, in0, sel;
  output out;
  bufif0 #(1:3:5, 2:4:6, 3:5:7) b1 (out, in0, sel);
  bufif1 #(1:3:5, 2:4:6, 3:5:7) b2 (out, in1, sel);
endmodule
```

(1:3:5 — min, 2:4:6 — Typ, 3:5:7 — max)

DATA FLOW MODELING

Continuous Assignment

Assignment Statements Types:
- Continuous assignment statement
- procedural assignment statement

* The continuous assignment is used to assign a value onto a wire in a Module.
* It is normal assignment outside of always (or) initial blocks.
* The continuous assignment statements are concurrent and are continuously executed during simulation.
* The order of assign statement does not matter.
* Any change in any of the right-hand-side inputs will immediately change a left-hand-side output.

Ex①
```
wire out = in1 & in2;   (OR) // implicit continuous assignment
assign out1 = in1 & in2; // Regular assignment but declare, wire out1.
```

* Generally used for combinational ckts Design.

Ex②:
```
assign {cout, sum[3:0]} = a[3:0] + b[3:0] + cin
```

characteristics:-

1. The left hand side of an assignment must be always scalar (or) vector net (or) a concatenation of scalar and 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 of the right hand side operands changes and the value is assigned to the left hand-side net.
3. The operands on right-hand side can be registers (or) nets (or) function calls. Register (or) nets can be scalars (or) vectors.
4. Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

Ex

```
//Regular assignment
wire out;
assign out = in1 & in2;
```

```
// same as regular (implicit assignment)
wire out = in1 & in2;
```

~~Delays~~

Design 4x1 mux using Dataflow model.

Case ①

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

    output out;
    input i0, i1, i2, i3;
    input s0, s1;

  //logic
    assign out = (~s1 & ~s0 & i0) |
                 (~s1 & s0 & i2) |
                 (s1 & ~s0 & i2) |
                 (s1 & s0 & i3);

    endmodule.
```

case ② using conditional operators.

```
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);

    output out;
    input i0, i1, i2, i3;
    input s0, s1;
  // use nested conditional operator.
  assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0);
endmodule.
```

4-bit full Adder using Dataflow Modelling.

Case ①

```
Module fulladd4 (sum, Cout, a, b, cin);

    Output [3:0] sum;
    output Cout;
    input [3:0] a, b;
    input cin;
    assign {Cout, Sum} = a + b + cin;
endmodule.
```

Case ② full Adder with carry lookahead

* n-bit ripple carry adder will have 2n gate levels.
* The propagation time limiting factor on the speed of the ckt.

```
module fulladd4 (sum, Cout, a, b, cin);
  output [3:0] sum;
  output Cout;
  input [3:0] a, b;
  input cin;
  // internal wires p0, g0, p1, g1, p2, g2, p3, g3
  wire p0, g0, p1, g1, p2, g2, p3, g3;
  wire c4, c3, c2, c1;
  // Compute the p for each stage
  assign p0 = a[0] ^ b[0],
         p1 = a[1] ^ b[1],
         p2 = a[2] ^ b[2],
         p3 = a[3] ^ b[3];
```

$g_i = x_i y_i$

$p_i = x_i \oplus y_i$

$c_{i+1} = g_i + p_i c_i$

$c_1 = g_0 + p_0 c_{in}$

$c_2 = g_1 + p_1 c_1$

$= g_1 + p_1 g_0 + p_1 p_0 c_{in}$

2

```
// compute g for each stage

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

// compute carry for each stage; assume cin = c0.

assign c1 = g0 | (p0 & cin),
       c2 = g1 | (p1 & g0) | (p1 & p0 & cin),
       c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & cin);
            (p3 & p2 & p1 & p0 & cin);

// compute Sum

assign sum[0] = p0 ^ cin,
       sum[1] = p1 ^ c1,
       sum[2] = p2 ^ c2,
       sum[3] = p3 ^ c3;

// Assign cally o/p

assign cout = c4;

endmodule.
```

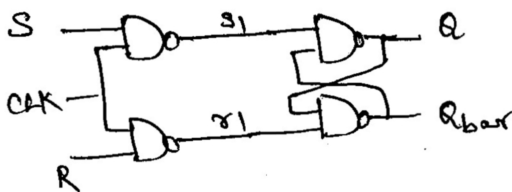
Comparator Design using Dataflow model.

```
module magcomp (A, B, ALB, AGB, AEB);
    input [3:0] A, B;
    output ALB, AGB, AEB;
    assign ALB = (A<B),
           AGB = (A>B),
           AEB = (A==B);
endmodule.
```

2x1 mux using Dataflow.

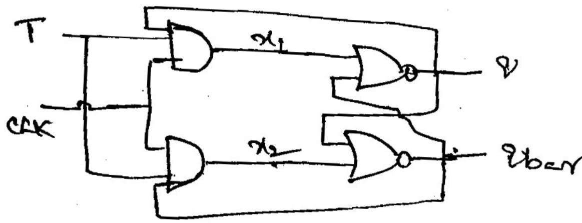
```
module mux2x1 (A, B, sel, out);
    input A, B, sel;
    output out;
    assign out = sel ? A : B;
endmodule.
```

SR Flipflop

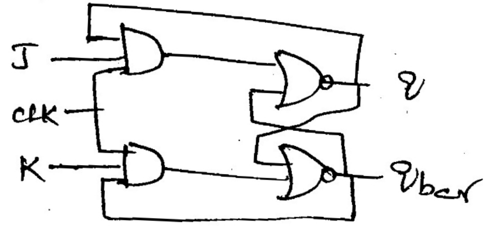



```
module SRFF (q, qbar, s, r, clk);
    output q, qbar;
    input s, r, clk;
    wire s1, r1;
    nand (s1, s, clk);
    nand (r1, r, clk);
    nand (q, s1, qbar);
    nand (qbar, r1, q);
endmodule
```

TFF

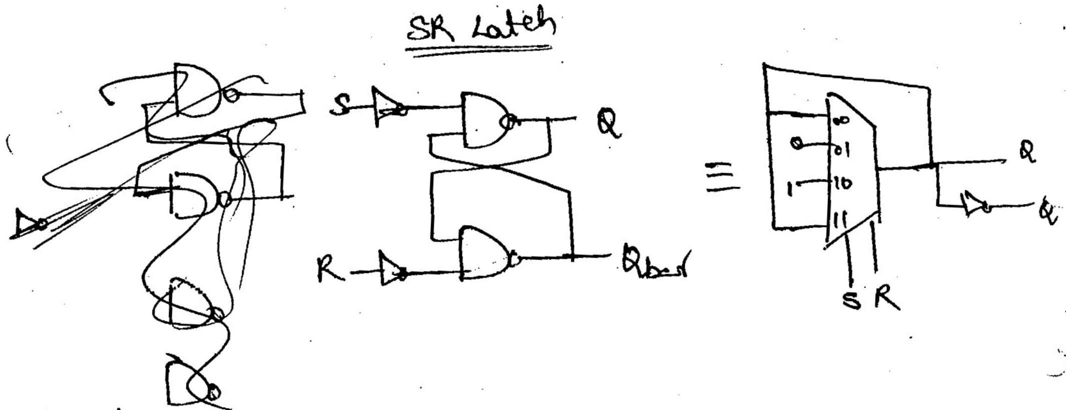


JKFF

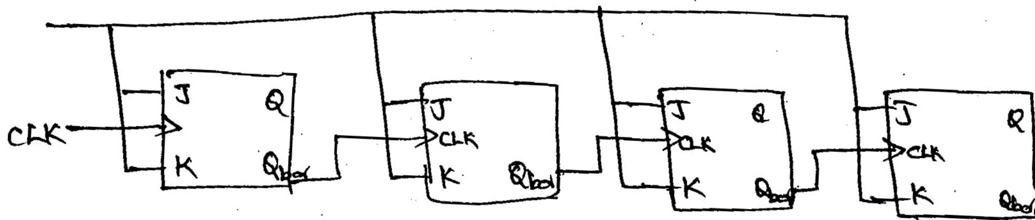


#

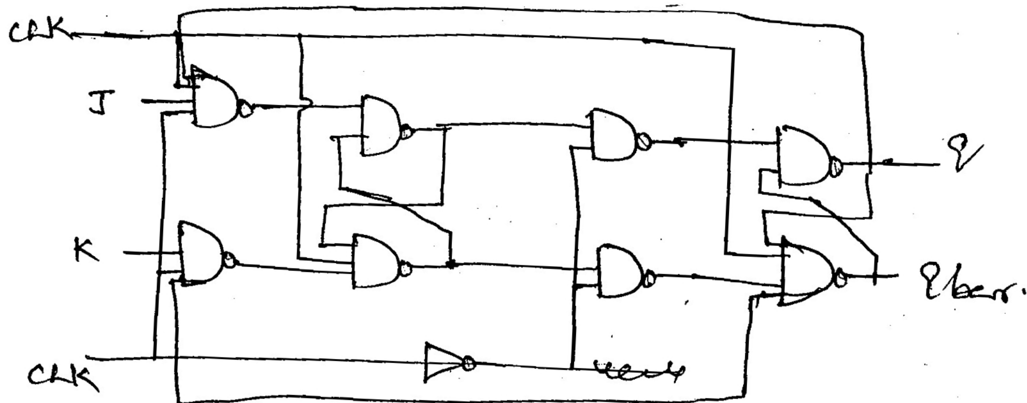
SR Latch



Ripple Counter



Master Slave JK FF



(or)

Does Verilog support an a^b operator?

Ans: Yes, Verilog supports the a^b operation by using two astrics, back to back. This operator was added with the Verilog-2001 release.

Ex:

```
module Powerof (in1, out1)
    Parameter Power = 2;
    input [1:0] in1;
    output [3:0] out1;
    assign out1 = in1 ** power; // out1 = in1 * in1
endmodule
```

Decoder Design

```
module decoder2x4 (A, B, En, Decodeout);
    input A, B, En;
    output [0:3] Decodeout;
    wire Abar, Bbar;
    assign Abar = ~A;
    assign Bbar = ~B;
    assign Decodeout[0] = ~(En & Abar & Bbar);
    assign Decodeout[1] = ~(En & Abar & B);
    assign Decodeout[2] = ~(En & A & Bbar);
    assign Decodeout[3] = ~(En & A & B);
endmodule.
```

En	x y	D_0	D_1	D_2	D_3
1	0 0	0	1	1	1
1	0 1	1	0	1	1
1	1 0	1	1	0	1
1	1 1	1	1	1	0
0	x x	1	1	1	1

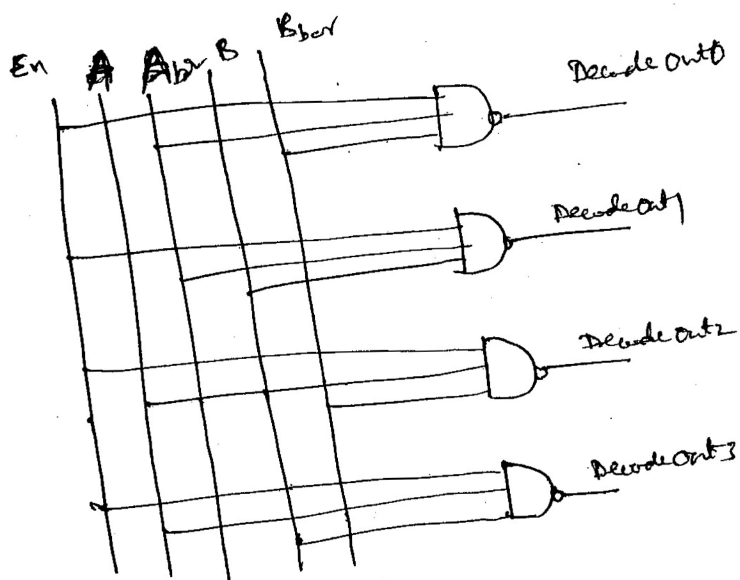


fig: Decoder logic

24/7 - (7) 67, 68, 75, 76, 78, 81, 84, 85, 86, 89, 91, 94, 96, 99, A4, A6, A7, A8, B1, B2, B5, B9, C0, 4C1, 3, 6, 9, 10, 4C7

Behavioral Modeling UNIT-III

* Verilog provides designers the ability to describe design functionality in an algorithmic manner (ie, behavior of the ckt).

* The behavioral modeling represents the ckt very high level of abstraction.

* Behavioral Verilog constructs are similar to c language constructs in many ways.

(1) Procedural constructs (or) structured procedures:

* There are two structured procedures statements in verilog.

  1. always
  2. initial

Initial statement (or) Initial block

* This is not supported for synthesis.

* ~~Initi~~ Initial statement executes only once.

* It begins its execution at start of simulation which is at time 0.

* It is typically used to initialize variables and specify signal waveforms during simulation.

* If there are multiple initial blocks, each block starts to execute concurrently at time 0.

EX:-

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

Report

Time	statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish.

Syntax

```
initial
begin
  ... statements ...
end
```

Always Block (or) Always statement

* Always block is the primary construct in RTL coding.
* Higher complexity combinational logic blocks are better described can only be used in always procedural blocks for several reasons.
* Powerful statement like if, if ... else, case and looping constructs can only be used in always block.
* All behavioural statement inside an always statement constitute an always block.
* Multiple nets assigned with in a single always block.
* If the same net can be assigned multiple times in always block; the last assignment take precedence.
* If the statements in always block are enclosed with in begin ... end, the statements executed sequentially.
* If enclosed with in fork ... join, they executed concurrently (for simulation only).

* The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion.

* Execution continuously repeats from the begin to the end of the process unless held by a wait construct through out the simulation.

Ex:

```
module clockgen;
initial
    clock = 1'b0;
always
  #10 clock = ~clock; // Time period = 20
initial
  #1000 $finish.
endmodule
```

* Note:

* always block executes the statements repeatedly (ie, once the statement finish execution they start executing all over again).

* 'initial' and 'always' procedural blocks cannot be nested.

② Assignment Statements

- Continuous assignment statement (discussed previously).
- procedural " "
  - Blocking procedural assignment
  - Non blocking " "

Procedural Assignment

Syntax: <assignment> :: = <lvalue> = <expression>

* Assign values primarily to reg variables.
* occurs inside always statement (or) initial statement (or) task (or) function.
* Results of calculations involving variables and nets can be stored into variables.
* Uses "=" (or) "<=" assignment symbol.
* Used to infer both storage elements like FFs and Latches and also combinational logic.
* The value of the previous assignment is held until another assignment is made to the variable.

Ex:

```
always @ (posedge CLK)
     reg1 <= in1;
always @ (a or b or s)
     y = (s == 1) ? a : b ;
```

* No assign Keyword (except procedural continuous assignment).

Continuous Assignment

- Assign values primarly to nets.
- occurs with in a module
- Variables and nets continuously drive values onto ports.
- used to infer combinational logic.
- Executes concurrently with other statement; executes when ever there is a change of value in an operand on its right hand side. (OR)

  In other words, Assignment occurs when ever the value on the RHS of the expression changes as a continuous process.

- occurs in assignments to wire, port, and net type.

  Ex:
  ```
  wire out1 = in1 & in2;  (or)
  assign out1 = in1 & in2;
  ```

- Uses "=" assignment symbol.

NOTE: Continuous Assignment Delay

```
✓ assign y = ~a;      // Continuous assignment - No delay.
✓ assign #5 y = ~a;   //      "          "     - LHS Delay
x assign y = #5 ~a;   // Illegal "       "     - RHS Delay.
x assign y <= ~a;     // Illegal "       "     - non blocking assignment.
```

- A blocking Assignment statement updates the left side variable with the value of the right side expression before proceeding to execute the next statement.
- A non-blocking Assignment statement, which schedules an update of the left side variable with the value of the right side expression and proceeds to execute the next statement.

⟹ Procedural Assignment Statement are ~~TWO~~ Three types — Blocking / Non blocking / procedural continuous assignment

* Blocking and non-blocking assignments behaviour given wrt simulation and synthesis as follows:

Blocking Assignment	Non-blocking Assignment
1. In a blocking assignment, the evaluation of the expression on the RHS is updated to the LHS variable autonomously based on the delay value (either 0 if no delay specified, or scheduled as a future event if a non-0 value is specified).	* In a Non-blocking assignment to LHS to scheduled to occur when the next evaluation cycle occurs in simulation and not immediately. Updates are not available immediately within the same time unit.
2. RHS value applied LHS immediately	* All RHS value applied to LHS after always block exit.
3. when multiple blocking assignments are present in a process, the trailing assignments are blocked from occuring until the current assignment is completed.	* Multiple non-blocking assignments can be scheduled to occur concurrently on the next evaluation cycle in simulation.
4. There is a possibility of race conditions on the variables of blocking assignments if assignment happen to it from two processes concurrently.	* The ~~rate~~ race conditions are avoided as the updated ~~the~~ value is assigned after evaluation.
5. Recommended to use within combinational always block	* Recommended to use within the sequential always block.
6. Can be used in procedural assignment like initial, always and continious assignments to nets like assign statements.	* Can be used only in the procedural blocks like initial and always; continuous assignment to nets like the assign statement is not permitted.
7. Represented by "=" operator sign b/n LHS and RHS	* Represented by "<=" operator sign b/n LHS and RHS.

Blocking assignment

EX:

```
initial begin
  reg1 = #10 2'b10;
  reg2 = #5 2'b01;
end
```

Starting from time 0, reg1 will be assigned 2'b10 at time 10 units and reg2 assigned 2'b01 at time 15 units. Assignment to reg2 happens after the assignment of reg1.

Non-blocking assignment

EX:

```
Initial begin
  reg1 <= #10 2'b10;
  reg2 <= #5 2'b01;
end
```

* Starting from time 0, reg2 will be assigned 2'b01 at time 5 units and reg1 will be assigned 2'b10 at time 10 units. Assignment to reg2 happens earlier than reg1.

Note:

* Don't mix "<=" (or) "=" in the same procedure
* "<=" assignment for inferring flipflops and latches ⇒ best use
* "=" assignment for inferring combinational logic ⇒ best use.

* assign LHS = expression;
  - The assign statement creates combinational logic.
  - LHS only be wire type.
  - Expression contain either wire (or) reg (or) mixed with operators

  Syntax: assign valuename [MSB:LSB] = expression;

*
```
module Add4 ( input [3:0] A, B ; input Ci , output reg [3:0] S, output reg Co );
  reg [4:0] A5, B5, S5 ;
  always @ (A, B, Ci) begin
    A5 = {1'b0, A} ; B5 = {1'b0, B} ; S5 = A5+B5+Ci ;
    S <= S5[3:0] ;   Co <= S5[4];
  end
endmodule
```

// write stimulus → If we use All statement are non blocking (A5, B5, S5) resulting values for S and Co would be incorrect.

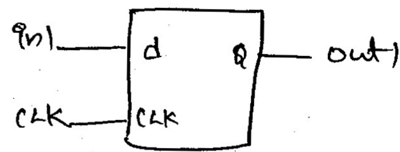
NOTE: Blocking assignment should be used for computing intermediate values in order to simplify the code. Avoid complex expression on right side of statement.

Diff b/w blocking, nonblocking in seq ckt and Combo ckt

Using blocking statement in a sequential logic

Ex1:

```
module reg-test (clk, in1, out1);
 input clk, in1;
 output out1;
 reg reg1, reg2, reg3, out1;
 always @(posedge clk) begin
    reg1 = in1;
    reg2 = reg1;
    reg3 = reg2;
    out1 = reg3;
 end
endmodule
```



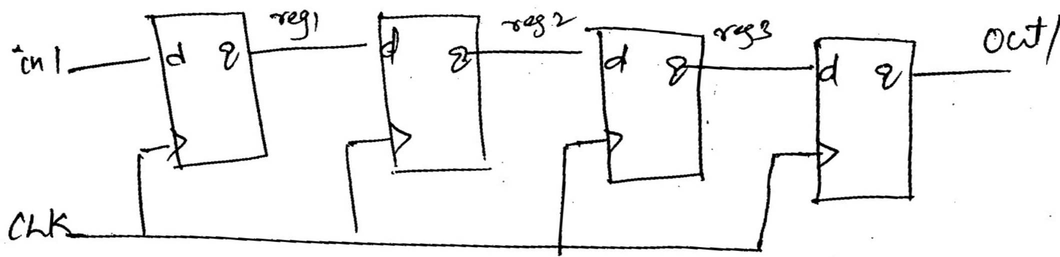
Synthesis result.
Single FF.

* Intermediate results b/w in1 and out1, removed (reg1, reg2, reg3) in blocking format.
* As a result, the evaluation of the final result to out1 didn't require waiting for all the events of the RHS to be completed. They were specified immediately assigned to the LHS.

Use nonblocking statement in sequential logic

```
module reg_test (clk, in1, out1);
input clk, in1;
output out1;
reg reg1, reg2, reg3, out1;
always @(posedge clk) begin
    reg1 <= in1;
    reg2 <= reg1;
    reg3 <= reg2;
    out1 <= reg3;
end
endmodule.
```

Synthesis result

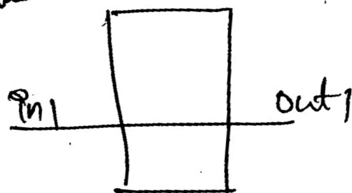


* This is because the intermediate results b/w in1 and out1 were stored in reg1, reg2, and reg3 in nonblocking format.
* The evaluation of the result to each individual reg required waiting for all the events of the RHS to be completed.
* In this case, it was the o/p of the previous register controlled by the clk event. As result, the o/p is a shift register.

Using blocking statement in combinational logic

```
module reg_test (CLK, in1, out1);
 input clk, in1;
 output out1;
 reg reg1, reg2, reg3, out1;
 always @ (in1) begin
   reg1 = in1;
   reg2 = reg1;
   reg3 = reg2;
   out1 = reg3;
 end
endmodule.
```

use <= we get same result



↑ Synthesized result

* All assignments have been immediate, and there is no event to wait upon.
* The logic synthesized simple wire b/w in1 to out1.

NOTE :-

Don't use "<=" and posedge in combination ckt.

Timing controls

1. Delay Based Timing control (#)
   - Regular Delay control
   - Intra assignment Delay control
   - Zero delay control

2. Event Based Timing control (@)
   - Regular event control
   - Named event control
   - Event OR control
   - level sensitive Timing control.

3. Level sensitive Timing control (wait)

Event Based Timing control

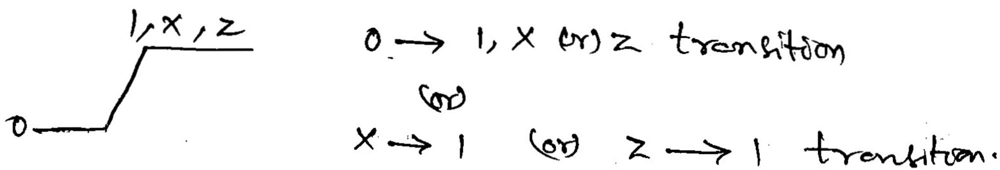
Syntax: @(edge signal or signal or ...)

* An event is the chang in the value on a register or a net.
* Events can be utilized to trigger execution of a statement or a block of statement.

Regular event control

* The @ symbol is used to specify an event control.

i) Posedge Keyword: @(posedge clk) q=d;



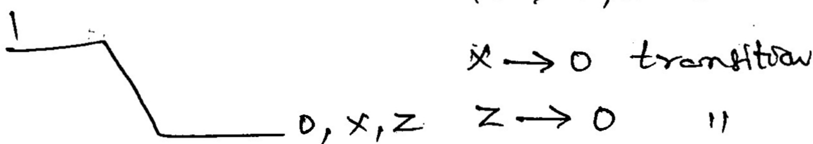
0 → 1, x (or) z transition
(or)
x → 1 (or) z → 1 transition.

ii) negedge Keyword: @(negedge clk) q=d;

1 → 0, x (or) z transition

x → 0 transition

z → 0 "



iii) q = @(posedge clock) d; / d is evaluated immediately and /assignd to q at the positive edge of clock.

iv) @ (clock) q=d; // q=d is executed whenever signal clock changes value.

Named event control

* Keyword: event
* The event does not hold any data
* An event is triggered by the symbol ->.
* The triggering of the event is recognized by the symbol @.

EX: Data buffer storing data after the last packet of data has arrived.

```
event received_data;

always @(posedge clock)
begin
    if(last_data_packet)   // If this is the last data packet
        -> received_data;  // trigger the event received_data

end

always @(received_data) // await triggering of event received_data
                        // when event is triggered, store all four
                        // packets of received data in data buf.
                        // use concatnation operator { }.

data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};
```

Event OR Control

* Sometimes a transition on any one of multiple signals (or) events can trigger the execution of a statement (or) block of statement.

EX:

```
always @ (reset or clock or d) // wait for reset (or) clk (or)
                               // d change
begin
    if (reset)         // if reset high, q=0
        q = 1'b0;
    else if(clock)     // if clk is high q= d.
        q=d
end
```

Levelsensitive control (wait)

Syntax : wait (condition)

* It delays execution until the condition evaluates as true.

Ex:

```
Initial
  begin
    wait(a==2)
      e = f &g;
  end
```

Here we wait for 'a' to be equal to 2 before evaluating 'e'.

* Levelsensitive control, is ability to wait for a certain condition to be true before statement (or) a block of statement is executed.

Delay Based Timing control

Syntax : # delay.

* It delays execution for a specific amount of time, 'delay' may be a number, a variable or an expression.

Ex:

```
always
  begin
    a= #2 (c&d);
    #3 c=(f|g);
  end.
```

⇒ Here '(c&d)' gets evaluated at time 0 but get assigned to a after 2 time units.
where as (f|g) evaluated after 3 time units and gets assigned to 'e' immediately.

Programming Statements

1. Conditional Statement
   - if.
   - if_else
2. Loops — while, for, Repeat, forever
3. Multiway Branching
   - case
   - Casex
   - Casez
4. Disable statement

① Conditional Statement

TYPE1: Statement executes (or) does not execute

```
if(expression)
    True_statement;
```

TYPE2: Conditional statement, and else statement
// either True (or) False statements executed.

```
if (expression)
    True_statement;
else
    false_statement;
```

TYPE3: Conditional Statement, Nested if_else_if
// choice of multiple statement, only one is executed.

```
if (exp1)
   True_statement1;
else if (exp2) True_Statement2;
else if (exp3) True_Statement3;
else default_statement;
```

Note :

* If statement specified but not specified else. then synthesis will generated extra latches.

* If condition evaluates to a value 0, x (or) z, the procedural statement is not executed and else statement (or) default statement is executed.

Loops

while loop

~~similar c/c++~~

Syntax:

```
while (expression)
  begin
    ... statements ...
  end
```

Ex:

```
while (!overflow) begin
  @(posedge clk)
  a = a+1;
end.
```

* while loop repeatedly executes a statement (or) block of statement until the expression in the while statement is evaluates to false.

* If the loop contains only one statement, the begin ... end statement may be omitted.

Note: To avoid combinational feed back during synthesis, a while loop must be broken with an @(posedge/neg edge clock).

* If the condition is an x or z, it is treated as a 0 (false).

* while loop is more general purpose than the for loop

for loop

* Similar to for loops in c/c++.

* Syntax:

```
for (initial_assignment ; condition ; step_assignment)
  begin
    ... statements ...
  end.
```

* They are used to repeatedly execute a statement (or) block of statement.

* If the loop contains only one statement, the begin...end statements may be omitted.

Ex:
```
for (j=0; j<=7; j=j+1)
  begin
    c[j] = a[j] & b[j];
    d[j] = a[j] | b[j];
  end
```

Repeat loop: (Not Synthesizable)

* The repeat statement executes a statement (or) block of statements a fixed no. of times.

* This statement is not supported in synthesis

* A repeat construct must contain a number, which can be a constant, a variable or a signal value.

* If the number is a variable (or) signal value, it is evaluated only when loop starts and not during the loop execution.

Syntax:
```
repeat (no.of_times)
  begin
    --- statements ----
  end
```

Ex:
```
repeat(2)
  begin           // after 50, a=00
    #50 a = 2'b00; // after 100, a=r~
    #50 a = 2'b01; // after 150, a=G
  end // after 200, a=01
```

* If ~~loop~~ no. of times statement expression is an x or z, then the no.of_times is treated as a 0.

Examples

① repeat (count)

sum = sum + 10;

② repeat (shift By)

P_Reg = P_Reg << 1;

③ repeat (count) // Repeat loop statement

@(posedge clk) sum = sum + 1;

which means for count times, wait for positive edge of clk and when this occurs, increment sum.

(or)

④ sum = repeat (count) @ (posedge clk) sum + 1;

// Repeat event control.

means to compute sum+1 first, then wait for count positive edges on clk, then assign to left hand side.

⑤ repeat (Num_of_Times) @ (negedge clkz)

It means to wait for num_of_Times negative clock edges before executing the statement following the repeat statement.

NOTE: A while loop statement is a loop that continues to execute its statements as long as the loop's condition evaluates true. It is used to testbenches. If the condition is ~~false~~ True, the loop's substatements is executed, and the condition is checked again.

→ A for loop typically used when the specific no of iteration is known, where as a while is used when the no of iteration is not known.

forever

* The forever statement executes an infinite loop of a statement (or) block of statement.
* The loop does not contain any expression and executes forever until the $finish task is encountered.
* The forever loop can be exited by use of the disable statement.
* A forever loop is typically used in conjunction with timing control constructs. otherwise forever-loop will loop forever in zero delay.
* Syntax :

```
forever
  begin
    ---statements---
  end.
```

Ex: ①

```
forever begin
    @ (posedge clk); //(or) use   a = #9 a+1;
    a = a+1;
end.
```

Ex ②

```
initial
  begin
    clock = 0;
    #5 forever
        #10 clock = ~clock;
  end.
```

Multiway Branching

Case :-

Syntax :

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

Imp * If no comparisons, including default, are true, synthesizers will generate unwanted (or) extra latches.

Case x :-

* It is special version of case statement where x, z and ? are treated as don't cares.
* Compare expression with each of case_item and executes statements (or) statement group associated with the first matching case_item.
* It executes the default if none of the case_item's match.
* The simulation variable would have to match a

  z = tristate
  x = unknown
  ? = either signal

Syntax:

```
casex (case_expr)
    alternative1 : statement1;
    alternative2 : statement2;
    ...
    default : default_statement;
endcase.
```

Ex:

```
Casex (a)
   2'b1x : msb = 1; // msb = 1 if a = 10 (or) a = 11
                    // If this were case(a) then only a = 1x would match.
   default: msb = 0;
endcase.
```

* Casex uses x as a don't care which can be used to minimize logic.

Casez :-

* It is special version of case statement where z and ? are treated as don't cares.
* An inadvertent x signal, will not be matched by a 0 or 1 in the case of choise.

Syntax :

```
casez ( expr )
   case_choice1 : statements1 ;
   case_choice2 : statements2 ;
   . . .
   default : default_statements ;
```

Eg:

```
casez (d)
   3'b1??: b = 2'b11; // b = 11 if d = 100 or greater
   3'b01?: b = 2'b10; // b = 10 if d = 010 or 011
   default: b = 2'b00; // default b = 00
endcase.
```

Advantages of Casex and Casez

* It reduces the no. of lines, especially if the no of bits had been more.
* Make code look more clear and less clustered.
* Simplifies the optimization, as it clear that the bits with x are to be ignored.

NOTE:

* For multiway branching, if we need more than one statement, (i.e. statement_group) then we need to use begin ... end (or) a fork ... join block.

Disable Statement

* Keyword : disable
* It is like 'c' keyword break statement
* Execution of a disable statement terminates a block and passes control to the next statement after the block.
* Disable statement can only be used with named blocks.

Syntax:

```
disable named block ;
```

Ex:

```
begin: accumulate;
forever
    begin
    @ (posedge clk)
        a=a+1;
        if(a == 2'b0011)
          disable accumulate;
    end
end // end accumulate.
```

Comparision

Computational Activity Flow ~~loop~~	features	Termination
* repeat	expression determines a no of repetations	False expression (or) "disable"
* for	loop variable determines no of repetations	"end" (or) "disable"
* while	Iterates while expression evaluates true	"end" (or) disable
* forever	Iterates Indefinitely	disable → Permenantly terminate the block.

Expression (or) Case item	Case	Casex	Casez
0	0	0	0
1	1	1	1
x	x	01xz	x
z	z	01xz	01xz
?	*	*	01xz
default :	01xz	01xz	01xz

NOTE:
* → not applicable

Examples

① Simple ALU

```
module ALU (z, x, y, alu-control);
    output z;
    input x, y;
    input alu-control;

    reg [1:0] alu-control;
  always @ (alu-control)
    case (alu-control)

      2'd0 : z = x + y;
      2'd1 : z = x - y;
      2'd2 : z = x * y;
      default: $ display ("Invalid ALU Control Signal");

    endcase
endmodule.
```

```
if(alu-control == 0)
    z = x+y;
else if(alucontrol ==1)
    z = x-y
else if (alu control ==2)
    z = x * y
else default: $display ("Invalid opcode")
```

② 4×1 mux using case.

```
module mux4to1 (out, i0, i1, i2, i3, s1, s0);
    output out;
    input i0, i1, i2, i3;
    input s1, s0;
    reg out;
    always @(s1 or s0 or i0 or i1 or i2 or i3)
    case ({s1, s0})

        2'd0 : out = i0;
        2'd1 : out = i1;
        2'd2 : out = i2;
        2'd3 : out = i3;
        default: $display ("invalid Control Signals");
                   (or)
        default : out = 1'bx;
    endcase
endmodule
```

③ Demux (1 to 4 Demux) case statement with casex and z

```
module demux1_to_4 (out0, out1, out2, out3, in, s1, s0);
    output out0, out1, out2, out3;
    reg out0, out1, out2, out3;
    input in;
    input s1, s0;

    always @(s1 or s0 or in)
    case({s1,s0})
        2'b00 : begin
            out0 = in;
            out1 = 1'bz;
            out2 = 1'bz;
            out3 = 1'bz;
        end
        2'b01 : begin
            out0 = 1'bz;
            out1 = in;
            out2 = 1'bz;
            out3 = 1'bz;
        end
        2'b10 : begin
            out0 = 1'bz;
            out1 = 1'bz;
            out2 = in;
            out3 = 1'bz;
        end
        2'b11 : begin
            out0 = 1'bz;
            out1 = 1'bz;
            out2 = 1'bz;
            out3 = in;
        end
        2'bx0, 2'b0x, 2'bx1, 2'b1x, 2'bxz, 2'bzx, 2'bxx :
        begin
            out0 = 1'bx; out1 = 1'bx; out2 = 1'bx; out3 = 1'bx;
        end
        2'bz0, 2'b0z, 2'bz1, 2'b1z, 2'bzz : begin
            out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz;
        end.
        default : $display ("Unspecified Control Signals");
    end case
endmodule
```

NOTE:
Select Signal = x then o/p = x
Select Signal = z then o/p = z
In x & z, higher priority is x.

While loop EX

① // Increment Count from 0 to 127, Exit on Count 128.

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

② for loop

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

③ // Initialize Array elements

```
`define MAX_STAGES 32
integer state [0: MAX_STAGES-1];
integer i;

initial
begin
    for (i=0; i<32; i=i+2)   // even locations
        state[i] = 0;
    for (i=1; i<32; i=i+2)   // odd locations
        state[i] = 1;
end
```

(4) Repeat loop

```
// increment & display count from 0 to 127

integer count;

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

    end
end
```

(5)

```
// Data buffer module
// After it receives a data_start signal
// Reads data for next 8 cycles.

module data_buffer ( data_start, data, clock);

parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;

reg [15:0] buffer [0:7];

integer i;

always @(posedge clock)
begin
    if (data_start)
    begin
        i = 0;
        repeat (cycles)        <- Store data at posedge of next 8 clock cycles
        begin
            @(posedge clock)   -> wait till next posedge to latch data
            buffer[i] = data;
            i = i + 1;
        end
    end
end
endmodule
```

Block statements < Sequential block / Parallel block.

Sequential block :-

* The Keyword : begin ... end for group statements

* Characteristics :

  - The statements in a sequential block are processed in the order they are specified. A statement executes only after its preceding statement completes execution (except for non blocking assignment with intra assignment control).

  - If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.

* Statements are executed sequentially.

Ex ①: Seq block without delay.

```
reg x, y;
reg [1:0] z, w;

initial
 begin
      x = 1'b0;
      y = 1'b1;
      z = {x, y};
      w = {y, x};
 end
```

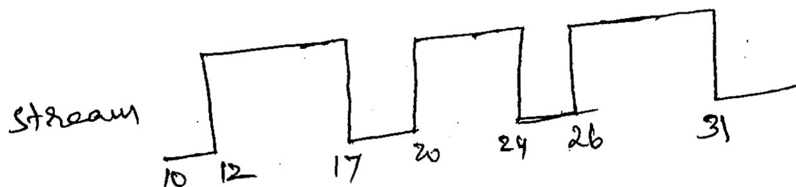
Ex ②: Seq blocks with delay.

```
reg x, y;
reg [1:0] z, w;

initial
 begin
    x = 1'b0;          // completes at simulation time 0
    #5 y = 1'b1;       //      "          "          5
    #10 z = {x, y};    //      "          "          15
    #20 w = {y, x};    //      "          "          35
 end
```

Ex ③: Wave form generation.

```
begin
   #2 stream = 1;
   #5 stream = 0;
   #3 stream = 1;
   #4 stream = 0;
   #2 stream = 1;
   #5 stream = 0;
end
```

Assume the Seq block gets executed at 10 time units.



Parallel blocks :

* Statements in this block executes concurrently.
* Keyword : fork... join for group of statements
* This is provided for simulation features.
* characteristics :-
  - Statements in a parallel block are executed concurrently.
  - ordering of statement 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 block was entered.

* In simulation time, all statements in the fork.. join block are executed at once.

Ex① : Parallel blocks with delay.

```
reg x, y;
reg [1: 0] z, w;

initial
fork
      x = 1'b0; // completes at simulation time 0
   #5 y = 1'b1; //    "         "          5
   #10 z = {x,y}; //  "         "          10
   #20 w = {y,x}; //  "         "          20.
join
```

* Different simulators execute statements in different order. Thus, the race condition is a limitation of today's simulators, not of the fork-join block.

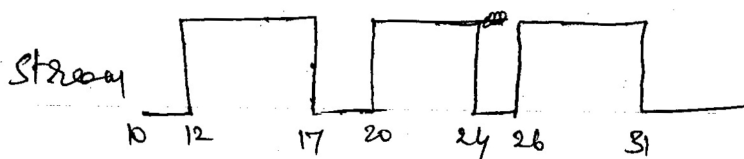
Ex(2): Parallel blocks with deliberate race condition.

```
reg x, y;
reg [1:0] z, w;
initial
fork
    x = 1'b0;
    y = 1'b1;
    z = {x, y};
    w = {y, x};
join
```

* The keyword fork can be viewed as splitting a single flow into independent flows.
* The keyword join can be seen the independent flows back into a single flow.
* Independent flows operate concurrently.

Ex(3): wave form generation.

```
fork
  #2 stream = 1;
  #7 stream = 0;
  #10 stream = 1;
  #14 stream = 0;
  #16 stream = 1;
  #21 stream = 0;
join
```



Special features of Blocks

Three types : 1. Nested blocks
2. named blocks
3. Disabling of named blocks.

~~① Named blocks:~~

① Nested blocks :

* Blocks can be nested
* Sequential and parallel blocks can be mixed.

Ex :

```
initial
begin
        x = 1'b0;
        fork
              #5 y = 1'b1;
              #10 z = {x, y};
        join
        #20 w = {y, x};
end
```

② Named blocks :

* Blocks can be given names.
  - Local variables can be declared for named blocks
  - Named blocks are a part of the design hierarchy. Variables in a named block can be accessed by using hierarchical name referencing.
  - Named blocks can be disabled. i.e, their execution can be stopped.

Ex

```
module top ;
  initial
  begin : block1 // Seq block named block1
  integer i ;  // integer i is static and local to block1
  . . .        // can be accessed by hierarchal name, top.block1.i
  . . .
  end

  initial
  fork : block2 // Parallel block named block2
  reg i ;       // reg i is static and local to block2
  . . .         // can be accessed by hierarchal name, top.block2.i
  . . .
  join
```

③ Disabling named blocks

- Keyword : disable.
- Disable statements can only be used with named blocks.

Syntax :

```
disable named-block ;
```

Notes

MSB — LSB

1) reg [15:0] addr ;   Syntax : reg [MSB : LSB] Variable;

reg [0 : 15] addr ;
MSB — LSB

2) integer count[0:7] ; // An array of 8-count Variable

reg bool [31:0]; // An array of 32- one-bit boolean reg Variable

reg [4:0] Port_id [0:7]; // An array of 8-port ids each port id is 5 bits word.

integer Matrix[4:0][0:255]; // Two dimensional array of integers.

3) Vector → single element that is n-bits wide

array → Multiple elements that is 1-bit or n-bits wide.

4) wire a;
   ↳ multiple Drivers driving a nets.

* Nets (or) Reg data types can be declared as Vectors (multiple bit widths)

Ex: wire a;
    wire [7:0] bus ; // 8-bit bus

    reg CLK;
    reg [0:40] data ; // Vector register, data 41 bits wide.

5) i) wire out ;
   assign out = in1 & in2; } // Regular assignment statement.

ii) wire out = in1 & in2; // Implicit continuous assignment statement

6) Procedural Assignment Delays.

i) Inter statement Delay: This is the delay by which a statements' execution is delayed. This we can call delayed assignment.

Ex : #4 sum = (A ^ B) ^ Cin ;

ii) Intra-statement delay :- This is the delay b/w computing the value of the right hand side expression and its assignments to the left hand side.

Ex : Sum = #3 (A^B) ^ Cin;

iii) Inter statement event Control

Ex:

```
begin
  Temp = D;
  @ (Posedge CLK)
     Q = Temp;
end
```

iv) Intra-Statement event control

Ex: `Q = @ (Posedge CLK) D;`

7) Procedural Continuous Assignments

(1) For Simulation

```
initial
  begin
    a=1; b=2; c=3;
  #5  a = b + c; // wait for 5 units, and execute a = b+c = 5
      d = a;  //Time continues from last line d=5= b+c at
                                                   t=5.
```

(2)

```
reg [6:0] sum;
reg a, b;
sum [7] = b[7] ^ c[7]; //execute now
  b = #15 CLK&a; // clk & a evaluated now; 'b' changed
                          after 15 time units.
  #10 W= b&C; //10 units after b changes, b&c is
                          evaluated and w changes.
```

(3)

```
initial begin                    ↗ take roughly by force
  # 3  b <= a;  // grab a at t=0, deliver b at t=3
  #6   x <= b+c; // grab b+c at t=0, wait and
                     assign x at t=6.
                 /* x is uneffected by b's change.
```

(4) initial begin

a=1; b=2; c=3; x=4;

5 a = b+c; // wait for 5 units, then grab b, c and execute a = 2+3;

d=a; // Time continues from last line, d=5 = b+c at t=5

x <= #6 b+c; // grab b+c now at t=5, don't stop, make x=5 at t=11

b <= #2 a; // grab a at t=5 (end of last blocking statement)
/* Deliver b=5 at t=7. previous x is uneffected by 'b' change */

y <= #1 b+c; // grab b+c at t=5; don't stop, make x=5 at = 6.

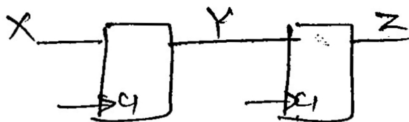
#3 z = b+c; // grab b+c at t=8 (#5, #3), make z=5 at t = 8

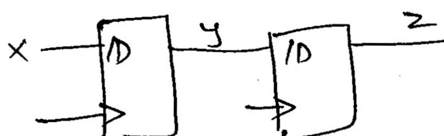
w <= x; // make w=4 at t=8.
// Starting at last blocking assign

(5)

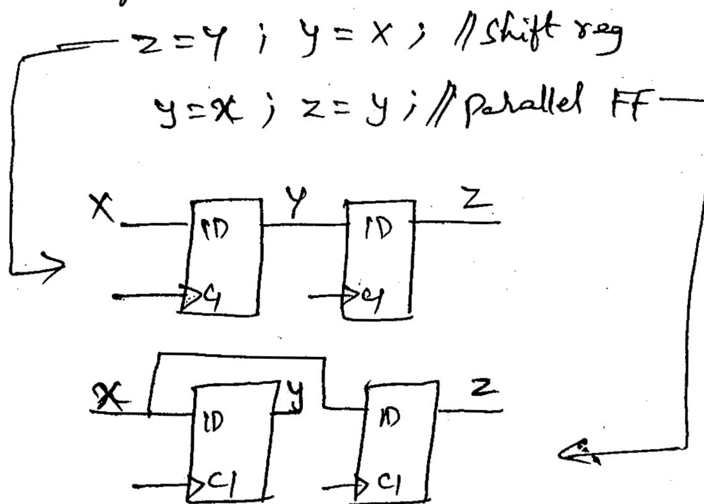
always @ (posedge clk)
begin

z <= y; y <= x; // shift register
y <= x; z <= y; // also a shift reg





(6) always @(posedge clk)
begin



(7)

Procedural Continuous Assignment

* A procedural continuous assignment is a procedural assignment.
* It appear with in an always (or) an initial statement.
* This assignment can override all other assignments to a net or register.

There are two kinds of procedural continuous assignments.

1) assign and deassign :- these assign to registers
2) force and release :- these assign primarily to nets, through they can also be used for registers

* The target of a procedural continuous assignment cannot be a part-select (or) a bit-select of a register.

① Assign - deassign

* Key word : assign . . . deassign
* The left hand side of a procedural continuous assignments can only be a register (or) concatenation of register.
* It can not be a part-select (or) bit select of a net (or) an array of registers.
* It can override the effect of regular procedural assignment.
* ~~procedural~~ normally used for controlled periods of time.

MODULE PARAMETERS — "defparam"

* Module parameters are associated with size of bus, register, memory, ALU, and so on.

* They can specified within the concerned module but their value can be altered during instatiation.

* The alterations can be brought about through assignments made with "defperm". Such defparam can appear (assignments) any where in a module.

* The rules of assigning values for the module parameters, deciding their size, type, etc., are all similar to those of specify parameter discussed before.

Syntax: defparam hier_path_name1 = value1,
hier_path_name2 = value2, . . . ;

What does it mean to "short-circuit" the evaluation of an expression?

A: * Verilog supports numerous operators that have rules of associativity and precedence.

* In some of the expressions, the result of the expression can be evaluated early on, due to the precedence and influence to override the rest of the expression. In that expression need not be evaluated. This is called short-circuiting and expression evaluation.

Ex:

```
input in1, in2, in3, in4;
wire wire1, wire2;
assign wire2 = (in1 > in2) & (in3 | in4);
```

In above exp.
If the result of (in1 > in2) is false (1'b0) then tools can already determine that the result of the AND operation will be 0. and rest of the sgn Thus, there is no need to evaluate (in3 | in4) is short circuited.

What is the order of precedence when both assign-deassign and force-release are used on the same variable?

A: * Force statement overrides the value of assign statement until it is released.

Ex:

```
module forcerelease;
  reg [1:0] w1;
  initial begin
  $display ("1 w1 = %0d, t = %0d", w1, $time);

  assign w1 = 1;
  #5 $display(" 2 w1 = %0d , t = %0d ", w1, $time);

  release w1;
  #5 $display ("3 w1 = %0d , t = %0d", w1, $time);

  deassign w1;
  #5 $display ("5 w1 = %0d , t = %0d", w1, $time);
  end
endmodule.
```

Output:

1 w1 = x, t = 0

2 w1 = 1, t = 5

3 w1 = 2, t = 10

4 w1 = 1, t = 15

5 w1 = 1, t = 20

As evident from the above, the force command has overridden the assigned value earlier and relinquished it back its the assigned value after the release command.

What is the main difference force-release and assign-deassign?

assign - deassign	force - release Construct.
* This is applicable only variables	* This is applicable for nets and variables.
* used for modelling Hardware behaviour, but the construct is not synthesizable by most logic synthesis tools.	* used for design verification, and not synthesizable.

Concurrency :-

* The functionality of the overall system is achived by concurrently active components communicating through their i/p & o/p ports.

* The use of concurrent construct, timing of interconnecting signals, and order of simulation of construct or components, a VHDL/Verilog simulator makes us (the users) think that such execution is being done concurrently.

EX: primitive

Continuous assignment Statement	Procedural assignment Statement (Blocking)	Non-blocking procedural assignment Statement	Procedural Continuous assignment Statement
* It occurs outside an always (or) initial Statement.	* Inside an always (or) initial Statement	* inside an always (or) initial Statements	* always (or) initial Statements.
Ex: wire [31:0] Z; assign Z = en ? Data : 32'b2;	Ex: reg y; always @ (posedge clk) y = 1;	Ex: reg y always ... y <= 1	Ex: always @(en) if (en) assign Q = 1 else deassign Q.
* Valid LHS data type is net.	* register (or) memory element	* register or memory element	* net
* Valid RHS of assignment (expression) is net, reg (or) memory element	* net, reg (or) memory element	* net, reg (or) memory element	* net, reg (or) memory element.

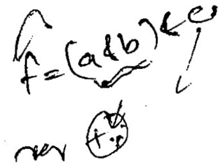
Structure of Verilog

// Declarations.

1. time scale
2. Module Name (input & outputs)
3. Parameter
4. Input/output Declarations
5. Internal signal register type declarations
   - reg types (only assigned with in always statement)

     .... reg Var1 ;

     reg [msb : LSb] reg_Var2 ;
6. Internal net type declarations (only assigned outside always statement) wire, net variables.
7. Hierarchical names – instantiating another module.
8. Synchronous procedures.
9. Combinational procedures.
10. Assign Net Variable = Combo logic ;
11. endmodule.



RTL Codes

1) Simple D-FF

* +ve edge triggered,
* no set (or) reset values

```
module dff (clk, d, q);
  input clk, d;
  output q;
  reg q;
  always @(posedge clk) begin
      q <= d;
  end
endmodule
```

2) Asynchronous Set FF

* +ve edge triggered
* active high asynch set.

```
module asff (clk, d, set, q);
    input clk, d, q;
    output q;

    reg q;
    always @ (posedge clk or posedge set)
    begin if (set)
        q <= 1'b1;
        else
          q <= d;
    end
endmodule
```

Asynchronous reset FF

```
module arff (clk, d, reset, q);
  input clk, d, reset;
   output q;
   reg q;
  always @ (posedge clk or posedge reset)
  begin
    if(reset)
       q <= 1'b0;
    else
       q <= d;
  end
endmodule
```

* (4) Asynch set & reset flipflop

```
module arsff (clk, d, set, reset, q);
  input clk, d, set, reset;
  output q;
  reg q;
always @ (posedge clk or posedge set
             or posedge reset)
  begin
    if(set)
      q <= 1'b1;
    elseif (reset)
        q <= 1'b0;
    else
      q <= d;
  end
endmodule.
```

5) Synchronous set FF

* +ve edge triggered
* active high synch set

```
module ssff(clk, d, set, Q);
 input clk, d, set;
 output Q;
 reg Q;
 always @ (posedge clk) begin
   if(set)
     Q <= 1'b1;
   else
     Q <= d;
   end
 endmodule
```

6) Synchronous reset FF

* +ve edge triggered
* active high synch reset.

```
module srff(clk, d, reset, Q);
  input clk, d, reset;
  output Q;
  reg Q;
 always @ (posedge clk) begin
   if(reset)
     Q <= 1'b0;
   else
     Q <= d;
  end
endmodule
```

7) Synchronous Set & reset Flipflop

```
module ssrff (clk, d, set, reset, Q);
 input clk, d, set, reset;
 output Q;
 reg Q;
 always @ (posedge clk) begin
  if(set)
   Q <= 1'b1;
  else if(reset)
    Q <= 1'b0;
  else
    Q <= d;
 end
endmodule
```

8) Latch Types

```
i) always @ (sel, d) begin
     if(sel)
       Q <= d; // else clause missing
   end

ii) always @ (*) // Implicit sensitivity list
     if(set)
       Q <= d;
    end

   Asynch set latch
iii) always @ (set, d, set)
     begin
      if(set)        } if (reset)
       Q <= 1'b1;    }   Q <= 1'b0;
      else if(sel)
         Q = d; // final else clause missing
     end

iv) Asynch set & reset FF
   always @ ( sel, d, reset, set) begin
     if(reset) Q <= 1'b0;
   else if(set) Q = 1;      // final else clause missing
   else if (sel) Q = d; end //
```

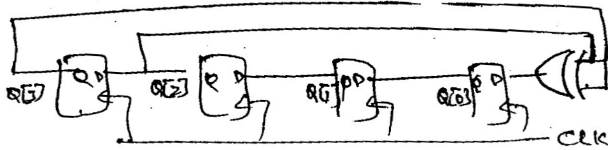
Shift registers

```
module (clk, Q)
    input clk, reset
    output [3:0] Q;
    reg [3:0] Q;
  always @ (posedge clk or posedge reset)
  begin
      if(reset)
          Q <=0;
      else
        begin
          Q <= Q<<1 ; //Leftshift
          Q[0] <= Q[3] ; //old Q[3] is sent to Q[0]
                          not revised Q[3] from the previous line.
        end
    end
endmodule.
```

Right Shift

```
// Q <=Q+1;
   Q[3] <= Q[0];
```

LFSR

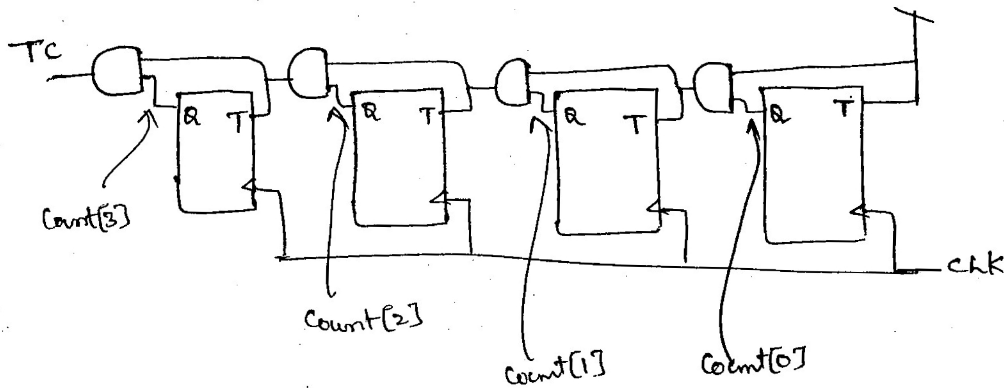



```
module ( clk, reset, Q
   input clk, reset;
   Output Q ;
   reg Q ;
   always @(posedge clk or posedge reset)
    Begin
       if(reset) Q <=0;
       else begin
          Q <= { Q[2:1] : Q[3] ^ Q[2] } ;
       end
    end
endmodule .
```

LFSR

Counter

* Counters are a simple type of FSM where separation of the flipflop generation code and the next state generation code is not worth the effort.




```
module (reset, count, clock);
    input reset, clock
    output [3:0] count;
    reg [3:0] count ;
    wire TC; // Terminal Count
    always @ (posedge clk or posedge reset)
    begin
       if(reset)
           count <=0 ;
       else
           count <= count +1;
     end
    assign TC = & count ;
endmodule.
```

4-bit binary counter

```
module counter (Q, clock, clear);
    output [3:0] Q;
    input clock, clear;
    reg [3:0] Q;
    always @ ( posedge clear or negedge clock)
    begin
        if(clear)
            Q = 4'd0;
        else
            Q = (Q+1) % 16;
    end
endmodule
```

8-function ALU

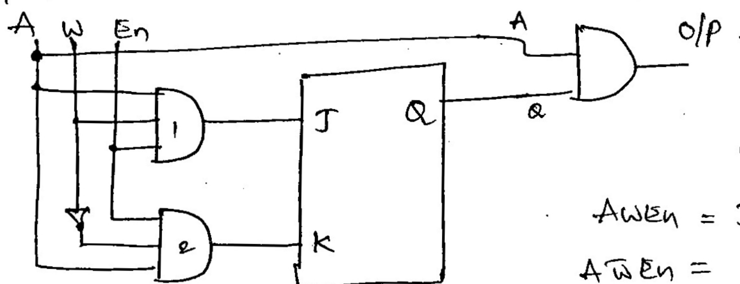
Design full Adder ckt by using Decoder.

Design Decoder ckt with Enable signal.

Design all basic logic gates using MUX.

Implement boolean logic function by usign MUX for given f(xyz) to

Basic RAM Cell



o/p = A and Q ;

AWEn = J = A and W and En ;

$A\bar{W}En$ = K = A and NW and En.

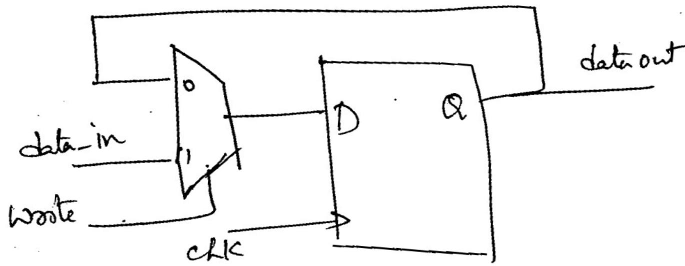
Initiall clear Q to zero. when data is to be written, and AND gate 1 and 2 are enabled by applying positive pulses to their respective enable i/ps. Now, a 1 is applied to the address lines A and the data is entered through the write input line W.

If W = 0, J = 0 and K = 1 then Q = 0

If W = 1, J = 1 and K = 0 then Q = 1

Thus whatever be the data on line W, the JK FF will store it. To read, we again address the cell by applying a pulse at A. AND3 is now enabled to Connect Q to the O/p.

Memory cell logic



Truth Table

Decoder

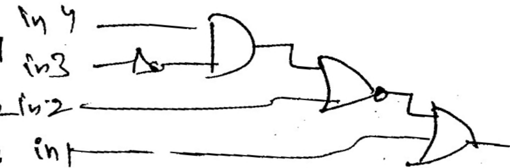
i/p	o/p
000	0000 0001
001	0000 0010
010	0000 0100
011	0000 1000
100	0001 0000
101	0010 0000
110	0100 0000
111	1000 0000

8'd1, 8'd2, 8'd3, 8'd4, 8'd5, 8'd6, 8'd7

Encoder o/p i/p

Priority Encoder

Consider



Shifter

Clock	0	1	2	3	4	5	6	7
Shift left	1001	0010	0100	1000	0000	0000	0000	0000
Shift right	1001	0100	0010	0001	0000	0000	0000	0000
Barrel shift Right	1001	1100	0110	0011	1001	1100	0110	0011
Barrel shift left	1001	0011	0110	1100	1001	0011	0110	1100

Memories

```
module my_Memory (datai, datao, clk, wr_n, addr);
  parameter width = 4;
  parameter depth = 16;
  input [width-1 : 0] datai, addr;
  input clk, wr_n, rd_n;
  output [width_1 : 0] datao;

  reg [width-1 : 0] memory [depth-1 : 0];
  reg [width-1 : 0] datao;

  always @(posedge clk)
  begin
    if( wr_n ==1'b0)
      memory [addr] <= datai;
    else if (rd_n ==0)
      datao <= memory [addr]; //synch read
  end
endmodule
```

another logic
combo read

```
assign datao = memory [addr]
```

NOTE:

①
```
always @(in1 or in2 or in3 or in4) // befor verilog2001
begin
  out1 = (in1 ^ in2) & (in3 | in4);
end
```

②
```
always @ (in1, in2, in3, in4)            // verilog 2001
  begin
    out1 = ( in1 ^ in2) & (in3 | in4);
  end
```

③
```
always @ (*)                              // verilog 2001
  begin
    out1 = ( in1^in2) & (in3 | in4);

  end.
```

Diff b/n if-else and ?: operator.

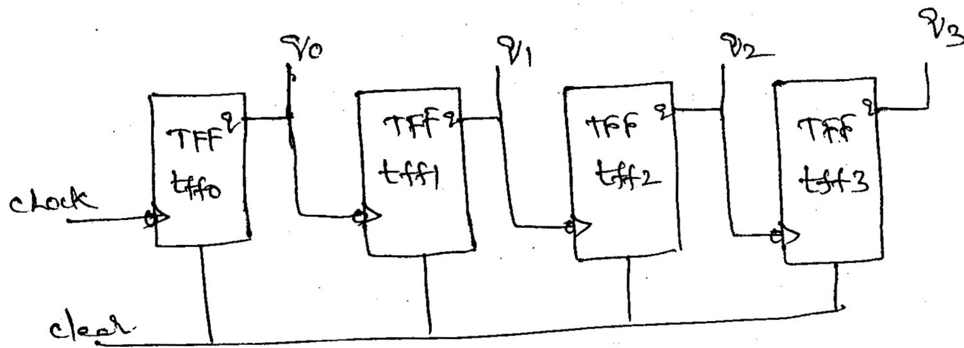
Conditional ?: operator	if-else
* Typically used in procedural (or) continuous assignments	* Typically used with in initial or always block.

* A TRUE and a FALSE expression is always required to be fully specified, that is, in EX:
`assign a=(b==0) ? c : d ;`
both the expression c and d are required
	* The else portion is optional in the if-else statement, for EX
	`if (en)`
	`q = d ; // else is not necessarily required`

* Expression to the left and right of the colon " : " can only be a single expression, that is, not a block of expressions, with in begin-end
EX:
`a = (b==0) ? c : begin d ; e;`
`end // wrong`
	* The expression is within the if-else can be block code enclosed within a begin-end.
	EX: `if(en) begin`
	`- - - // many expressions`
	`end`
	`else begin`
	`- - - // many expressions`
	`end`

* While the ?: operator is useful in specifying simple expressions, readability is an issue when it is deeply nested.	* if-else is visually more readable code in all expressions, especially when it is nested.

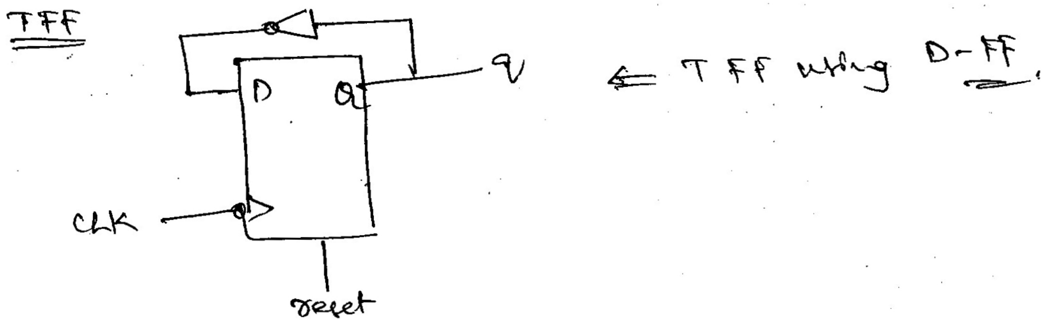
4-bit Ripple Counter



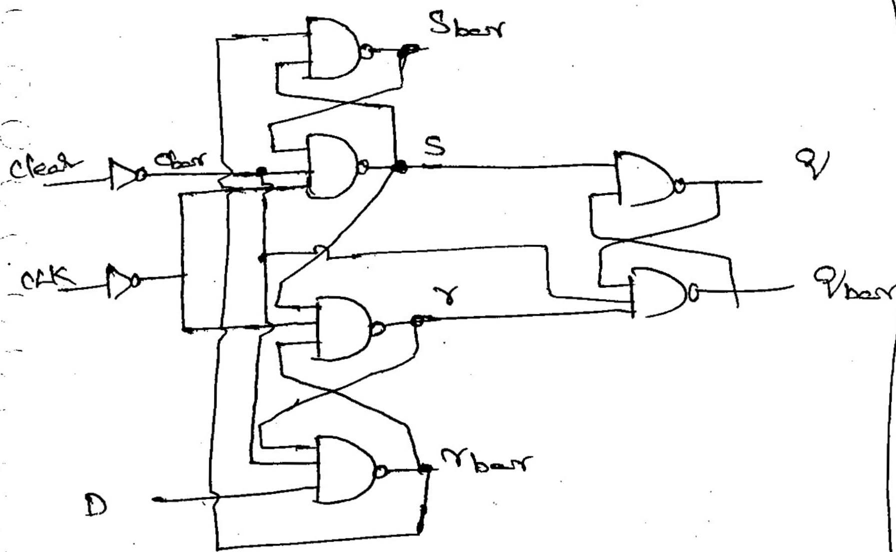
clock & clear – i/p signals.

q0, q1, q2 & q3 – o/ps.

TFF



D-FF



cbar = ~clear

Sbar = ~(rbar & S)

S = ~(Sbar & cbar & ~clk)

r = ~(rbar & ~clk & S)

rbar = ~(r & cbar & d)

q = ~(S & qbar)

qbar = ~(q & r & cbar)

Program // Ripple counter

```
module counter (Q, clock, clear);
// IO ports
  output [3:0] Q;
  input clock, clear;

  T_FF tff0( Q[0], clock, clear);
  T_FF tff1( Q[1], Q[0], clear);
  T_FF tff2( Q[2], Q[1], clear);
  T_FF tff3( Q[3], Q[2], clear);
endmodule.

// Edge triggered T-FF

  module T_FF(q, clk, clear);
    output q;
    input clk, clear;
    edge_dff FF1(q, ~q, clk, clear);
  endmodule.

// Edge Triggered D-FF
  module edge_dff (q, qbar, d, clk, clear);
      output q, qbar;
      input d, clk, clear;
      wire s, sbar, r, rbar, cbar;
      assign cbar = ~clear;
      assign sbar = ~(rbar & s),
             s = ~(sbar & cbar & ~clk),
             r = ~(rbar & ~clk & s),
             rbar = ~(r & cbar & d);

    // o/p latch
    assign q = ~(s & qbar),
           qbar = ~(q & r & cbar);
  endmodule
```

UNIT- VII

Digital logic Devices

1) Transistors are basic components of for all other devices. They are used in very simple circuits as on-off switches.

2) Primitive digital devices such as AND, OR, NOT, XOR & FF. These components encapsulated in small SCI (or) chips. And LSI are chips that encapsulate complex functions such as decoders, muxs, adders and arithmetic units.

3) VLSI are complex chips that integrate several functions and components such as processors, CPUs, controllers, communication protocols, memories and more.

4) Programmable logic Devices (PLD) are integrated ckts ~~through fuses.~~ with internal logic gates that are connected together through fuses.

* The functionality of the chip is defined by a process that is called by programming.
* The programming the chip is simply blowing some fuses b/n some of the onchip resources (gates). This operation disconnects the path b/n the resources and keeps some connections b/n others.

   Ex: ROM, PAL and PLAs.

* The main difference b/n these (PLA, PAL & ROM) devices is the position of the fuse and the fixed connection b/n gates.

NOTE:- Since the on-chip resources are connected through fuses, the programming process can be done once.

5) ASICs are custom chips builds for specific needs. There are two main kinds of ASICs, sea-of-gates and cell-based chips.

* Sea of gates chips are made of onchip gates and wires. These connections b/n the gates are defined during the fabrication of the chip.
* The custom chips are fully custom and all chip resources are defined during the fabrication of the chip.

6) CPLDs :- A more complex PLD that consists of an arrangement of multiple PLD-like blocks on a single chip.

~~* It is more like complex PALs~~

7) FPGAs : Feild programmable Gate Arrays are devices featuring a general structure that allow very high logic capacity.

Why Programmable logic ?

Ans: 1) There is no need for complex fabrication process to implement design using FPGAs. If the board exists with the FPGA component on it, only the bit stream file is needed to program the FPGA and changes its functionality.

2) It is easier to design, debug and modify FPGAs rather than ASICs.

FPGA Boards : 1. Xilinx Boards 2. Altera Boards.

Difference b/n CPLD & FPGAs.

FPGAs	CPLDs
* field programmable Gate Array	* Complex programmable logic Devices like PAL.
* It contains arrays of identical logic cells	* It contains no of macro cells that are grouped into function blocks.
* Speed : Application Dependent	* Fast, predictible.
* Interconnections are through programly inter connection (Routing).	* through interconnection array. (cross bar)
* Based on CLB configuration logic blocks	* Based on PLAs & PALs.
* large no of FF are used	* large no of FFs are not necessary.
* Applied to integrated SPLDs.	* consists of multiple SPLDs.
* HW cannot be reconfigure easily	* HW can be easily configure without power down.
* classes : Symmetrical array, row based array, hierarchical PLD, sea-of-gates.	* Power consumption is high.
* Power consumption is medium.	

NOTES :

* PLDs Types : ROM, PLA & PAL.

* A PROM (Programmable Read only memory) is one time Programmable device that consists of an array of read only cells.

* PROM Types : 1. Mask programmable 2. Field-Programmable.

Field Programmable	Mask programmable
* User can program the logic	* During manufacturing time period, the logic will be programmed.
* Programming time is with in a minute.	* Program time period of weeks (or) months.
* Program can erased & re programmed many time using EPROM, EEPROM.	* only one time programmable.
* chips are less Expensive at low volumes.	* chips are highly expensive.
* product Cost is low	* cost of product is high.

Applications of FPGAs

* currently used mask-programmed gate arrays, Below present few categories of such designs.

1) ASICs (Application specific ICs)

* FPGA is completely general medium for implementing digital logic. They are particularly implementation of ASICs.

Ex: 1 Mega bit FIFO controller,
IBM PS/2 micro channel Interface,
DRAM controller with error correction,
Printer controller and
Graphics engine, T1 n/w Transmitter / receiver, &
many telecommunication applications.

2) Implementation of Random logic

* RANDOM logic circuitry is usually implemented using PALs.
* If the speed of the ckt is not of critical concern (PALs are faster than most FPGAs), such circuitry can be implemented advantageously with FPGAs.
* Currently one FPGA can implement a ckt that might require ten to twenty PALs. In future, this factor will increase dramatically.

3) Replacement of SSI chips for Random logic

* Existing ckts in commercial products often include a no of SSI chips.
* In many cases these chips can be replaced with FPGAs, which often results in a substantial reduction in the required area on ckt boards that carry such chips.

4) Prototyping

* FPGAs are almost ideally suited for prototyping applications.

5) FPGA-Based Compute Engines

* A whole new class of computers has been made possible with the advent of in-circuit reprogrammable FPGAs. These m/cs consists of a board of such FPGAs, usually with the pins of neighboring chips connected.
* The idea is that a software program can be "compiled" (using high level, logic-level and layout level synthesis techniques or by hand) into hardware rather than software. This hardware is then implemented by programming the board of FPGAs.
* This approach has two major advantages.
  1) No instruction fetching are required by traditional μprocessor. as the hardware directly embodies the instructions. This can result in speed up of the order of 100.
  2) The computing medium can provide high levels of parallelism, resulting further speed increase.

Complex Programmable Logic Devices (CPLDs)

1. Altera MAX 7000 & MAX 9000 families
2. Atmel ATF and ATV families
3. Lattice isp LSI family
4. Lattice (Vantis) MACH family
5. xilinx XC 9500 family.

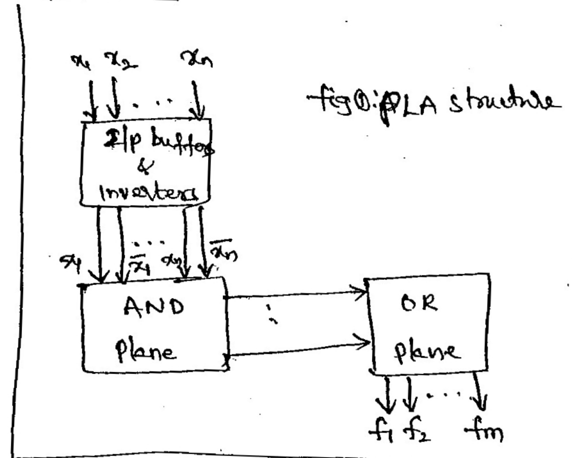


fig ①: PLA structure

Basics:

PLA :- It contains both AND plane and OR plane are programmable.

Drawback: 1. Hard to fabricate correctly
2. Reduce the speed-performance of the ckt implementation.

PAL :- It consists programmable AND plane and fixed OR plane. It have become more Popular in practical applications.

→ For PAL, OR plane is fixed and from OR gate contains extra circuit. (macro cell).

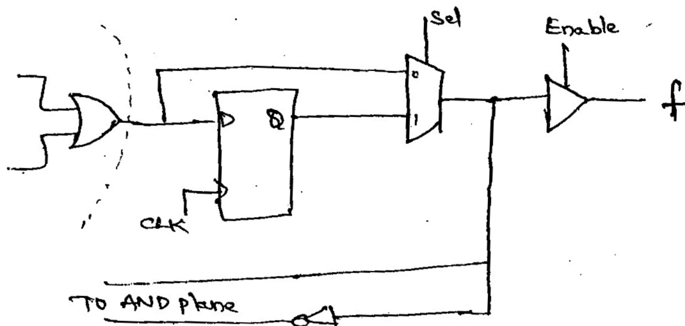


fig ②: Macro cell

→ In many PAL, this extra circuitry is added at the o/p of each OR gate to provide additional flexibility (called macro cell).

* The MUX o/p is feed back to the AND plane in the PAL. This feedback connections allows the logic function produced by the MUX to be used internally in the PAL, which allows the implementation of ckt that have multiple stages (or) levels of logic gates.

→* The PLA & PAL supports combined no of inputs plus outputs of not more than 32.

Programming of PLAs and PALs

* A Computer System that runs CAD tools is Connected by a cable to a dedicated Programming unit.
* Once the user has completed the design of a ckt, the CAD tool generate a file, often called programming file or fuse map, that specifies the state that each switch in PLD should have, to realize correctly the designed ckt.
* The PLD is placed into the programming unit, and programming file is transfered from the computer system.
* The programming unit then places the chip into a special programming mode and configures each switch individually.
* The programming procedure may take few minutues to complete.
* The programming unit can automatically "read back" the state of the switch after programming to verify that the chip has been programmed correctly.
* NOT supported in-system programming (ISP). ~~means to~~
* The ISP means to perform the programming while chip still attached to the circuit board.

CPLDs

* It contains multiple circuit blocks on a single chip, with internal wiring resources to connect the ckt blocks.
* Each block is similar to a PLA or a PAL (PAL-like blocks).
* The PAL-like blocks that are connected to a set of interconnection wires.
* Each PAL-lik block is also connected to the sub ckt labeled I/O block, which is attached to a no of chip's input and output pins.
* The CPLD typically have about 16 macrocells in a PAL-like block.
* The CPLD usually provide b/n 5 and 20 i/p to each OR gate.

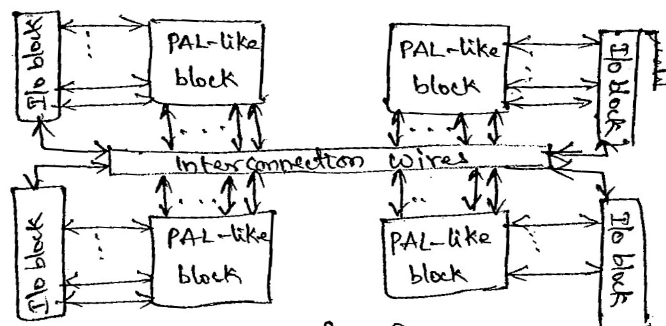
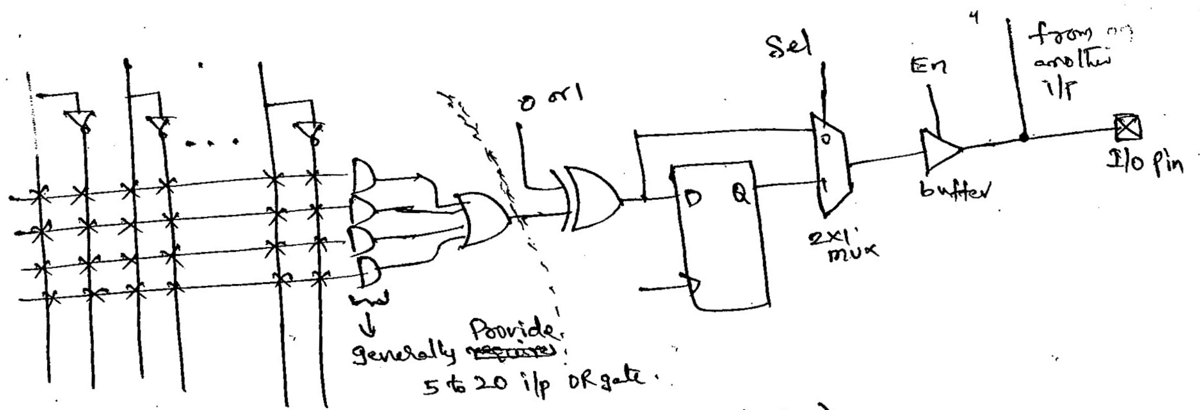


fig ③: structure of CPLD.



fig(4): PAL like block (It contains set of blocks)

* one i/p to XOR gate in fig (4) can be programmably connected to 1 or 0.
 if 1, then the XOR gate complements the OR-gate o/p.
 if 0, then XOR gate has no effect.

* The flipflop is used to store the o/p value produced by OR gate.

* Each tri-state buffer is connected to a pin on the CPLD package.

* The tri state buffer acts as a switch that allows each pin to be used either as an o/p from the CPLD or as an i/p.

→ To use pin as o/p corresponding tri-state buffer is enabled, acting as a switch that is turned on.

→ If pin is used as i/p, then tri-state buffer is disabled, acting as a switch that is turned off.

→ In this case an external source can drive a signal onto the pin which can be connected to other macro cells using the interconnection wiring.

* The interconnection wiring contains programmable switches that are used to connect the PAL-like blocks.

* Each horizontal wire can be connected to some of the vertical wires that is crosses, but not all of them.

* Extensive research has been done to decide how many switches should be provided for connection b/n wires.

NOTE: (1) when a pin is used as an i/p, the macrocell is associated with that pin cannot be used and therefore wasted.

(2) Some CPLDS include additional connection b/n the macro cell and the interconnection wiring that avoids wasting macrocells in such situations.

NOTE ①: For CPLD programming we won't use programming unit because inconvenient for larger CPLDs for two reasons.

i) More than 200 pin on the chip package, these pin are often fragile and easily bent. (easily breakable)

ii) Programmed in a programming unit, a socket is required to hold the chip. Sockets for larger QFP packages are very expensive; they some times cost more than the CPLD device itself.

NOTE ②: Once a CPLD's programmed, it remains the programmed state permanently, even when the power supply for the chips is turned off. This property known as non-volatile programming.

NOTE ③: CPLD support in-system programming (ISP) technique.

NOTE ④: QFP (Quad Flat pack) package → > 200 pins

PLCC (plastic Leaded chip carrier) package → fewer 100 pins.

* QFP package has pin on all four sides, pin extended outward from package, with down ward-curving shape.
* PLCC's package pins wrap around the edges of the package.
* QFP much thinner than PLCC.
* The PLCC package much thinner than DIP packages.

NOTE ⑤ PLA & PAL ⟶ DIP packages

CPLDs ⟶ PLCCs & QFP.

* most CPLDs contain the same type of programmable switches that are used in SPLDs.

⇒ Each macro cell contains 20 equivalent gates (NAND equivalent),

So, for SPLD contains 8 macro cells ⇒ equivalent gates are $20 \times 8 = 160$.

for CPLD " 500 macrocells ⇒ " " $20 \times 500 = 10000$

* Nowadays 10000 equivalent gates is not large. To implement larger ckts choose FPGA chips & it is convenient to use a different type of chip that has larger capacity.

Field-Programmable Gate Arrays (FPGAs)

* FPGAs are quite different from SPLDs and CPLDs because FPGAs do not contain AND or OR planes.

* FPGAs contain main three types of resources:
  1. logic-blocks
  2. I/O blocks for connecting to the Pin of the package.
  3. Interconnection wires and switches.

* The logic blocks are arranged in a two-dimensional array, and interconnection wires are organized as horizontal and vertical routing channels b/n row and columns of logic blocks.

* The routing channels contain wires and programmable switches that allow the logic blocks to be interconnected many ways.

* Programmable connections also exists b/n the I/O blocks and the interconnection wires.

* The actual no. of programmable switches and wires in an FPGA varies in commercially available chips.

** The FPGAs can be used to implement logic ckt of more than a million equivalent gate size.

* FPGA products: Altera and Xilinx.

* The FPGA chips are available in a variety of packages, including PLCC and QFP package and also Pin Grid Array (PGA) package and another Ball Grid Array (BGA) package.

⇒ A PGA package may have upto a few hundred pins total, which extended straight outward from the bottom of the package, in grid pattern.

⇒ The BGA is similar to the PGA except that pins are small round balls, instead of posts.

⇒ The Advantage of BGA package is that the pins are very small, hence more pins can be provided on a relatively small package.

* Each logic block in an FPGA typically has small no of i/ps and o/ps. This logic blocks contain different types.

* The most commonly used logic block is a Lookup Table (LUT), which contains storage cells that are used to implement a small logic function. Each cell is capable of holding a single logic value, either 0 or 1. The stored value is produced as the o/p of the storage cell.

* LUTs of various sizes may be created, where the size is defined by the no of i/ps.

LUT

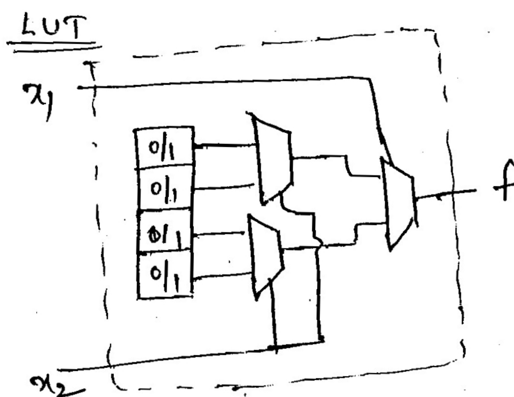


fig ① ckt for a two i/p LUT

$x_1, x_2 \rightarrow$ i/p

$f \rightarrow$ o/p

* Two variable truth-table has four rows, this LUT has four storage cells.

Example

x_1	x_2	f_1
0	0	1
0	1	0
1	0	0
1	1	1

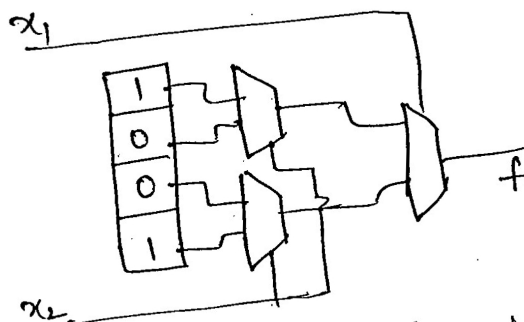


fig ② Implementation for given Truth Table.

* The one storage cell corresponds to the o/p value in each row of the truth table.

NOTE: In commercial chips, LUTs usually have either four (or) five i/ps, which requires 16 or 32 storage cells respectively.

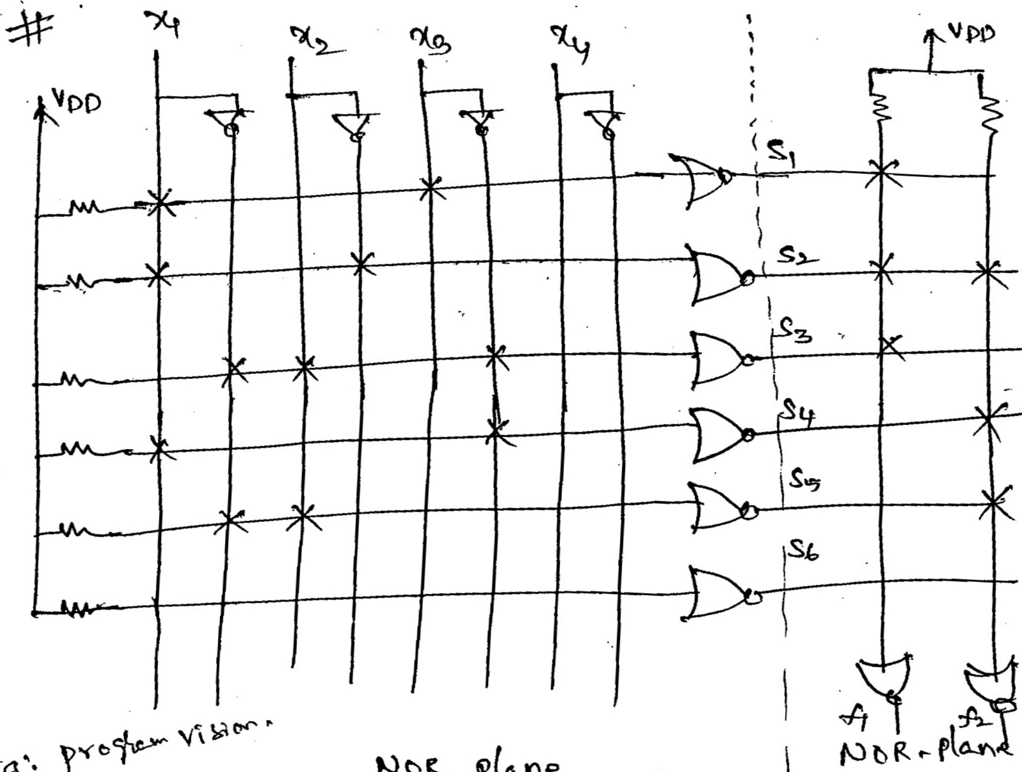
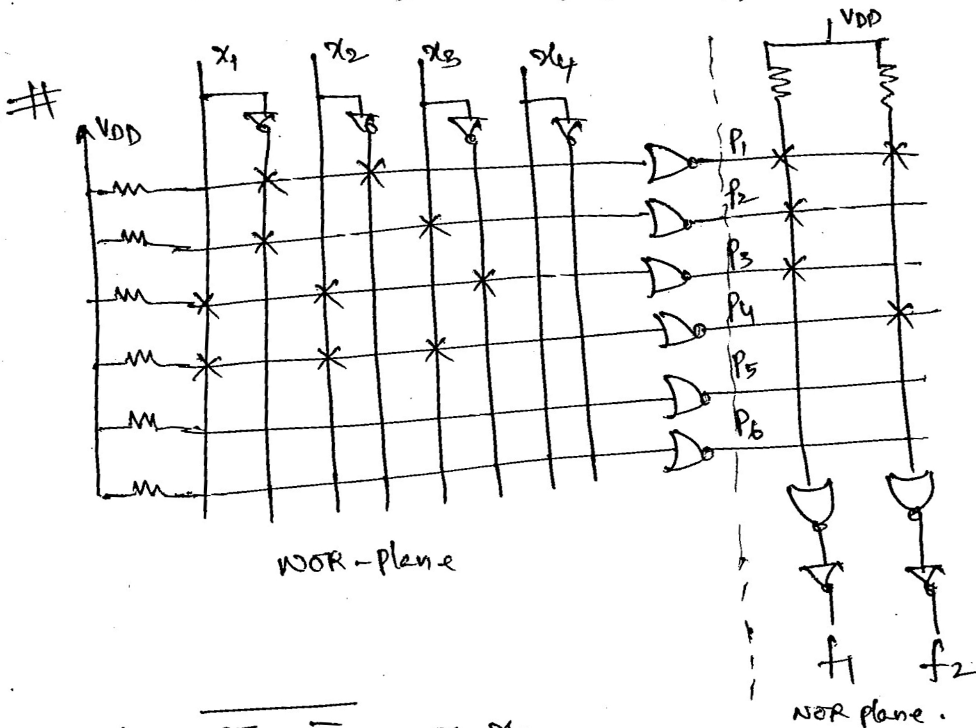


fig: program vision.

$$f_1 = (x_1 + x_3)(x_4 + \bar{x}_2)(\bar{x}_1 + x_2 + \bar{x}_3)$$

$$f_2 = (x_1 + \bar{x}_3)(\bar{x}_1 + x_2)(x_4 + \bar{x}_2)$$



$$P_1 = \overline{\bar{x}_1 + \bar{x}_2} = x_1 x_2$$

$P_2 = x_1 x_3$; $P_3 = \bar{x}_1 \bar{x}_2 x_3$; $P_4 = \bar{x}_1 \bar{x}_2 \bar{x}_3$

$$f_1 = P_1 + P_2 + P_3 = x_1 x_3 + x_1 x_2 + \bar{x}_1 \bar{x}_2 x_3$$

$$f_2 = P_1 + P_4 = x_1 x_2 + \bar{x}_1 \bar{x}_2 \bar{x}_3$$

* From fig EEPROM, the floating gate extends down words so that it is Very close to the top surface of the channel.

* A high current flowing through the channel causes an effect, known as <u>Fowler-Nordheim tunniling</u>, in which some of electrons in the channel "tunnel" through the insulating glass is thinnest point and become trapped under the floating gate.

* After the programming process is completed, the trapped electrons repel other electrons from entering channel.

* When $V_e = 5V$ is applied to the EEPROM transistor, which would normally cause it to turn on, the trapped electrons keep the transistor turned off. Hence NOR plane in fig, programming is used to "disconnect" inputs from the NOR gates.

* For the I/ps that should be connected to each NOR gate, the corresponding EEPROM transistors are left in the unprogrammed state.

→ For erasing step, we have to use opposite polarity of voltages for programming. In this case, the applied voltage causes the electro that are ~~applied~~ trapped under floating gate to tunnel back to the channel. The EEPROM transistor returns orginal state and again acts like a normal NMOS transister.

⇒ For <u>EPROM</u> programming is similar to EEPROM. but, erase process is it must be exposed to light energy of specific wavelengths. To facilitates this process, chip based on EPROM technology are housed in packages with a clear glass window through which the chip is visible. To erase chip, it is placed under an <u>ultraviolet light</u> source for several minutes. Because erasure of EPROM transistors is more awkward than the electrical process used to erase EEPROM transistors.

PLA Implementation-using EEPROM transistor

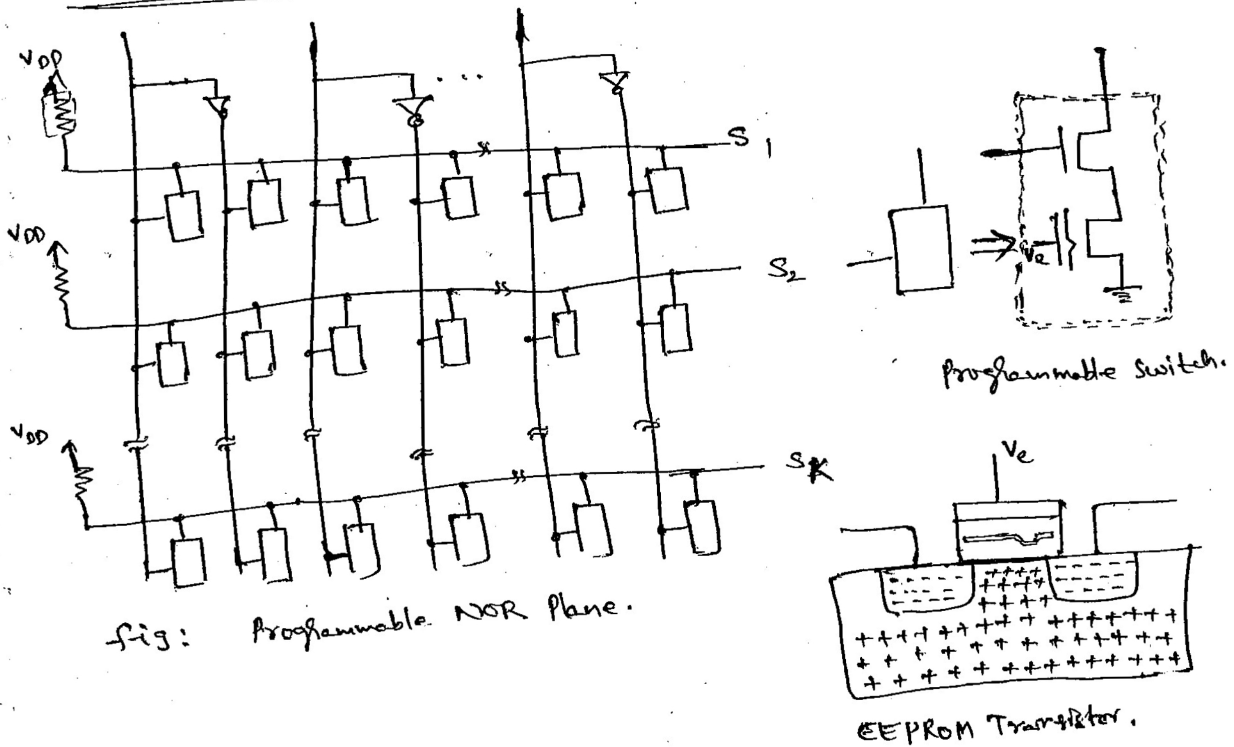


fig: Programmable NOR Plane.

* Now a days common usage of programmable switch is EEPROM Transistor.
* The switch consists two transistors connected in series, an NMOS transistor and an electrically erasable programmable read-only memory (EEPROM) transistor.
* The EEPROM Transistor has two gates:
  1. Normal gate that an NMOS transistor
  2. floating - gate.
* * The floating - gate is named because it is surrounded by insulating glass and it is not connected to any part of the transistor.
* When the transistor is in the original unprogrammed state, the floating gate has no effect on transistor's operation and it works as a normal NMOS transistor.
* During normal use of the PLA, the voltage turn on the floating gate Ve is set to VDD by circutary and the EEPROM transistor is turned on. (typically Ve = 12V for turning on the transistor with higher than normal voltage level, which causes a large amount of current to flow through the transistor channel.

Implementation Details for SPLDs, CPLDs and FPGAs.

X → Symbol represents programmable switch in PLA or PAL, SPLDs.

* In commercial SPLDs two main technologies are used to manufacture the programmable switches.

① The oldest technology is based on using metal-alloy fuses as programmable links.

② Programmable switch implemented using programmable transistor.

NOTE ①: Oldest technology — metal-alloy.

* The metal-alloy fuses; In this technology PLAs and PALs are manufactured so that each pair of horizontal and vertical wires that cross is connected by a small metal fuse. When chip is programmed, for every connection that is not wanted in the circuit being implented, the associated fuse is melted. This programming process is not reversible, because the melted fuses are destroyed. So, now a days we are not used this technology.

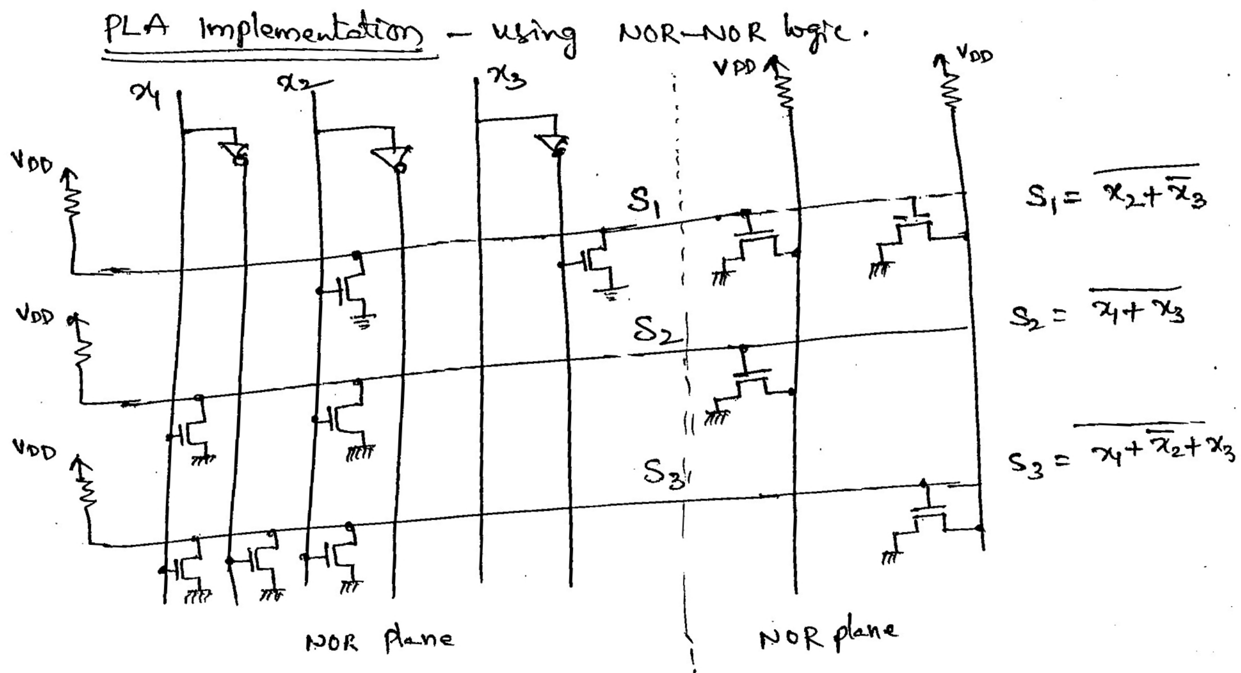
NOTE ②: New technology — programmable transistor.

* This technology is used in SPLDs is also applicable to CPLDs.

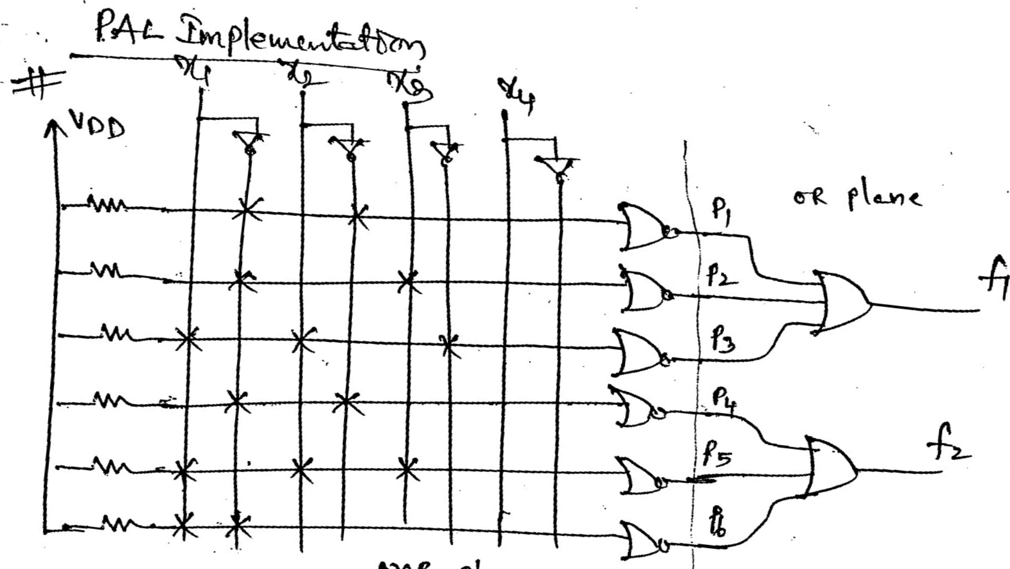
* The PLA to be useful for implementing a wide range of logic functions, it should support both functions of only few variables and functions of many variables.

* The fan-in high, the best type of gate to use is the NMOS NOR gate. Hence PLAs are usually based on this type.

PLA Implementation — using NOR-NOR logic.



PAL Implementation



$$P_1 = \left(= \overline{\bar{x}_1 + \bar{x}_2} = x_1 x_2 \right) = x_1 x_2$$

$$P_2 = \overline{\bar{x}_1 + x_3} = x_1 \bar{x}_3$$

$$P_3 = \bar{x}_1 \bar{x}_2 x_3$$

$$P_4 = x_1 x_2$$

$$P_5 = \bar{x}_1 \bar{x}_2 \bar{x}_3$$

$$P_6 = x_1 \bar{x}_1 = 0$$

$$\therefore f_1 = P_1 + P_2 + P_3 = x_1 x_2 + x_1 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3$$

$$f_2 = P_4 + P_5 + P_6 = x_1 x_2 + \bar{x}_1 \bar{x}_2 \bar{x}_3 .$$

Implementation of FPGA

* FPGAs do not use EEPROM technology to implement the programmable switches. Instead the programming information is stored in memory cells, called SRAM cells.
* An SRAM cell is used for each truth-table value stored in LUT. SRAM cells are also used to configure the interconnection wires in an FPGA.

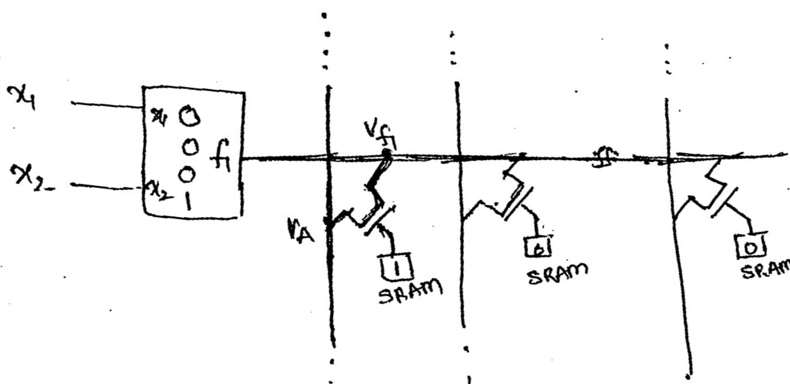


fig: Pass transistor switches in FPGAs.

⇒ The logic block shown produces the o/p f_1, which is driven onto the horizontal wire draw in Red. This wire can be connected to some of the vertical wires that it crosses, using programmable switches.

* Each switch is implemented using NMOS transistor, with its gate terminal controlled by a SRAM cell. Such a switch is known as a pass-transistor switch.

* If a 0 is stored in an SRAM cell, then the associated NMOS transistor is turned off. But if a 1 is stored in an SRAM cell, then the associated NMOS transistor is turned ~~off~~ ON. This switch forms a connection b/n the two wires attached to its source and drain terminals.

* The no of switches that are provided in the FPGA depends on the specific chip architecture.

<u>Switch types</u>: 1. Pass transistor
2. tri-state buffers
3. multiplexer
* Transmission gate.

#

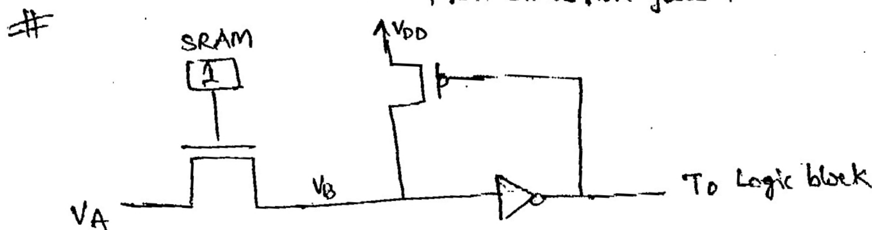


fig: Restoring a high voltage level.

* Assume V_A signal passes through another pass transistor switch before reaching its destination at another logic block.

* The signal V_B has the same value as V_A because the threshold voltage drop occurs only when passing through the first pass-transistor switch

* To restore the level of V_B, it is buffered with an inverter. A PMOS transistor is connected b/n the i/p of the inverter and V_{DD}, and transistor is controlled by the inverter's o/p.

* The PMOS transistor has no effect on the inverter's o/p voltage level when $V_B = 0V$. But when $V_B = 3.5V$ then the inverter o/p is low, which turns on PMOS transistor. ~~This transistor quickly restores~~

This (PMOS) transistor quickly restores V_B to the proper level of V_{DD}, thus preventing current from flowing in the steady state.

* Instead of using this pull-up transistor solution, another possible approach is to alter the threshold voltage of PMOS transistor (during the integrated ckt manufacturing process) in the inverter above fig: such that magnitude of its threshold voltage is large enough to keep the transistor turned off when $V_B = 3.5V$.

* In commercial FPGAs both (SRAM & Restoring high voltage level switch) of these solutions are used in different chips.

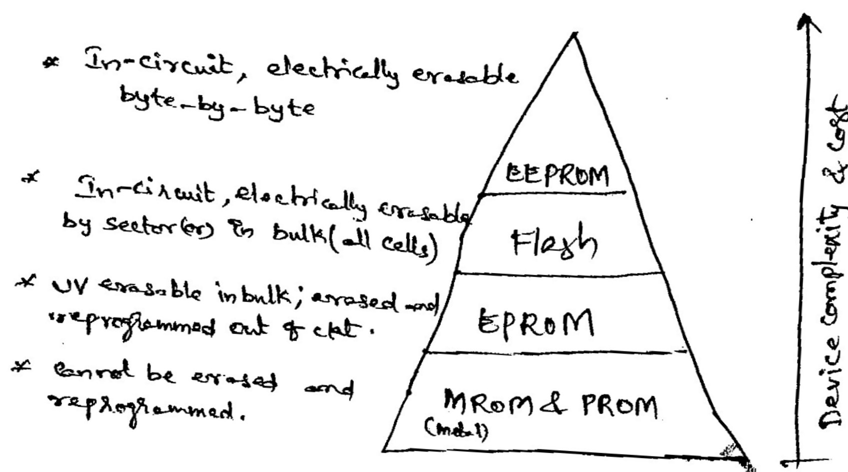
NOTE: NMOS, PMOS & Transmission gates ⇒ Voltage level problem

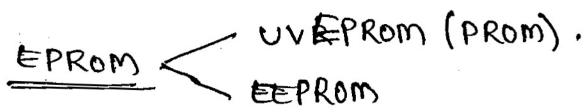
* Each switch has two drawbacks.

1. Both NMOS & PMOS transistor in the switch increases the capacitive loading on the interconnection wires, which increases the propagation delays and power consumption.
2. The transmission gate take more chip area than does a single NMOS transistor.

Imp: Commercial FPGA chips do not currently use transmission gate switches.

Trade off for semiconductor non volatile memories show that the complexity and cost as erase and programming flexibilities.





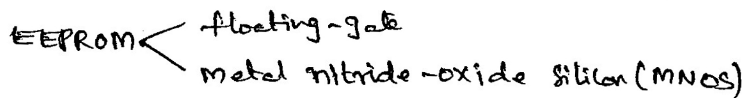
* EPROM uses an NMOS array with an isolated-gate structure. This isolated transistor has no electrical connections and can store an electrical charge for indefinite periods of time. The data bits in this type of array are represented by the presence or absence of a storage gate charge. Ensure of a data bits is a process that removes the gate charge.

UVPROMs

* The isolated gate in the FET of an ultra violet EPROM is "floating" with oxide insulating meterial.
* The programming process causes an electrons to be removed from the floating gate. Ensure is done by exposure of the memory array chip to high-intensity ultra-violate radiation through Quartz window on top of the package.
* The positive charge stored on the gate is neutralized after several minitues to an hour of exposure time.

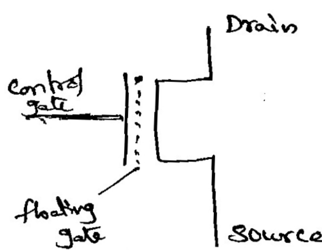
EEPROM * It can be erased & program with electrical pulses.

* It is in-circuit programming.

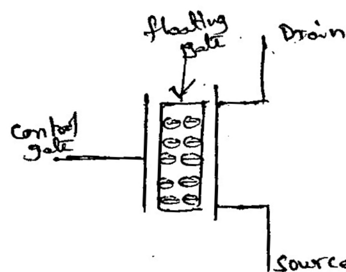


* The application of a voltage on the control gate in floating gate structure permits the storage and removed charge from the floating gate.

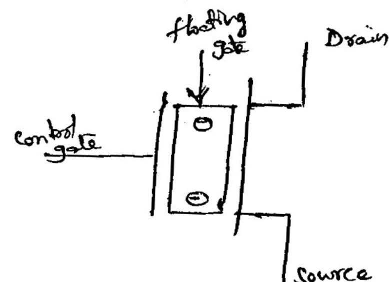
Flash memory



1) Symbol



2) Many electrons = more charge = Stored '0'



3) few electrons = less charge = stored '1'

→ For programming initializing all cells are at the '1' state because charge was removed from each cell in a previous erase operation. The programming operation add an electrons (charge) to the floating gate of those cells that are to store a '0'.

Programming

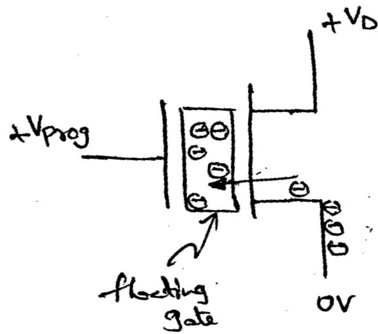


fig (3): To store a '0', a sufficient positive voltage is applied to the control gate with respect to the source to add charge to the floating gate during programming.

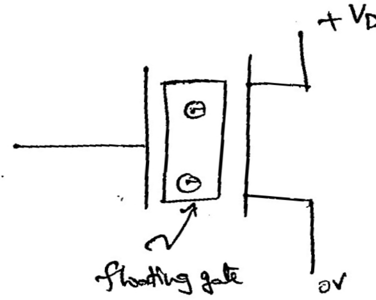


fig (4): To store '1' no charge no charge is added and the cell is left in the erased condition.

Reading

⇒ For read operation; a positive voltage applied to the control gate.

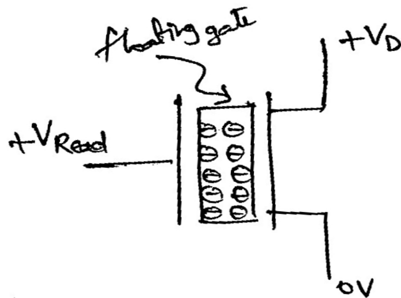


fig (5): When '0' is read, the transistor remains off because the charge on the floating gate prevents the read voltage from exceeding the turn-on threshold.

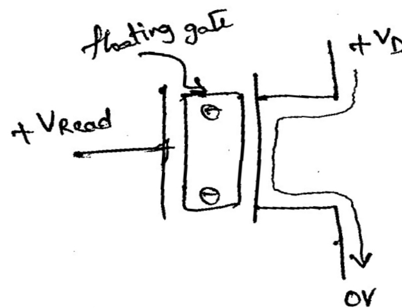


fig (6): When a 1 is read, the transistor turns on because the absence of charge on the floating gate allows the read voltage to exceed the turnon threshold.

Erase

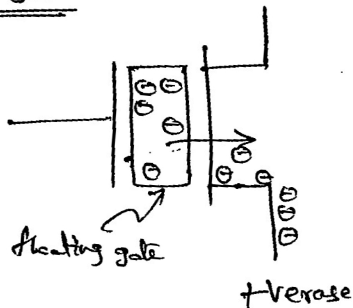


fig (6): To erase a cell, a sufficient positive voltage is applied to the source with respect to the control gate to remove charge from the floating gate during the erase operation.

NOTE:

* A flash memory is always erased. Prior to being reprogrammed.
* Once programmed, a cell can retain the charge for upto 100 years without any external power.

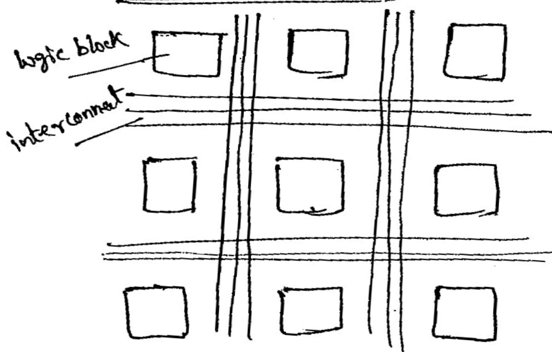
Programming Technologies

1) Static RAM Technology
2) Anti-fuse Programming Technology
3) EPROM & EEPROM programming Technology
4)

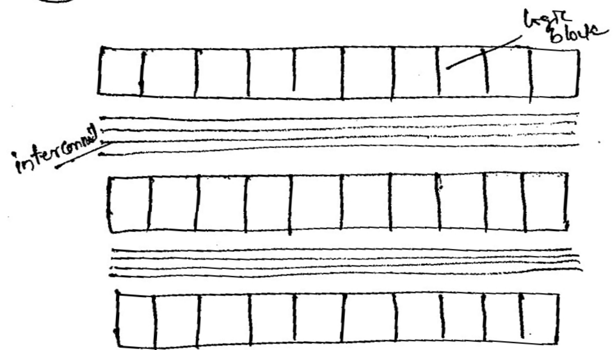
General Architectures of FPGA & CPLD

1) Symmetrical Array
2) Row-based
3) Hierarchical PLD
4) Sea-of-gates.

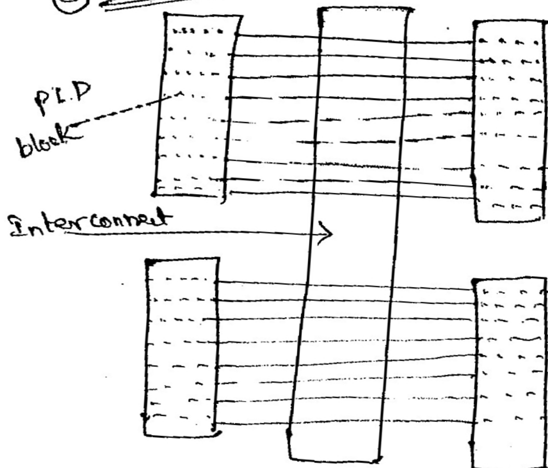
① Symmetrical Array



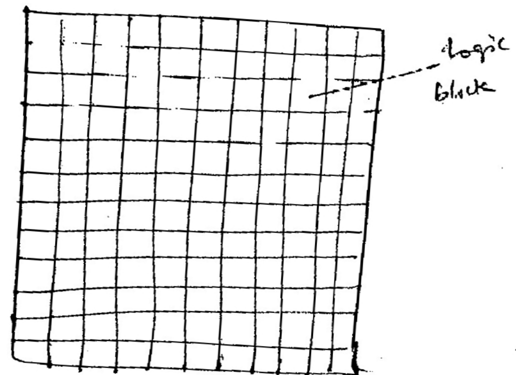
② Row-based



③ Hierarchal PLD



④ Sea of gates



interconnect overlayed on logic blocks.

Programming Technologies

* The word "switch" refers to the entities that allow programmable connections b/n wire segments. This can also called programming element.

* Programming Technologies that are currently in use in commercial products are: Static RAM cells, anti-fuse, EPROM transistors and EEPROM transistors.

→ The programming elements are used to implement the programmable connections among the FPGA's logic block, and a typical FPGA may contain more than 100,000 programming elements. For these reasons, the element should have following properties:

* the programming element should consume as little chip area as possible.

* the programming element should have low ON resistance and a very high OFF resistance.

* the programming element should contribute low parasitic capacitance to the wiring resources to which it is attached, and

* it should be possible to reliably fabricate a large no of programming element on a single chip.

→ Depending upon the application in which the FPGA is to be used, it may also be desirable for the programming element to possess other features.

EX: a programming element that is non-volatile might be attractive, as well as an element that is re-programmable. Reprogrammable elements make it possible to re-configure the FPGA, perhaps without even removing from the circuit board. Finally, in terms of ease of manufacture, it might be desirable if the programming element can be produced using a standard CMOS technology.

Static RAM Technology

- The static RAM programming technology is used in FPGAs produced by several companies: Algotronix, Concurrent logic, Plessey semiconductors and Xilinx.

- In these FPGAs, programmable connections are made using pass-transistors, transmission gates or multiplexers that are all controlled by SRAM cells.

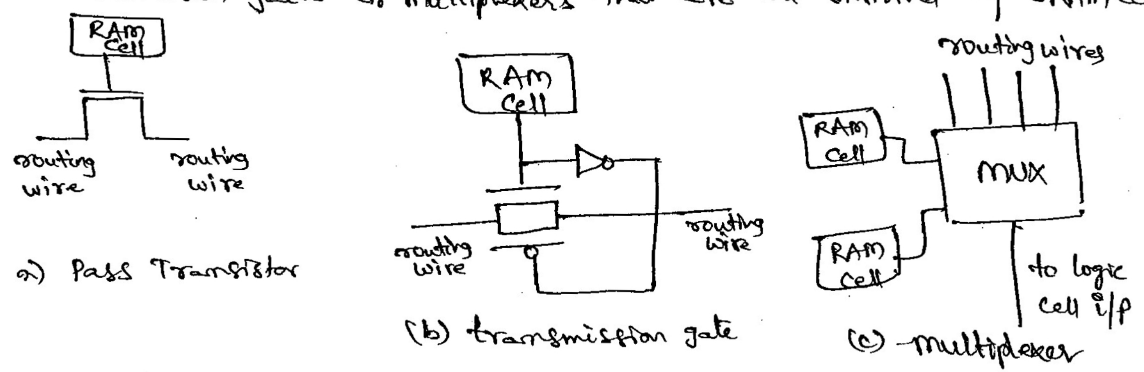


Fig ① Static RAM Programming Technology.

In Fig① (a) and (b) the RAM cell controls whether the pass gates are on or off. When off, the pass gate presents a very high resistance b/n the two wires to which it is attached.

→ When pass gate is turned on, it forms a relatively low resistance connection b/n two wires.

→ For multiplexer approach, fig ①(c), the RAM cells controls which of the multiplexer's i/p should be connected to its o/p. This scheme would typically be used to optionally connect one of several wires to a single i/p of a logic block.

→ The logic blocks may be interconnected using a combination of pass-gates and multiplexers.

→ Static RAM is volatile, area is more, at least five transistors needed for each RAM cell & permanent storage cell.

→ The major advantage of this technology is that provides an FPGA that can be reconfigured (in-circuit) very quickly and it can be produced using a standard CMOS technology.

(2) EPROM and EEPROM Programming Technology

* This technology used Altera Corp. and plus logic.
* This technology is the same as that used in EPROM memories.
* Unlike a simple MOS transistor, an EPROM transistor comprises two gates, a floating gate and a select gate.

→ The floating gate positioned b/w the select gate and transistor channel, is so named because it is not electrically connected to any circuitry.

→ In its normal (unprogrammed) state, no charge exists on the floating gate and transistor can be turned ON in the normal fashion using the select gate.

→ However when the transistor is programmed by causing a large current to flow b/w the source and drain, a charge is trapped under the floating gate.

→ This charge has the effect of permanently turning the transistor OFF. In this way, the EPROM transistor can function as programmable element.

→ An EPROM transistor can be reprogrammed by first removing the trapped charge from the floating gate.

→ Exposing the gate to ultra violete light excites the trapped electrons to the point where they can pass through the gate oxide into the substrate.

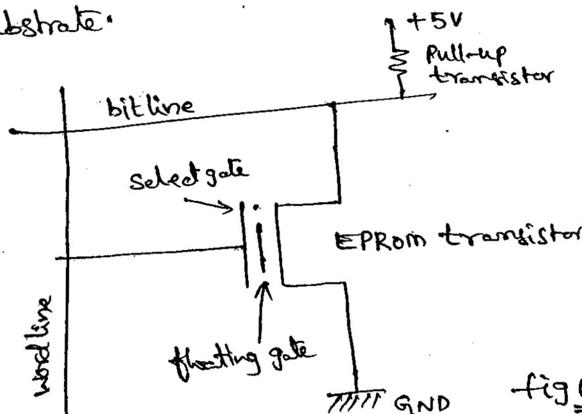


fig ② EPROM programming technology

* The EPROM connected two wires, and used as pull-down transistor, for logic block input.

* In Fig (2),

one wire is called the "wordline" (using memory technology) is connected to the select gate of EPROM transistor.

→ As long as the transistor has not been programmed into the OFF state, the word line can cause the "bit line", which is connected to a logic block input to be pulled to logic zero.

NOTE! In many EPROM transistors, each driven by a different word line, are connected to the same bit line. Since a pull-up tr is present on the bit line, this scheme allows EPROM tr to not only implement connections but also to realize wired-AND logic functions.

* Advantage :- re-programmable but not require external storage.
  - Can not be re-programmed in-circuit but in static RAM we can re-programmed in-circuit.

DisAdvantage: the resistor consumes static power.

* In EE PROM approach (offered by AMD) is similar to the EPROM.

Adv

It can be re-programmed in-circuit.

DisAdv

- Consume the chip area is twice as EPROM transistors.
- require multiple voltage sources (for re-programming)

MULTI LEVEL Synthesis

* The logic function has two level stages (1) SOP (2) POS

1. SOP ⇒ first level comprises AND gates that are connected to second level OR gate

2. POS ⇒ first level OR gates feeds the second level AND gate.

* Assume both (SOP & POS) true and complemented version of the i/p variables are available so that NOT gates are not complement the variables.

Consider $f(x_1, x_2 \ldots x_7) = x_1 x_3 \bar{x}_6 + x_1 x_4 x_5 \bar{x}_6 + x_2 x_3 x_7 + x_2 x_4 x_5 x_7$

This function we can implement easily using CPLD.

But in FPGA can also implement, using 2-i/p LUTs.

If SOP expression for f requires three- and four-input AND operation and a four-input OR, it can not be directly implemented in this FPGA. The problem is that the fan-in required to implement the function is too high for our target chip architecture.

→ To solve fan-in problem, f must be expressed in a form that has more than two levels of logic operations. Such a form is called a multi level logic expression.

⇒ Different approaches for synthesis of multilevel ckts.

Here we will discuss
1. factoring
2. functional decomposition.

(1) Factoring

$$f = x_1 \bar{x}_6 (x_3 + x_4 x_5) + x_2 x_7 (x_3 + x_4 x_5)$$

$$= (x_1 \bar{x}_6 + x_2 x_7)(x_3 + x_4 x_5).$$

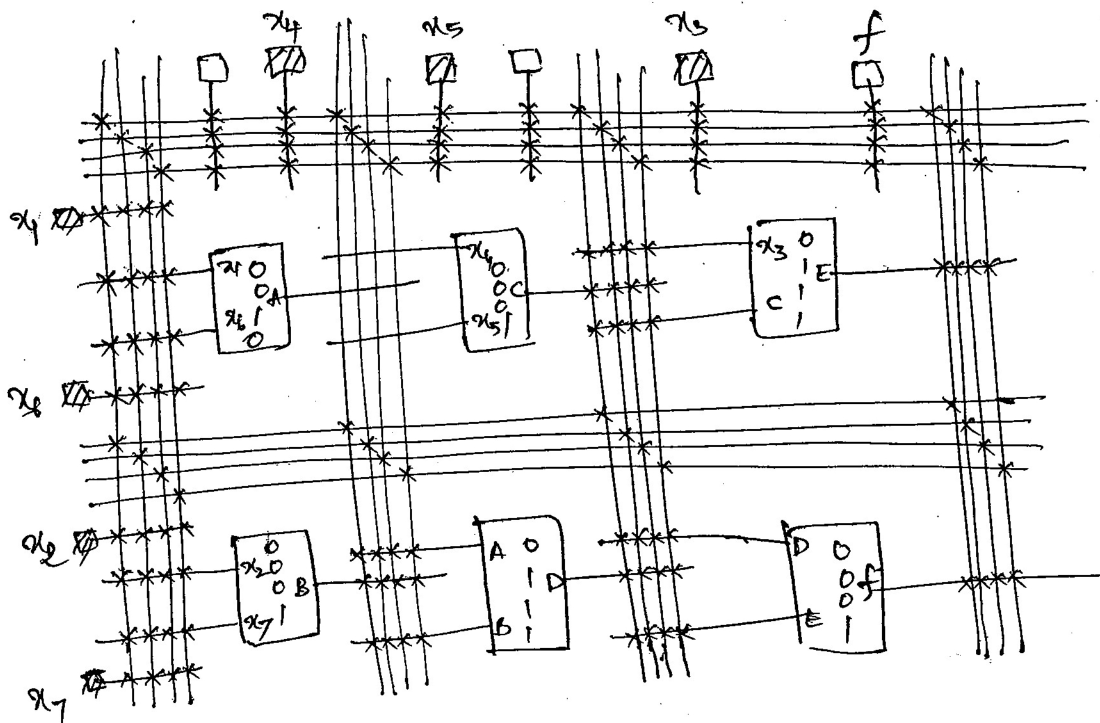


Fig: Implementation in an FPGA.

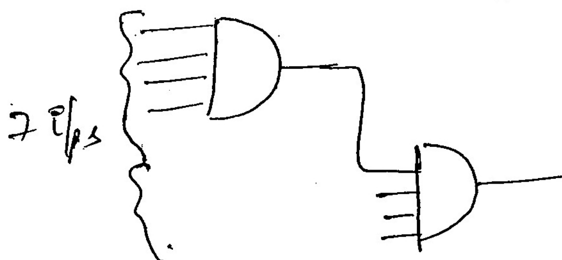
Fan-in Problem

* The fan-in restrictions were caused by the fixed structure of FPGA, where each LUT has only two i/ps. However even when target chip is not fixed, the fan-in may still be an issue.

$$f = x_1 \bar{x}_2 x_3 \bar{x}_4 x_5 x_6 + x_1 x_2 \bar{x}_3 \bar{x}_4 \bar{x}_5 x_6$$

$$= x_1 \bar{x}_4 x_6 (\bar{x}_2 x_3 x_5 + x_2 \bar{x}_3 \bar{x}_5)$$

(= 3 AND + 1 OR gates)



Functional Decomposition

* Complexity of a logic ckt, in terms of wiring and logic gates, can often be reduced by decomposing a two-level ckt into subckts, where one or more subcircuits implement functions that may be used in several places to construct the final ckt.

Consider minimum cost SOP expression

$$f = \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 x_4 + \bar{x}_1 \bar{x}_2 x_4$$

sol: The function consists 4 AND

3 OR

2 NOT

18 i/p (wires) to all gates.

* The fan-in is three for AND gates

four for OR "

* The reader should observe that this case we have included the cost NOT gate needed to complement x_1 & x_2.

⇒ factoring x_3 from first two & x_4 from last two

$$f = x_3 (\bar{x}_1 x_2 + x_1 \bar{x}_2) + (x_1 x_2 + \bar{x}_1 \bar{x}_2) x_4$$

let $g = \bar{x}_1 x_2 + x_1 \bar{x}_2$

$$\Rightarrow \bar{g} = x_1 x_2 + \bar{x}_1 \bar{x}_2$$

$$\therefore f = g x_3 + \bar{g} x_4$$

Here fan in has been reduced to two and only 15 i/ps.

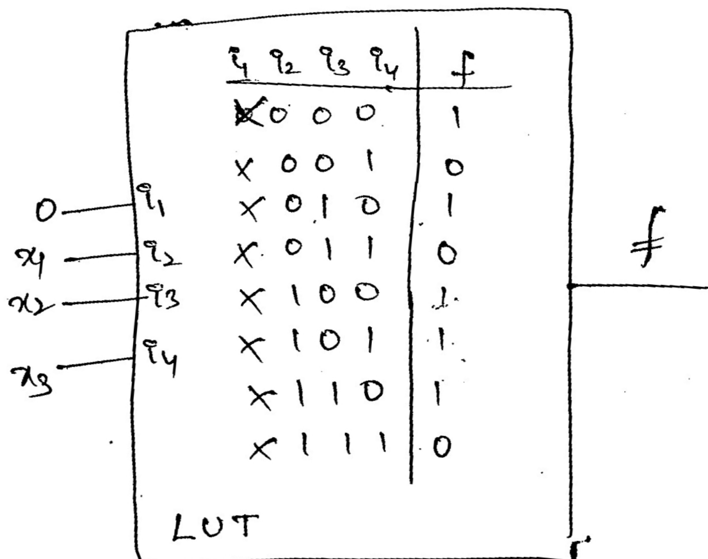
The cost of this ckt is lower than the cost of its two-level equivalent. The trade-off is an increased propagation delay because the ckt has three or more levels of logic.

The SOP $f = \bar{x}_3 + x_1\bar{x}_2$ implement Verilog code in CPLD & FPGA

```
module example1 (x1, x2, x3, f);
  input x1, x2, x3;
  output f;
  assign f = (~x1 & ~x2 & ~x3) | (~x1 & x2 & ~x3) |
             (x1 & ~x2 & ~x3) | (x1 & ~x2 & x3) |
             (x1 & x2 & ~x3);
endmodule
```

But in FPGA

we have to derive LUT. If we consider four-i/p Look up tables,



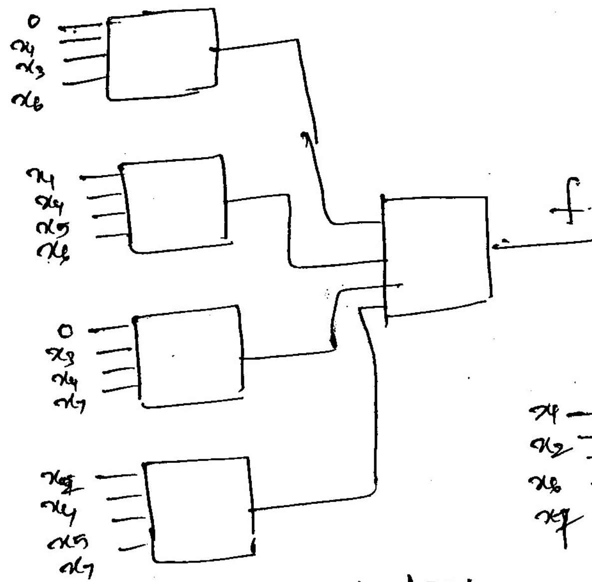
$f = \bar{x}_2 x_3 + x_1 \bar{x}_3 x_4$ Write a Verilog code.

```
module example2 (x1, x2, x3, x4, f);
  inputs x1, x2, x3, x4;
  output f;
  assign f = (~x1 & ~x2 & x3 & ~x4) | (~x1 & ~x2 & x3 & x4) |
             (x1 & ~x2 & ~x3 & x4) | (x1 & ~x2 & x3 & ~x4)
             | (x1 & ~x2 & x3 & x4) | (x1 & x2 & ~x3 & x4);
endmodule.
```

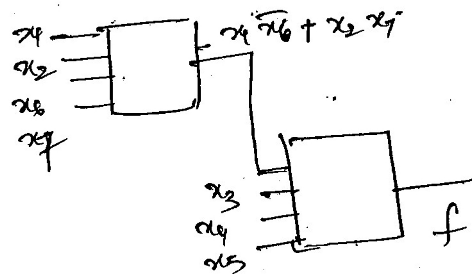
$f = (x_1 \bar{x}_6 + x_2 x_7)(x_3 + x_4 x_5)$

for implementation using CPLD ⇒ we need one macro cell

" " FPGA ⇒ we need more than one LUT. (4-i/p LUTs). because there are seven i/ps.



fig① SOP realization.



fig② factored realization.

* One LUT produces the term $S = x_1 \bar{x}_6 + x_2 x_7$

The other LUT implement four i/p function $f = S x_3 + S x_4 x_5$.

```
module example3 (x1, x2, x3, x4, x5, x6, x7, f)
    input x1, x2, x3, x4, x5, x6, x7;
    output f;
    assign f = (x1 & x3 & ~x6) | (x1 & x4 & x5 & ~x6) |
               (x2 & x3 & x7) | (x2 & x4 & x5 & x7);

endmodule
```