Basics Covered from Zero Level

NPTEL COURSE

VERILOG NOTES

Made by:
Harshit Gupta
Verification Engineer @INTEL

verilog :- It is a language using which a designer can specify the behaviour or the functionality of the structure of some given hardware or specified hardware ckt.

(Hardware Description language) → basic hardware building blocks ⇓ chip or IC

VLSI Design Process :-

Design complexity increasing rapidly
- increased size and complexity
- fabrication technology improving
- CAD tools are essential
- conflicting requirements like area, speed and energy consumption.

The present Trend :-
- standardize the design flow
- Emphasis on low power design and increased performance.

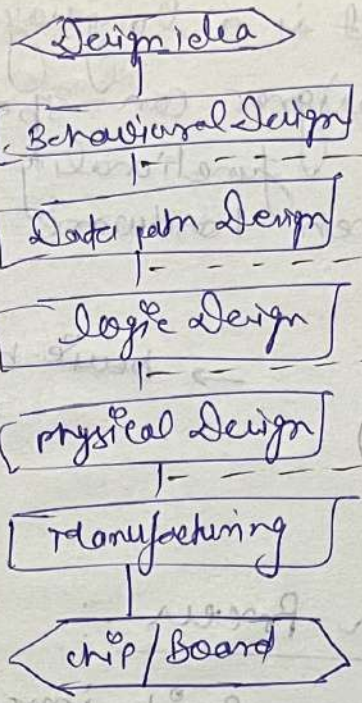
Hardware description languages (HDL's) -

↓ provides formats for representing the outputs of various design steps.

→ A CAD tool transforms its HDL i/p into a HDL o/p that contains more detailed information about the hardware.

Two competing HDLs → (1) verilog (2) VHDL (3) SystemC (4) systemverilog ! etc.

Simplistic view of Design flow



Behavioral Design :- specify the functionality of the design in terms of its behaviour.

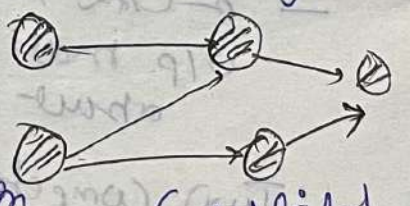
→ various ways of specifying :-
- Boolean expression or truth table.
- finite state machine behaviour
- In the form of a high-level algorithm.

→ Needs to be synthesized into more detailed specifications for hardware realization.

Data path Design :-

Generate a netlist of register transfer level components like registers, adders, multiplexers, multipliers, decoders etc.

→ A Netlist is a directed graph, where the vertices indicate components, and the edges indicate interconnections.



→ A netlist specification is also refered to as structural design.
- Netlist may be specified at various levels, where the components may be functional modules, gates or transisters.
- systematically transformed from one level to the next.

Logic Design -

- Generate a netlist of gates / flip-flops or standard cells.

→ A standard cell is a pre-designed ckt module (like gates, flip-flops, multiplexer etc) at the layout level.

→ various logic optimization techniques are used to obtain a cost effective design.

→ There may be conflicting requirements during optimization

→ min no of gates
→ minimize no of gate levels (i.e. delay)
→ minimize signal transition activities (dynamic Power)

physical Design and Manufacturing -

Generate the final layout that can be sent for fabrication.

→ The layout contains a large no of regular geometric shapes corresponding to the different fabrication layers.

→ Alternatively, the final target may be field programmable Gate Array (FPGA), where technology mapping from the gate level netlist is used.

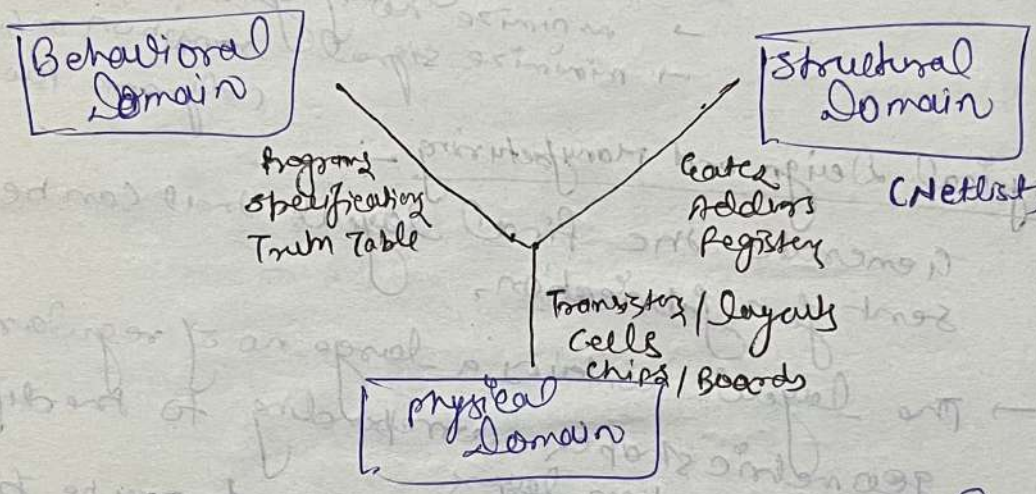
→ can be programmed in-field.
→ much greater flexibility, but less speed.

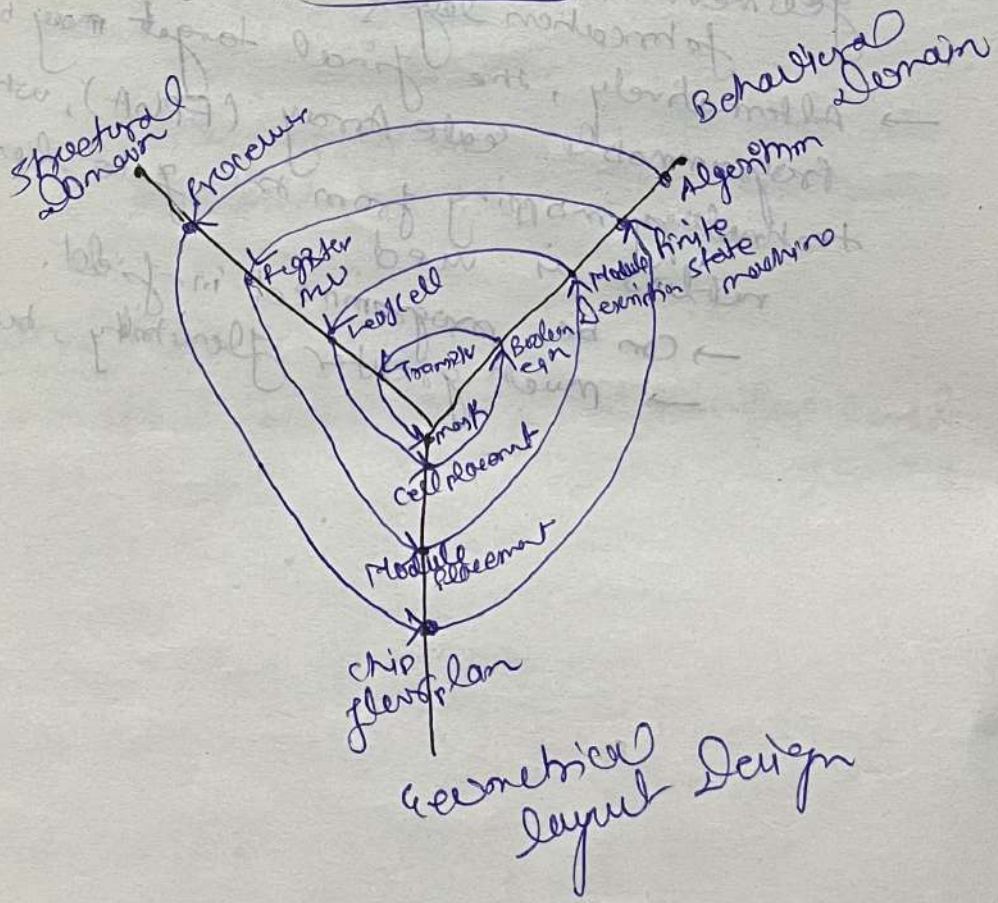
Design Representation :-

A design can be represented at various levels from 2 or 3 different points of view :-

1. Behavioral
2. Structural
3. Physical

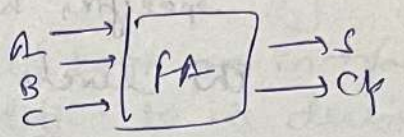
Can be conveniently expressed by Y-diagram





Behavioural Representation – example –

Full Adder



→ two operands i/p A and B
→ a carry input C
→ a carry output Cy
→ a sum output S

Express in terms of Boolean expression.

$$S = A.B'.C' + A'.B'.C + A'.B.C' + A.B.C$$
$$= A \oplus B \oplus C$$

$$Cy = A.B + A.C + B.C$$

Express in verilog –

```
Module carry (S, Cy, A, B, C);
    input A, B, C ;
    output S, Cy;
    assign S = A ^ B ^ C;
    assign Cy = (A & B) | (B & C) |
                (C & A);
end module.
```

Express in verilog in terms of truth table (only Cy is shown)

```
primitive carry (Cy, A, B, C);
    input A, B, C;
    output Cy;
    table
    //   A  B  C     Cy
         1  1  ?  :  1;
         1  ?  1  :  1;
         ?  1  1  :  1;
         0  0  ?  :  0;
         0  ?  0  :  0;
         ?  0  0  :  0;
    endtable
endprimitive
```

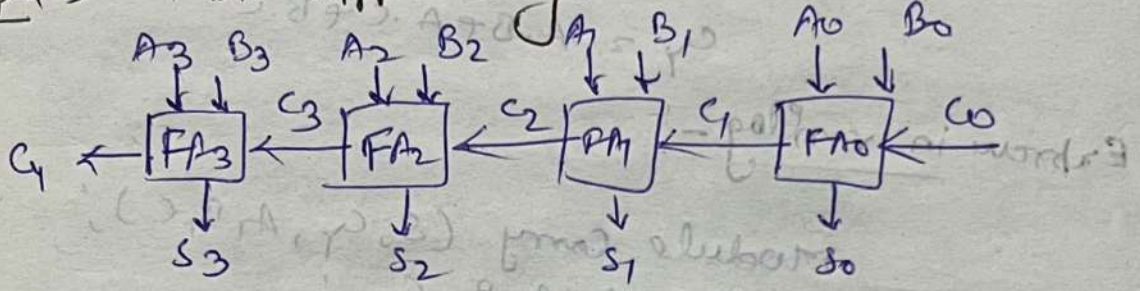
Structural Representation –

Specifies how components are interconnected.

The levels of abstraction –

- → The module level
- → The gate level
- → The transistor level
- → The combination of above.

Example → 4 bit Ripple carry Adder



- → Consists of 4 full Adder
- → Each full Adder consists of a sum circuit and a carry circuit.

why do we use verilog ?

To describe a digital system as a set of modules.

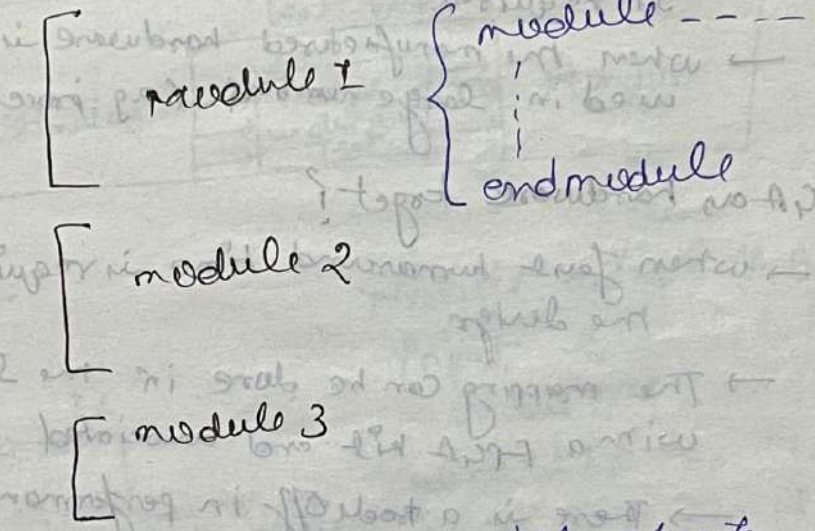
→ Each of the modules will have an interface to other modules, in addition to its description.

→ Two ways to specify a module :-

(a) By specifying its internal logical structure (called structural representation)

(b) By describing its behaviour in a program like manner (called behavioral representation)

→ The modules are interconnected wire nets which allow them to work with each other.



(All the modules together will specify the specification of the Hardware)

→ After specifying the system in verilog, we can do two things :

(a) simulate the system and verify the operation
→ just like running a program written in high-level language.
→ Requires a test bench or test harness, that specifies the inputs that are to be applied and the way the outputs are to be displayed.

(b) use a synthesis tool to map it to hardware.
→ Converts it to a netlist of low-level primitives

→ The hardware can be Application Specific Integrated circuit (ASIC)

→ or else, it can be Field Programmable Gate Array (FPGA)

(A programmable hardware)

↳ when the design is mapped to hardware, we don't need test bench for simulation any more.

↳ Signals can be actually applied for some source (e.g. signal generator), and Response evaluated by some equipment (e.g. oscilloscope or logic analyzer)

ASIC as hardware target?

→ when high performance and high packing density is required.

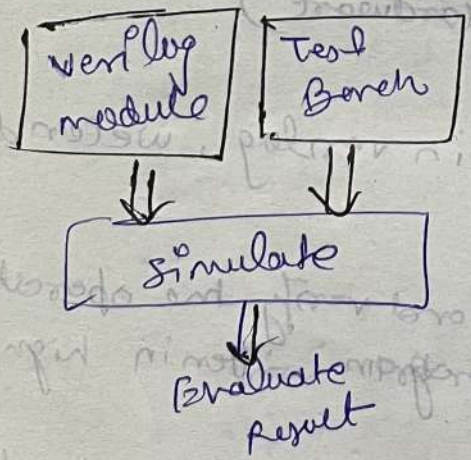
→ when the manufactured hardware is expected to be used in large numbers (e.g. processor chips)

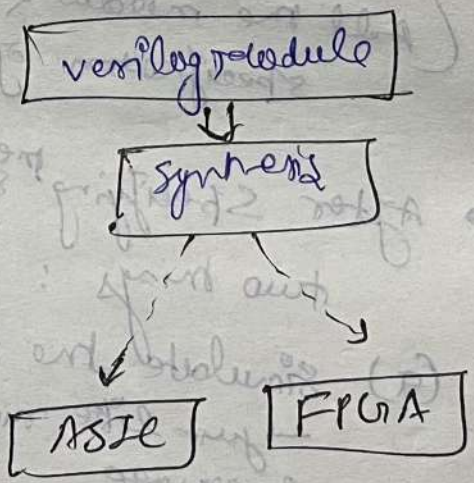
FPGA as hardware target?

→ when fast turnaround time is required to validate the design.

→ The mapping can be done in the laboratory itself with a FPGA kit and associated software.

→ There is a trade off in performance.





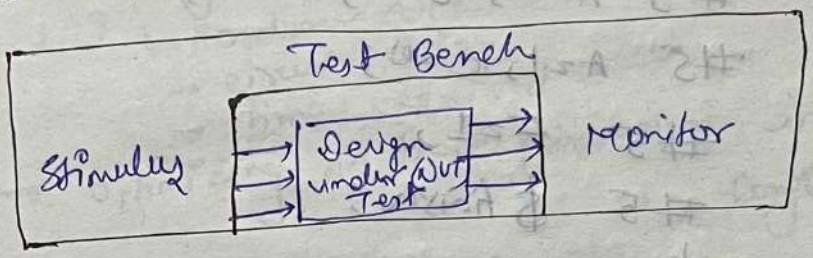
How to simulate verilog module :-

→ using a test bench to verify the functionality of a design coded in verilog (called Design-under-Test or DUT) comprising of :

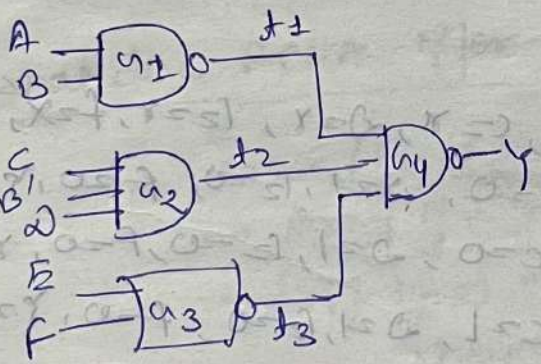
→ A set of stimulus for the DUT
→ A monitor, which captures or analyses the outputs of the DUT

Requirement

→ The inputs of the DUT need to be connected to the test bench.
→ The outputs of the DUT needs also to be connected to the test bench.



example :-




```
module example(A, B, C, D, E, f, Y);
input A, B, C, D, E, f, ;
output Y;
wire t1, t2, t3, Y;
nand #1 u1 (t1, A, B);
and #2 u2 (t2, C, ~B, D);
nor #1 u3 (t3, E, f);
nand #1 u4 (Y, t1, t2, t3);
endmodule
```

→ delay

~B = B'

we can combine declaration of same type of gate types.

```
nand #1 u1 (t1, A, B),
        u4 (Y, t1, t2, t3);
```

```
module testbench;
  reg A, B, C, D, E, F; wire f;
  example UUT (A, B, C, D, E, f, y);
  initial
    begin
      $monitor ($time, "A=%b, B=%b,
                        C=%b,
        D=%b, E=%b, f=%b, y=%b",
        A, B, C, D, E, f, y);
    #5 A=1; B=0; C=0; D=1; E=0; F=0;
    #5 A=0; B=0; C=1; D=1; E=0; F=0;
    #5 A=1; C=0;
    #5 F=1;
    #5 $finish;
  end
endmodule
```

Simulation Results :-

```
0  A=x, B=x, C=x, D=x, E=x, f=x, y=x
5  A=1, B=0, C=0, D=1, E=0, f=0, y=x
8  A=1, B=0, C=0, D=1, E=0, F=0, y=1
10 A=0, B=0, C=1, D=1, E=0, f=0, y=1
13 A=0, B=0, C=1, D=1, E=0, f=0, y=0
15 A=1, B=0, C=0, D=1, E=0, f=0, y=0
18 A=1, B=0, C=0, D=1, E=0, f=0, y=1
20 A=1, B=0, C=0, D=1, E=0, f=1, y=1
```

Command in iverilog :-

(a) iverilog -o mysim example.v example_test.v

(b) vvp mysim

various design styles –

→ Programmable logic Devices
  ↳ field programmable gate Array (FPGA)
    ↳ Gate Array

ASIC Design { → standard cell (semi-custom design)
              → full custom design.

which design style to use?

→ Basically a tradeoff among several design parameters,
  ↳ hardware cost
  → circuit delay
  → time required
→ optimizing on these parameters is often conflicting.

→ FPGA is easist but delay high
  in ASIC performance is good but cost and time required more.

what does FPGA offer –

- user / field programmability.
  → Array of logic cells connected via routing channels.
  → Diff types of cells –
    special I/O cells.
    logic cells (mainly lookup tables (LUT) with associated registers)
  → interconnection b/w cells –
    → using SRAM based switches
    → using Anti fuse elements.

Ease of use.

→ FPGA chips are manufactured by a no of vendors:
  – Xilinx, Altera, Actel etc.
  – Products vary widely in capability
→ FPGA Development boards and CAD software available from many sellers. ↳ Allows rapid prototyping in laboratory.

Concept of verilog module :-

→ in verilog, the basic unit of hardware is called a module.

↳ A module can't contain definition of other modules

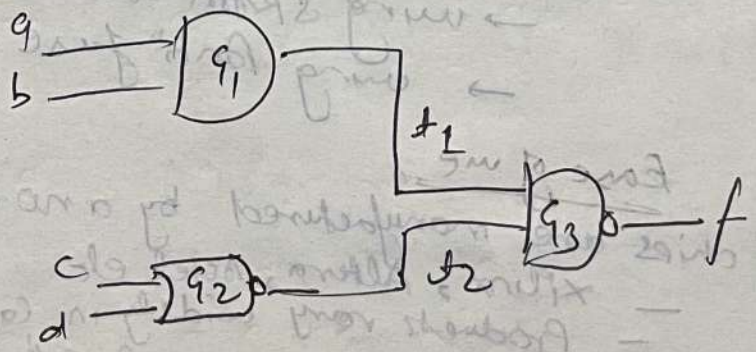
↳ A module can, however be instantiated within another module.

→ instantiation allows the creation of a hierarchy in verilog description.

```
module module_name (list_of_ports);
  input/output declarations
  local net declarations
  parallel statements
endmodule
```

```
// A simple AND function
module simpleand (f, x, y);
    input x, y;
    output f;
    assign f = x & y;
endmodule
```

ex :-



// 2-level combinational ckt

```
module two_level (a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire t1, t2;  // intermediate line
    assign t1 = a & b
    assign t2 = ~(c | d);
    assign f = ~(t1 & t2);
endmodule
```

⇒ The assign statement represents continuous assignment, whereby the variable on the LHS gets updated whenever the expression on the RHS changes.

```
assign variable = expression;
```

→ The LHS must be a "net" type variable, typically a "wire".

→ The RHS can contain both "register" and "net" type variables.

→ A verilog module can contain any number of "assign" statements; they are typically placed in the beginning after the port declarations.

→ The "assign" statement models behavioral "dataflow" style and is typically used to model combinational circuits.

Data types in verilog

A variable in verilog belongs to one of two data types :-

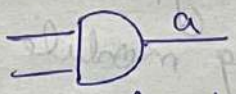
(a) Net
- ↳ must be continuously driven
- ↳ can't be used to store a value
- → used to model connections between continuous assignments and instantiations.

(b) Register
- → Retains the last value assigned to it.
- → often used to represent storage elements but something it can translate to combinational circuit also.

(a) Net data type

↳ Nets represents connection b/w hardware elements.

→ Nets are continuously driven by the outputs of the devices they are connected to



→ Net 'a' is continuously driven by the output of the AND gate.

→ Nets are 1-bit values by default unless they are declared explicitly as vectors.

→ Default value of a net is "z".
↓
high impedance or tri state.

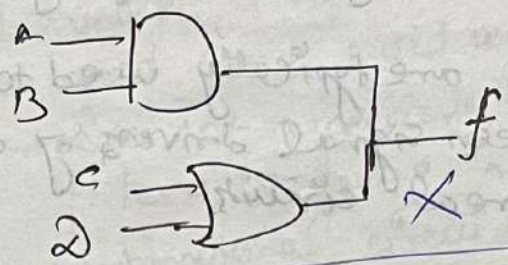
→ various "Net" data types are supported for synthesis in verilog :- wire, wor, wand, tri, supply0, supply1

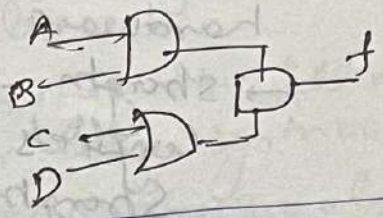
→ "wire" and "tri" are equivalent ; when nets are multiple drivers driving them, the driver outputs are shorted together. etc

→ "wor" and "wand" inserts an OR and AND gate respectively at the connection.

→ "supply0" and "supply1" model the power supply connections.

eg:-






```
module we_wire (A,B,C,D,f);
  input A, B, C, D;
  output f;
  wire f;
  //net f declared as wire
  assign f = A & B;
  assign f = C | D;
endmodule
```

for A=B=1 and C=D=0
f will be indeterminate

```
Module we_wand (A,B,C,D,f);
  input A, B, C, D;
  output f;
  wand f;
  //net f declared as 'wand'
  assign f = A & B;
  assign f = C | D;
  endmodule
```

Here function realized will be
$f = (A \& B) \& (C | D)$

```
module wire_supply_wire (A,B,C,f);
  input A, B, C;
  output f;
  supply 0 grd;
  supply 1 vdd;
  nand g1 (f1, vdd, A, B);
  XOR g2 (f2, C, grd);
  and g3 (f, f1, f2);
endmodule
```

Data values and signal strengths

- Verilog supports 4 value levels and 8 strength levels to model the functionality of real hardware.
  - → strength levels are typically used to resolve conflicts between signal drivers of diff strengths in real circuits.

value level	Represents
0	logic 0 state
1	logic 1 state
x	unknown logic state
z	high impedance state

initialization :-

- → All unconnected nets are set to "z".
- → All register variables set to "x".

(b). Register Data type

- In a verilog, a "register" is a variable that can hold a value.
  - ↳ unlike a "net" that is continuously driven and can't hold any value.
  - → does not necessarily mean that it will map to a hardware register during synthesis
  - → Combinational circuit specification can also use register type variables.
- Register data types supported by verilog:
  - (i) reg : most widely used
  - (ii) integer : used for loop counting
  - (iii) real : used to store floating-point numbers
  - (iv) time : keeps track of simulation time (not used in synthesis)

(i) "reg" data type

- → Default value of a "reg" data type is "x".
- → it can be assigned a value in synchronism with a clock or even otherwise.
- → The declaration explicitly specifies the size (default is 1-bit):

```
reg x,y;          // single-bit register variables x,y
reg [15:0] bus    // A 16 bit bus
```

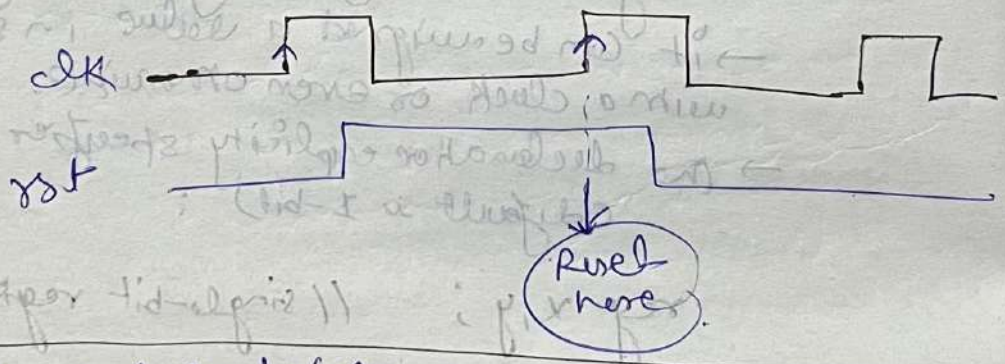
- → Treated as an unsigned number in arithmetic expressions.
- → must be used when we model actual sequential hardware elements like counters, shift registers etc.

ex:- 32 bit counter with synchronous reset

↳ count value increases at the positive edge of the clock.

↳ If "rst" is high, the counter is reset at the positive edge of the next clock.

```
module simple_counter (clk, rst, count);
    input clk, rst;
    output count;
    reg [31:0] count;
    always @ (posedge clk)
    begin
        if (rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```




```
module simple_counter (clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;
    always @ (posedge clk or posedge rst)
    begin
        if (rst)
            count = 32'b0
        else
            count = count + 1;
    end
endmodule
```

32 bit counter with asynchronous reset
↳ Here reset occurs whenever 'rst' goes high.
→ Does not synchronize with clock.

(ii) "integer" Data type :-
→ It is a general purpose register data type used for manipulating quantity.
→ more convenient to use in situations like loop counting than 'reg'.
→ It is treated as a 2's complement signed integer in arithmetic expressions.
→ Default size is 32 bits; however, the synthesis tool tries to determine the size using data flow analysis.
→ Example :- wire [15:0] x, y;
integer c;
c = x + y;
size of c can be deduced to be 17 (16 bits plus a carry).

(iii) "real" data type :-
→ used to store floating point numbers
→ when a real value is assigned to an integer, the real number is rounded off to the nearest integer.
→ Example :
real e, pi;
initial
begin
e = 2.718;
pi = 314.159e-2;
end

integer x;
initial
x = pi; // gets value 3

(iv) "time" Data type

→ in verilog, simulation is carried out with respect to a logical clock called simulation time.

→ The "time" data type can be used to store simulation time.

→ The system function "$time" gives the current simulation time.

→ Example :-

```
time curr_time;
initial
  ----
    curr_time = $time;
```

Vectors

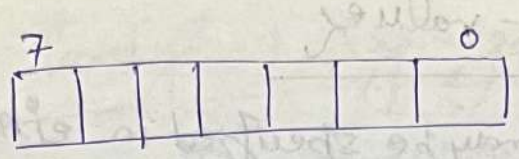
→ Nets or "reg" type variables can be declared as vectors, of multiple bit widths.

- if bit width is not specified, default size is 1-bit.

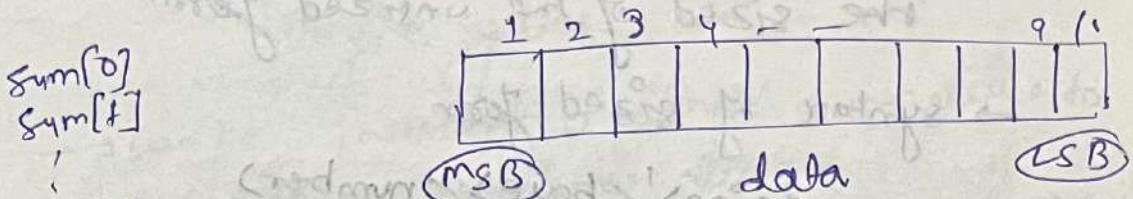
→ vectors are declared by specifying a range [range1 : range2], where range1 is always the most significant bit and range2 is the least significant bit.

• Examples :-

```
wire x, y, z;        // single bit variables
wire [7:0] sum;      // MSB is sum[7], LSB is sum[0]
reg [31:0] mdr;
reg [1:10] data;     // MSB is data[1], LSB is data[10]
reg clock;
```



sum



[1 : 10]

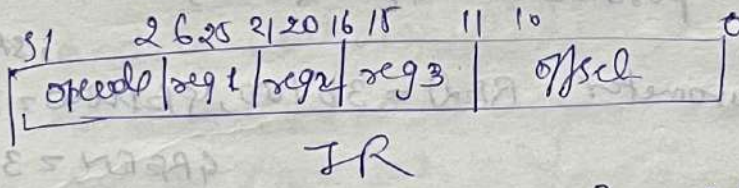
→ parts of a vector can be addressed and used in an expression.

→ Example :-

↳ A 32 bit instruction register, that contains a 6 bit opcode, 3 register operands of 5 bits each, and an 11 bit offset.

```
reg [31:0] IR;
reg [5:0] opcode;
reg [4:0] reg1, reg2, reg3;
reg [10:0] offset;
```

```
opcode = IR[31:26];
reg1 = IR[25:21];
reg2 = IR[20:16];
reg3 = IR[15:11];
offset = IR[10:0];
```



IR

sum = IR[25:21] + IR[20:16]

Multi dimensional Arrays and memories :-

→ multi dimensional arrays of any dimension can be declared in verilog.

Example :- reg [31:0] register_bank [15:0]; // 16, 32 bit registers

integer matrix [7:0][15:0]; → 8 row x 16 column (2-D) matrix

→ memories can be modeled in verilog as a 1-D array of registers

→ Each element of the array is addressed by a single array index.

Example - reg mem_bit [0:2047] // 2K x 1 bit words

reg [15:0] mem_word [0:1023] // 1K 16 bit words

Specifying Constant values

→ A constant value may be specified in either the sized or the unsized form.

→ syntax of sized form.

<size>'<base><number>

Examples :-

```
4'b0101   // 4-bit binary number 0101
1'b0      // 1 bit logic 0
12'hB3C   // 12 bit number 1011 0011 1100
```

Parameters :- It is a constant given a name.

→ we can't specify the size of a parameter

→ the size gets decided from the constant value itself ; if size is not specified it is taken to be 32 bits.

example :-

```
parameter HI=25, LO=5;
parameter up=2'b00, down=2'b01
                , steady=2'b10;
parameter RED=3'b100, YELLOW=3'b010,
          GREEN=3'b001;
// parameterized design :- an N-bit counter
module counter (clear, clock, count);
    parameter N=7;
    input clear, clock;
    output [0:N] count; reg [0:N] count;
always @(negedge clock)
    if (clear)
        count <=0;
    else
        count <= count+1;
endmodule
```

Predefined logic gates in verilog :-

→ verilog provides a set of predefined logic gates.

→ can be instantiated within a module to create a structured design.

→ The gates respond to logic values (0, 1, x or z) in a logical way.

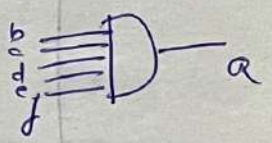
2 input AND	2-i/p OR	2 i/p EXOR
0 & 0 = 0	0 | 0 = 0	0 ^ 0 = 0
0 & 1 = 0	0 | 1 = 1	0 ^ 1 = 1
1 & 1 = 1	1 | 1 = 1	1 ^ 1 = 0
1 & x = x	1 | x = 1	x ^ x = x
0 & x = 0	0 | x = x	0 ^ x = x
1 & z = x	1 | z = x	1 ^ z = x
z & x = x	z | x = x	z ^ x = x

List of primitive gates

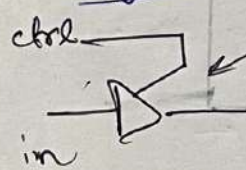
```
and g1 (out, in1, in2);
nand g1 (out, in1, in2);
or g1 (out, in1, in2);
nor g1 (out, in1, in2);
xor g1 "
xnor g1 "
not g1 (out, in);
buf g1 (out, in);
```

```
and g1 (a, b, c, d, e, f);
```
a = output; b, c, d, e, f = i/p

(5 i/p AND gate)

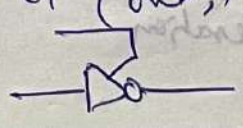


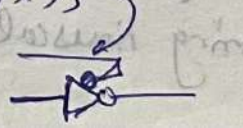
buffer

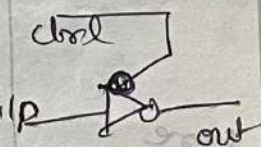


if ctrl=1, out=in
=0, out=z

```
bufif1 g1 (out, in, ctrl);
bufif0 g1 (out, in, ctrl);
notif0 g1 (out, in, ctrl);
notif1 g1 (out, in, ctrl);
```







if ctrl=0, out=in
ctrl=1, out=z

→ some restriction when instantiating primitive gates :-

↳ The output port must be connected to a net (e.g. a wire)

Can "output" signal is a wire by default, unless explicitly declared as a register.

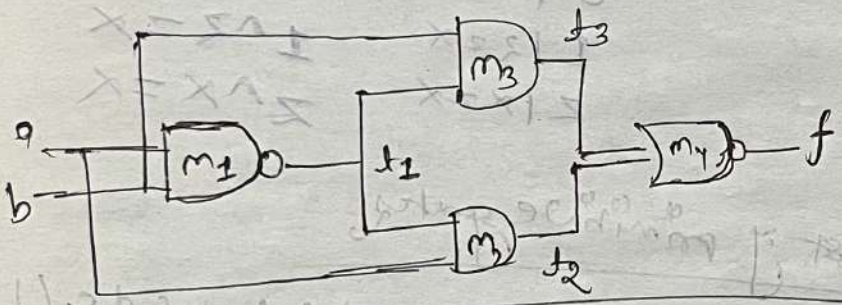
↳ The input ports may be connected to nets or register type variables.

→ They have a single output but can have any number of inputs (except NOT and BUF).

→ when instantiating a gate, an optional delay may be specified

↳ used for simulation

↳ Logic synthesis tools ignore the time delays.




```
`timescale 10ns/1ns
module exclusive_or (f, a, b);
    input a, b;
    output f;
    wire t1, t2, t3;
    nand #5 m1 (t1, a, b);
    and #5 m2 (t2, a, t1);
    and #5 m3 (t3, t1, b);
    nor #5 m4 (f, t2, t3);
endmodule
```

here #5 means ↓ (50ns) as specified during timescale declaration

The 'timescale directive -'

Syntax :

`timescale <reference_time_unit> / <time_precision>`

→ The <reference_time_unit> specifies the unit of measurement for time.

→ The <time_precision> specifies the precision to where the delays are rounded off during simulation.

→ valid values for specifying time unit and time precision are 1, 10 and 100.

Hardware modelling issues

In terms of the hardware realization, the value computed can be assigned to :

→ A "wire"

→ A "flip-flop" (edge triggered storage cell)

→ A "latch" (level-triggered storage cell)

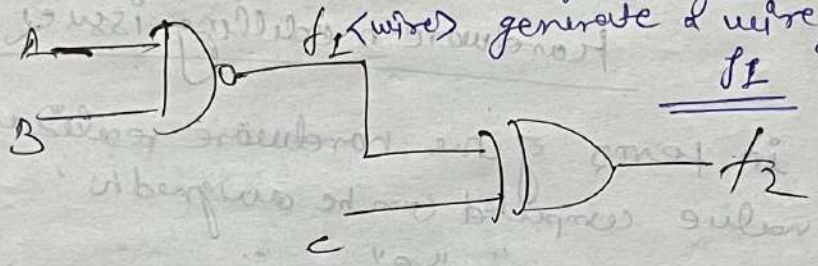
→ A variable in verilog can be either "net" or "register".

→ A "net" data type always map to a "wire" during synthesis

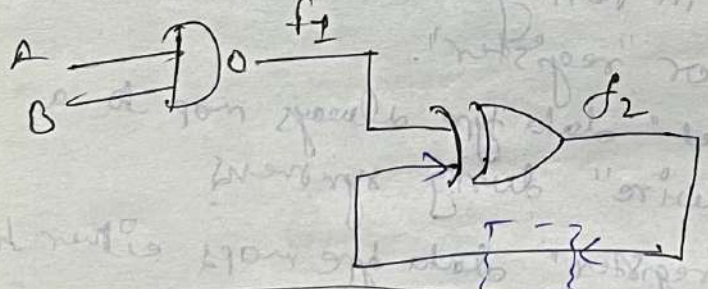
→ A "register" data type maps either to a "wire" or a "storage cell" depending upon the context under which a value is assigned.

```
module reg_maps_to_wire (A, B, C, f1, f2);
  input A, B, C;
  output f1, f2;
  wire A, B, C;
  reg f1, f2;
  always @ (A or B or C)
    begin
      f1 = ~(A & B);
      f2 = f1 ^ C;
    end
endmodule
```

→ The synthesis system will generate a wire for f1



{ so whenever any signal (A or B or C) is changing, always block will be executed }




```
module a_problem_core (A, B, C,
                       f1, f2);
  input A, B, C;
  output f1, f2;
  wire A, B, C;
  reg f1, f2;
  always @ (A or B or C)
    begin
      f2 = f1 ^ f2;
      f1 = ~(A & B);
    end
endmodule
```

storage cell or latch generated for f2

example of latch:-

```
// A latch gets inferred here
module simple_latch (data, load, d_out);
  input data, load;
  output d_out;
  wire t;
  always @(load or data)   // whenever load or data changes
  begin
    if (!load)             // if load is zero
      t = data             // data will go to t
    d_out = !t;
  end
endmodule
```

Verilog operators :-

Arithmatic operators : → Unary (+A, −B, +(A+B))
→ Binary (A+B, A−B, A*B)

+	unary (sign) plus
−	unary minus
+	binary plus (add)
−	binary minus
*	multiply
/	divide
%	modulus
**	exponentiation

logical operators :-

!	logical negation
&&	logical AND
||	logical OR

&&

A	B	A && B
f	f	f
f	T	f
T	f	f
T	T	T

||

A	B	A || B
f	f	f
f	T	T
T	f	T
T	T	T

!

A	!A
f	T
T	f

Relational operators :-

!=	not equal
==	equal
>=	greater or equal
<=	less or equal
>	greater
<	less

Bitwise operators :-

~	bitwise NOT
&	bitwise AND
|	bitwise OR
^	bitwise exclusive-OR
~^	bitwise exclusive-NOR

Example :-

Shift operators -

>> shift right

<< shift left

>>> arithmetic shift right

example -: wire [15:0] data, target;

assign target = data >> 3;

assign target = data >>> 2;

conditional operator :

cond_expr ? true_expr : false_expr;

example -: wire a, b, c;

wire [7:0] x, y, z;

assign a = (b > c) ? b : c;

assign z = (x == 4) ? x+2 : x-2;

if b > c then b will assign if false then c will assign.

if x == 4 then x+2 else x-2

Concatenation operator -:

Join together bits from two or more comma-separated expressions

{.. , ... , ...}

Replication operator -:

Join together n copies of an expression m, where n is a constant.

{n{m}}

assign f = {a, b};

assign f = {a, 3'b101, b};

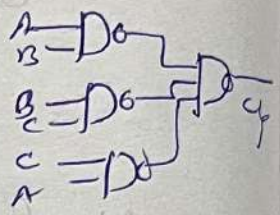
assign f = {2'b10, 3{2'b01}, x};

10 010101 x → 9bit vector

Modeling

- Behavioral
  - starting point
- structural
  - hardware implementation

→ we start with a behavioral description and convert into structural description by some steps.



example → The structural hierarchical description of 16:1 MUX.

(a) using pure behavioral modelling
(b) structural modelling using 4:1 mux specified using behavioral model
(c) make structural modelling of 4:1 mux using behavioral modelling of 2:1 mux
(d) make structural gate level modelling of 2:1 mux to have a complete structural hierarchical description.

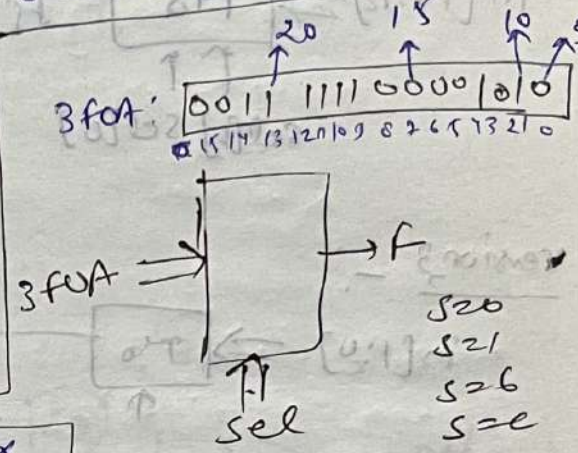
Version 1
(a) using pure behavioral modelling

```
module mux16to1 (in, sel, out);
  input [15:0] in;
  input [3:0] sel;
  output out;
  assign out = in[sel];
endmodule
```

Select one of the input bit depending upon the value of "sel";

Testbench :-

```
module muxtest;
reg [15:0]A; reg [3:0]S; wire f;

MUX16to1 M (.in(A), .sel(S), .out(f));

initial
  begin
   $dumpfile ("mux16to1.vcd");
   $dumpvars (0, muxtest);
   $monitor ($time, "A=%h, S=%h, f=%b, A, S, f);
   #5 A = 16'h3f0a; S = 4'h0;
   #5 S = 4'h1;
   #5 S = 4'h6;
   #5 S = 4'hc;
   #5 $finish;
  end
endmodule
```

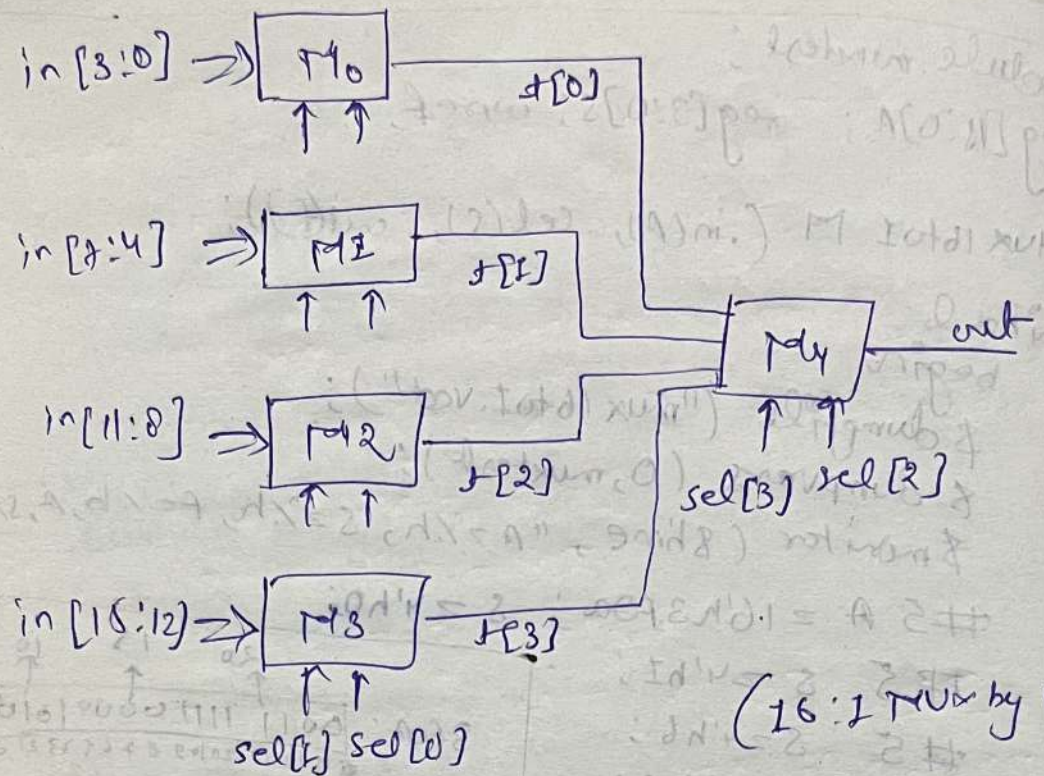



```
0  A=xxxx, S=x, f=x
5  A=3f0a, S=0, f=0
10 A=3f0a, S=1, f=1
15 A=3f0a, S=6, f=0
20 A=3f0a, S=e, f=1
```

Version 2 :- Behavioral modeling of 4:1 mux
Structural modeling of 16:1 mux

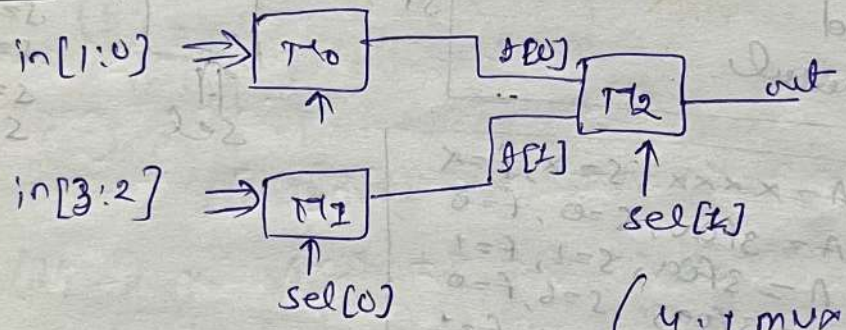
```
module mux4to1 (in, sel, out);
input [3:0] in;
input [1:0] sel;
output out;
assign out = in[sel];
endmodule
```

```
module mux16to1 (in, sel, out);
input [15:0] in;
input [3:0] sel;
output out;
  wire [3:0] t;
mux4to1 M0 (in[3:0], sel[1:0], t[0]);
mux4to1 M1 (in[7:4], sel[1:0], t[1]);
mux4to1 M2 (in[11:8], sel[1:0], t[2]);
mux4to1 M3 (in[15:12], sel[1:0], t[3]);
mux4to1 M4 (t, sel[3:2], out);
endmodule
```



(16:1 MUX by 4:1 mux)

version 3 :-



(4:1 MUX by 2:1 MUX)

```
module mux2to1 (in, sel, out);
  input [1:0] in;
  input sel;
  output out;
  assign out = in[sel];
endmodule
```

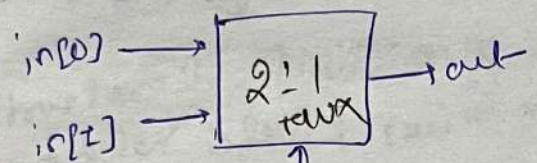
```
module mux4to1 (in, sel, out);
  input [3:0] in;
  input [1:0] sel;
  output out;
  wire [1:0] t;

mux2to1 M0 (in[1:0], sel[0], t[0]);
mux2to1 M1 (in[3:2], sel[0], t[1]);
mux2to1 M2 (t, sel[1], out);
endmodule
```

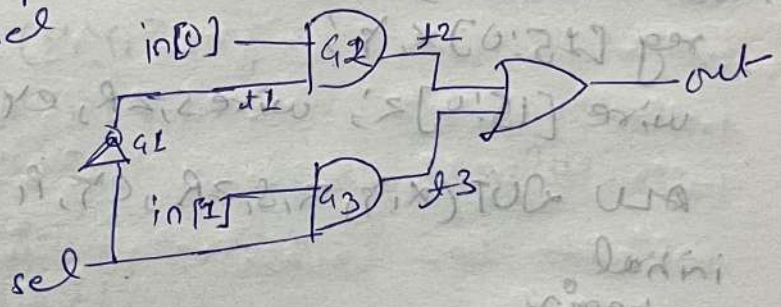
Version-4 structural modeling of 2:1 mux

```
module mux2to1 (in, sel, out);
  input [1:0] in;
  input sel;
  output out;
  wire t1, t2, t3;
  NOT G1 (t1, sel);
  AND G2 (t2, in[0], t1);
  AND G3 (t3, in[1], sel);
  OR G4 (out, t2, t3);
endmodule
```

→ Same testbench can be used for all the versions.

→ The versions illustrate hierarchical refinement of design.





Generation of status flags :-

Sign : whether the sum is negative or positive

zero : whether the sum is zero.

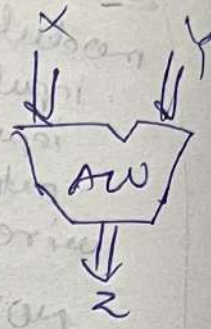
carry : whether there is a carry out of last stage

Parity : whether the number of '1's in the sum is even or odd

overflow : whether the sum can't fit in 16 bits.

example-2

version 1 : Behavioral description of a 16 bit adder.




```
module ALU (x, y, z, sign, zero, carry, parity,
            overflow);
  input [15:0] x, y;
  output [15:0] z;
  output sign, zero, carry, parity, overflow;
  assign {carry, z} = x+y ; //16 bit addition
    assign sign = z[15];
    assign zero = ~|z;
    assign parity = ~^z;
    assign overflow = (x[15] & y[15] & ~z[15] |
                       (~x[15] & ~y[15] & z[15]);
    endmodule
```

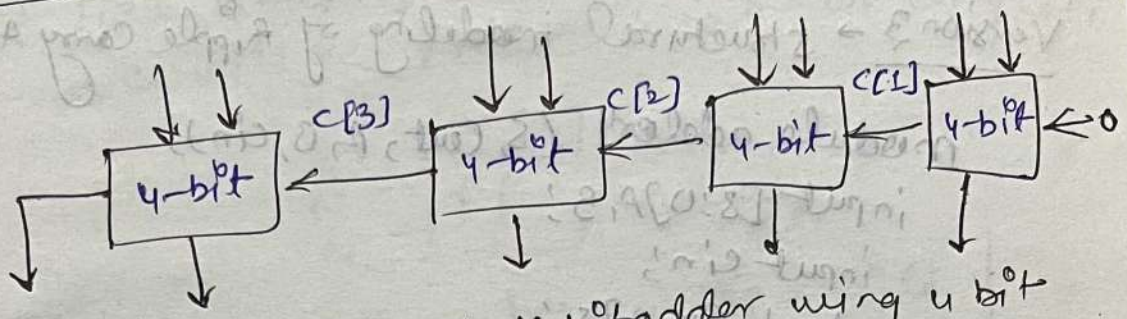
```
module alutest;
  reg [15:0] x, y;
  wire [15:0] z; wire s, zR, cy, p, v;
  ALU OUT (x, y, z, s, zR, cy, p, v);
  initial
    begin
    $dumpfile ("alu.vcd"); $dumpvars (0, alutest);
    $monitor ($time, " x=%h, y=%h, z=%h,
              s=%b, z=%b, cy=%b, p=%b,
          v=%b", x, y, z, s, zR, cy, p, v);
    #5 x=16'h 8fff ; y=16'h 8000;
    #5 x=16'hfffe ; y=16'h 0002;
    #5 x=16'hAAAA ; y=16'h 5555;
    #5 $finish
    end
  endmodule
```

Simulation o/p :-

0 x=xxxx, y=xxxx, z=xxxx, s=x, z=x, cy=x, p=x, v=x;
5 x=8fff, y=8000, z=0fff, s=0, z=0, cy=1, p=1, v=1
10 x=fffe, y=0002, z=0000, s=0, z=1, cy=1, p=1, v=0
15 x=aaaa, y=5555, z=ffff, s=1, z=0, cy=0, p=1, v=0

Version2 :-

16 bit adder by 4 4-bit adder :-



* Structural discription of 16 bit adder using 4 bit adder blocks (with ripple carry b/w blocks)

```
module ALU (X, Y, Z, sign,
input [15:0] X, Y;
output [15:0] Z;
output sign, zero, carry, parity, overflow;
wire c[3:1];
assign sign = Z[15];
assign zero = ~|Z;
assign parity = ^Z;
assign overflow = (X[15] & Y[15] & ~Z[15])
                | (~X[15] & ~Y[15] &
                   Z[15]);
adder4 A0 (Z[3:0], c[1], X[3:0], Y[3:0], 1'b0);
adder4 A1 (Z[7:4], c[2], X[7:4], Y[7:4], c[1]);
adder4 A2 (Z[11:8], c[3], X[11:8], Y[11:8], c[2]);
adder4 A3 (Z[15:12], carry, X[15:12], Y[15:12], c[3]);
endmodule
```

Behavioral description of a 4 bit adder

```
module adder4 (S, Cout, A, B, Cin);
input [3:0] A, B;  input Cin;
output [3:0] S;  output out;
assign {Cout, S} = A + B + Cin;
endmodule
```

Version 3 → structural modeling of Ripple Carry Adder

```
module adder4 (S, Cout, A, B, Cin);
  input [3:0] A, B;
  input Cin;
  output [3:0] S;
  output Cout;
  wire C1, C2, C3;

fulladder FA0 (S[0], C1, A[0], B[0], Cin);
fulladder FA1 (S[1], C2, A[1], B[1], C1);
fulladder FA2 (S[2], C3, A[2], B[2], C2);
fulladder FA3 (S[3], Cout, A[3], B[3], C3);
endmodule
```

Carry lookahead Adder :-

→ The t_{pd} delay of an n bit ripple carry adder is proportional to n.
(Due to the rippling effect of carry sequentially from one stage to the next)

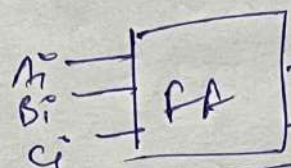
→ One possible way to speedup the addition
↳ Generate the carry signals for the various stages in parallel
↳ Time complexity reduces from O(n) to O(1).
↳ Hardware complexity increases rapidly with n.

→ we define the carry generate and carry propagate functions as:

$$g_i = A_i \cdot B_i$$
$$p_i = A_i \oplus B_i$$

→ $g_i = 1$ represents the condition when a carry is generated in stage-i independent of the other stages.

→ $p_i = 1$ represents the condition when an i/p carry c_i will be propagated to the o/p carry c_{i+1}



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

$$c_{i+1} = g_i + \sum_{K=0}^{i-1} g_K \prod_{j=K+1}^{i} p_j + c_0 \prod_{j=0}^{i} p_j$$

Generation of Carry and sum bits :-

$$c_4 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + c_0 p_0 p_1 p_2 p_3$$
$$c_3 = g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$$
$$c_2 = g_1 + g_0 p_1 + c_0 p_0 p_1$$
$$c_1 = g_0 + c_0 p_0$$

$$S_0 = A_0 \oplus B_0 \oplus c_0 = p_0 \oplus c_0$$
$$S_1 = p_1 \oplus c_1$$
$$S_2 = p_2 \oplus c_2$$
$$S_3 = p_3 \oplus c_3$$

4 AND 2 gates
3 AND 3 gates
2 AND 4 gate
1 AND 5 gate
1 OR 2, 1 OR 3, 1 OR 4 and 1 OR 5 gate
4 XOR 2 gate

Description styles in verilog -:

Two diff styles -

(1) Dataflow (using Assignment statements)
- Continuous Assignment

(2) Behavioral
- Procedural assignment (using Procedural statements similar to a program in high level language)
  - Blocking
  - Non Blocking

⇒ Dataflow style: Continuous Assignment

→ keyword → assign
→ forms a static binding b/w
- The "net" being assigned on the LHS
- The expression on RHS which may consists of both "net" and "Register" type variable

{assign a = b+c;
assign sign = z[7:5];

⇒ A verilog module can contain any no of "assign" statements.
⇒ Typically the "assign" statements are followed by procedural descriptions
⇒ The assign statements are used to model behavioral descriptions

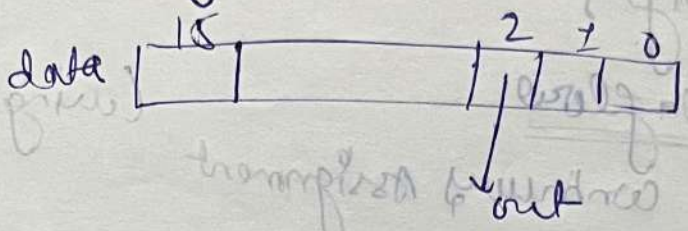
```
module generate_mux (data, select, out);
  input [15:0] data;
  input [3:0] select;
  output out;
  assign out = data[select];
endmodule
```

How does this syntax that a mux is generated?

Know because this non constant index in expression on RHS generates a MUX

→ If the index is a constant, just a wire will be generated

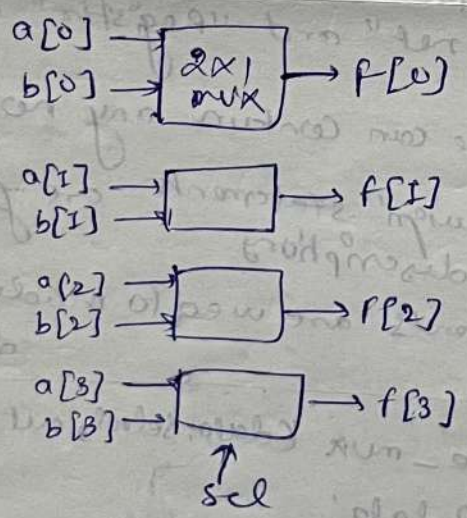
ex:- assign out = data[2];




```
module generate_set_of_MUX (a, b, f, sel);
  input [0:3] a, b;
  input sel;
  output [0:3] f;
  assign f = sel ? a : b;
endmodule
```

sel → 1 (or true) then select a otherwise b

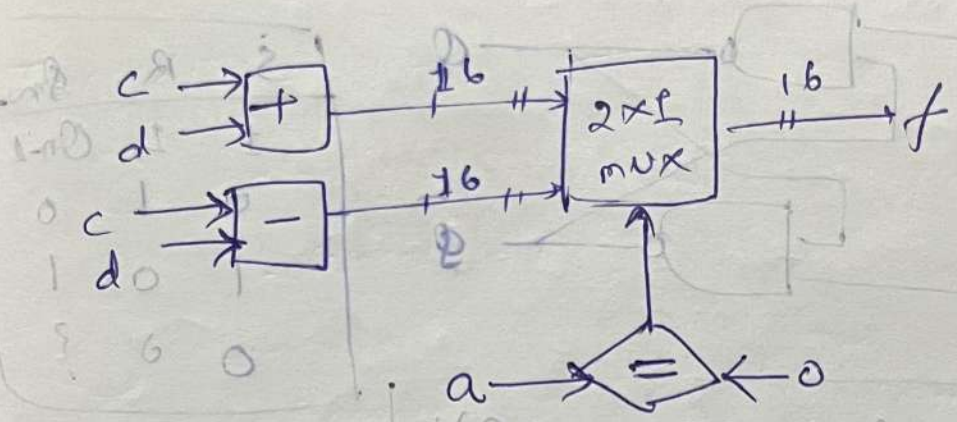
conditional operator generates a MUX



→ so whenever a conditional is encountered in the RHS of an expression, a 2 to 1 mux is generated.

→ in previous ex., since the variables a, b and f are vectors, an array of 2 to 1 MUXes are generated

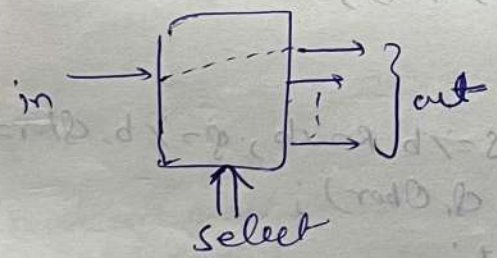
```
assign f = (a==0) ? (c+d) : (c-d);
```



example of generating a decoder :-

```
module generate_decoder(out, in, select);
  input in;
  input [0:1] select;
  output [0:3] out;
  assign out[select] = in;
endmodule
```

Non constant index in expression on LHS generates a decoder



→ A constant index in the expression on the LHS will not generate a decoder.

ex:- `assign out[5] = in;` (This will generate a wire connection)

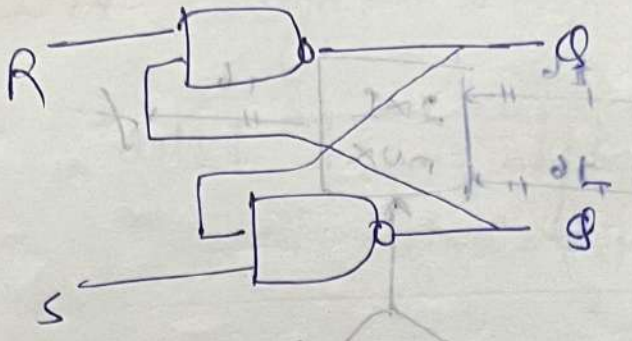
```
module level_sensitive_latch(D, Q, En);
  input D, En;
  output Q
  assign Q = En ? D : Q;
endmodule
```

En	D	Qn
0	X	Qn-1
1	0	0
1	1	1

example to describe a sequential logic element using assign statement.

Generates a D type latch

Modelling a simple S-R latch :



S	R	Q_n
1	1	Q_{n-1}
0	1	0
1	0	1
0	0	?

```
module sr_latch (Q, Qbar, S, R);
  input S, R;
  output Q, Qbar;
    assign Q = ~(R & Qbar);
    assign Qbar = ~(S & Q);
  endmodule
```

```
module latchtest;
  reg S, R; wire Q, Qbar;
  sr_latch LAT(Q, Qbar, S, R);
  initial
    begin
    $monitor ($time, " S=%b R=%b, Q=%b, Qbar=%b",
                S, R, Q, Qbar);
    S = 1'b0; R = 1'b1;
    #5 S = 1'b1; R = 1'b1;
    #5 S = 1'b1; R = 1'b0;
    #5 S = 1'b1; R = 1'b1;
    #5 S = 1'b0; R = 1'b0;
    #5 S = 1'b1; R = 1'b1;
    end
  endmodule
```

Simulation O/P :-

```
0 S=0, R=1, Q=0, Qbar=1
5 S=1, R=1, Q=0, Qbar=1
10 S=1, R=0, Q=1, Qbar=0
15 S=1, R=1, Q=1, Qbar=0
20 S=0 R=0, Q=1, Qbar=1
and then simulator hangs.
```

⇒ Behavioral style :- Procedural Assignment

Two kinds of procedural blocks are supported in verilog :-

(1) The "initial" block
- ↳ Executed once at beginning of simulation.
- ↳ used only in testbenches; can't be used in synthesis.

(2) The "always" block
- ↳ A continuous loop that never terminates.

→ A procedural block defines :-
- ↳ A region of code containing sequential statements.
- ↳ The statements execute in the order they are written.

initial block

```
initial
  begin
    :
    :
  end
```

(executed one time)

→ If there are multiple initial blocks, all the blocks will start to execute concurrently at time 0.

→ If there is single statement, we don't need a begin and end.

ex :-

```
module testbench_example;
  reg a, b, cin, sum, cout;
  initial
    cin = 1'b0;   } don't need begin, end for single statement
  initial
    begin
      #5 a = 1'b1; b = 1'b1;
      #5 b = 1'b0;
    end
  initial #25 $finish;
endmodule
```

all three "initial" blocks execute concurrently.

shortcuts in Declarations :-

→ "output" and "reg" can be declared together in the same statement.

output reg [7:0] data;

instead of output [7:0] data; reg [7:0] data;

→ A variable can be initialized when it is declared.

reg clock = 0; instead reg clock;
initial clock = 0;

"Always" block :-

always block

```
always (----)
begin
  =
  =
  =
end
```

(repitative execution in always block)

→ statements inside always blocks executed repeatedly and never stops.

ex:- A clk signal that is generated continuously.

Basic syntax of "always" block :-

```
always @ (event_expression)
begin
    sequential_statement_1;
    sequential_statement_2;
    -
    -
    sequential_statement_n;
end
```

→ The @(event_expression) part is required for both combinational and sequential circuit descriptions.

→ only "reg" type variable can be assigned within an "initial" or "always" block

Sequential statements in verilog :-

→ multiple assignment statements inside a begin--end block may either execute sequentially or concurrently depending upon the type of assignment.

→ Two types of assignment statements → blocking (a = b+c;)
→ non blocking (a <= b+c;) &

(a) begin ---- end

begin
sequential_statement ;
sequential_statement ;
.
.
end

} if n>1 begin--end not required.

(b) if --- else

```
if (<expression>)
  sequential_statement;
```

```
if (<expression>)
  sequential_statement;
else
  sequential_statement;
```

```
if (<expression1>)
  sequential_statement;
else if (<expression2>)
  sequential_statement;
else if (<expression3>)
  sequential_statement;
else default_statement;
```

(c) case

```
case (<expression>)
expr1 : sequential_statement;
expr2 : sequential_statement;
...
exprn : sequential_statement;
default : default_statement;
endcase
```

Each sequential_statement can be a single statement or a group of statements within "begin--end".

→ can replace a complex "if--else" statement for multiway branching.

→ The expression is compared to the alternatives (expr 1, 2, n) in order they are written.
→ If none of the alternative matches, the default statement is executed.

→ Two variations of case : casez and casex

↳ The "casez" statement treats all "z" values in the case alternatives or the case expression as dont cares.

↳ The "casex" statement treats all "x" and "z" values in the case item as dont cares.

• example -:

if state is "4'b01zx", the second expression will give match and next_state will be 1.

```
reg [3:0] state; integer next_state;
casex (state)
4'b1xxx : next_state = 0;
4'bx1xx : next_state = 1;
4'bxx1x : next_state = 2;
4'bxxx1 : next_state = 3;
default : next_state = 0;
endcase
```

→ initial and always block can coexist within same verilog module

→ They all execute concurrently ; "initial" only once and "always" repeatedly.

```
module generating_clock;
  output reg clk;
  initial
    clk = 1'b0;      // initialized to 0 at time 0.
  always
    #5 clk = ~clk; // toggle after time 5 units
  initial
    #50 finish;
endmodule
```

(d) "while" loop :-

```
while (<expression>)
  sequential - statement;
```

(while loop executes until the expression is not true)

```
while (<expression>)
  begin
    sequential - statement1;
    sequtal - stat2;
    :
    :
  end
```

Example :-

```
integer mycount;
initial
  begin
    mycount=0;
  while (mycount <=255)
    begin
      $display ("mycount: %d", mycount);
      mycount = mycount +1;
    end
  end
```

(e) "for" loop :-

```
for (expr1; expr2; expr3)
  sequential - statement;
```

The for loop executes as long as expr2 is true.

→ group of statements with "begin...end".

expr1 → initial condn

expr2 → terminating condn

expr3 → a procedural assignment to change the value of control variable.

→ The for loop can be used to initialize an array or memory.

example-1

```
integer mycount;
reg [low-1] data;
integer i;
initial
  for (mycount=0; mycount<=255;
                  mycount=mycount+1)
$display("mycount : %d", mycount);
initial
  for (i=1; i<=low; i=i+1)
    data[i] = 1'b0;
```

(f) "repeat" loop :-

executes the loop a fixed no of times.

```
repeat (<expression>)
  sequential_statement;
```

→ expr in repeat can be a constant, variable or a signal value.

↳ if it is a variable or signal value, it is evaluated only when the loop starts and not during execution of the loop.

```
repeat (a)
begin
  —
  —
  =
end
```

} if a=10, loop executed 10 times

Example-1

```
reg clock;
initial
begin
  clock = 1'b0;          // Exactly low clock pulses generated
  repeat (low)
    #8 clock = ~clock;
end
```

(9) "forever" loop

```
forever
  sequential-statement;
```

→ does not use any expression and executes forever until $finish is encountered in the test-bench.

→ exact to while loop for which the expression is always true.

→ if delay is not specified, Rest of design will not be executed.

example :-

```
reg clk;
initial
  begin
    clk = 1'b0;
    forever #5 clk = ~clk;
  end
```

other constructs :-

# (time-value)	→ makes a block suspend for "time-value" units of time.
@ (event_expression)	→ makes a block suspend until "event-expression" triggers.

↓

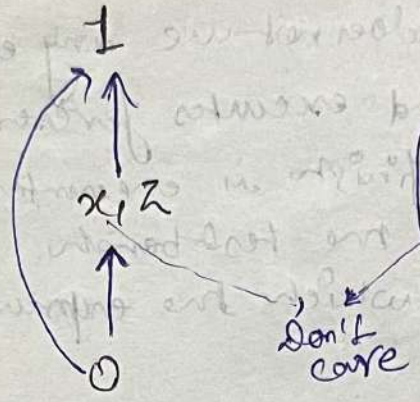
This specifies the event that is required to resume execution of procedural block.

This can be any one of the following :-

1. change of a signal value
2. positive or negative edge occuring on signal (posedge or negedge)

→ A posedge is any transition from {0, x, z} to 1 and from 0 to {z, x}.

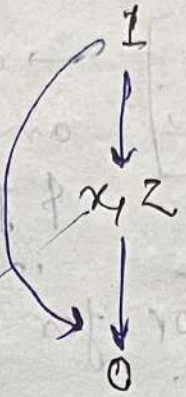
(posedge)



{0, x, z} to 1

0 to {x, z}

(negedge)



{1, x, z} to 0

1 to {x, z}

ex:-

always @ (posedge x)

{

=

=

=

@ (in) // "in" changes

@ (a or b or c) // any of "a", "b", "c" changes

@ (a, b, c) // -- do --

@ (posedge clk) // positive edge of "clk"

@ (posedge clk or negedge reset) // positive edge of "clk" or negative edge of "reset"

@ (*) // any variable changes

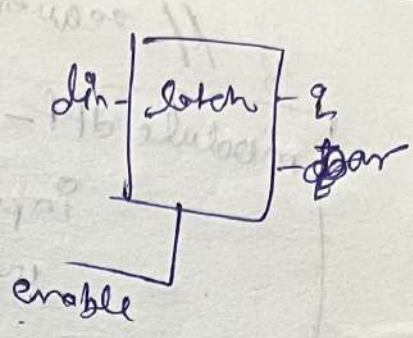
// D flip-flop with synch set and Reset

```
module dff (q, qbar, d, set, Reset, clk);
  input d, set, Reset, clk;
  output reg q; output qbar;
  assign qbar = ~q;
  always @ (posedge clk)
    begin
      if (reset==0)
        q<=0;
      else if (set==0)
        q<=1;
      else q<=d;
    end
endmodule
```

// set, Reset are active low

// Transparent latch with enable

```
module latch (q, qbar, din, enable);
  input din, enable;
  output reg q ; output qbar;
  assign qbar = ~q;
always @( din or enable)
  begin
    if (enable)
      q=din;
  end
endmodule
```



Procedural Assignment examples :-

// A combinational logic example

```
module mux21 (in1, in0, s, f);
    input in1,in0, s;
    output reg f ;  // declaring f as reg because
                    inside always blocks f
always @ (in1 or in0 or s)   is on left side.
    if (s)
        // if s is true s→1
        f=in1;
    else
        f=in0;
endmodule
```

→ The event expression in always blocks triggers whenever at least one of in1, in0 or s changes.

alternate ways to identify the event condition :-

(1) always @ (in1, in0, s)

(2) always @ (*) → means any i/p changes

// sequential logic example

```
module dff_negedge (d, clock, q, qbar);
    input d, clock;
    output reg q, qbar;
    always @ (negedge clock)
        begin
            q = d;
            qbar = ~d;
        end
endmodule
```

// 4-bit counter with asynchronous reset

```
module counter (clk, rst, count);
    input clk, rst;
    output reg [3:0] count;
    always @ (posedge clk or posedge rst)
    begin
        if (rst)
            count <= 0;
        else
            count <= count + 1;
    end
endmodule
```

→ The event condition triggers when either a positive edge of "clk" comes or a positive edge of "rst".

in 4 bit counter with synchronous reset

always @ (posedge clk) ~~or posedge rst~~

always @ (_ _ _)
case ( )

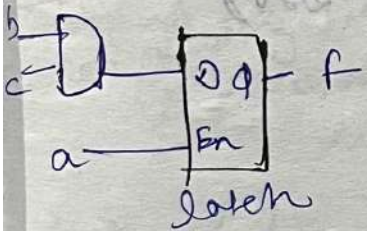
endcase.

→ when a "case" statement is incompletely decoded, the synthesis tool will infer the need for a latch to hold the residual output when the select bits take unspecified values.

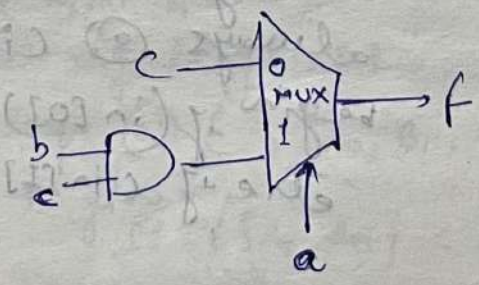
ex :-

```
module xyz (input a,b,c, output reg f);
  always @ (*)
    if (a==1) f=b&c;
endmodule
```

→ for a=0, value of f is unspecified.




```
module xyz (input a,b,c, output reg f);
  always @ (*)
    begin
      f=c;
      if (a==1) f=b&c;
    end
endmodule
```



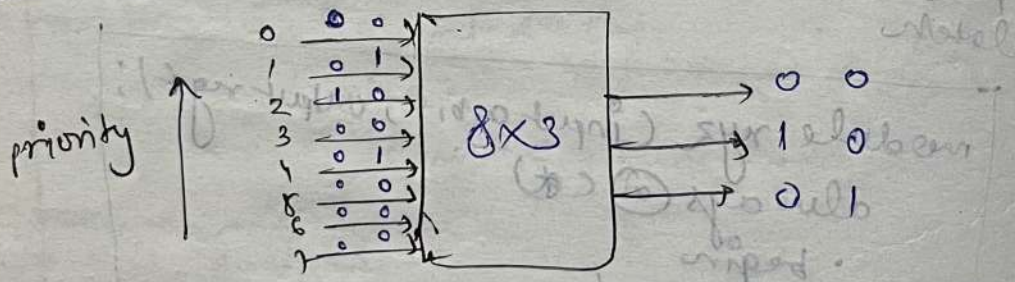
// A simple 4-function ALU

```
module ALU_4bit (f, a, b, op);
  input [1:0] op;  input [7:0] a, b;
  output reg [7:0] f;
parameter ADD = 2'b00, SUB = 2'b01, MUL = 2'b10,
                                DIV = 2'b11;
  always @(*)
    case (op)
      ADD: f = a+b;
      SUB: f = a-b;
      MUL: f = a*b;
      DIV: f = a/b;
    endcase
endmodule
```

---

op specifies 1 of 4 operations (ADD, SUB, MUL, DIV)

Priority encoder :-




```
module priority_encoder (in, code);
  input [7:0] in;
  output reg [2:0] code;
  always @ (in)
  begin
    if (in[0]) code = 3'b000;
    else if (in[1]) code = 3'b001;
    :
    :
    else
    :
    :
    code = 3'bxxx;
end
endmodule
```

→ "in[0]" → highest priority

→ for simultaneously active inputs, the first active input encountered will be encoded.

Bcd to 7 segment display :-

```
module bcd_to_7seg (bcd, seg);
    input [3:0] bcd;
    output reg [6:0] seg;
    always @(bcd)
    case
        0 : seg = 6'b0000001;
        1 : seg = 6'b1001111;
        2 : seg = 6'b0010010;
        3 : seg = 6'b0000110;
        .
        .
        9 : seg = 6'b0000100;
    default : seg = 6'b1111111;
    endcase
endmodule
```

(segment bit assignment)
(a, b, c, d, e, f, g)

→ A segment glows when corresponding bit of seg is zero.

for 0,

g will not glow

↓

g → 1

seg (0 0 0 0 0 0 1)
a b c d e f g

// A 2 bit comparator

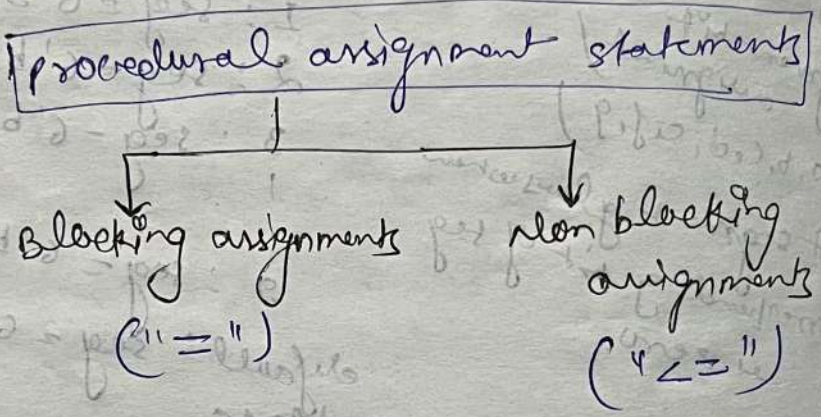
```
module compare (A1, A0, B1, B0, lt, gt, eq);
    input A1, A0, B1, B0;
    output reg lt, gt, eq;
    always @ (A1, A0, B1, B0)
    begin
        lt = ({A1, A0} < {B1, B0});
        gt = ({A1, A0} > {B1, B0});
        eq = ({A1, A0} == {B1, B0});
    end
endmodule
```

Procedural Assignment statements

→ These can be used to update the variables of types "reg", "integer", "real" or "time".

→ The value assigned to a variable remains unchanged until another procedural assignment statement assigns a new value to the variable

(different from continuous assignments, using assign, that results in the expression on the RHS to continuously drive the "net" type variable on the left.



→ procedural assignments statements can be only appear within procedural blocks ("initial" or "always")

Blocking assignment :-

syntax :- variable_name = [delay_or_event_control] expression;

"=" operator is used to specify blocking assignment.

→ These assignment statements are executed in the order they are specified in procedural block.

→ They donot block execution of statements in other procedural blocks.

→ recommended style for modeling combinational logic.

→ These assignments can also generate sequential ckt elements during synthesis (e.g. incomplete specification with "case")

example :-

```
module blocking_assgn;
integer a,b,c,d;
always @(*)
  repeat (4)          //repeat block 4 times
  begin
  #5 a=b+c;
  #6 d=a-3;
  #5 b=d+10;
  #5 c=c+1;
end
```

```
module blocking_assgn_test;
initial
  begin
  $monitor($time, "a=%4d, b=%4d, c=%4d, d=%4d",
    a,b,c,d);
  a=30; b=20; c=15; d=5;
  #100 $finish;
  end
endmodule
```

Non blocking assignment :-

syntax :- variable_name <= [delay_or_event_control] expression;

→ the assignment to the target gets scheduled for the end of the simulation cycle (at end of the procedural block)

→ statements subsequent to the instruction under consideration are not blocked by the assignment.

→ allows concurrent procedural assignment, suitable for sequential logic.

→ recommended for modelling sequential logic.

ex :-

```
integer a,b,c;
initial
  begin
    a=10; b=20; c=15;
  end
initial
  begin
    a <= #5 b+c;
    b <= #5 a+5;
    c <= #5 a-b;
  end
```

at time 5
a = 35
b = 15
c = -10

} evaluated together, assigned together at time 5

Swapping values of two variables "a" and "b" :-

```
always @ (posedge clk)
  a=b;
always @ (posedge clk)
  b=a;
```

→ either a=b will execute before b=a, or viceversa, depending on simulator.
→ Both registers will get same value (either "a" or "b")
(Race condition)

```
always @ (posedge clk)
  a<=b;
always @ (posedge clk)
  b<=a;
```

→ variables are correctly swapped
→ all RHS variables are read first and assigned to LHS variables at positive edge of clk

To swap using blocking assign :-

```
always @ (posedge clk)
  begin
    a=b;
    b=a;
  end
```

→ both a and b will be getting value previously stored in b.
=

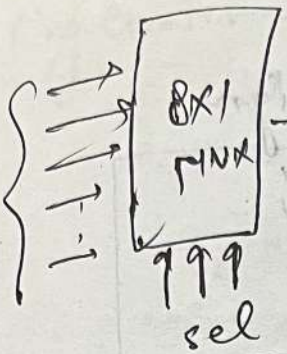
```
always @ (posedge clk)
  begin
    ta=a;
    tb=b;
    a=tb;
    b=ta;
  end
```

→ correct swapping but need two temporary variables ta and tb.

some Rules to be followed :-

→ blocking & non blocking assigns are not mixed in same block.
→ verilog synthesizer ignores the delay specified in a procedural assignment statement.
↳ may lead to functional mismatch b/w design model and synthesized netlist
→ A variable can't appear as the target of blocking and non blocking

```
x=x+5;
x<=y;
```
→ not permissible.



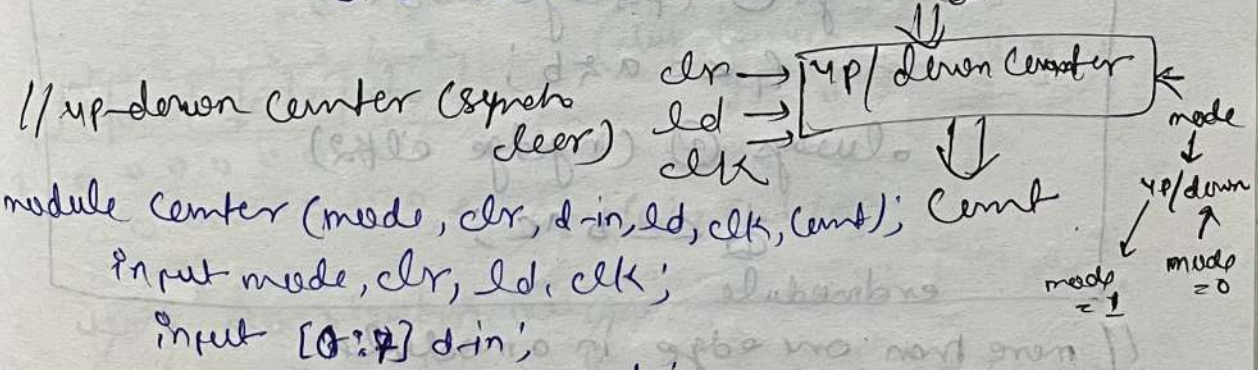
Dataflow modelling :-

```
assign out = in[sel];
```

MUX

// 8-to-1 mux : behavioral description

```
module mux_8to1 (in, sel, out);
  input [7:0] in;
  input [2:0] sel;
  output reg out;
  always @ (*)
    begin
    case (sel)
    3'b000 : out = in[0];
    3'b001 : out = in[1];
      :
      :
    3'b111 : out = in[7];
    default : out = 1'bx;
    endcase
  end endmodule
```



// up-down counter (synchronous clear)

```
module counter (mode, clr, d_in, ld, clk, count);
  input mode, clr, ld, clk;
  input [0:7] d_in;
  output reg [0:7] count;
  always @ (posedge clk)
    if (ld)    count <= d_in;
    else if (clr) count <= 0;
    else if (mode) count <= count + 1;
    else          count <= count - 1;
endmodule
```

(here no statements are not concurrent because of if else)

// Parameterized design : an N bit counter
↳ to generalize the design

```
module counter (clear, clock, count);
    parameter N=7;
    input clear, clock;
    output reg [0:N] count;
    always @ (negedge clock)
        if (clear)
            count <= 0;
        else count <= count + 1;
endmodule
```

→ parameter values are substituted before simulation or synthesis.

// using more than one clk in a module

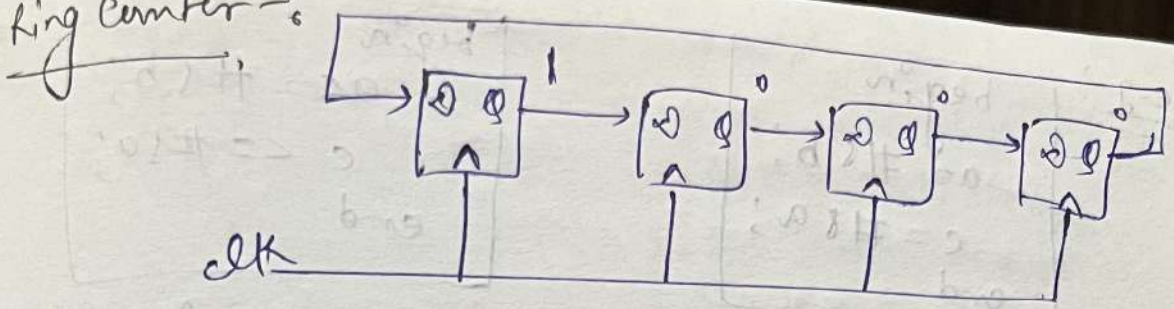
```
module multiple_clk (clk1, clk2, a, b, c, f1, f2);
    input clk1, clk2, a, b, c;
    output reg f1, f2;
    always @ (posedge clk1)
        f1 <= a & b;
    always @ (negedge clk2)
        f2 <= b ^ c;
endmodule
```

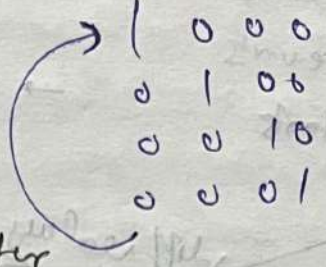
// more than one edge in a module

```
module multi_edge clk (a, b, c, d, f, clk);
    input clk; input [7:0] a, b, c, d;
    output reg [7:0] f;
    always @ (posedge clk)
        c <= a + b;
    always @ (negedge clk)
        f <= c - d;
endmodule
```

Two operations are carried out every clock cycle.
→ c is assigned at rising edge
→ f is at falling edge

Ring counter :-



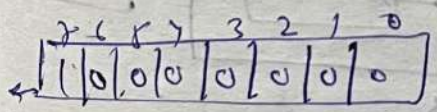


// 8bit ring counter

```
module ring_counter (clk, init, cout);
  input clk, init;
  output reg [7:0] count;
  always @ (posedge clk)
    begin
      if (init) count = 8'b10000000;
      else begin
        count = count << 1;
        cout[0] = count[7];
      end
    end
endmodule
```

blocking output

(Not correct for functionality)

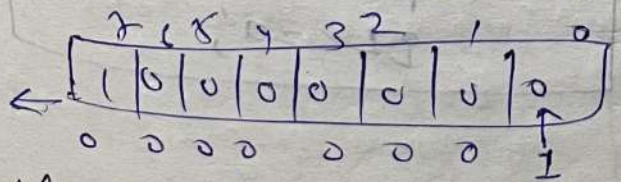


1 0 0 0 0 0 0 0 0 ← count = count << 1

0 0 0 0 0 0 0 0 ← cout[0] = cout[7]

using non blocking assignment :-

```
count <= count << 1;
cout[0] <= cout[7];
```



both concurrently

using blocking :-

```
    else
      count = {count[6:0], count[7]};
  end
endmodule
```

e.g:-

```
begin
  a = #5 b;
  c = #5 a;
end
```

→ here value of b will be assigned to c 10 time units after the block starts

```
begin
  a <= #5 b;
  c <= #5 a;
end
```

→ a gets value of b after 5 time units.

→ c gets value of a after 5 time units

c ← b → different values of "c" ← c ← a

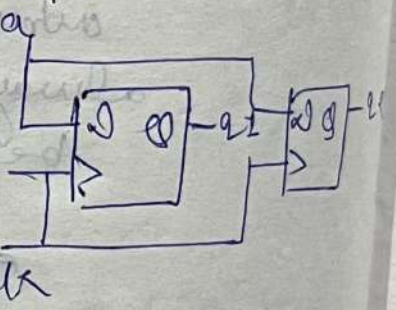
=

e.g →

blocking →

```
always @ (posedge clk)
  begin
    q1 = a;
    q2 = q1;
  end
```

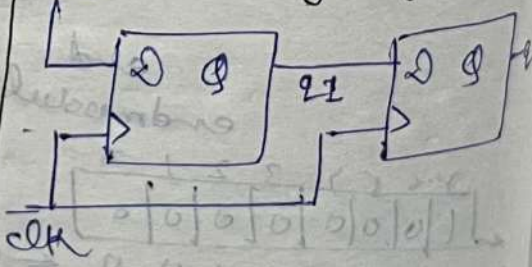
parallel flip flops




```
always @ (posedge clk)
  begin
    q1 <= a;
    q2 <= q1;
  end
```

→ non blocking

shift register



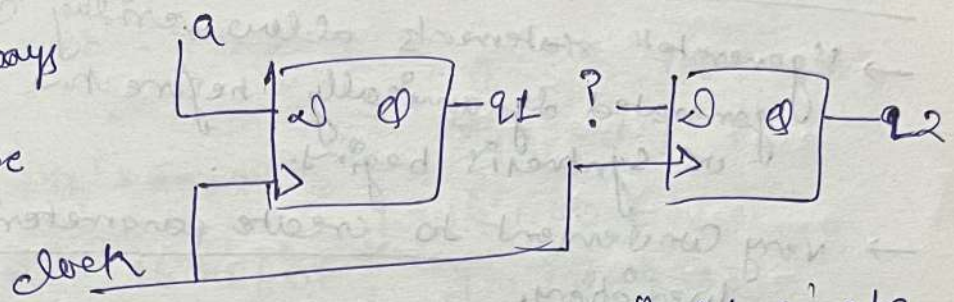
(same operation)

```
always @ (posedge clk)
  q2 <= q1
always @ (posedge clk)
  q1 <= a;
```

eg:

```
always @ (posedge clock)
  q1 = a;
always @ (posedge clock)
  q2 = q1;
```

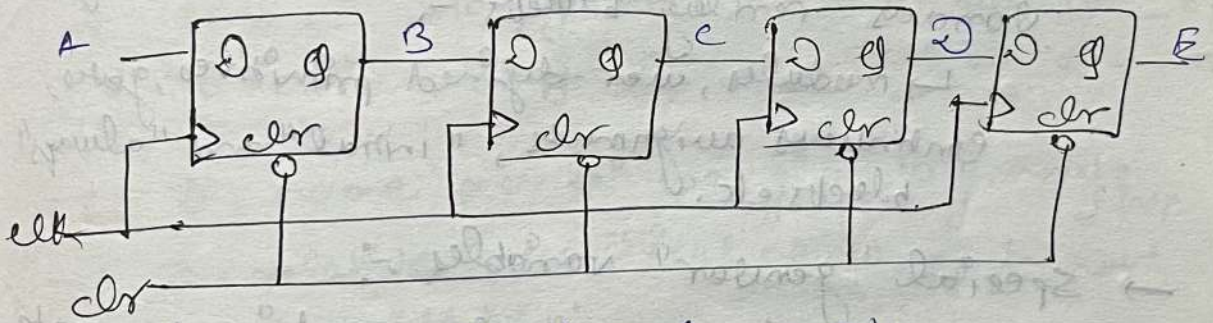
Due to always blocks, they will be executed concurrently.



so q2 can be previous value of q1 or value of a given to q1.

q2 → indeterminate

eg: shift Register




```
module shiftreg_4bit (clock, clear, A, E);
  input clock, clear, A;
  output reg E;
    reg B, C, D;
always @ ( posedge clk or negedge clear)
  begin
    if (!clear)
      begin B=0; C=0; D=0; E=0;
        end
    else begin
          E=D;
          D=C;
          C=B;
          B=A;
        end
  end
endmodule
```

ex:

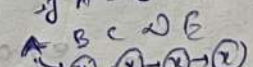
A	B	C	D	E
0	!	?	!	!
0	0	!	?	!
0	0	0	!	?

acting as single F/F

but if

```
B=A;
C=B;
D=C;
E=D;
```

if A=x



E=A

→ using non blocking ~~stat~~ assignments, statements can occur in any order.

Generate Blocks :-

→ "generate" statements allow verilog code to be generated dynamically before the simulation or synthesis begins.

→ very convenient to create parameterized module discription.

→ Example - N bit ripple carry adder for arbitary value of N.

→ Requires the keywords "generate" and "endgenerate".

→ Generate instantiations can be carried out for various verilog blocks :-

↳ modules, user-defined primitives, gates, continuous assignments, "initial" and "always" blocks etc.

→ Special "genvar" variables :-

→ to declare variables used in generate block.

→ These variables don't exist during simulation or synthesis.

→ value of "genvar" defined only in generate loop.

→ every generate loop assigned a name.

Example -

```
module xor_bitwise (f, a, b);
    parameter N=16;
    input [N-1:0] a, b;
    output [N-1:0] f;
    genvar p;
  generate for (p=0; p<N; p=p+1)
      begin xorblp
          xor xg (f[p], a[p], b[p]);
      end
  endgenerate
endmodule
```

```
module generate_test;
  reg [15:0] x, y;
  wire [15:0] out;
xor_bitwise c1 (.f(out), .a(x), .b(y));
  initial
    begin
      $monitor ("x : %b, y = %b, out = %b",
                x, y, out);
      x = 16'haaaa ; y = 16'h0000;
#10 x = 16'h0f0f ; y = 16'h3333;
  #20  $finish;
    end
endmodule
```

simulation results :

```
x : 1010 ---- 10 , y : 00 ---- 11, out : 10 -- 01
x : 00 ---- 11 , y : 00 ---- 11, out : 00 -- 00
```

→ The name "xorlp" was given to generate loop.

→ The relative hierarchical names of the xor gates will be :

xorlp[0].xg, xorlp[1].xg, ---, xorlp[15].xg

example :- Design of N bit Ripple carry Adder

```
// structural gate-level description of full Adder
module full_adder (a, b, c, sum, cout);
  input a, b, c;
  output sum, cout;
wire t1, t2, t3;
xor g1 (t1, a, b), g2 (sum, t1, c);
and g3 (t2, a, b), g4 (t3, t1, c);
  or g5 (cout, t2, t3);
endmodule
```

```
module RCA (carry_out, sum, a, b, carry_in);
    parameter N=8;
input [N-1:0] a, b;     input carry_in;
output [N-1:0] sum;     output carry_out;
wire [N:0] carry;       // carry[N] is carryout
  assign carry[0] = carry_in;
  assign carry_out = carry[N];
  genvar i;
    generate for (i=0; i<N; i+1)
        begin fa_loop
            wire t1, t2, t3;
    xor u1 (t1, a[i], b[i]), u2 (sum[i], t1, carry[i]);
  and u3 (t2, a[i], b[i]), u4 (t3, t1, carry[i]);
    or u5 (carry[i+1], t2, t3);
        end
    endgenerate
endmodule
```

→ instance names generated are :-

fa_loop[0].u1, fa_loop[1].u1 --- etc

some of nets (wires) :-

fa_loop[0].t1 ; fa_loop[1].t2 etc.

User defined primitives (UDP):

→ To define behaviour in terms of lookup tables.

→ They can specify:
  → Truth Table for combinational functions
  → State Table for sequential functions.
  → don't care, rising and falling edges, etc can be specified.

→ for combinational functions, truth table entries:
  <input1> <input2> ... <inputN> : <output>;

→ for sequential functions, state table entries are:
  <input1> <input2> ... <inputN> : <present_state> : <next_state>;

→ input terminals to a UDP can only be scalar variables and input entries in table must be in same order as the "input" terminal list.

→ only one scalar output terminal must be used

→ for combinational UDP's, the output terminal is declared as "output."

→ for sequential UDP's output terminal is declared as "reg".

→ for sequential UDP's, the state can be initialized with an "initial" statement (optional).

→ UDPs model functionality only (not timing or process technology)

→ A functional block can be modeled as UDP only if it has exactly one output.

for more than one O/P → it has to be modeled as a module.

(as an alternative, multiple UDPs can be used, one per O/P)

→ inside simulator, a UDP is typically implemented as lookup table in memory.

→ for unspecified cases in table, O/P is set to "X".

example :- // full adder sum generation using udp

```
primitive udp_sum (sum, a, b, c);
    input a, b, c;
    output sum;
    table
    //  a  b  c  :  sum
        0  0  0  :  0 ;
        0  0  1  :  1 ;
        0  1  0  :  1 ;
        0  1  1  :  0 ;
        1  0  0  :  1 ;
        1  0  1  :  0 ;
        1  1  0  :  0 ;
        1  1  1  :  1 ;
    endtable
endprimitive
```

→ we can specify don't care input combination as "?"

```
// full adder carry generation
primitive udp_carry (cout, a, b, c);
    input a, b, c;
    output cout;
    table
    //  a  b  c  :  cout
        0  0  0  :  0 ;
        0  0  1  :  0 ;
        0  1  0  :  0 ;
        0  1  1  :  1 ;
        1  0  0  :  0 ;
        1  0  1  :  1 ;
        1  1  0  :  1 ;
        1  1  1  :  1 ;
    endtable
endprimitive.
```

```
// full adder carry
// using don't care ("?")
primitive udp_cy (cout, a, b, c);
    input a, b, c;
    output cout;
    table
    //  a  b  c  :  cout
        0  0  ?  :  0 ;
        0  ?  0  :  0 ;
        ?  0  0  :  0 ;
        1  1  ?  :  1 ;
        1  ?  1  :  1 ;
        ?  1  1  :  1 ;
    endtable
endprimitive
```

using don't care reducing the rows to make design compact.

// instantiating UDP's

// A full adder description

```
module full_adder (sum, cout, a, b, c);
    input a, b, c;
    output sum, cout;
    udp_sum SUM (sum, a, b, c);
    udp_cy CARRY (cout, a, b, c);
endmodule
```

Seq. ckt example :-

```
// D type level sensitive latch
primitive dlatch (q, d, clk, clr);
    input d, clk, clear;
    output reg q;
    initial
        q = 0; // optional
    table
    // d  clk  clr  :  q  :  q_new
        ?   ?    1   :  ?  :  0;   // latch cleared
        0   1    0   :  ?  :  0;   // latch reset
        1   1    0   :  ?  :  1;   // latch set
        ?   0    0   :  ?  :  -;   // retains previous state
    endtable
endprimitive
```

// A T flip-flop

```
primitive TFF (q, clk, clr);
  input clk, clr;
  output reg q;
  table
  // clk  clr   q   q_new
       ?    1   :  ?  :  0 ;  // ff cleared
       ?   (10) :  ?  :  - ;  // ignore -ve edge of clr
     (10)   0   :  1  :  0 ;  // ff toggles at -ve edge
     (10)   0   :  0  :  1 ;  // -do-
     (0?)   0   :  ?  :  - ;  // ignore posine edge of clk
  endtable
endprimitive
```

(10) → means 1 to 0 in clr

(in clk if (10) → means falling edge)

// constructing a 6 bit ripple counter using TFF :-

```
module ripple_center (Qcount, clk, clr);
  input clk, clr;
  output [5:0] Qcount;

  TFF f0 (Qcount[0], clk, clr);
  TFF f1 ( Qcount[1], Qcount[0], clr);
  :
  :
  TFF f5 (Qcount[5], Qcount[4], clr);
endmodule
```

// A negative edge sensitive JK flf

```
primitive JKff (Q, J, K, clk, clr);
    input J, K, clk, clr;
    output reg Q;
table
// J   K   clk  clr   Q   Q_new
   ?   ?   ?    1   : ? : 0;  // clear
   ?   ?   ?   (10) : ? : -;  // no change
   ?   ?   ?    0   : ? : -;  // no change
   0   0  (10)  0   : ? : 0;  // reset condn
   0   1  (10)  0   : ? : 1;  // set condn
   1   0  (10)  0   : ? : 1;  // toggle condn
   1   1  (10)  0   : 0 : 0;  // toggle condn
   1   1  (10)  0   : 1 : 0;  // toggle condn
   1   1  (10)  0   : ? : -;  // no change
   ?   ?  (01)  0   : ? : -;
endtable
endprimitive
```

for SRFF:

```
// S   R   clk  clr   Q   Q_new
   1   1  (01)  0   : ? : x;  // invalid condn because S, R → 1
```

other table rows same as JKFF.

Some rules:-

→ "?" symbol can't be used in an o/p field.

→ "-" → no change in state value (previous value)

→ "r" indicating rising edge can be used instead of (01).

→ "f" indicating falling edge can be used instead of (10).

→ "*" indicates any value change in the signal.

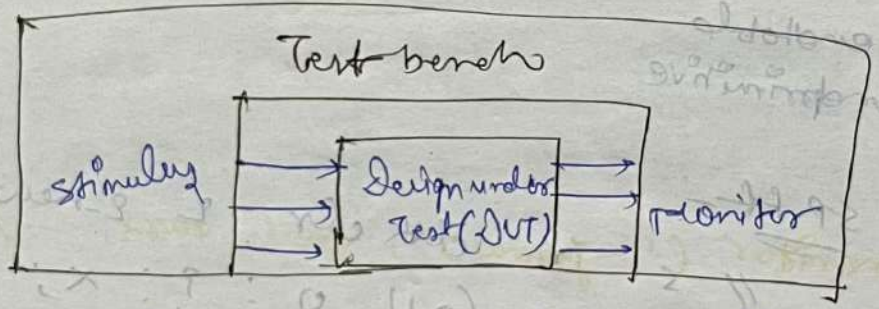
Verilog Testbench

Testbench :- A verilog procedural block that executes only once
↳ used for simulation. (Not for synthesize) so we initial blocks mostly.

→ Testbench generates clk, reset and required test vectors for a given DUT.
→ it can monitor the DUT outputs.
→ dump the values in a file from where waveforms can be viewed.

⇒ inputs and outputs of DUT need to be connected to the test bench.

can also we "always" block for generating some test inputs like a clk signal.



→ o/p of testbench connected to i/p of DUT.
and o/p of DUT is monitored using Test bench.

⇒ stimulus are code written in HDL to test the design blocks.

```
module example
  (A,B,C,D,E,f,Y);
input A,B,C,D,E,f;
output Y;
wire t1,t2,t3,Y;
nand #1 G1 (t1,A,B);
.
.
endmodule
```

```
module testbench;
reg A,B,C,D,E,f; wire Y;
example DUT (A,B,C,D,E,f,Y);
initial
  begin
    $monitor ($time, "A=%b, B=%b,
    C=%b, D=%b, E=%b, f=%b,
    Y=%b", A,B,C,D,E,f,Y);
    #5 A=1; B=0; C=0; D=1; E=0;
    #5 A=0; B=1; C=0; f=0;
    #5 $finish;
  end
endmodule
```

How to write :- (1) create a dummy template

↳ Declare i/ps to DUT as "reg" and the outputs as "wire".

(because we have to initialize the DUT i/p's inside procedural blocks (initial) where only "reg" type variables can be assigned).

↳ instantiate the DUT

(2). initialization and monitoring.

→ Assign some known values to DUT i/p's

→ monitor the DUT o/p's for functional verification.

simulator directives -: $display, $monitor, $finish, $dumpfile, $dumpvars

→ $display ("<format>", expr1, expr2 ---);

→ print the immediate values of text or variables

write → additional format specifiers are supported like "b" (binary), "h" (hexadecimal) etc.

for every line to be printed

→ $monitor ("<format>", var1, var2, ---);

→ does not print immediately.

write only once → print when any variable changes

$monitor display ("%d %b %h", a, b, c);

reg a;
integer b;
reg [7:0] c;

$monitor ("%d %b %b", $time, a, b);

(simulation time)

$dumpfile (<filename>); → specifies the file that will be used for storing the values of selected variables so they can be visualized later

(.vcd file) ← (value change dump)

$dumpvars; → to start dumping of variables

$dumpoff; → to stop dumping of variables

→ $dumpvars (level, list_of_variables_or_modules);

if level=0, dumping will be done at module level.
if level=1, only listed variables will be dumped.

$dumpvars (0, module1, module2);

$dumpvars (1, a, b, c, d);

→ $dumpall; → current values of all variables will be written to the file.

→ $dumplimit (filesize);
↳ used to set the max size of .vcd file.

ex: 2 bit binary up counter

Code:

```
module counter (clear, clock, count);
  parameter d=2;
  input clear, clock;
  output reg [0:d] count;
  always @ (negedge clock)
    if (clear)
      count <= 0;
    else
      count <= count+1;
  endmodule
```

Testbench:

```
module testbench_counter;
  reg clk, clear;
  wire [1:0] out;
  counter cnt (clk, clear, out);
  initial clk = 1'b0;
  always clk = #clk;
  initial begin clr=1'b1;
    #15 clr = #
```

#10 $finish;