NOTES

VERILOG

Hardware Modeling Using Verilog

→ Design complexity increasing rapidly
① Increasing size and complexity
② Fabrication technology improving
③ CAD tools are essential
④ Conflicting requirements.

→ The present trend
① Standardize the design flow
② Emphasis on low-power design, and increased perfor...

Moore's law → Exponential growth / increase in no. of transistors per unit area

→ Today technology → CMOS
→ FinFET
Future → Quantum.

→ Standardized design procedure → starting from the design idea down to actual implementation

Steps:- ① Specification
② Synthesis
③ Simulation
④ Layout
⑤ Testability analysis
⑥ and more.

→ Computer Aided Design (CAD) tool
① Based on hardware Description language (HDL)
② A CAD tool transforms HDL input into RTL output that contains detailed information about the hardware
① Behavioral level to register transfer level
② Register transfer level to gate level.
③ Gate level to transistor level.
④ Transistor level to layout level.

Two competing HDLs → Verilog / VHDL

Design flow

↳ Design idea ⇄ Behavioral Design (flowgraph / Pseudo Code) ⇒ Data path design (Bus/Register) → Logic Design (Gate/FF) → Physical Design ⇄ Logic Design

Physical Design → Manufacturing (chip/Board); Physical Design → Transistor Layout

Behavioral Design — ① Boolean expression or truth table
② Finite state machine behavior
③ In the form of High-level algorithm

Data path design → Generate a netlist of register transfer level components, like register, adder, multiplier, multiplexer.

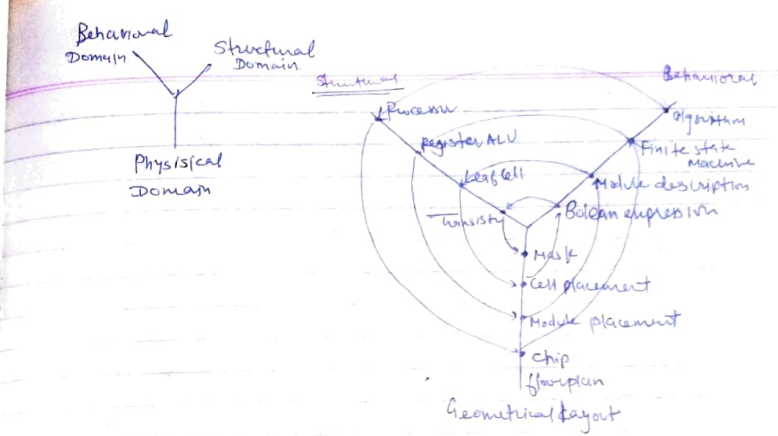
netlist →

Logic design → Generate a netlist of gates / flipflop or standard cells

Design Representation

- A design can represent in different levels from three view points ① Behavioral ② Structural ③ Physical

↳ Conveniently expressed in Y shape



Behavioral Representation

→ Specifies how a particular design should respond to a given set of input

↳ ① Boolean equation
② Table of input and output value
③ Algorithm written in standard HLL like C
④ Algorithm written in special HDL like verilog or VHDL

eg Full adder → ① two operand input A and B
② a carry input C
③ a carry output C_y
④ a sum output S

Boolean → $S = A \oplus B \oplus C$

$C_y = AB + B.C + C.A$

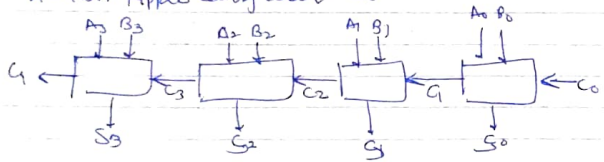
Verilog in terms of Boolean Expression

```
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);
endmodule
```

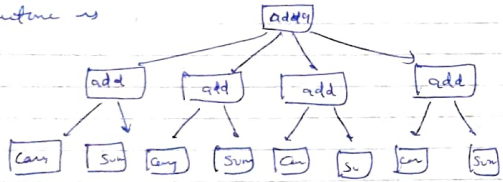
Structural Representation

↳ specifies how components are interconnected
↳ interconnection of netlist
↳ Specified in various level
- → Module level
- → Gate level
- → transistor level

e.g. A-4 Bit Ripple Carry adder →



structure →




```
module add4 (S, Cy4, Cyin, x, y);
  input [3:0] x, y;
  input Cy_in;
  output Cy4;
  output [3:0]S;
  wire [2:0] Cy_out;
  add B0 (Cy_out[0], S[0], x[0], y[0], 4 );
  |
  |
  add B3 (                          );
```

→ we already made ~~dis~~ this function, • it is called instantiation as it makes hardware copy & works (in we calls function calling)

Physical Representation

↳ lowest level of physical specification → photo mask information required by the various processing steps in the fabrication process.

→ Note :- To describe a digital system as a set of module
Modules are interconnected.

→ Two ways to specify a module
① Specifying its internal logic structure (structural Representation)
② describing its behavior (Behavioral Representation)

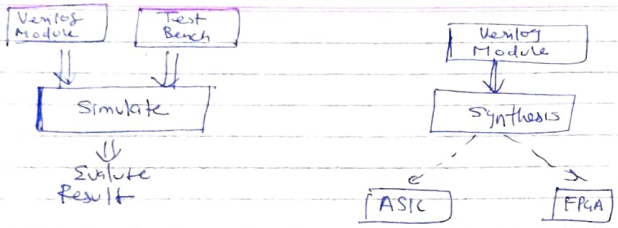
Module 1 { Module 1 ... end module 1

Module 2

Modul 3

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

- Simulate the system and verify the operation → need test bench
- Use a synthesis tool to map it to hardware
  - Application Specific Integrated Circuit (ASIC) → not need test bench
  - Field programmable Gate array (FPGA)



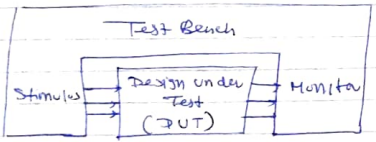
How to simulate verilog Module

→ ① Using a test bench to verify the functionality of a design Coded in Verilog (called Design-Under-Test or DUT), Composing of:

② → A set of stimulus for the DUT

→ A monitor which Capture or analyses the outputs of the DUT.

②



Command in iVerilog

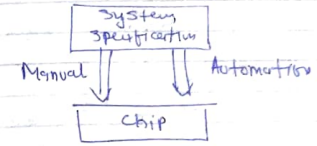
a) iverilog -o mysim example.v example_test.v

b) vvp mysim

To display the waveform

→ gtkwave gtkwave example.vcd

VLSI Design Styles



→ VLSI Design Cycle

① System specification
② Functional design
③ Logic design
④ Circuit design
⑤ Physical design
⑥ Design verification
⑦ Fabrication
⑧ Packaging, testing

Convert circuit description into a geometric description

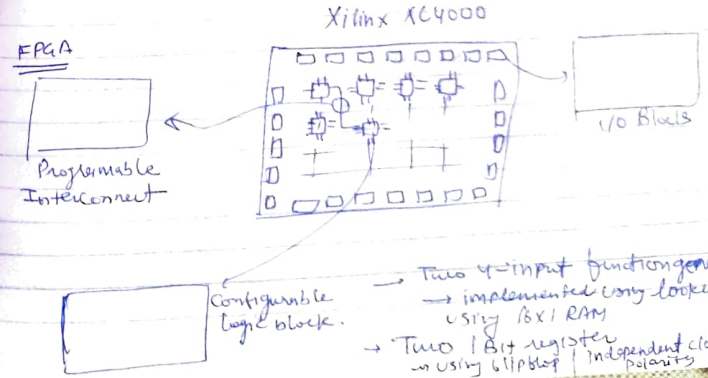
① Partitioning, floorplanning and placement
② Routing
③ Static timing analysis
④ Signal integrity and crosstalk analysis
⑤ Physical verification and signoff

Various design styles →

① Programmable
- ① FPGA
- ② Gate array

② Standard cell (Semi custom design) } ASIC

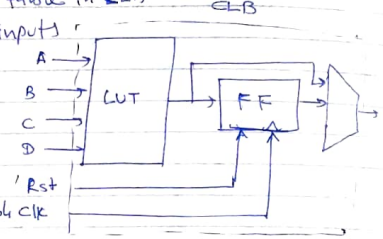
③ Full custom design



Look up table (LUT)

① ↳ Combinational output can be store in 16×1 SRAM look up table in CLB

② Capacity is limited by no of inputs



LUT Mapping eg

↳ f = A'B + B'.C.D

↳ Create truth table of available clk

↳ Load output in SRAM

↳ Apply input and get output

→ Use LUT such a way that delay is minimum

Gate Array → Come after FPGA

↳ it is view in speed of prototyping capabilities.

design implementation → FPGA → done by user programming

→ Gate array → done with metal mask design and processing

two step manufacturing

① developed transistor array but not inter-connected.

② Based on design, inter connect transistor

↳ So, utilization of hardware is more than FPGA

↳ Speed is higher.

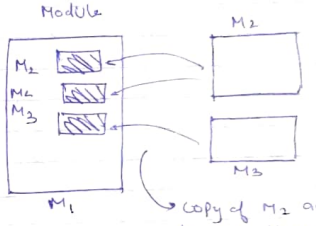
Verilog features

↳ In verilog, the basic unit of hardware is called module

→ Module cannot contain definition of other module

→ Module can be intentiated with another module

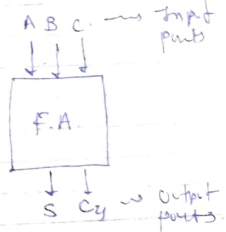
eg



→ Instantiation allows the creation of hierarchy in verilog description

Module syntax

```
module module_name (list_of_ports);
    input and output declaration
    local net declaration
    parallel statements
endmodule
```



eg

```
module simpleand(f, x, y);
    input x, y;
    output f;
    assign f = x & y;
endmodule
```

↳ So, this is the behavioral description. So synthesis tool will decide how to realize f

① SIMPLE AND

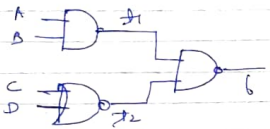
② NAND with NOT Gate

eg :- A 2-level combinational circuit

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

→ This is also a behavioural description.

→ One possible gate level realize is shown



→ intermediate line

Note :- The 'assign' statement represents continuous assignment, whereby the variable on the LHS get updated whenever the expression on the RHS changes

```
assign variable = expression ;
```

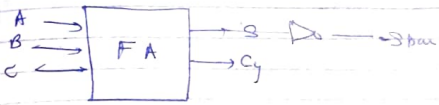
→ LHS → "net" type variable, typically a "wire"

→ RHS → can both "register" or "net" variable.

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

Data type in Verilog

a) Net
- Must be continuous driven
- can't be used to store value
- Used to model connections b/w continuous assignments and instantiation.




```
wire Sbar
assign Sbar = ~S
```

b) Registers
- Retains last value assigned to it
- Often used to represent storage elements but sometimes it can translate to combinational circuits also

→ NET → ① Nets represents connection b/w hardware elements
② Nets are continuously driven by the output of the devices they are connected to

→ a is continuously driven by the output of the AND gate

③ Net is 1 bit values by default unless they are declared explicitly as vector
↳ by default value is "z" i.e High impedance state.

→ ④ Net datatype supports → wire, wor, wand, tri, supply0, supply1, etc

→ wire and tri are equivalent, when there are multiple drivers driving them, the driver outputs are shorted together.

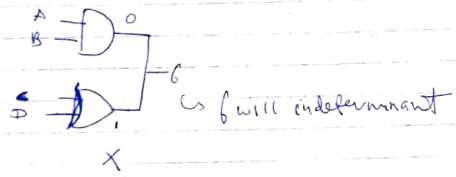
→ 'wor' and 'wand' inserts an OR and AND gate respectively at the connection.

→ 'supply0' and 'supply1' model power supply connection

→ ~~Net~~ wire net are most common.

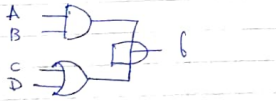
```
module use_wire (A,B,C,D,f) ;
  input A,B,C,D ;
  output f;
  wire f;   // net f declared as 'wire'
  assign f = A & B;
  assign f = C | D;
endmodule
```

} are can't use this type of module



↳ f will indeterminant

```
module use_wire (A,B,C,D,f) ;
  input A,B,C,D ;
  output f;
  wand f;
  assign f = A & B;
  assign f = C | D;
endmodule
```



f = (A&B) & (C|D)

```
module using_supply_wire (A,B,C,f)
  input A,B,C;
  output f;
  supply0 gnd;
  supply1 vdd;
  nand G1(f1, vdd, A,B);
  nor  G2(f2, C, gnd);
  and  G3(f, f1, f2);
endmodule
```

→ supply0 and supply1 has high signal strength.

Data Values and signal strengths

↳ Verilog supports 4 value levels and 8 strength level to model the functionality of the real hardware

→ Strength levels are typically used to resolve conflicts b/w signal driven of different strengths in real circuits

value level → 0 → Logic 0 state
1 → " 1 "
X → unknown logic state
Z → High impedance ~~logic~~ state

At initialization → all unconnected nets are set to 'z'
→ All register variables set to "x"

→ If two signals ~~and~~ of unequal strengths get driven on a wire, the stronger signal will prevail.

Register Data type

→ a 'register' can hold a value.

→ It does not necessarily mean that it will map to a hardware register during synthesis.

→ Combinational circuits specifications can also use register type variable.

Register Data type — ① ~~reg~~ reg —
② integer — used for loop counting
③ real — used to store floating point no
④ time — keep track of simulation time

'reg' datatype
→ Default value of a "reg" datatype is "x".
→ It can assigned a value in synchronism with clock or even otherwise.
→ The declaration explicitly specifies the size (~~if~~ default is 1-bit)

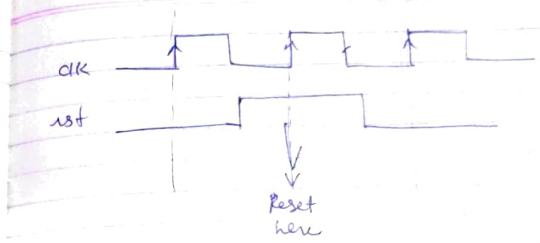
```
A = B + C;
A <= B + C;
```
here, A is ~~a~~ "reg" type

```
reg x, y;
reg [15:0] bus;
```
16 bit bus

→ Treated as unsigned number in arithmetic expressions.
→ Most used in case of → counters, shift register etc

```
module simple_counter( clk, rst, count );
  input clk, rst;
  output [31:0] count;
  reg [31:0] count;

  always @(posedge clk)
  begin
    if (rst)
      count = 32'b0;
    else
      count = count + 1;
  end
endmodule
```

→ 32 bit counter with synchronous reset
→ count value increase at positive edge of the clock
→ If "rst" is high, the counter is reset at the positive edge of the next clock.



For 32 bit counter with asynchronous ~~rst~~ reset
↳ always @(posedge clk or posedge rst)

"integer" data type
→ It is a general purpose register data type used for manipulating quantities
→ More convenient to use in situations like loop counting than "reg".
→ It is treated as 2's complement signed integer in arithmetic expression.
→ By default ~~B~~ size is 32 bits; however, the synthesis tool tries to determine the size using data flow analysis.
→ eg

```
wire [15:0] x, y;
integer c;
c = x + y;
```
• size of c can be deduced to be 17 (16 bit + carry)

"real" data type
→ Used to store floating point no.
→ When a real value is assigned to an integer, the real no. is rounded off to the nearest integer.

Ex → real e, pi ;
initial
begin
e = 2.718;
pi = 314.159E-2;
end
integer n;
initial
n = pi // n → get value 3

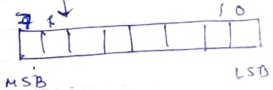
"time" datatype
→ In verilog, simulations is carried out with respect to a logical clock called simulation time
→ The "time" data type can be used to store simulation time
→ The system function "$time" gives the current simulation time
eg:
time curr_time;
initial
curr_time = $time;

Vectors
- Nets and "reg" type variable can be declared as vectors, of multiple bit widths. → If bit width is not declared, default size is 1-bit.
- Vector are declared by specifying range[range1 : range2]

Most significant bit ← range1 : range2 → Least significant bit

eg :- wire x,y,z;
wire [7:0] sum;
reg [31:0] MDR;
reg [1:10] data;
reg clock

// MSB = 7, LSB = 0
// MSB = 31, LSB = 0
// MSB = 1, LSB = 10



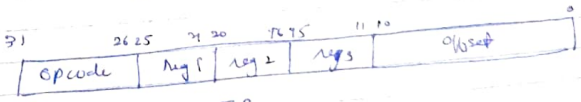
data[0]
data[1]
:

Sum[0]
Sum[1]
:
Sum[7]

→ Part of vector can be addressed and used in an expression

eg → 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];



We can do also
sum = IR[25:21] + IR[20:16]

Multi-dimensional Arrays and Memories

→ Multi-dimensional arrays of any dimension can be declared in Verilog.

eg:- `reg [31:0] reg_bank [15:0];` // 16 32-bit registers

reg_bank [5]

⇓

32 bit value

`integer matrix [7:0][15:0];`

→ Memory Model { 1D → each memory of the array is address by single array index
2D

`reg mem_bit [0:2047];` // 2k 1bit 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 & the unsized form.

→ Syntax of sized form:-
`<size>'<base><number>` { integer/real type data type expressed in unsized form.

eg:- `4'b0101` // 4-bit binary number 0101
`1'b0` // Logic 0 (1bit)
`12'hB3C` // 12 bit num. 1011 0011 1100
`12'h8xF` // 12-bit num. 1000 xxx 1111
`25` // signed no. in 32 bits (Size is not specified)

Parameter

→ A parameter is a constant with a given name
→ we can't specify the size of parameter.
→ The size gets decided from the constant value itself; if size is not specified, it is taken to be 32 bits

eg. `parameter MI = 25, LO = 5;`
`parameter up = 2'b00, down = 2'b01, steady = 2'b10;`

// 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

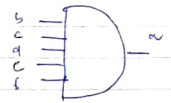
• Verilog provides a set of predefined logic gates.
→ Can be instantiated within a module to create a structured design.
→ The gates responds to logic values (0, 1, x, or z) in a logical way.

2 input AND	2 input OR	2 input 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	1 ^ 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

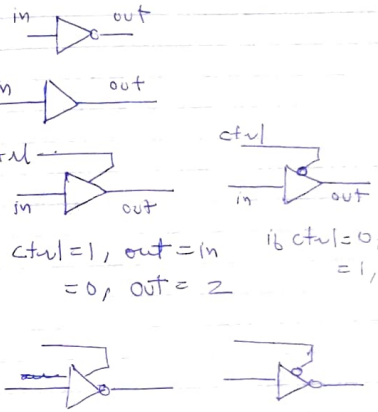
① and G (out, in1, in2);
↳ no. of input can be arbitary.

and G (a, b, c, d, e, f)
output, input



② nand G (out, in1, in2);
③ or G ( );
④ nor G ( );
⑤ xor G ( );
⑥ xnor G ( );
⑦ not G (out, in);
⑧ buf G (out, in);

⑨ bufif1 G (out, in, ctrl);
⑩ bufif0 G (out, in, ctrl);
⑪ notif0 G (out, in, ctrl);
⑫ notif1 G (out, in, ctrl);



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

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

Some restriction when instantiating primitive gates

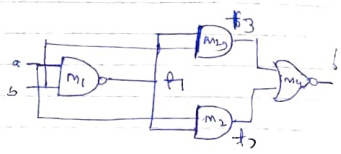
→ The output port must be connected to a net (eg wire)
- An 'output' signal is a wire by default, unless explicitly declared as a register

→ The input ports may be connected to nets or register type variable

→ When instantiating a gate, an optional delay may be specified.
- used for simulation
- Logic synthesis tools ignore the time delay

eg and #5 G1 (b, A, B);
↳ this delay has does not any meaning in synthesis purpose.

```
'timescale 1ns/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
```



→ The 'timescale directives

- Often in a single simulation, delay values in one module need to be specified in terms of some input unit, while those in some module need to be specified in terms of some other time unit
- Syntax :-

'timescale <reference_time_unit>/<time_precision>

- <reference_time_unit> : specifies the unit measurement of time
- <time_precision> : specifies the precision

eg → 'timescale 1ns/1ns

- 1ns : Reference time unit
- 1ns : Simulation precision

#5 as delay means → 5ns

time units → s (seconds), ms (millisec), us, ns, ps, fs

Specifying connectivity during instantiation

→ When a module is instantiated within another module, there are two ways to specify the connectivity of the signal lines to the two modules.

① Positional association.
The parameters of the module being instantiated are listed in the same order as in the original module description.

② Explicit association
→ The parameters of the module being instantiated are listed in arbitrary order.
→ Chances of error is less.

eg → Positional association

```
module technench;
    reg X1, X2, X3, X4, X5, X6; wire OUT;
    example DUT( X1, X2, X3, X4, X5, X6, OUT );

    initial
      begin
        ...
      end
endmodule
```

```
module example (A, B, C, D, S, F, Y);
                 ↑  ↑  ↑  ↑  ↑  ↑  ↑
                 X1 X2 X3 X4 X5 X6 OUT
    ...
endmodule
```

Explicit association

```
module testbench;
  reg X1, X2, X3, X4, X5, X6 ; wire OUT;
  example DUT (.OUT(Y), .X1(A), .X2(B), .X3(C),
    .X4(D), .X5(E), .X6(F) ) ;

  initial
  begin
    ____
    ____
    ____
    ____
  end
endmodule
```

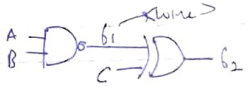
Hardware Modeling issues

→ the value computed can be assigned to
- → a "wire"
- → a latch (level triggered)
- → a flip-flop (edge triggered storage cells)

- 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 "storage cell" depending upon the content under which a value is assigned.

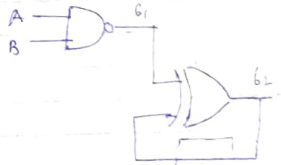
```
module reg_maps_to_wire (A,B,C, f1, f2);
  input A,B,C;
  output f1, f2;
  wire A,B,C;
  reg f1, f2;
  always @(A or B or C)   // this block will execute if A or B or C changes its state
  begin
    f1 = ~(A&B);
    f2 = f1 ^ C ;
  end
endmodule
```

→ Note:- The synthesis system will generate a wire for f1.



now change the block :-

```
begin
  f2 = f1 ^ f2
  f1 = ~(A & B)
end
```



→ Note:- The synthesis system will generate a wire for f1 and a storage cell for f2 (latch generated (A or B or C change))

another example

```
module simple_latch (data, load, d_out);
  input data, load;
  output d_out;
  wire
  always @(load or data)
```

```
begin
  if (!load)
    t = data;
  dout = !t;
end
endmodule
```

→ Here 'if' statement is not execute then data should maintain in the 't' and latch is created.

Note:- The "else" part is missing so a latch will be generated for 't'.

Verilog Operators

Arithmetic Operators

Unary	Binary		
		+	unary (sign) plus
		−	unary (sign) minus
+A	A+B	+	binary plus
−B	A * B	−	binary minus
−(A+B)		*	multiply
		/	devide
		%	modulus
		**	exponential

Logical operators

!	logical negation
&&	logical AND
||	logical OR

→ The value 0 is treated as logic false while any non-zero value is treated as TRUE.

→ Logical operator return 0(False) or 1 (True)

Relational Operators

①	!=	not equal
②	==	equal
③	>=	greater and equal
④	<=	less and equal
⑤	>	greater
⑥	<	less

Bitwise Operators

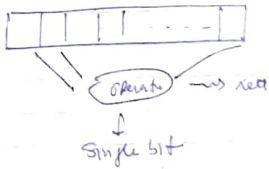
~	b ||	bitwise NOT
&	b ||	AND
|	||	OR
^	||	exclusive OR
~^	||	exclusive NOR

eg.→
```
wire a,b,c,d,f1,f2;
assign f1 = a|b;
assign f2 = (a&b)|(c^d);
```

→ Bitwise operator operates on bit and return a value that is also a bit

Reduction Operator:-

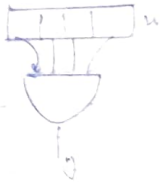
Reduction operator accepts a single word operands and produce a single bit as output.



⇒ unary

```
wire [3:0] n;  wire y;
assign y = &n;
```

reduction OP



Reduction operator

&	&	bitwise AND
|	|	bitwise OR
~&	b ||	NAND
~|	||	NOR
^	||	XOR
~^	||	XNOR

```
wire [3:0] a,b,c;
wire f1,f2,f3;
assign a = 4'b0111;
assign b = 4'b1100;
assign c = 4'b0100;
assign f1 = ^a;   → 1
assign f2 = &(a|b);   → 0
assign f3 = ~a & ~^b;   → 1
```

// An 8-bit adder description

```
module parallel_adder (sum, count, in1, in2, cin);
input [7:0] in1, in2,
input cin;
output [7:0] sum;
output cout;

assign #20 {cout, sum} = in1 + in2 + cin;
endmodule.
```

Operator Precedence

```
+  -  !  ~
   *  /
<<  >>  >>>
<  <=  >  >=
==  !=  ===  !==
&  ~&
^  ~^
|  ~|
&&
||
?:
```

Precedence increases

`==` `!=` } check for equality:

a: 1 0 1 x
b: 1 0 1 0

a == b

↳ True for this case

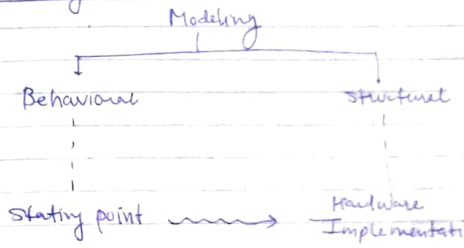
`===` `!==` } check for exact equality

↳ a === b

a: 1 0 1 x
b: 1 0 1 x

↳ Only this it will true.

Verilog Modeling Example



Example 1 → Hierarchical description of a 16 to 1 multiplexer

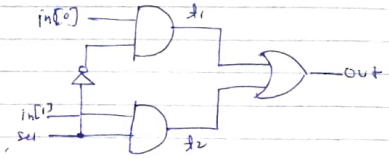
a) Using pure behavioral Modeling
b) Structural modeling using 4 to 1 multiplexer specified using behavioral model.
c) Make structural modeling of 4 to 1 multiplexer using behavioral modeling of 2-to-1 multiplexer.
d) Make structural gate-level modeling of 2-to-1 multiplexer, to have a complete structural hierarchical description.

Version 1 :- Using pure behavioral Modeling

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

Version 4 :- Structural modeling of 2to1 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
```



Example 2

Version 1 :- Behavioral description of a 16-bit adder

- Generation of status flag
  - Sign :- whether the sum is negative or positive
  - Zero :- whether the sum is zero
  - Carry :- whether there is a carry out of the last step
  - Parity :- whether the no. of 1's in sum is even or odd.
  - Overflow :- whether the sum cannot fit in 16 bits.

```
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;
  assign Zero = ~|Z;
  assign Parity = ~^Z;   // even | -> 1, odd | -> 0
  assign Sign = Z[15];
  assign Overflow = ((X[15] & Y[15] & ~Z[15]) |
                     (~X[15] & ~Y[15] & Z[15]));
endmodule
```

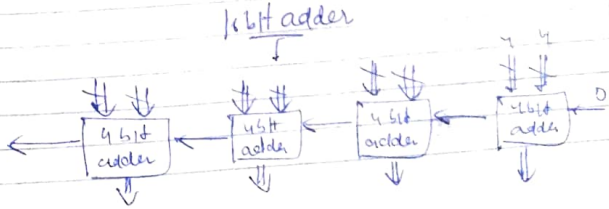
Overflow → $x.y.\bar{z} + \bar{x}.\bar{y}.z$

```
module alutest;
  reg [15:0] X, Y;
  wire [15:0] Z;  wire S, ZR, CY, P, V;
  ALU DUT(X, Y, Z, S, ZR, CY, P, V);
  initial
    begin
      $dumpfile("alu.vcd"); $dumpvars(0, alutest);
      $monitor("$time, " X = %h, Y = %h, S = %b, Z = %b, CY = %b,
                P = %b, V = %b", X, Y, Z, S, ZR, CY, P, V);
      #5 X = 16'h8666; Y = 16'h 8000;
      #5 X = 16'h 6666; Y = 16'h 0002;
      #5 X = 16'h AAAA; Y = 16'h 5555;
      #5 $finish;
    end
endmodule
```

Simulation output

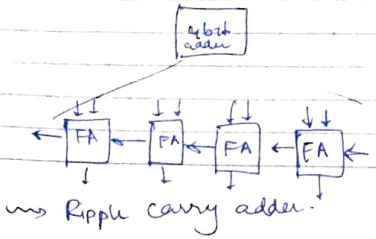
```
0   X = xxxx, Y = xxxx, Z = xxxx, S = x, Z = x, CY = x, P = x, V = x,
5   X = 8666, Y = 8000, Z = 0666, S = 0, Z = 0, CY = 1, P = 1, V = 1,
10  X = 6666, Y = 0002, Z = 6668, S = 0, Z = 0, CY = 0, P = 1, V = 0,
15  X = AAAA, Y = 5555, Z = FFFF, S = 1, Z = 0, CY = 0, P = 1, V = 0,
```

Version 2 :- Structural description of 16 bit adder using 4bit bradder block (with ripple carry b/w block)



```
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;
    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

module adder4 (S, Cout, A, B, Cin);
    input [3:0] A, B;
    input Cin;
    output [3:0] S; output Cout;

    assign {Cout, S} = A + B + Cin;
endmodule
```

Verilog Version 3



⇒ 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 AFA0 (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

module fulladder (S, Cout, a, b, c);
    input a, b, c;
    output S, Cout;
    wire S1, C1, C2;

    nor G1 (S1, a, b), G2 (S, S1, c),
        G3 (Cout, C2, C1);
    and G4 (C1, a, b), G5 (C2, S1, c);
endmodule
```

⇒ Carry to Ripple carry adder

Version 4 :- Structural Modeling of carry lookahead Adder

Note:- Generation of delay in an n-bit ripple carry adder is proportional to n.

how its work?

→ Generate the carry signal parallelly of various stages

→ Time complexity = O(1) but hardware complexity increases.

For ith stage → we define

① carry generate
$g_i = A_i \cdot B_i$

② Carry propagate
$P_i = A_i \oplus B_i$

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

P_i = 1 represents the condition when an input carry c_i will be propagated to the output carry c_{i+1}.

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

$c_{i+1} = g_i + P_i c_i = g_i + P_i (g_{i-1} + P_{i-1} c_{i-1})$

$= g_i + P_i g_{i-1} + P_i P_{i-1} c_{i-1}$

$= g_i + P_i g_i + P_i P_{i-1} (g_{i-2} + P_{i-2} c_{i-2})$

$= g_i + P_i g_i + P_i P_{i-1} g_{i-2} + P_i P_{i-1} P_{i-2} c_{i-2} \ldots$

$= \ldots$

$$c_{i+1} = g_i + \sum_{k=0}^{i-1} g_k \prod_{j=k+1}^{i} P_j + c_0 \prod_{j=0}^{i} P_j$$

Generation of the Carry and Sum bits

$C_4 = g_3 + g_2 P_3 + g_1 P_2 P_3 + g_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3$

$C_3 = g_2 + g_1 P_2 + g_0 P_1 P_2 + C_0 P_0 P_1 P_2$

$C_2 = g_1 + g_0 P_1 + C_0 P_0 P_1$

$C_1 = g_0 + C_0 P_0$

$S_0 = A_0 \oplus B_0 \oplus C_0 = P_0 \oplus C_0$

$S_1 = P_1 \oplus C_1$

$S_2 = P_2 \oplus C_2$

$S_3 = P_3 \oplus C_3$

→ module adder4(S, cout, A, B, cin)
input [3:0] A, B; input cin;
output [3:0] S; output cout;
wire p0, g0, p1, g1, p2, g2, p3, g3;
wire c1, c2, c3;

assign p0 = A[0] ^ B[0], p1 = A[1] ^ B[1],
p2 = A[2] ^ B[2], p3 = A[3] ^ B[3];

assign g0 = A[0] & B[0], g1 = A[1] & B[1],
g2 = A[2] & B[2], g3 = A[3] & B[3];

assign c1 = g0 | (p0 & cin),
c2 = g1 | (p1 & g0) | (p1 & p0 & cin),
c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & cin),
cout = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
(p3 & p2 & p1 & p0 & cin);

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

endmodule.

Description Styles in Verilog

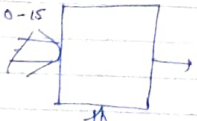
① Data flow → already studied.
↳ Continuous assignment ⇒ using assignment statement

② Behavioral
↳ Procedural assignment ⇒ using procedural statements similar to a program in High-level language
→ Blocking
→ Non Blocking

module
  signal declaration
  assign statement
  =
  =
  =
  =
endmodule

Note:- Now we will see that how synthesis tool will understand the standard behavioral statement

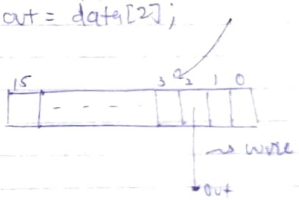
① module generate_mux (data, select, out)
input [15:0] data;
input [3:0] select;
output out;
assign out = data[select];
endmodule

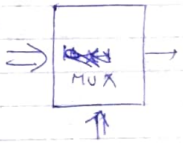


↳ Non constant index in expression on RHS generates a MUX.

Note:- whenever there is an array reference on the RHS with a variable index, a MUX is generated by the synthesis tool.

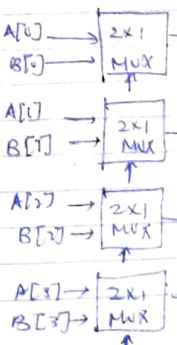
→ If the index is constant, just a wire will be generated.
eg → assign out = data[2];



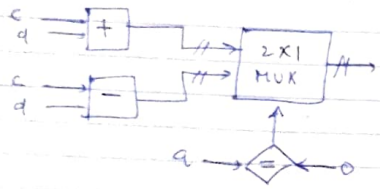


eg assign out = data[sel];

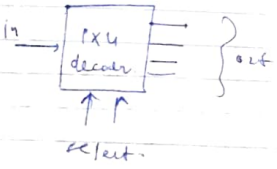
② module generate_set_of_MUX (a,b,f,sel);
input [0:3] a,b;
input sel;
output [0:3] f;
assign f = sel ? a : b;
endmodule

~~~> Conditional operator generate MUX.



Point to Note
① Whenever a conditional is encountered in the RHS of an expression, a 2 to 1 MUX is generated.
② If the input and output is vector then, (array of 2 to) MUX generate.
~~~

→ assign f = (a==0) ? (c+d) : (c-d);



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



assign out[select] = in;
endmodule

↳ Non-constant in expression on LHS generates a decoder.

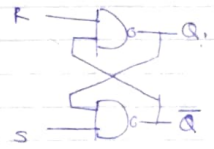
→ Point to note -
① A constant index in the expression on the LHS will not generate a decoder. → eg assign out[5] = in;

④ module level_sensitive_latch (D, Q, En);
input D, En;
output Q;
assign Q = En ? D : Q;
endmodule

En	D	Q_n
0	X	Q_{n-1}
1	0	0
1	1	1

↳ Sequential logic element using "assign" statement.

Modeling a simple S-R latch



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

NAND

a	b	y
0	0	1
0	1	1
1	0	1
1	1	0

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

→ When we make testbench for this SR-Latch and S=0, R=0, then Q=1 and Qbar=1, after this simulator hangs.

Procedural assignment

→ Two kinds of procedural block
① "Initial" block → used only in testbench.
→ Execute at beginning of the simulation
② "Always" block → A continuous loop that never terminates.

→ Procedural block is A region of code containing sequential statements.
↳ The statement execute in order they are appear written

Initial Block

```
Initial
  begin
    ==========
    ==========
    ==========
  end
```

→ Start at time = 0 and execute only once

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

eg.
```
module testbench_example;
  reg a,b, cin, sum, cout;
  initial
    cin = 1'b0;

  initial
    begin
      #5 a = 1'b1; b = 1'b1;
      #5 b = 1'b0;
    end

  initial
    #25 $finish;
endmodule
```

→ All initial block will execute concurrently

Some shortcuts

① → output reg [7:0] data; instead of output [7:0] data;
reg [7:0] data;

② Variable can initialize when it is declared.

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

Always block → An "always" statement starts at time 0 and executes the statement inside the block repeatedly, and never stops

```
always
  begin
    ==========
    ==========
    ==========
  end
```

→ "Initial" and "always" block can exist within the same verilog module

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

```
module generating_clock;
  output reg clk;
  initial
    clk = 1'b0;
  always
    #5 clk = ~clk;
      ↳ Toggle after time 5 units
  initial
    #500 $finish;
endmodule
```

→ A module can contain any no. of "always" block, all of which execute concurrently.

→ The @(event_expression) part of required for both Combinational and sequential circuit description

```
always @ (event_expression)
  begin
    ==========
    ==========
    ==========
  end
```

↳ syntax

↳ Statement in this loop, when the event expression triggered and other time block is doing nothing

↳ It is necessary to remember the last value assign.

Reg type ← (a) = 2 + b;

→ LHS

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

RHS we any kind of may be appear (reg, wire etc)

Sequential Statement → Statements are execute one by one or sequentially.

↳ In ~~verilog~~ Verilog, one or more sequential statements can be present inside an "initial" or "always" block.

→ Multiple assigment statement inside "begin-end" block may execute sequentially or concurrently, depending upon on type of assigment.

blocking → a = b+c;

non-blocking → a <= b+c;

(a) begin-end → A no. of sequential statements can be grouped together using "begin-end".

if n=1, "begin-end" is not required.

```
begin
  sequential_statement_1;
  ;
  ;
  ——— ——— ——— n. ;
end
```

b) if... else

```
if(<expression>)
    sequential_statement;

if(<expression>)
    sequential_statement
else
    sequential_statement

if (<exp>)
    sequential_ste. ;
else if (< ex )
    ——— ——— ;
else if (   )
    ==
else
    ==
```

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

c) case

```
case (<expression>)
  exp1: sequential_statement;
  exp2: sequential_statement;
  .
  expn: sequential_statement;
  default: default_statement;
endcase
```

→ Two variation → Case z
→ Case x

treat all 'z' as don't care.

treat all 'z' and 'x' values in the case item as don't care.

```
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
```

(d) "while" loop

```
while ( <exp>)
    sequential_statement ;
```

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

e) for (exp1 ; exp2 ; exp3) sequential_statement ;
(exp1: initial condition; exp2: termination condition; exp3: change value)

```
initial count
reg [100 : 1] data1
integer i;
initial
  for (mycount = 0 ; mycount <= 255 ; mycount = mycount + 1)
    $display ("My count" : %d", mycount) ;
initial
  for (i = 1; i <= 100 ; i = i + 1)
    data[i] = 1'b0;
```

f) "repeat" loop — here we tell how many times the loop will execute

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

→ Expression in the loop can be constants, variable or single value.

```
integer n = 10;
repeat (n)
begin
  ≡
end
always n = 15;
```

No. of execution will be evaluated only when the loop starts and not during execution of the loop

It will execute 10 time only

g) "forever" loop

```
forever
    sequential_statement ;
```

No execute forever until $finish is encountered

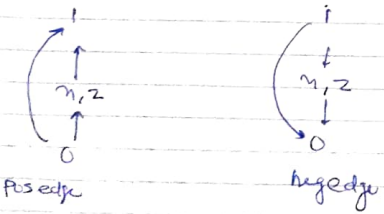
→ The "forever" loop is typically used along with timing specifier.
- If timing is not specified, the simulator would execute this statement indefinitely without advancing $ time.
- Rest of design will never be executed

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

Other constructs
(time_value)

@(event_exp) ⇝ The event expression specifies the event that is required to resume execution of the procedural block

- A "posedge" is any transition from {0, x, z} to 1 or from 0 to {x, z}.
- A "negedge" is any transition from {1, x, z} to 0, or from 1 to {x, z}.



→ always @(posedge a)

→ eg ⇝ @(in) // "in" changes
@(a or b or c) // any of "a", "b", "c" changes
@(a, b, c) // — same —
@(posedge clk) // positive edge of "clk"
@(posedge clk or negedge reset)
// positive edge of "clk" or negative edge of "reset".
@(*) // any variable changes

D-flipflop with synchronous 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

For asynchronous set and reset
↓
always @(Posedge clk or negedge set or negedge reset)

// Transparent latch with enable
module latch (q, qbar, din, enable);
input din, enable;
output q, qbar; reg q;
assign qbar = ~q;
always @(din or enable)
begin // (enable) q = din;

→ Ex: using procedural assignment

① A combinational logic

```
module mux21 (m1, in0, s, f);
  input in1, in0, s;
  output reg f;
  always @(in1 or in0 or s)
    if (s)
      f = in1;
    else
      f = in0;
endmodule
```

Block triggered whenever at least of "in1" or "in0" or "s" changes

→ always @ (in1, in0, s)
→ always @ (*)

② A sequential logic

```
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
```

→ another sequential logic

```
module incomp_state_spec (current_state, flag);
  input [0:1] curr_state;
  output reg [0:1] flag;
  always @(curr_state)
    case (curr_state)
      0,1 : flag = 2;
      3 : flag = 0;
    endcase
endmodule
```

→ so this program is a sequential circuit but ??

curr-state	flag
0	2
1	2
2	
3	0

→ The variable "flag" is not assigned a value in all the branches of the case statement
→ then a latch will generate for "flag".

flag will remain same as previous
→ which will done only if there is a latch is present

→ Ex. using procedural assignment

① A combinational logic

```
module mux21 (in1, in0, s, f);
  input in1, in0, s;
  output reg f;
  always @(in1 or in0 or s)
    if (s)
      f = in1;
    else
      f = in0;
endmodule
```

Block triggered whenever at least of "in1" or "in0" or "s" changes

→ always @ (in1, in0, s)
→ always @ (*)

② A sequential logic

```
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.
```

→ another sequential logic

```
module incomp_state_spec (current_state, flag);
  input [0:1] curr_state;
  output reg [0:1] flag;
  always @(curr_state)
    case (curr_state)
      0,1 : flag = 2;
      3 : flag = 0;
    endcase
endmodule
```

→ so this program is a sequential circuit but ??

curr-state	flag
0	2
1	2
2	
3	0

→ The variable "flag" is not assigned a value in all the branches of the case statement
→ then a latch will generate for "flag".

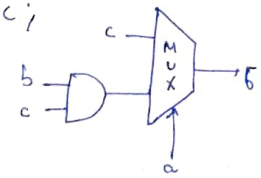
flag will remain same as previous
→ which will done only if there is a latch is present

Note:- when a "case" statement is incompletely decoded, the Synthesis tool will infer the needed for a latch to hold the residual output when the select bits take the unspecified values.

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

↳ again, For a=0, value of f is unspecified, and synthesis tool will generate latch.


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

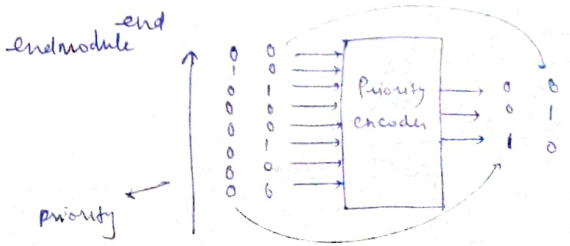


// A simple 4-func. ALU

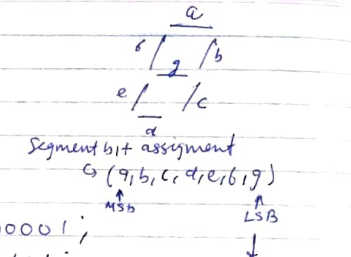
```
module ALU_4bit ( f, a, b, op);
  input [1:0] op; input [7:0] a, b;
  output reg [7:0] f;
  parameter ADD = 1'b00, SUB = 1'b01, MUL = 1'b10, DIV = 1'b11;
  always @(*)
    case (op)
      ADD : f = a + b;
      SUB : f = a - b;
      MUL : f = a * b;
      DIV : f = a / b;
    endcase
endmodule
```

// 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 if ( in[2] ) code = 3'b010;
      else if ( in[3] ) code = 3'b011;
      .
      .
      .
      else if ( in[7] ) code = 3'b111;
      else else code = 3'bxxx;
    end
endmodule
```



```
module bcd_to_7seg (bcd, seg);
  input [3:0] bcd;
  output [6:0] seg
    reg.
  always @ (bcd)
    case (bcd)
      0: seg = 7'b0000001;
      1: seg = 7'b1001111;
      2: seg = 7'b0010010;
      3: seg =
      4: seg =
      5: seg =
      6: seg =
      7: seg =
      8: seg =
      9. seg = 7'b0000100;
    default : 7'bXXXXXXX ;
  endcase.
endmodule
```



n-bit Comparator

```
module Compare (A, B, lt, gt, eq) ;
  parameter word_size = 16;
  input [word_size - 1 : 0] A, B;
  output reg lt, gt, eq ;

  always @(*)
    begin
      gt = 0; lt = 0; eq = 0;
      if (A > B) gt = 1;
      else if (A < B) lt = 1;
      else eq = 1;
    end
endmodule
```

→ 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.
```

```
module alu_example (alu_out, A, B, operation, en);
  input [2:0] operation ; input [7:0] A, B;
  input en;
  output [7:0] alu_out ; reg [7:0] alu_reg,

  assign alu_out = (en == 1) ? alu_reg : 4'bz;
  always @(*)
    case (operation)
      3'b000 : alu_reg = A + B;
      3'b001 : alu_reg = A - B;
      3'b010 : alu - reg = ~A;
      default : alu - reg = 4'b0;
    endcase
endmodule
```

Blocking and non-blocking assignments

For procedural assignment statements can be used to update 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 assignment, that results in the expression on the ~~net type~~ RHS to continously driven the "net" type variable on the left.

Procedural assignment ~~state~~ statement → Blocking ("=")
→ Non-blocking ("<=")

→ Procedural assignment statements can only appear within procedural block ("initial" or "always").

→ also can use, `{a[5], b[5:2], c} = a*b+q;`

Blocking Assignment

- General syntax:-
  variable_name = [delay or event control] expression;
- '=' is used for blocking assignment
- Blocking assignment statements are executed in the order they are specified in a procedural block →
- They do not block execution of statements in other procedural blocks

→ recommended style for modeling combinational logic

Eg.
```
integer a,b,c;
initial
  begin
    a = 10; b = 20; c = 15;      → initially a=10, b=20, c=15
    a = b + c;                     a = 35
    b = a + 5;                     b = 40
    c = a - b                      c = -5
```

```
module blocking_example;
  reg x,y,z;
  reg [31:0] A,B; integer sum;
  initial
    begin
      x = 1'b0; y = 1'b0; z = 1'b1;     // At time = 0
      sum = 1;                          // At time = 0
      A = 31'b0; B = 31'habababab;       // At time = 0
      #5 A[5] = 1'b1;                   // At time = 5
      #10 B[31:29] = {x,y,z};           // At time = 15
      sum = sum + 5;                    // At time = 15
    end
endmodule
```

~~Procedure~~ Non blocking Assignment

~~Eve~~ General syntax:-
variable_name <= [delay_or_event_control] expression;

- "<=" operator is used to specify non-blocking assignment.
- Non blocking assignment statement allow scheduling of assignments without blocking execution of statement that follows within the procedural block.

→

allows concurrent procedural assignment, suitable for sequential logic

→ Several "reg" type variables can be assigned synchronously under the control of common clock.

eg 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

- Initially a=10; b=20, c=15
- a become 35 at time=5
- b " 15 " " " = "
- c " -10 " " " " "

→ all assignment will happen concurrently

→ always @(posedge clk)
begin
a<= b+c; → Recommended style for modeling Synchronous circuit, where assignments take place in synchronism with clock.
b<= a+d;
c<= a+b;
end

↳ all assignment take place synchronously at the rising edge of the clock

Swaping 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 vise versa, depending on the simulator implementation,
• Both registers will get the same value (either "a" or "b").
↳ Race condition

① always @(posedge clk)
a<= b;
always @(posedge clk)
b<= a;

→ this will work fine
→ Here the variables are correctly swap.
→ All RHS variables are read first, and assigned to LHS variables at the positive edge

Some Rules to be followed

① It is recommended that blocking and non-blocking assignment are not mixed in same "always" block.

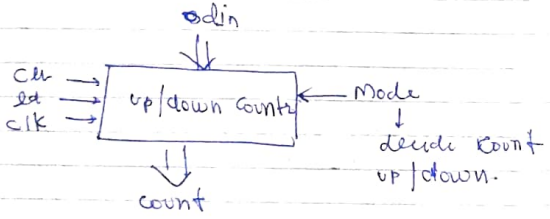
→ 8 to 1 multiplexer: behavioral description

```
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, ld, d_in, 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
```

non blocking assignment used.



// parameterized design: an N-bit counter

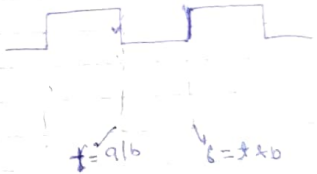
```
module counter (clear, clock, count);
    parameter N=7;
    input clear, clock;
    output reg [0:N] count;

    always @(negedge clock)
        if (clear)
            count <= 0;
        else
            count <= count + 1;
endmodule
```

// using more than one clock in a module

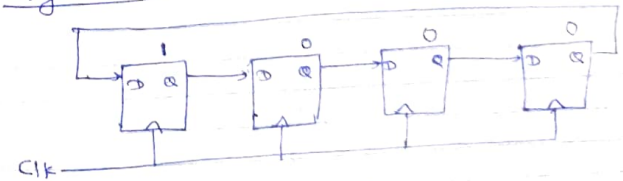
```
module multiple_clk (clk1, clk2, a, b, c, f1, f2);
    input clk1, clk2, a, b, c;
    output reg f1, f2;

    always @(posedge clk1)
        f1 <= a & b;
    always @(negedge clk2)
        f2 <= b ^ c;
endmodule
```

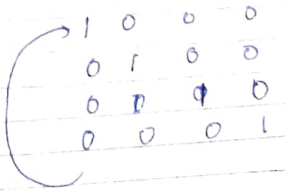
// using multiple edges of the same clock

```
always @(posedge clk)
    f <= f & b;
always @(negedge clk)
    f <= a | b;
```

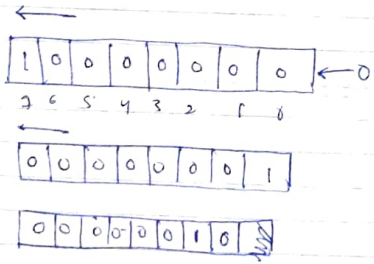


// A ring counter





```
module ring_counter (clk, init, count)
  input clk, int;
  output reg [7:0] count;

  always @ (posedge clk)
    if (init) count = 8'b10000000;
    else
      begin
        count <= count<<1;
        count[0] = count[7];
      end
endmodule
```



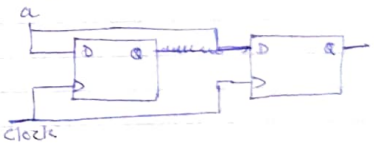

```
// Ring counter → Version 2

always @ (posedge clk)
  if (init) count <= 8'b10000000;
  else
    count = { count[6:0], count[7] }
```

Different cases for Blocking and non-blocking assignment

eg①
```
begin
  a = #5 b;
  c = #5 a;
end
```
↓ here c will be equal to b, after 10 unit time

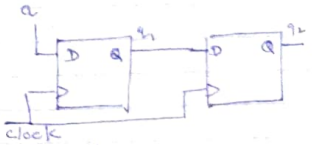
```
begin
  a <= #5 b;
  c <= #5 a;
end
```
↓ here 'a' will be 'b' and 'c' will be 'a' after 5 unit time

eg②
```
always @ (posedge clock)
  begin
    q1 = a;
    q2 = q1;
  end
```



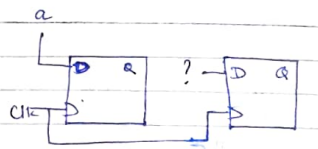
③
```
always @ (Posedge clock)
  begin
    q2 = q1;
    q1 = a;
  end
```

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

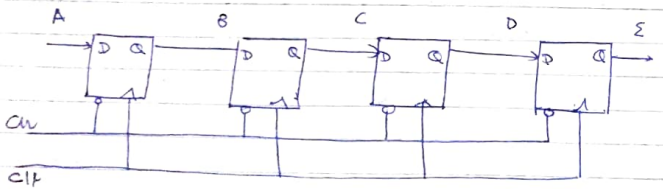


④
```
always @ (posedge clock)
  q2 <= q1;
always @ (posedge clock)
  q1 <= a;
```

⑤ always@(posedge clock)
q1 = q;
always @ (posedge clock)
q2 = q1;



⑥ ~~module~~ Shift register.




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

→ this will work as shifting of data.

but if we do,

```
A B = A
C = B
D = C
Σ = D;
```

then it ~~eventually~~ works as Σ = A

But we can do,

```
Σ <= D;
D <= C;
C <= B;
B <= A;
```

OR

```
A B <= A;
C <= B;
D <= C;
Σ <= D;
```

→ Here both are working same

~~Start~~

Generate Block

- 6 "generate" statements allow Verilog code to generated dynamically before the simulation or synthesis begins
  - It is very convinient to ~~create~~ create parameterize module descriptions
  - eg :- N bit ripple carry adder for arbitrary value of N.
- Requires the keywords "generate" and "endgenerate".

- Special "genvar" variables;
- The keyword "genvar" can be used to declare variables that are ~~only~~ used only in the evaluation of generate block.
- → These variables do not exist during simulation or synthesis

```
module nor_bitwise (G, a, b);
  parameter N=16;
  input [N-1:0] a, b;
  output [N-1, 0] G;
  genvar P;

  generate for (P=0; P<=N; P=P+1)
    begin :nor
      nor ~~XG~~ (G[P], a[P], b[P]);
    end
  endgenerate
endmodule
```

→ Copied:
```
xor g0 ( b[0], a[0], b[0] );
// g1 ( b[1], a[1], b[1] );
.
.
.
```

→ In the bitwise nor eg, the name "xorlp" was given to the generate loop.

→ The relative hierarchical names of the xor gates will be
xorlp[0].xg, xorlp[1].xg, ... xorlp[15].xg
↑
xor g0

// N bit Ripple carry adder

```
module RCA ( carryout, 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;
  assign carry[0] = carry_in;
  assign carry_out = carry[N];
  genvar i;
  generate for (i=0; i<N; i++)
    begin fa_loop
      wire t1,t2,t3;
      xor g1 (t1, a[i], b[i]), g2 ( sum[i], t1, carry[i]);
      and g3 (t2, a[i], b[i]), g4 (t3, t1, carry[i]);
      or g5 ( carry[i+1], t2, t3);
    end
  endgenerate
endmodule
```

→ Some of the relative hierarchical instance names that are generated are:
— fa_loop[1].g1, fa_loop[1].g1, fa_loop[7].g1, etc
fa_loop[0].g1

→ Some of the nets ("wires") that are generated are:
— fa_loop[0].t1, fa_loop[1].t2, ... fa_loop[7].t3 etc.

User defined Primitives (UDP)

→ They are used to define custom Verilog primitives by the use of lookup tables.

→ They can specify
① Truth table for combinational functions
② State table for sequential function
③ don't care, rising and falling edge etc can be specified

→ For combinational function, truth table can be specified as
<input1> <input2> <input3> ... <inputN> : <output>

→ For Sequential function
<input1> <input2> ... <inputN> : <present_state> : <next_state>

Some Rules

① Input entries in the table must be in the same order as the "input" terminal list.

② For combinational UDPs, the output terminal is declared as "output".

③ For sequential UDPs, the output terminals is declared as "reg".

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

→ For unspecified cases, the output is set to "x"

→ Modeling of Combinational circuit

```
// 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 also specify don't care input combination as "?"

eg
```
// a  b  c   sum
   0  0  ?   0
   0  ?  0   0
   ?  0  0   0
   1  1  ?   1
   1  ?  1   1
   ?  1  1   1
```

→
```
// 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
```

→ UDPs can be instantiated just like any other Verilog module

→ UDP benifits in verilog where don't care are present

eg // 4-input AND func.
```
primitive udp_and4 (f, a, b, c, d);
    input a, b, c, d;
    output f;
    table
    // a  b  c  d  f
       0  ?  ?  ?  : 0 ;
       ?  0  ?  ?  : 0 ;
       ?  ?  0  ?  : 0 ;
       ?  ?  ?  0  : 0 ;
       1  1  1  1  : 1 ;
```

4-input OR gate
```
   a  b  c  d  f
   1  ?  ?  ?  : 1 ;
   ?  1  ?  ?  : 1 ;
   ?  ?  1  ?  : 1 ;
   ?  ?  ?  1  : 1 ;
   0  0  0  0  : 0 ;
```

→ It is also used for MUX.

→ Modeling of Sequential Circuits

```
// A level sensitive D type latch.
primitive Dlatch (q, d, clk, clr);
    input d, clk, clr;
    output reg q;

    initial
      q = 0;    // This is optional
    table
    // d  clk  clr  q  :  q_new
       ?  ?    1    ?  :  0 ;
       0  1    0    ?  :  0 ;
       1  1    0    ?  :  1 ;
       ?  0    0    ?  :  -  ;
```

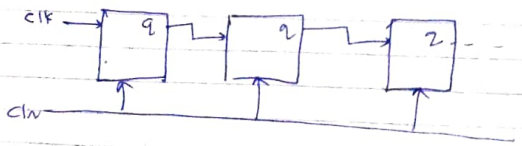
- → Remains previous state

// A T flipflop

```
primitive TFF (q, clk, clr);
    input clk, clr;
    output reg q;

    table
    // clk    clr   :  q   :  q_new
        ?      1    :  ?   :   0   ;
        ?     (10)  :  ?   :   -   ;   // ignore -ve edge of "clr"
       (10)    0    :  1   :   0   ;
       (10)    0    :  0   :   1   ;
       (01)    0    :  ?   :   -   ;   // ignore +ve edge of "clk"
```

// Constructing a 6-bit ripple counter using primitive TFF.

```
module ripple_counter (count, clk, clr);
    input clk, clr;
    output [5:0] count;
    TFF F0 (count[0], clk, clr);
    TFF F1 (count[1], count[0], clr);
    TFF F2 (
    TFF F3 (
    TFF F4 (
    TFF F5 (
endmodule
```



Modeling of Finite State Machine

→ In a combinational circuit, the output depend only on the applied input value not on the past history.

→ In a sequential circuit, the output depend only on the applied value but also on the internal state
- The internal states also change with time
- The no. of states is finite, and hence a sequential circuit is also refferred to as a FSM (Finite state machine)

Can be represented either in the form of a state table or in the form of state transition diagram.

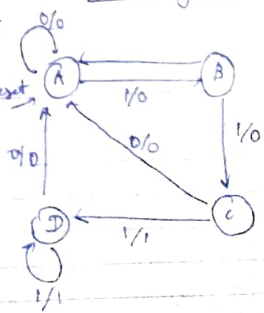
→ Algorithm State Machine (ASM) Chart

eg → • A circuit to detect 3 or more 1's in a serial bit stream.
- The bits are applied serially in synchronism with a clock.
- The output will become 1 whenever it detects 3 or more consecutive 1's in the stream.

State table

Reset	PS	input	NS	output
1	–	–	A	0
0	A	0	A	0
0	A	1	B	0
	B	0	A	0
	B	1	C	0
	C	0	A	0
	C	1	D	1
	D	0	A	0
	D	1	D	1

State diagram



Mealy and Moore FSM Types

A deterministic FSM can be mathematically defined as a 6-Tuple

1. Σ → Set of input combinations
2. Γ → Set of output combination
3. S → Finite set of states
4. S_0 → $S_0 \in S$ is initial state
5. δ → state-transition func.
6. ω → output function.

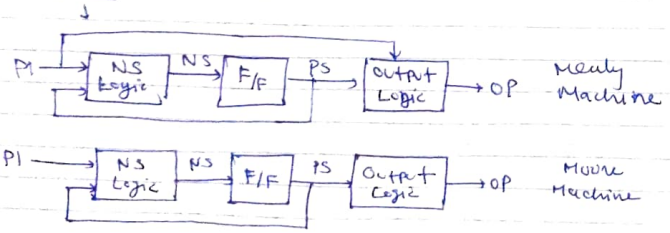
$\delta : S \times \Sigma \rightarrow S$

↳ Present state and present input determines the next state (NS)

For Mealy machine → $\omega : S \times \Sigma \rightarrow \Gamma$ (output depend on input + state)

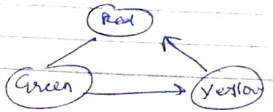
moore machine → $\omega : S \rightarrow \Gamma$ (output depends only the state)

Pictorial Depiction



Example 1

There are three lamps, Red, Green, and yellow, that should glow cyclically with fixed time interval (say, 1 sec).



∴ moore machine because it is not depend on input

```
module Cyclic_lamp(clock, light);
  input clock;
  output reg [0:2] light;
  parameter S0=0, S1=1, S2=2;
  parameter RED = 3'b100, GREEN = 3'b010, YELLOW = 3'b001;
  reg [0:1] state;
  always @(posedge clock)
    case (state)
      S0: begin
            light <= GREEN; state <= S1;
          end
      S1: begin
            light <= YELLOW; state <= S2;
          end
      S2: begin
            light <= RED; state = S0; end
      default: begin
                 light <= RED;
                 state = S0;
               end
    endcase
endmodule
```

↳ Here we are generating 3 FF for output and 2 FF for latch.

But it is not necessary to do like this, we can use 2 "always" block to get the same output.

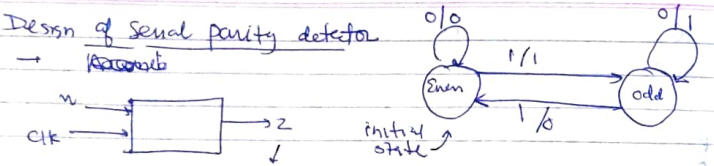
sequential logic

```
always @(posedge clk)
  case (state)
    S0: state <= S1;
    S1: state <= S2;
    S2: state <= S3;
  endcase
  default: state <= S0;
```

Combinational logic

```
always @(state)
  case (state)
    S0: light = RED;
    S1: light = GREEN;
    S2: light = YELLOW;
    default: light = RED;
  endcase
endmodule
```

Design of serial parity detector



0 → even no. of one

1 → odd no. of one

```
module parity_gen (n, clk, z);
  input n, clk;
  output reg z;
  reg even_odd;                    // The machine state
  parameter EVEN=0, ODD=1;
  always@(posedge clk)
    case (even_odd)
      EVEN: begin
              z<=n?1:0;
              even_odd <=n, ODD: EVEN;
      ODD: begin
              z<=n?0:1
              even_odd <= n, EVEN : ODD;
      default : even_odd <= EVEN;
    endcase
endmodule
```

```
always @(posedge clk)
  case (even_odd)
    EVEN: even_odd <= n?ODD : EVEN ;
    ODD : even_odd <= n? EVEN: ODD ;
    default : even_odd <= EVEN;
  endcase
```

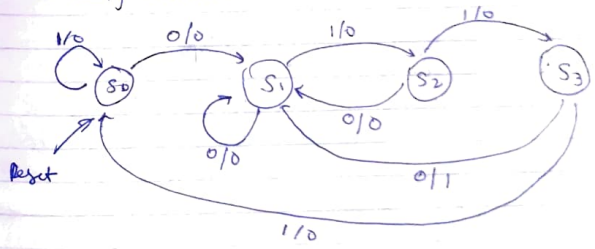
```
always @(even_odd)
  case(even_odd)
    EVEN: z=0;
    ODD : z=1;
  endcase
```

→ here, this design will generate a latch for the output "z".

Ex-3 ⇝ Sequence detector → 0110 → z=1
otherwise → z=0

eg n: 0110110
z = 0001001

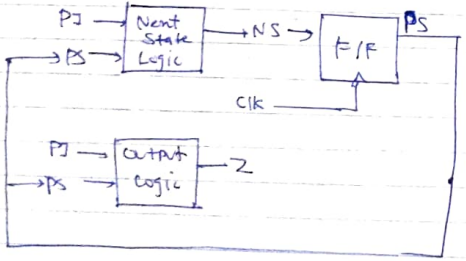
→ Mealy machine



→ Sequence detector

```
module seq_detector (n, clk, reset, z);
  input n, clk, reset;
  output reg z;
  parameter S0=0, S1=1, S2=2, S3=3;
  reg [0:1] PS,NS;
  always @(posedge clk or posedge reset)
    if(reset) PS <= S0;
    else      PS <= NS;
  always @(PS, n)
    case (PS)
      S0: begin
            z= n?0:0;
            NS = n? S0: S1;
      end
```

```
S1 : begin
       z = x ? 0 : 0 ;
       NS = x ? S2 : S1 ;
     end
S2 : begin
       z = x ? 0 : 0 ;
       NS = x ? S3 : S1 ;
     end
S3 : begin
       z = x ? 0 : 1 ;
       NS = x ? S0 : S1 ;
     end
endcase
endmodule
```



Datapath and controller path

In complex digital system, the hardware is typically partitioned into two parts

a) Datapath :- which consists of the functional unit where all Computations are carried out
   → Typically consists of registers, multiplexers, adder, multiplier, counter and functional block

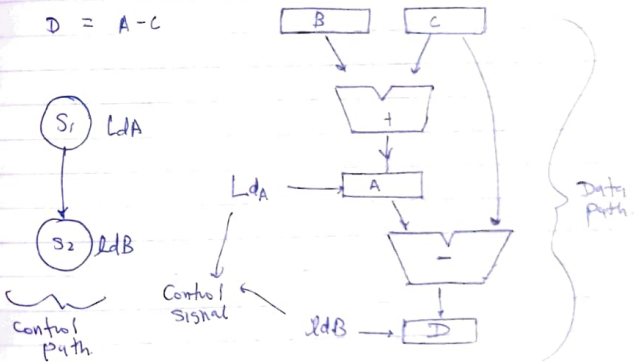
b) Control Path :- which implements a finite-state machine and provides control signals to the data path in the proper sequence

Also takes inputs from the datapath regarding various status information.

eg reg [15:0] A, B, C, D;

A = B + C

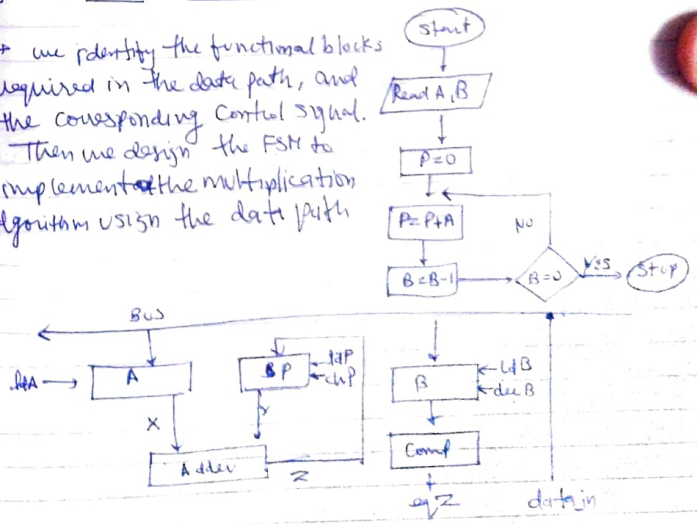
D = A - C

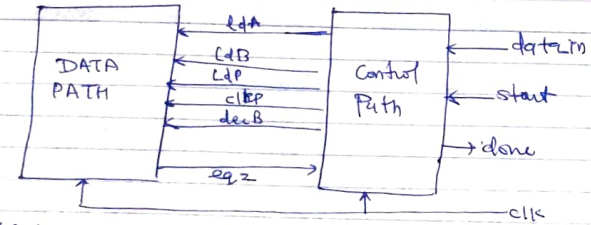


Eg :→ Multiplication by repeated addition

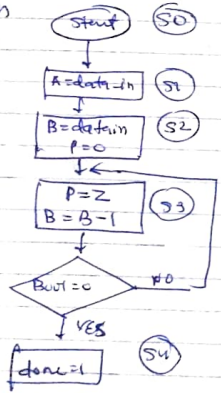
→ we identify the functional blocks required in the data path, and the corresponding control signal.

→ Then we design the FSM to implement the multiplication algorithm using the data path





Control Path




```
module FUC_datapath(eqz, LdA, LdB, clrP, LdP, decB, data_in, clk);
  input LdA, LdB, LdP, clrP, decB, clk;
  input [15:0] data_in;
  output eqz;
  wire [15:0] x, y, z, Bout, Bus;

  PIPO1 A (X, Bus, LdA, clk);
  PIPO2 P (Y, z, LdP, clrP, clk);
  CNTR B (Bout, Bus, LdB, decB, clk);
  ADD AD (z, X, Y);
  EQZ COMP (eqz, Bout);
endmodule
```

```
module PIPO1 (dout, din, ld, clk);
  input [15:0] din;
  input ld, clk;
  output reg [15:0] dout;
  always @(posedge clk)
    if (ld) dout <= din;
endmodule

module PIPO2 (dout, din, ld, clr, clk);
  input [15:0] din;
  input ld, clr, clk;
  output reg [15:0] dout;
  always @(posedge clk)
    if (clr) dout <= 16'b0;
    else if (ld) dout <= din;
endmodule

module EQZ (eqz, data);
  input [15:0] data;
  output eqz;
  assign eqz = (data == 0);
endmodule

module ADD (out, in1, in2);
  input [15:0] in1, in2;
  output reg [15:0] out;
  always @(*)
    out = in1 + in2
endmodule
```

```
module CNTR (dout, din, ld, dec, clk);
  input [15:0] din;
  input ld, dec, clk;
  output reg [15:0] dout;
  always @ (posedge clk)
    if (ld) dout <= din;
    else if (dec) dout <= dout - 1;
endmodule.
```

```
module datapath_controller (LdA, LdB, LdP, clrP, decB, done, clk, eqz, start);
  input clk, eqz, start;
  output reg LdA, LdB, LdP, clrP, decB, done;
  reg [2:0] state;
  parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011,
            S4 = 3'b0100;
  always @ (posedge clk)
    begin
      case (state)
        S0: if (start) state <= S1;
        S1: state <= S2;
        S2: state <= S3;
        S3: #2 if (eqz) state <= S4;
        S4: state <= S4;
        default : state <= S0;
      endcase
    end
  always @ (state)
    begin
      case (state)
        S0: begin #1 LdA = 0; LdB = 0; LdP = 0; clrP = 0;
            decB = 0; end
        S1: begin #1 LdA = 1; end
        S2: begin #1 LdA = 0; LdB = 1; clrP = 1; end
        S3: begin #1 LdB = 0; LdP = 1; clrP = 0; decB = 1; end
        S4: begin #1 done = 1; LdB = 0; LdP = 0; decB = 0; end
        default: begin #1 LdA = 0; LdB = 0; LdP = 0; clrP = 0;
            decB = 0; end
      endcase
    end
endmodule
```

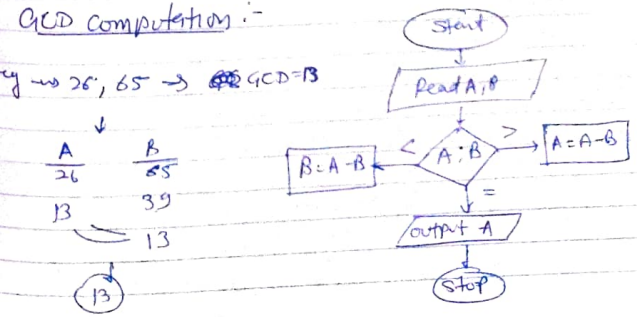
A better style of Modeling data / control path

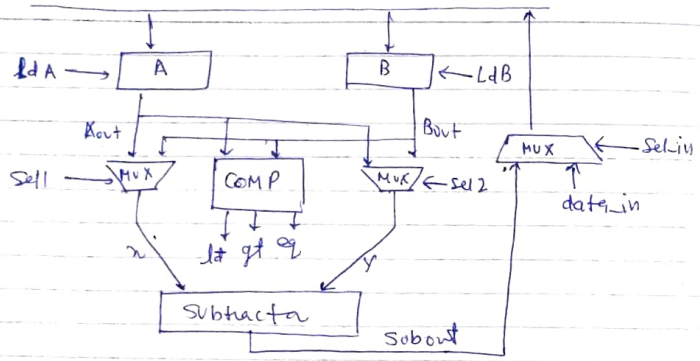
→ In the "always" block activated by clock edge, both state change as well as computation of the next state is performed.

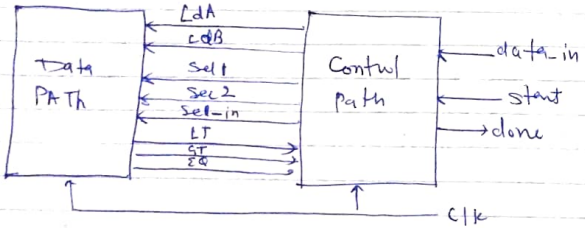
→ A better and recommended approach:-

- Only trigger the state change in the clock activated "always" block.
- In a separate "always" block using blocking assignments, compute the next state
- As in the previous example, in a separate "always" block, generate the control signals for the data path.

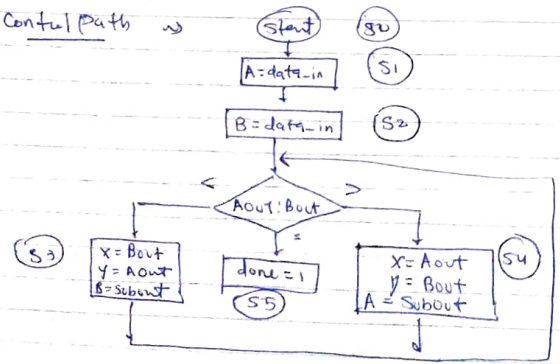
GCD computation :-

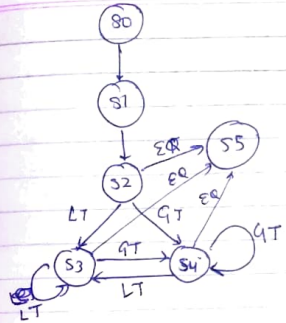






Contwl path →






```
module GCD_datapath (gt, lt, eq, ldA, ldB, Sell, sel2, Sel_in,
                     data_in, clk);
    input ldA, ldB, sell, sel2, selc_in, clk;
    input [15:0] data_in;
    output gt, lt, eq;
    wire [15:0] Aout, Bout, ax, Y, Bus, SubOut;

    PIPOA (Aout, Bus, ldA, clk);
    PIPOB (Bout, Bus, ldB, clk);
    MUX MUX_in1 (X, Aout, Bout, sell);
    MUX MUX_in2 (X, Aout, Bout, sel2);
    MUX MUX_load( Bus, subout, data_in, sel_in);
    SUB SB(Subout, X, Y);
    COMPARE COMP(lt, gt, eq, Aout, Bout);
endmodule.

module PIPO (data_out, data_in, load, clk);
    input [15:0] data_in;
    input load, clk;
    output reg [15:0] data_out;
    always @(posedge clk)
        if (load) data_out <= data_in;
endmodule.
```

```
module SUB(out, in1, in2);
    input [15:0] in1, in2;
    output reg [15:0] out;
    always @(*)
        out = in1 - in2;
endmodule

module COMPARE (lt, gt, eq, data1, data2);
    input [15:0] data1, data2;
    output lt, gt, eq;
    assign lt = data1 < data2;
    assign gt = data1 > data2;
    assign eq = data1 == data2;
endmodule

module MUX (out, in0, in1, sel);
    input [15:0] in0, in1;
    input sel;
    output [15:0] out;
    assign out = sel ? in1 : in0;
endmodule

module Controller( ldA, ldB, sel1, sel2, sel_in, done, clk, lt, gt, eq,
    start);
    input clk, lt, gt, eq, start;
    output reg ldA, ldB, sel1, sel2, sel_in, done;
    reg [2:0] state;
    parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011,
        S4 = 3'b100, S5 = 3'b101;
    always@ (posedge clk)
        begin
            case( state)
                S0: if (start) state <= S1;
                S1: state <= S2;
                S2: #2 if (eq) state <= S5;
                    else if (lt) state <= S3;
                    else if (gt) state <= S4;
                S3: #2 if(eq) state <= S5;
                    else if (lt) state <= S3;
                    else if (gt) state <= S4;
                S4: #2 if(eq) state <= S5;
                    else if (lt) state <= S3;
                    else if (gt) state <= S4;
                S5: state <= S5;
                default: state <= S0;
            endcase
        end

always@(state)
    begin
    case (state)
        S0: begin sel_in = 1; ldA = 1; ldB = 0; done = 0; end
        S1: begin sel_in = 1; ldA = 0; ldB = 1; end
        S2: if(eq) done = 1;
            else if (lt) begin
                sel1 = 1; sel2 = 0; sel_in = 0;
                #1 ldA = 0; ldB = 1;
                end
            else if (gt) begin
                sel2 = 1; sel1 = 0; sel_in = 0;
                #1 ldA = 1; ldB = 0;
                end
```

```
S3: if(eq) done=1;
    else if(lt) begin
        sel1=1; sel2=0; sel_in=0;
        #1 ldA=0; ldB=1;
        end
    else if(gt) begin
        sel1=0; sel2=1; sel_in=0;
        #1 ldA=1; ldB=0
        end
S4: if(eq) done=1;
```

```
S5: begin
    done=1; sel1=0; sel2=0; ldA=0;
    ldB=0; end
    end
endcase
end
endmodule
```

Note:- Alternative Approach

```
in controller.. ns reg [2:0] state, nextstate;
always @(posedge clk)
  begin
    state <= next_state;
  end
always @(state)
  begin
  case(state)
    S0:
    S1:
    S2:
```

```
S0;
S1;
S2: if(eq) begin done=1; next_state = S5; end
    else if(lt) begin
        ____ - - - ; nextstate = S3;
        ____
    end.
endmodule
```

- Function in Verilog

```
module fulladder(S, cout, a, b, cin);
  input a, b, cin;
  output S, cout;
  assign S = sum(a,b,cin);
  assign cout = carry(a,b,cin);
endmodule
function sum;
  input x, y, z;
  begin
    sum = x^y^z;
  end
function carry;
  input x, y, z;
  begin
    carry = (x&y) | (y&z) | (z&x);
  end
```

- A function in verilog returns a single value
- Can be used to make code more readable
- Input arguments appear in same order
- Typically used with "assign"

Tasks in Verilog

```
module fulladder( s, cout, a, b, cin);
  input a, b, cin;
  output reg s, cout,

  always @ (a or b or cin)
    FA (s, cout, a, b, cin);

  task FA;
    output sum, cout;
    input A, B, C;
    begin
      #2 sum = A^B^C;
      carry = (A&B) | (B&C) | (C&A);
    end
  endtask
endmodule
```

- The arguments must be specified in the same order as they appear in the task declaration.
- More than one output value can be return.

Difference b/w Function and Task

Function	Task
1) can call function but not task	1) can call both function and task
2) execute in 0 simulation time	2) may execute in non-zero simulation time
3) can't contain any delay, event or timing control statement	3) can contain delay, event, timing control.
4) always return single value	4) A task can pass multiple values through "output" and "inout" type argument
5) must have atleast one input agument	5) can have zero or more arguments

Constructs to avoid for combinational Synthesis

1. Edge dependent event control
2. Combinational feedback loop
3. Procedural or continuous assignment containing event or delay control
4. Procedural loop with timing
5. Data dependent loop
6. Sequential user define primitives
7. Other miscellaneous constructs like "fork-join", "wait", "disable", etc.

Modeling Memory

4K byte → 2^{12} byte

- How to model Memory
  1. Memory is typically included by instantiating a pre-designed module from the design library.
  2. Alternatives → we can model memory using two-dimentional arrays.
     - Array of register variable
       - → Mainly used for simulation purpose
       - → Even used for synthesis of small-size memory

```
module memory_module ( )
  reg [7:0] mem [0:1023];
endmodule
```

↳ memory words of 8 bits
↳ memory words can be accessed by mem[0], mem[1] ...

```
initial begin
mem[0] = 8'b00010001;
mem[2] = 8'b01010110;
end
```

~~Practice Programs~~

By reading memory data from specified disk file
- used for simulation
- used for test benches

① $readmemb (filename, memname, start-addr, stop-addr)
- (Data is read in binary format)

② $readmemh (filename, memname, start-addr, stopaddr)
- (Data is read in hexadecimal format)

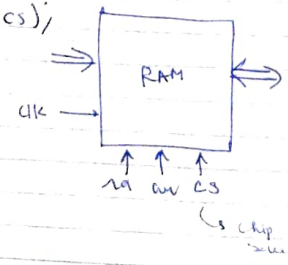
→ If "startaddr" and "stopaddr" are not describe, then the entire memory is read.

```
eg module memory(...);
     reg [7:0] mem [0:1023];
     initial
       begin
         $readmemh("mem.dat", mem);
       end
   endmodule
```

```
$Readmemh("mem.data", mem, 200, 50);
```

Eg :- Single-port RAM with synchronous read/write

```
module ram_1(addr, data, clk, rd, wr, cs);
  input [5:0] addr;
  inout [7:0] data;        → wire type
  input clk, rd, wr, cs;
  reg [7:0] mem [0:1023];
  reg [7:0] d_out;
```



→ Bidirectional

```
  assign data = (cs && rd) ? d_out : 8'bz;
  always @(posedge clk)
      if (cs && wr && !rd) mem[addr] = data;
  always @(posedge clk)
      if (cs && rd && !wr) dout = mem[addr];
endmodule
```

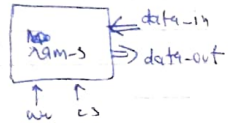
→ A ROM / EPROM

```
module rom(addr, data, rd_en, cs);
  input [2:0]addr; input rd_en, cs;
  output [7:0] data;
  always @(addr or rd_en or cs)
    case(addr)
      0: data = 22;
      1: data = 45;
      .
      .
      7: data = 12;
    endcase
endmodule
```

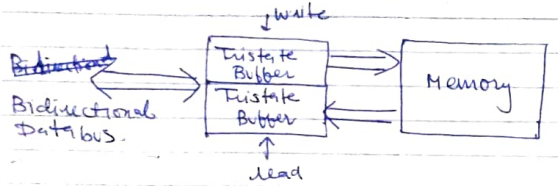
Note :- Some simulation and synthesis tools gives inconsistent behaviour when using "inout", and we should avoid.

→ Eg :-

```
module ram_3(data_out, data_in, addr, wr, cs);
  parameter addr_size = 10, word_size = 8, memory_
            size = 1024;
  input [addr_size-1:0]addr;
  input [word_size-1:0] data_in,
  input wr, cs;
  output [word_size-1:0] data_out;
  reg [word_size-1:0] mem [memory_size-1:0];
  assign data_out = mem[addr];
  always @(wr or cs)
      if(wr) mem[addr] = data_in;
endmodule
```



→ bidirectional data bus ; using tristate buffer.

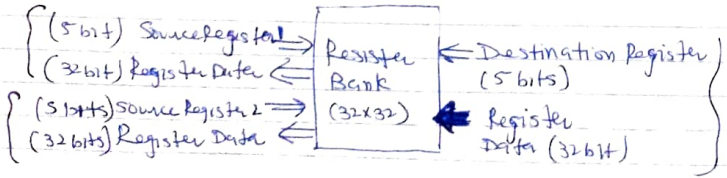


↳ inout [7:0] Bus;
wire [7:0] data_out, datain;
assign Bus = read ? data_out : 8'hzz;
assign Bus = write ? Bus : 8'hzz;
datain

Modeling Register Bank

↳ A register bank or register file is a grp of registers, any of which can be randomly access

→ Register banks often allow concurrent accesses
→ MIPS 32 allows 2 register read and 1 register write every clock cycle




```
// 4x32 register file
module regbank_v1( rdData1, rdData2, wrData, sr1, sr2, dr,
                   write, clk);
  input clk, write;
  input [1:0] sr1, sr2, dr; // source and destination Register
  input [31:0] wrData;
  output reg [31:0] rdData1, rdData2;
  reg [31:0] R0, R1, R2, R3;
  always @(*)
    begin
      case (sr1)
        0 : rdData1 = R0;
        1 :   "   = R1;
        2 :   "   = R2;
        3 :   "   = R3;
        default : 32'hxxxxxxxx;
      endcase
    end
  always @(*)
    begin
      case (sr2)
        0 :
        1 :
        2 :
        3 :
        default :
  always @(posedge clk)
    begin if(write)
      case (dr)
        0 : R0 <= wrData;
        1 : R1 <= wrData;
        2 :
        3 :
      endcase
    end
endmodule
```

```
Module regbank_v2 (
    input clk, write);
    input [1:0] sr1, sr2, dr;
    output [31:0] R0, R1, R2, R3;
    assign rdData1 = (sr1==0) ? R0:
                     (sr1==1) ? R1:
                     (sr1==2) ? R2:
                     (sr1==3) ? R3: 0;
    assign rdData2:
    always @(posedge clk)
      begin
        if(write)
          case (dr)
            0: R0 <= wrData;
            1: R1 <= wrData;
          endcase
      end
endmodule

// 32x32 register file
module regbank_v3(
    input clk, write;
    input [4:0] sr1, sr2, dr;
    input [31:0] wrData;
    output [31:0] rdData1, rdData2;
    reg [31:0] regfile [0:31];
    assign rdData1 = regfile[sr1];
    assign rdData2 = regfile[sr2];
    always @(posedge clk)
        if(write) regfile[dr] <= wrData;
endmodule
```

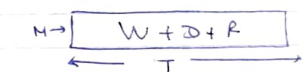
Pipelining concepts

A mechanism for overlapped execution of several inputs sets partitioning some computation into a set of k sub-computations (or stages)

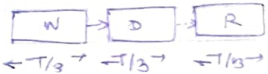
→ Very nominal increase in the cost of implementation

→ Very significant speedup

eg



For n cloths, time $T_1 = n.T$



For n cloths, time

$T_2 = (2+n)\,T/3$

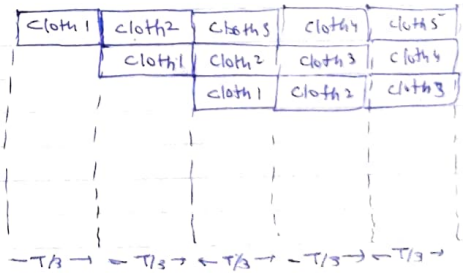
machine M that can wash(W), dry(D), and iron(R) cloths

$T_1 = 1000T$

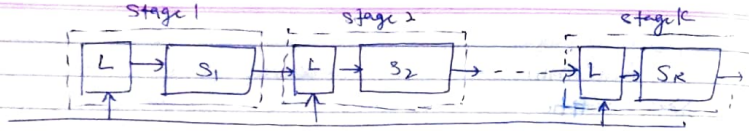
$n = 1000$

$T_2 = \frac{1000+2}{3} T$

$\approx \frac{1000}{3} T$



→ Here we need latch b/w successive stages to hold the intermediate results temporarily.

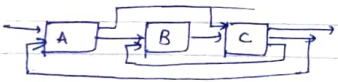


→ The latches made with master-slave flip-flops, and serve the purpose of isolating inputs from output
→ Pipeline stages are combinational circuits

Structure of pipeline

• Linear Pipeline :-

• Non linear pipeline :-



Speeding up and Efficiency

τ :: clock period of the pipeline
t_i :: time delay of the circuitry in stage S_i
d_L :: delay of a latch

maximum stage delay $\Rightarrow \tau_m = \max(t_i)$
Thus, $\tau = \tau_m + d_L$
Pipeline frequency $\Rightarrow f = 1/\tau$

Total time to process N data set is given by

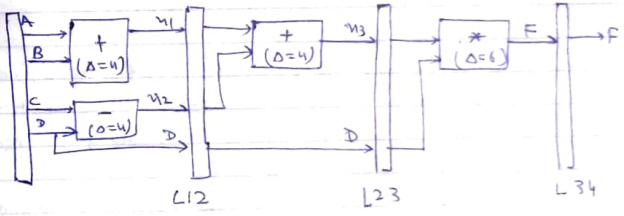
$$T_k = [(k-1)+N]\cdot\tau$$

→ $T_1 = Nk\tau$ → For an equivalent non-pipeline processor

Speedup of the k-stage

$$S_k = \frac{T_1}{T_k} = \frac{Nk\tau}{k\tau + (N-1)\tau} = \frac{Nk}{k+(N-1)}$$

Eg ① a) S1: $n_1 = A+B$; $n_2 = C-D$;
S2 & $n_3 = n_1 + n_2$
S3 &: $F = n_3 * D$;



→ module pipe_en(F, A, B, C, D, clk);
parameter N=10;
input [N-1:0] A, B, C, D;
input clk;
reg [N-1:0] L12_n1, L12_n2, L12_D, L23_n3, L23_D, L34_F;
assign F = L34_F;
always @(posedge clk)
begin
L12_n1 <= #4 A+B;
L12_n2 <= #4 C-D;
L12_D <= D;
end
always @(posedge clk)
begin
L23_n3 <= #4 L12_n1 + L12_n2;
L23_D <= L12_D;
end
always @
begin
L34_F <= L23_n3 * L23_D;
end
endmodule

→ Here we are using single clock which can generate race around condition. So, generally 2 two alternative clocks are preferred.

A more complex eg

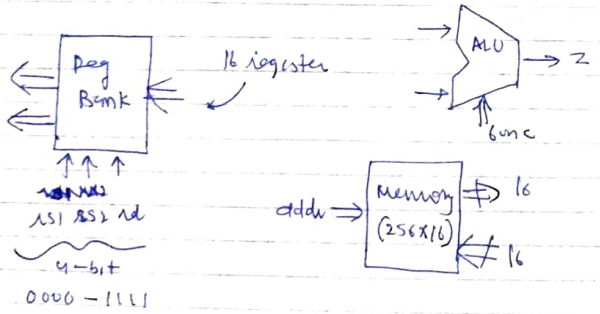
→ Inputs : Three register addresses (rs1, rs2, and rd), an ALU function (func), and memory address (addr)

→ stage1: Read two 16-bit no. from the register specified by "rs1" and "rs2" and store them in A and B

→ stage2: Performed an ALU operation on A and B specified by func and store it in "Z".

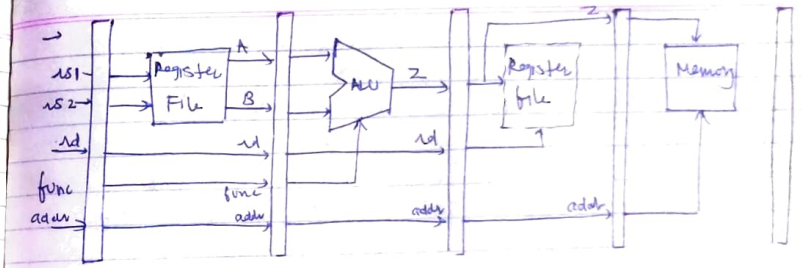
→ stage3: Write the value of Z in the register specified by "rd".

→ stage4: Also write the value of Z in the memory location "addr"



→ The ALU functions

0000 : ADD → Z = A+B
0001 : SUB Z = A-B
0010 : MUL Z = A×B
0011 : SELA Z = A
0100 : SELB Z = B
0101 : AND Z = A&B
0110 : OR Z = A|B
0111 : XOR Z = A^B
1000 : NEGA Z = ~A
1001 : INVB Z = ~B
1010 : SRA Z = A>>1
1011 : SLA Z = A<<1

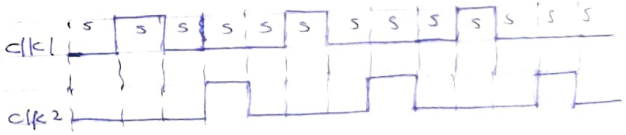


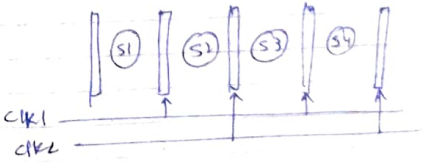
→ It is important that the consecutive stages be applied suitable clocks for correct operation.

Two options:-

↳ ① Use master-slave flipflop in latches to avoid race condition

② Use non-overlapping two phase clock for the consecutive pipeline stages.






```
module pipe_env2 ( zout, rs1, rs2, rd, func, addr, clk1, clk2);
    input [3:0] rs1, rs2, rd, func;
    input [7:0] addr;
    input clk1, clk2,
    output [15:0] zout;
```

```
reg [15:0] L12_A, L12_B, L23_Z, L34_Z;
reg [3:0] L12_rd, L12_func, L23_rd;
reg [7:0] L12_addr, L23_addr, L34_addr;

reg [15:0] regbank [0:15]; // Register bank
reg [15:0] mem [0:255]; // 256 x 16 memory

assign Zout = L34_Z;
always @(posedge clk1)
  begin
    L12_A    <= #2 regbank [rs1];
    L12_B    <= #2 regbank [rs2];
    L12_rd   <= #2 rd;
    L12_func <= #2 func;
    L12_addr <= #2 addr;
  end

always @(posedge clk2)
  begin
    case(func)
      0: L23_Z <= #2 L12_A + L12_B
      1: L23_Z <= #2 L12_A - L12_B
      2: L23_Z <= #2 L12_A * L12_B
      3: L23_Z <= #2 L12_A;
      4: L23_Z <= #2 L12_B;
      5: L23_Z <= #2 L12_A & L12_B;
      6: L23_Z <= #2 L12_A | L12_B;
      7: L23_Z <= #2 L12_A ^ L12_B;
      8: L23_Z <= #2 ~ L12_A;
      9: L23_Z <= #2 ~ L12_B;
      10: L23_Z <= #2 L12_A >>1;
      11: L23_Z <= #2 L12_A<<1;
      default: L23_Z <= #2 16'hzzzz;
    endcase

    L23_rd <= #2 L12_rd;
    L23_rd <= #2 L12_addr;
  end

always @(posedge clk1)
  begin
    regbank [L23_rd] <= #2 L23_Z;
    L34_Z <= #2 L23_Z;
    L34_addr <= #2 L23_addr;
  end

always @(posedge clk2)
  begin
    mem[L34_addr] <= #2 L34_Z;
  end
endmodule
```

Switch level modeling

Verilog provides the ability to model digital circuits at the MOS transistor level

→ four logic ~~levels~~ levels → 0, 1, Z, X.

Various Switch primitives in Verilog

- Ideal MOS switches
  - nMOS, pMOS, CMOS
- Resistive MOS switches
  - rnMOS, rpMOS, rcMOS
- Ideal bidirectional switches
  - tran, tranif0, tranif1
- Resistive Bidirectional switches
  - rtran, rtranif0, rtranif1.
- Power and Ground nets.
  - supply1, supply0.
- Pullup and pulldown.
  - pullup, pulldown

→ Test bench with clock

```
module testbench;
timeunit 1ns;
time precision 100ps;
initial begin
    $display ($time," << Starting the simulation >>");
        rstn = 1'b0;
        clk = 0;
        # rstn = 1'b1;
end.

always #period clk = ~clk;

initial begin
    $dumpfile("your-choice_of-name.vcd");
    $dumpvars;
end
initial begin
  // whatever you come up.
end.
endmodule
```

Test bench → A verilog procedural block that executes only once and used for simulation.

① Test bench generates clock, reset, and required test vectors for given DUT (designed under test)

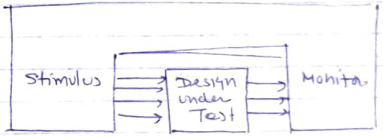
② The test bench can monitors the outputs and present them in a way as specified by the creator.

- print the value of the signal lines.
- Dump the values in a file from where waveform can view.

→ instantiation is calling one module in other module., and passing parameters, (input output)

→ Requirement → input and output should connected with test bench.

→ test bench can use "initial" procedural block, and can also use the "always" for generating some test inputs like clock signal.



Eg

```
1) module example (A,B,C,D,E,F,Y);
    input A,B,C,D,E,F;
    output Y;
    wire t1,t2,t3,X;
    nand #1 g1 (t1,A,B);
    and #2 g2 (t2,C,~B,D);
    nor #1 g3 (t3,E,F);
    nand #1 g4 (Y,t1,t2,t3);
endmodule
```

① input is → "reg"
Output → "wire"

② Instantiate the DUT

③ Initialization and monitoring

```
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; 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
```

Synchronous

- For sequential circuits → we need some clock generation logic
  → Various ways to specify clock signal.
- Test Bench can include various simulator directives.
  → $display, $monitor, $dumpfile, $dumpvars, $finish, etc

The Simulator Directives

- $display ("<format>", exp1, exp2...);
  → Used to print the immediate values of text or variable to stdout.
  → Syntax very similar to C.
  → Additional format specifiers are supported, like "b" (binary), "h" (hexadecimal), etc.

- $monitor ("<format>", var1, var2, ...);
  → Similar in syntax to $display, but does not print immediately.
  → It will print the value(s) whenever the value of some variables in the given list changes.
  → Has the functionality of event-driven print.

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

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

here we can give exp. as well a+b, a*b, c-a.

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

Simulation time

- $finish;
  Terminates the simulation process. eg #100 $finish.

- $dumpfile (<filename>);
  → Specifies the file that will be used to store the values of the selected variables so that they can be graphically visualized later;
  → The file typically has an extension .vcd (value change dump), and contains information about any value changes on the selected variables.

- $dumpoff;
  → This directive stops the dumping of variables. All variables are dumped with "x". Values and the event change of variable will not dumped.

- $dumpon;
  This directives starts previously stopped dumping of variables.

eg initial #100 $dumpoff;
initial #150 $dumpon;

- $dumpvars (level, list_of_variables_or_modules);
  ↳ Specified which variable should be dumped.
  ↳ both parameters are optional, if both are omitted all variables and dumped.
  ↳ If level=0 → then all variables within the modules from the list will be dumped.
  ↳ If level=1 → then only listed variables and variable of listed module will be dumped.

- $dumpall, $dumpvars(0, module1, module2, ...)

eg

dump all.

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

$dumpall;
- The current values of all variables will be written to the file, irrespective of whether there has been any change in their value or not.

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

eg 2-bit equality checker

```
`timescale 1ns/100ps
module comparator (x,y,z);
input [1:0] x,y;
output z;
assign z = (x[1]&y[1]&x[0]&y[0]) |
           (~x[1]&~y[1]&x[0]&y[0]) |
           (~x[0]&y[0]&~x[1]&~y[1]) |
           (x[0]&~y[0]&~x[1]&~y[1]);
endmodule
```

```
`timescale 1ns/100ps module testbench;
reg [1:0] x,y; wire z;
comparator C(.x(x), .y(y), .z(z));

initial
begin
  $dumpfile("comp.vcd");
  $dumpvars(0, testbench);
  x = 2'b01; y = 2'b00;
  #10 x = 2'b10; y = 2'b10;
  #10 x = 2'b01; y = 2'b11;
end
```

```
initial
  begin
  $monitor("t=%g, x=%2b, y=%2b, z=%d", $time,
           x, y, z);
  end
endmodule
```

Other examples

① Test bench for combinational circuit
↳ Full adder → module adder(S, Cout, A, B, C)
input A, B, C;
output S, Cout;
assign S = A^B^C;
assign cout = (A&B) | (B&C) | (C&A);
endmodule

test bench: module testbench;
reg a, b, c; wire sum, cout;
adder ADUT (.A(a), .B(b), .C(c), .S(sum), .Cout(cout));
initial
begin
$monitor($time, "a=%b, b=%b, c=%b, sum=%b, cout=%b", a, b, c, sum, cout);
#3 a=0; b=0; c=1;
#5 b=1;
#5 a=1;
#5 a=0; b=0; c=0;
#5 $finish;
end
endmodule

```
module testbench ;
  reg a,b,c ; wire sum, cout;
  full_adder FA(sum, cout, a,b,c);
  initial
    begin
      a=0; b=0; c=1; #5;
      $display("T=%d, a=%b, b=%b, c=%b, sum=%b, cout=%b",
                $time, a,b,c, sum, cout);
      a=1; #5;
       $display("          ", a,b... );
      a=1 ;#5;
      $display("          ", a,b,...);
```

end
endmodule.

↳ Here we can see that when we want to display the output, we have to display every time.

```
module testbench.;
  integer i;
  initial.
    begin
      for(i=0; i<8; i=i+1)
        begin
          {a,b,c} = i ; #5;
          $display("T=%d, a=%b,...");
        end.
      #5 $finish;
    end
endmodule.
```

② Sequential circuit, test bench

- 4bit shift register

```
module shiftreg_4bit(clock, reset, A, E);
 input clock, reset, A;
 output E;
 reg B,C,D;
 always@(posedge clock) begin.
   if (reset)
      begin
        A<=0; B<=0; C<=0; D<=0; E<=0; end
   else
     begin.
        E <= D;
        D <= C;
        C <= B;
        B <= A;
     end.
 end
endmodule.

module shift;
 reg clk, clr, in; wire out; integer i;
 shiftreg_4bit(clk, clr, in, out);

 initial.
   begin clk = 1'b0; #2 clr = 0; #5 clr = 1; end.
 always #5 clk = ~clk;

 initial begin #2;
   repeat (2)
     begin #10 in=0; #10 in=0; #10 in=1; #10 in=1;
   end
 end.
initial begin. $dumpfile("shifter.vcd");
  $dumpvars(0, shift_test); end endmodul
```