HARDWARE MODELING USING VERILOG

PROF. INDRANIL SENGUPTA
-IIT KHARAGPUR

by

Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

NPTEL
Arti_tyagi_Verilog_2023

HARDWARE MODELING USING VERILOG [IIIK]

⊙ VLSI DESIGN FLOW

- Standardized design procedure
  - starting from the design Idea down to the actual implementation.
- Encompasses many steps:
  - Specification
  - Synthesis
  - Simulation
  - layout
  - Testability analysis
  - and many more

- need to use Computer Aided Design (CAD) tools.
  - Based on Hardware Description language (HDL)
  - HDL's provide formats for representing the output of various design step.
  - A CAD tool transforms its HDL input into a HDL output that contains more detailed information about the hardware.
    - behavioral level to register transfer level
    - Register transfer level to gate level
    - Gate level to transistor level
    - Transistor level to the layout level.

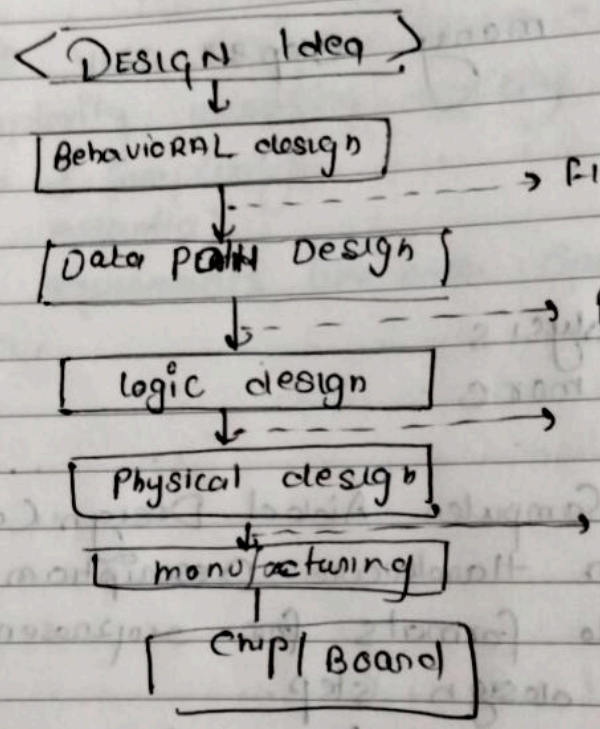
Notes

⊙ Two Competing HDL's :-

① VERILOG
② VHDL

- design are created typically using HDLs, which get transformed from one level of abstraction to the next as the design flow progresses

◉ Simplistic View of Design Flow :-



◉ Step in the Design Flow

- Behavioral Design
- Specify the functionality of the design in terms of its behavior.
- Various way of specifying:
  - Boolean expression or truth table.
  - finite-state machine behavior (eg: State diagram or table
  - in the form of high-level algorithm
- need to be synthesized into more detailed specification for hardware realization.

07 Sunday

Notes

⊙ DATA PATH DESIGN :-

- Generate a netlist of register transfer level components like registers, adders, multipliers, multiplexers, decoders etc.
- A netlist is a directed graph, where the vertices indicates components, and the edges indicates interconnections.
- A netlist specification is also referred to as structural design.
- Systematically transformed from one level to the next.

⊙ logic DESIGN :-

- generate a netlist of gates / flip-flops or standard cells.
- A standard cell is a predesigned circuit module (like gates, flip-flops, multiplexer, etc) at the layout level.
- Various logic optimization techniques are used to obtain a cost effective design.
- There may be conflicting requirements during optimization:
  - minimum number of gates.
  - minimum num of gates levels (i.e delay)
  - minimum signal transition activities (ie. dynamic power)

Notes

⊙ PHYSICAL DESIGN AND MANUFACTURING

- Generates the final layout that can be sent for fabrication.

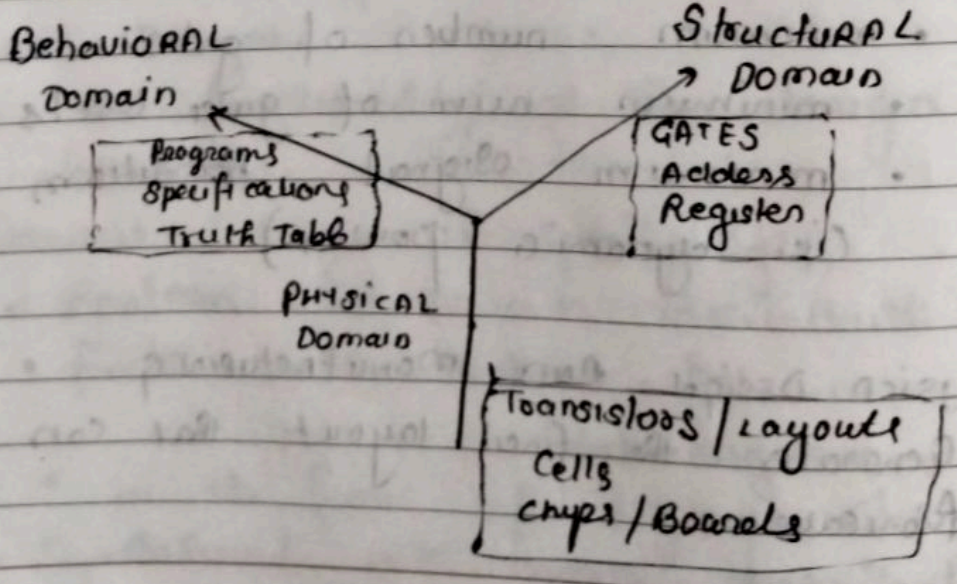
OTHER STEP IN THE DESIGN FLOW

- Simulation for verification
  - At various level: logic level, switch level, circuit level.
- FORMAL VERIFICATION
  - used to verify the design through formal techniques
- TESTABILITY analysis and TEST PATTERN GENERATION
  - required for testing the manufactured devices.

Lecture ② DESIGN REPRESENTATION

- A design can be represented at various levels from three different point of view:
  1. Behavioral
  2. Structural
  3. Physical
- Can be conveniently expressed by Y-diagram.

∴ Y diagram means it has shape Y.



Notes

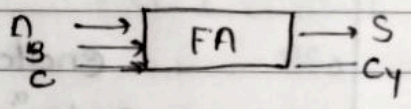
⊙ BEHAVIORAL REPRESENTATION

- Specifies how a particular design should respond to a given set of inputs.
- may be specified by :-
  - Boolean equation
  - Tables of Input and output values
  - Algorithms written in standard HLL like C
  - Algorithms written in special HDL like verilog or VHDL.

eg :-

Full Adder :-

- two operand input A and B
- a carry input C
- a carry output C_4
- a sum output S



- expression in terms of Boolean expression;

$$S = A \cdot B' \cdot C' + A'B'C + A'BC' + ABC$$

$$S = A \oplus B \oplus C$$

$$C_4 = AB + AC + BC$$

⊙ Expression in verilog in terms of Boolean expression

```
module carry (S, C4, A, B, C);
    input A, B, C;
    output S, C4;
    assign S = A^B^C;
    assign C4 = (A&B) | (B&C) | (C&A);
endmodule
```

name module

Parameters

∴ this is like the function in C.

Notes

163-203

⊙ Express in VERILOG in terms of Truth table (only C_y is shown) ; $C_y \rightarrow$ carry

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

Keyword → primitive; name → Carry; Parameters → (Cy, A, B, C);

? means don't care

∴ when $C_y = AB + BC + CA$

$\Downarrow$

A	B	C
1	1	0
1	0	1
0	1	1
1	1	1

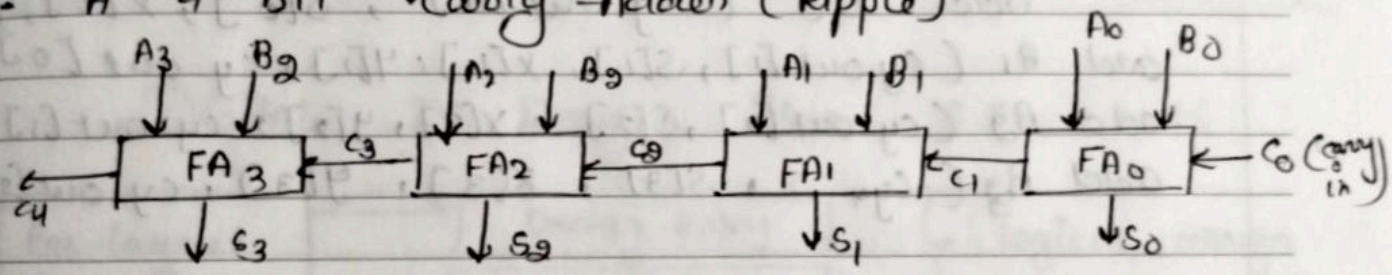
∴ All other cases $C_y = 0$

⊙ STRUCTURAL REPRESENTATION:

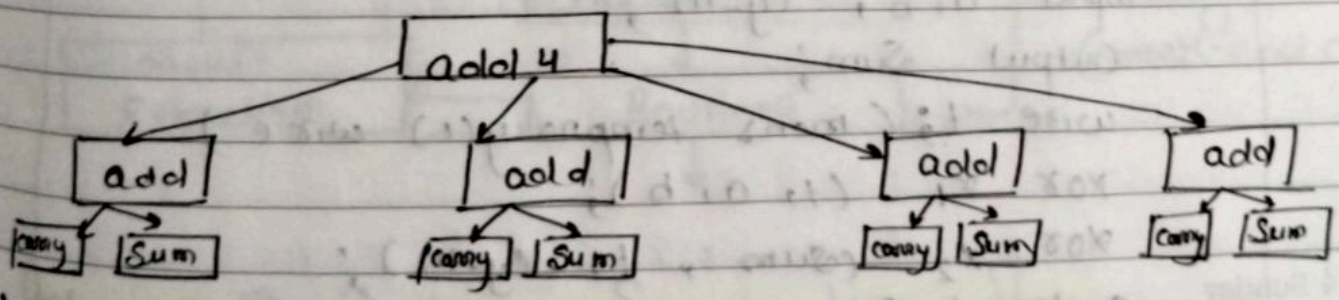
- Specifies how components are interconnected.
- In general, the description is a list of modules and their interconnections.
- called netlist.
- Can be specified at various levels.

- At the structural level, the levels of abstraction are:
  - The module (Functional level)
  - The gate level
  - The transistor level
  - Any Combination of above
- In each successive level detail in revealed about the Implementation.

eg:- A 4-bit Carry-Adder (Ripple)



- Consist of four full adders.
- Each full adder consists of a Sum circuit and a Carry circuit.



Carry = $A \cdot B + B \cdot C + C \cdot A$
Sum = $A \oplus B \oplus C$

- we instantiate Carry and Sum Circuits to create a full adder.
- we instantiate four full adders to create the 4 bit adder.

→ expression in verilog

```
module add4 ( S, Cy4, Cy_in, x, y );
    input [3:0] x, y;
    input Cy_in;
    output [3:0] S;
    output Cy4;
    wire [2:0] Cy_out;

        add B0 ( Cy_out [0] , S[0] , x[0], y[0], Cy_in);
    add B1 ( Cy_out [1], S[1], x[1], y[1], Cy_out [0]);
    add B2 ( Cy_out [2], S[2] , x[2], y[2], Cy_out [1]);
    add B3 ( Cy4 , S[3] , x[3], y[3], Cy_out [2]);

end module
```

[3:0] means 4 bit num

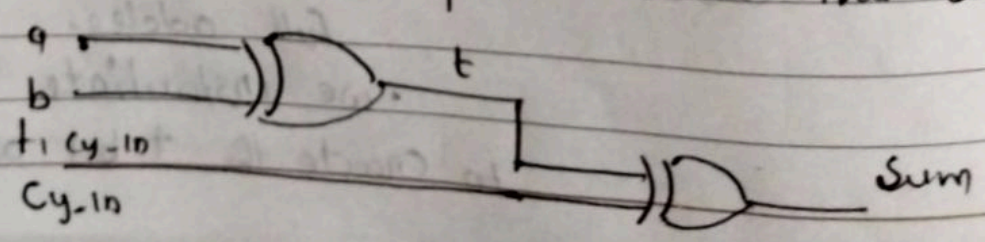
∴

```
module Sum (Sum, a, b, Cy_in);
    input a, b, Cy_in;
    Output Sum;
    wire t; ( mens temprary(t) wire )
    xor x1 (t, a, b);
    xor x2 (Sum, t, Cy_in);
endmodule
```

14 Sunday

∴ this is structural desige

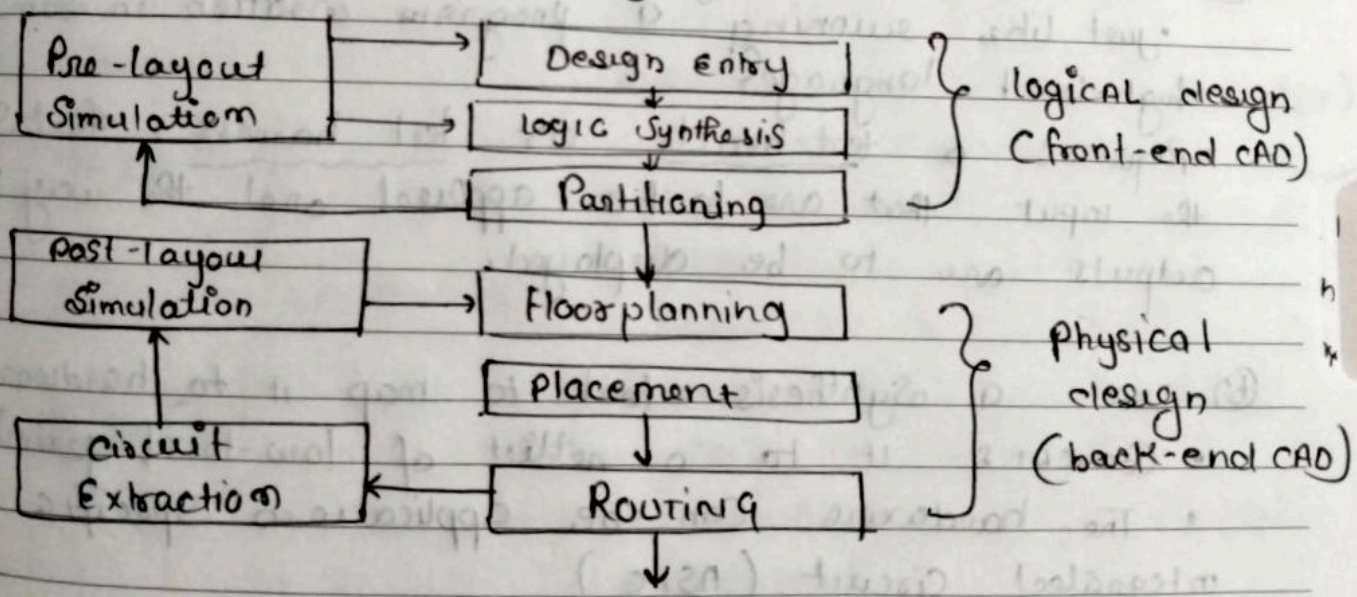
∴ xor x1 (t, a, b) → representation the wire



∴ xor x2 (Sum, t, Cy_in

```
module Carry (cy_out, a, b, cy_in);
  input a, b, cy_in;
  Output cy_out;
  wire t1, t2, t3;
  and g1 (t1 a, b);
  and g2 (t2 a, c);
  and g3 (t3, b, c);
  or g4 (cy_out, t1 t2, t3);
endmodule
```

Digital IC Design Flow : A quick look



Lecture ⇒ ③ Getting Started with Verilog

→ Why do we use Verilog?

- To describe a digital System as a set of modules.
  - each of the module will have an interface to other modules, in addition to its description

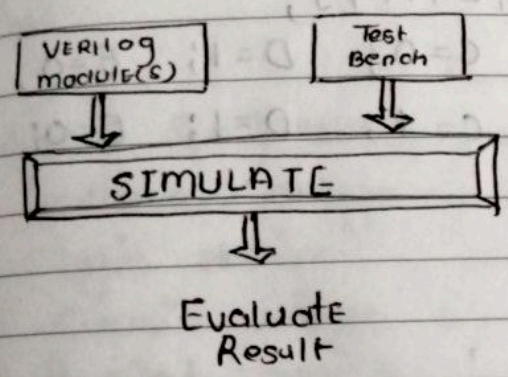
168-198

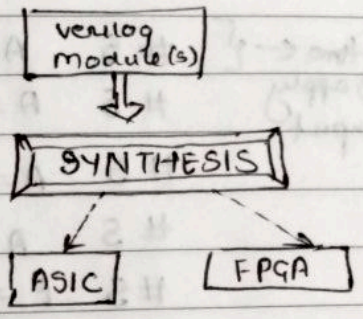
- Two way to specify a module
  - (a) By specifying its internal logical structure (called structural representation).
  - (b) By describing its behavior in a program-like manner (called behavioral representation).
- The modules are interconnected using nets, which allow them to work with each other. (nets → just like wire)

- (•) AFTER SPECIFYING the system in VERILOG, we can do two things:
  - (a) Simulate the system and verify the operation.
    - just like running a program written in some high-level language.
    - Requires a test-bench or test harness, that specify the input that are to be applied and the way the outputs are to be displayed.
  - (b) Use a synthesis tool to map it to hardware.
    - Converts it to a netlist of low-level primitives.
    - The hardware can be application specific integrated circuit (ASIC)
    - or else, it can be field programmable gate array (FPGA).

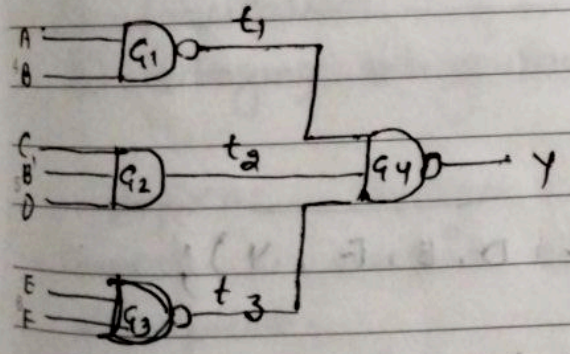
Notes (•) Using ASIC as hardware target?
- when high performance & high packing density is required.
- when the manufactured hardware is expected to be used in large numbers (e.g process

- using FPGA as hardware target?
- When Fast turnaround time is required to validate the design.
- The mapping can be done in the laboratory itself with a FPGA kit and associated software.
- There is a tradeoff in performance.





An Example :-




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

we can combine declarations of same type of gate together

```
nand #1 G1 (t1, A, B),
        G4 (Y, t1, t2, t3);
```

Notes

Now we work to TestBench :-

```
module Testbench;
  reg A, B, C, D, E, F; wire Y;
```

170-196

```
example   DUT(A,B,C,D,E,F,Y);

  initial
    begin   (like printf)                  but means binary
      $monitor ($time, " A = %b , B = %b , C = %b, D = %b
                        E = %b, Y = %b ",
                        A, B, C, D, E, F, Y);

time ← { #5  A=1;  B=0;  C=0;  D=1;  E=0;  F=0;
mis 5 apply  #5  A=0;  B=0;  C=1;  D=1;  E=0;  F=0;
input
         #5  A=1;  C=0;
         #5  A=1;
         #5  F=1;
         #5  $finish;
    end
  endmodule
```

∴ Orignal file
file name — example.v

```
module example (A, B, C, D, E, F, Y);
  wire t1, t2, t3, Y;
    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
```

Notes

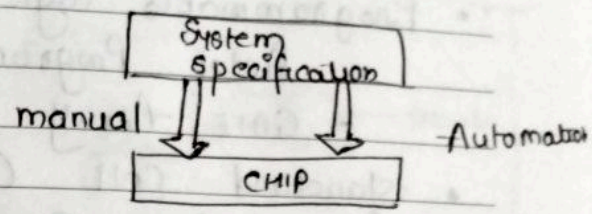
∴ (value change dump)

```
$dumpfile ("example.vcd");
$dumpvars (0, testbench)
```

Lecture :=> ④ VLSI DESIGN STYLES PART - 1

VLSI DESIGN CYCLE

- large number of device
- optimization requirements for high performance
- Time-to-market Competition
- Cost



① System Specification
② functional design
③ logic design
④ Circuit design
⑤ Physical design
⑥ Design Verification
⑦ fabrication
⑧ Packaging, testing and debugging.

PHYSICAL DESIGN

- Converts a circuit description in to a geometric description.
  - This description is used for fabrication of the chip.
- Basic steps in the physical design cycle:
  ① Partitioning, floorplaning and placement
  ② Routing
  ③ Static time Analysis
  ④ Signal integrity and crosstalk analysis
  ⑤ Physical Verification and Sign off.

Notes

Various design Styles

- Programmable logic devices
  - field Programmable Gate Array [FPGA]
  - GATE Array
- Standard Cell (Semi-Custom Design)
- full-Custom Design

→ which Design Style to use?

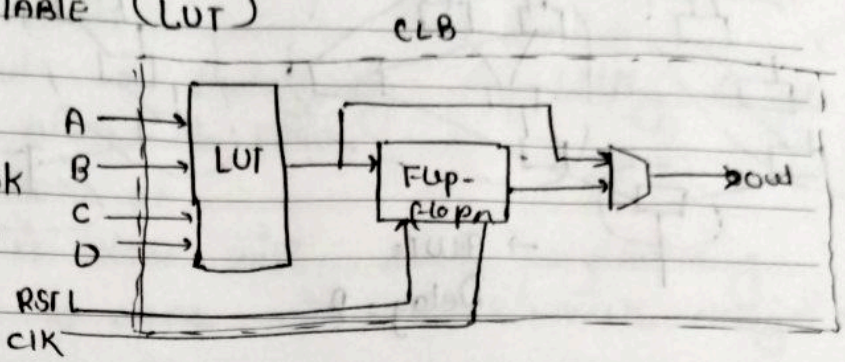
- Basically a tradeoff among several design Parameters
  - Hardware Cost
  - Circuit delay
  - Time Required
- optimizing on these Parameters is often Conflicting

⊙ FIELD PROGRAMMABLE GATE ARRAY (FPGA)

- what does FPGA OFFER?
- USER / Field PROGRAMMABILITY.
  - Array of logic cells Connected via routing channels
  - different types of cells:
    - Special I/O cells;
    - Logic cells (mainly lookup tables (LUT) with associated registers).

  - Interconnection Between Cells:
    - using SRAM based Switches.
    - using anti-fuse elements.

Look up Table (LUT)

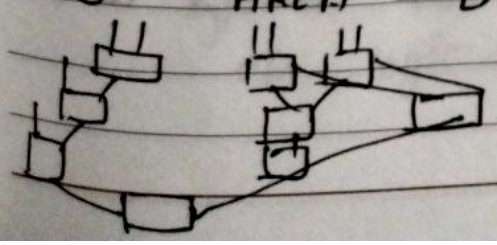


- Combinatorial logic is stored in 16x1 SRAM look up Tables (LUTs) in CLB.
- Capacity is limited by number of input, not complexity.
- Choose to use each function generator as 4-input logic (LUT) or as high-speed RAM.

⊙ LUT mapping : An Example

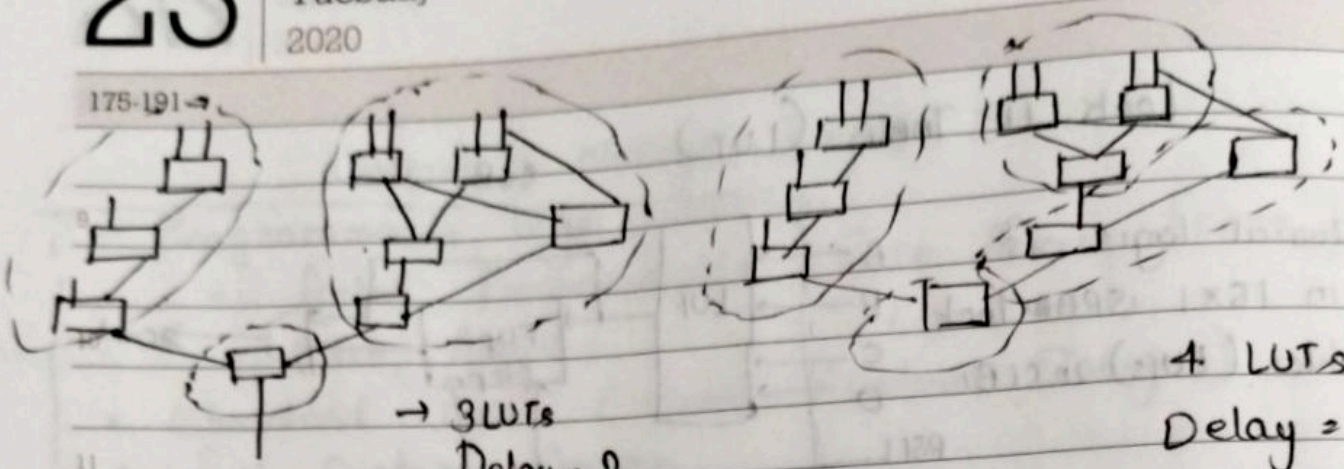
- A function : $f = A'B + B'C \cdot D$
- The mapping process:
  - Create the truth table of the 4-variable function
  - Load the output column into the SRAM corresponding to the LUT.
  - Apply the function input to the LUT inputs.
- Any 4-variable function can be realized.
  - Mellot to LUT mapping is an interesting closing trade off.

Notes ⊙ AREA - DELAY TRADE OFF



given netlist

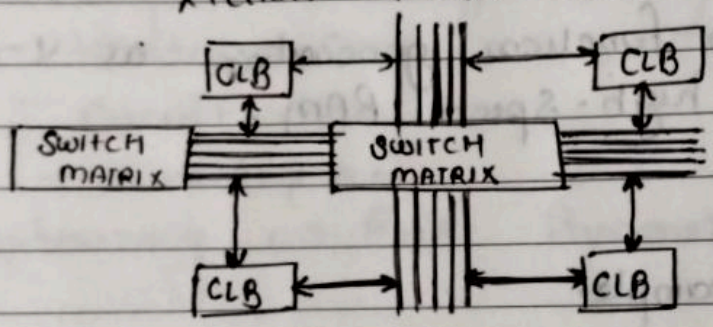
175-191



→ 3 LUTs
Delay = 2

4 LUTs
Delay = 3

⊙ XILINX FPGA ROUTING



① First direct interconnect.
CLB to CLB

② General purpose interconnect
- uses switch matrix

⊙ FPGA Design Flow

- ⊙ Design Entry
  - in Schematic, VHDL or verilog.
- Implementation
  - Placement & Routing
  - Bitstream generation
  - Analyze timing, view layout, simulation, etc.
- Download
  - Directly to xilinx hardware devices with unlimited reconfigurations.

Notes

LECTURE :- ⑤ INTRODUCTION

- In view of the Speed of Prototyping capability, the gate array (GA) comes after the FPGA.
- Design implementation of
  - FPGA chip is done with user programming.
  - Gate array is done with metal mask design and processing.

- Gate array implementation requires a two-step manufacturing process:

ⓐ The first phase, which is based on generic (standard) masks, results in an array of uncommitted transistors on each GA chip.

ⓑ These uncommitted chips can be customized later, which is completed by defining the metal interconnected between the transistor of the array.

- The GA chip utilization factor is higher than that of FPGA.
  - The used chip area divided by the total chip area.

- Chip speed is also higher.
  - more customized design can be achieved with metal mask designs.

Notes
- Typical gate array chips can implement millions of logic gates.

⊙ STANDARD Cell BASED DESIGN

ASIC
↓ Standard cell (semi custom)
↓ Full custom

→ INTRODUCTION:

- one of the most prevalent design styles.
  - Also called semi-custom design style.
  - Requires developing full custom mask set.

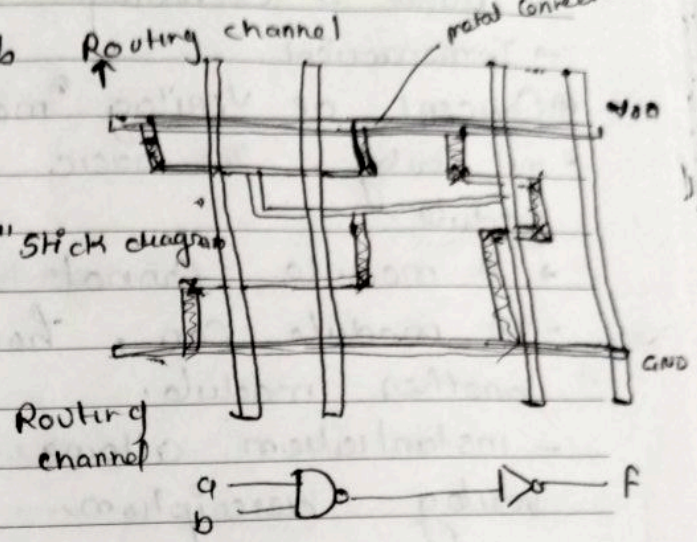
- Basic Idea:
  - Commonly used logic cells are developed, and stored in a standard cell library.
  - Typical library may contain a few hundred cell (inverters, nand gates, NOR Gates, AOI gates, 2-to-1 mux, D-latches, flip-flops, etc)

⊙ CHARACTERISTIC OF THE CELL'S

- Each cell is designed with a fixed heigh.
  - To enable automated placement of the cells and routing of inter-cell connections.
  - A number of cell can be abutted side-by-side to form rows.
- The Power & ground rails typically run parallel to upper and lower boundaries of cell.
  - Neighboring cells share a common power and ground bus.
- The I/P and O/P

Notes

- Standard cell example
  - made to stack side by side
    - fixed height
    - width can vary
    - can about VDD and GND.



Notes

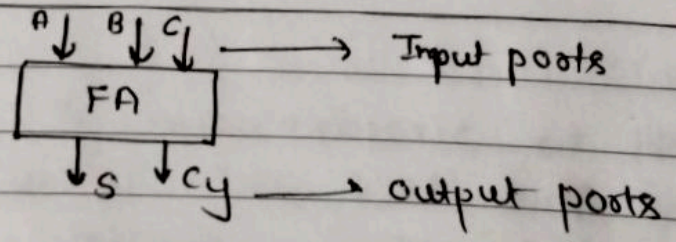
LECTURE ⇒ ⑥ VERILOG LANGUAGE FEATURES

→ INTRODUCTION

◉ CONCEPT OF VERILOG "module" →

- IN verilog, The basic unit of hardware is called a module.
- → A module cannot contain defination of other module
- A module can, however, be instantiated within another module.
- instantiation allows the creation of hierarchy in verilog description.

```
module module_name (list_of_Ports);
    Input/output declarations
    local net declarations
    Parallel statements
endmodule
```

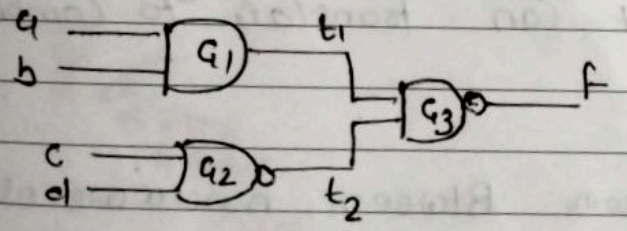



```
// A Simple AND Function
module simpleand (f, x, y);
    input x, y;
    output f;
    assign f = x & y;
endmodule
```

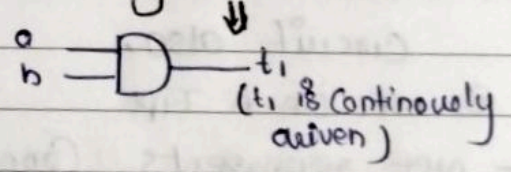
eg:- /* A 2-level Combinational Circuit */

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

- This is behavioral description.
- one possible gate level realization is shown.
- t_1 and t_2 are intermediate lines; termed as wire data type.



assign t_1 = a&b;



◉ Point to NOTE:

- The "assign" statement represents continuous assignment, whereby the variable on the LHS gets updated whenever the expression on the RHS change.

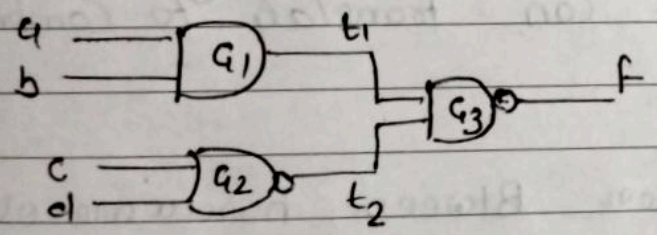
  assign variable = expression.

- The LHS must be a "net" type variable, typically a "wire"
- The RHS can contain both "register" and "net" type variables.

eg:- /* A 2-level Combinational Circuit */

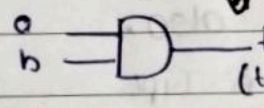
```
module two_level (a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire t1, t2;   //intermediate lines
    assign t1 = a&b;
    assign t2 = ~(c|d);
    assign f = ~(t1|t2);
endmodule
```

- This is behavioral description.
- one possible gate level realization is shown.
- t_1 and t_2 are intermediate lines; termed as wire data type.



assign t_1 = a&b;

⇓



(t_1 is continuously driven)

◉ Point to NOTE:

- The "assign" statement represents continuous assignment, whereby the variable on the LHS gets updated whenever the expression on the RHS change.

  assign variable = expression.

- The LHS must be a "net" type variable, typically a "wire"
- The RHS can contain both "register" and "net" type variables.

- The "assign" statement model behavioral design style, and is typically used to model combinational circuits

⊙ DATA TYPES IN VERILOG

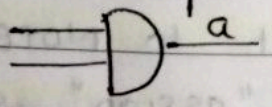
- A variable in verilog belongs to one of two data types:

ⓐ Net
- must be continuously driven.
- Cannot be used to store a value.
- used to model connections between continuous assignments and instantiations.

ⓑ Register
- Retains the last value assigned to it
- often used to represent storage elements, but sometimes it can translate to combinational circuit also.

⊙ NET DATA TYPE

- net represents connection between hardware elements
- Net are continuously driven by the outputs of the devices they are connected to.



- Net "a" is continuously driven by the o/p of the AND Gate.
- Nets are 1-bit values by default unless they are declared explicity as vectors.

Notes
- Default value of a net is "z".
- Various "Net" data types are supported for synthesis in verilog:
- wire, wor, wand, tri, supply0 etc.

183-183

DATA VALUES and Signal Strengths

- Verilog supports 4 value levels and 8 strength levels to model the functionality of real hardware,
- strength level are typically used to resolve conflicts between signal drivers of different strengths in real ckt.

Value level	Represents
0	Logic 0 state
1	logic 1 state
X	Unknown logic state
Z.	High impedance state.

initialization

- All unconnected nets are set to "Z".
- All register variable set to "X".

Lecture ⇒ ① VERILOG LANGUAGE FEATURES

ⓑ REGISTOR DATA TYPE

⊙ in verilog, a "register" is a variable that can "hold" a value.

- Unlike a "net" that is continuously driven and cannot hold any value.
- Does not necessarily mean that it will map to a hardware register during synthesis.
- Combinational circuit specifications can also use register type variables.

Notes

Register data types supported by verilog:

(i) reg : most widely used
(ii) integer : used for loop counting (typical use)
(iii) real : used to store floating-point numbers.
(iv) time : keeps track of simulation time (not used in synthesis)

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

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

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

Example

```
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
```

apply for whenever there positive edge of the coming.

∴ 32-bit Counter with synchronous reset.
- Count value increase at the positive edge of the clock."
- If "rst" is high, the Counter is reset at the +ve edge of the next clk.

Notes

185-181

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

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

- 32-bit Counter with asynchronous reset.
  - Here reset occurs whenever "rst" goes high.
  - Does not synchronize with clk.

◉ "integer" data type:
- It is a general-purpose register data type used for manipulating quantities.
- more convenient to use in situation like look counting than "reg".
- it is treated as a 2's complement signed integer in arithmetic expressions.
- default size is 32 bits; however, the synthesis tool tries to determine the size using data flow analysis.

example -

185-181

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

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

- 32-bit Counter with asynchronous reset.
  - Here reset occurs whenever "rst" goes high.
  - Does not synchronize with clk.

◉ "integer" data type:
- It is a general-purpose register data type used for manipulating quantities.
- more convenient to use in situation like loop counting than "reg".
- it is treated as a 2's complement signed integer in arithmetic expressions.
- default size is 32 bits; however, the synthesis tool tries to determine the size using data flow analysis.

example -

```
wire [15:0] x, y;
integer c;
c = x+y;
```

- size of c can be deduced to be 17 (16 bit plus a carry)

- "Real" data type:
  - used to store floating-point numbers.
  - when a real value is assigned to an integer, the real number is rounded off to the nearest integer.
  - Example:

```
real e, pi;
initial
    begin
    e = 2.718
    pi = 314.159e-2;
  end
```

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

- "time" data type:
  - in verilog, simulation is carried out with respect to a logical clock called simulation time.
  - The "time" data type can be used to store simulation time.
  - The system function "$time" gives the current simulation time.
  - eg

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

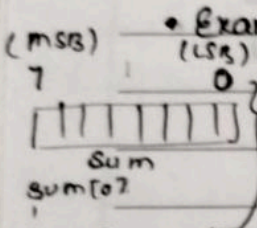
188-178

⊙ VECTORS

- Nets or "reg" type variable can be declared as vectors of multiple bit widths.
  - → if bit width is not specified, default size is 1-bit.

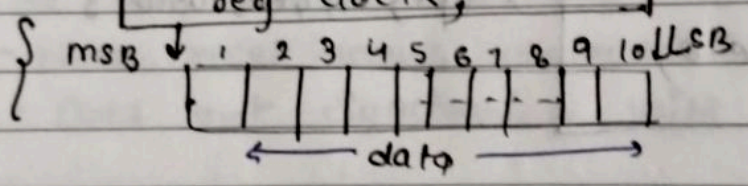
- vectors are declared by specifying a range [range1 : range2] where range1 is always the most significant bit and "range2" is the least significant bit.

- Example :




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

|| Single bit variables
|| MSB is sum[7], LSB is sum[0]
|| MSB is data[1], LSB is data[10]

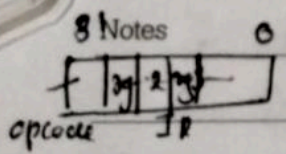


→ indivisual bit i can access by writing
sum[0]
sum[1]
⋮
sum[7]

∴ this is just like an array

⊙ Parts of a vector can be addressed and used in an expression.

- Example
  - A 32 bit instruction register, that contains a 6-bit opcode, three register operands of 5 bit each, and an 11-bit offset.




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

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

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
opcode	reg1	reg2	reg3	offset

Instruction register (IR)

⊙ Multi-Dimensional Arrays And memories

- multi-dimensional array of any dimension can be declared in verilog.
- example;

```
reg [31:0] register_bank [15:0]; // 16-32 bit registers
integer matrix [7:0][15:0];
```

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

```
reg mem_bit [0:2047];          // 2K 1-bit words
reg [15:0] mem_word [0:1023];  // 1K 16-bit words
```

⊙ Specifying Constant Values

- A constant value may be specified in either the sized of the unsized form.
- syntax of sized form:

  <size>'<base><number>

- Example:

4'b0101	// 4-bit binary number 0101
1'b0	// logic 0 (1-bit)
12'hB3C	// 12 bit number 1011 0011 1100
12'h8xF	// 12 bit number 1000 xxxx 1111
25	// signed number, in 32 bits (size not specified)

variable of type integer and real are typically expressed in unsized form.

Notes

190-176

- PARAMETER

- A parameter is a constant with a given name.
  - we cannot specify the size of a parameter.
  - The size gets decided from the constant value itself; if size is not specified, its taken to be 32 bits.

```
Parameter HI = 25 , LO = 5;
Parameter up = 2'b00 , down = 2'b01 , steady = 2'b10
```

Example

```
// Parameter 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
```

* any variable assigned within the "always" block must be of type "reg".

Notes

Asti_tyagi_verilog

Lecture :- ⑧ Verilog language features

⊙ Predefined logic gates in Verilog

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

∴ x → undefined ∴ z → hugimpedace

⊙ List of Primitive Gates :-

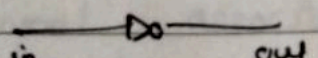
```
and  G(out, in1, in2);
nand G(out, in1, in2);
or   G(out, in1, in2);
nor  G(out, in1, in2);
XOR  G(out, in1, in2);
XNOR G(out, in1, in2);
not  G(out, in);
buf  G(out, in);
```

```
buf if1  G(out, in, ctrl);
buf if0  G(out, in, ctrl);
notif 0  G(out, in, ctrl);
notif 1  G(out, in, ctrl);
```

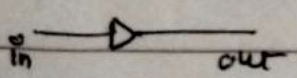
∴ there are gates with tristate controls.

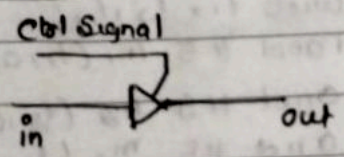
Notes

∴ Not GATE



∴ buf

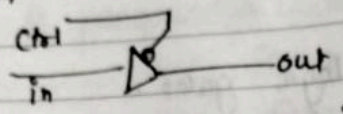




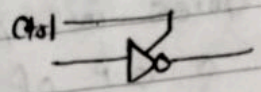
if ctrl = 1 out = in
= 0 out = Z

this is $\boxed{\text{buf if 1}}$

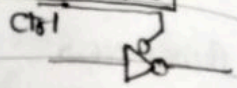
[buf if 0]



[Notif 1]



[notif 0]



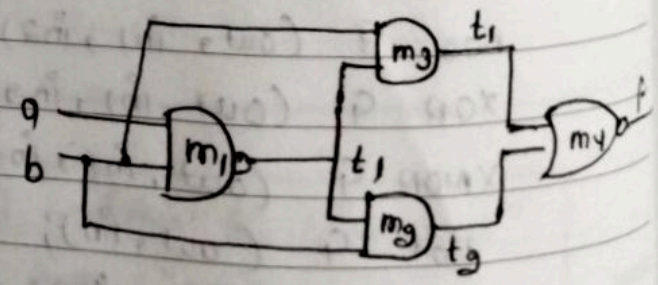
If ctrl = 0 , out = in
= 1 , out = Z

⊙ Some restriction when Instantiating primitives gates:

- The output port must be connected to a net (e.g a wire)
  - An "output" Signal is a wire by default, unless explicitly declared as a register.
- The input port may be connected to net or register type variable.
- They have a Single output but can have any number of inputs (except NOT & Buf).
- when instantiating gate, an optional delay may be Specified
  - Used for simulation.
  - logic Synthesis tool ignore the time delays.

Example :- shows this delays :-

```
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
```



Notes

- THE 'timescale DIRECTIVE

- often in a single simulation, delay values in one module need to be specified in terms of some time unit, while those in some other module need to be specified in terms of some other time unit.
- The 'timescale compiler directive can be used:

'timescale <refence_time_unit> / <time_precision>

- The <reference_time_unit> specifies the unit of measurement for time.
- The <time_precision> specifies the precision to which the delays are rounded off during simulation.
  - valid values for specifying time unit & time precision are 1, 10 and 100.
- Example:

'timescale 10ns/1ns

- Reference time unit is 10ns, and simulation precision is 1ns.
  - if we specify #5 as delay, it will mean 50ns.
  - The time units can be specified in s (second), ms (millisecond) us (microsecond), ps (picosecond) & fs (femtosecond)

- Specifying Connectivity during Instantiation

- when a module is instantiated within another module, there are two ways to specify the connectivity of the signal line btw the two modules
  (a) positional association
  (b) Explicit association.

Arti_Tyagi_verilog

195-171

HARDWARE MODELING ISSUES

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

- A "wire"
- A "Flip-flop" (edge triggered storage cell)
- A "latch" (level triggered storage cell)

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

- A "net" data type always map to a "wire" ~~or "register"~~, during synthesis.
- A "register" data type maps either to "wire" or a "storage cell" depending upon the context under which a value assigned.

```
module reg_maps_to_wire (A,B,C,F1,F2);
    input A,B,C;
    output F1,F2;
    wire A,B,C;
    reg F1,F2;

    always @ (A or B or C)
    begin
        F1 = ~(A & B);
        F2 = F1^C;
    end
endmodule
```

Notes

Arti_Tyagi_verilog

Lecture ⇒ (9) Verilog Operators

- Arithmatic operators:

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

example
- (b+c) [binary operator
(a-b)+(c*d)
a % b
a ** 3

- Logical operators:

!	logical negation (meaning not
&&	logical AND
||	logical OR

example
(done && ack)
(all b)
!(a && b)
((a>b) && !(b>c))

- The value 0 is treated as logical FALSE while any non-zero value is treated as TRUE.
- logical operators return either 0 (FALSE) or 1 (TRUE)

→ RELATIONAL OPERATOR

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

eg →
(a != b)
((a+b) == (c-d))
(count <= 0)

Notes

197-169

- Relational Operators operate on numbers and Return Boolean Value (true or false).

→ Bitwise operator

~	bitwise not
&	bitwise AND
|	Bitwise OR
^	Bwise XOR
~^	Bitwise XNOR

eg:

```
wire a,b,c,d,f1,f2,f3,f4;
assign f1 = a~|b;
assign f2 =(a&b)|(b&c);
assig f3 = a^b^c;
```

- Bitwise operators operate on bits & return a value that is also a bit.

- Reduction operator accept a single word operand and produce a single bit as output.
  - operator on all the bits within the word.

Reduction operator

&	Bitwise AND
|	Bitwise OR
~&	Bitwise NAND
~|	Bitwise NOR
^	Bitwise exclusive-OR
~^	Bitwise exclusive-NOR

example:

```
wire [7:0] a,b,c;
wire f1, f2, f3;
assign a =4'b0111;
assign b =4'b1100;
assign c =4'b0100;
assign f1 = ^a; //
assign f2 =&(a^b); //
                   gives a0
assign f3 = ^a & ~^b
//gives a 1
```

Notes ◉ The relational operator (>, <, <=, >=, !=, ==) also evaluate to a 1 bit result (0 or 1).

→ Shift Operators:

>>	Shift right
<<	left shift
>>>	arithmatic shift right

eg:
```
wire [15:0] data target
assign target = data >> 3;
assign target = data >> 2
```

∴ >>> :: 2's Complement number System

1	

1 → -ve, 0 → +ve

∴ Shift right → ÷2

shift left → ×2

→ Conditional operator:

Cond_expr ? true_expr : false_expr;

• Concatenation operator:
(..., ..., ...)

→ joins together bits from two or more comma-separated expression

• Replication operator:
{n{m}}

→ joins together n copies of an expression m, where n is an constant

example

```
assign f = {a, b};
assign f = {a, 3'101, b}
assign f = {x[2], y[0], a};
assign f = {2'b10, 3{2'b01}, x}
```

Example

```
module operator_example (x, y, f1, f2);
    input x, y;
    output f1, f2;
    wire [9:0] x, y;   wire [4:0] f1;   wire f2
      assign f1 = x[4:0] & y[4:0];
      assign f2 = x[9] ^ f1[9];
      assign f2 = ~ X;
      assign f1 = f2 & x[2:5] : x[4:0];
```

Notes

// 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

- operator on same line have the same precedence.
- All operators associate left or right in an expression except ?.
- parenthesis can be used to change the precedence.

∴ {==, !=} Check for equality

{===, !==} exact equality

+ - ! ~ (unary)	
**	
* / %	
<< >> >>>	
< <= > >=	
== != === !==	
& ~&	
^ ~^	
| ~|	
&&	
||	
?:	

Precedence increase

⊙ Some POINTS :

- The presence of a 'z' or 'x' in reg or wire being used in an arithmetic expression result in the whole expression being unknown ('x').

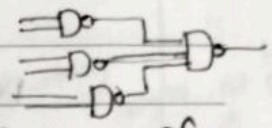
⊙ The logical operators (!, &&, ||) all evaluate to a 1-bit result (0, 1, or x).

Aditya / - Verilog

Lecture ⇒ (10) VERILOG MODELING Examples.

Modeling
- Behavioral → Starting point
- Structural → Hardware Implementation

CY = 4 NAND gate

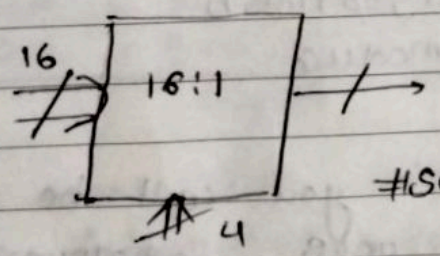


Example (1) The structural hierarchical description of a 16-to-1 multiplexer.

(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.

Sol



Select line = $\log_2$ (input line)

Sunday 19

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
```

- Selects one of the input bits depending upon the value of 'Select line (Sel)'.

```
module muxtb;
  reg [15:0]A; reg [3:0]S; wire F;

  mux16to1 M(.in(A), .sel(S), .out(F));

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

Test bench purpose :- you will be testing your module once whatever you specified those values will be applying to the inputs and we will seeing what the outputs are coming. that is the purpose of the test bench.

Notes

[explaining test Bench.]

→ $ monitor → we are monitoring a few of the variables

→ $time → it is a System variable that indicates the simulation time.

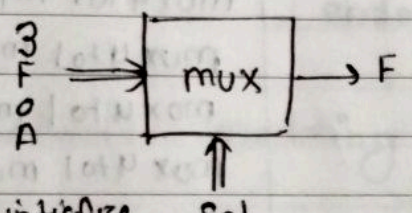
→ "A = %h" → mean see like, the we want to print the values A,S,F. h is hexadecimal specifier ~~becaCoz~~ because A is 16 bit number, S is 4 bit and F is single bit (F = %b (binary)).

→ #5 A = 16'h 3F0a , 90 3 F0A → (90 delay) (15 delay) (delay to at 5 time)

90 · 3F0A →

0 0 1 1	1 1 1 1	0 0 0 0	1 0 1 0
15 14 3 3 12	11 F 10 9 8	7 0 6 5 4	3 A 2 9 0

mux :



S → select line = 4'h 0

Select line = 0

Select line = 1

Select line = 6

S = C = 12

0A	= xxxx,	S = x	, F = x
5A	= 3f0a,	S = 0	; F = 0
10A	= 3f0a,	S = 1	, F = 1
15A	= 3f0A,	S = 6	, F = 0
20A	= 3f0A,	S = C	, F = 1

at time 0 not in ticks; output; time (delay); input

→ this is tabular form

$dumpfile → It means these variations with time this information you can also dump in to a file VCD.

"mux 16 to 1.vcd" → file name

Notes

→ why need dumfile and dump variable?

→ seeing the simulation output in a tabular form or just on the screen is not good enough in form for text.

22 July Wednesday 2020

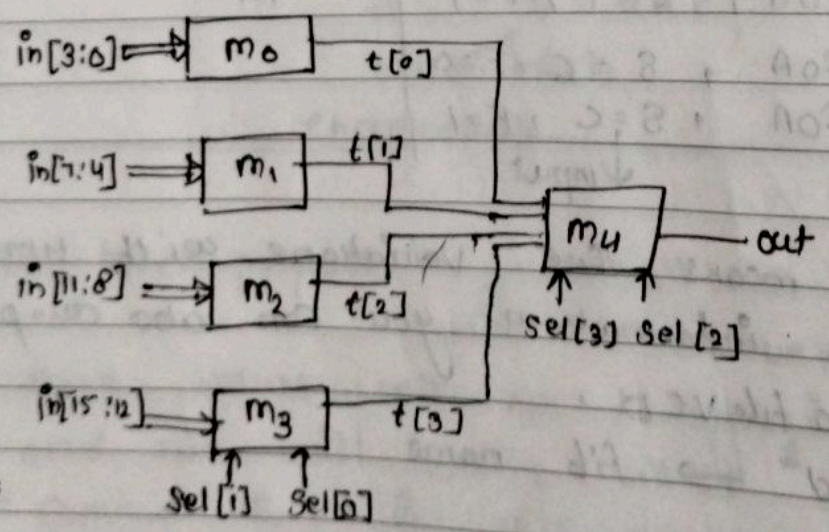
204-162

→ VERSION ② Behavioral modeling OF 4-to-1 mux
STRUCTURAL modeling OF 16-to-1 mux

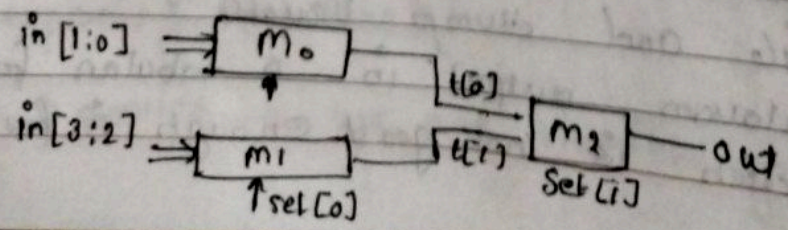
```
module mux 4 to 1 (in, sel, out);
    input [3:0] in;
    input [1:0] sel;
    output out;

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

```
module mux 16 to 1 (in, sel, out);
    input [15:0] in;
    input [3:0] sel;
    output out;
    wire [3:0] t;

    mux4to1 m0 (in[3:0], sel[1:0], t[0]);
    mux4to1 m1 (in[7:4], sel[1:0], t[1]);
    mux4to1 m2 (in[11:8], sel[1:0], t[2]);
    mux4to1 m3 (in[15:12], sel[1:0], t[3]);
    mux4to1 m4 (t, sel[3:2], out);
end module
```

: Circuit diagram → 16-to-1 multiplexer using 4-to-1 multiplexer



Notes



← 4 to 1 multiplexer using 2-to-1 multiplexer

→ VERSION ③ Behavioral modeling OF 2-to-1 mux
Structural modeling OF 4-to-1 mux

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

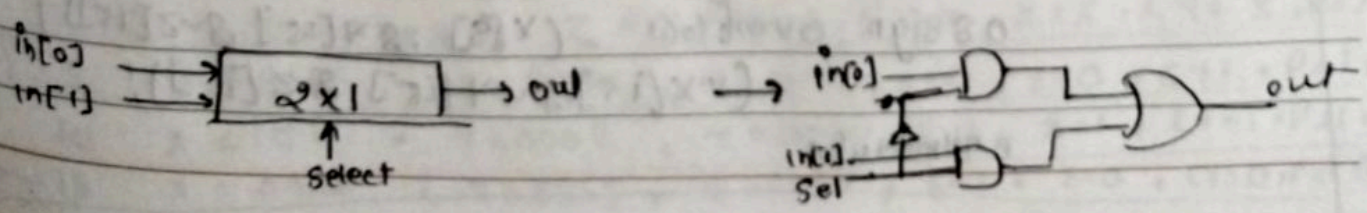
```
module mux4to1 (in, sel, out);
    input [3:0] in;
    input [1:0] sel;
    output out;
    wire [1:0] t;
    mux2to1 m0 (in[1:0], sel[0], t[0]);
    mux2to1 m1 (in[3:2], sel[0], t[1])
endmodule
```

VERSION ④ Structural modeling OF 2-to-1 mux

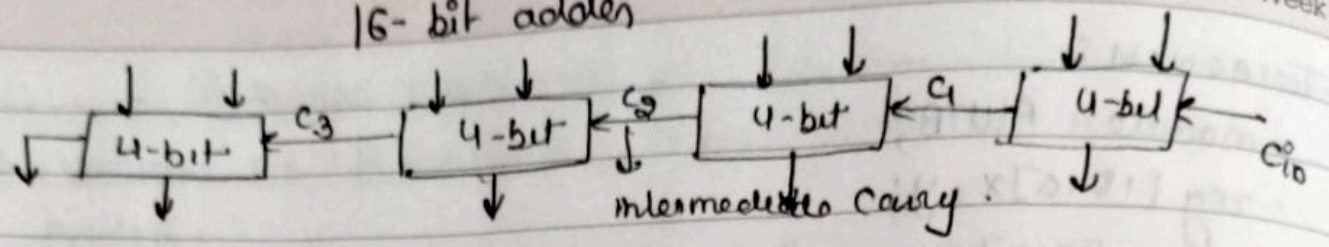
```
module mux 2to1 (in, sel, out);
    input [1:0] in;
    input sel;
    output out;
    wire t1, t2, t3;

    NOT G1 (t1, sel);
    AND G2 (t2, in[0], t1);
    AND G3 (t3, in[1], sel);
    OR G4 (out, t2, t3);
endmodul
```

Point to NOTE:
- Same test bench can used for all the version
- Two Versions illustrate hierarchical refinement OF design.

Notes



16- bit adder



VERSION ② Structural description of 16-bit adder using 4-bit adder blocks (with ripple carry Between blocks).

```
module ALU (x, y, z, sign, zero, carry, Parity
                                  overflow);
    Input [15:0] x, y;
    Output [15:0] z;
    output sign, zero, 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[5:12], c[3]);
endmodule
```

Notes

→ BEHAVIORAL description of a 4-bit Adder:

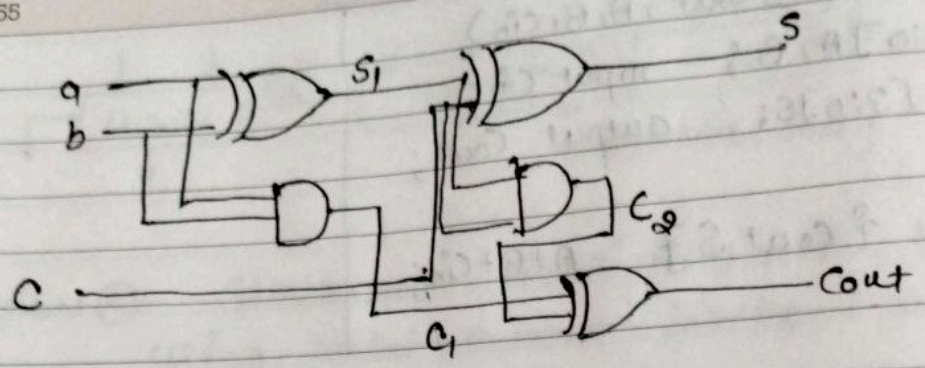
```
module adder4 (S, Cout, A, B, Cin)
    input [3:0] A, B;   input Cin;
    output [3:0] S;     output Cout;

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

VESION → ③ STRUCTURAL MODELING OF Ripple Carry Adder

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

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

endmodule
```

```
module full adder (S, Cout, a, b, c);
  input a, b, c;
  output S, Cout;
  wire S1, C1, C2;
  xor G1(S1, a, b); G2 (S, S1, c),
      G3 (Cout, C2, C1);
  and G4 (C1, a, b), G5 (C2, S1, c);
endmodule
```

Notes



full-adder.

VERSION-4 structural modeling of carry lookahead Adder

```
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);
```

Notes

```
assign S[0] = P0 ^ Cin;
       S[1] = P1 ^ C1,
       S[2] = P2 ^ C2,
       S[3] = P3 ^ C3;
endmodule
```

sum

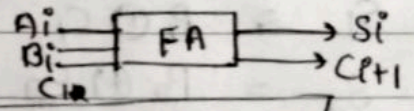
→ How does Carry look-ahead adder work?

- The propagation delay of an n-bit ripple carry adder is proportional to n.
  - due to the rippling effect of carry sequentially from one stage to the next.
- One possible way to speedup the addition.
  - Generate the carry signals for the various stages in parallel.
  - Time complexity reduces from $O(n)$ to $O(1)$.
  - Hardware complexity increase rapidly with n.

- Consider the i-th stage in the addition process.
- We define the carry generate and carry propagate function as;

Carry generate ← $g_i = A_i B_i$

Carry propagate ← $P_i = A_i \oplus B_i$



$$\boxed{C_{i+1} = g_i + P_i C_i}$$

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

Notes

- $P_i = 1$ represents the condition when an input carry C_i will be propogated to the output carry C_{i+1}.

213-153

Unrolling the Recurrence

$$C_{i+1} = g_i + P_i C_i = g_i + P_i(g_{i-1} + P_{i-1} C_{i-1}) \Rightarrow g_i + P_i g_{i-1} + P_i P_{i-1} C_{i-1}$$

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

$$= g_i + P_i g_{i-1} + P_i P_{i-1} g_{i-2} + P_i P_{i-1} P_{i-2} C_{i-2} = \cdots$$

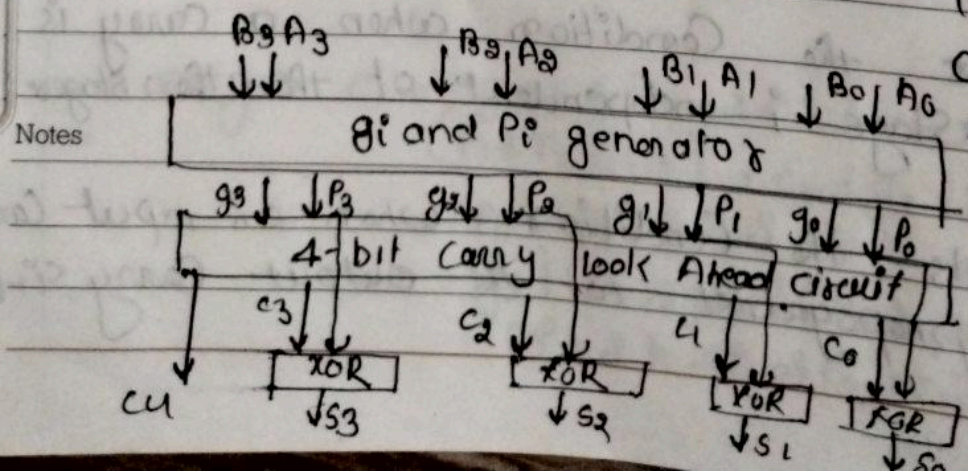
$$\boxed{C_{i+1} = g_i + \sum_{k=0}^{i-1} g_k \prod_{j=k+1}^{i} P_j + C_0 \prod_{j=0}^{i} P_j}$$

Generation of the Carry and Sum bits:

$$C_4 = g_3 + g_2 P_3 + g_1 P_2 P_3 + g_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3$$
$$C_3 = g_2 + g_1 P_2 + g_0 P_1 P_2 + C_0 P_0 P_1 P_2$$
$$C_2 = g_1 + g_0 P_1 + C_0 P_0 P_1$$
$$C_1 = g_0 + C_0 P_0$$

$$S_0 = A_0 \oplus B_0 \oplus C_0 = P_0 \oplus C_0$$
$$S_1 = P_1 \oplus C_1$$
$$S_2 = P_2 \oplus C_2$$
$$S_3 = P_3 \oplus C_3$$

4 AND2 gates
3 AND 3 gates
2 AND 4 gates
1 AND 5 gate
1 OR 2, 1 OR 3, 1 OR 4
and 1 OR 5 gate
4 XOR 2 gates



Notes

Aditiyagi-verilg

214-152

Lecture ⇒ (12) Verilog Description Style

- Two different styles of description:
  - (1) Data Flow — using assignment statement
    - Continuous assignment
  - (2) Behavioral
    - Procedural assignment — using procedural statement
      - Blocking
      - Non-blocking

Similar to a program in high level language.

- Two different styles of description:
  - (1) Data flow style : Continuous Assignment
    - Identified by the keyword "assign".
    - Forms a static binding between:
      - The "net" being assigned on the left-hand side (LHS).
      - The expression on the right-hand side (RHS), which may consist of both "net" and "register" type variables.

```
assign a=b+c;
assign sign=z[15]
```

- The assignment is continuously active:
  - Almost exclusively used to model combinational circuits.
- We shall also see some example of modeling sequential circuit elements.
- Some points to note:
  - A Verilog module can contain any number of "assign" statements.

- Typically, the "assign" statements are followed by procedural descriptions.
- The "assign" statement are used to model behavioral descriptions.

• we shall illustrate various usages of "assign" statement for modeling combinational and also some sequential logic blocks.

```
module
  [ signal declaration
  { assign statements
  { ≡ c statement
end module
```

eg:-

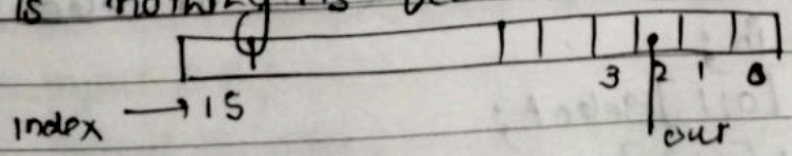
```
module generate_mux(data, sel, out);
  input [15:0] data;
  input [3:0] select;
  output out;
  assign out = data[select];
endmodule
```

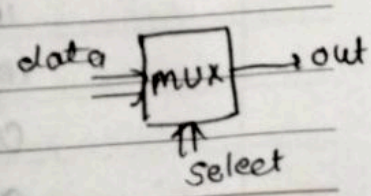
non constant index in expression on RHS generates a mux.

• Point to 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 a constant, just a wire will be generated.

Example: assign out = data[2];

→ data is nothing its vector. vector means collection of bits



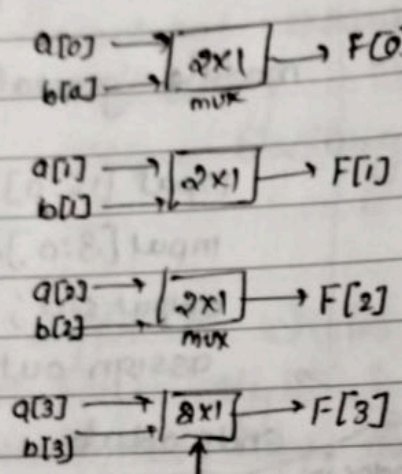


Notes

217-149

```
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
```

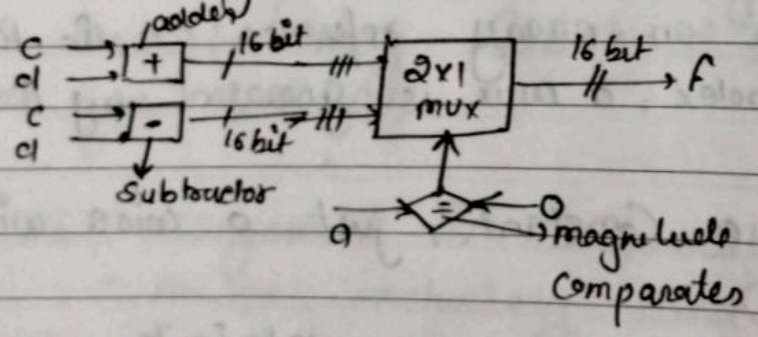
conditional operator generates a mux



⊙ Point to NOTE :-
- whenever a Conditional is encountered in the RHS of an expression, a 2 to 1 mux is generated
- in the previous eg, since the variables "a", "b", and "f" are vectors, an array of 2-to-1 mux-es are generated.

→ what hardware will be generated by the following

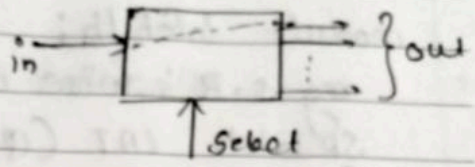
assign f = (a==0) ? (c+d) : (c-d);




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

Notes

Non-Constant index in expression on LHS generates a decoder



- Point to NOTE :-
  - A Constant index in the expression on the LHS will not generate a decoder.
    - → eg : assign out[5] = in;
      This will be generate a wire connection.
  - As a rule of thumb, whenever the Synthesis tool detects a variable index in the LHS, a decoder is generated.

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

Here is an example to describe a Sequential logic element using "assign" statement.

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

generate a D-type latch

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

Notes

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

Simulation output

#0	S=0,	R=1,	Q=0,	Qbar = 1
#5	S=1,	R=1,	Q=0,	Qbar = 1
#10	S=1,	R=0,	Q=1,	Qbar = 0
#15	S=1,	R=1,	Q=1,	Qbar = 0
#20	S=0,	R=0,	Q=1,	Qbar = 1

Verilog lec ⇒ (13) PROCEDURAL ASSIGNMENT

- Two kinds of procedural blocks are supported in Verilog:
  - The "initial" block
    - executed once at the begining of simulation.
    - used only in test bench & cannot used in synthesis.

- The "always" block
  - A continuous loop that never terminates

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

⊙ The "initial" Block

- All statement inside an "initial" statement consistute an "ititial block."
  - Grouped inside a "begin .... end" structure for multiple statements.
  - The statements starts at time 0, and execute only once.
  - If there are multiple "initial" blocks, all the blocks will start to execute concurrently at time 0.
- The "initial" block is typically used to write test benches for simulation:
  - Specifies the stimulus to be applied to the design under test (DUT).
  - Specifies how the DUT outputs are to be displayed / handled.
  - Specifies the files where the waveform information is to be dumped.

```
initial
  begin
  ----
  end
```

} initial block

Notes

```
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;
endmodul
```

- The three "initial" blocks execute concurrently.
- The first block executes at time 0.
- The third block terminates simulation at time 25 units

⊙ Some short cuts in Declarations

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

output reg [7:0] data;

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

→ A variable can be intialized when it is declared.

reg clock = 0;

instead of reg clock; initial clock = 0;

09 Sunday ⇒ The "always" Block

Notes All behavioral statements inside an "always" statement constitute an "always block".

- multiple statements are grouped using "begin --- end"

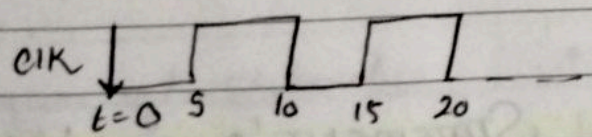
- An "Always" statements starts at time 0 and execute the statements inside the block repeatedly, and never stops.
  - used to model a block of activity that is repeated indefinitly in a digital circuit.
  - for example, a clock signal that is generated continuously.
  - we can specify delay for simulation; however for real circuits; the clock generator will be active as long as there is power supply.

```
always
begin
  ....
end
```

→ always block

```
module generating_clock;
  output reg clk;

initial
  clk = 1'b0; // initialized to 0 at time 0

always
  #5 clk = ~clk; // toggle after time 5 unit

intial
  #500 $finish;
endmodule
```

"initial" and "always" blocks can execute within the same verilog module.
- They all execute concurrently; "initial" only once and "always" repeatedly



Period = 10

Notes
→ A module can contain any number of "always" blocks, all of which execute concurrently.
→ The @(event_expression) part is required for both combinational and sequential circuit description

224-142

* Basic syntax of "always" block:

```
always @(event expression)
begin
    Sequential statement_1;
    Sequential_statement_2;
    ....
    Sequential_statement_n;
end
```

- only "reg" type variable can be assign within an "intial" or "always" block.

→ Basic reason:

→ The sequential "always" block executes only when the event expression triggers

- at other times the block is doing nothing
- an object being assigned to must therefore remember the last value assigned (not continuously driven).
- so, only "reg" type variable can be assign within the "always" block.
- of course, any kind of variable may appear in the event expression (reg, wire, etc).

SEQUENTIAL STATEMENT'S in VERILOG

Notes

- in verilog, one of more sequential statement can be present inside an "initial" or "always" block.

- The statements are executed sequentially.
- multiple assignment statement inside a "begin...end" block may either execute sequentially or concurrently depending upon on the type of assignment.
  - Two type of assignment statement: blocking (a = b+c;) or non-blocking (a<= b+c;).
- The sequential statements are explained next.

ⓐ begin.....end

```
begin
  Sequential_statement_1;
  Sequential_statement_2;
  ....
  Sequential_statement_n;
end
```

- A number of sequential statements can be grouped together using "begin...end".
- If n=1, "begin...end" is not required.

ⓑ if ----else

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

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

```
if (<expression1>)
  Sequential_statement;
else if (<expression2>)
  Sequential_statement;
else default statement;
```

- each sequential statement can be a single statement or a group of statement within "begin...end".

Notes

226-140

© Case

```
case (<expression>)
  expr1 : sequential_statement;
  expr2 :      "          ";
  ------;
  exprn : sequential statement;
  default: default_statement;
endcase
```

- Each sequential_statement can be a single statement or a group of statements within "begin ... end".
- can replace a complex "if ... else" statement for multiway branching.

- The expression is compared to the alternatives (expr1, expr2, etc) in the order they are written.
- if none of the alternative matches the default statement is executed.

⊙ Two variation: "casez" and "casex"

- The "casez" statements treates all "z" values in the case alternative or are cases expression as don't cares.
- The casex statement treats all "x" and "z" values in the case item as don't care

```
reg [3:0] state; integer next_state;
case x (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
```

Notes

Arti-tyagi-verilog

Lecture :- (14) PROCEDURAL ASSIGNMENT

(a) "while" loop

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

- The "while" loop executes until the expression is not true
- The sequential_statement can be a single statement or a group of statements within "begin ... end".

example

```
integer mycount;
initial begin
    mycount=0;
    while (mycount <= 250)
    begin
        $display ("my count : %d", mycount);
        mycount = mycount+1;
    end
endmodule
```

(e) "for" loop

```
for (exp1; exp2; exp3)
    sequential_statement;
```

- The "for" loop consist of three parts:
  - (a) An initial condition (exp1).
  - (b) A check to see if the terminating condition is true (exp2).
  - (c) A procedural assignment to change the value of the control variable (exp3)
- The "for" loops can be conveniently used to initialize an array or memory.

Notes

- The "for" loop executes as long as the expression exp2 is true
- The sequential_statement can be a single statement or a group of statements "begin ---- end"

Example

```
integer mycount;
reg [100:1] data;
integer i;
initial
① ex   for (myCount = 0; myCount <= 255; mycount = myCount+1)
       $display ("my Count: %d", myCount);

ex② initial
       for (i=1; i<=100; i=i+1)
           data [i] = 1'b0;
```

① "repeat" loop

```
repeat (<expressions>)
    Sequential_statement;
```

- The "repeat" Construct executes the loop a fixed number of times
- It cannot be used to loop on a general logical expr like "while".

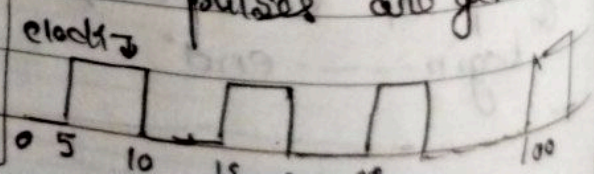
16 Sunday

Notes

example

```
reg clock;
initial begin
    clock = 1'b0;
    repeat (100)
        #5 clock = ~clock;
end
```

exactly 100 clock pulses are generated



(8) "FOREVER" loop

```
forever
    Sequential_statement;
```

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

- The "forever" Construct does not use any expression and executes forever until $finish is encountered the testbench
  - Equivalent a "while" loop for which the expression is always true.
- The Sequential_statement can be a single statement or a group of statement within "begin ... end".

```
//clock generation using "forever" Construct reg clk;
    intial
        begin
            clk = 1'b0;
            forever #5 clk = ~clk;   //clk period of 10 units
        end
```

OTHER CONSTRUCTS - Available

#(time value)	• mask a block suspend for "time-value" units of time. • The time unit can be specified using the 'timescale Command.

Notes

231-135

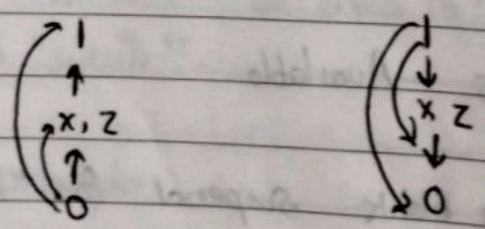
[@(event_expression)]

- makes a Block suspend for "time-value" units of time.
- The time unit can be specified using the 'timescale comm

- makes a Block suspent until "event_expression" triggers.
- Various keywords associated with "event_expression" shall be discussed with examples.

- The event expression specifies the event that is required to resume execution of the procedural block.
- The event can be any one of the following:
  - (a) Change of a single value.
  - (b) Positive or negative edge occuring on signal (Posedge or negedge).
  - (c) List of above-mentioned events, separated by "or" or comma.

- A "Posedge" is any transition from {0, x, z} to 1, and from 0 to {z, x}.



always @ (Posedge x)
{ ...

Notes { for Posedge } { negedge }

Example

```
→ @(in)
→ @(a or b or c)
→ @(a, b, c)
```

// "in" changes
// any of "a", "b" & "c" changes
// -- do --

→ @ (Posedge clk) // Posedge of "clk"
→ @ (Posedge clk or negative reset) // Positive edge of "clk" or negative edge of "reset"
→ @ (*) // any variable changes

Sample // D Flip-flop with Syncronous set and reset

```
module dff (q, qbar, d, set, reset, clk);
input 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
```

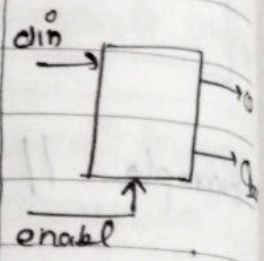
Example // D flip-flop with asynchronous set and reset

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

Notes

// Transparent (means no clock) latch with enable

```
module latch (q, qbar, din, enable);
    input din, enable;
    output reg q;

    assign qbar = ~q;
    always @ (din or enable)
      begin
        if (enable) q = din;
      end
endmodule
```



Latch means level triggers

Lecture :- (15) PROCEDURAL ASSIGNMENT EXAMPLES

// A Combinational logic example ~~module~~ mux21

```
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
```

- An event expression in the "always" block triggers whenever at least on of "in1", "in0" or "s" changes
- The "or" keywords specifies the condition

// A Combinational logic example

```
module mux21 (in1, in0, s, f);
    input in1, in0, s;
    output reg f;

    always @ (in1, in0, s)
        if (s)
            f = in1;
        else
            f = in0;
    endmodule
```

- An alternate way to specify the event condition by using comma instead of "or".
- supported in later version of verilog.

A Combinational logic example

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

- An alternate way to specify the event condition by using a "*", instead of naming the variable
→ "*" is activated whenever any of the variable change.

- A Sequential logic Example

```
module dff_negedge (D, Clock, Q, Qbar);
    input D, clock;
    output reg Q, Qbar;

    always @(negedge clk)
    begin
        Q = D;
        Qbar = ~D;
    end
endmodule
```

- The Keyword "negedge" means at the negative going edge of the specified signal.

23 Sunday • Similarly, we can use "Posedge".

Notes • we can combine various triggering conditions by separating them by commas or "or".

// 4-bit Counter with asynchronous reset

```
module Counter (clk, rst, count);

    input clk, rst;
    output reg [3:0] count;

    always @(Posedge clk or Posedge rst)
      begin
        if(rst)
          count <=0;
        else
          count <= count+1;
      end
    endmodule
```

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

// Another Sequential logic Example

```
module incomp_state_spec (curr_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
```

Notes

Lecture :⇒ (16) INTRODUCTION to Blocking AND Non Blocking

Procedural Assignment :

- Procedural assignment statement can be used to update Variable 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.

→ this is different from Continuous assignment (using "assign") that result in the expression on the RHS to continously drive the "net" type variable on the left.

→ Two types of Procedural assignment statements:

ⓐ Blocking (denoted by "=").
ⓑ Non-blocking (" " "<=")

- The left-hand side of a procedural assignment can be one of:

ⓐ A register type variable ("reg", "integer", "real", or "time")
ⓑ A bit select of these variable (eg Sum[15])
ⓒ A Part select of these variable (eg IR[31:26])
ⓓ A Concatenation of any of the above.

- The right-hand side can be any expression consisting of "net" and "register" type variable that evaluates to a value.
- Procedural assignment statement can only appear within procedural blocks ("initial" or "always").

(i) Blocking Assignment

→ General Syntax

variable_name = [delay_or_event_control] expression;

→ The "=" operator is used to specify blocking assignment

→ Blocking assignment statement are executed in the order they are specified in a procedural block.

- The target of an assignment gets update before the next sequential statement in the procedural block is executed.

- They do not block execution of statements in other procedural blocks.

- This is the recommended style for modeling combinational logic.

• Blocking assignment can also generate sequential circuit elements during synthesis (eg incomplete specification in multi-way branching with "case").

• An example of blocking assignment

```
integer a, b, c;
initial
  begin
    a = 10; b = 2; c = 15;
    a = b+c
    b = a+5
    c = a-b;
  end
```

Notes

```
module blocking_example;
  reg x,y z;
  reg [31:0] A, B;      integer Sum;

  intial begin
    x=1'b0 ; y=1'b0 ; z=1'b1          //at time=0
    Sum= 1;                           //at time=0
    A= 31'b0 ;  B= 31'hababababab;    //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
```

→ Simulation of An Example :-

- for Blocking Assignment :-

Ex

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

* → it means any variable changes.

(ii) NON-BLOCKING ASSIGNMENT

- General Syntax:

  variable_name <= [delay_or_event_control] expression;

- The "<=" operator is used to specify non-blocking assignment.
- Non-blocking assignment statements allow scheduling of assignment without blocking execution of statement that follows within the procedural block.
  - The assignment to the target gets scheduled for the end of the simulation cycle (at the end of the procedural block).
- statement subsequent to the instruction under consideration are not blocked by the assignment.
- allows concurrent procedural assignment, suitable for sequential logic.

- This is the recommended style for modeling sequential logic
  - Several "reg" type variable can be assigned syncronous under the control of a common clock.

```
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
```

Notes

eg → Non-blocking assignment with a clock :-

```
always @ (Posedge clk)
    begin
        a <= b & c;
        b <= a^d;
        c <= a|b;
    end
```

All assignments take place Synchronously at the rising edge of the clock.

mp
- Swaping value 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 vice versa depending on simulator implementation.
- Both registers will get the same value (either "a" or "b").
  - Race Condition

30 Sunday

Notes

-in case of non-Blocking assignment

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

- Here the variable are correctly swapped.
- All RHS variable are read first and assigned to LHS variables at the positive clock edge.

⊙ Trying to swap using blocking assignment

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

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

⊙

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

- Correct swapping will occur, but we need to two temporary variable "ta" and "tb".

⊙

```
module nonblocking_assgn;
integer a, b, c, d;
reg clock
always @ (posedge clock)
begin
```

Notes

245-121

```
    a <= b+c;
    d <= a-8;
    b <= d+10;
    c <= c+1;
  end
```

TEST BENCH

```
  initial
    begin
      $monitor ($time, "a=%d , b=%d , c=%d,
                a=%d", a, b, c, d);

      a=30 ; b=20 ; c=15 ; d=8 ;
      clock = 0;
      forever #5 clock = ~clock;
    end
  initial
      #100 $finish;
  endmodule
```

SOME RULE TO FOLLOWED:-

- It is recommended that blocking and non-blocking assignments are not mixed in the same "always" block.
  - Simulator may allow, but this is not good design practice.
- Verilog synthesized ignores the delay specified in a procedural assignment statement (blocking or nonblocking)
  - may lead to functional mismatch between the design model and the synthesized netlist.
- A variable cannot appear as the target of both a blocking and a nonblocking assignment.
  - This is not permissible → $\boxed{\begin{array}{l} x = x+5 \\ x <= y; \end{array}}$

Notes

Lecture: (17) INTRODUCTION

→ we shall be looking at some examples of modeling using blocking and non-blocking assignments.

→ objective is to get a feel of the type of assignment statement to use for some particular scenario.

→ Avoid some of the "not-so-good" design practices in modeling.

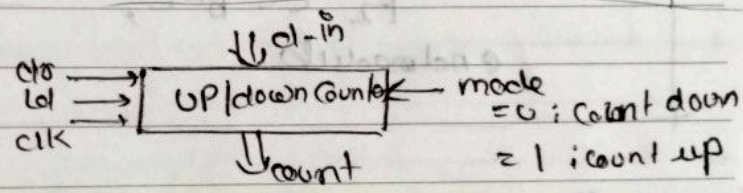
// 8-to-1 multiplexer : Behavioural description

```
module mux_8to1 (in, sel, out);
    input [8:0] in;  input [2:0] sel;
    output reg out;
    always @(*)
    begin
        case (sel)
        3'b000 : out = in[0];
        3'b001 : out = in[1];
        3'b010 : out = in[2];
        3'b011 : out = in[3];
        3'b100 : out = in[4];
        3'b101 : out = in[5];
        3'b110 : out = in[6];
        3'b111 : out = in[7];
        default : out = 1'bx;
        endcase
    end
endmodule
```

// up-down Counter (Synchronous clear)

```
module Counter (mode, clr, ld, din, clk, Cout);
    input mode, clr, ld, clk;
    input [0:7] din;
    output reg [0:7] Count

always @(Posedge clk)
    if (ld)          Count <= din;
    else if (clr)    Count <= 0;
    else if (mode)   Count <= Count+1;
    else             Count <= Count-1;
endmodule
```

load input
count



// 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
```

248-118

→ make a design general for any number of bits.
- using the keyword "Parameter".
- Parameter value are substituted before Simulator or Synthesis.

◉ // using more than one clocks in a module

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

◉ // using multiple edges of the same clock

```
module multi_edge_clk (a, b, f, clk);
    input a, b, clk;
    output reg f; reg t;
    always @ (Posedge clk)
        f <= t & b;
    always @ (Negedge clk)
        t <= a | b;
endmodule
```

Notes

Asti_tyagi_verilog

// A Ring Counter

```
module ring Counter (clk, init, Count);
    input clk, init;
    output reg [7:0] Count;
    always @ (Posedge clk)
    begin
        if (init) Count = 8'b10000000;
        else begin
            Count <= Count <<1;
            Count[0] <= Count [7];
        end
    end
endmodule.
```

Lecture :⇒ (18) Blocking Vs. Non-blocking Assignment

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

- The value of "b" will be assigned to "c" 10 time unit after the "begin..end" block starts.

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

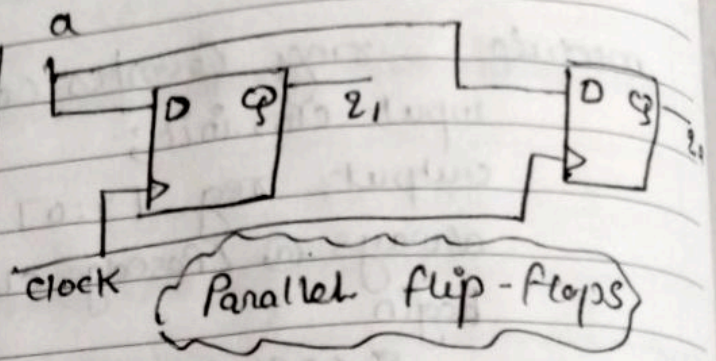
- "a" is scheduled to get the value of "b" 5 time units into the future.
- "c" is also scheduled to get the value of "a" 5 time units into the future.

Value of "c" will be different for the two cases.

251-115

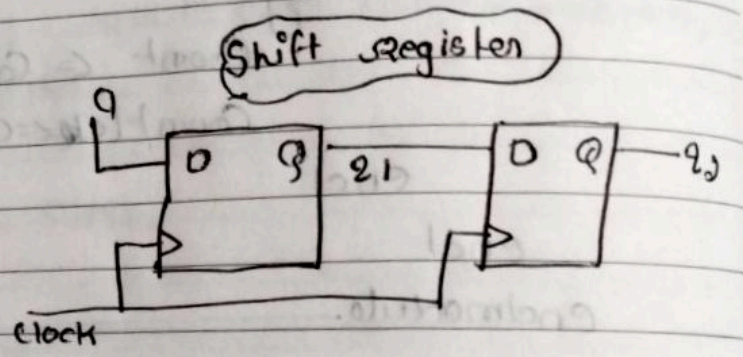
- Blocking Example

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



Example :-

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

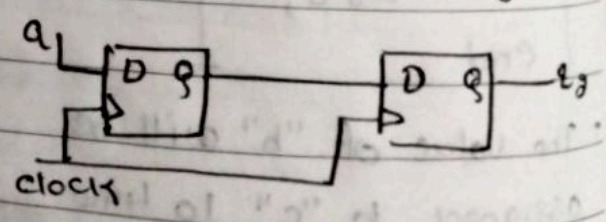


- if we simply interchange them it will become a shift register

eg →

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

→ here is also generating the same shift Register



Notes eg :-

Lecture :→ (19) Blocking and Nonblocking Assignment Feator

Basic Idea

- It is Posible to Combine both blocking and non-blocking assignment in the same procedural block ("always").
  - Simulator or synthesis tool support this type of usage
- However, interpretation of the circuit behavior under mixed usage is not very straighforward
  - Not recommended for designers.
  - we shall explain the semantics using two simple examples.
- Such mixing should be avoid in a design.

Example

```
always @(*)
  begin
    x = 10;
    x = 20;
    y = x;
    #10;
  end
```

```
always @(*)
  begin
    x = 10;
    x = 20;
    y <= x;
    #10;
  end
```

→ Same result will be shown for both the version
- "x" will be assigned 10 and then 20 both at time 0.
- The value of "x" at time 0 will be assigned to y. [x = 20 ; y = 20]

253-113

GENERATE Blocks

- "generate" (Keyword) statement allows verilog code to be generated dynamically before the simulation or synthesis begins.
  - Very convenient to create parameterized module descriptions.
  - Example: N-bit carry adder for Arbitrary value of N.

- Requires the keywords "generate" and "endgenerate"
- Generate instantiations can be carried out for various verilog blocks:
  - modules, user-defined primitives, gates, continuous assignments, "initial" and "always" blocks etc.
  - Generate instances have unique identifier names and can be referenced hierarchically.

- Special "genvar" variables:-
  - The keyword "genvar" can be used to declare variable that are used in the evaluation of generate block.
  - These variable do not exist during simulation or synthesis.
  - The value of a "genvar" can be defined only in a generate loop.
  - Every generate loop is assigned a name so that variable inside the generate loop can be referenced hierarchically.

Notes

example

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

  generate for (P=0; P<N; P=P+1)
      begin xorlp                          ∴ lp → label
        XOR  xg (f[P], a[P], b[P]);
        end
      endgenerate
    endmodule
```

TEST BENCH :-

```
module generate_Tb;
  reg [15:0] x, y;
  wire [15:0] out;

  xor_bitwise G(.P(out), .a(x), b(y));
  initial begin
    $monitor ("x: %b , y: %b , out: %b",
                x, y, out);
      x = 16'h0000 ; y = 16'h00ff;
      #10  x = 16'h0f0f ; y = 16'h3333;
      #20  $finish;
      end
      end module
```

Notes

- In the bitwise XOR example, the name "XORIP" was given to the generate loop.
- The relative hierarchical names of the XOR gates will be
  xorip[0].xg XORIP[1].xg, ---, xorip[15].xg

eg② DESIGN OF N-bit Ripple CARRY Adder

// structural gate-level description of a full adder

```
module full_adder (a,b,c,Sum,Cout);
    input a,b,c;
    output, sum, Cout;
    wire t1, t2, t3;
    XOR G1(t1, a,b), G2(Sum, t1, c);
    And G3(t2,a,b), G4(t3, t1, c);
      Or G5(Cout, t2,t3);
endmodule
```

Notes

Arti-Iyagi_Verilog

Verilog Lecture: 20 USER - DEFINED PRIMITIVES

USER DEFINED PRIMITIVES (UDP):-

- They are used to define Custom Verilog primitieves by the use of lookup tables.
- They Can Specify:
  - Truth Table for Combinational functions.
  - State-tables for Sequential functions.
  - Don't Care, rising and falling edges, etc. can be specified

for combinational <input 1> <input 2> ---- <input N> : <output>;

→ for Sequential functions, state table entries are specified.
<input 1> <input 2> --- input<N> : <present_state> : <Next state>

Some RULES FOR USING UDP's :-

- The Input terminals to a UDP can only be scalar variables.
→ multiple input terminals can be used.
→ The Input terminals are declared as "input".
→ Input entries in the table must be in the same order as the "input" terminal list.

⊙ only one scalar output terminal must be used.
→ The output terminal must appear in the beginning of the terminal list.
→ for Combinational UDP's, the output terminal is declared as "output".
→ for Sequential UDP's the output terminal is declared as "reg"

- For Sequential UDP's, the state can be initialized with an "initial" statement.
  - This is optional.

Some Guide lines :-

- user defined Primitives (UDP's) model functionality only.
  - They do not model timing or process technol
- A functional block can be modeled as a UDP on if it has exactly one output.
  - if a block has more than one output it has to be modeled as a module.
  - as an alternatives, multiple UDP's can be used, one per unit.
- Inside the simulator, a UDP is typically implemented as a lookup table in memory.
- The UDP state tables should be specified as completely as possible.
  - for an specified case, the output is set to "x".

Modeling Combinational Circuits :-

Notes

```
// 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
```

```
// full adder Carry generation
// using don't Care ("?")
Primitive udp_cy (cout, a, b, c);
    input a, b, c;
    output cout;
    tables
    // a   b   c   :  cout
       0   0   ?   :  0 ;
       0   ?   0   :  0 ;
       ?   0   0   :  0 ;
       1   1   ?   :  1 ;
       1   ?   1   :  1 ;
       1   1   ?   :  1 ;
    endtable
endprimitive
```

```
// instantiating UDP's
// A full adder description
module fullAdder (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.

```
// A 4-Input AND Function
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;
endprimitive;
```

MODELING 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   // latch
       ?   ?     1     :   ?   :   0       // force
       0   1     0     :   ?   :   0       // table
       1   1     1     :   ?   :   1       // return
endprimitive ?   0     0     :   ?   :   -       // state
```

Notes

// A T Flip-Flop

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

// Constructing a 6 bit ripple Counter using T Flip-Flops.

```
primitive 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 (Count[2], Count[1], clr);
    TFF F3 (Count[3], Count[2], clr);
    TFF F4 (Count[4], Count[3], clr);
    TFF F5 (Count[5], Count[4], clr);
endprimitive
```

Notes

// A negative edge Sensitive JK Flip-flops

```
primitive JKFF (q, J, K, clk, clr);
  output reg q;
  table
    // J   K   clk   clr   :  q   :  q_now
       ?   ?   ?     1     :  ?   :  0   ;   // clear
       ?   ?   ?     (10)  :  ?   :  -   ;   // ignore --- no ch
       ?   ?   ?     (10)  :  ?   :  -   ;   // no change
       0   0   (10)  0     :  ?   :  0   ;   // reset condition
       0   1   (10)  0     :  ?   :  1   ;   // set condition
       1   0   (10)  0     :  ?   :  1   ;   // toggle condition
       1   1   (10)  0     :  0   :  1   ;   // toggle condition
       1   1   (10)  0     :  1   :  0   ;   // toggle condition
       ?   ?   (01)  0     :  ?   :  -   ;   // no change
```

⊙ Some Rule To Follow

→ The "?" Symbol cannot be specified in an output field

→ The "_" Symbol, indicating no change in the state values, can be specified only in an output field.

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

- The shortcut "f" indicating falling edge, can be used instead of (10).

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

Notes

- Verilog is used to describe the low-level hardw