Verilog Tutorial

ELL201: Digital Electronics

Components of Verilog Code

1. Introduction
2. Module and End Module Declaration
3. Parameter Declaration
4. Port Declaration
5. Wire/Reg Declaration
6. Module Instantiations
7. Logic - Combinational, Sequential
8. Basic Testbench
9. CPLD : Overview
10. CPLD : Programming Overview

Introduction

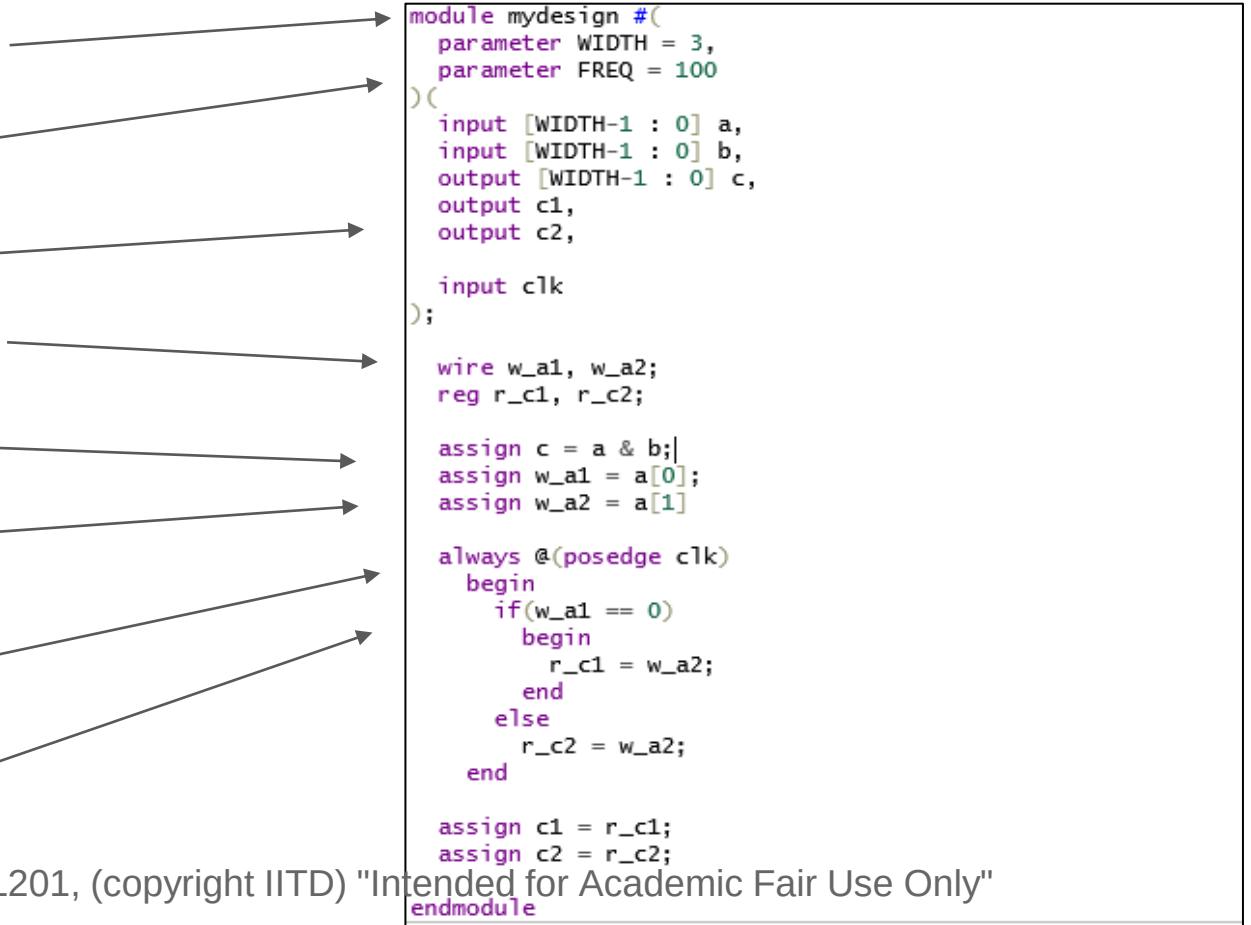
Verilog is a **Hardware Description Language :** Any Digital Design can be described using Verilog.

Use : For Describing Huge Designs (Even Microprocessors!).

A verilog file is given to EDA(Electronic Design Automation) tool to program programmable Logic Devices like Complex Programmable Logic Devices (CPLDs) or Field Programmable Gate Arrays (FPGAs).

General Verilog Code Components

1. Module Declaration
2. Parameter Declaration
3. Port Declaration
4. Wire/ Reg Declaration
5. Assignment Statements
6. Combinational Logic
7. Always Block
8. If Statements

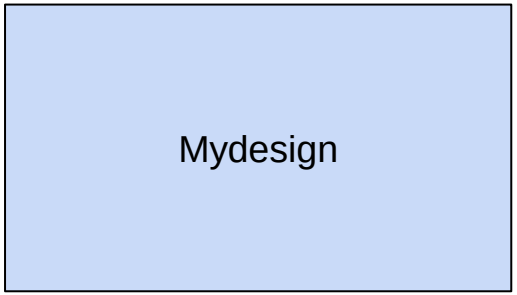


```
module mydesign #(
  parameter WIDTH = 3,
  parameter FREQ = 100
)(
  input [WIDTH-1 : 0] a,
  input [WIDTH-1 : 0] b,
  output [WIDTH-1 : 0] c,
  output c1,
  output c2,

  input clk
);

  wire w_a1, w_a2;
  reg r_c1, r_c2;

  assign c = a & b;
  assign w_a1 = a[0];
  assign w_a2 = a[1]

  always @(posedge clk)
    begin
      if(w_a1 == 0)
        begin
          r_c1 = w_a2;
        end
      else
        r_c2 = w_a2;
    end

  assign c1 = r_c1;
  assign c2 = r_c2;
endmodule
```

Module Declaration

```
module mydesign (
// Port Declarations
);
 // Module Body
end module
```

Design Identifier

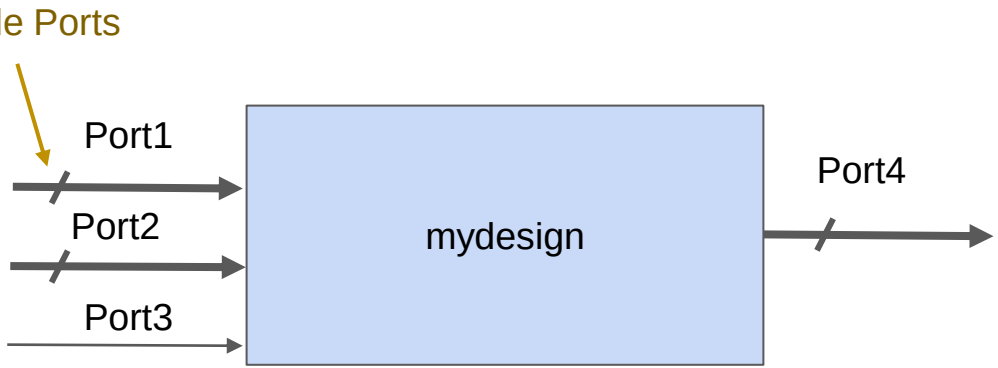


- Marks the beginning and end of the module

Port Declarations

```
module mydesign (
input   [1 : 0] Port1,
input   [1 : 0] Port2,
input   Port3,
output [1 : 0] Port4 );
```

Port Direction Port Width Port Name

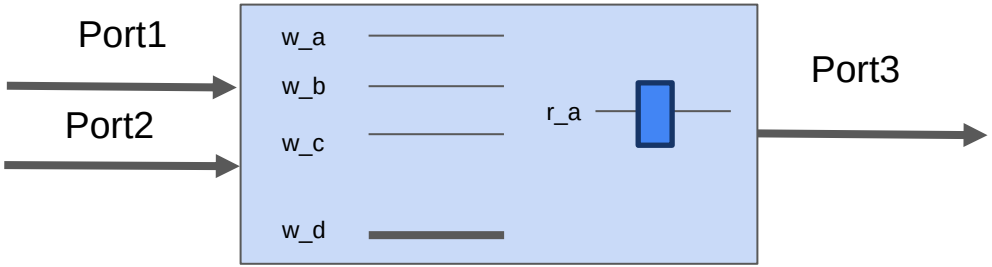


- Above code declares the input output and inout ports of the module.
- Parameter used to declare the width of the ports.

Wire and Reg Declaration

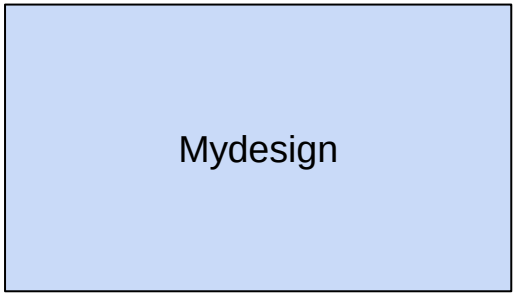
```
module module_name(
   input [1 : 0] port1,
   input [1 : 0] port2,
   output [1 : 0] port3 );
   wire w_a, w_b, w_c;
   wire [1:0] w_d;
   reg r_a;
endmodule
```

wire/ reg WIDTH Name



Parameter Declaration

```
module mydesign #(
parameter parameter_name = value
) (
// Port declaration
);
endmodule
```

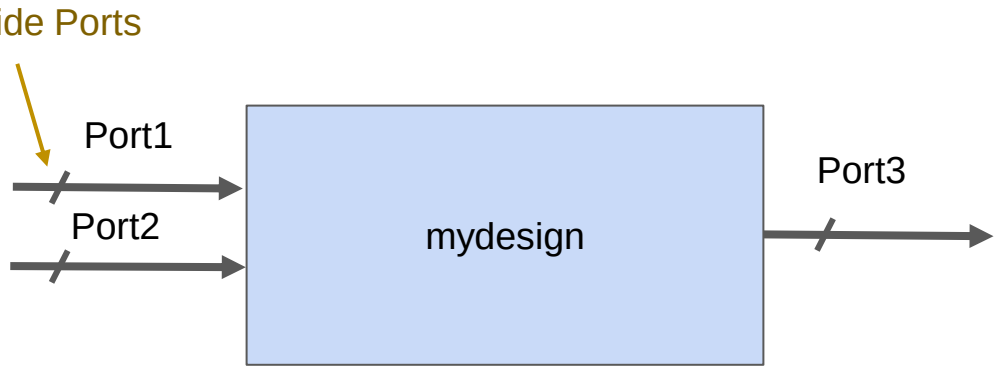


- Parameters are defined for using constants throughout the design.
- Scalability becomes easier when verilog codes are parameterised

Port Declarations with Parameter declaration

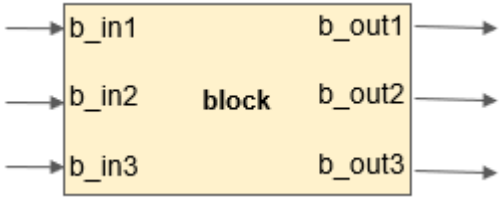
```
module mydesign #(
parameter WIDTH = 32
)(
Input   [WIDTH-1 : 0] port1,
input   [WIDTH-1 : 0] port2,
output [WIDTH-1 : 0] port3 );
```

Port Direction Port Width Port Name

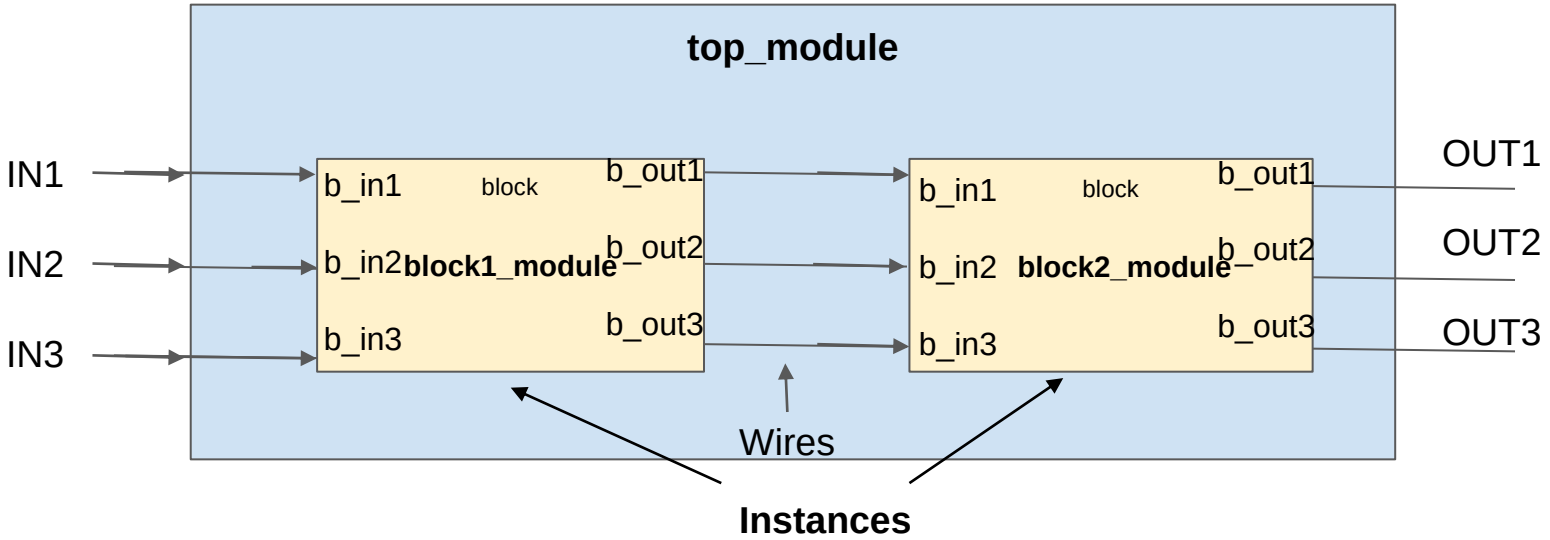


- Use symbol # (hash) after module name while declaring the parameters

Module Instantiations



- Various designs follow hierarchies, with top level (Wrapper) module instantiating other blocks of design, integrated together in the wrapper
- The wrapper may also contain other elements of verilog code.



Module Instantiation Example

```
module top_module(
    input IN1,
    input IN2,
    input IN3,

    output OUT1,
    output OUT2,
    output OUT3);

    wire W_1, W_2, W_3;

    block block1_MODULE (
        .b_in1(IN1),
        .b_in2(IN2),
        .b_in3(IN3),
        .b_out1(W_1),
        .b_out2(W_2),
        .b_out3(W_3)
    );
```

Block Instantiation

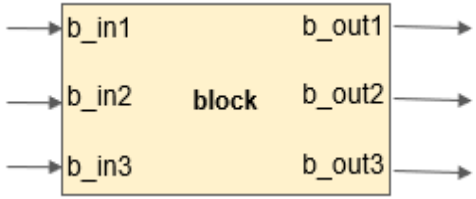
Block identifier

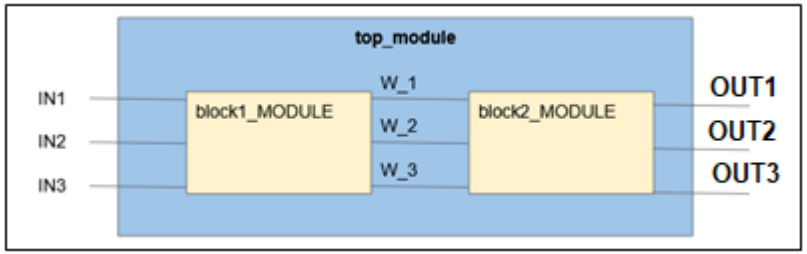
Block instance name/identifier

Block ports

Top Module Wires/Ports

```
    block block2_MODULE (
        .b_in1(W_1),
        .b_in2(W_2),
        .b_in3(W_3),
        .b_out1(OUT1),
        .b_out2(OUT2),
        .b_out3(OUT3)
    );


endmodule
```





Port connection - Name

```
module design_top;
    wire [1:0]  a;
    wire        b, c;

    mydesign d0  ( .x (a[0]),    // signal "x" in mydesign should be connected to "a[0]"
                   .y (b),       // signal "y" in mydesign should be connected to "b" in
                   .z (a[1]),
                   .o (c));
endmodule
```

- A better way to connect ports is by explicitly linking ports on both sides using the respective ***port name.***
- The dot **.** represents that the port name following the dot belongs to the design.
- The signal name in the module to which the port has to be connected is specified within parentheses.

As the connections are made by name, the order in which they are specified is not important.

Port connection - Ordered list

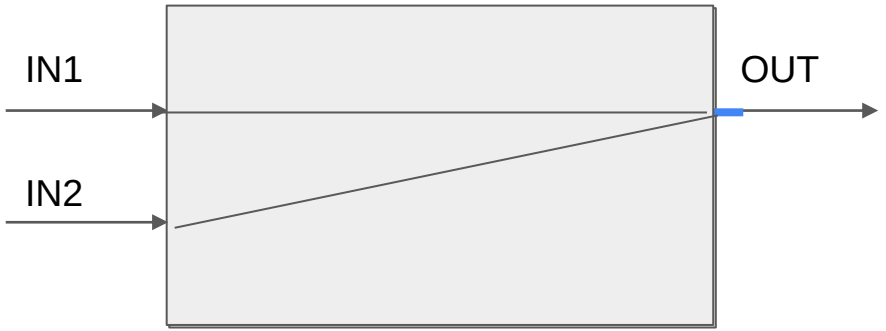
```
module mydesign ( input  x, y, z,     // x is at position 1, y at 2, x at 3 and
                  output o);          // o is at position 4

endmodule

module tb_top;
    wire [1:0]  a;
    wire        b, c;

    mydesign d0  (a[0], b, a[1], c);  // a[0] is at position 1 so it is automatically connected to x
                                      // b is at position 2 so it is automatically connected to y
                                      // a[1] is at position 3 so it is connected to z
                                      // c is at position 4, and hence connection is with o
endmodule
```

- The ports are connected to the design module based on their position in the parameters for module instantiation.
- This port connection style requires us to remember the exact order of ports in the design module (Cumbersome, prone to mistakes)

Combinational Logic : i) Continuous Assignment

Continuous Assignment Statements are used to form connections between two wires / reg/ input/ output entities. Inputs cannot be on the LHS of assign statements.

```
module top_module(
    input IN1,
    input IN2,
    output OUT );

    wire W1;

    assign OUT = IN1;
    assign OUT = IN2;

endmodule
```



Continuous Assignment Statements

1. **Assigning Constant Values**

```
module constant_assignment (
output a,
output [3:0] b  );
assign a = 1'b0;
assign b = 4'b1010;   // 4'b1010 = 4'hA
endmodule
```

Continuous Assignment Statements

2. Implementing Bitwise Operation Using Assign Statements -

assign out = a ___ b; // Bitwise operation between “a” and “b” and assigns the result to “out”.

Logical Operation	Operator
and	&
or	|
not	!
nand	~&
nor	~|
xor	^

Combinational Logic : ii) Behavioural vs Structural

- Behavioural Description is highest level of abstraction provided by verilog.
- Function is described logically.
- Behavioural Description of a module using AND Gate as one of its components -

```
module top_module(
    input a,
    input b,
    output out );

    assign out = a & b;

endmodule
```

- Structural Description uses Gate/ Module Instantiation for logic description.
- Structural Description of module using AND Gate as one of its components -

```
module top_module(
    input a,
    input b,
    output out );

    and and_gate (out, a, b);

endmodule
```

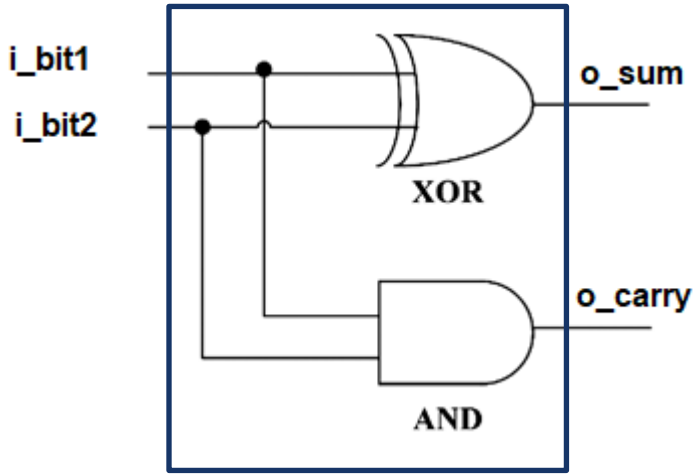
Gate level modelling

```
module gates (      input a, b,
                                        output c, d, e);
          and (c, a, b);       // c is the output, a and b are inputs
          or  (d, a, b);       // d is the output, a and b are inputs
          xor (e, a, b);       // e is the output, a and b are inputs
     nand (c, a, b);           // c is the output, a and b are inputs
          nor  (d, a, b);                // d is the output, a and b are inputs
          xnor (e, a, b);      // e is the output, a and b are inputs

endmodule

module gates (      input a,
                                        output c, d);
  buf (c, a);                  // c is the output, a is input
  not (d, a);                  // d is the output, a is input
endmodule
```

Note: No need to create modules for these logic gates, just instantiate the in-built modules.

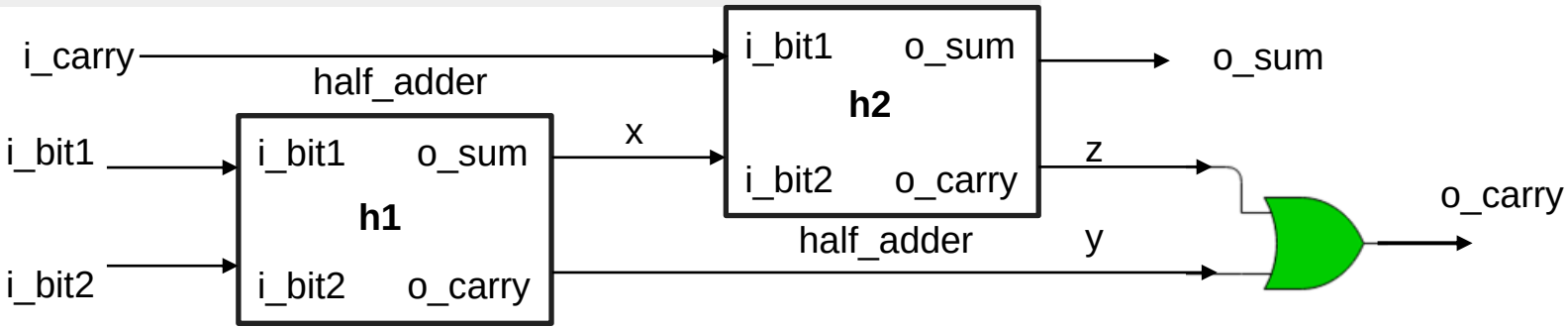
Example: Half Adder Implementation in Verilog

```
module half_adder
  (
   i_bit1,
   i_bit2,
   o_sum,
   o_carry
   );

  input  i_bit1;
  input  i_bit2;
  output o_sum;
  output o_carry;

  assign o_sum   = i_bit1 ^ i_bit2;  // bitwise xor
  assign o_carry = i_bit1 & i_bit2;  // bitwise and

endmodule // half_adder
```



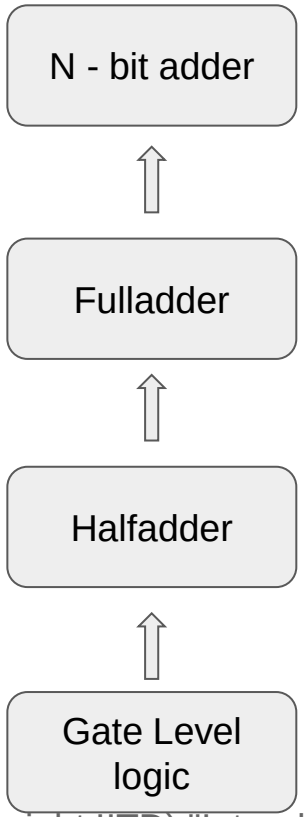
Example : Full Adder Implementation in Verilog

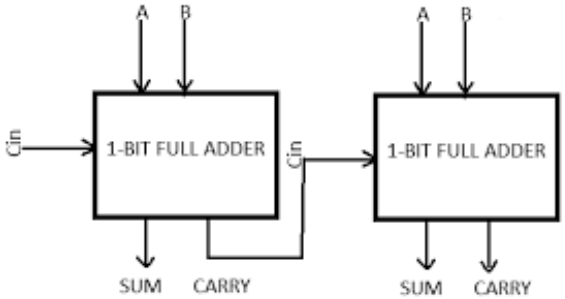
```
module full_adder(i_bit1, i_bit2, i_carry, o_sum, o_carry);
        input i_bit1, i_bit2, i_carry;
        output o_sum, o_carry;
        wire x, y, z;
half_adder  h1(.i_bit1(i_bit1), .i_bit2(i_bit2), .o_sum(x), .o_carry(y));
half_adder  h2(.i_bit1(x), .i_bit2(cin), .o_sum(sum), .o_carry(z));
or or1(cout,z,y);

endmodule
```



Example : 2-Bit Adder Implementation using full adders

```
`include "full_adder.v"
module ripple_carry_adder_2_FA
  (
   input [1:0]  i_add_term1,
   input [1:0]  i_add_term2,
   output [2:0] o_result
   );
  wire [2:0]    w_CARRY;
  wire [1:0]    w_SUM;

  // No carry input on first full adder
  assign w_CARRY[0] = 1'b0;

  full_adder full_adder_1
    (
     .i_bit1(i_add_term1[0]),
     .i_bit2(i_add_term2[0]),
     .i_carry(w_CARRY[0]),
     .o_sum(w_SUM[0]),
     .o_carry(w_CARRY[1])
     );

  full_adder full_adder_2
    (
     .i_bit1(i_add_term1[1]),
     .i_bit2(i_add_term2[1]),
     .i_carry(w_CARRY[1]),
     .o_sum(w_SUM[1]),
     .o_carry(w_CARRY[2])
     );
  assign o_result = {w_CARRY[2], w_SUM};   // Verilog
Concatenation
endmodule // ripple_carry_adder_2_FA
```

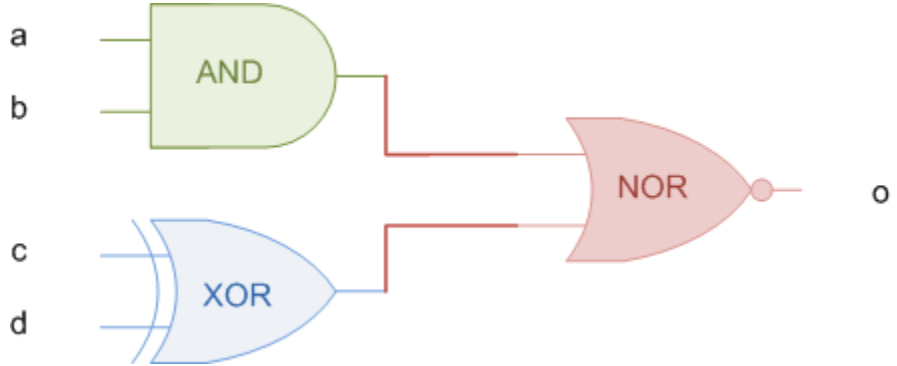




2-bit full adder

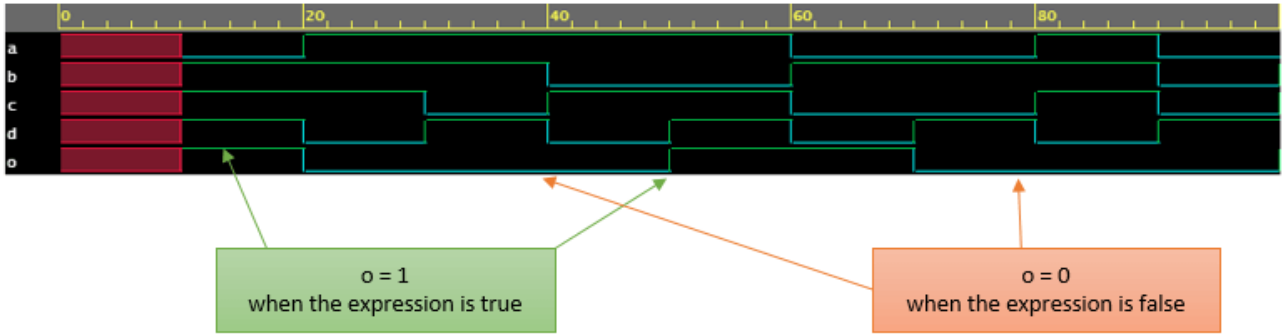
Combinational Logic : iii) Always Block

```
module combo (     input     a,
                   input     b,
                   input     c,
                   input     d,
                  output reg o);
  always @ (a or b or c or d) begin
    o <= ~((a & b) | (c^d));
  end
endmodule
```



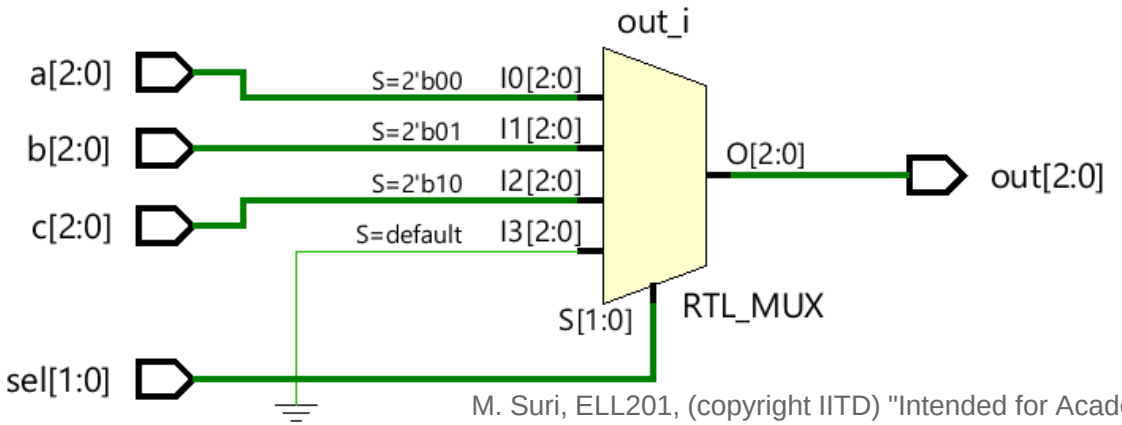
Always block executes whenever any of the signals in the sensitivity list changes its value.

Statements inside always block are executed sequentially.



Combinational Logic : iv) Case statement

```
module my_mux (input       [2:0] a, b, c,          // Three 3-bit inputs
               input       [1:0] sel,              // 2-bit select signal
               output reg  [2:0] out);             // Output 3-bit signal

  // This always block is executed whenever a, b, c or sel changes in value
  always @ (a, b, c, sel) begin
    case(sel)
      2'b00    : out = a;                 // If sel=0, output is a
      2'b01    : out = b;                 // If sel=1, output is b
      2'b10    : out = c;                 // If sel=2, output is c
      default  : out = 0;                 // If sel is anything else, out is always 0
    endcase
  end
endmodule
```



Sequential Logic

- Wherever it is interpreted that the value needs to be stored (when waiting for clock, or when incomplete cases or if conditions are defined), sequential logic gets implemented.

- If sensitivity list of always block involves all inputs used within the block, the always block implements combinational logic.

```
module top_module(
    input clk,
    input a,
    input b,
    output wire out_assign,
    output reg out_always_comb,
    output reg out_always_ff   );

    assign out_assign = a ^ b;
    always @(*)
        begin
            out_always_comb = a ^ b;
        end
```

WIRE

REG

Flip Flop

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

endmodule
```

More on Assignment statements : Continuous and Procedural

Verilog has three types of assignments:

1. Continuous assignment

```
assign muxout = (sel&in1) | (~sel&in0);
assign muxout = sel ? in1 : in0;
```

2. Blocking procedural assignment " = "

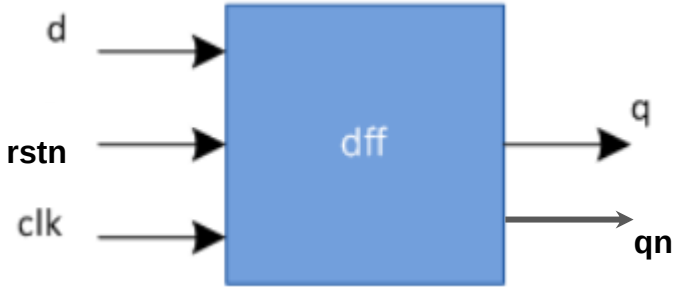
```
// assume initially a=1;
a = 2;
b = a;
// a=2; b=2;
```

3. Non-blocking procedural assignment "=>"

```
// assume initially a=1;
a <= 2;
b <= a;
// a=2; b=1;
```

Flip-flop example

```
module dff (   input d,
               input clk,
               input rstn,
               output reg q,
               output qn);
   always @ (posedge clk or negedge rstn)
      if (!rstn)
         q <= 0;
      else
         q <= d;

   assign qn = ~q;
endmodule
```



3 input ports and 1 output port

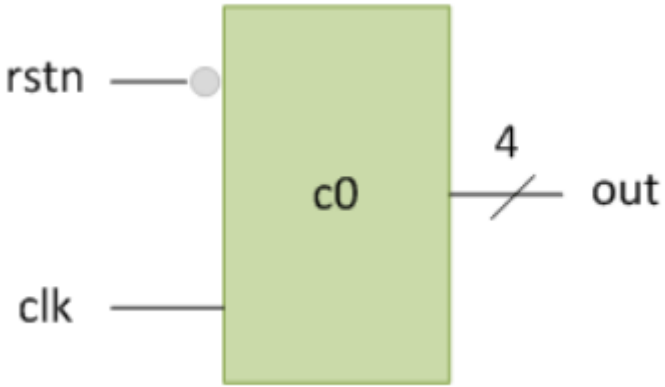
Module is active only at positive edge of CLK.
At positive edge, if rstn (active-low reset) is 0, then reset the flip-flop, i.e. q = 0.
Else, q gets the value of d.

Counter example

Let us try to build a 4-bit counter.

- The 4-bit counter starts incrementing from 4’h0000 to 4’h1111 and then rolls over back to 4’h0000.
- It keeps counting as long as it is provided with a running clock and resetn (active low) is held high.

Block diagram/ schematic view of 4-bit counter




```
0000
0001
0010
...
1110
1111
          rolls over
0000
0001
...
```

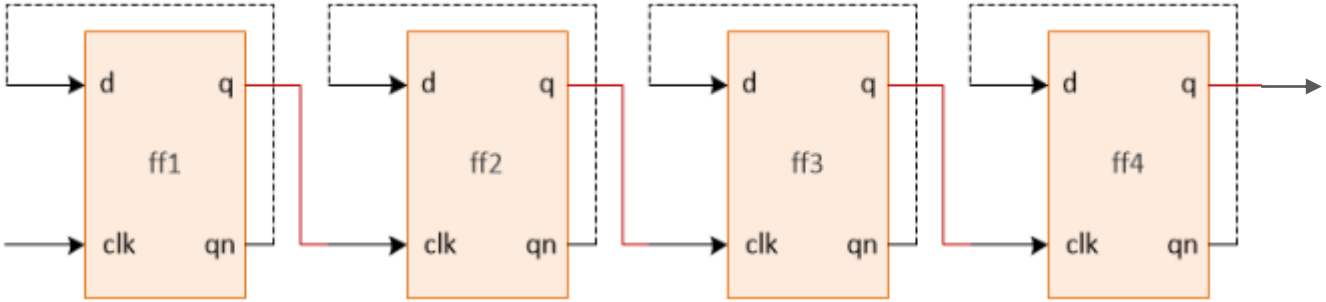
Counter example

```
module counter (  input clk,              // Declare input port for clock to allow counter to count up
                  input rstn,             // Declare input port for reset to allow the counter to be re
                  output reg[3:0] out);   // Declare 4-bit output port to get the counter values

  // This always block will be triggered at the rising edge of clk (0->1)
  // Once inside this block, it checks if the reset is 0, if yes then change out to zero
  // If reset is 1, then design should be allowed to count up, so increment counter
  always @ (posedge clk) begin
    if (! rstn)
      out <= 0;
    else
      out <= out + 1;
  end
endmodule
```

This counter is implemented using behavioral level of abstraction.
We do not know exactly how this counter will be realized in hardware (what type of flip-flops used, how they are connected etc.)

Counter example - Using D-flip-flops

Now, we will build a 4-bit down counter at a slightly lower level of abstraction, using D flip-flops.




```
module ripple ( input clk,
                input rstn,
                output [3:0] out);
   wire  q0;
   wire  qn0;
   wire  q1;
   wire  qn1;
   wire  q2;
   wire  qn2;
   wire  q3;
   wire  qn3;
```

```
dff   dff0 ( .d (qn0),
             .clk (clk),
             .rstn (rstn),
             .q (q0),
             .qn (qn0));

dff   dff1 ( .d (qn1),
             .clk (q0),
             .rstn (rstn),
             .q (q1),
             .qn (qn1));
```

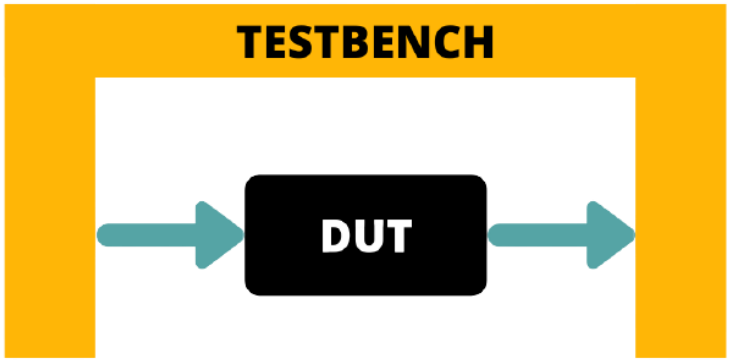
```
   dff   dff2 ( .d (qn2),
                .clk (q1),
                .rstn (rstn),
                .q (q2),
                .qn (qn2));

   dff   dff3 ( .d (qn3),
                .clk (q2),
                .rstn (rstn),
                .q (q3),
                .qn (qn3));

   assign out = {qn3, qn2, qn1, qn0};

endmodule
```

For loop in verilog

```
module lshift_reg (input clk,                              // Clock inputClock input
                   input rstn,                             // Active low reset input
                   input [7:0]load_val,     // Load value
                   input load_en,                          // Load enablenable
                   output reg [7:0] op);    // Output register value
            integer i;dge of clock, if reset is low set output to 0
            // At posedge of clock, if reset is low set output to 0
            // If reset is high, load new value to op if load_en=1eft
            // If reset is high, and load_en=0 shift register to left
            always @ (posedge clk) begin
               if (!rstn) begin
                 op <= 0;egin
               end else begind_en) begin
                      // If load_en is 1, load the value to op
                      // else keep shifting for every clock
                      if (load_en) begin
                      op <= load_val;
                 end else begin1];
            for (i = 0; i < 8; i = i + 1) begin
              op[i+1] <= op[i];3];
            end    op[5] <= op[4];
            op[0] <= op[7]; op[5];
                 endp[7] <= op[6];
               endnd
             endnd
endmodule     end
endmodule
```

Testbench

- To test the functionality of the design.
- Used to provide inputs to the design and monitor outputs

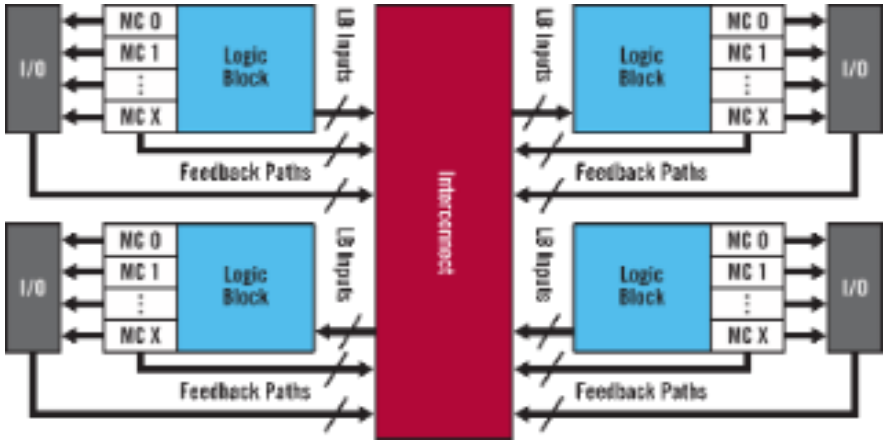


Testbench Example : Half Adder

```
`timescale 1 ns/10 ps  // time-unit = 1 ns, precision = 10 ps

module half_adder_tb;

    reg a, b;
    wire sum, carry;



    half_adder DUT (.a(a), .b(b), .sum(sum), .carry(carry));

    initial // initial block executes only once
        begin
            // values for a and b
            a = 0;
            b = 0;
            #20;

            a = 0;
            b = 1;
            #20;
            a = 1;
            b = 0;
            #20;

            a = 1;
            b = 1;
            #20;
        end
endmodule
```

CPLD : Overview

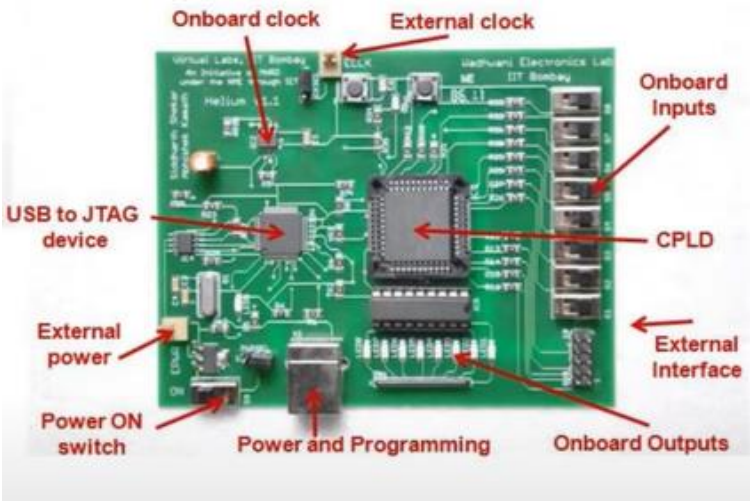
- Complex Programmable Logic Device
- Consists of And/OR logic and Logic Blocks (macrocells), Interconnected within.
- Can Implement Combinational and Sequential Circuits.
- Can be programmed using an EDA tool.
- Has onboard clock and external clock for sequential logic.



Ref : https://www.xilinx.com/products/silicon-devices/cpld/cpld.html

CPLD Programming : Overview

1. In Altera Quartus → Create New Project
2. Create New Verilog File and Run the code
3. Pin Planning : Map Inputs, Outputs, Clock
4. Convert Program to Support Vector Format
5. Program CPLD Device using USB to JTAG shell.



Thank you !