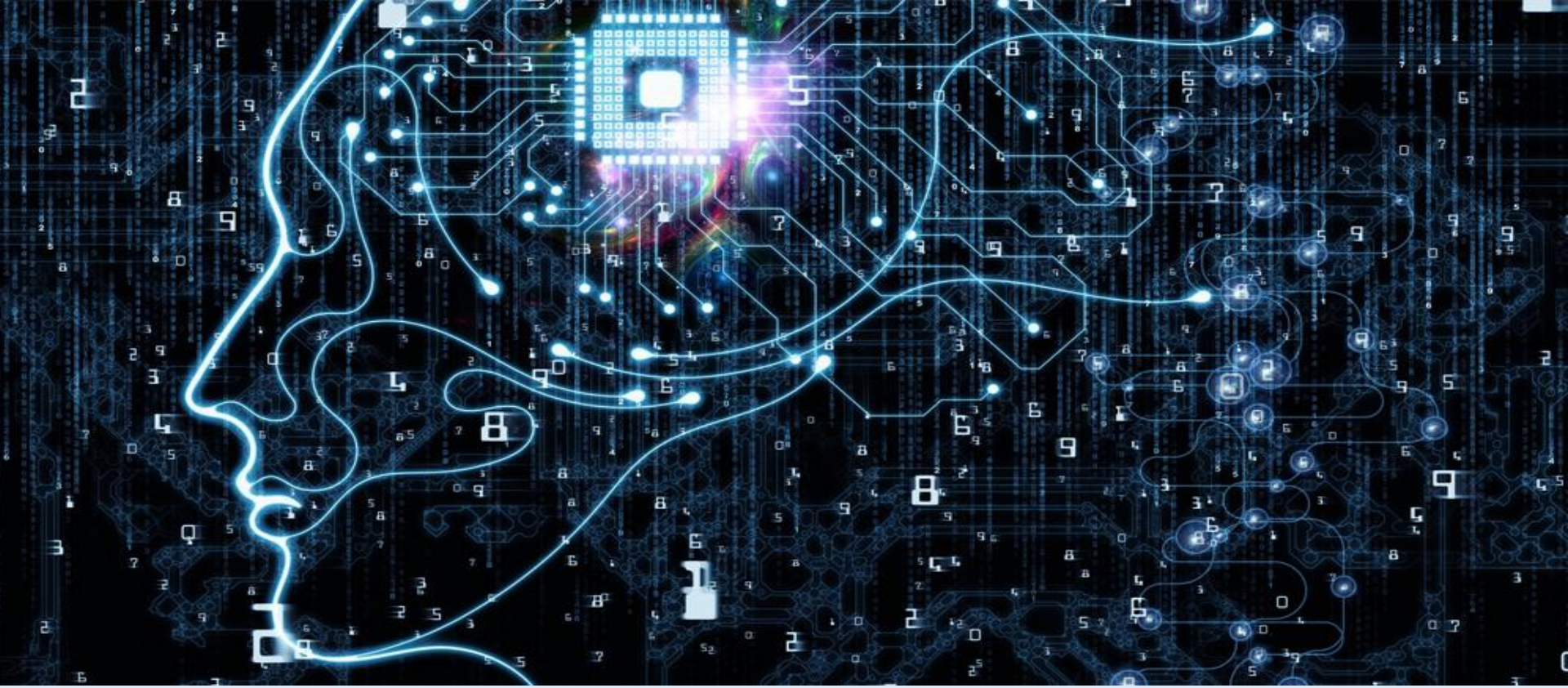


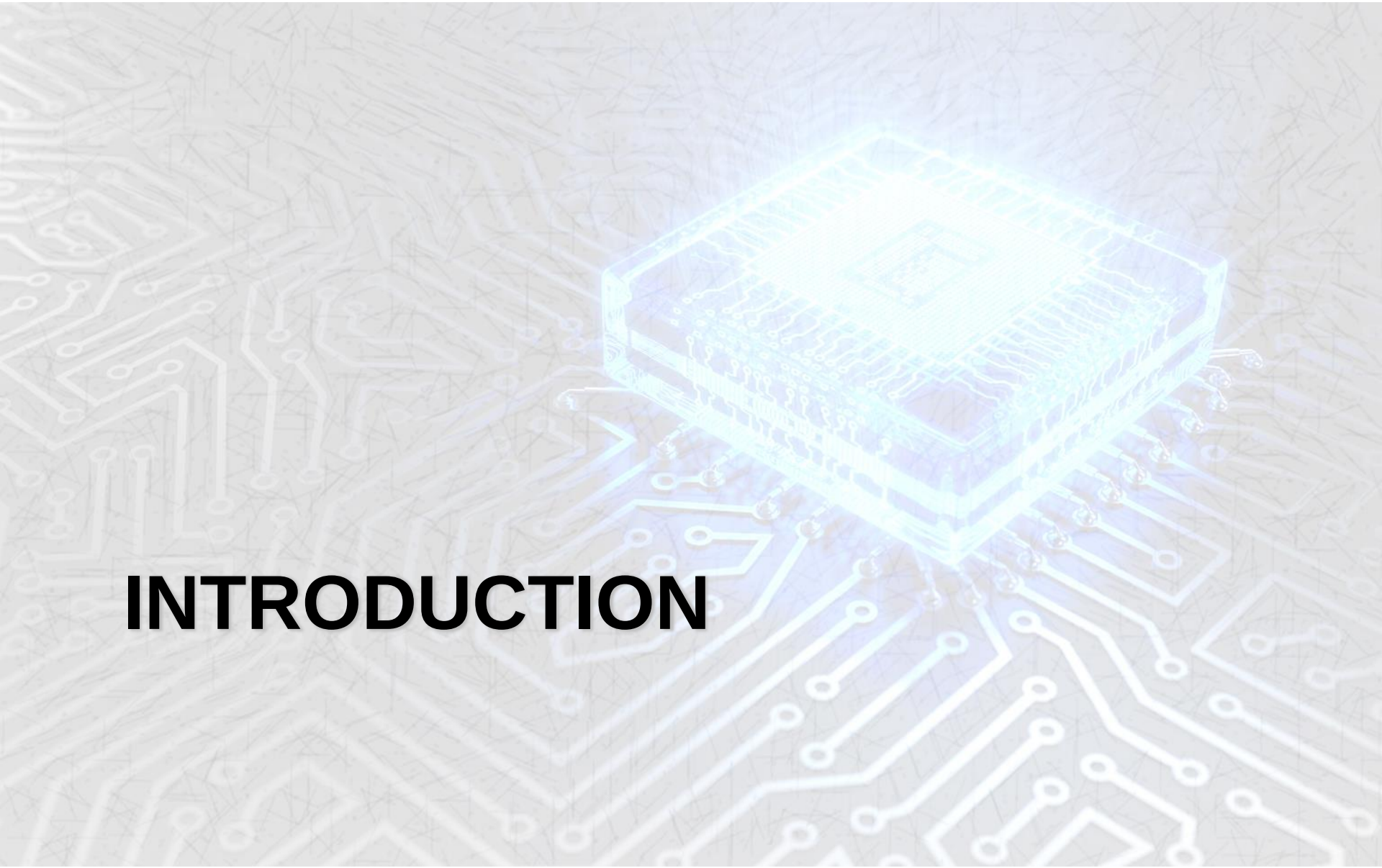
ALL PROGRAMMABLE

Hardware Description Language - Verilog

Combinational Circuits

Outline

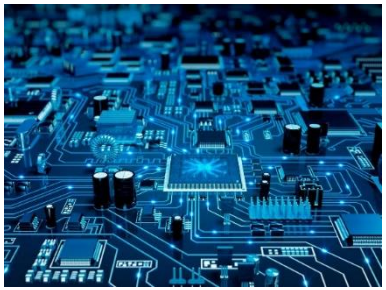
- Introduction
  - Hardware Description Language (HDL)
  - Levels of Modeling
- Hierarchical Design Methodology
  - Module Declaration
  - Ports Declaration
  - Module Instantiation
  - Ports Connection
  - Case Study
- Verilog Syntax
  - Four-Valued Logic System
  - Vector Declaration
  - Number Representation
  - Assignments
  - Operators
  - Condition Statements



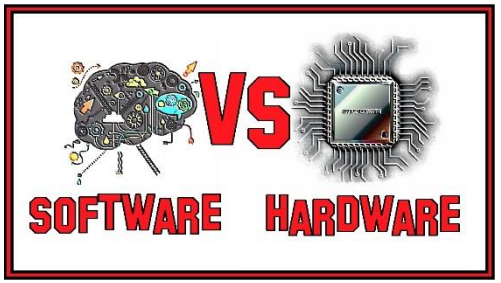
INTRODUCTION

Hardware Description Language

- **H**ardware **D**escription **L**anguage (HDL) is any language from a class of computer languages and/or programming languages for formal description of electronic circuits, and more specifically, digital logic.
- HDL can
  - Describe the circuit's operation, design, organization
  - Verify its operation by means of simulation

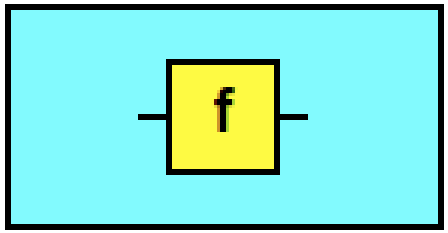


- Difference between HDL and Software
  - Support concurrency
  - Support the simulation of the progress of time
  - Support the simulation of the model of system



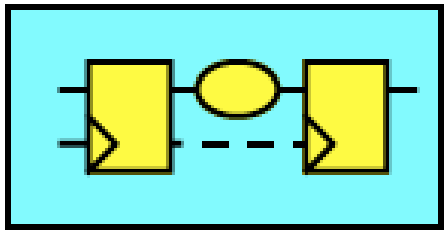
Levels of Modeling

Behavioral Level



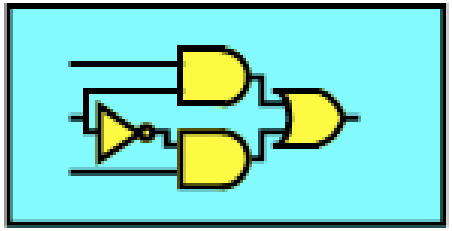

```
initial begin
    #(`CYCLE * `End_CYCLE );
    $display( "\n" );
    $display("-------------------------------------------------------\n");
    $display("Error!!! Something is wrong with your code ...!\n");
    $display("------------------------FAIL-------------------------\n");
    $display( "Terminated at: ", $time, " ns" );
    $display( "\n" );
    $stop;
end
```

Register Transfer Level (RTL)



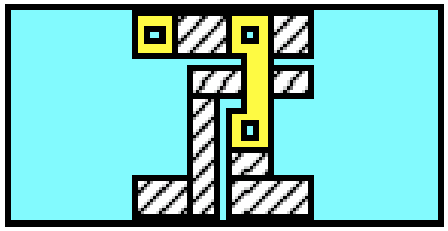
```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    assign  out=sel?in1:in2;
endmodule
```

Structural/Gate Level



```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    and a1(a1_o,in1,sel);
    not n1(iv_sel,sel);
    and a2(a2_o,in2,iv_sel);
    or  o1(out,a1_o,a2_o);
endmodule
```

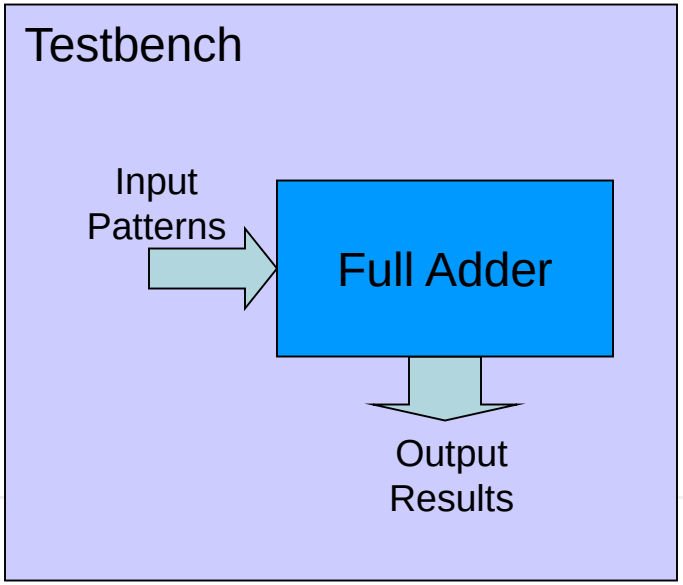
Transistor/Physical Level



```
module inv(out, in);
// port declaration
  output  out;
  input   in;
// declare power and ground
  supply1  pwr;
  supply0  gnd;
// Switch level description
  pmos  S0(out, pwr, in);
  nmos  S1(out, gnd, in);
endmodule
```

An Example – Testbech in Behavioral Level

- Behavior Level
  - Testbench
  - Non-synthesizable



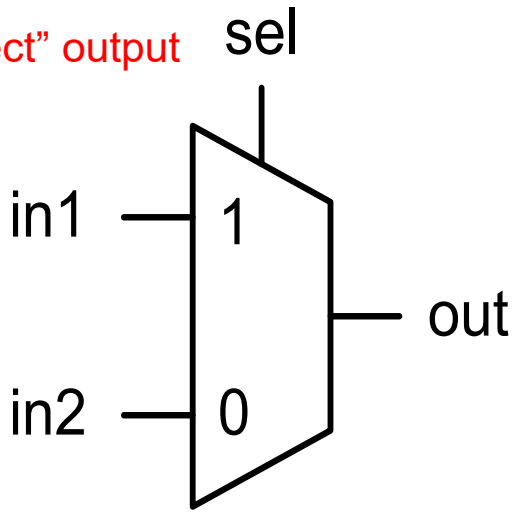

```
initial begin
    #(`CYCLE * `End_CYCLE );
    $display( "\n" );
    $display("-------------------------------------------------------\n");
    $display("Error!!! Something is wrong with your code ...!\n");
    $display("------------------------FAIL-------------------------\n");
    $display( "Terminated at: ", $time, " ns" );
    $display( "\n" );
    $stop;
end
```

An Example - 1-bit Multiplexer in RTL Level

- RTL Level
  - Synthesizable

```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    assign  out=sel?in1:in2;
endmodule
```

To "select" output



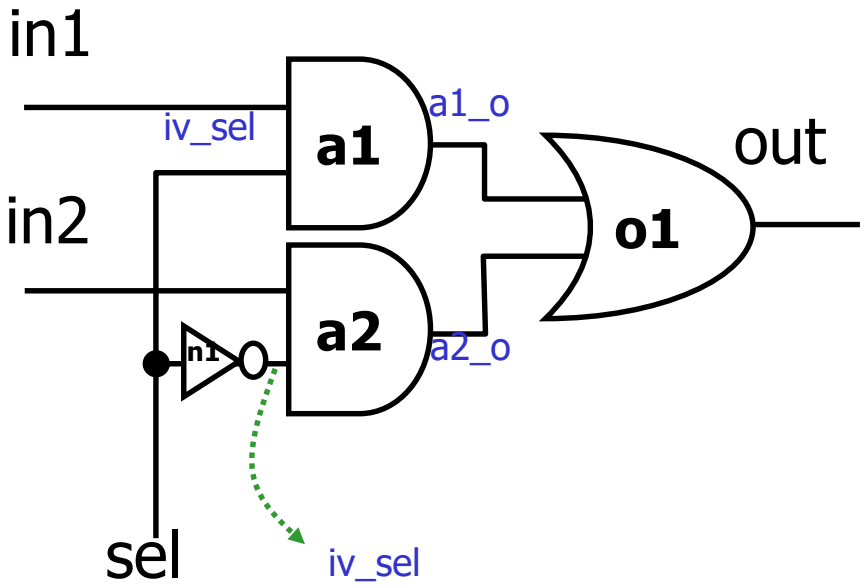
Continuous assignment

```
if (sel == 1)
    out = in1;
else
    out = in2;
```

out = sel? in1 : in2 =

An Example - 1-bit Multiplexer in Gate Level

- Gate Level
  - **Only netlist (gates and wires) in the code**
  - Synthesizable

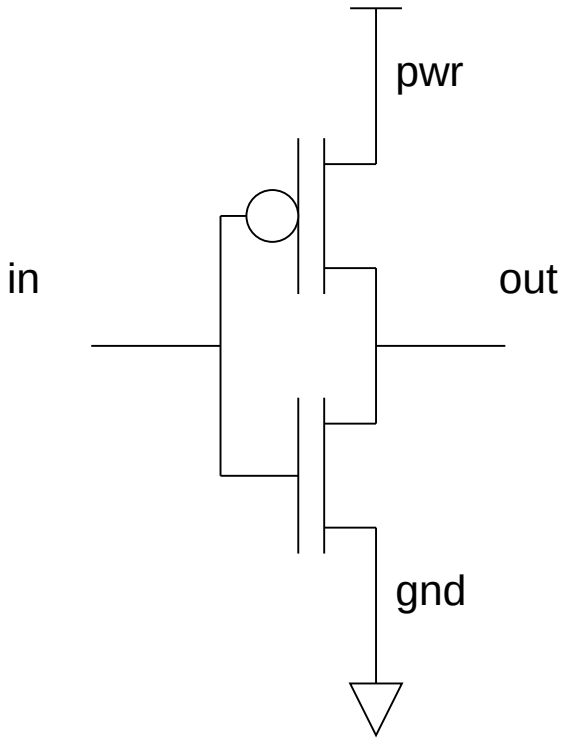



```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;
    wire    iv_sel, a1_o, a2_o;
    and a1(a1_o,in1,sel);
    not n1(iv_sel,sel);
    and a2(a2_o,in2,iv_sel);
    or  o1(out,a1_o,a2_o);
endmodule
```

An Example - 1-bit Inverter in Transistor Level

- Transistor level Verilog description
  - Using **NMOS and PMOS** to describe a circuit (Analog)

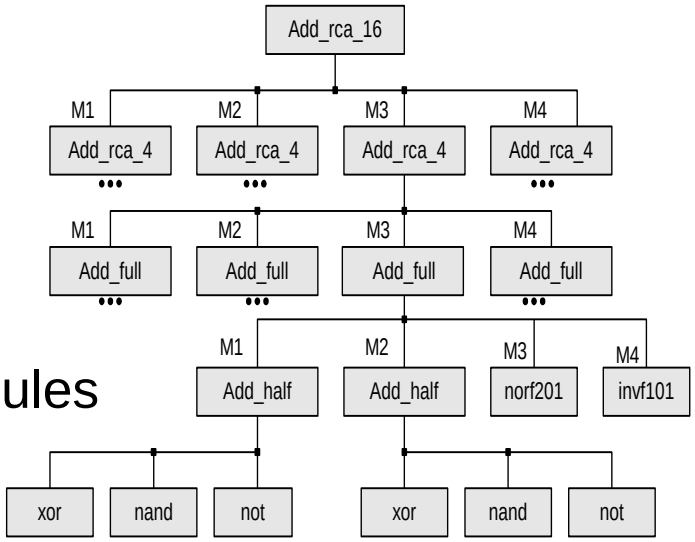
```
module inv(out, in);
// port declaration
  output  out;
  input    in;
// declare power and ground
  supply1  pwr;
  supply0  gnd;
// Switch level description
  pmos  S0(out, pwr, in);
  nmos  S1(out, gnd, in);
endmodule
```



HIERARCHICAL DESIGN METHODOLOGY

Hierarchical Modeling in Verilog

- A Verilog design consists of a hierarchy of modules.
- Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports.
- Module
  - Basic building block in Verilog
  - Created by "declaration"
  - Used by "instantiation"
  - Interface is defined by ports
  - May contain instances of other modules
  - All modules run concurrently



Module Declaration

- Encapsulate structural and functional details in a module

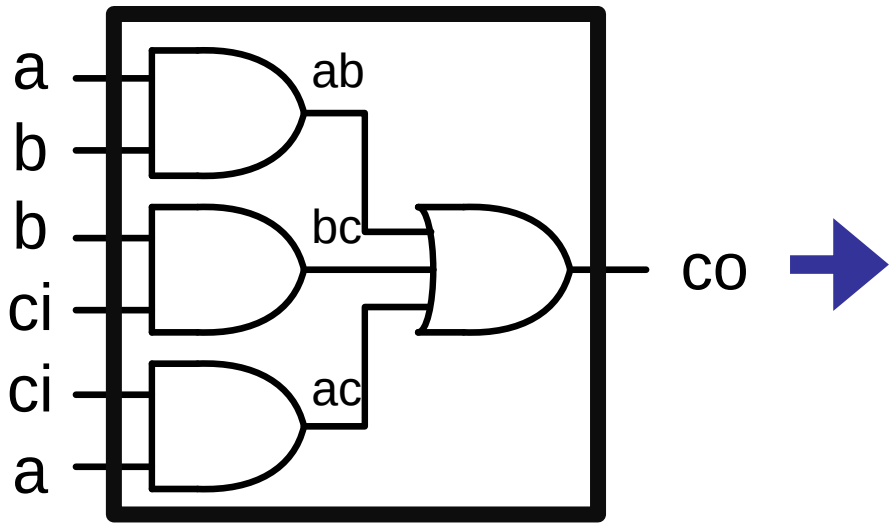
```
module <Module Name> (<PortName List>);

 // Structural part
      <List of Ports>
      <Lists of Nets and Registers>
      <SubModule List>  <SubModule Connections>

// Behavior part
      <Timing Control Statements>
            <Parameter/Value Assignments>
      <Stimuli>
      <System Task>
endmodule
```

```
module adder(out,in1,in2);
    output  out;
    input   in1,in2    ;

    assign  out=in1 + in2;
endmodule
```

- Encapsulation makes the model available for instantiation in other modules

Ports Declaration

- Two port types
  - Input port
    - input a;
  - Output port
    - output b;
- Two net types
  - **wire (can be used for module connection)**
    - wire c;
  - reg
    - reg d;




```
module FA_co( co, a, b, ci);
        output co;
        input a, b, ci;
        wire ab, bc, ac;

        and g0 ( ab, a, b );
        and g1 ( bc, b, ci );
        and g2 ( ac, ci, a );
        or   g3 ( co, ab, bc, ac );
endmodule
```

Module Instantiation

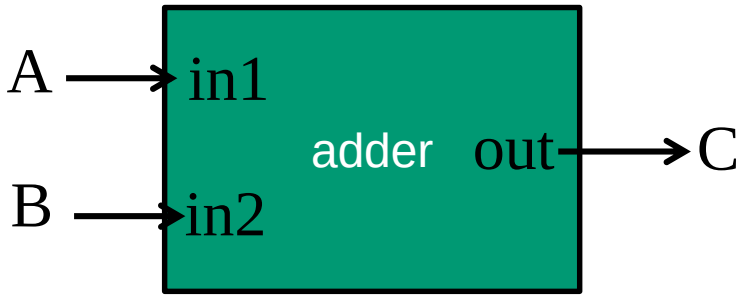
- A module provides a template from which you can create actual objects.
- When a module is invoked, Verilog creates a unique object from the template.
- Each object has its own name, variables, parameters and I/O interface.

```
module adder_tree (out0,out1,in1,in2,in3,in4);
    output  out0,out1;
    input   in1,in2,in3,in4;

    adder   add_0 (out0,in1,in2);
    adder   add_1 (out1,in3,in4);
endmodule
```

instance name IO interface

Ports Connection

```
module adder (out,in1,in2);
  output out;
  input  in1 , in2;

  assign out = in1 + in2;
endmodule
```

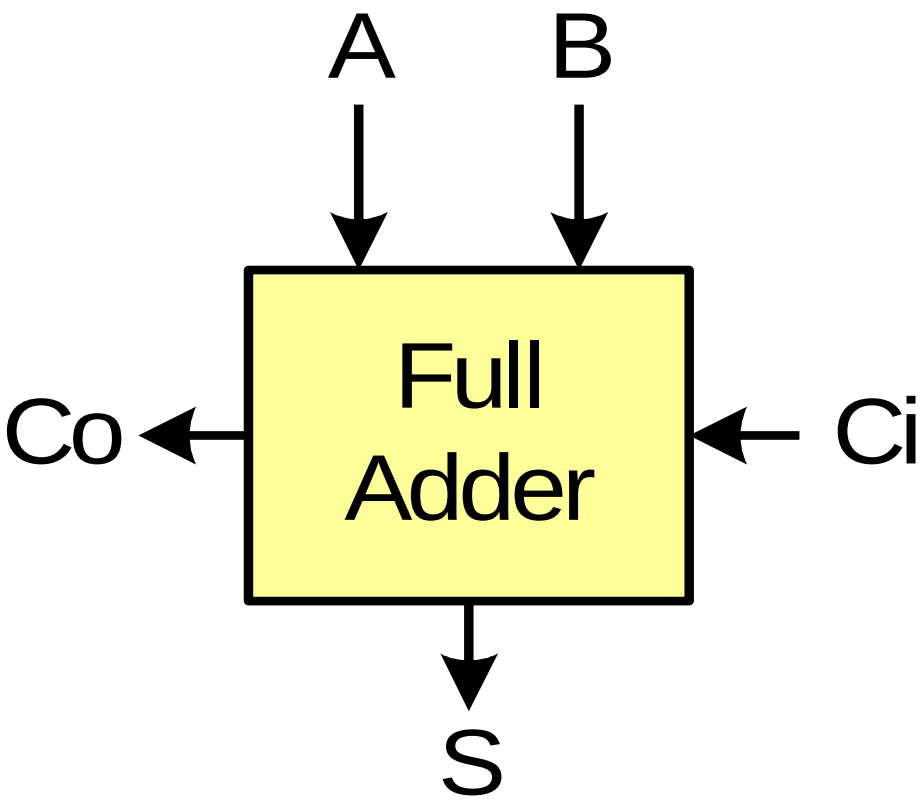


- Connect module ports by *name* **(Recommended!!!!!)**
  - Usage: .PortName (NetName)
  - `adder adder_0 ( .out(C) , .in1(A) , .in2(B) );`
- Connect module ports by order list
  - `adder adder_1 ( C , A , B );    // C = A + B`
- Not fully connected
  - `adder adder_2 ( .out(C) , .in1(A) , .in2() );`

Gate Level Modeling

- Steps
  - Develop the Boolean function of output
  - Draw the circuit with logic gates/primitives
  - Connect gates/primitives with net (usually wire)

Case Study: 1-bit Full Adder (1/4)

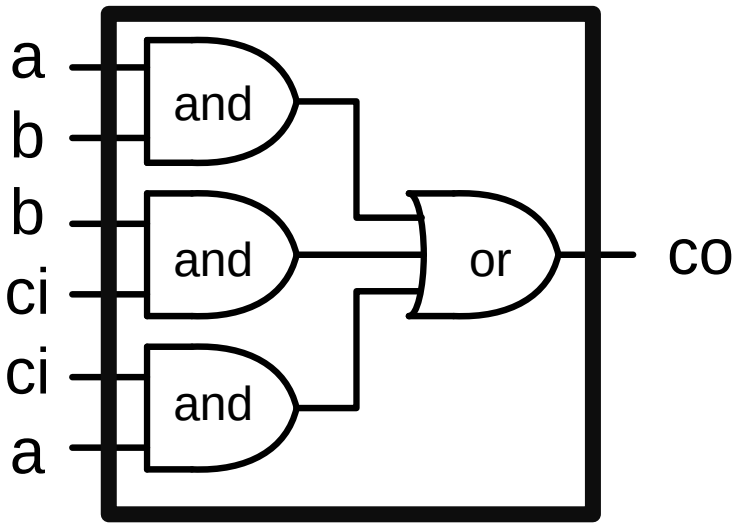


Ci	A	B	Co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Case Study: 1-bit Full Adder (2/4)

- co = (a • b) + (b • ci) + (ci • a);

Be careful the gate name is different from lib.v in Lab2. Your gate name should be based on lib.v !!!!



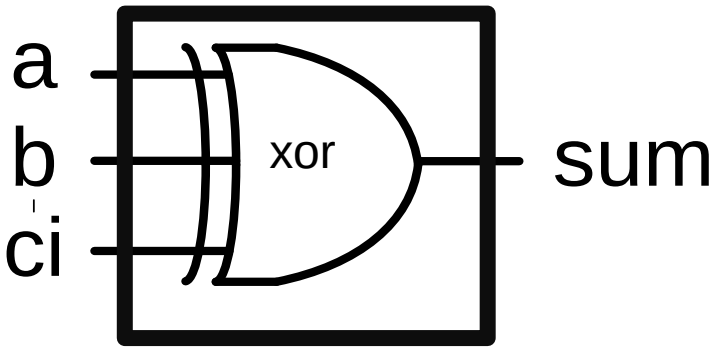

```
30
31 module FA_co ( co, a, b, ci );
32
33   input    a, b, ci;
34   output   co;
35   wire     ab, bc, ca;
36
37   and g0( ab, a, b );
38   and g1( bc, b, c );
39   and g2( ca, c, a );
40   or  g3( co, ab, bc, ca );
41
42 endmodule
43
```

instance name IO interface

Case Study: 1-bit Full Adder (3/4)

- sum = a ⊕ b ⊕ ci

Be careful the gate name is different from lib.v in Lab2. Your gate name should be based on lib.v !!!!




```
44 module FA_sum ( sum, a, b, ci );
45
46   input    a, b, ci;
47   output   sum, co;
48
49   xor g1( sum, a, b, ci );
50
51 endmodule
52
```

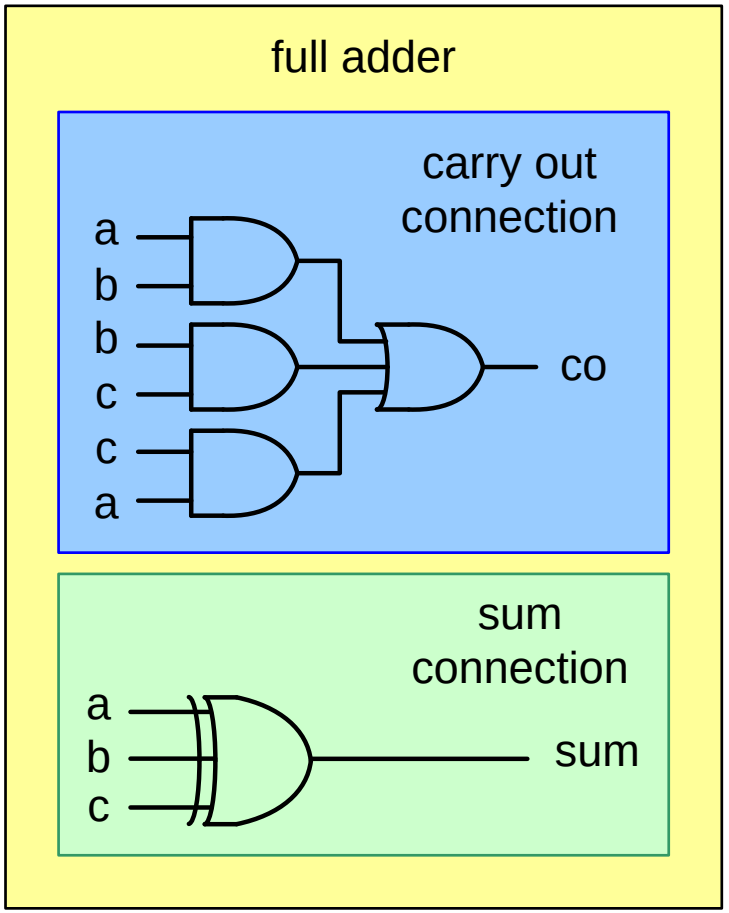
Instance name IO interface

Case Study: 1-bit Full Adder (4/4)

- Full Adder Connection
  - Instance *ins_c* from FA_co
  - Instance *ins_s* from FA_sum

```
20
21 module FA_gatelevel( sum, co, a, b, ci);
22
23   input    a, b, ci;
24   output   sum, co;
25
26   FA_co    ins_c( co, a, b, ci );
27   FA_sum   ins_s( sum, a, b, ci );
28
29 endmodule
30
```

instance name IO interface



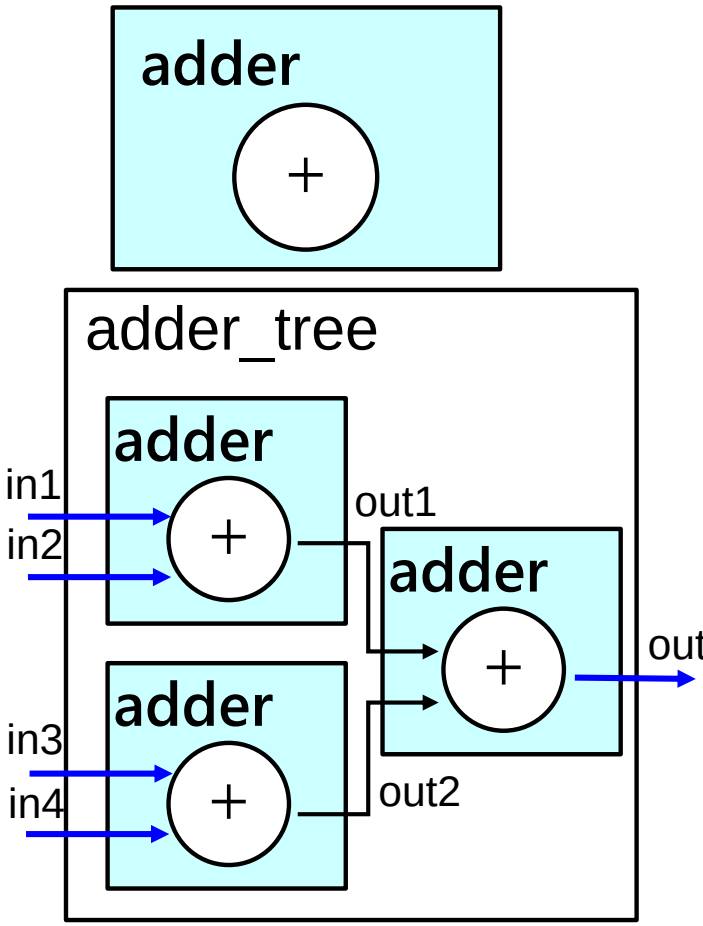
Case Study: Adder Tree

```
module adder(out, in1,in2);
        output out;
        input in1, in2;

        assign out = in1 + in2;
endmodule
```

instance example

```
module adder_tree(out, in1,in2,in3,in4);
        output out;
        input in1, in2, in3, in4;
        wire out1, out2;

        adder add_1 (.out(out1), .in1(in1), .in2(in2));
        adder add_2 (.out(out2), .in1(in2), .in2(in3));
        adder add_3 (.out(out), .in1(out1), .in2(out2));
endmodule
```

instance name IO interface



VERILOG SYNTAX

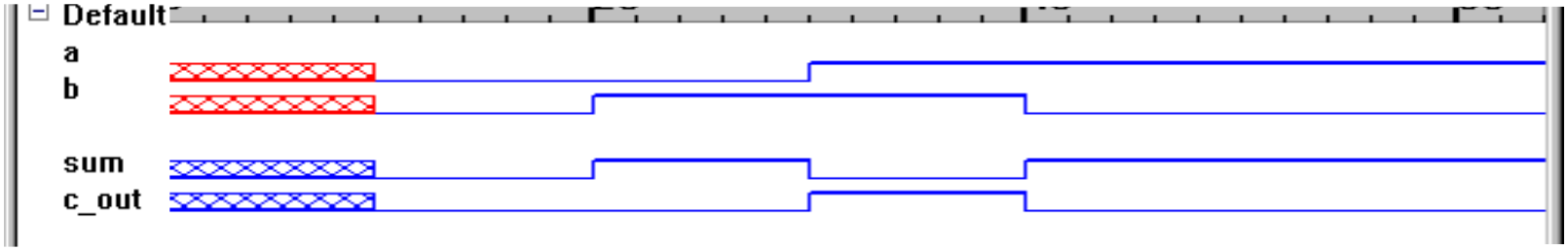
Verilog Language Rules

- **Identifiers**
  - upper and lower case letters from the alphabet
    - Cnt, cnt -> don't use !!!
  - digits (*0, 1, ..., 9*)
    - 12_reg → illegal!
  - underscore (_)
  - *$* symbol (for system tasks)
  - Max length of 1024 symbols
- Terminate lines with semicolon **;**
- Single line comments:
  - // A single-line comment goes here
- Multi-line comments:
  - /* Multi-line comments like this
    Multi-line comments like this */

```
//============  Parameter ===============
    localparam  STATE_IDLE  = 1'b0;
    localparam  STATE_CNT   = 1'b1;

/*=======================================
Author: Yu Chuan, Chuang
Module: Counter
Description:
When getting start_i signal, counter starts
to count from 0 to 15.
=======================================*/
```

Four-Valued Logic System

- Verilog's nets and registers hold four-valued data
  - 0 represent a logic **low** or **false** condition
  - 1 represent a logic **high** or **true** condition
  - z
    - High impedence
    - Usually occurs when a wire isn't connected
  - x
    - Don't care
    - When the simulator can't decide the value – uninitialized or unknown logic value
      - When a wire is being driven to 0 and 1 simultaneously



Vector Declaration

- wire and reg can be defined vector, default is 1bit
- vector is multi-bits element
- Format: **[High#:Low#]** or [Low#:High#]
- Using range specify part signals

```
wire        a;      // scalar net variable, default
wire [7:0]  bus;    // 8-bit bus
reg         clock;  // scalar register, default
reg [0:23]  addr;   // Vector register, virtual address 24 bits wide
```

```
bus[7]      // bit #7 of vector bus
bus[2:0]    // Three least significant bits of vector bus
            // using bus[0:2] is illegal because the significant bit should
            // always be on the left of a range specification
addr[0:1]   // Two most significant bits of vector addr
```

Number Representation

- Format: <size>′<base_format><number>
  - <size> - decimal specification of number of bits
  - <base format> - ' followed by arithmetic base of number
  - <number> - value given in base of <base_format>
- Examples:
  - 6′b010_111 gives 010111
  - 8′b0110 gives 00000110
  - 4′bx01 gives xx01
  - 16′H3AB gives 0000_0011_1010_1011
  - 24 gives 0...0011000
  - 5′O36 gives 11_100
  - 16′Hx gives xxxxxxxxxxxxxxx
  - 8′hz gives zzzzzzzz

Values Assignment

- Assignment: Drive value onto nets and registers
- There are two basic forms of assignment
  - continuous assignment, which assigns values to *wire* type
  - procedural assignment, which assigns values to *reg* type
- Basic form

Assignments	Left Hand Side	Example
Continuous Assignment	wire	wire a; assign a = 1'b1;
Procedural Assignment	reg	reg a; always@(*) a = 1'b1;

P.S. Left hand side (LHS) = Right hand side (RHS)

Continuous Assignments

- Example 1

```
module holiday_1(sat, sun, weekend);
      input sat, sun;
      output weekend;
      assign weekend = sat | sun;      // outside a procedure
endmodule
```

- Example 2

```
module counter (
    input           clk,
    input           rst,
    input           start_i,
    output  [3:0]   count_o
);

reg [3:0]   cnt, nxt_cnt;

assign count_o = cnt;
```

Procedural Assignment

- Procedure blocks
  - **`always`** block
- Only ***reg*** type can be LHS in procedure blocks
  - It's syntax. Not relevant to whether it's a flip-flop or a metal wire!
  - RHS is not restricted.
- Variable is updated by *procedural assignment.*
  - For combinational circuit
    - Pure logic circuit
    - Use blocking assignment (=)
    - Use always @ (*)
  - For sequential circuit
    - D-FF circuit
    - Use non-blocking assignment (<=)
    - Edge trigger of clock or reset signal

```
//============ Combinational ==========
    always@(*) begin
        if(state == STATE_CNT) begin
            nxt_cnt = cnt + 1;
        end
        else begin
            nxt_cnt = 0;
        end
    end
```

Operators

Operator	Sign	Example
Concatenation and replications	{,}	{{8{byte[7]},byte}, {a[1:0],b[0]}
Negation	**! , ~**	!2'01 //0 , ~2'b01 // 10
Bitwise	**~ , & , | , ^**	2'b01 | 2'b11 // 2'b11
Arithmetic	**+ , - , * , / , %**	3%2 //1
Shift	>> , <<	3'b011 >> 2 // 3'b000
Relational	< , <= , > , >=	
Equality	**== , !=**	
Conditional	**? :**	assign out = sel ? 1'b1 : 1'b0

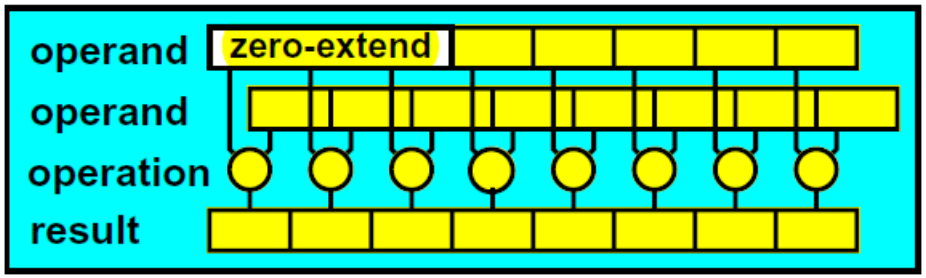
Negation Operators

- The logical negation operator (**!**)
  - produces a 0, 1, or X scalar value
  - An operand is logically false if **all** of its bits are 0
  - An operand is logically true if **any** of its bits are 1
- The bitwise negation operator (**~**)
  - inverts each individual bit of the operand.

```
module negation;
  initial begin
      $displayb ( !4'b0100 ); // 0
      $displayb ( !4'b0000 ); // 1
      $displayb ( !4'b00z0 ); // x
      $displayb ( !4'b000x ); // x
      $displayb ( ~4'b01zx ); // 10xx
    end
endmodule
```

Bit-Wise Operators

- Bit-wise operators (&,|,^,^~)
  - operate on each individual bit of a vector




```
module bit_wise;
  initial begin
      $displayb ( 4'b01zx  & 4'b0000 ); // 0000
      $displayb ( 4'b01zx  & 4'b1100 ); // 0100
      $displayb ( 4'b01zx  & 4'b1111 ); // 01xx
      $displayb ( 4'b01zx  | 4'b1111 ); // 1111
      $displayb ( 4'b01zx  | 4'b0011 ); // 0111
      $displayb ( 4'b01zx  | 4'b0000 ); // 01xx
      $displayb ( 4'b01zx  ^ 4'b1111 ); // 10xx
      $displayb ( 4'b01zx ^~ 4'b0000 ); // 10xx
    end
endmodule
```

Arithmetic Operators

- The arithmetic operators (*, */*, %, +, -)
  - Produce numerical or unknown results
  - Integer division discards any remainder
  - An unknown operand produces an unknown result
  - Assignment of a signed value to an unsigned register is 2's-complement

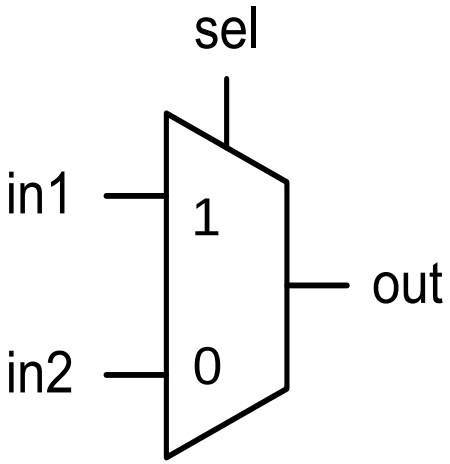
```
module arithmetic;
  initial begin
      $display  ( -3 * 5 ); // -15
      $display  ( -3 / 2 ); // -1
      $display  ( -3 % 2 ); // -1
      $display  ( -3 + 2 ); // -1
      $display  (  2 - 3 ); // -1
      $displayh ( 32'hfffffffd / 2 ); // 7ffffffe
      $displayb ( 2 * 1'bx); // xx...
    end
endmodule
```

Equality Operators

- Equality operators
  - “==“ , “!=“

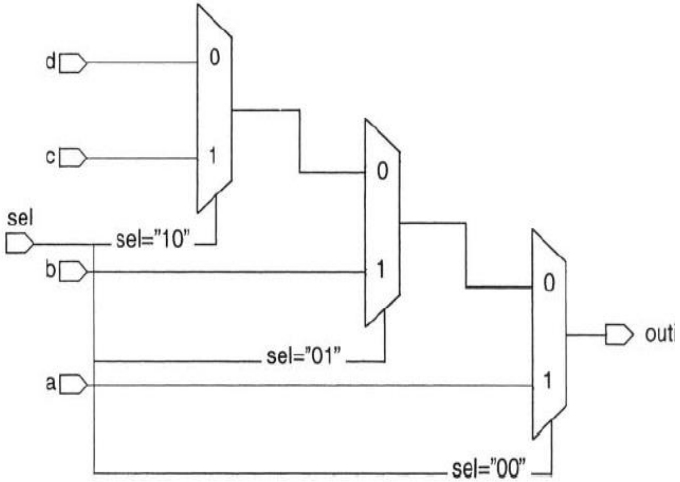
```
module equality;
  initial begin
      $displayb ( 4'b0011 == 4'b1010 ); // 0
      $displayb ( 4'b0011 != 4'b1x10 ); // 1
      $displayb ( 4'b1010 == 4'b1x10 ); // x
      $displayb ( 4'b1x10 == 4'b1x10 ); // x
      $displayb ( 4'b1z10 == 4'b1z10 ); // x
    end
endmodule
```

Conditional Operators



Continuous assignment

```
assign out = sel ? in1 : in2;
```

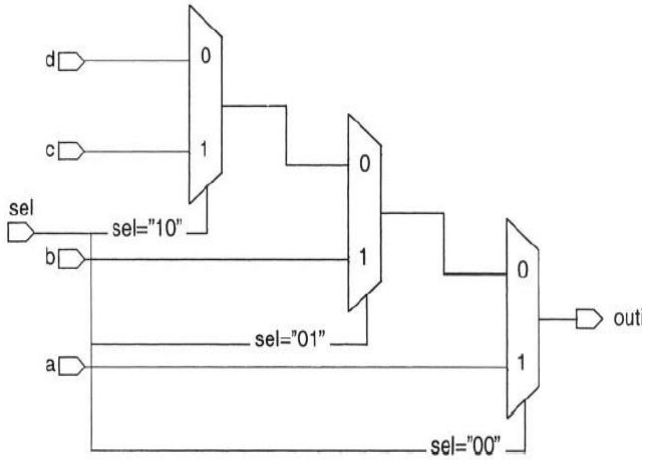



```
assign out =
(sel == 2'b10) ? c :
(sel == 2'b01) ? b :
(sel == 2'b00) ? a : d;
```

Condition Statement

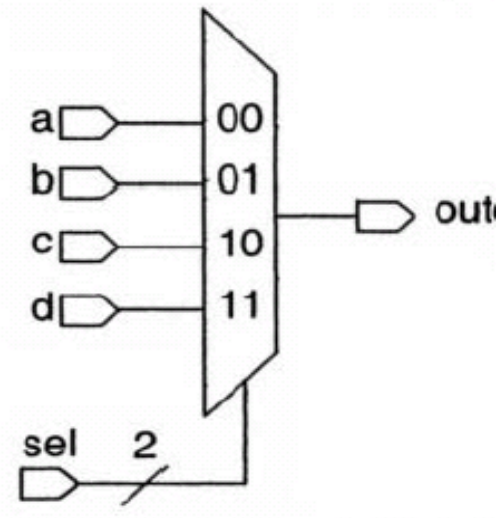
- **If-else** statement

```
always@(*) begin
    if (sel == 2'b00)
      out = a;
    else if (sel = 2'b01)
      out = b;
    else if (sel = 2'b10)
      out = c;
    else
      out = d;
end
```



- **Case** statement

```
always@(*)
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
      default: out = d;
    endcase;
end
```



```
/*=======================================
Author: Yu Chuan, Chuang
Module: Counter
Description:
When getting start_i signal, counter starts
to count from 0 to 15.
=======================================*/
module counter (
    input           clk,
    input           rst,
    input           start_i,
    output  [3:0]   count_o
);

//============  Parameter ===============
    localparam  STATE_IDLE  = 1'b0;
    localparam  STATE_CNT   = 1'b1;

//=========  Reg/Wire Declaration ======
    reg         state, nxt_state;
    reg [3:0]   cnt, nxt_cnt;
//========== Finite State Machine =======
    always@(posedge clk or posedge rst) begin
        if(rst)
            state <= STATE_IDLE;
        else
            state <= nxt_state;
    end

    always@(*) begin
        case(state)
            STATE_IDLE: begin
                if(start_i)
                    nxt_state = STATE_CNT;
                else
                    nxt_state = STATE_IDLE;
            end
            STATE_CNT: begin
                if(cnt == 4'd15)
                    nxt_state = STATE_IDLE;
                else
                    nxt_state = STATE_CNT;
            end
            default: nxt_state = STATE_IDLE;
        endcase
    end

//============ Combinational ===========
    assign count_o = cnt;

    always@(*) begin
        if(state == STATE_CNT) begin
            nxt_cnt = cnt + 1;
        end
        else begin
            nxt_cnt = 0;
        end
    end

//============= Sequential =============
    always@(posedge clk or posedge rst) begin
        if(rst)
            cnt <= 0;
        else
            cnt <= nxt_cnt;
    end

endmodule
```

Header/Comment
Module instantiation
Input/output declaration
Parameter
Number Representation
Reg/WIre
FSM
Procedure Block
If statement
case statement
Combinational
Continuous Assignment
Sensitivity list
Operator
Procedure Assignment
Sequential

Thanks! Feel free to ask me any questions!