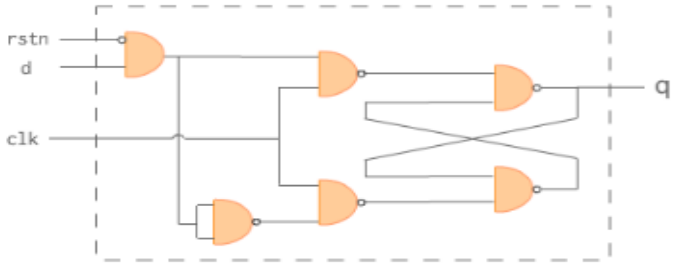
VERILOG HDL and MODELSIM Introduction

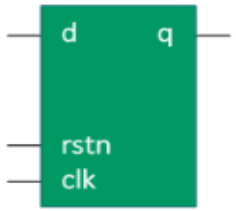
CENG311

LAB #6

What is Hardware Schematic?



Hardware Schematic



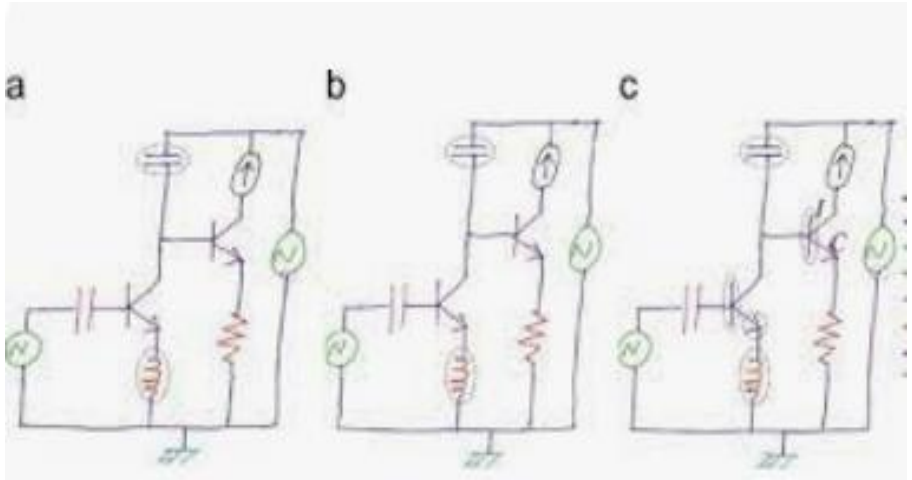
Black-box Representation

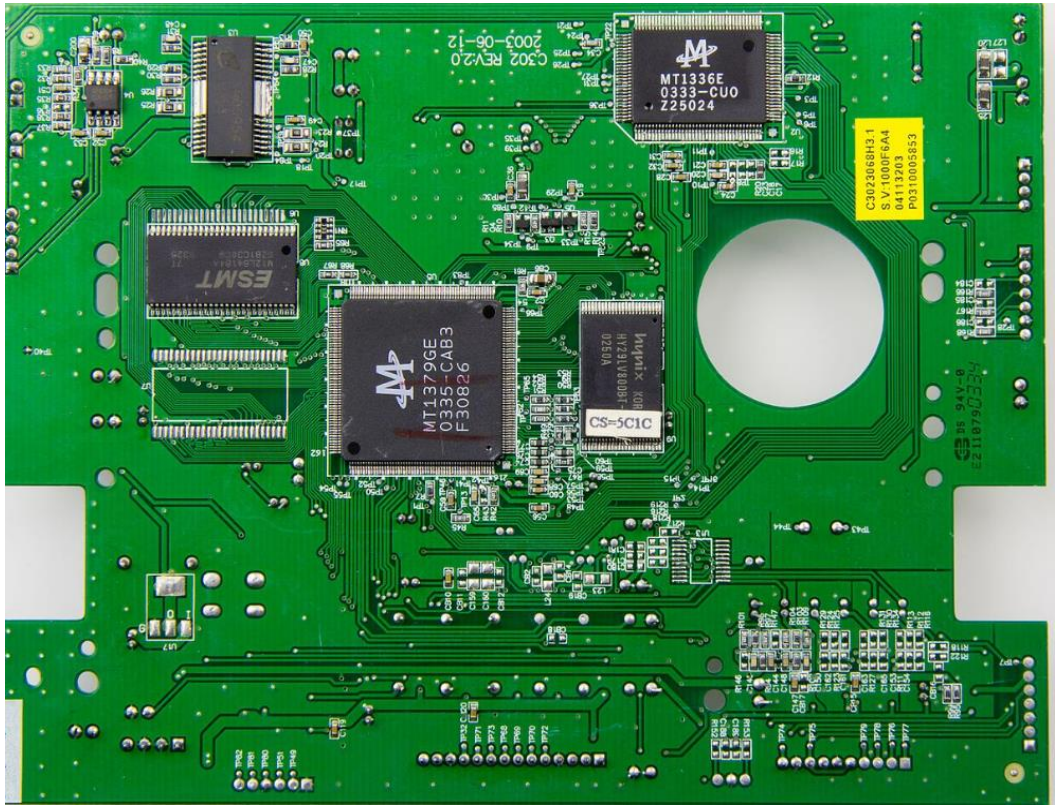
rstn	d	q
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table

- Hardware schematic shows the connection of combinational gates to achieve particular hardware functionality.
- If we now the truth table, hardware schematic can be encapsulated into a black-box schema.

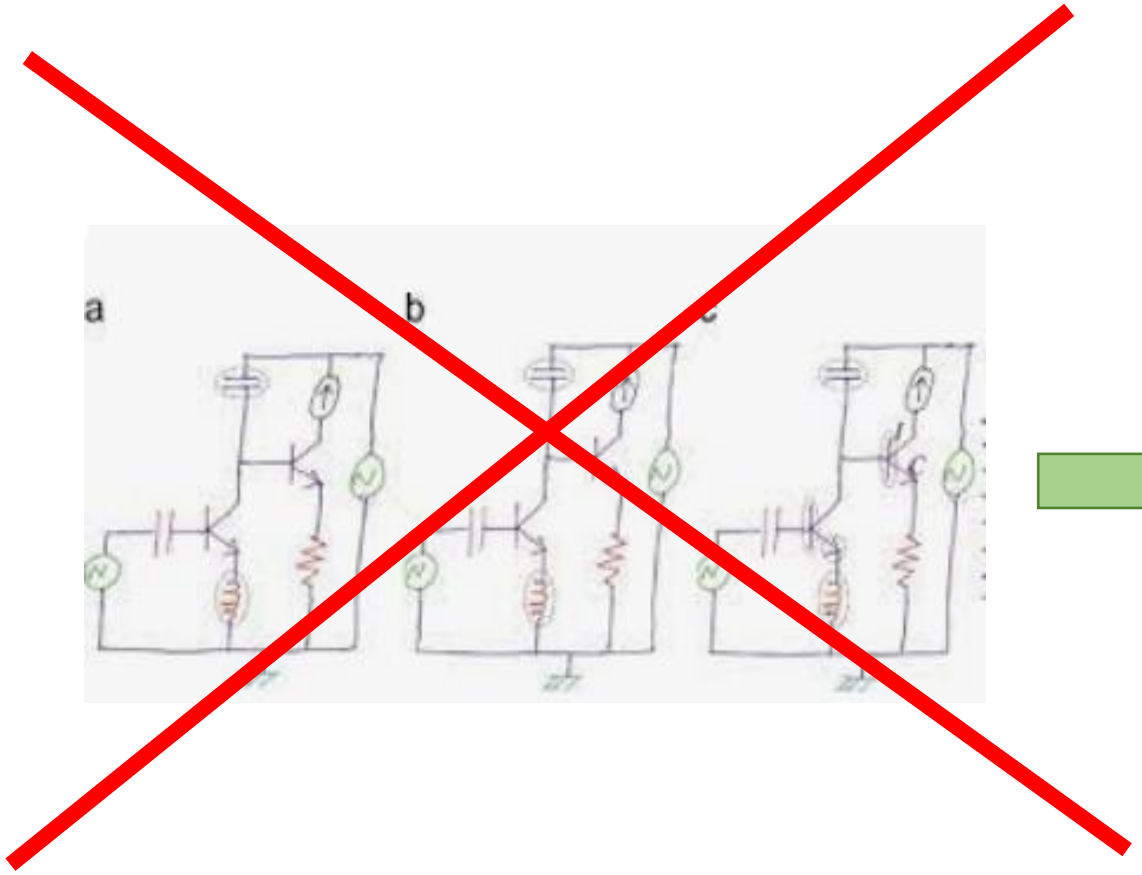
What is Hardware Schematic?

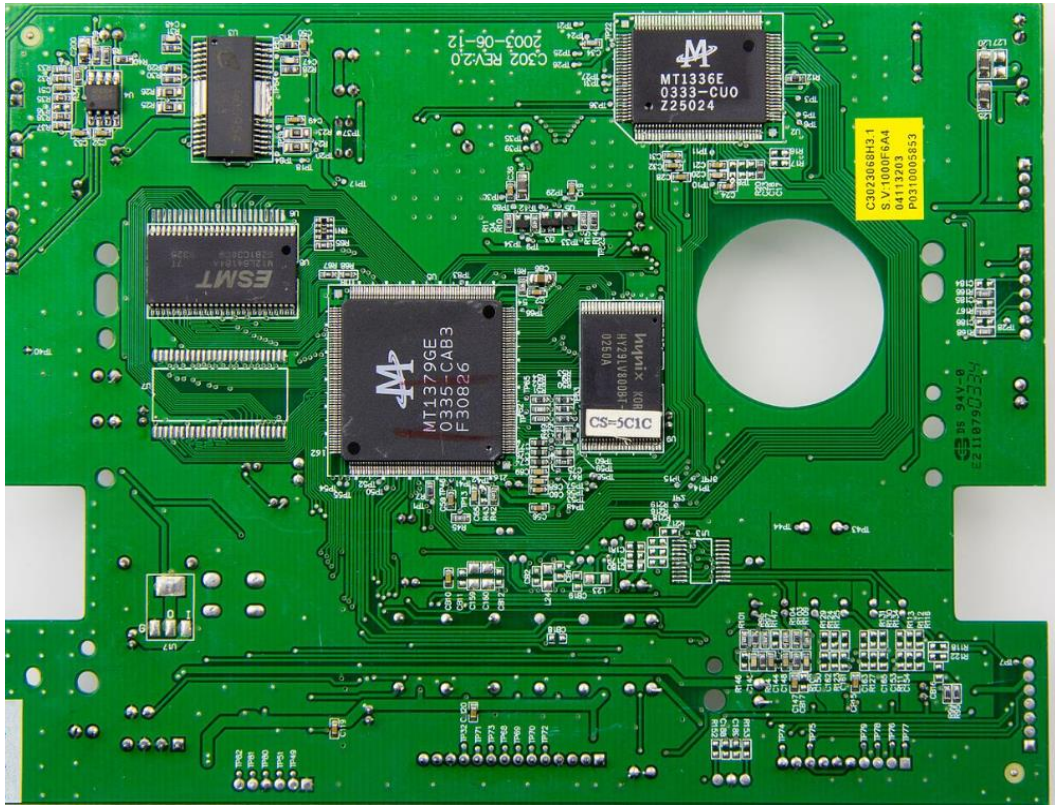




PCB: printed circuit board

What is Verilog?

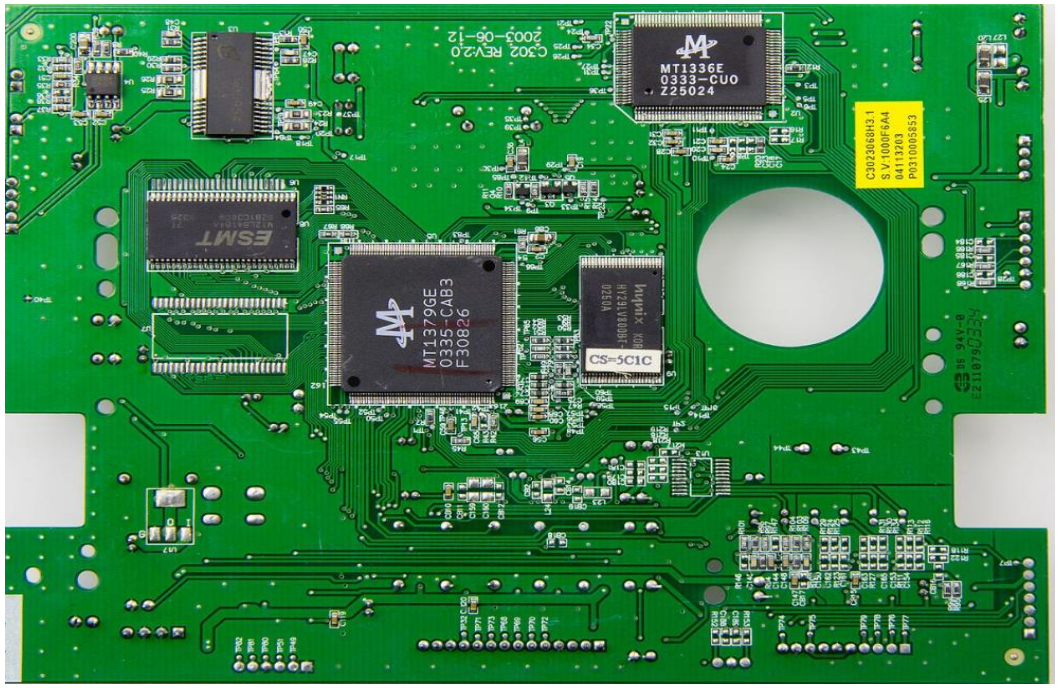




PCB: printed circuit board

What is Verilog?

```
ClkDivP : process(Mclk,SeqReset)
   begin
   if SeqReset = '0' then
      ADCClk <= '0';
      ADC_div <= "001001";
   elsif Mclk = '0' and Mclk'event then
      if ADC_div = "000000" then
         ADCClk <= not(ADCClk);
         case ClkSel is                          -- "000" is used to conditionally
                                                 -- select the input clock
                                                 -- and is excluded in the cases
                                                 -- below
            when "001" =>                        -- 20MHz - divide by 2
            when "010" =>                        -- 10MHz
               ADC_div <= "000001";                 -- divide by 4
            when "011" =>                        -- 4MHz
               ADC_div <= "000100";                 -- divide by 10
            when "100" =>                        -- 2MHz
               ADC_div <= "001001";                 -- divide by 20
            when "101" =>                        -- 1MHz
               ADC_div <= "010011";                 -- divide by 40
            when others =>                       -- 400KHz
               ADC_div <= "110001";                 -- divide by 100
         end case;
      else
         ADC_div <= (unsigned(ADC_div) - 1);
      end if;
   end if;
```



- If we can describe how this black-box block should behave and then let software tools convert that behavior into actual hardware schematic.
- **The language** that describes hardware functionality is called as **Verilog** which is classified as **H**ardware **D**escription **L**anguage (HDL)

PCB: printed circuit board

Syntax

- Verilog is case-sensitive -> var_a and var_A are different
- Identifiers cannot start with a digit or dollar sign

  integer 2var; -> INVALID

  integer $var_a -> INVALID

  integer 234 -> INVALID

  integer var2_g -> VALID
- All lines terminate with a semicolon -> ;
- Single line comment -> //
- Multiple line comment block -> /* ..... */

Operators

There are three types of operators: *unary*, *binary*, and *ternary or conditional*.

- Unary operators shall appear to the left of their operand
- Binary operators shall appear between their operands
- Conditional operators have two separate operators that separate three operands

```
x = ~y;                 // ~ is a unary operator, and y is the operand
x = y | z;              // | is a binary operator, where y and z are its operands
x = (y > 5) ? w : z;    // ?: is a ternary operator, and the expression (y>5), w and z are its operands
```

If the expression (y > 5) is true, then variable *x* will get the value in *w*, else the value in *z*.

Strings

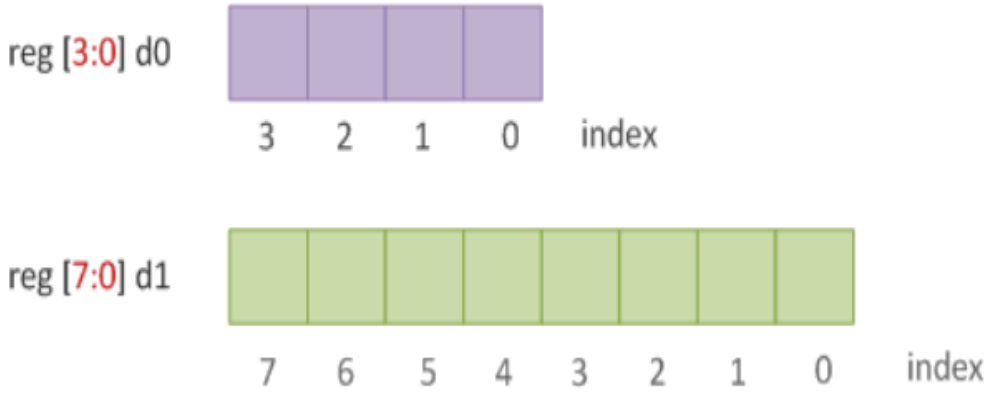
A sequence of characters enclosed in a double quote `" "` is called a string. It cannot be split into multiple lines and every character in the string take 1-byte to be stored.

```
"Hello World!"          // String with 12 characters -> require 12 bytes
"x + z"                  // String with 5 characters

"How are you
feeling today ?"        // Illegal for a string to be split into multiple lines
```

Wires and Registers

- **The wire** is just simply the wire that is used to provide econnection between devices and carries current. **In verilog**, these wires are used to connect modules to each other, make the connection between registers and combinational circuitry.
- Registers are the storage elements. (One should perceive those elements as flip-flops you are familiar from Logic course.)

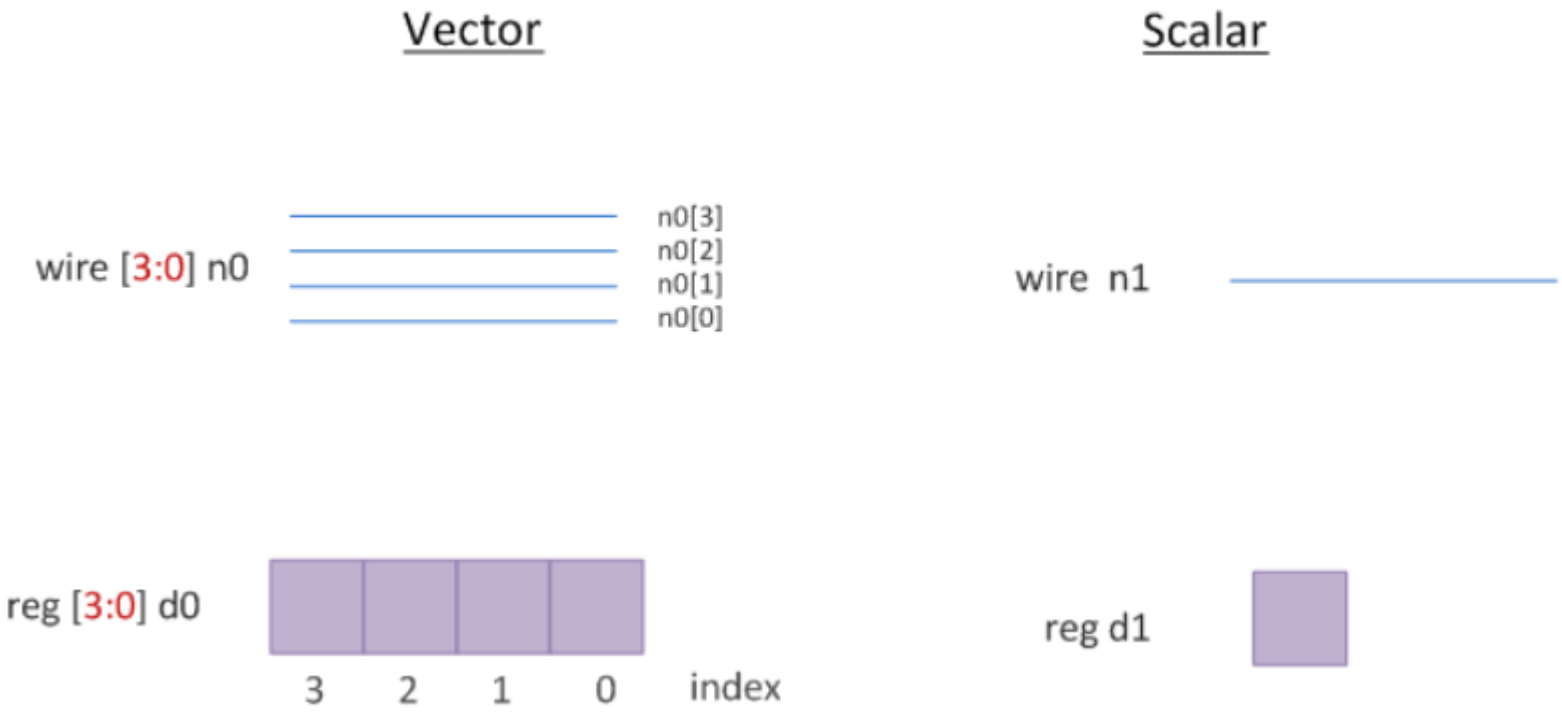
```
wire [3:0] n0;        // 4-bit wire -> this is a vector
```



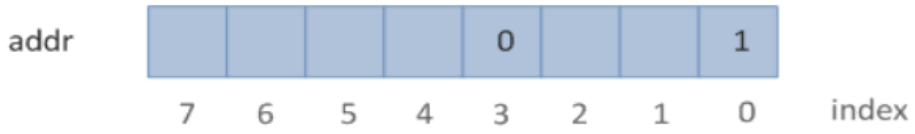
32 registers each with 128 bits

```
reg [31:0] x[127:0];        // 128-element array of 32-bit wide reg
```

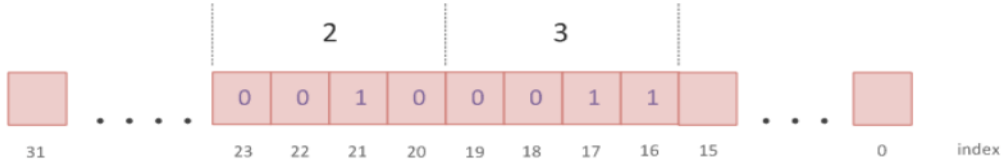
Scalar and Vectors



Bit-select and Part-select




```
reg [7:0]      addr;          // 8-bit reg variable [7, 6, 5, 4, 3, 2, 1, 0]

addr [0] = 1;                 // assign 1 to bit 0 of addr
addr [3] = 0;                 // assign 0 to bit 3 of addr
addr [8] = 1;                 // illegal : bit8  does not exist in addr
```




```
reg [31:0]    addr;

addr [23:16] = 8'h23;        // bits 23 to 16 will be replaced by the new value 'h23 -> constant part
```

Number Format

```
16          // Number 16 in decimal
0x10        // Number 16 in hexadecimal
10000       // Number 16 in binary
20          // Number 16 in octal
```

```
[size]'[base_format][number]
```

- *base_format* can be either decimal ('d or 'D), hexadecimal ('h or 'H) and octal ('o or 'O) and specifies what base the *number* part represents.
- *number* is specified as consecutive digits from 0, 1, 2 ... 9 for decimal base format and 0, 1, 2 .. 9, A, B, C, D, E, F for hexadecimal.

Examples

- 3'b010; // size is 3, base format is binary ('b), and the number is 010 (indicates value 2 in binary)
- 3'd2; // size is 3, base format is decimal ('d) and the number is 2 (specified in decimals)
- 8'h70; // size is 8, base format is hexadecimal ('h) and the number is 0x70 (in hex) to represent decimal 112
- 9'h1FA; // size is 9, base format is hexadecimal ('h) and the number is 0x1FA (in hex) to represent decimal 506

- 4'hA = 4'd10 = 4'b1010 = 4'o12 // Decimal 10 can be represented in any of the four formats
- 8'd234 = 8'D234 // Legal to use either lower case or upper case for base format
- 32'hFACE_47B2; // Underscore (_) can be used to separate 16 bit numbers for readability

Verilog Code Template

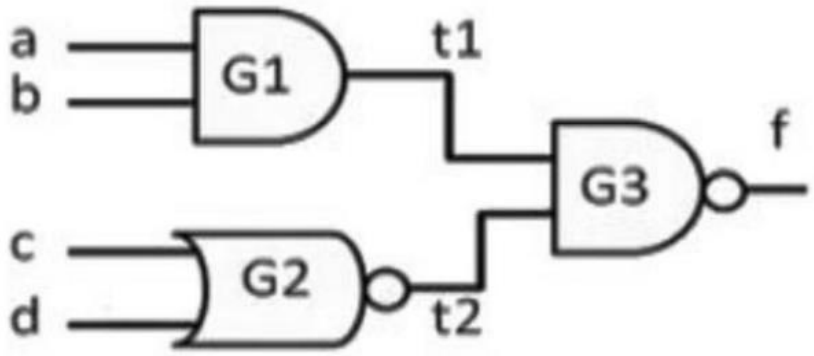
```
module adder(a,b,out);
input [31:0]  a,b;
output [31:0] out;
assign out=a+b;
endmodule
```

```
module fa (  input   a, b, cin,
             output   sum, cout);

assign sum  = (a ^ b) ^ cin;
assign cout = (a & b) | ((a ^ b) & cin);
endmodule
```

- A module is a block of Verilog code that implements a certain functionality
- As a default input or output ports declared as wire format.
- Assign statement continuously assign value to the net called as out. Out is updated when any of the variables on the RHS change in value.

Example

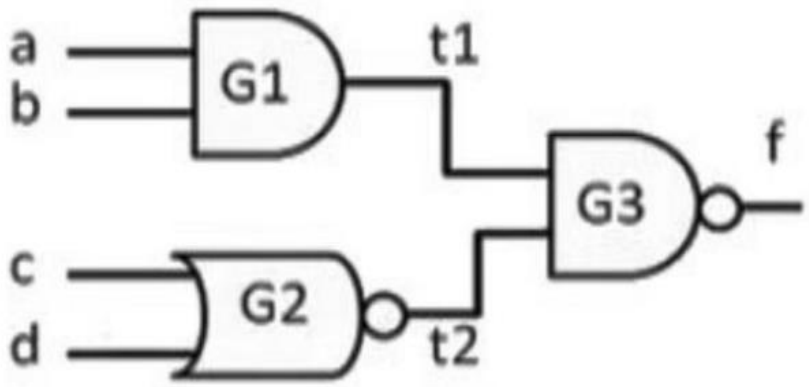
What are the inputs and output of this circuit?



Example

```
// A 2-level combinational circuit
module two_level(a, b, c, d, f);
    input a, b, c, d;
    output f;
```

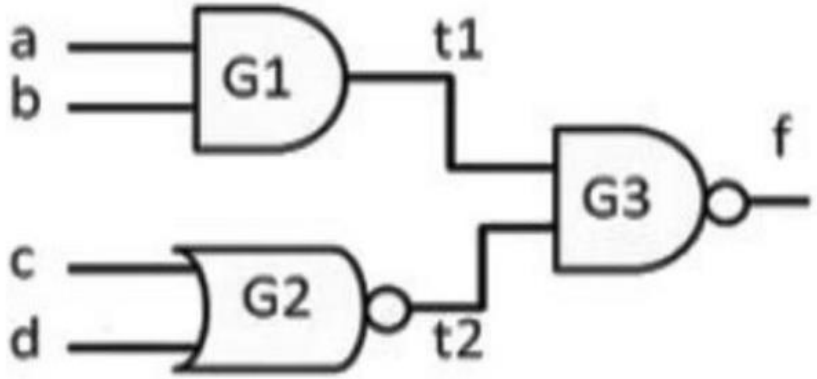
What are the internal connections/wires of this circuit?



Example

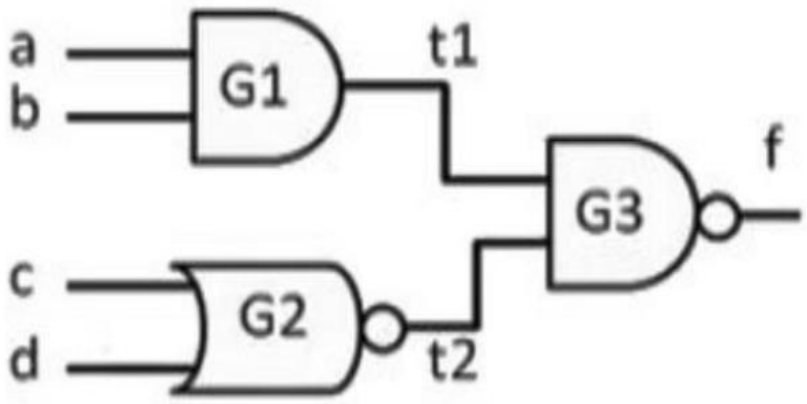
```
// A 2-level combinational circuit
module two_level(a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire t1, t2; // intermediate lines
```

Please try to formulate is t1, t2 and f.



Example

```
// 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
```



Always Block

```
always @ (event)
    [statement]

always @ (event) begin
    [multiple statements]
end
```

- Statements inside always block are executed sequentially
- Always block is triggered or executed according to some predefined sensitivity list.

```
// Execute always block whenever value of "a" or "b" change
always @ (a or b) begin
    [statements]
end
```

```
// Execute always block at positive edge of signal "clk"
always @ (posedge clk) begin
    [statements]
end
```

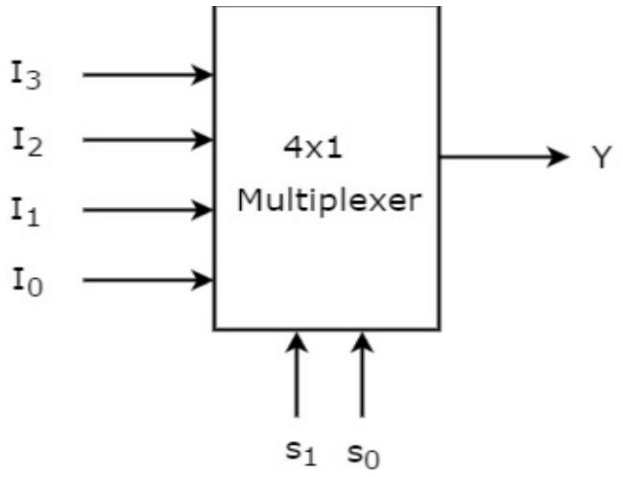
Always block implementation: and gate

```
module and_gate(out,a,b);
output reg out;
input  a,b;

always @ ( a or b) begin
  if ( a==1'b1 & b==1'b1)begin
          out = 1'b1;
       end
       else
          out = 1'b0;
      end

endmodule
```

Switch-case structure

```
// Here 'expression' should match one of the items (item 1,2,3 or 4)
case (<expression>)
    case_item1 :    <single statement>
    case_item2,
    case_item3 :    <single statement>
    case_item4 :    begin
                        <multiple statements>
                        end
    default      : <statement>
endcase
```

Switch-case implementation: 4x1 Multiplexer(Mux)




```
module mux4to_1(out,sel,i0,i1,i2,i3);
output reg out;
input i0,i1,i2,i3;
input [1:0] sel;

always @ (i0 or i1 or i2 or i3 or sel)
begin

case(sel)
2'b00 : out <= i0;
2'b01 : out <= i1;
2'b10 : out <= i2;
2'b11 : out <= i3;
endcase

end

endmodule
```

Modelsim

- You can download Modelsim via the link: https://www.intel.com/content/www/us/en/software-kit/750368/modelsim-intel-fpgas-standard-edition-software-version-18-1.html

- Choose proper operating system: Linux / Windows (Windows preferred)
- If you have a problem in Ubuntu version, follow the instructions in the link: https://gist.github.com/Razer6/cafc172b5cffae189b4ecda06cf6c64f

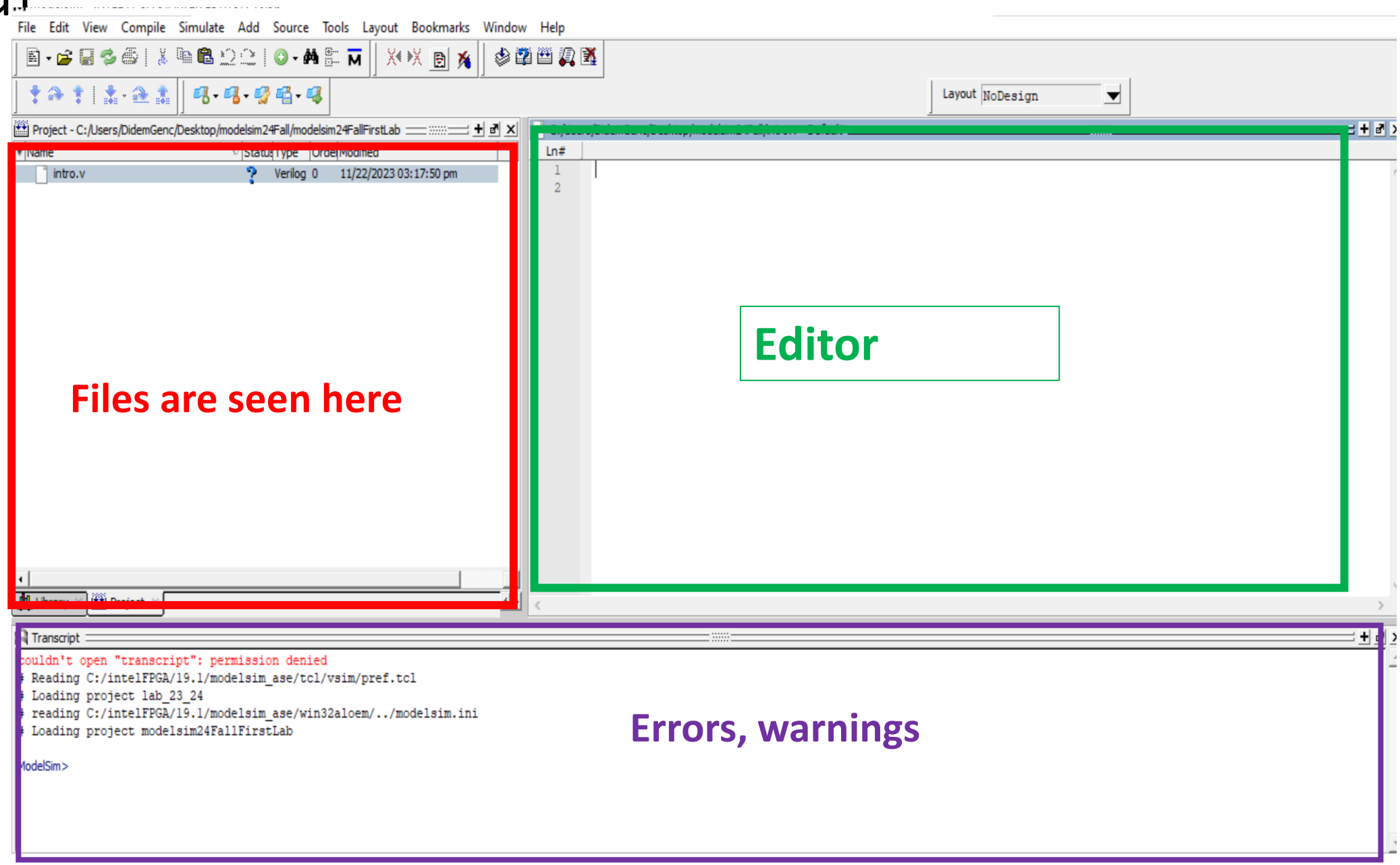
- Download and install instructions:

1. Download the ModelSim-Intel® FPGA software into a temporary directory.
2. Run the .exe file

Creating a Project

1. File-> New -> Project
2. Specify the Project Location
3. In the pop-up window choose the new file (or you can choose add existent file if you have already)
4. Write your file name (File name and Project name are different!, one Project may have many files)
5. Specify the file type as Verilog!!
6. **Now, You are ready to coding.**

Layout


File Edit View Compile Simulate Add Source Tools Layout Bookmarks Window Help
Layout NoDesign
Project - C:/Users/DidemGenc/Desktop/modelsim24Fall/modelsim24FallFirstLab
Name
Status
Type
Order
Modified
intro.v
Verilog 0
11/22/2023 03:17:50 pm
Ln#
Files are seen here
Editor
Transcript
couldn't open "transcript": permission denied
Reading C:/intelFPGA/19.1/modelsim_ase/tcl/vsim/pref.tcl
Loading project lab_23_24
reading C:/intelFPGA/19.1/modelsim_ase/win32aloem/../modelsim.ini
Loading project modelsim24FallFirstLab
ModelSim>
Errors, warnings

Warm up activities with MODELSIM

Difference of = and <=

- = is blocking
- <= non-blocking

- In always block it should be non-blocking; codes executed sequentially

Adder-Subtractor Implementation

```
module add_sub(a,b,sel,out);

input [5:1] a,b;
input sel; //add is sel is 1, sub if sel is 0
output reg [7:1] out;

always @ (sel)
begin
if(sel)
  out <= a+b;
else
  out <= a-b;

end
endmodule
```

Let’s start to construct a 32bits processor components!!

- First create a new Project! Accumulate your all files under this Project.
- Construct an adder circuit that takes 32bit 2 inputs a and b, gives a 32bit output by using given template.

```
module adder(a,b,out);



endmodule
```

```
module alu32(sum,a,b,zout,gin);//ALU operation according
output [31:0] sum;
input [31:0] a,b;
input [2:0] gin;//ALU control line
reg [31:0] sum;
reg [31:0] less; // to check less than
output zout; //zero out
reg zout;
always @(a or b or gin)
begin



end
endmodule
```

s2	s1	s0	Operation
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A + B
0	1	1	31'bx
1	0	0	31'bx
1	0	1	31'bx
1	1	0	A - B
1	1	1	SLT

```
module mux4to_1(out,sel,i0,i1,i2,i3);
output reg out;
input i0,i1,i2,i3;
input [1:0] sel;

always @ (i0 or i1 or i2 or i3 or sel)
begin

case(sel)
2'b00 : out <= i0;
2'b01 : out <= i1;
2'b10 : out <= i2;
2'b11 : out <= i3;
endcase

end

endmodule
```

ALU codes

```
module alu32(sum,a,b,zout,gin);//ALU operation according to the ALU control line values
output [31:0] sum;
input [31:0] a,b;
input [2:0] gin;//ALU control line
reg [31:0] sum;
reg [31:0] less; // to check less than
output zout; //zero out
reg zout;
always @(a or b or gin)
begin
        case(gin)
        3'b010: sum=a+b;                //ALU control line=010, ADD
        3'b110: sum=a+1+(~b);   //ALU control line=110, SUB
        3'b111: begin less=a+1+(~b);    //ALU control line=111, set on less than
                        if (less[31]) sum=1;
                        else sum=0;
                  end
        3'b000: sum=a & b;      //ALU control line=000, AND
        3'b001: sum=a|b;                //ALU control line=001, OR
        default: sum=31'bx;
        endcase
zout=~(|sum);
end
endmodule
```