# Levels of Abstraction

- **Behavioral Level :**Module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Very similar to C programming
- **Dataflow Level**: Module designed by specifying dataflow. The designer is aware of how data flows between hardware registers and how the data is processed in the design
- **Gate Level:** Module implemented in terms of logic gates like (and ,or) and interconnection between gates
- **Switch Level:** Module implemented with switches and interconnects. Lowest level of Abstraction



# Basic Unit (Module)

- A module is the basic building block in Verilog.
- In Verilog a module is declared by the keyword **"module"**.
- Elements are grouped into modules to provide the common functionality that is used at many places in the design.
- A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs).
- A corresponding keyword **"endmodule"** must appear at the end of the module definition.



# Module- Basic building block

```
module <module name> (<port list>);
    <declarations>;      // input, output, inout
                         // wire, register, etc.
    <statements>;        // initial, begin, end, always
                         // dataflow statements
endmodule
```



# Structure of a module

- The <**module name**> is an identifier that uniquely names the module.
- The <**port list**> is a list of input, inout and output ports which are used to connect to other modules.
- The <**declarations**> section specifies data objects as registers, memories and wires as wells as procedural constructs such as functions and tasks.
- The <**statements**> may be initial constructs, always constructs, continuous assignments or instances of modules.



# Typical structure of a module

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
```
→ Module Name and Ports List

```
output out;
input i0, i1, i2, i3, s1, s0;
reg out;
```
→ Declarations

```
always @ (s1, s2, i0, i1, i2, i3)
begin
    case ({s1, s0})
        2'd0: out = i0;
        2'd1: out = i1;
        2'd2: out = i2;
        2'd3: out = i3;
        default:$display ("invalid control signals");
    endcase
end
```
→ Statements

```
endmodule
```
→ End of Module

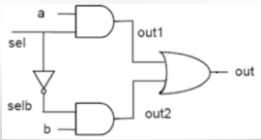


# Modules (Cont…)

- Modules CANNOT be nested.
- The process of creating objects from a module template is called **instantiation** and the objects are called **instances**.
- One module can instantiate another module.
- Module instantiation is like creating actual objects (Instances) from the common template (module definition).
- Each instance of module has all the properties of that module.
- Module instantiations are used for:
  - connecting different parts of the designs, and
  - connecting test bench to the design.

# Multiplexer 2x1


| sel | out |
|---|---|
| 0 | b |
| 1 | a |






```
1 module mux2x1_str(out,a,b,sel);
2 //declaration of input and output
3 output out;
4 input a,b,sel;
5 //declaration of reg, wire and other data types
6 wire selb,out1,out2;
7
8 //Statements describing the functionality of the design
9 //Structural model includes instantiations
10 //The order in which the modules are instantiated doesn't matter
11 and A1(out1,a,sel);
12 and A2(out2,b,selb);
13 not INV1(selb,sel);
14 or O1(out,out1,out2);
15
16 endmodule
```

Input and output declarations done outside the portlist

```
1 module mux2x1_str(output out,input a,b,sel);
2
3 wire selb,out1,out2;
4
5 and A1(out1,a,sel);
6 and A2(out2,b,selb);
7 not INV1(selb,sel);
8 or O1(out,out1,out2);
9
10 endmodule
```

Input and output declarations done inside the portlist

Module instantiations can be done in any order as the statements here will be executed concurrently

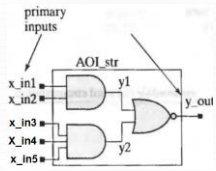


# Testbench

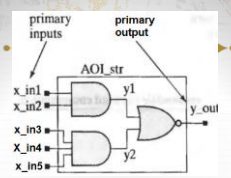
```
1 module tb_mux2x1_str;
2 //Declare all the inputs as data type reg
3 //Declare all the outputs as data type wire
4 reg a, b, sel;
5 wire out;
6
7 //Instantiate the Design Under Test (DUT)
8 //Signals in the portlist need to be in the same order as in design module
9 mux2x1_str DUT(out,a,b,sel);
10
11 //All the input signal combinations are generated
12 initial
13 begin
14              a=0;b=0;sel=0;
15          #10 a=0;b=0;sel=1;
16          #10 a=0;b=1;sel=0;
17          #10 a=0;b=1;sel=1;
18          #10 a=1;b=0;sel=0;
19          #10 a=1;b=0;sel=1;
20          #10 a=1;b=1;sel=0;
21          #10 a=1;b=1;sel=1;
22 end
23
24 endmodule
```



# AND-OR-INVERT module



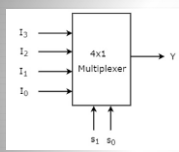



```
1 module aoi_str(y_out,x_in1,x_in2,x_in3,x_in4,x_in5);
2 output y_out;
3 input x_in1,x_in2,x_in3,x_in4,x_in5;
4
5 wire y1,y2;
6
7 nor N1(y_out,y1,y2);
8 and A1(y1,x_in1,x_in2);
9 and A2(y2,x_in3,x_in4,x_in5);
10
11 endmodule
```

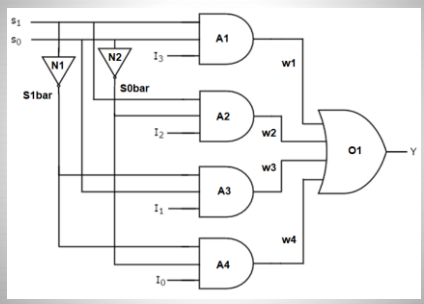


# Multiplexer 4x1




| Selection Lines | | Output |
|---|---|---|
| $S_1$ | $S_0$ | Y |
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |

# Multiplexer 4x1






# Multiplexer 4x1 - Design

```
1 module mux4x1_str(output y, input s0,s1,i0,i1,i2,i3);
2
3 wire s0bar,s1bar,w1,w2,w3,w4;
4
5 and A1(w1,s1,s0,i3);
6 and A2(w2,s1,s0bar,i2);
7 and A3(w3,s1bar,s0,i1);
8 and A4(w4,s1bar,s0bar,i0);
9
10 not N1(s1bar,s1);
11 not N2(s0bar,s0);
12
13 or O1(y,w1,w2,w3,w4);
14
15 endmodule
```

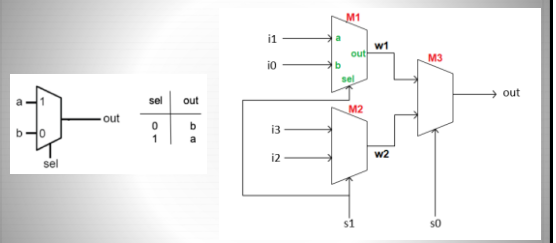


# Multiplexer 4x1 - Testbench

```
1 module tb_mux4x1_str;
2
3 wire ty;
4 reg sel0, sel1, x0,x1,x2,x3;
5
6 mux4x1_str DUT(ty, sel0, sel1, x0,x1,x2,x3);
7
8 initial
9 begin
10   sel0=0; sel1=0; x0=0; x1=0; x2=0; x3=0;
11   #10;
12   sel0=0; sel1=0; x0=0; x1=0; x2=0; x3=1;
13   #10;
14   sel0=0; sel1=0; x0=0; x1=0; x2=1; x3=0;
15   ......
16   ......
17   ......
18   ......
19   ......
20 end
21
22 endmodule
```



# Mux 4x1 using Mux 2x1






# Design

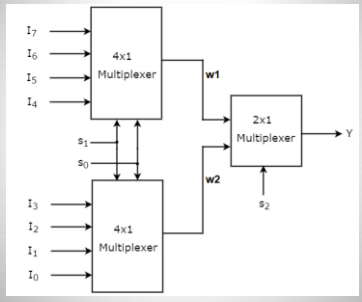
```
1 module mux4x1_2x1_str(output out, input s0,s1,i0,i1,i2,i3);
2
3 wire w1,w2;
4
5 //port connection using ordered list
6 mux2x1_str M1(w1,i1,i0,s1);
7
8 //Port connection using signal names where order does not matter
9 mux2x1_str M2(.out(w2), .a(i3), .b(i2), .sel(s1));
10 mux2x1_str M3(.sel(s0), .a(w2), .out(out), .b(w1));
11
12 endmodule
```



# Testbench

```
1 module tb_mux4x1_2x1_str;
2
3 wire ty;
4 reg sel0, sel1, x0,x1,x2,x3;
5
6 mux4x1_2x1_str DUT(ty, sel0, sel1, x0,x1,x2,x3);
7
8 initial
9 begin
10   sel0=0; sel1=0; x0=0; x1=0; x2=0; x3=0;
11   #10;
12   sel0=0; sel1=0; x0=0; x1=0; x2=0; x3=1;
13   #10;
14   sel0=0; sel1=0; x0=0; x1=0; x2=1; x3=0;
15   ......
16   ......
17   ......
18   ......
19   ......
20 end
21
22 endmodule
```

# Mux 8x1 (Homework)





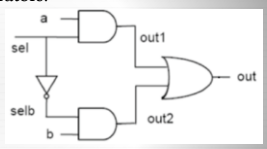

# Dataflow Modeling

- Combinational circuits are described by their function rather than by their gate structure.
- Uses a number of operators that act on operands to produce the desired results.
- Uses continuous assignments and the keyword **assign.**
- A continuous assignment is a statement that assigns a value to a net.
- The datatype net is used in Verilog HDL to represent a physical connection between circuit elements.



- The value assigned to the net is specified by an expression that uses operands and operators.


| sel | out |
|---|---|
| 0 | b |
| 1 | a |




$$out = a.sel + b.\overline{sel}$$



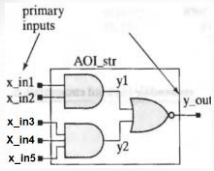
```
1 //Structural modelling of 2x1 Mux
2 module mux2x1_str(output out,input a,b,sel);
3
4 wire selb,out1,out2;
5
6 and A1(out1,a,sel);
7 and A2(out2,b,selb);
8 not INV1(selb,sel);
9 or O1(out,out1,out2);
10
11 endmodule
```

```
1 //Dataflow modelling of 2x1 Mux
2 module mux2x1_df(output out,input a,b,sel);
3
4 assign out = (a & sel) | (b & ~sel);
5 //This can be modelled using ternary operator also
6 //assign out = sel ? a : b;
7
8 endmodule
```

- Left Hand side of the assign statement should be of type wire
- By default output signals are of type wire



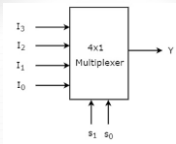
# AOI – Dataflow modelling




```
assign y_out = ~((x_in1 & x_in2) | (x_in3 & x_in4 & x_in5));
```



# 4x1 Multiplexer (Dataflow)




```
assign y = (s1 & s0 & i3) | (s1 & ~s0 & i2) | (~s1 & s0 & i1) | (~s1 & ~s0 & i0);
```

**OR**

Using Conditional operator

```
assign y = so ? (s1 ? i1:i0) : (s1?i3:i2);
```

# Behavioral Modelling

- Behavioral modeling represents digital circuits at a functional and algorithmic level.
- Behavioral description use the keyword **always** followed by a list of procedural assignment statements.
- The target output of procedural assignment statement must be of the **reg** data type.



```
1 //Behavioral modelling of 2x1 Mux
2 module mux2x1_beh(output reg out,input a,b,sel);
3
4 always @ (a or b or sel)
5 begin
6         if(sel)
7                 out = a;
8         else
9                 out = b;
10 end
11
12
13
14 endmodule
```

**Homework**

Behavioral Modelling of Mux 4x1 and Mux 8x1