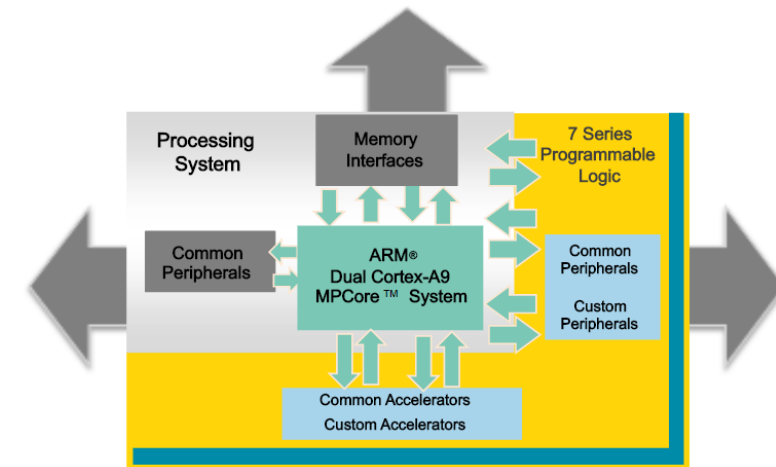
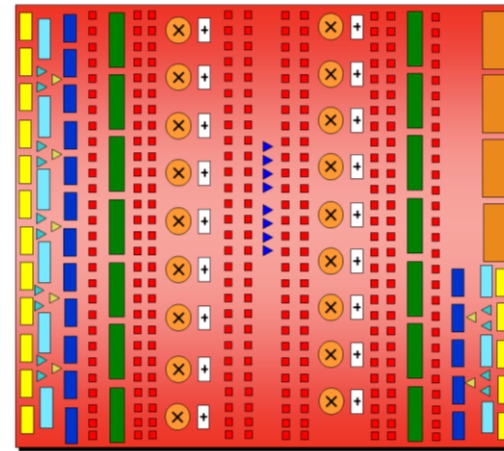




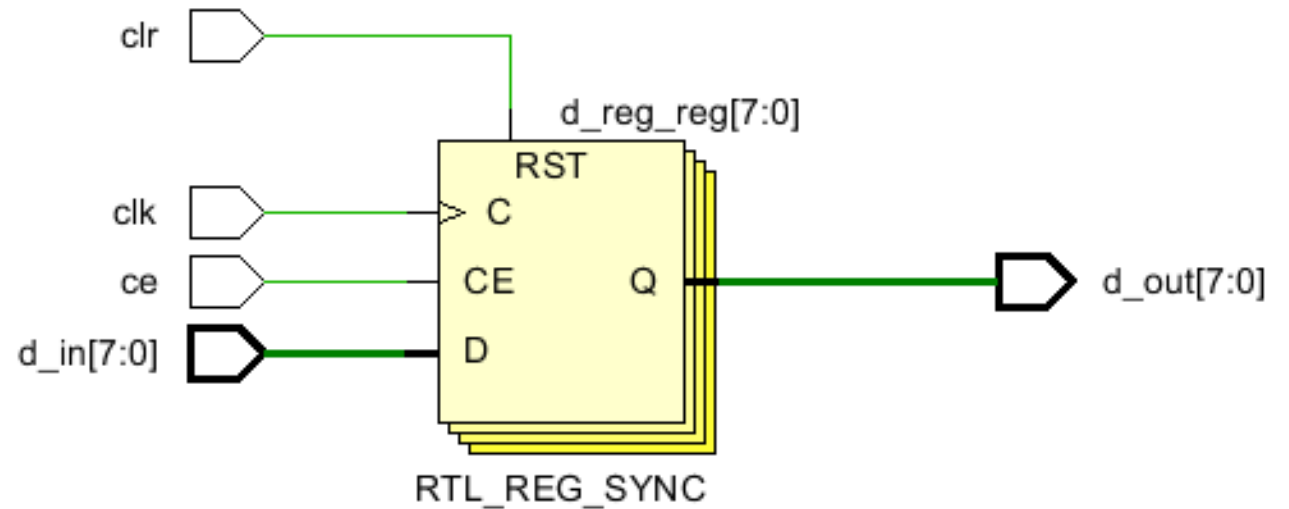
# ECE 270: Embedded Logic Design



# Verilog: Register

```
module test_1(  
    input [7:0] d_in,  
    input ce,  
    input clk,  
    input clr,  
    output [7:0] d_out  
);
```

```
    reg [7:0] d_reg;  
    always@(posedge clk)  
    begin  
        if(clr)  
            d_reg <= 8'b00000000;  
        else if (ce)  
            d_reg <= d_in;  
    end  
    assign d_out = d_reg;  
endmodule
```



always@ (posedge clk)  
→ d\_reg ← 4'b0000;

# Register and Wire

# Module Ports

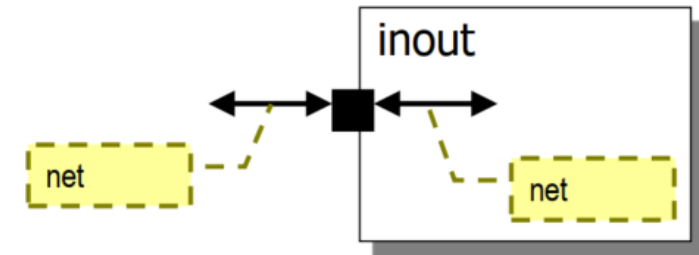
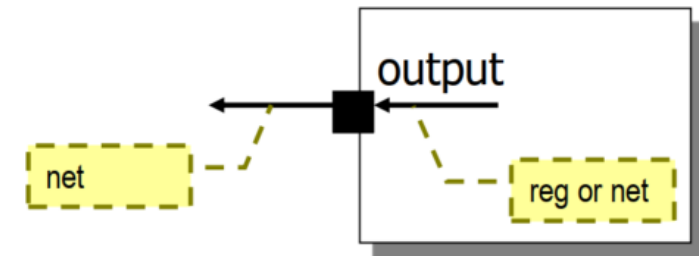
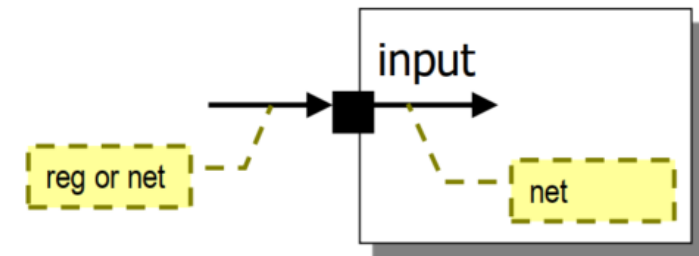
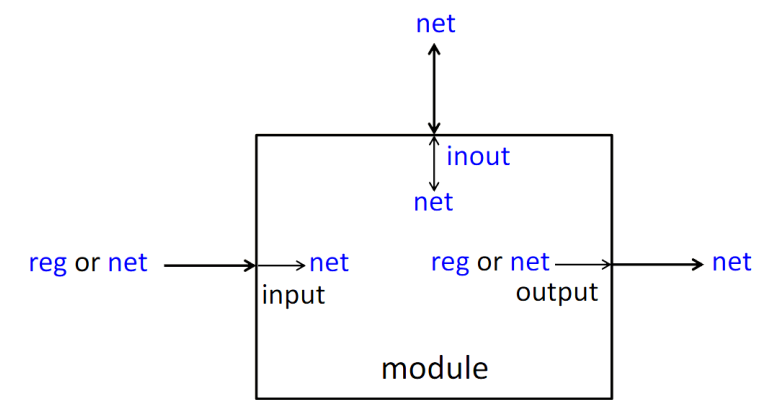
# Verilog: Module Ports

- Ports provide **interface** for the module to **communicate** with its environment
- Declaration: **<Port direction> <width> <port\_name>;**
- Port **direction** can be *input, output, inout*.

```
module my_module (my_input_port, my_inout_port,  
                  my_output_port );  
    input [4:0] my_input_port ;  
    inout  my_inout_port ;  
    output wire (or reg) [14:0] my_output_port ;  
endmodule
```

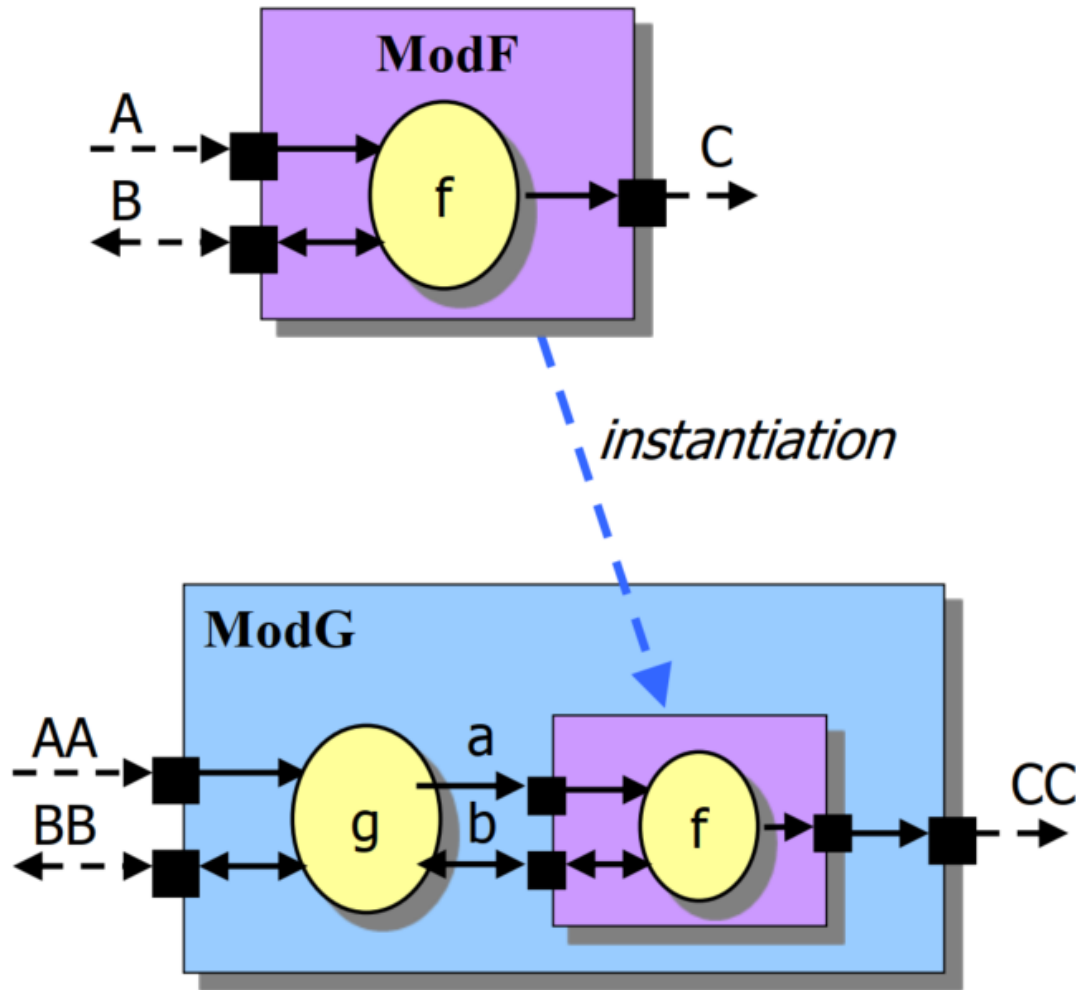
# Verilog: Module Ports

- An **input port** specifies an internal name for a vector or scalar, **driven by external entity**.
- An **output port** specifies an internal name for a vector or scalar, **driven by internal entity**, available external to the module.
- An **inout port** specifies an internal name for a vector or scalar **driven either by an internal or external entity**.
- Input or inout ports **cannot** be declared as of type **register**.
- Port is always considered as **net**, unless declared elsewhere as **reg** (only for output port)



# Module Interconnections

# Verilog: Module Interconnections



```
module ModG (AA, BB, CC);  
    input      AA;  
    inout [7:0] BB;  
    output [7:0] CC;  
    wire      a;  
    wire [7:0] b;  
    // description of 'g'  
    ModF Umodgf(.A(a), .B(b), .C(CC));  
endmodule
```

Module name

Instance name

Port connection

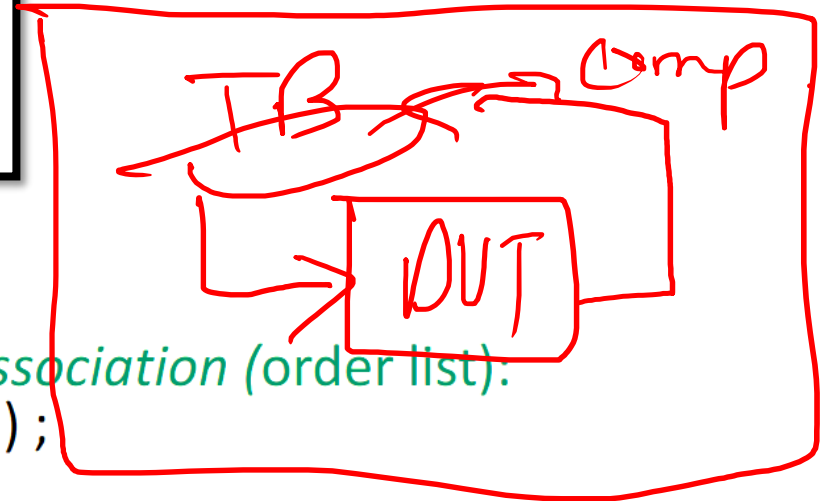


# Verilog: Module Interconnections

- Ports of the instances could be connected by name or by order list.

```
module fa_tb ;  
  reg [3:0] A, B ;  
  reg CIN ;  
  wire [3:0] SUM ;  
  wire COUT ;  
endmodule
```

```
module FA4 (sum, cout, a, b, cin) ;  
  output wire [3:0] sum ;  
  output wire cout ;  
  input [3:0] a, b ;  
  input cin ;  
endmodule
```



// Instantiate/connect by *Positional association* (order list):


```
FA4 fa_byorder (SUM, COUT, A, B, CIN) ;
```

// Instantiate/connect by *Named association* (port name):

```
FA4 fa_byname (.cout(COUT), .sum(SUM), .b(B), .cin(CIN), .a(A) ;
```


# Verilog: Module Interconnections

```
module topmod;  
  wire [4:0] v;  
  wire a,b,c,w;  
  modB b1 (v[0], v[3], w, v[4]);  
endmodule
```



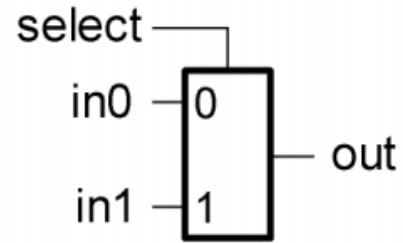
```
module modB (wa, wb, c, d);  
  inout wa, wb;  
  input c, d;  
  tranif1 g1 (wa, wb, cinvert);  
  not #(2, 6) n1 (cinvert, int);  
  and #(6, 5) g2 (int, c, d);  
endmodule
```

```
module topmod;  
  wire [4:0] v;  
  wire a,b,c,w;  
  modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));  
endmodule
```

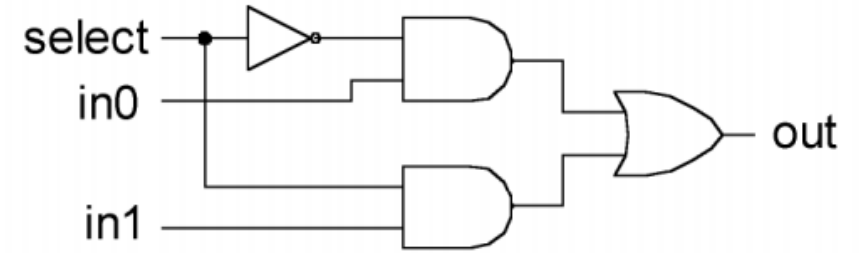


```
module modB (wa, wb, c, d);  
  inout wa, wb;  
  input c, d;  
  tranif1 g1 (wa, wb, cinvert);  
  not #(6, 2) n1 (cinvert, int);  
  and #(5, 6) g2 (int, c, d);  
endmodule
```

# Multiplexer



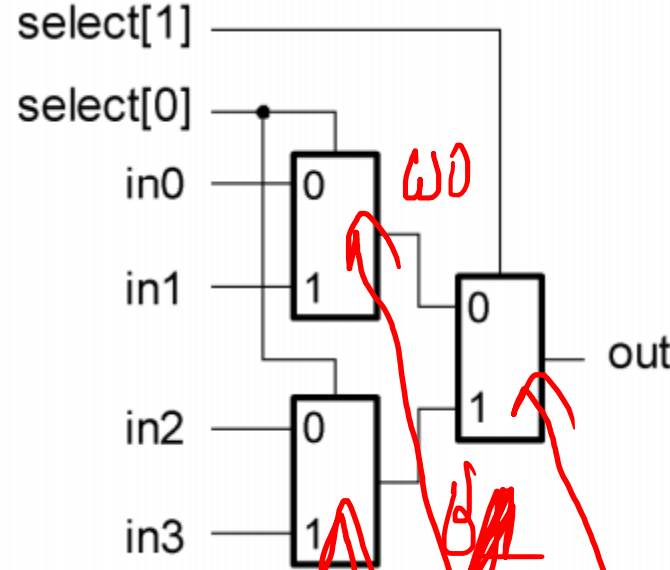
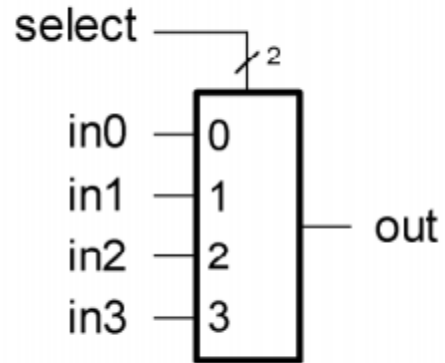
a) 2-input mux symbol



b) 2-input mux gate-level circuit diagram

```
module mux2 (in0, in1, select, out);  
  input in0, in1, select;  
  output out;  
  wire s0, w0, w1;  
  
  not (s0, select);  
  and (w0, s0, in0),  
      (w1, select, in1);  
  or (out, w0, w1);  
  
endmodule
```

# Multiplexer



```
module mux2 (in0, in1, select, out);
    input in0, in1, select;
    output out;
    wire s0, w0, w1;

    not (s0, select);
    and (w0, s0, in0),
        (w1, select, in1);
    or (out, w0, w1);

endmodule
```

```
module mux4 (in0, in1, in2, in3, select, out);
    input in0, in1, in2, in3;
    input [1:0] select;
    output out;
    wire w0, w1;

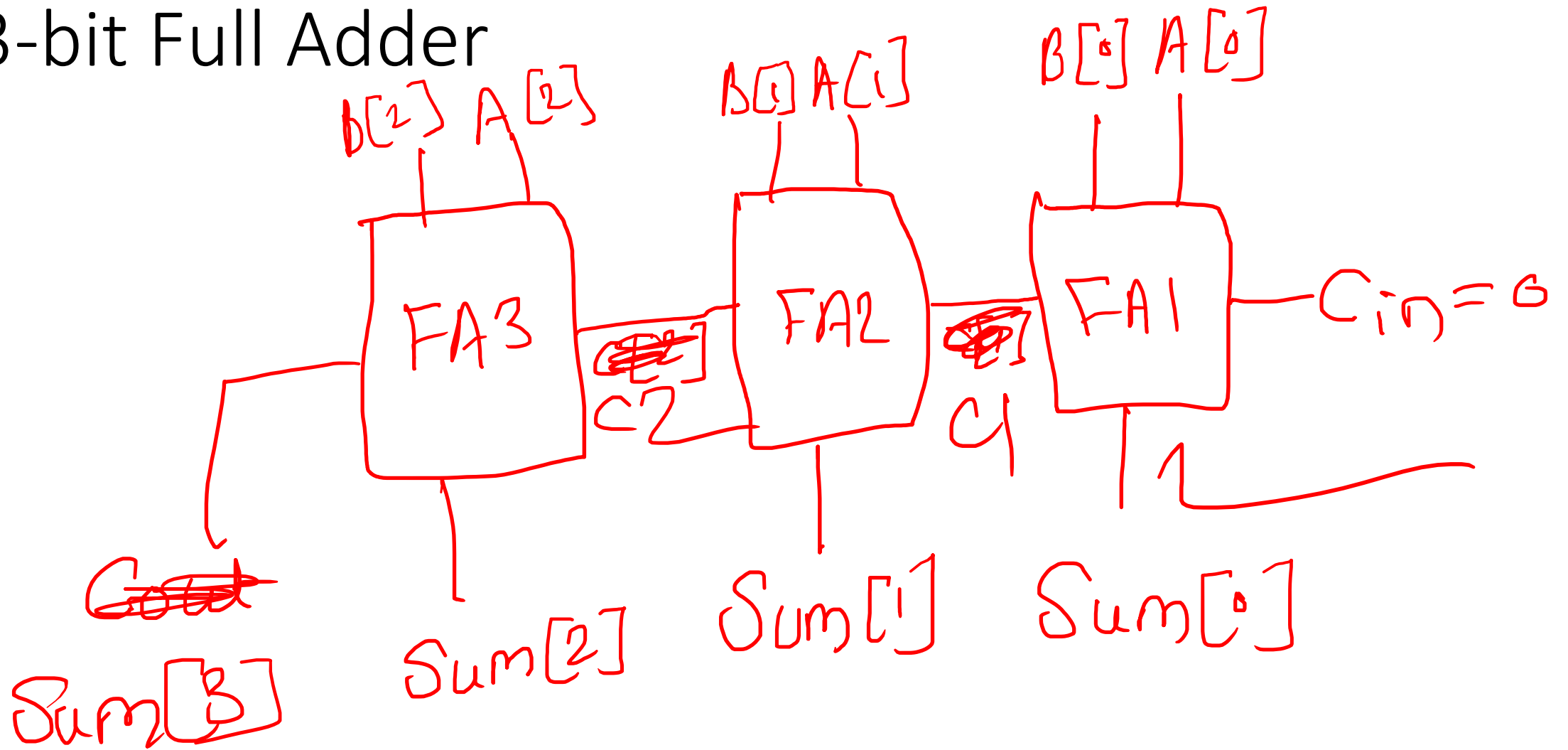
    mux2
        mo (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
        m2 (.select(select[0]), .in0(in2), .in1(in3), .out(w1));
    mo (.select(select[1]), .in0(w0), .in1(w1), .out(out));

endmodule
```

If your design contains more than one module, put each in a separate file.

**Instantiation:** Prefer named association. It prevents incorrect connections for the ports of instantiated components.

# 3-bit Full Adder



# 3-bit Full Adder

```
→ module top_adder(  
→   → input [2:0] A,  
→   → input [2:0] B,  
→   → output [3:0] Sum  
→   → );  
→   → wire c1,c2;  
→   →  
→   → fa.in1(.A(A[0]),.B(B[0]),.C(1'b0),.Sum(Sum[0]),.Carry(c1));  
→   → fa.in2(.A(A[1]),.B(B[1]),.C(c1),.Sum(Sum[1]),.Carry(c2));  
→   → fa.in3(.A(A[2]),.B(B[2]),.C(c2),.Sum(Sum[2]),.Carry(Sum[3]));  
→ endmodule
```

```
→ module fa(  
→   → input A,  
→   → input B,  
→   → input C,  
→   → output Sum,  
→   → output Carry  
→   → );  
→   →  
→   → assign Sum = A^B^C;  
→   → assign Carry=((A^B)&C)|(A&B);  
→ endmodule
```

# 3-bit Multiplier (Self Study)

```
→ module top_multiplier(  
→   input [2:0] A,  
→   input [2:0] B,  
→   output [5:0] Mul_Op  
→ );  
→  
→ wire c1,c2,c3,c22,c32;  
→ wire s1,s2;  
→ assign Mul_Op[0]=A[0] & B[0];  
→ fa.in1(.A(A[0] & B[1]),.B(A[1] & B[0]),.C(1'b0),.Sum(Mul_Op[1]),.Carry(c1));  
→ fa.in2(.A(A[2] & B[0]),.B(A[1] & B[1]),.C(c1),.Sum(s1),.Carry(c2));  
→ fa.in3(.A(A[0] & B[2]),.B(s1),.C(1'b0),.Sum(Mul_Op[2]),.Carry(c22));  
→ fa.in4(.A(A[2] & B[1]),.B(1'b0),.C(c2),.Sum(s2),.Carry(c3));  
→ fa.in5(.A(A[1] & B[2]),.B(s2),.C(c22),.Sum(Mul_Op[3]),.Carry(c32));  
→ fa.in6(.A(A[2] & B[2]),.B(c3),.C(c32),.Sum(Mul_Op[4]),.Carry(Mul_Op[5]));  
→ endmodule
```

```
→ module fa(  
→   input A,  
→   input B,  
→   input C,  
→   output Sum,  
→   output Carry  
→ );  
→  
→ assign Sum = A ^ B ^ C;  
→ assign Carry = ((A ^ B) & C) | (A & B);  
→ endmodule
```

# Homework

- Using module for 2:1 mux (**data flow level** approach), design 8:1 mux via module interconnections
- Design comparator for 2-bit inputs using **data flow level** approach.
- Using comparator for 1-bit inputs, design comparator for 2-bit inputs via module interconnections



# Number Representation

# Verilog: Number Representation

- Verilog HDL allows integer numbers to be specified as: Sized or Unsized numbers ( Unsized is 32 bits)
- In a radix of **binary**, **octal**, **decimal**, or **hexadecimal**
- Syntax: **<size> '<radix> <value>**  
size in bits, radix in b, d, o, h
- **Spaces are allowed** between the size, radix and value
- **Underscore character ( \_ )** is ignored and can be used to enhance readability. It cannot be the first character in number.

# Verilog: Number Representation

549 // unsized decimal number

'h 8FF // unsized hex number

'o765 // unsized octal number

4'b11 // 4-bit binary number 0011

3'b10x // 3-bit binary number with LSB unknown

5'd3 // 5-bit decimal number

# Verilog: Number Representation

```
792 // a decimal number  
8d9 // Illegal, hexadecimal must be specified with 'h'  
'h 7d9 // an unsized hexadecimal number - 000007d9  
'o 7746 // an unsized octal number - 00000007746  
1 // stored as 000000000000000000000000000000000001  
  
12 'h x // a 12 bit unknown number  
10 'd 17 // a 10 bit constant with the value 17  
4 'b 110z // a 4 bit binary number  
8'hAA // stored as 10101010
```

# Verilog: Negative Numbers

- Any number that does not have negative sign prefix is a **positive number**. Or indirect way would be "**Unsigned**"
- Negative numbers can be specified by putting a minus sign before the size for a constant number, thus become signed numbers.
- Verilog internally represents negative numbers in **2's compliment format**.

`-4'b11` // 4-bit two's complement of 0011 = 1101 = 4'hd

`4'd-2` // Illegal specification

`-5'ha` // stored as 10110

`-4'b101` // stored as 1011

# Verilog: Number Representation

number	stored value	comment
5'b11010	11010	_ ignored
5'b11_010	11010	
5'o32	11010	
5'h1a	11010	
5'd26	11010	
5'b0	00000	0 extended
5'b1	00001	0 extended
5'bz	zzzzz	z extended
5'bx	xxxxx	x extended
5'bx01	xxx01	x extended
-5'b00001	11111	2's complement of 00001
'b11010	000000000000000000000000000000011010	extended to 32 bits
'hee	000000000000000000000000000011101110	extended to 32 bits
1	000000000000000000000000000000000001	extended to 32 bits
-1	111111111111111111111111111111111111	extended to 32 bits

# Verilog: Register vs. Integer Numbers

- The **reg** register is a **1-bit** wide data type. If more than one bit is required then **range declaration** should be used (next slide).
- The **integer** register is a **32-bit** wide data type.
- Integer declarations **cannot** contain range specification.

```
module test_1(  
    input [3:0] in_1,  
    input [3:0] in_2,  
    input sel,  
    output integer [3:0] out_1  
);
```

Error: Cannot have packed dimensions of type integer

- Typically used for **constants or loop variables** in Verilog.
- Vivado will automatically trim any unused bits in integer. For e.g., if we declare an integer with a value of 255 then it will be trimmed to 8 bits.

# Verilog: String

- A sequence of characters enclosed in double quotes ("").
- Must be contained in a single line
- Considered as a **sequence of one-byte (8-bit) ASCII values**
- String variables should be declared as **reg** type vectors

```
reg [8*12:0] string_1;  
string_1="Hello Verilog";  
#10 string_1="Hello";  
#10 string_1="I am overloaded";
```

Accept C-like escape characters: \n = newline, \t = tab,  
\b = backslash, \" = quote mark ("), %% = % sign



# Verilog: String

```
reg [8*12:0] string_1;
```

```
string_1="Hello Verilog";
```

```
#10 string_1="Hello";
```

```
#10 string_1="I am overloaded";
```

```
> string_1[96:0] m overlo  
> string_1[96:0] 1011011001100101011011000110110001101111001000000101011001100101011100100
```

[illegible]

string\_1[96:0] m overloaded

string\_1[96:0] 1011011010010000001101111011101100110010101110010011011000110111101100001011001000110010101100100

# Verilog: String (Self Study)

- **reg** [8\*10:1] s1, s2;

```
reg [8*14:1] s3;
```

```
reg [8*14:1] s4;
```

```
s1 = "Verilog";           // s1="000Verilog"
s2 = "-HDL";              // s2="000000-HDL"
s3= {s1,s2};              //s3= "ilog000000-HDL"
s4= {"Verilog", "-HDL"};  // s4= "000Verilog-HDL"
```

[illegible]

# Verilog: String (Self Study)

```
// string variable declaration
reg [8*12:1] stringvar; // 8*12 or 96-bit wide
initial begin
    stringvar="Helloworld!";
end

// string manipulation
module string_test;
    reg [8*14:1] stringvar;
    reg [8*5:1] shortvar;
    initial begin
        stringvar = "Hello world";
        $display("%s is stored as %h", stringvar,stringvar);
        stringvar = {stringvar,"!!!"};
        $display("%s is stored as %h", stringvar,stringvar);
        shortvar = "Hello world";
        $display("%s is stored as %h", shortvar,shortvar);
    end
endmodule

// the output
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
world is stored as 776f726c64
```

# Verilog: Real Numbers (Self Study)