EE2026 Digital Design

MODELING STYLES IN VERILOG

Chua Dingjuan elechuad@nus.edu.sg

Lab 1 Debrief…

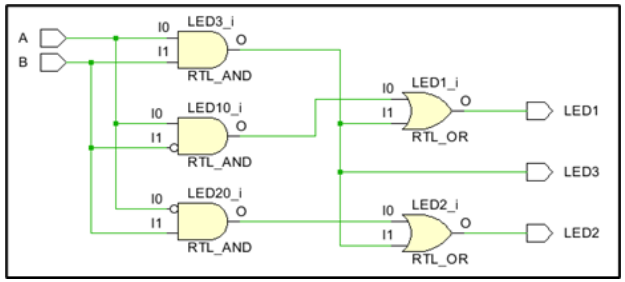
In Lab 1… Design Workflow!

idea!

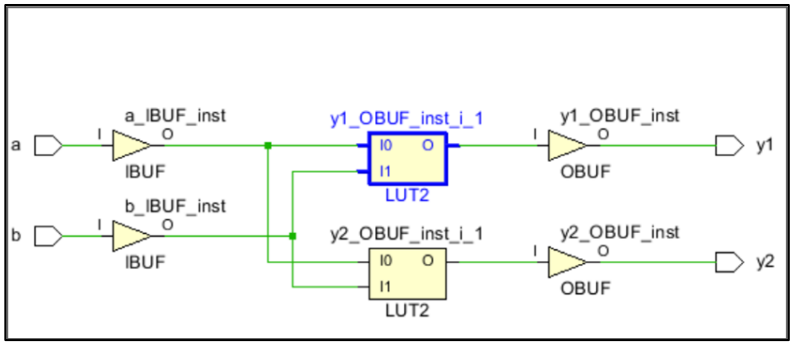
.v (verilog)

```
assign LED1 = (A & ~B) | (A & B);
```

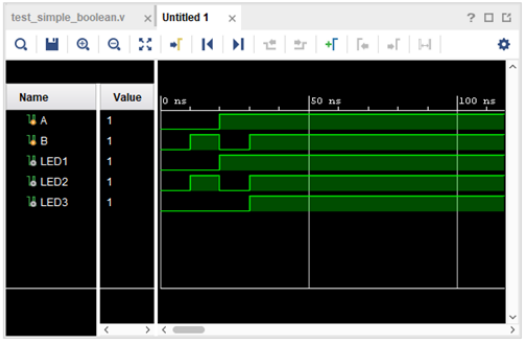
RTL Schematic



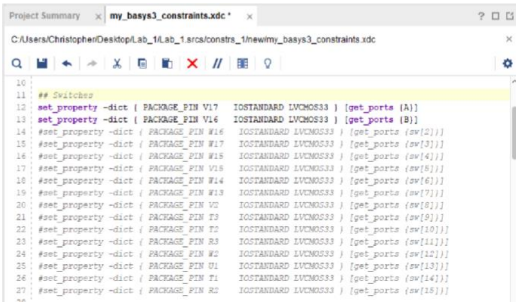
Synthesized Schematic



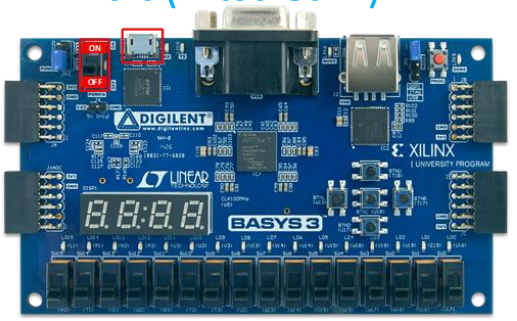
Simulation Results



.xdc (Constraints Mapping)



.bit (Bitstream)



Bad Example 1 – Multi-Driven Net

```
module notgood(….);

    assign x = a | b;
    assign x = a + b;

endmodule
```

There are two conflicting instructions presented!

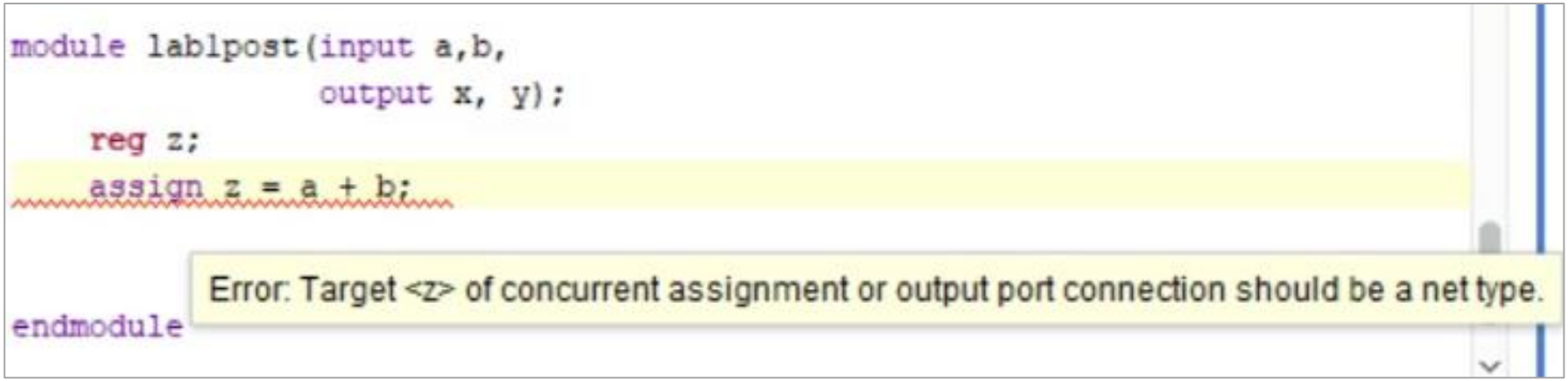
Signal x is connected to TWO drivers. Thus, the Multiple Driver Nets error would occur.

Bad Example 2 – Multi-Driven Net

```
module notgood
(input a,b);

  reg z;
   assign z = a | b;

endmodule
```

Assign statements are used for wire or net types.

Thus error states that <z> signal should be a net type!



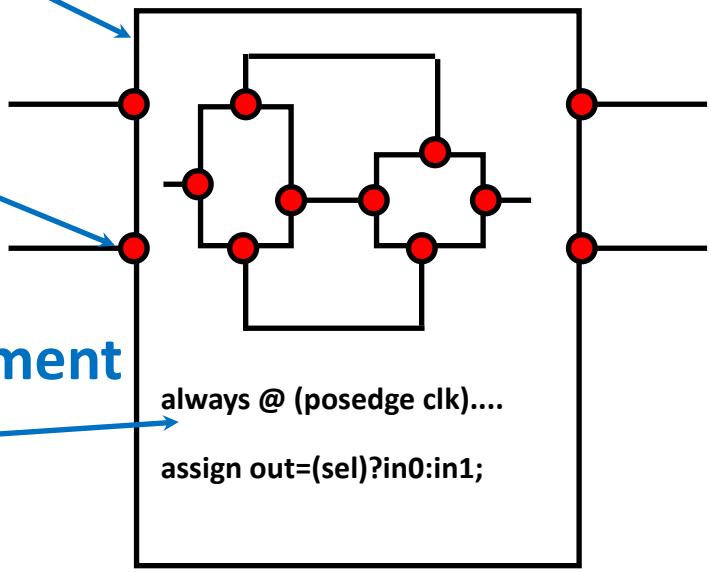
Pictorial Summary of Module Structure

Module

I/O Ports
interfaces

Continuous assignment
Concurrent blocks
Instances

functionality



Covers different levels of abstraction

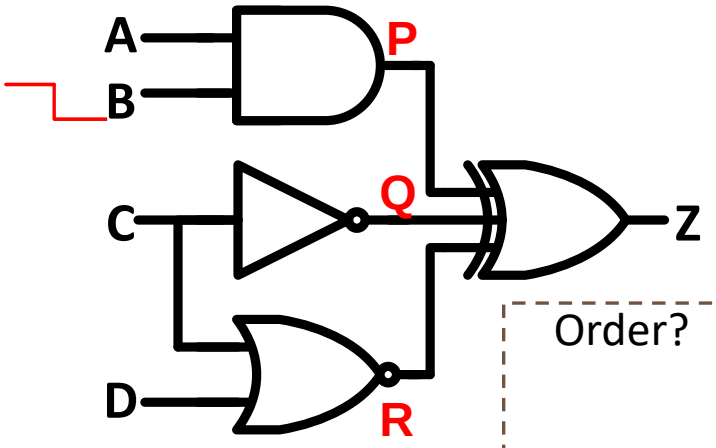
Dataflow vs
Behavioral vs
Structural description

Design partitioning

manages hierarchy (sub-modules), enables modularity (reusable) and regularity (many times)

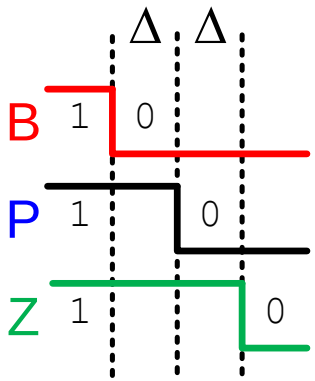
Continuous Assignment (Dataflow)

`assign` statements are often used to model combinational logic



Order?

```
module bigbox (input a, b, c, d,
               output z);
wire p, q, r;        internal connections

assign z = p ^ q ^ r; 1
assign q = ~c;        2
assign p = a & b;     3
assign r = ~(c | d);  4

endmodule
```



- Whenever there's an ***event*** on the RHS signal, expression is evaluated and assigned (Δ) ➔ *continuously monitored*
- Multiple statements can be executed in parallel (concurrently)
- **`wire`** is used to represent an internal connection

Useful Operators

- Boolean (bit-wise), logical, arithmetic, concatenation.
- Use brackets for readability, take note if **synthesizable*.

High

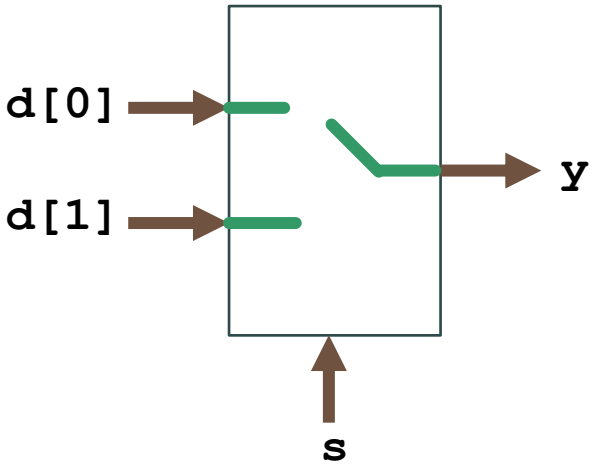
Precedence

Low

Operator	Description	Examples: a = 4'b1010, b=4'b0000
!, ~	Logical negation, Bit-wise NOT	**!a = , !b = , ~a=4'b , ~b=4'b**
&, |, ^	Reduction (Outputs 1-bit)	**&a = 0, |a=1, ^a = 0**
{__,__}	Concatenation	{b, a} = 8'b00001010
{n{____}}	Replication	{2 {a} } = 8'b10101010
*, /, %,	Multiply, *Divide, *Modulus	3 % 2 = 1, 16 % 4 = 0
+, -	Binary addition, subtraction	a + b = 4'b1010
<< , >>	Shift Zeros in Left / Right	a << 1 = 4'b0100, a >> 2 = 4'b0010
<, <=, >, >=	Logical Relative (1-bit output)	**(a > b) =**
==, !=	Logical Equality (1-bit output)	**(a == b)= (a != b)=**
&, ^, |	Bit-wise AND, XOR, OR	**a&b = a|b =**
&&, ||	Logical AND, OR (1-bit output)	**a&&b = a||b =**
?:	Conditional Operator	<out> = <condition> ? If_ONE : if_ZERO

Conditional Operator…

The `?:` conditional operator allows us to select the output from a set of inputs based on a condition.




```
module mystery ( input s, input [1:0] d,
                 output y);

assign y = s ? d[1] : d[0];

endmodule
```

<output> = <condition> ? If_ONE : If_ZERO

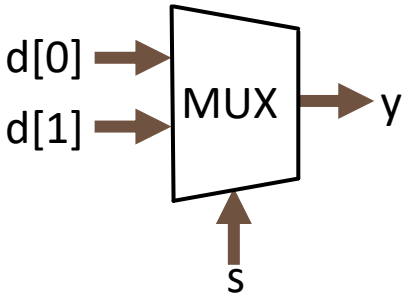
s= 1 ➔ y = d[1], s= 0 ➔ y= d[0]

- This expression is evaluated whenever there is an event on any input.
- What is this block?

Procedural Assignment : `always`

Behavioral, higher-level description of logic.

2 assignment types : **Blocking** & **Non-Blocking**




```
module mux21( input s, input d[1:0],
              output reg y);

always @ (s, d)

begin
   if (sel == 1'b0)
       y= d[0];
   else
       y= d[1];

end
endmodule
```

Anything assigned in an **`always`** block must be declared as type **`reg`**

Conceptually, the **`always`** block runs o*nce* when any signal in *sensitivity list* (s,d) changes value.

Statements executed sequentially & evaluated instantaneously. ➔ Order matters!

`begin` and **`end`** behave like parentheses/brackets for conditional statements.

Some notes on: `always`

- **`always@(*)`** includes all signals that are read in statements.

- Statements within **`always`** block are executed ***sequentially***.
- Variables within sensitivity list are very important!
- **`if`**--**`else if`**--**`else`**, **`case`**, **`for`**, **`while`** can only be used in procedural assignments (always blocks)
- Multiple always blocks run in parallel, concurrently, watch out for multi-driven nets and race conditions. (next slide)
- No ~~**`assign`**~~ in always blocks!

Registers

- Anything assigned in an **`always`** block must be declared as type **`reg`**
- In Verilog, the term register (**`reg`**) simply means a variable that can hold a value. (cf. **`wire`** which must be driven)

Bad Example – Multi-Driven Net

```
module notgood(….);
   always @ (*)
      y = y + 1;

   always @ (*)
      y = y + 3;

endmodule
```

There are two conflicting instructions presented.

The first instruction would be to increment y by 1, while the second instruction would be to increment y by 3. The compiler would not be able to know which instruction should be executed.

Thus, the Multiple Driver Nets error would occur.

Ref – Conditional Statements

If - else if - else

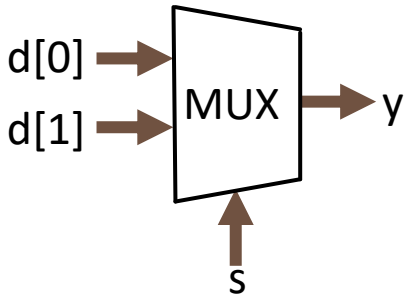
```
if ( expr )
   statement;


if ( expr )
   statement;
else
   statement;


if ( expr )
   statement;
else if ( expr )
   statement;
else if ( expr )
   statement;
else
   statement;
```

case

```
case ( expr )

   value1 : statement;
   value2 : statement;
   value3 : statement;
   ...
   default : statement;

endcase
```

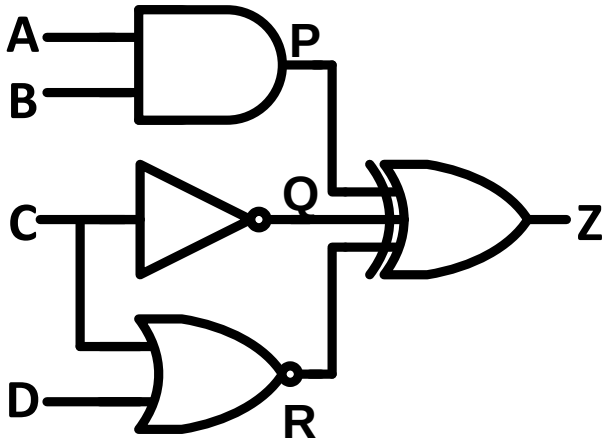
Equivalence

```
module mux21( input s, input [1:0] d, output reg y);

always @ (s, d)
begin
    if (s == 1'b0)
        y= d[0];
    else
        y= d[1];
end
endmodule
```



```
module mux ( input s, input [1:0] d, output y);

assign y = s ? d[1] : d[0];

endmodule
```

Equivalence…



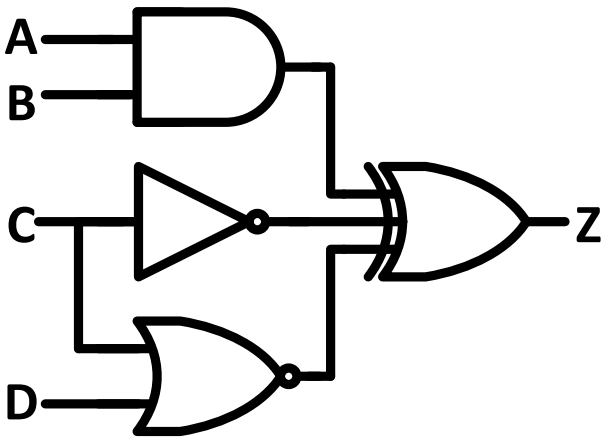

```
module bigbox
(input a,b,c,d, output z);

wire p, q, r;

assign q = ~c;
assign z = p ^ q ^ r;
assign p = a & b;
assign r = ~(c | d);

endmodule
```

```
module bigbox
(input a,b,c,d, output ___ z);

always @ (      )
   begin




   end
endmodule
```

Structural Modeling – Primitives

Structural modeling can be used to connect multiple modules and gates.

Verilog provides a standard set of primitive such as basic logic gates.



Dataflow

```
module bigbox (input a,b,c,d, output z);

   assign z = (a & b) ^ ~c ^ ~(c | d) ;

endmodule
```

Structural (using primitives)

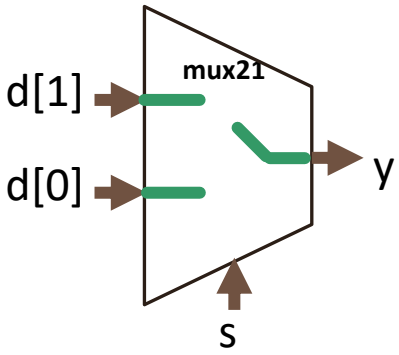
```
module bigbox (input a,b,c,d, output z);
wire p, q, r;

   and u1 (p, a, b); //output, then inputs
   not u2 (              );
   nor u3 (              );
   xor u4 (              );

endmodule
```

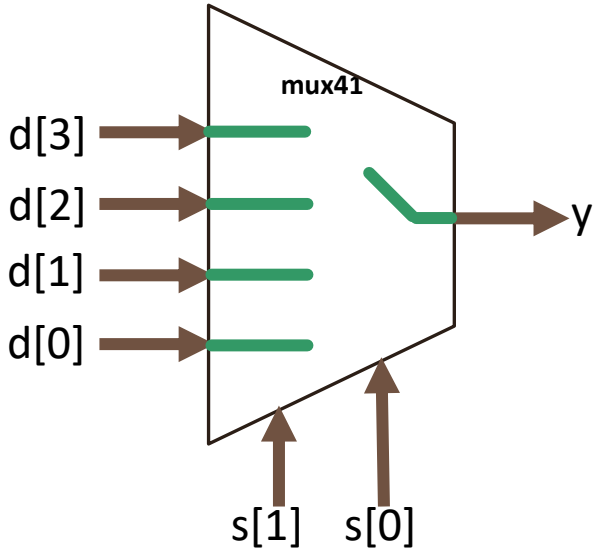
Structural Modeling

Structural modeling can be used to connect multiple modules via port connection by name and position.

```
module mux21( input s,
              input [1:0] d,
              output y);

  assign y = s ? d[1] : d[0];

endmodule
```

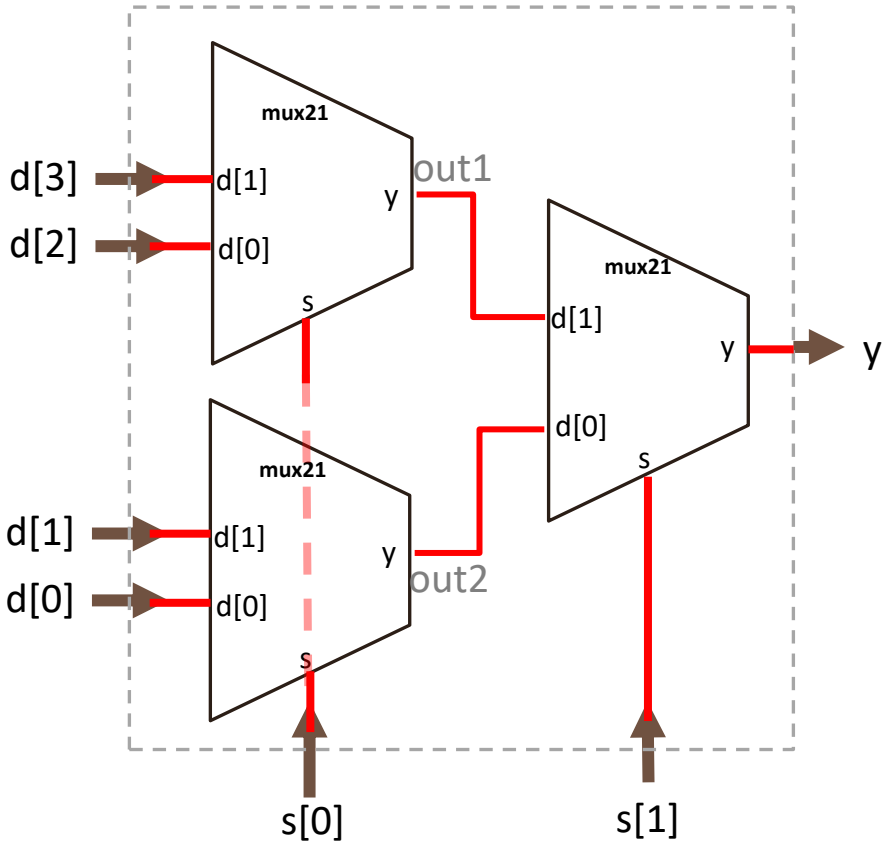
```
module mux41( input [1:0] s,
              input [3:0] d,
              output y);

… … … ?

endmodule
```





Structural Modeling

For example, a 4-to-1 multiplexer can also be implemented by combining several 2-to-1 multiplexers.



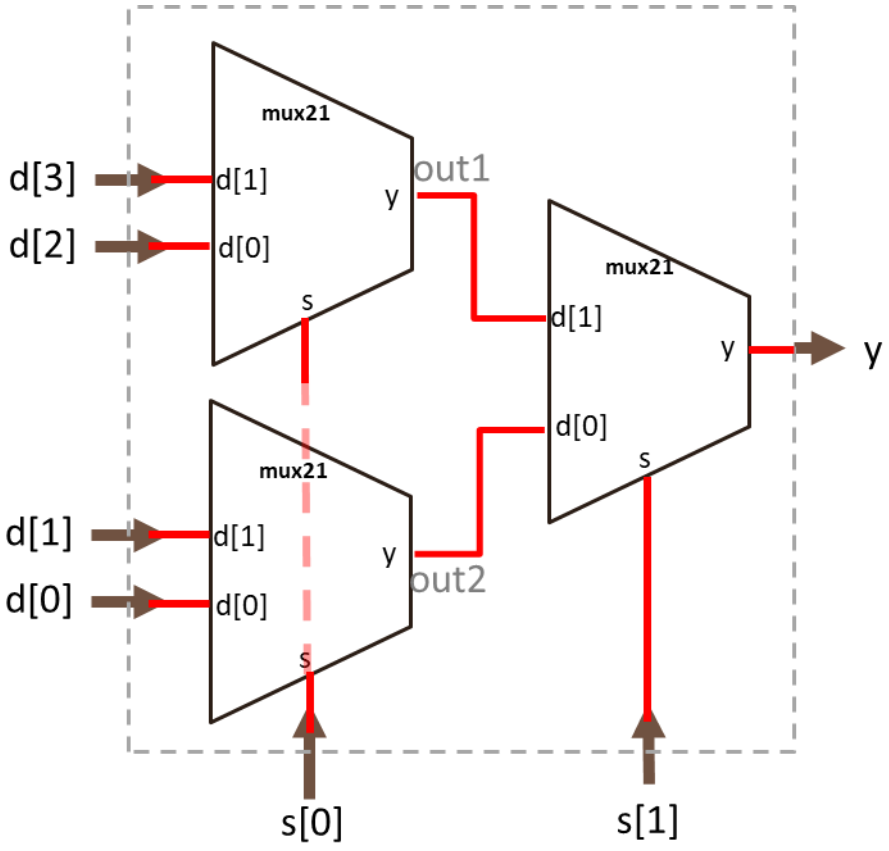

```
module mux21( input s, input [1:0] d,
              output y);
   assign y = s ? d[1] : d[0];
endmodule
```

Port Connection by Position

```
module mux41( input [1:0] s,
              input [3:0] d,
              output y);

wire out1, out2;

//check for port order!
mux21 u1 (s[0], d[3:2], out1);

mux21 u2 (                    );

mux21 u3 (                    );

endmodule
```

Structural Modeling

For example, a 4-to-1 multiplexer can also be implemented by combining several 2-to-1 multiplexers.



Port Connection by Name

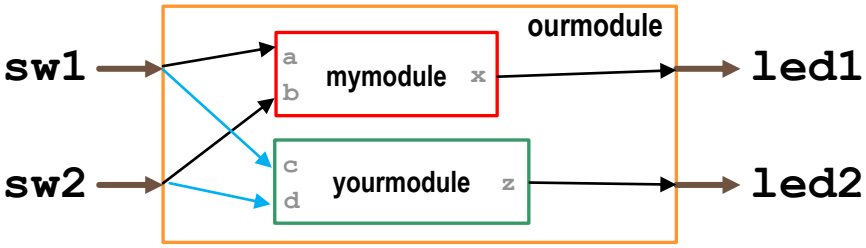
```
module mux41( input [1:0] s,
              input [3:0] d,
              output y);

wire out1, out2;

mux21 u1 (.s ( s[0]),
          .d ( d[3:2] ),
          .y ( out1 ) );

mux21 u2 (.s ( s[0]),
          .d ( d[1:0] ),
          .y ( out2) );

mux21 u3 (.s ( s[1]),
          .d ( {out1, out2} ),
          .y ( y ) );


endmodule
```

Example - Structural Modeling

- For modular designs, the top design is often specified as interconnected blocks.
- Two examples below demonstrate port connection by position / name.

```
module mymodule (input a, b, output x);
...
endmodule
```

```
module yourmodule (input c, d, output z);
…
endmodule
```



Port Connection by Position

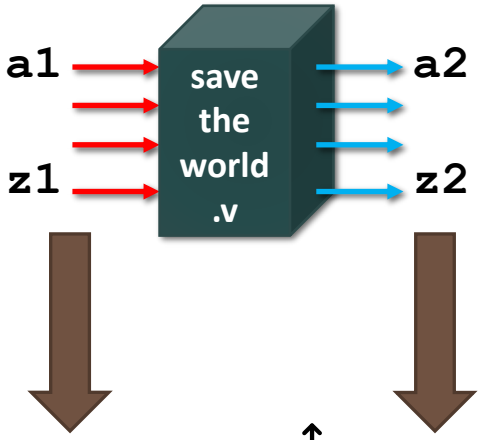
```
module ourmodule (input sw1, sw2,
                  output led1, led2);

mymodule M1 (sw1, sw2, led1);
yourmodule M2 (sw1, sw2, led2);

endmodule
```

Port Connection by Name

```
module ourmodule (input sw1, sw2,
                  output led1, led2);

mymodule M1 (  .a (sw1),
               .b (sw2),
               .x (led1) );

yourmodule M2( .z(led2),
               .c (sw1),
               .d(sw2) );

endmodule
```

Recall Simulation?

```
module savetheworld (input a1, … z1,
                     output a2, …,z2);
…….  …….  …….  …….  …….
…….  …….  …….  …….  …….
…….  …….  …….  …….  …….
endmodule
```



How do we know our design actually works?

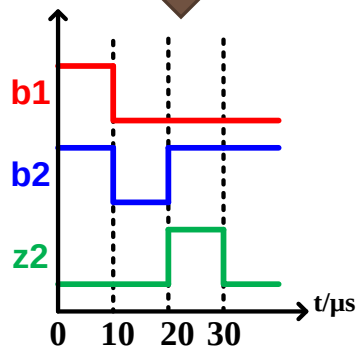
- Functional Simulation (Xilinx)

Verilog Code

```
module
…….  …….
…….  …….
endmodule
```

Test Bench

```
call module
a1=9;
b1=1;
//wait for 10u
#10
b1=0;
```



Method

- Designer applies input values to the code
- Simulator produces corresponding outputs in truth tables / timing diagrams
- Simulators usually assume negligible propagation gate delays.

Simulation Testbench Example

```
module mux21( input s,
              input [1:0] d,
              output y);
  assign y = s ? d[1] : d[0];
endmodule
```

```
module mux_test();

reg [1:0] ip = 0;
reg sel = 0;
wire op;


mux21 dut (sel, ip, op);

initial begin
   ip = 2'b10;
   sel = 1'b0;
   #10; //wait 10 time units
end

endmodule
```