EE2026 Digital Design

COMBINATIONAL LOGIC IN VERILOG

Massimo ALIOTO
Dept of Electrical and Computer Engineering
Email: massimo.alioto@nus.edu.sg

Get to know the latest silicon system breakthroughs from our labs in 1-minute video demos

Explore | Follow | Discover more | What's happening | Stay tuned | Be connected | Communicate

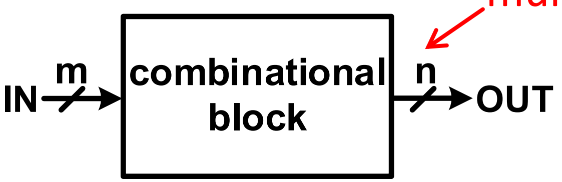
Outline

- Combinational vs. sequential logic circuits
- Review of code snippets and common errors
- Continuous assignment
- Procedural assignment
- Equivalence of continuous and procedural assignment
- Structural design
- Testbenches

Combinational vs. Sequential Logic

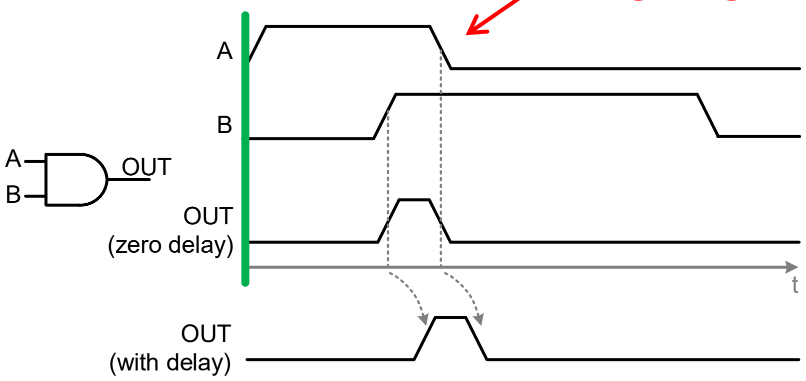
- Any digital system partitioned into combinational and sequential logic
- Combinational logic
  - Output at given time depends only on inputs at same time (apart from gate delays)
  - No memory

multi-bit (vector, bus)

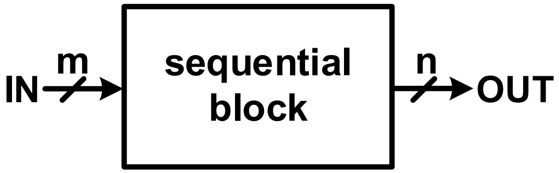


$$OUT(t_i) = F\big(IN(t_i)\big)$$

timing diagram



- Sequential logic
  - Output at given time depends on inputs at same time and past inputs
  - Includes some form of memory elements (e.g., flip-flops, registers, memory)



$$OUT(t_i) = F(IN(t_i), IN(t_{i-1}) \dots)$$

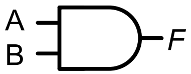
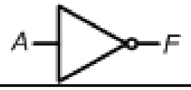
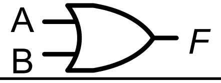
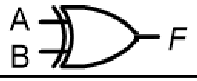
Review of Code Snippets and Common Errors

Verilog level of abstraction

```
module box(input  a, b,
           input  [1:0] c,
           output [3:0] y );
wire tmp;
reg [1:0] one = 3;
reg two;
integer three = 1;

assign y[3] = one[0];
assign y[2:1] = a + c;
assign y[0] = ( a > b );


endmodule
```

Net / Variable Name	**Number of bits?**	**Value in dec / bin**
a	1	1 / 1'b1
b	1	0 / 1'b0
c	2	2 / 2'b10
tmp	1	Z / 1'bZ
one	2	3 / 2'b11
two	1	X / 1'bX
three	32	1 / 32'h00000001
y	4	15 / 4'b1111

gate level of abstraction

Gate	Symbol	Function (F)	Verilog Operator	Gate	Symbol	Function (F)	Verilog Operator
AND		$A \cdot B$	F = A & B	NOT		$\bar{A}$	F = ~A
OR		$A + B$	F = A | B	XOR		$A \oplus B$	F = A ^ B

Boolean level of abstraction

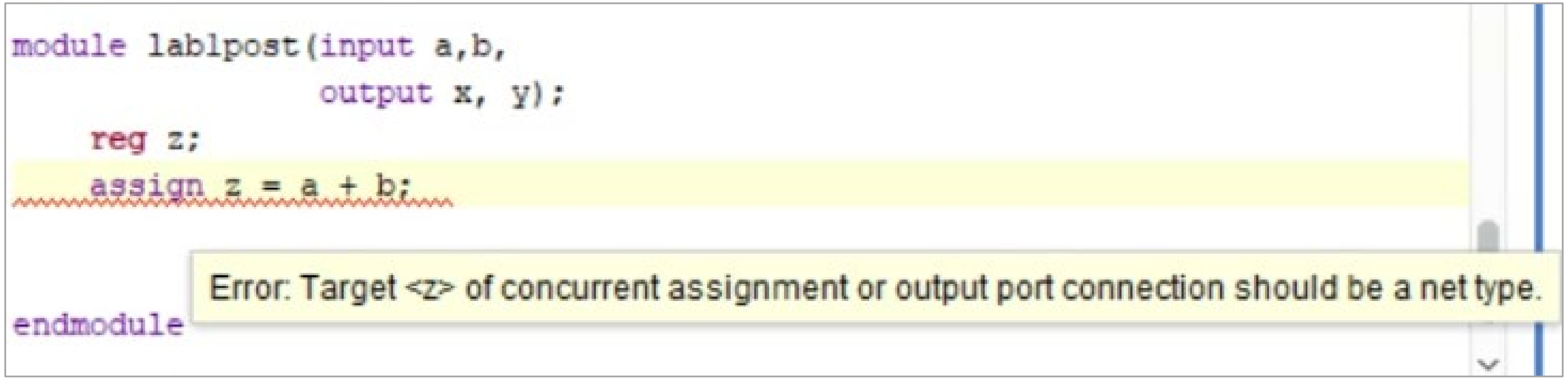
assign F = ~w & ~x & z | ~w & x & z | w & y & z | x & y & z;

Review of Code Snippets and Common Errors

```
module notgood
 (…)

    reg z;
    assign z = a | b;
endmodule
```

Assign statements are used only for wire or net types

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



```
module lab1post(input a,b,
                output x, y);
    reg z;
    assign z = a + b;


endmodule
```

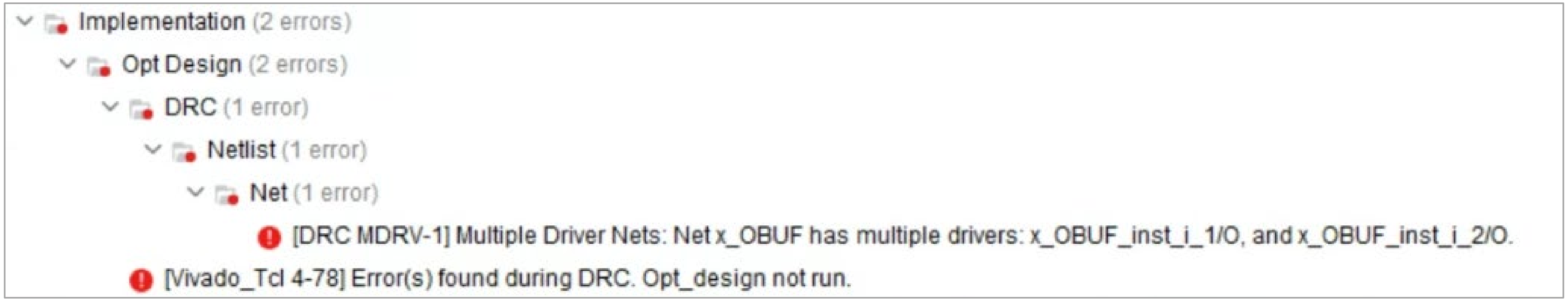
Error: Target <z> of concurrent assignment or output port connection should be a net type.

Review of Code Snippets and Common Errors

```
module notgood(….);
 (…)
    assign x = a | b;
    assign x = a + b;

endmodule
```

There are two conflicting instructions (multi-driven net)

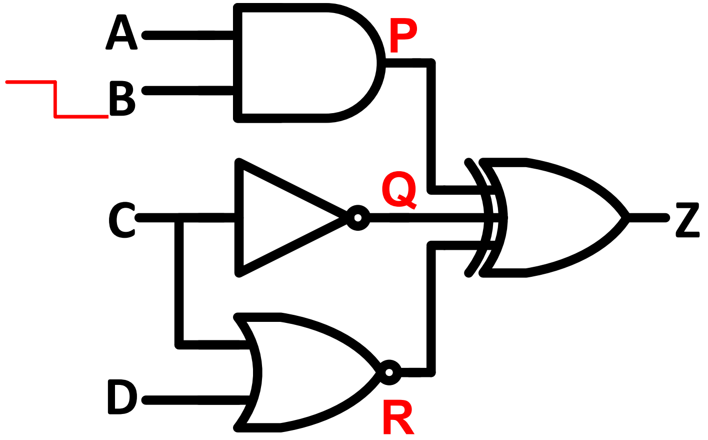
Signal x is connected to TWO “drivers”
→ Multiple Driver Nets error will occur



```
Implementation (2 errors)
  Opt Design (2 errors)
    DRC (1 error)
      Netlist (1 error)
        Net (1 error)
          [DRC MDRV-1] Multiple Driver Nets: Net x_OBUF has multiple drivers: x_OBUF_inst_i_1/O, and x_OBUF_inst_i_2/O.
    [Vivado_Tcl 4-78] Error(s) found during DRC. Opt_design not run.
```

Continuous Assignment

- **assign** statements are used to model combinational logic



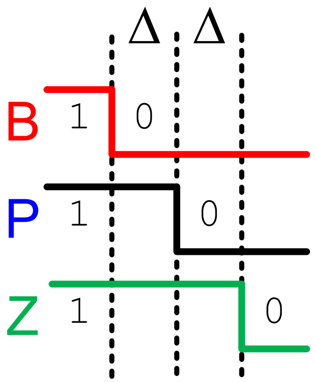
```
module bigbox (input a, b, c, d,
                     output z);
wire p, q, r;
```

internal connections

order?

irrelevant

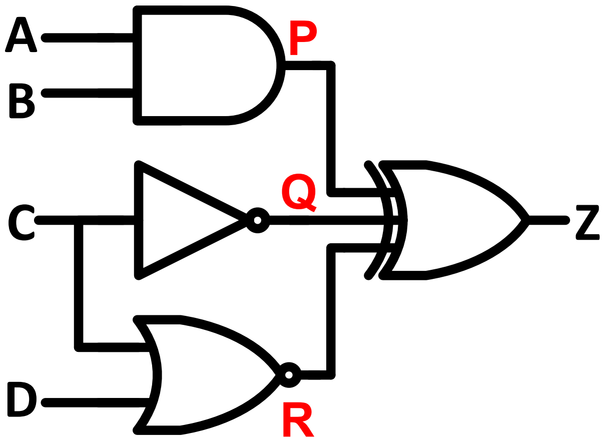
```
assign z = p ^ q ^ r; 1
assign q = ~c;        2
assign p = a & b;     3
assign r = ~(c | d);  4

endmodule
```



- Whenever there is an event on the RHS signal, expression is evaluated and assigned (after Δ delay) → continuously updated (assigned)
- Multiple statements executed concurrently (in parallel)
- **wire** used to represent an internal (physical) connection

Continuous Assignment

- **assign** statements can also be merged arbitrarily



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

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

endmodule
```

  - Concurrently anyway, no sequence whatsoever

Useful Operators

- Boolean (bit-wise), logical, arithmetic, concatenation
  - Use brackets for readability, use only synthesizable statements (apart from testbenches)

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 (merges all bits)	**&a = , |a= , ^a =**
{___,___}	Concatenation	{b, a} = 8'b
{n{____}}	Replication	{2 {a} } = 8'b
* , / , % ,	Multiply, *Divide, *Modulus	3 % 2 = 1, 16 % 4 = 0
+ , -	Binary addition, subtraction	a + b = 4'b1010
<< , >>	Shift Zeros in Left / Right	a << 1 = 4b' , a >> 2 = 4'b
< , <= , > , >=	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

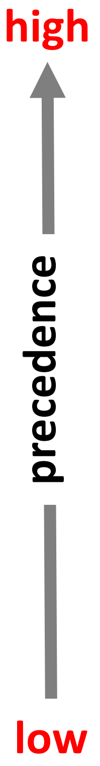
if all bits=0 ("false vector") → 1
0 otherwise

<< towards MSB (0s inserted)
>> towards LSB (0s inserted)
(independent of index order)

vector 00…0
equivalent to 0
(1 otherwise)

Useful Operators

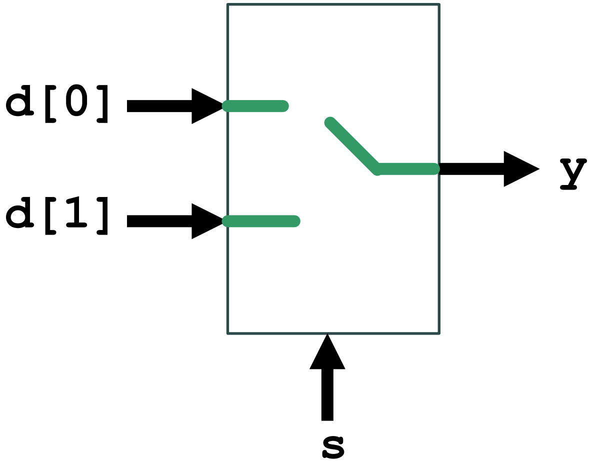
- Boolean (bit-wise), logical, arithmetic, concatenation
  - Use brackets for readability, use only synthesizable statements (apart from testbenches)



Operator	Description	Examples: a = 4'b1010, b=4'b0000
`! , ~`	Logical negation, Bit-wise NOT	!a = 0, !b = 1, ~a = 4'b0101, ~b = 4'b1111
`&, |, ^`	Reduction (merges all bits)	**&a = 0, |a=1, ^a = 0**
`{__,__}`	Concatenation	{b, a} = 8'b**00001010**
`{n{____}}`	Replication	{2 {a} } = 8'b**10101010**
`*, /, %,`	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) = 1**
`==, !=`	Logical Equality (1-bit output)	**(a == b)= 0 (a != b)= 1**
`&, ^, |`	Bit-wise AND, XOR, OR	**a&b = 4'b0000 a|b = 4'b1010**
`&&, ||`	Logical AND, OR (1-bit output)	**a&&b = 0 a||b = 1**
`?:`	Conditional Operator	<out> = <condition> ? If_ONE : if_ZERO

Conditional Operator

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



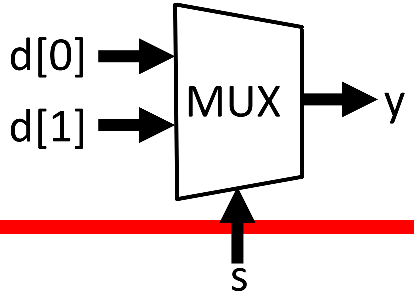
```
module mystery ( input s, input [1:0] d,
                 output y);

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

<output> = <condition> ? If_ONE : If_ZERO

s= 1 ➔ y = d[1],   s= 0 ➔ y= d[0]

```
endmodule
```

- This expression is evaluated whenever there is an event on any input
  - Continuous assignment
- What is this block? ← 2:1 multiplexer (MUX, see next week)



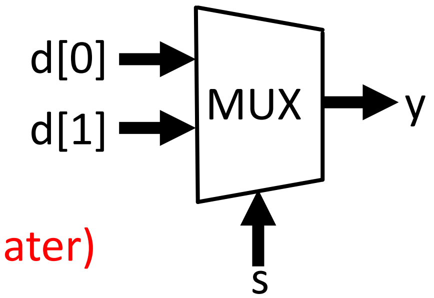
Procedural Assignment: always @

- Behavioral (higher-level description of logic)
  - Two assignment types: blocking vs. non-blocking

`y = d;` `y <= d;`

expresses combinational logic

expresses sequential logic (see later)



```
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
```

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

conceptually, the **always** block runs *once* 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.

Procedural Assignment: always @

- **always@(*)** for combinational logic
  - Sensitivity list must include **all input signals** that are read in current block
    - otherwise, no longer combinational: memory is inferred (see part II)
  - All outputs always explicitly assigned for any input value (including z, x…)
    - otherwise, no longer combinational: latch is inferred (see part II)

- Statements within always block are executed sequentially
  - But irrelevant with blocking assignments/combinational logic (see later for non-blocking)

- Multiple always blocks run concurrently (like continuous assignment)

```
always @ (…)     always @ (…)
begin            begin
                                    (…)
…                …
end              end
```

- No ~~assign~~ in always blocks

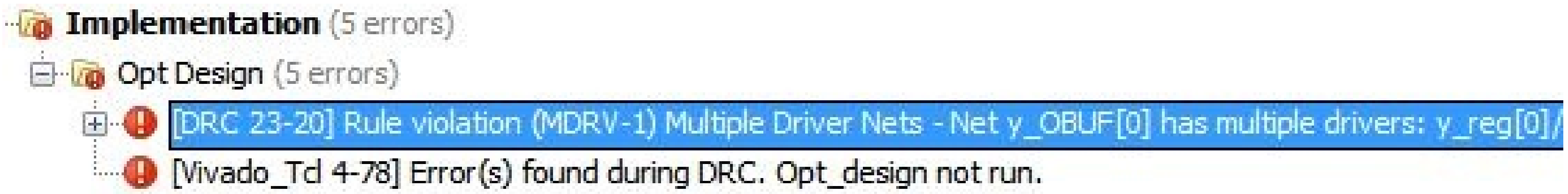
Review of Code Snippets and Common Errors

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


  always @ (*)
      y = y + 3;


endmodule
```

There are two conflicting instructions:

- first instruction would be to increment y by 1
- second would be to increment y by 3

Another case of multi-driven net (here with multiple procedural blocks)



```
Implementation (5 errors)
  Opt Design (5 errors)
    [DRC 23-20] Rule violation (MDRV-1) Multiple Driver Nets - Net y_OBUF[0] has multiple drivers: y_reg[0]/
    [Vivado_Tcl 4-78] Error(s) found during DRC. Opt_design not run.
```

Control-Flow Statements: Conditional

- Conditional statements: computations executed depending on conditions/value of variables
  - Always within procedural block (**always @**)
- **if**, **if-else** and **else-if**
  - If expression evaluates to true → execute statement(s)
  - Otherwise, do not (or else…)
- Nested **if-else**
  - Entails sense of priority
  - Use it if intended
  - If not, redundant logic is generated
    - unless conditions are mathematically guaranteed to be mutually exclusive

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;
```

Nested if-else

```
if (cond1)          signal_name<=value1;
else if (cond2)     signal_name<=value2;
else if (cond3)     signal_name<=value3;
…
else            signal_name<=defaultvalue;
```

synthesis

```
signal_name <= cond1*value1+not(cond1)*cond2*value2+...
not(cond1)*not(cond2)*...*defaultvalue
```

Control-Flow Statements: Conditional

- Conditional statements: computations executed depending on conditions/value of variables
  - Always within procedural block (**always @**)
- **case** compares expression with each case item
  - If none match, the default statement is executed
  - Default clause: unknown/unspecified values, shortens notation

case

```
case ( expr )

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

endcase
```

- No priority implied (mutually exclusive values)
  - logic checks only if expression matches one case item

```
case  (an-1...a0)
   item1: value1;
   item2: value2;
   ...
   itemn: valuen;
endcase
```

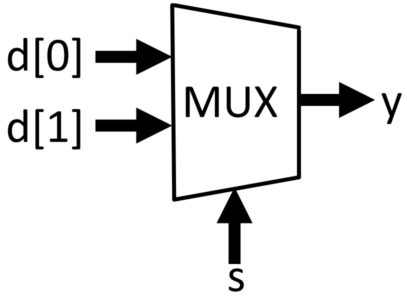
synthesis

- No redundant logic is generated

```
signal_name <= cond1*value1+cond2*value2+ ... condn*valuen
```

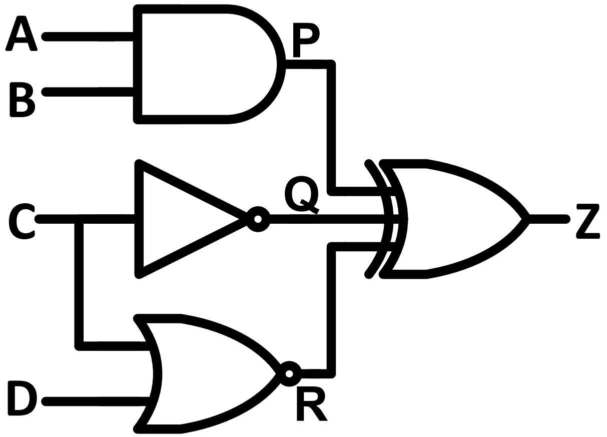
Equivalence of Procedural and Continuous Assign.

```
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 of Procedural and Continuous Assign.

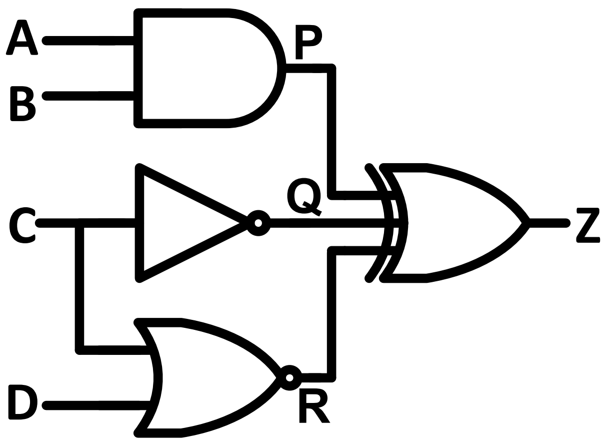


```
module bigbox
(input a,b,c,d, output z);

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 reg z);
reg p,q,r;
always @ ( a, b, c, d )
    begin
        q = ~c;
        p = a & b;
        r = ~(c | d);
        z = p ^ q ^ r;
    end
endmodule
```

Structural Modeling

- Structural modeling connects modules and gates
  - Practice with equivalent dataflow and structural description styles



Dataflow

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

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

endmodule
```

Structural (using primitives)

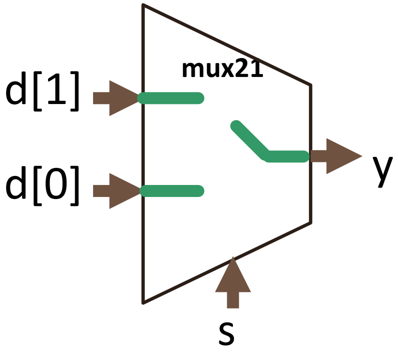
```
module bigbox (input a,b,c,d, output z);
wire p, q, r;

   and u1 (p, a, b); //output, then inputs
   not u2 (q, c);
   nor u3 (r, c, d);
   xor u4 (z, p, q, r);

endmodule
```

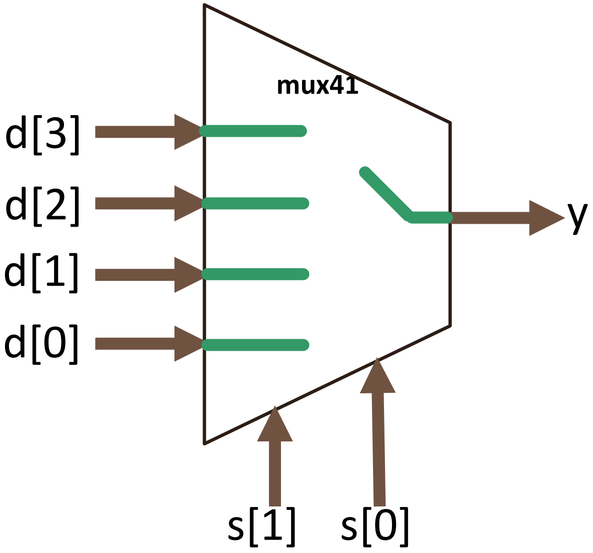
Structural Modeling

- Examples: 2:1 MUX and 4:1 MUX

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

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

endmodule
```

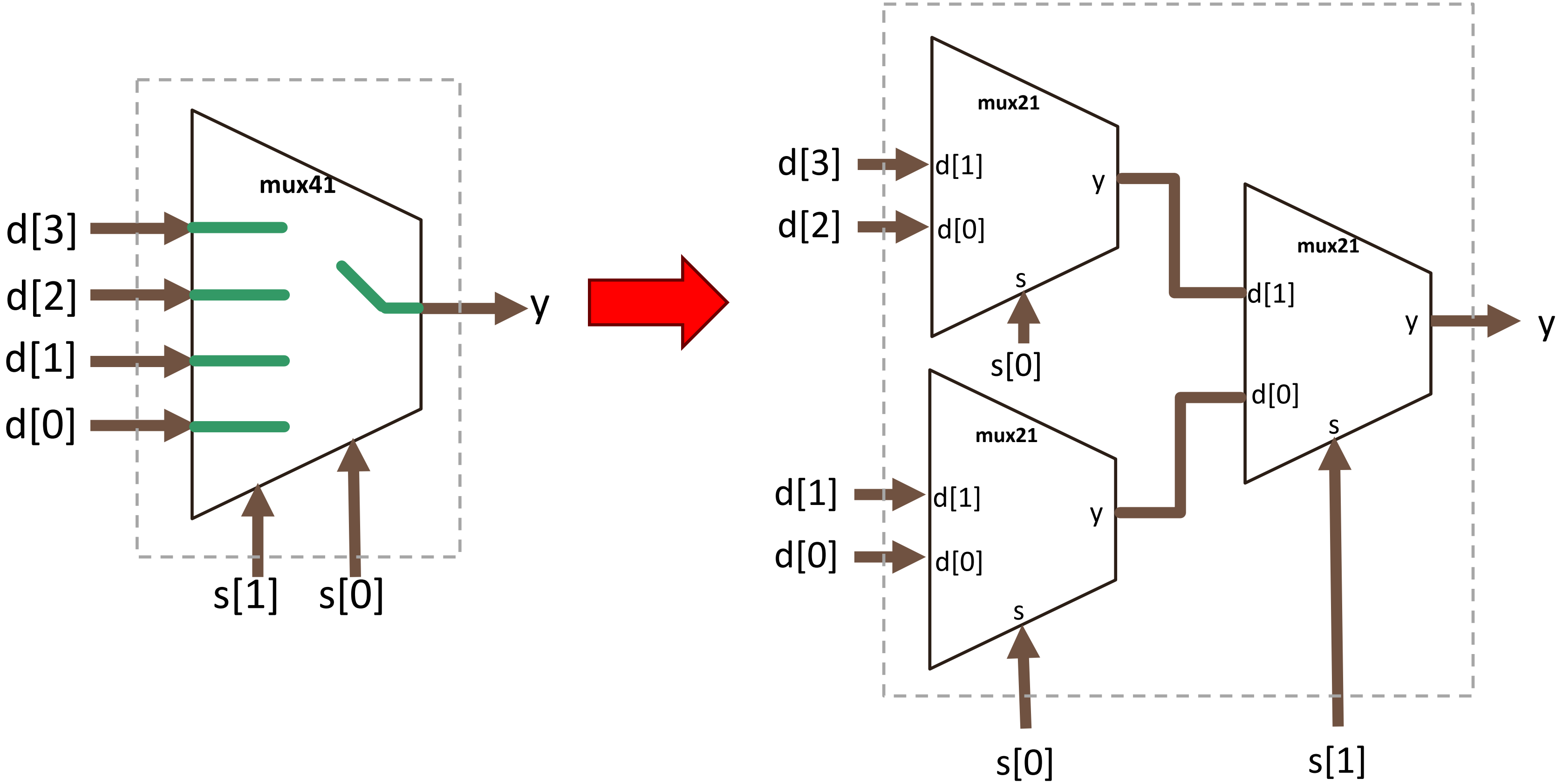
```
module mux41( input [1:0] s,
              input [3:0] d,
              output y);
… … … ?

endmodule
```





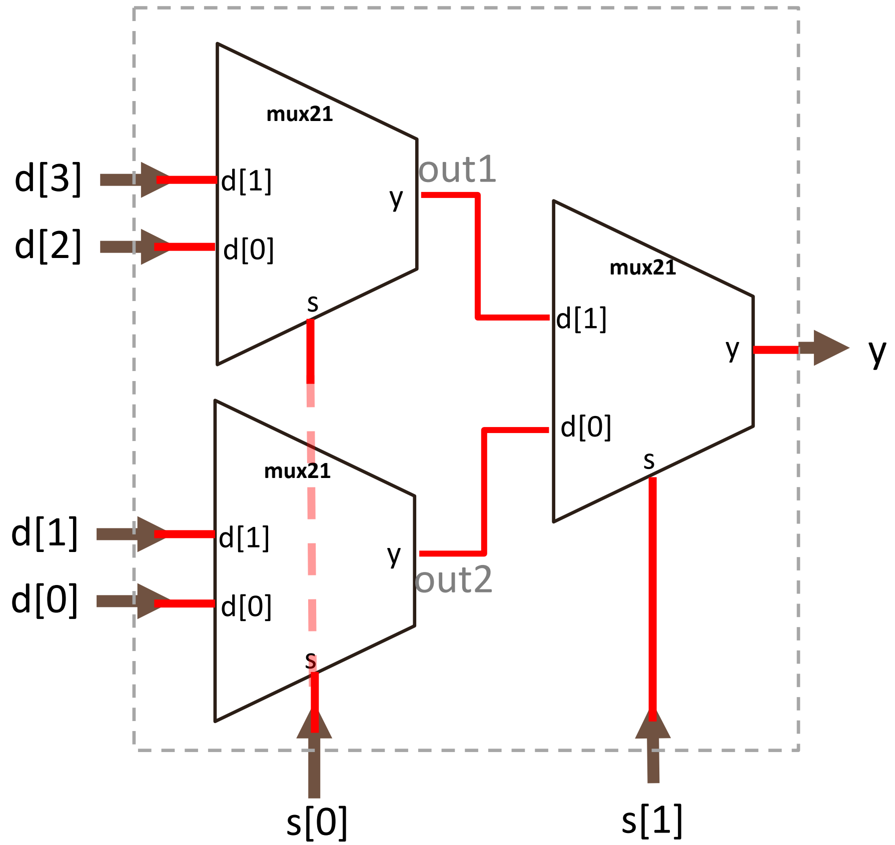
Structural Modeling

- 4:1 multiplexer can also be implemented by combining several 2-to-1 multiplexers



Structural Modeling

- 4:1 multiplexer can also be implemented by combining several 2-to-1 multiplexers



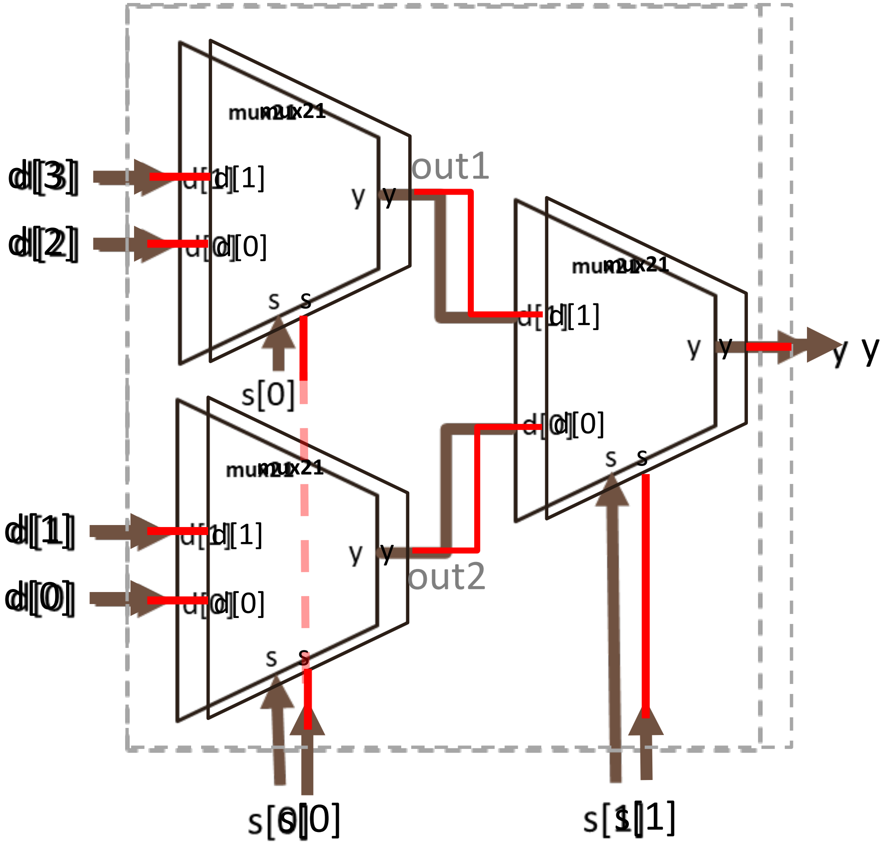
```
module mux21( input s, input [1:0] d,
              output y);
   assign y = s ? d[1] : d[0];
endmodule
```

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

- 4:1 multiplexer can also be implemented by combining several 2-to-1 multiplexers



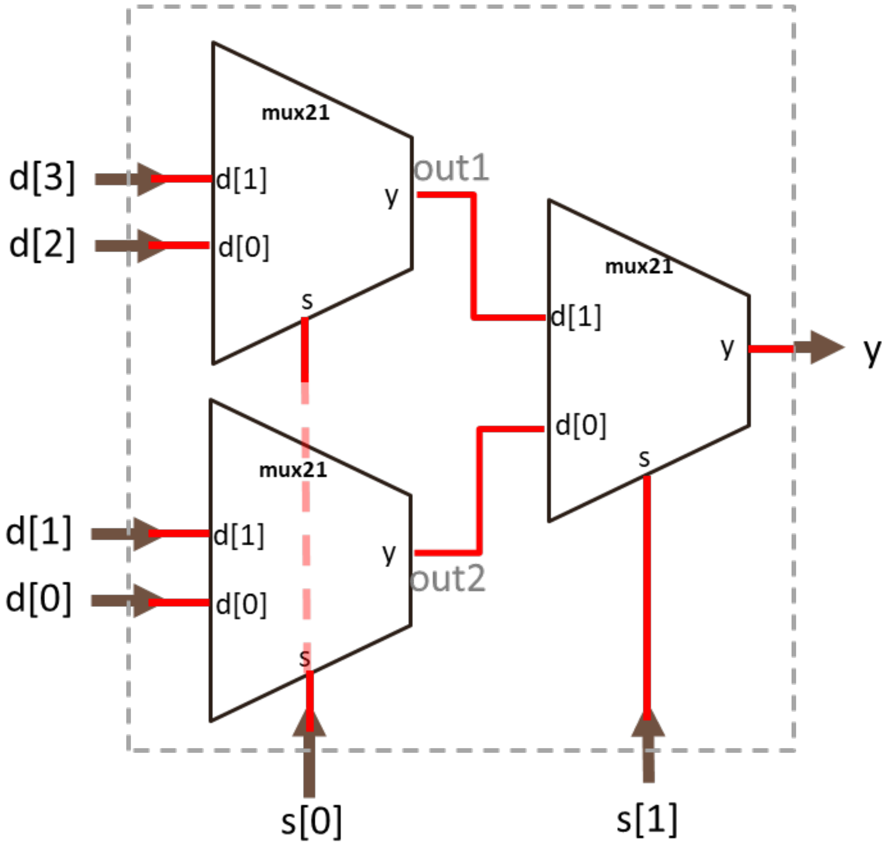
```
module mux21( input s, input [1:0] d,
              output y);
   assign y = s ? d[1] : d[0];
endmodule
```

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 ( s[0], d[1:0], out2 );

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

endmodule
```

Structural Modeling

- 4:1 multiplexer can also be implemented by combining several 2-to-1 multiplexers



Port Connection by Name

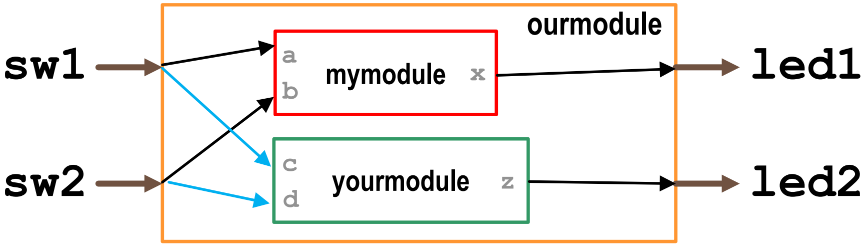
```
module mux41( input  [1:0] s,
              input  [3:0] d,
              output  y);

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
```

Structural Modeling

- For modular designs, the top view is specified as interconnected blocks
  - these examples 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

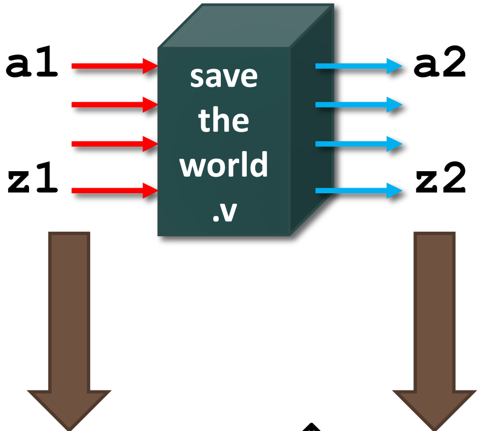
```
module ourmodule (input sw1, sw2,
                  output led1, led2);

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
```

Logic Simulations

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



How do we know our design actually works?

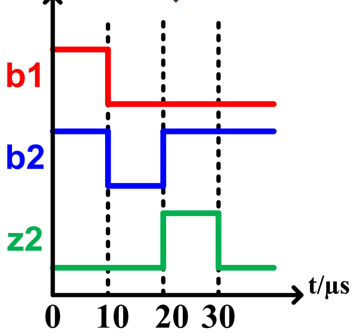
- ○ Functional Simulation (Verification)

Verilog Code

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

Test Bench

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



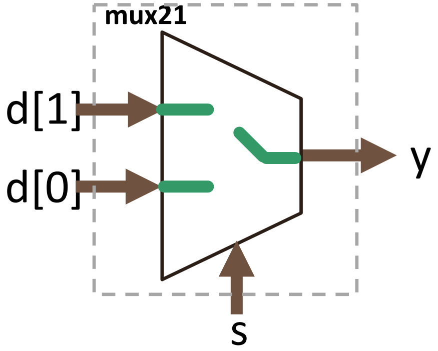
Method

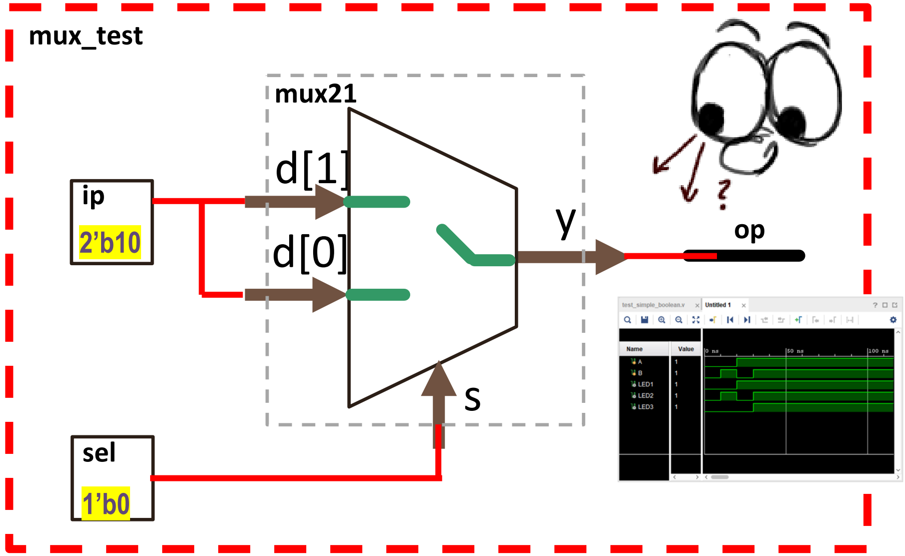
- ○ Designer applies input values to the code
- ○ Simulator produces corresponding outputs in truth tables / timing diagrams
- ○ Simulators usually assume negligible propagation gate delays.

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
```


mux21
d[1]
d[0]
y
s


mux_test
mux21
ip
2’b10
d[1]
d[0]
y
?
op
s
sel
1’b0
Name
Value
A
B
LED1
LED2
LED3