Lab 4
Introduction to Verilog - 1

Instructor: Chia-Chi, Tsai

Speaker: Parker Wei

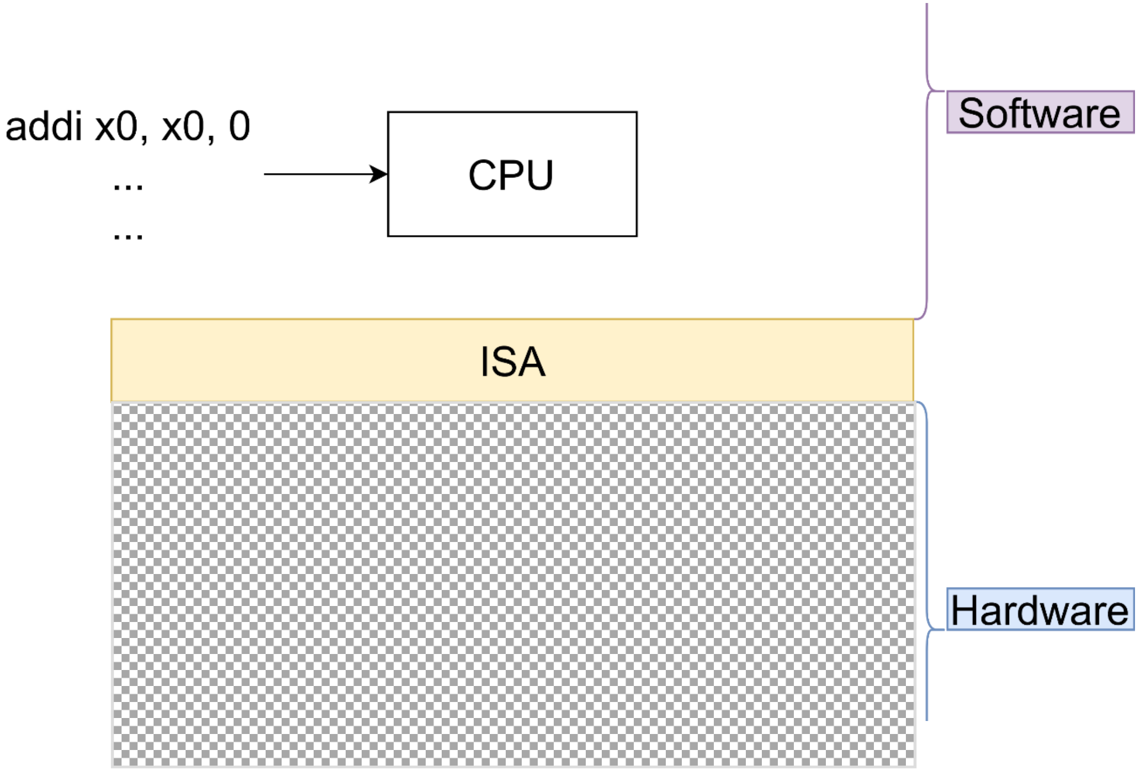
Outline

1. Why Do We Need to Know Verilog?
2. How Do We Describe a Circuit with Verilog?
3. Further into Nets and Continuous Assignments
4. Further into Regs and Procedural Assignments
5. What Would be Taught in Coming Classes?
6. Reference

Why Do We Need to Know Verilog?

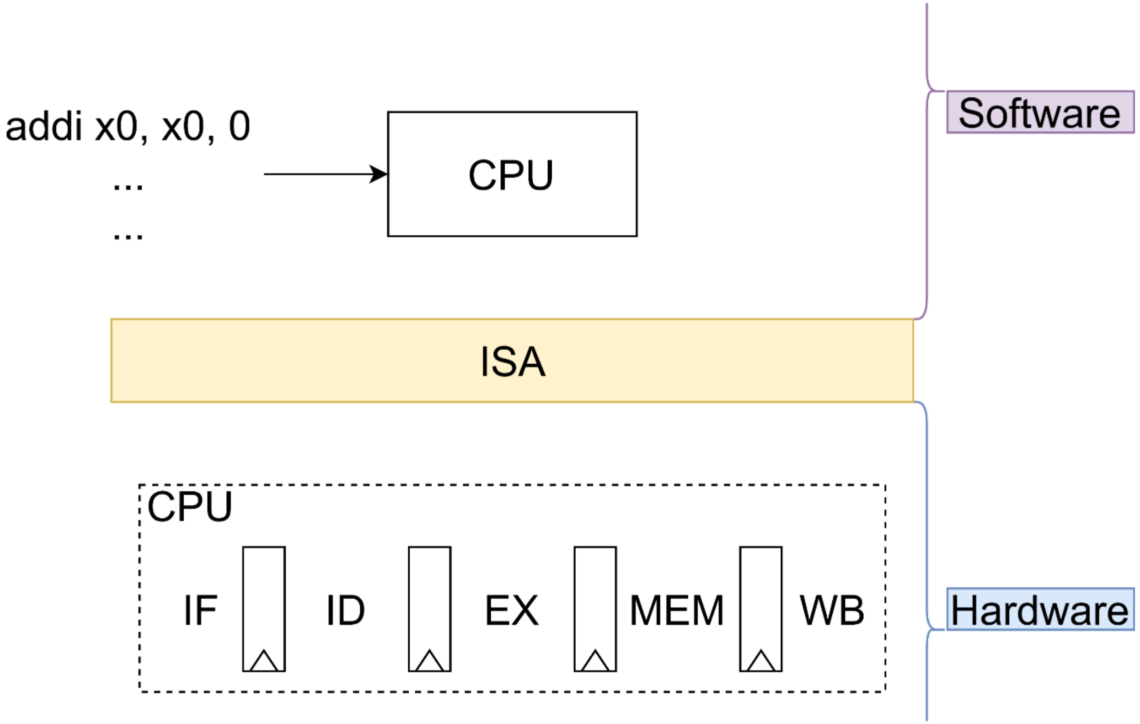
From Instructions to Architecture

- We've learnt ISA of RISC-V, which means we deemed CPU as a black box.



From Instructions to Architecture

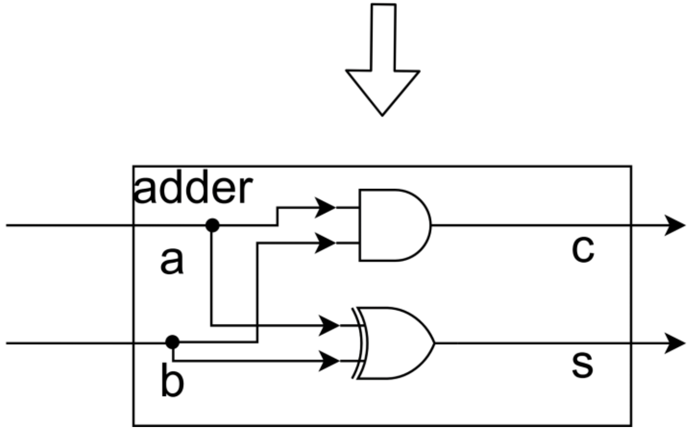
- However, as an architect, we need to know **why CPU can work this way**.



Map Your Design from Blueprint to Words

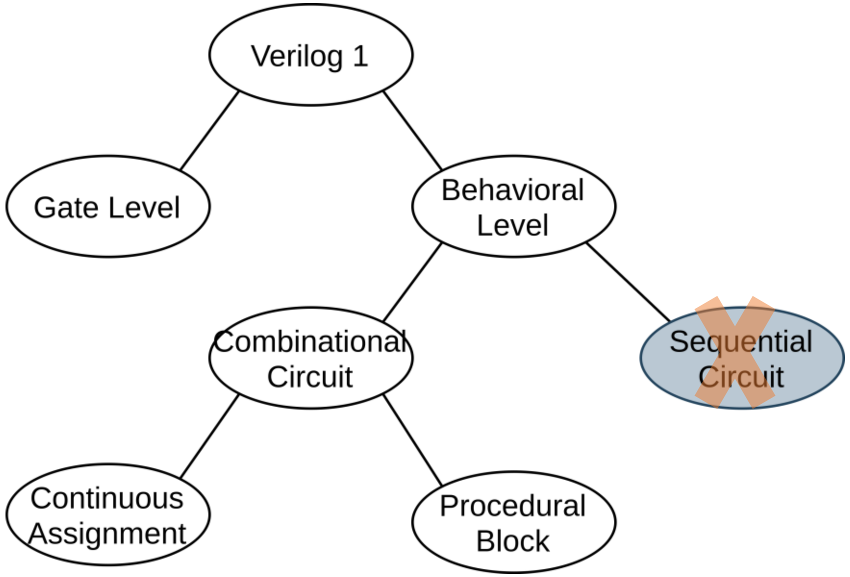
- To design CPU on your own, you need a way to let computer know your design.
- In these three classes, we will recap Verilog you've leant in digital system design.

```
module adder (input a, input b, output c, output s);
        assign c = a & b;
        assign s = a ^ b;
endmodule
```

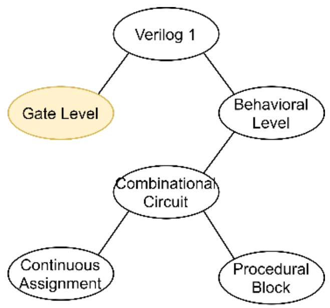


What would be included in this class

- We would go bottom-up, from **gate level** to **behavior level**.
- This class would focus on **combinational logic**, i.e.: no latch or flip-flop. This would be in next class.
- There are two simple exercises and one applied (harder) exercise in this class.

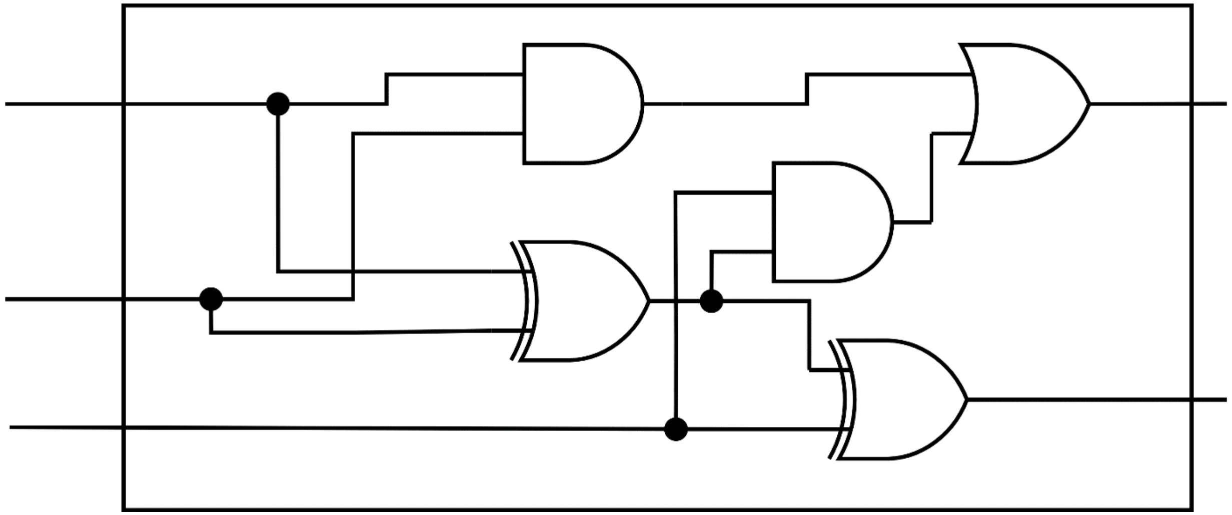


How Do We Describe a Circuit with Verilog?

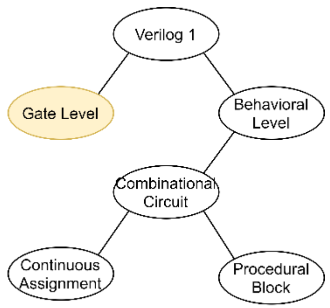


How to Construct a Circuit

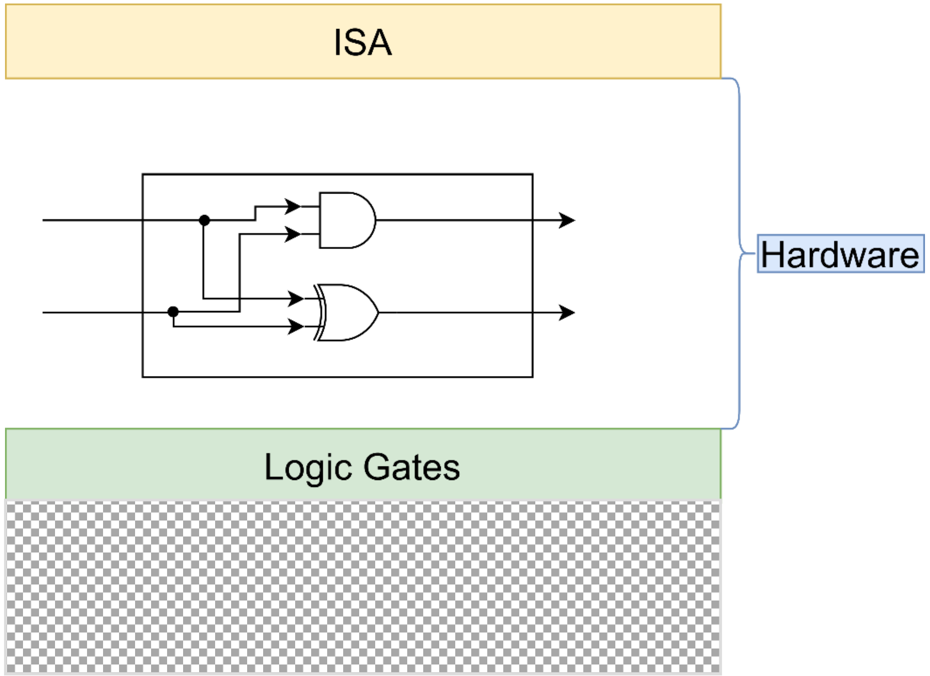
- Here's a circuit, which can function as a full adder. We need to describe this circuit with **text** so that simulator, design compiler, and other designers can know **how it is constructed**.

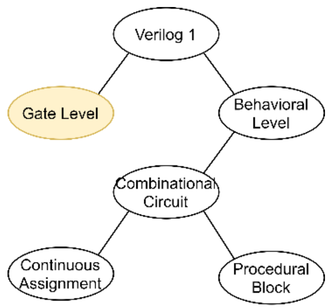


Basic Components



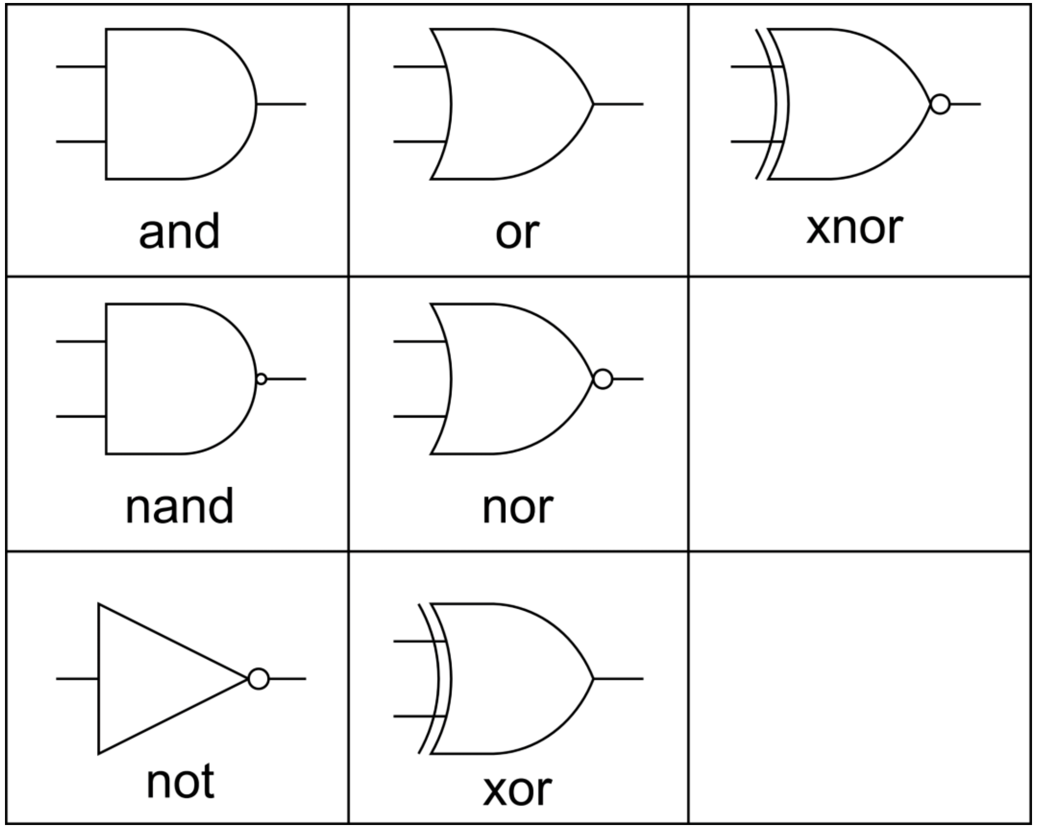
- Also, we need to know **what we can construct the circuit with**, so that we can optimally utilize them. In this section, our basic components are **logic gates**.

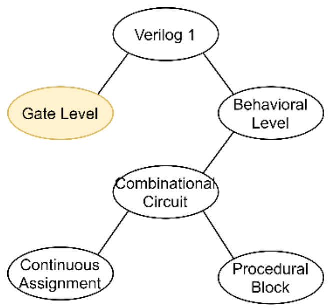




Basic Components

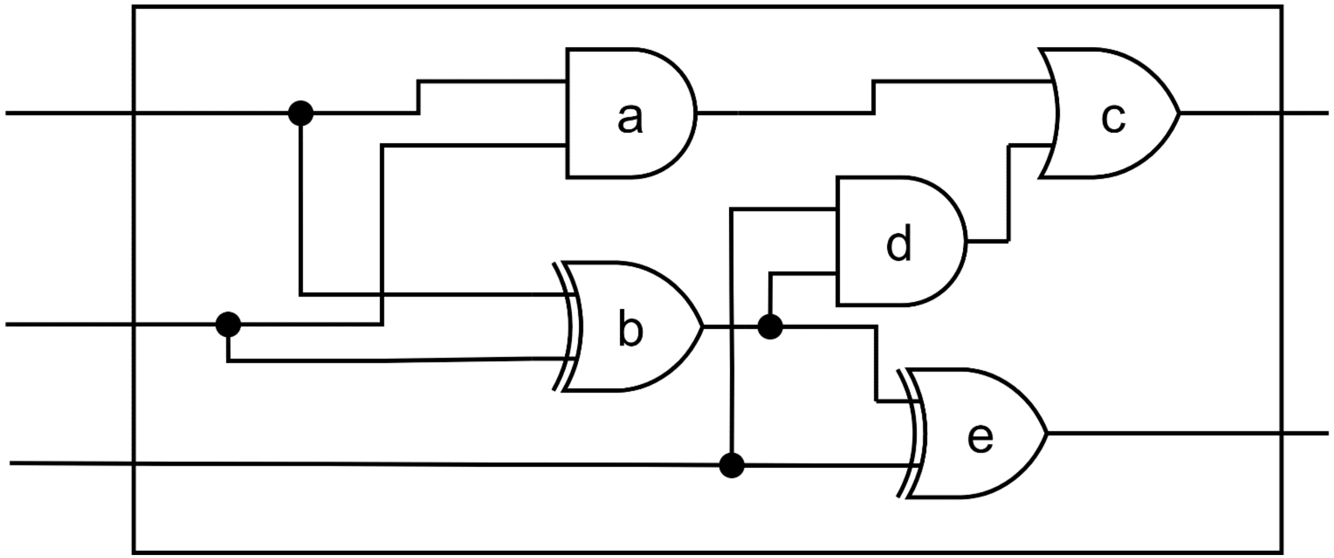
- There are several built-in gates that you can use at this abstraction level. These gates would be useful later.





How to Construct a Circuit - 1

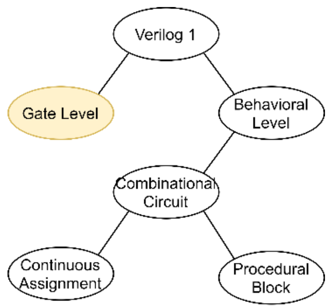
- Let get back to our full adder. First, we need to name the components inside this circuit.



Circuitry we want to express

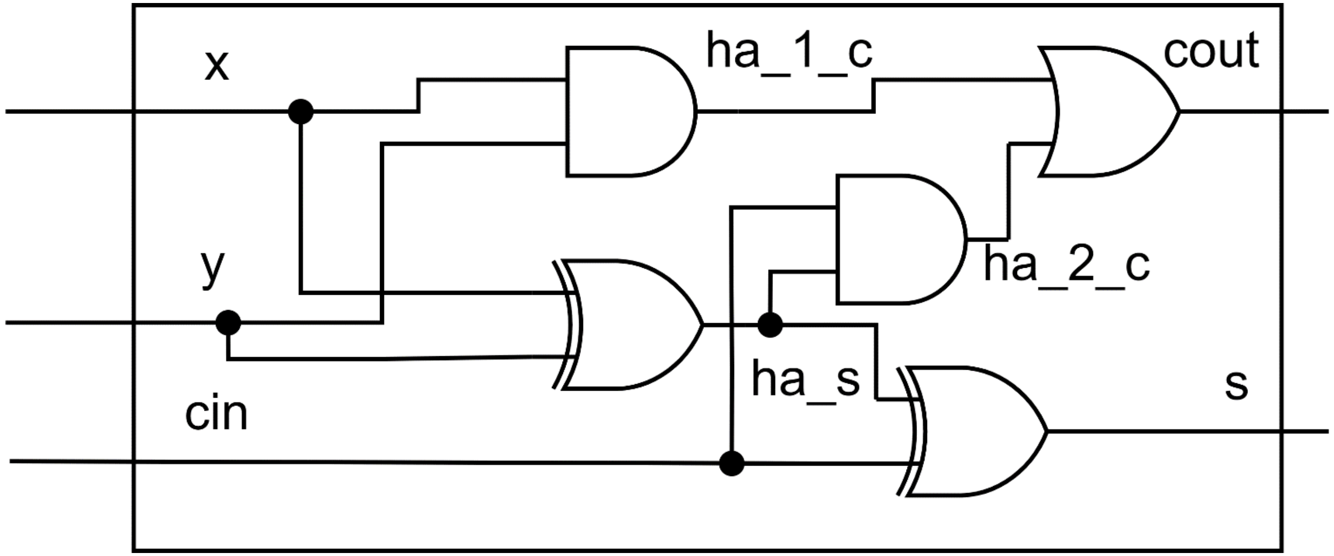
```
and a;
xor b;
or  c;
and d;
xor e;
```

Code under construction



How to Construct a Circuit - 2

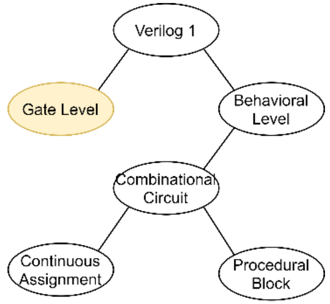
- Second, instantiate the interconnections. We have `wire` as our connection among each components.



Circuitry we want to express

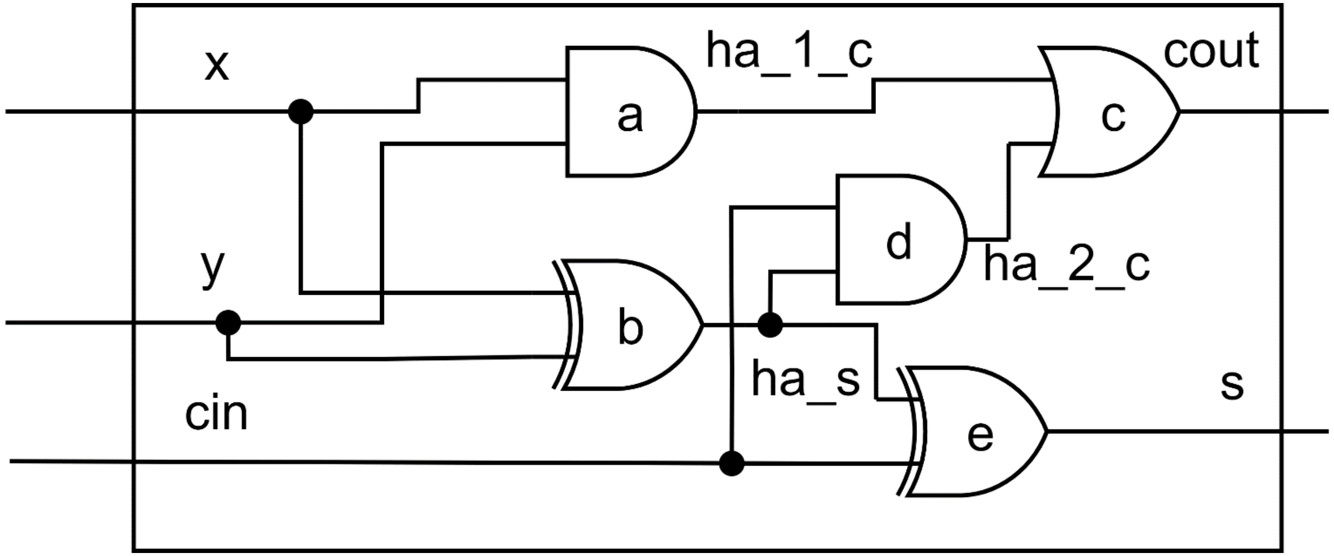
```
wire x, y, cin;
wire ha_1_c, ha_2_c, ha_s;
wire cout, s;
```

Code under construction



How to Construct a Circuit - 3

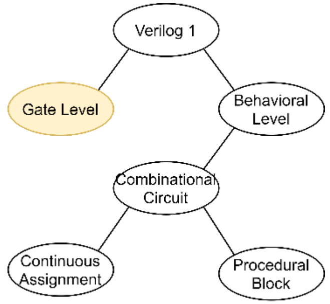
- Third, make the interconnection. The connections are made by putting wires in the parenthesis following the gate instance.



Circuitry we want to express

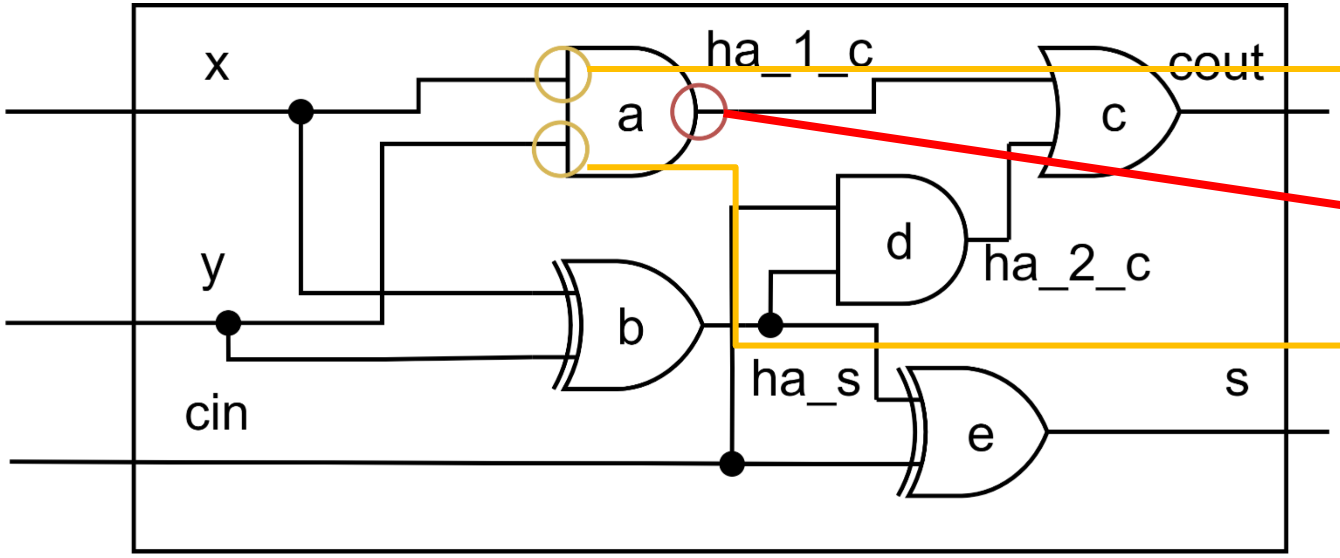
```
wire x, y, cin;
wire ha_1_c, ha_2_c, ha_s;
wire cout, s;

and a(ha_1_c, x, y);
xor b(ha_s, x, y);
or  c(cout, ha_1_c, ha_2_c);
and d(ha_2_c, cin, ha_s);
xor e(s, ha_s, c);
```

Code under construction



How to Construct a Circuit - 3

- Note: For gates, first port to be connected is the **output port**, and the rest of the ports would be **inputs**.

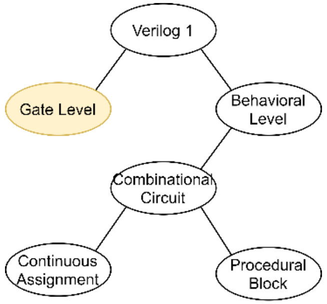


Circuitry we want to express

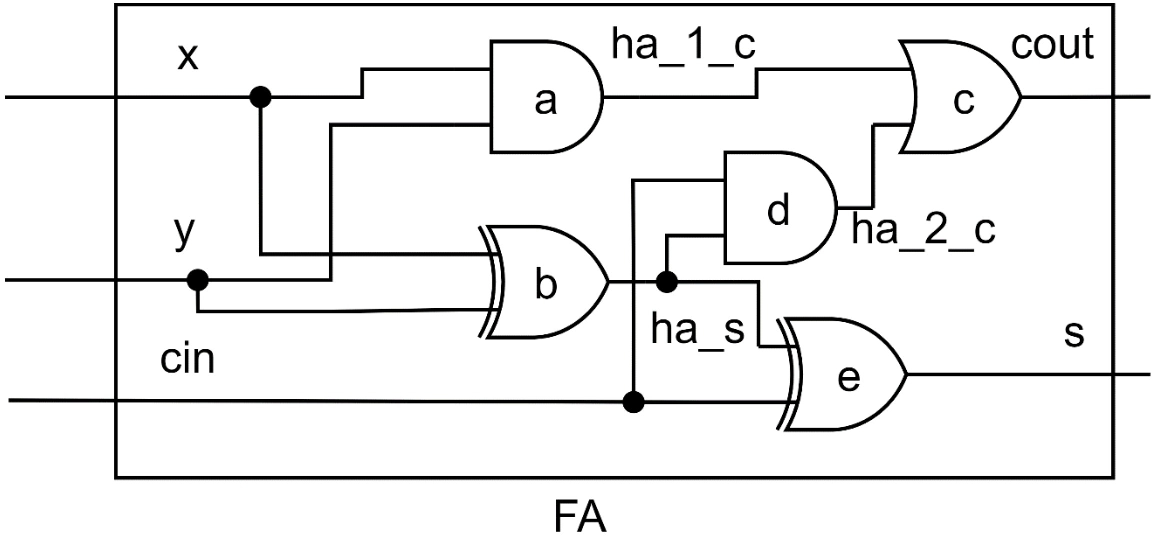
```
wire x, y, cin;
wire ha_1_c, ha_2_c, ha_s;
wire cout, s;

and a(ha_1_c, x, y);
xor b(ha_s, x, y);
or  c(cout, ha_1_c, ha_2_c);
and d(ha_2_c, cin, ha_s);
xor e(s, ha_s, c);
```

Code under construction

How to Construct a Circuit - 4



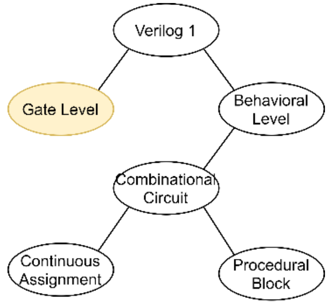
- Fourth, name your module.



Circuitry we want to express

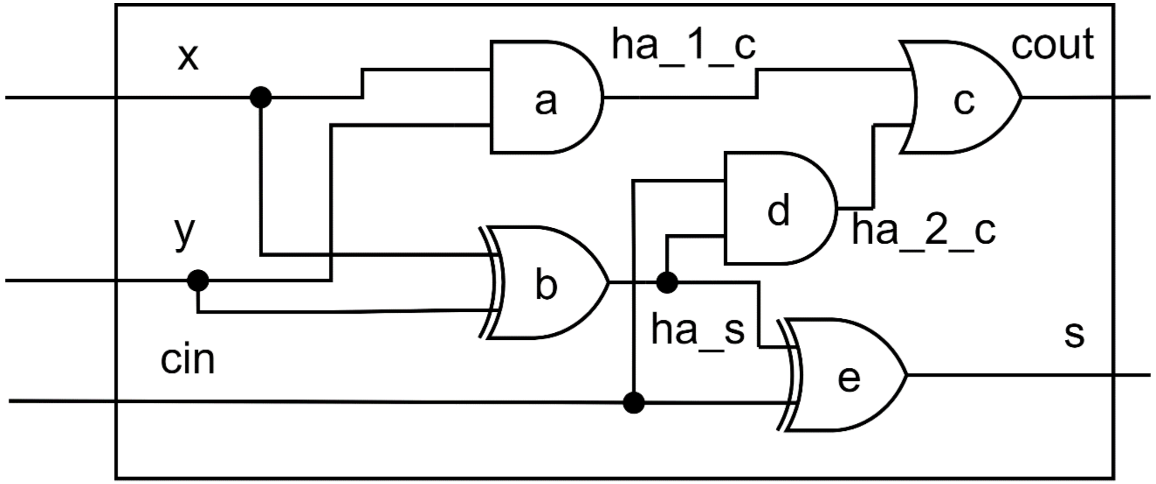
```
module FA;
    wire x, y, cin;
    wire ha_1_c, ha_2_c, ha_s;
    wire cout, s;

    and a(ha_1_c, x, y);
    xor b(ha_s, x, y);
    or  c(cout, ha_1_c, ha_2_c);
    and d(ha_2_c, cin, ha_s);
    xor e(s, ha_s, c);
endmodule
```

Code under construction



How to Construct a Circuit - 4

- Use `module xxx;` and `endmodule` to pack the circuitry.

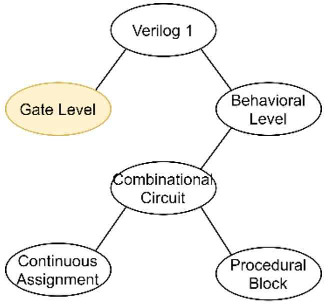


FA

Circuitry we want to express

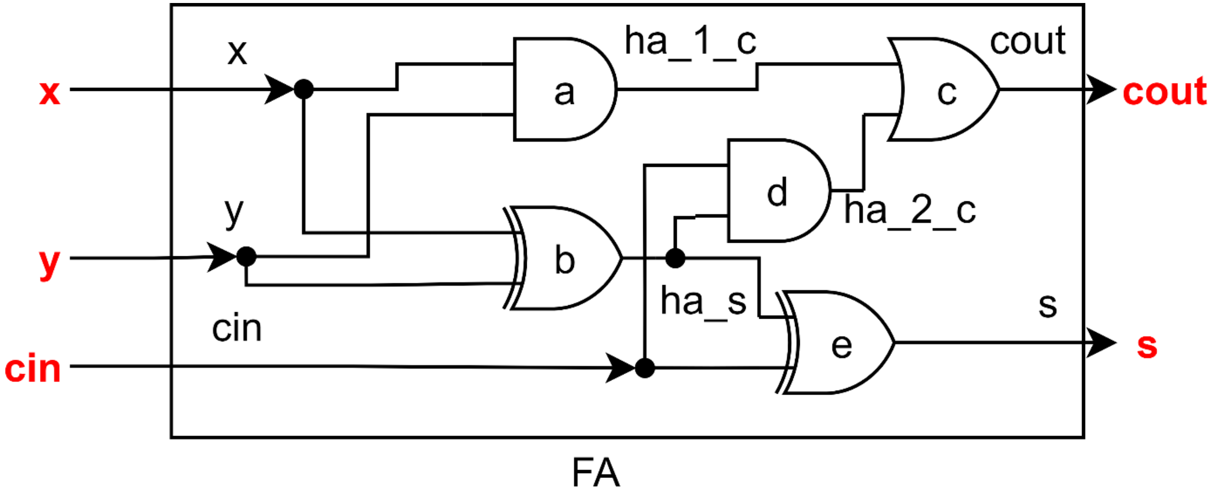
```
module FA;
    wire x, y, cin;
    wire ha_1_c, ha_2_c, ha_s;
    wire cout, s;

    and a(ha_1_c, x, y);
    xor b(ha_s, x, y);
    or  c(cout, ha_1_c, ha_2_c);
    and d(ha_2_c, cin, ha_s);
    xor e(s, ha_s, c);
endmodule
```

Code under construction

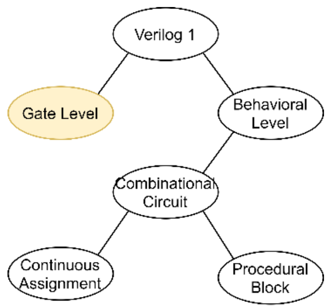


How to Construct a Circuit - 5

- Fifth, declare input and output ports for this module.
- Ports you want them to be connected externally should be added after module name.
- Moreover, `input` and `output` keywords need to be added before those ports.

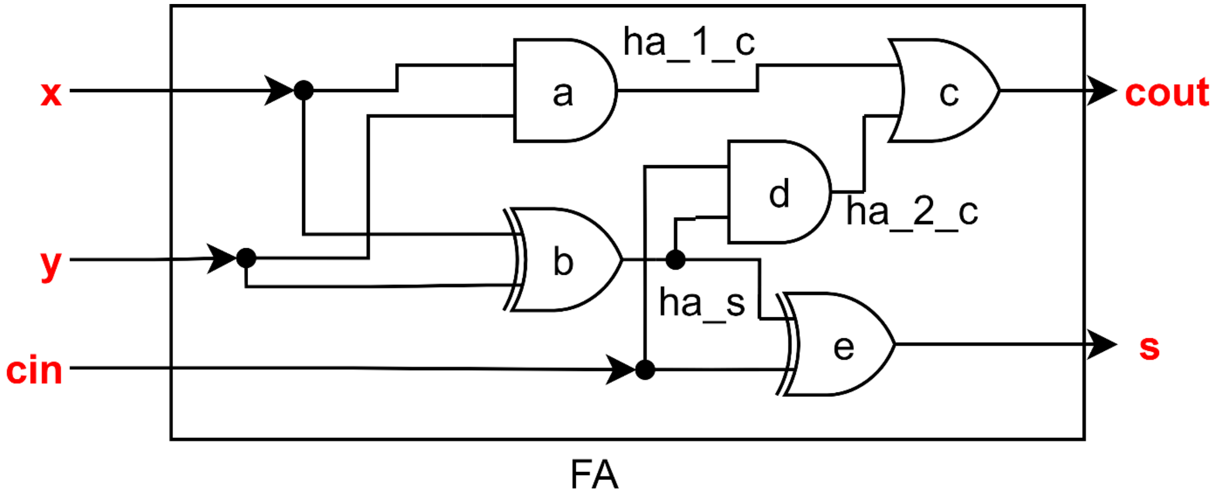



```
module FA(cout, s,  x, y, cin);
    input wire x, y, cin;
    wire ha_1_c, ha_2_c, ha_s;
    output wire cout, s;

    and a(ha_1_c, x, y);
    xor b(ha_s, x, y);
    or  c(cout, ha_1_c, ha_2_c);
    and d(ha_2_c, cin, ha_s);
    xor e(s, ha_s, cin);
endmodule
```

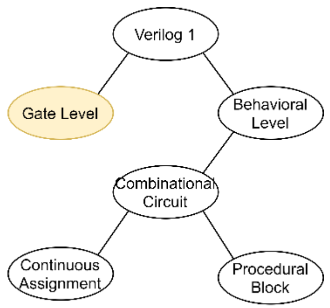


How to Construct a Circuit - 5

- However, in fact, there's no need to declare `wire` when ports are declared as `input` or `output`.

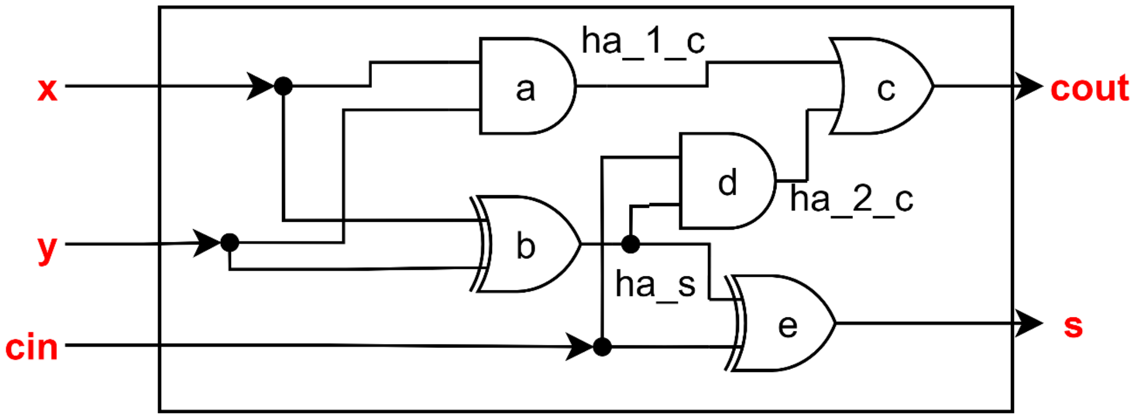



```verilog
module FA(cout, s,  x, y, cin);
    input x, y, cin;
    wire ha_1_c, ha_2_c, ha_s;
    output cout, s;

    and a(ha_1_c, x, y);
    xor b(ha_s, x, y);
    or  c(cout, ha_1_c, ha_2_c);
    and d(ha_2_c, cin, ha_s);
    xor e(s, ha_s, cin);
endmodule
```



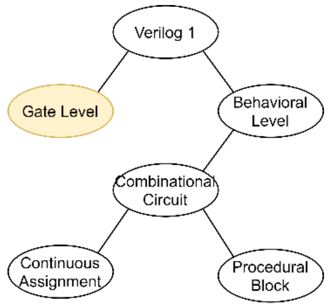
How to Construct a Circuit - 5

- Moreover, we can put input and output on port list and no need to retype them in body of the module.



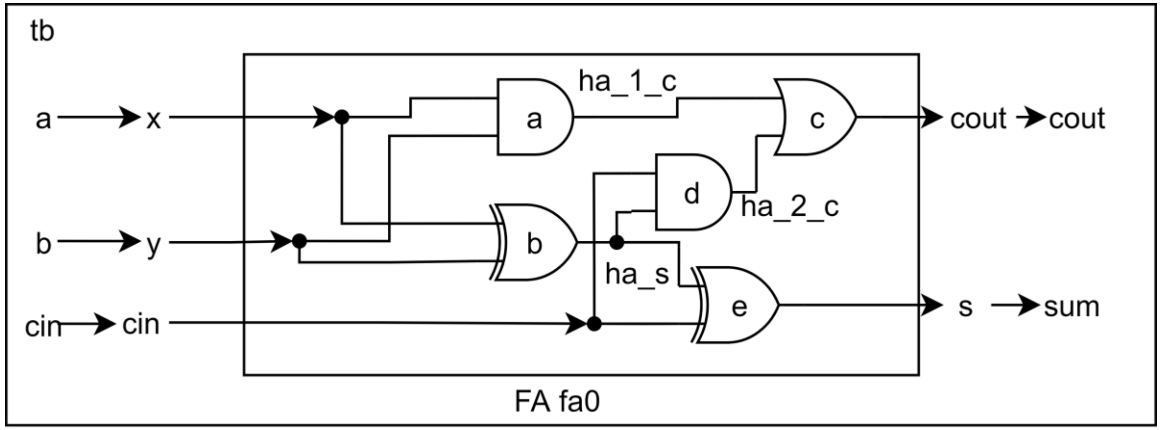
FA

```verilog
module FA(output cout, output s, input x, input y, input cin);
    wire ha_1_c, ha_2_c, ha_s;

    and a(ha_1_c, x, y);
    xor b(ha_s, x, y);
    or  c(cout, ha_1_c, ha_2_c);
    and d(ha_2_c, cin, ha_s);
    xor e(s, ha_s, cin);
endmodule
```



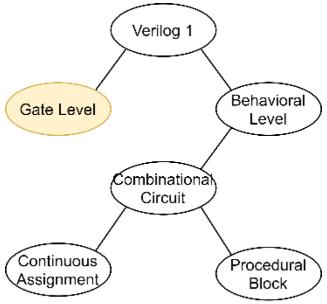
Connect Your Own Module

- That’s it. Your circuitry is now finished and can be instantiated by another module.
- One thing worth noting, the connection that full adder used is different from gates.



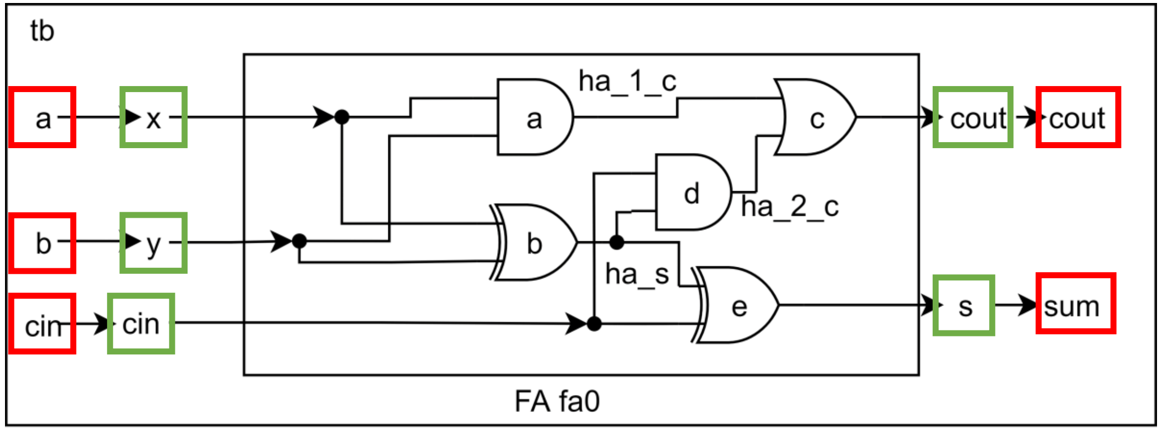

```
`include "FA.v"
module tb;
reg      a, b, cin;
wire     cout, sum;
FA fa0(.x(a), .y(b), .cin(cin), .cout(cout), .s(sum));
initial begin
        {a, b, cin} = 3'd0;
    #10 {a, b, cin} = 3'd1;
    #10 {a, b, cin} = 3'd2;
    #10 {a, b, cin} = 3'd3;
    #10 {a, b, cin} = 3'd4;
    #10 {a, b, cin} = 3'd5;
    #10 {a, b, cin} = 3'd6;
    #10 {a, b, cin} = 3'd7; #10;
end
endmodule
```

example1/testbench



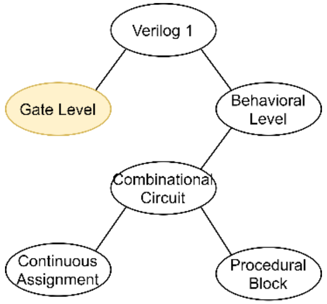
Connect Your Own Module - By Name

- The way we used now is called **connecting module instance ports by name**, which is quite useful when you have **lots of ports**.



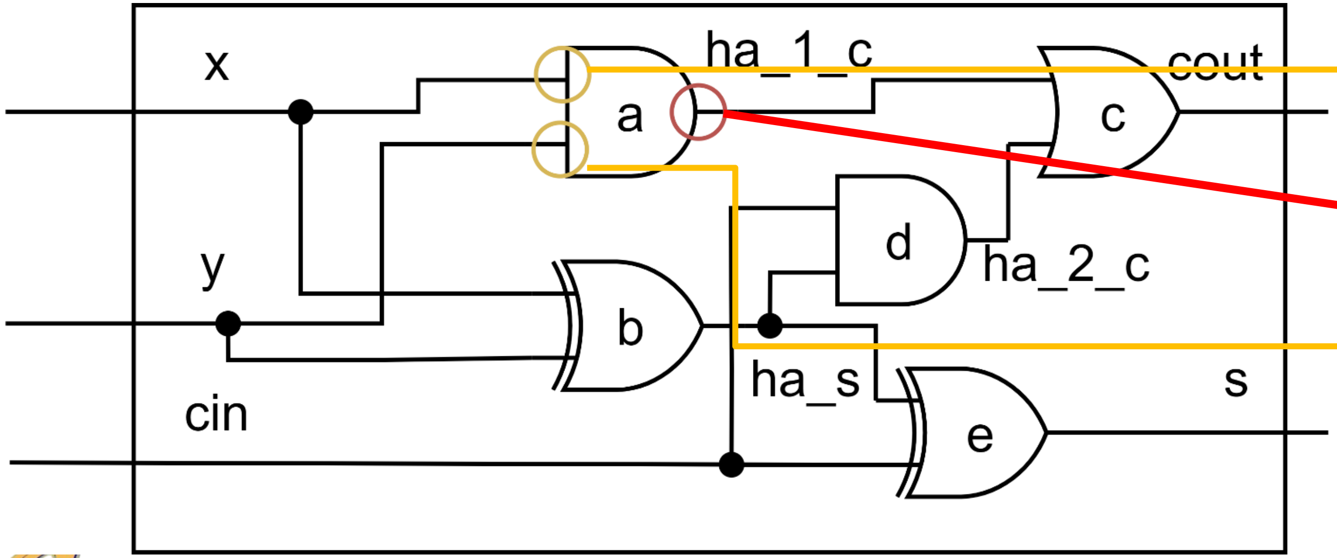

```
`include "FA.v"
module tb;
reg     a, b, cin;
wire    cout, sum;
FA fa0(.x(a), .y(b), .cin(cin), .cout(cout), .s(sum));
initial begin
        {a, b, cin} = 3'd0;
    #10 {a, b, cin} = 3'd1;
    #10 {a, b, cin} = 3'd2;
    #10 {a, b, cin} = 3'd3;
    #10 {a, b, cin} = 3'd4;
    #10 {a, b, cin} = 3'd5;
    #10 {a, b, cin} = 3'd6;
    #10 {a, b, cin} = 3'd7; #10;
end
endmodule
```

example1/testbench

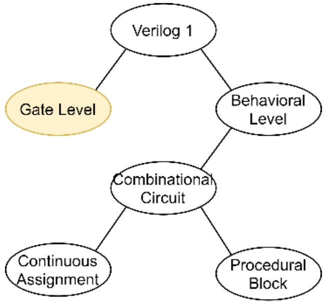


Ordered List

- On the other hand, we use **connecting module instance ports by ordered list** to connect our gates a little earlier.
- For primitive gates, you can’t use **by name** connection, so **by ordered list** connection is used. The order “output, in1, in2, in3…” must be obeyed.




```
wire x, y, cin;
wire ha_1_c, ha_2_c, ha_s;
wire cout, s;

and a(ha_1_c, x, y);
xor b(ha_s, x, y);
or  c(cout, ha_1_c, ha_2_c);
and d(ha_2_c, cin, ha_s);
xor e(s, ha_s, c);
```

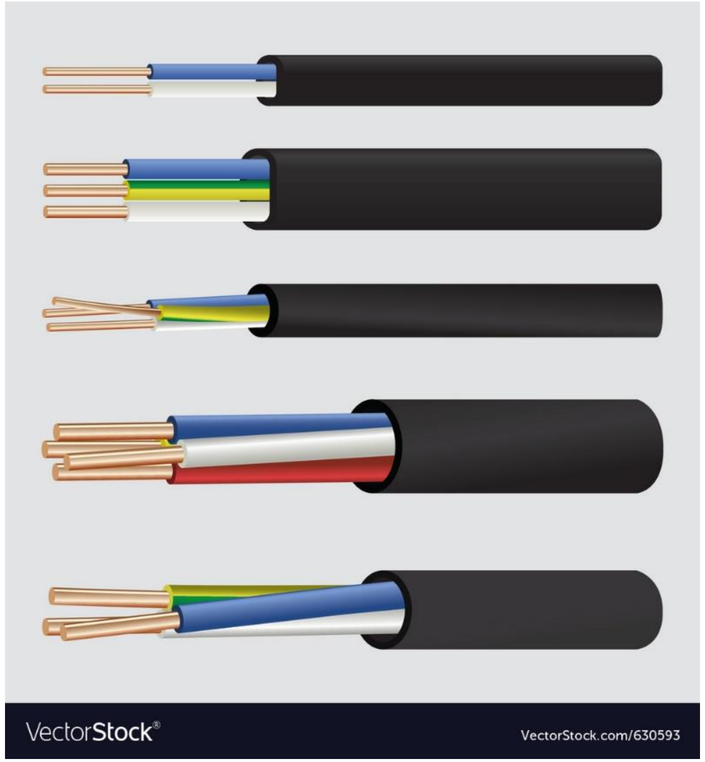


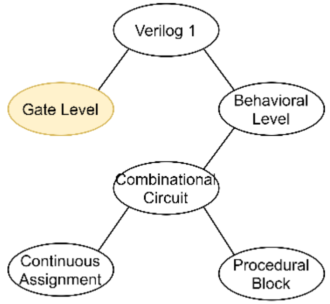
What If Multiple Lines are Needed?

- Scalar - A single wire/reg.

- Vectors - Declare a bunch of wires/regs.

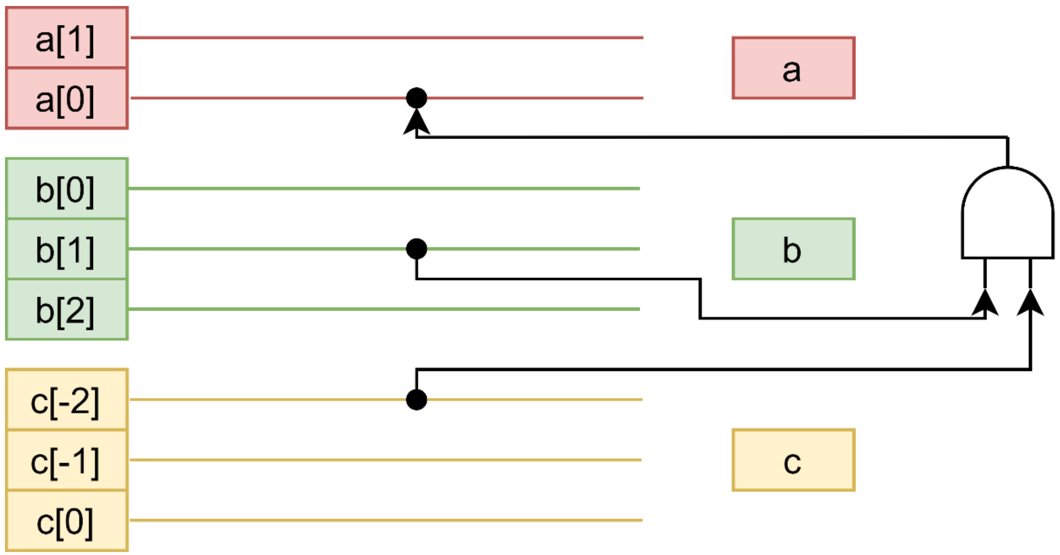

```
wire [1:0] a;
wire [0:2] b;
wire [-2:0] c;
wire [3:0] d;
wire [2:0] e;
```





What If Multiple Lines are Needed?

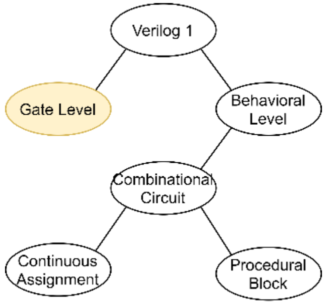
- Can one of the wires/regs be referenced?




```
wire [1:0] a;
wire [0:2] b;
wire [-2:0] c;

and and0(a[0], b[1], c[-2]);
```

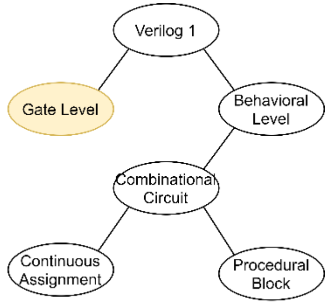
- Can multiple lines in a vector be referenced? **Yes**, and we would talk about it later.

What If Multiple Lines are Needed?



- Does MSB and LSB matters?
- For n-bit vector, ranging **from n-1 to 0** is recommended.


```
wire [3:0] d;
```



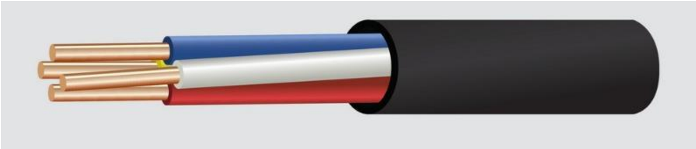
What If Multiple Lines are Needed?

- Array - Declare a lot of scalars/vectors.

```
wire        arr_scalar [0:7];
wire [3:0]  arr_vector [0:7];
```

x8

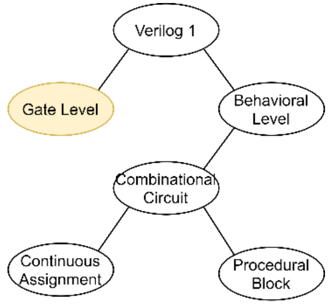
arr_scalar



x8

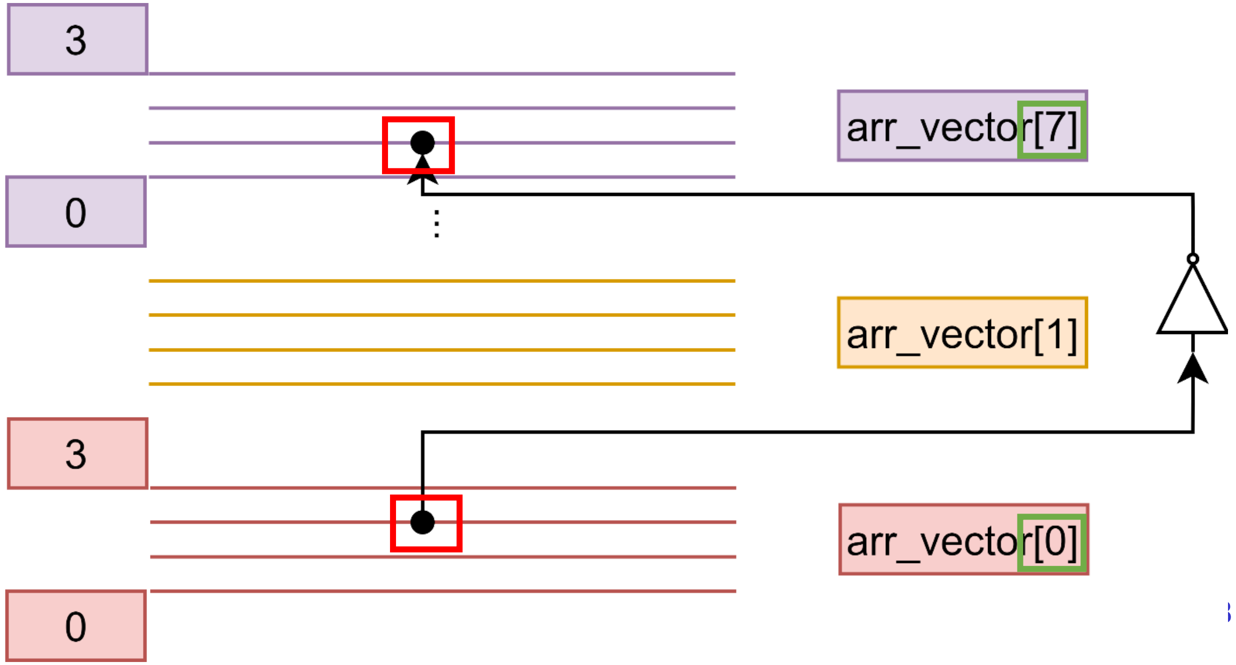
arr_vector

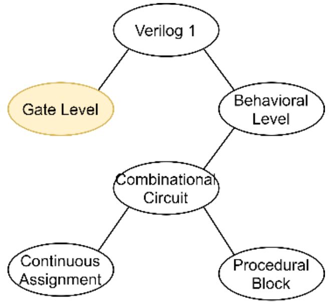
What If Multiple Lines are Needed?



- How to locate a single wire if we have an array of vectors?
- First index of array of vectors would select corresponding **vector**, and second index would choose **the line** in the vector.

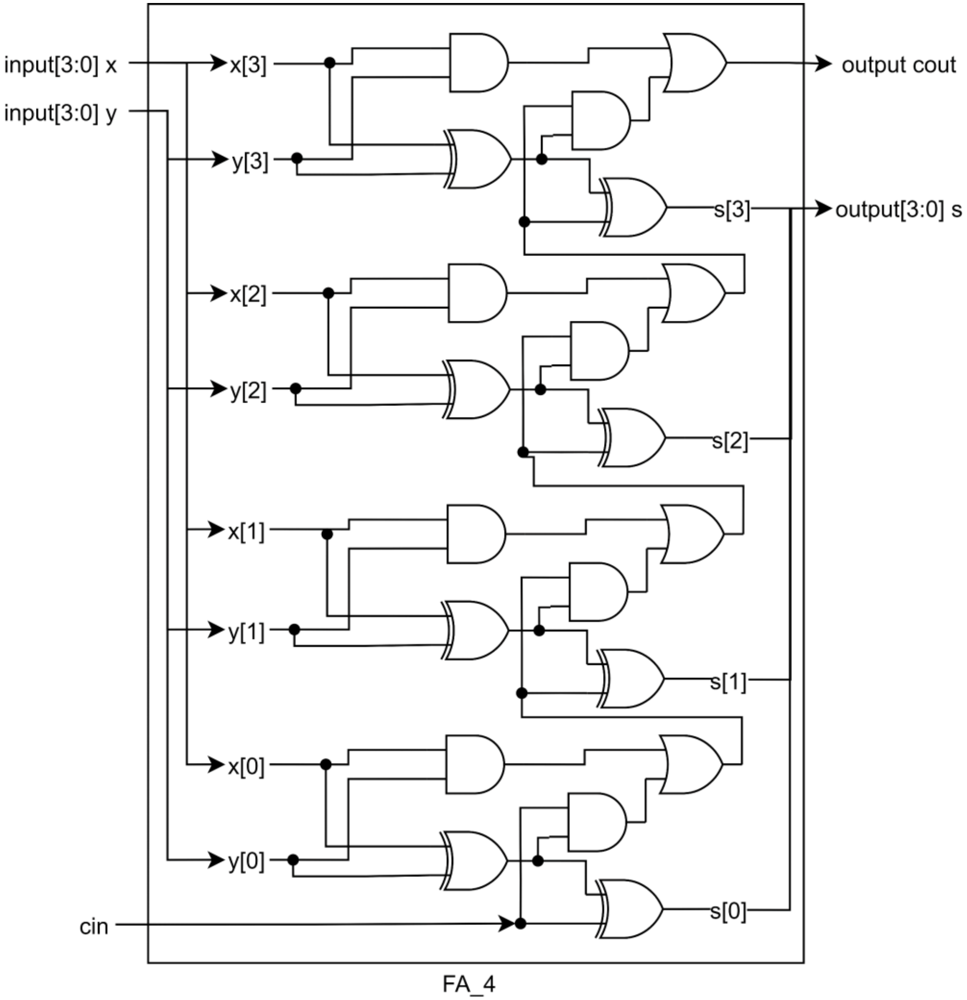
```
wire [3:0]  arr_vector [0:7];
not not0(arr_vector[0][2], arr_vector[7][1]);
```





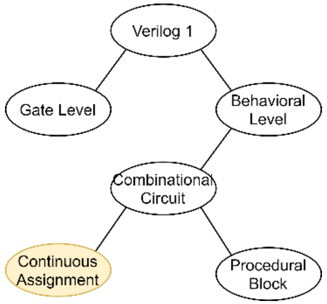
Exercise 1 - Gate-level 4-bit Adder

- In `Ex1/Adder.v`, you need to finish your own module named `Adder` in gate-level model, which would perform `{cout, s} = x + y + cin`
  - Two 4-bit inputs `x` and `y`
  - One 1-bit input `cin`
  - One 4-bit output `s`
  - One 1-bit output `cout`



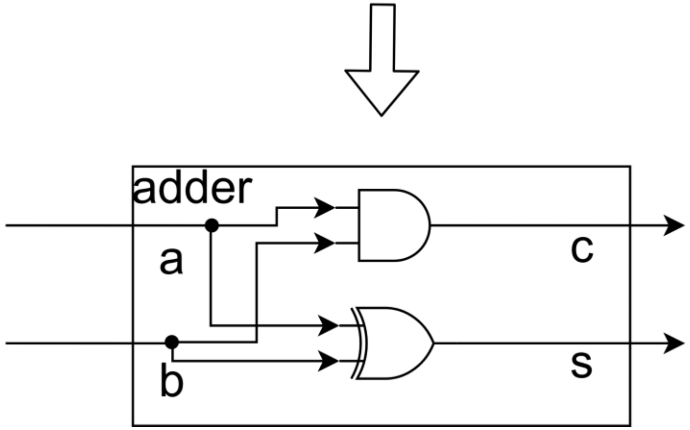
Further into Net Lists and Continuous Assignments

If We Want to Do More than Connection

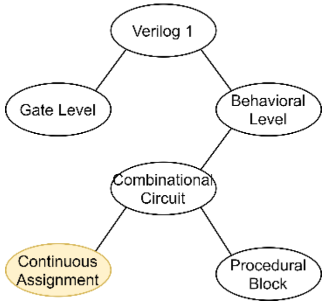


- Behavioral model is a more versatile way to represent the relationship between wires.

```
module adder (input a, input b, output c, output s);
        assign c = a & b;
        assign s = a ^ b;
endmodule
```



- However, you need to remember that **you are still modeling a circuit**.



What Can We Do in Behavioral Level?

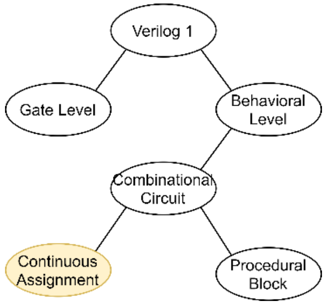
- There are two ways that we can use behavioral model.
  - Continuous assignment

```
assign a = b + c;
```

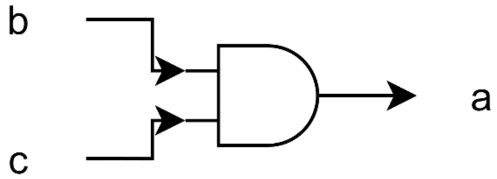
  - Procedural block

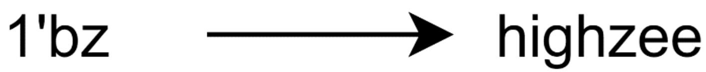
```
always @(*) begin
    a = b + c;
end
```

On Continuous Assignment



- Basic continuous assignment is composed of 4 parts, `assign`, `lhs`(left hand side), `=` , `rhs`(right hand side).
- It can be imagined as directly connect the evaluated result on rhs to lhs.
- We can change RHS as we need, which can be an expression composed of operands (wires, constant, regs) and operators.

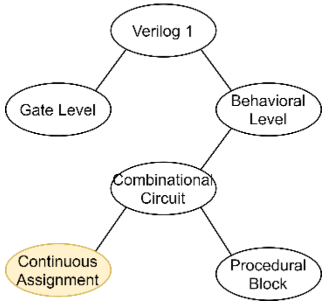





```
wire a, b, c;
assign a = b & c;
```

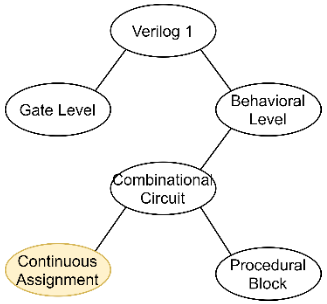
```
wire highzee;
assign highzee = 1'bz;
```

What Can We Use for Behavioral Model?



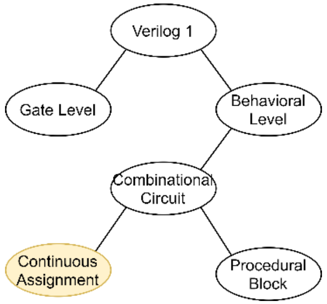
- First, let's talk about the data type in Verilog.
- As you know, there are 0's and 1's in binary computation. However, in real life, there might be cases that the data is unknown(x) and high impedance(z).
- As a result, in a wire/reg or a constant, there are four possibilities of the values, 0, 1, x, z.

What Can We Use for Behavioral Model?



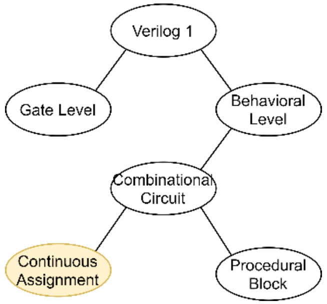
- Because data can have different bit length in Verilog, constant data can be annotated with data bit length. Default bit length is 32.
- Data can also be expressed in different format, including decimal, hexadecimal, octal, and binary format. Default format is decimal.
- Moreover, underscore ("_") is legal anywhere in the constant number except for the first digit.

What Can We Use for Behavioral Model?



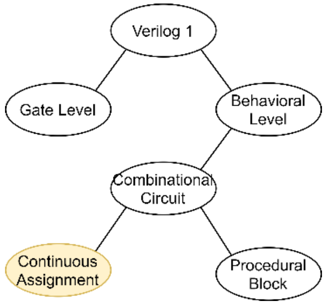
- Examples:
  - `659` is a decimal number
  - `'h837FF` is a hexadecimal number
  - `'o7460` is an octal number
  - `6'd32` is a 6-bit decimal number
  - `4'b1001` is a 4-bit binary number
  - `3'b01x` is a 3-bit number with LSB unknown

What Can We Use for Behavioral Model?



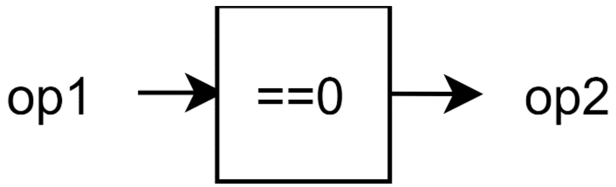
- After knowing the operands, let's have a look at operators.

Operator	Description	Usage	Operator	Description	Usage
+, -, *, /, %	Arithmetic operator	op1 + op2	&, |, ^, ^~(~^)	Bitwise operation	op1 & op2
!	Logical negation	!op	~	Bitwise negation	~op
&&, ||, ==, !=	Logical operator	op1 || op2	&, ~&, |, ~|, ^, ~^(^~)	Reduction operator	&op
<<, <<<, >>, >>>	Shift operator	op1 << op2	?:	Conditional	Condition? op1 : op2
{}	Concatenation	{op1, op2,…}	{{}}	Replication	{op1{op2}}

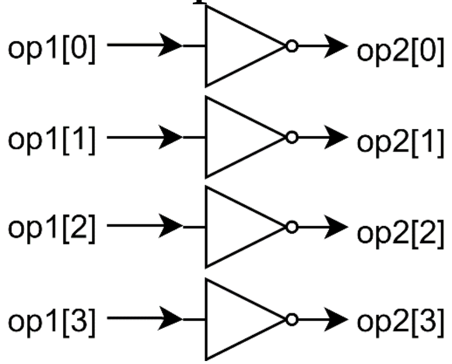


What Can We Use for Behavioral Model?

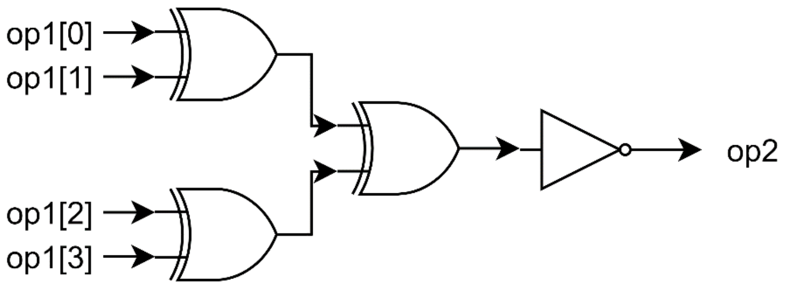
- Let’s get deeper into negation and reduction operations.
  - Logical negation would make output of zero input as 1 and make output of nonzero input as 0.
  - Bitwise negation would toggle all input bits.
  - Reduction would make all bits in input vector perform the same bitwise operation.



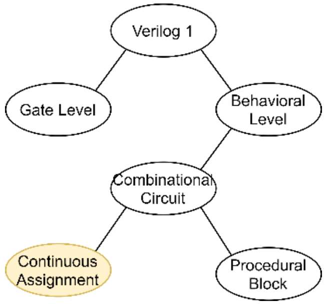

```
wire [3:0]  op1;
wire        op2;
assign op2 = !op1;
```




```
wire [3:0]  op1, op2;
assign op2 = ~op1;
```



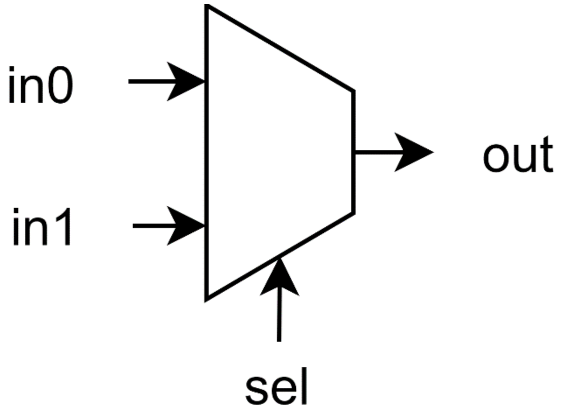

```
wire [3:0]  op1;
wire        op2;
assign op2 = ~^op1;
```

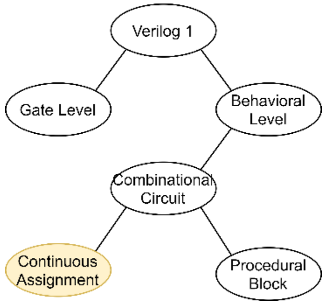


What Can We Use for Behavioral Model?

- Ternary operator (Conditional)
  - Composed of three parts: condition, value if true, value if false.
  - Can be deemed as mux.

```
wire        sel;
wire[3:0]   in0, in1, out;
assign out = (sel) ? in0 : in1;
```

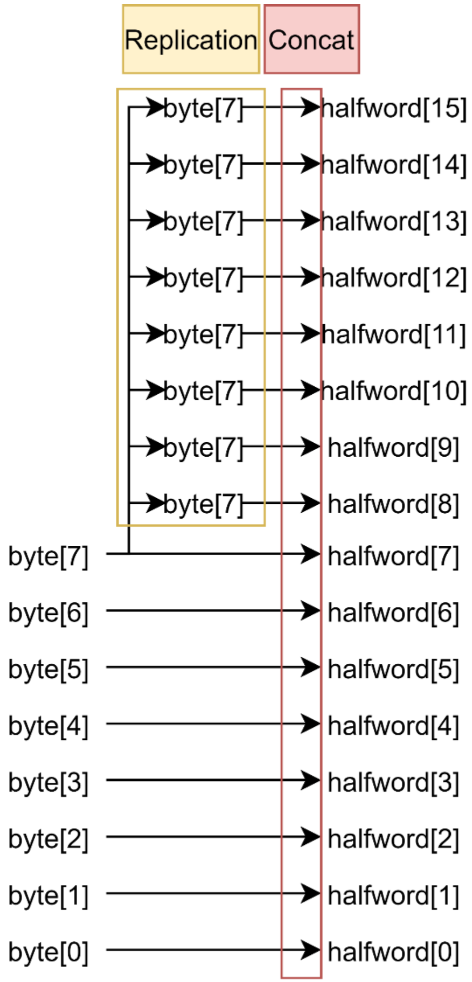


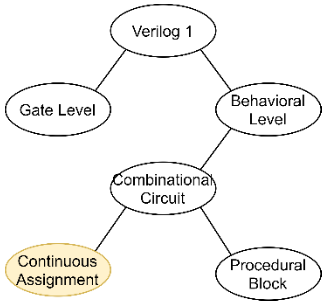


What Can We Use for Behavioral Model?

- Concatenation and replication
  - These two are useful when dealing with vectors.
  - Concatenation puts inputs together.
  - Replication would copy corresponding line with n times.

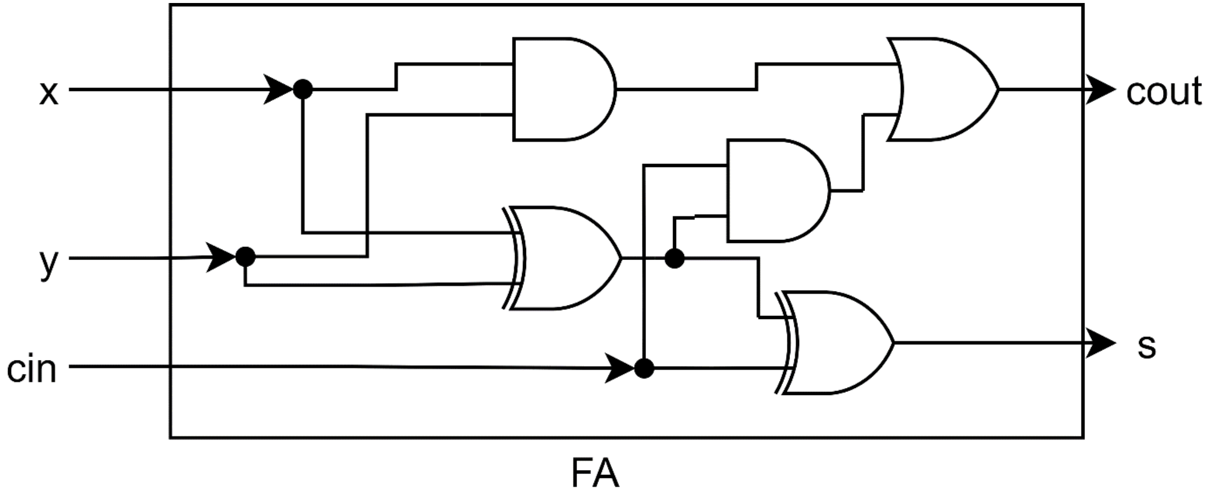
```
wire [7:0]  byte;
wire [15:0] halfword;
assign halfword = {{4'd8{byte[7]}}, byte};
```





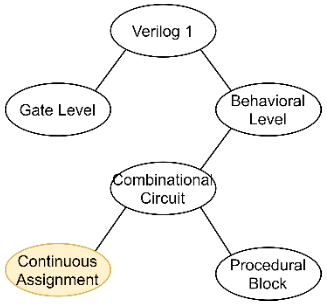
Apply Behavioral Model

- If we want to redo the full adder in behavioral model, here’s how it would go.



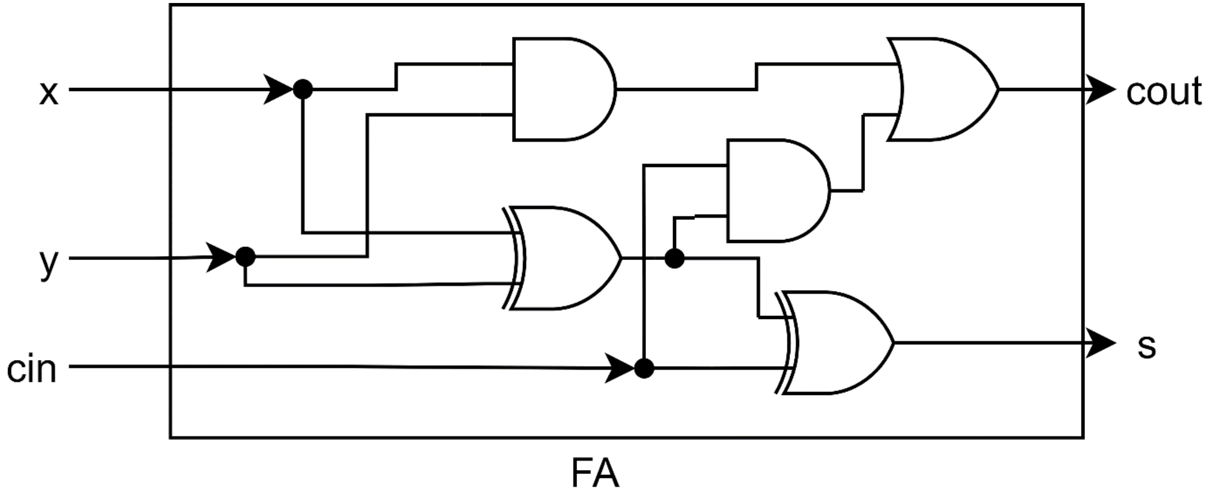
- First, declare ports required in the module named `FA`.

```
module FA(output cout, output s, input x, input y, input cin);


endmodule
```



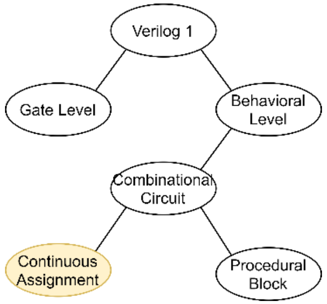
Apply Behavioral Model

- If we want to redo the full adder in behavioral model, here's how it would go.



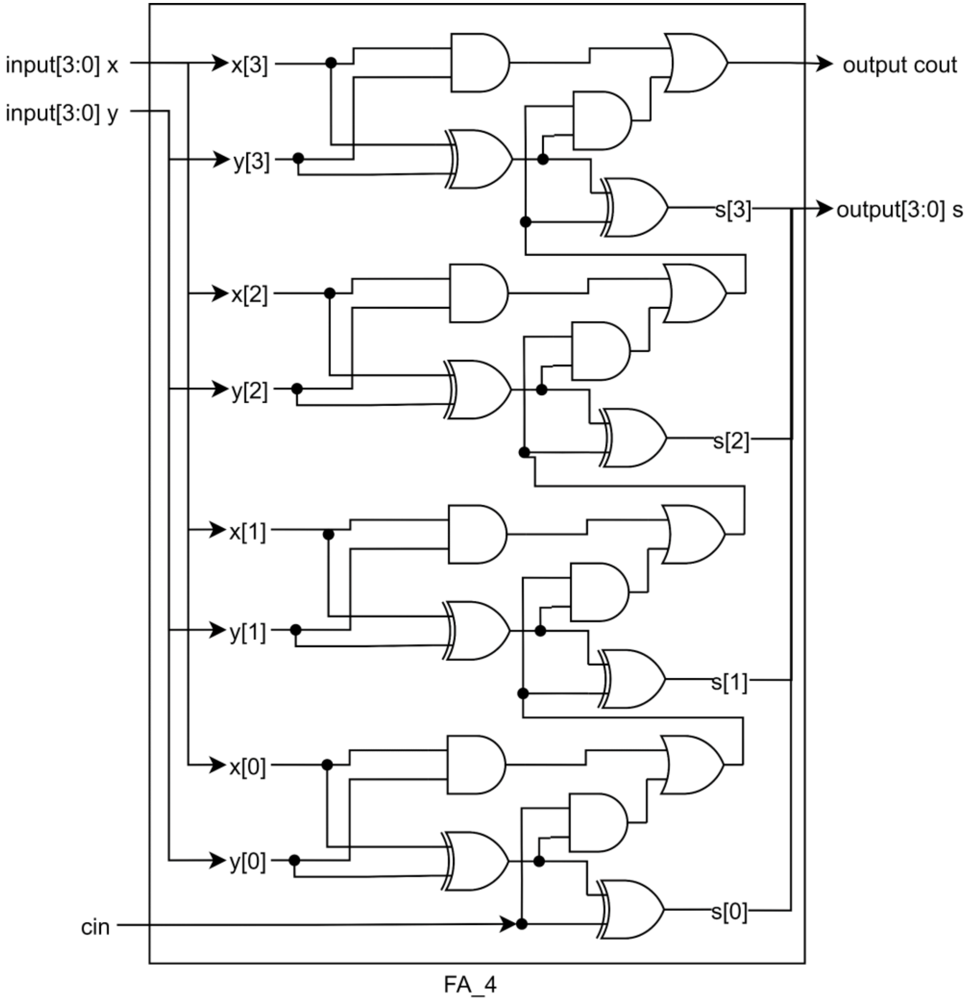
- Second, make continuous assignment according to the schematic.

```
module FA(output cout, output s, input x, input y, input cin);
    assign cout = (x & y) | (cin & (x ^ y));
    assign s = cin ^ x ^ y;
endmodule
```



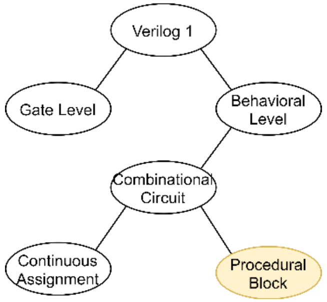
Exercise 2 - Behavior 4-bit Adder

- In `Ex2/Adder.v`, you need to finish your own module named `Adder` in behavioral model, which would perform `{cout, s} = x + y + cin`
  - Two 4-bit inputs `x` and `y`
  - One 1-bit input `cin`
  - One 4-bit output `s`
  - One 1-bit output `cout`



Further into Regs and Procedural Assignments

Can We be More Behavioral?



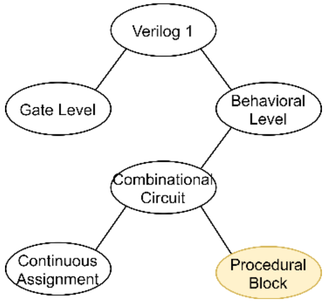
- There is one more “program-like” way to model our circuit - procedural block.

```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x, y, cin;
always @(x, y, cin) begin
    {cout, s} = x;
    {cout, s} = {cout, s} + y;
    {cout, s} = {cout, s} + cin;
end
endmodule
```

- This can also perform as a 1-bit full adder.

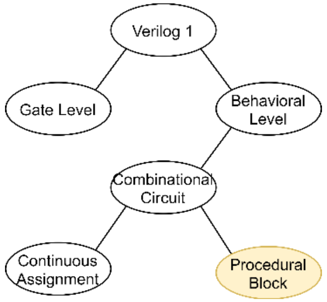
- However, you need to remember that **you are still modeling a circuit**.

Can We be More Behavioral?



- To model with procedural block, there are several things to notice:
  - `reg`
  - `always`
  - Sensitive list

```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x, y, cin;
always @(x, y, cin) begin
    {cout, s} = x;
    {cout, s} = {cout, s} + y;
    {cout, s} = {cout, s} + cin;
end
endmodule
```

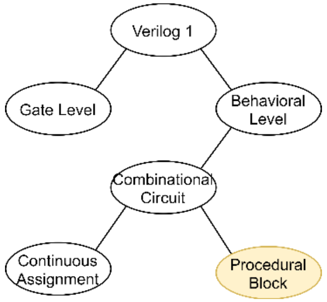


Always Block

- Because we are not **continuously attaching the LHS to RHS**, thus the LHS should be evaluated **every time this procedural block has been triggered**.
- To simulate continuous assignment, we need `always` block. The `always` means every time this procedural block is triggered, the evaluation would be done once. Then this block would wait for another trigger.

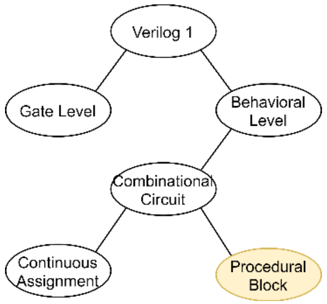
```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x, y, cin;
always @(x, y, cin) begin
    {cout, s} = x;
    {cout, s} = {cout, s} + y;
    {cout, s} = {cout, s} + cin;
end
endmodule
```

Always Block



- Because we have more than one line of content in this always block, we can use `begin` and end to define the body of this block, pretty similar to curly brackets ("{", "}") in C or Java.

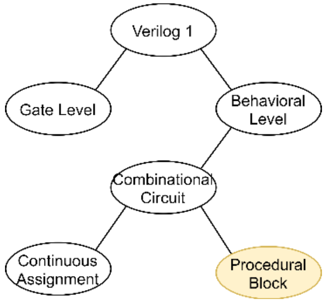
```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x, y, cin;
always @(x, y, cin) begin
    {cout, s} = x;
    {cout, s} = {cout, s} + y;
    {cout, s} = {cout, s} + cin;
end
endmodule
```



Sensitive List

- We have always block to evaluate the statements every time, but we still need to **determine when to trigger this block**.
- In sensitive list, we should put the signals that we want to keep track with. I.e., when one of `x`, `y`, `cin` changes its value, the always block would be evaluated again.

```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x, y, cin;
always @(x, y, cin) begin
    {cout, s} = x;
    {cout, s} = {cout, s} + y;
    {cout, s} = {cout, s} + cin;
end
endmodule
```

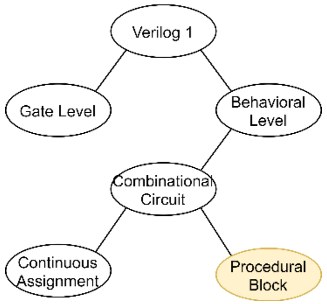


Sensitive List

- However, we usually needs to keep track of **every input because no registers here to store values**. Thus, we can use an alternative form of sensitive list to express that **we want to keep track of every input in sensitive list**.

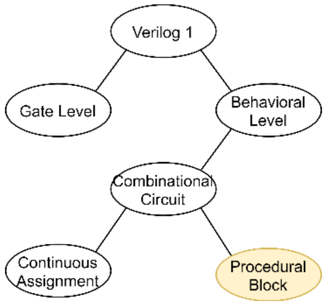
```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x, y, cin;
always @(*) begin
    {cout, s} = x;
    {cout, s} = {cout, s} + y;
    {cout, s} = {cout, s} + cin;
end
endmodule
```

Reg Type



- The `reg` here **doesn't mean register** and in this case, **it would not instantiate a register either**. This `reg` type only means after always block is evaluated, the LHS would hold the value of last evaluation

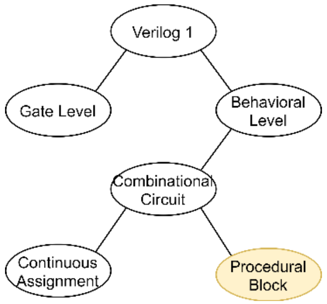
```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x, y, cin;
always @(*) begin
    {cout, s} = x;
    {cout, s} = {cout, s} + y;
    {cout, s} = {cout, s} + cin;
end
endmodule
```



Blocking/Non-blocking

- Since we are modeling a circuit **with only gates and interconnections**, we use the **blocking assignment "="** to assign the value to LHS.
- With blocking assignment, line 4 would be evaluated first, then line 5 and following would be line 6. Thus, we can perform x + y + cin.

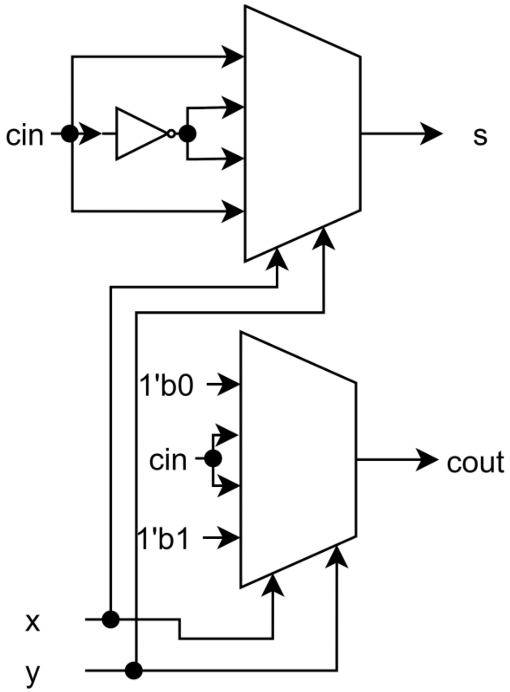
```
1   module FA(cout, s, x, y, cin);
2   output reg cout, s;
3   input x, y, cin;
4   always @(*) begin
5       {cout, s} = x;
6       {cout, s} = {cout, s} + y;
7       {cout, s} = {cout, s} + cin;
8   end
9   endmodule
```

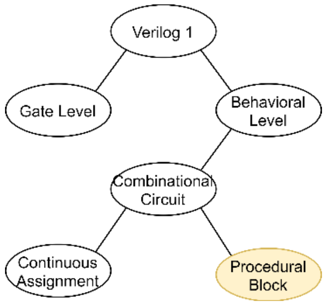


What’s the benefit of it?

- Except we can use `reg` type to make evaluation take place subsequently, we can change conditional operator into **if/else** and **case**.
- Later, I would use a mux adder to demonstrate how to use if/else and case.

```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x, y, cin;
            * begin
    case({x, y})
    2'd0: begin
        cout = 1'b0;
    end
    2'd1: begin
        cout = cin;
    end
    2'd2: begin
        cout = cin;
    end
    2'd3: begin
        cout = 1'b1;
    end
    default: begin
        cout = 1'b0;
    end
    endcase

    if (x ^ y) begin
        s = !cin;
    end
    else if (!x && !y) begin
        s = cin;
    end
    else begin
        s = cin;
    end
end
endmodule
```



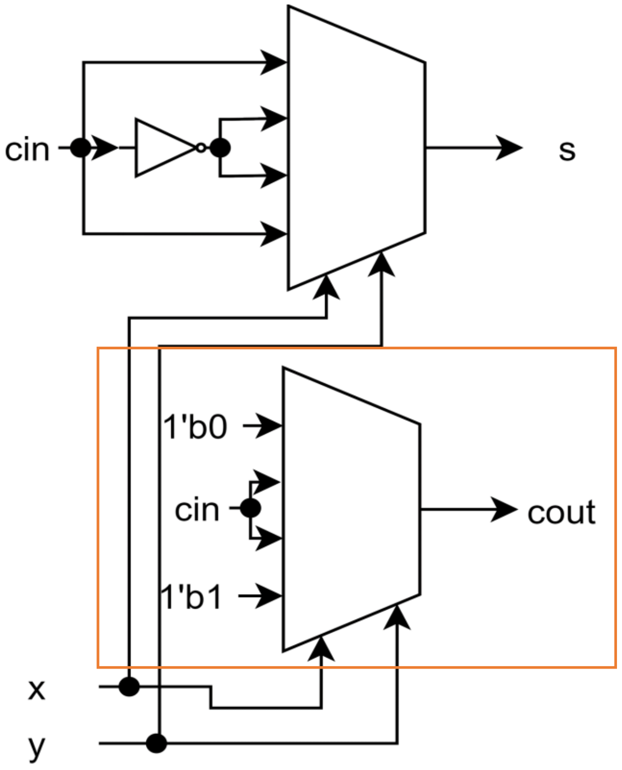


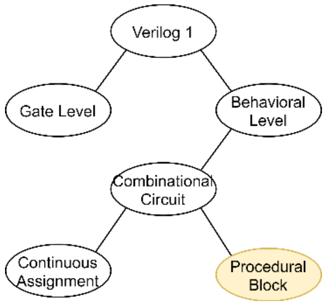
Case in Procedural Block

- For usage of case, we need to use `case(variable)` and `endcase` to enclose all cases, and use **the value** followed by **the behavior if variable matches the value** to denote all cases.

```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x, y, cin;
always @ * begin
    case({x, y})
    2'd0: begin
        cout = 1'b0;
    end
    2'd1: begin
        cout = cin;
    end
    2'd2: begin
        cout = cin;
    end
    2'd3: begin
        cout = 1'b1;
    end
    default: begin
        cout = 1'b0;
    end
    endcase

    if (x ^ y) begin
        s = !cin;
    end
    else if (!x && !y) begin
        s = cin;
    end
    else begin
        s = cin;
    end
end
endmodule
```

```
case({x, y})
2'd0: begin
    cout = 1'b0;
end
2'd1: begin
    cout = cin;
end
2'd2: begin
    cout = cin;
end
2'd3: begin
    cout = 1'b1;
end
default: begin
    cout = 1'b0;
end
endcase
```



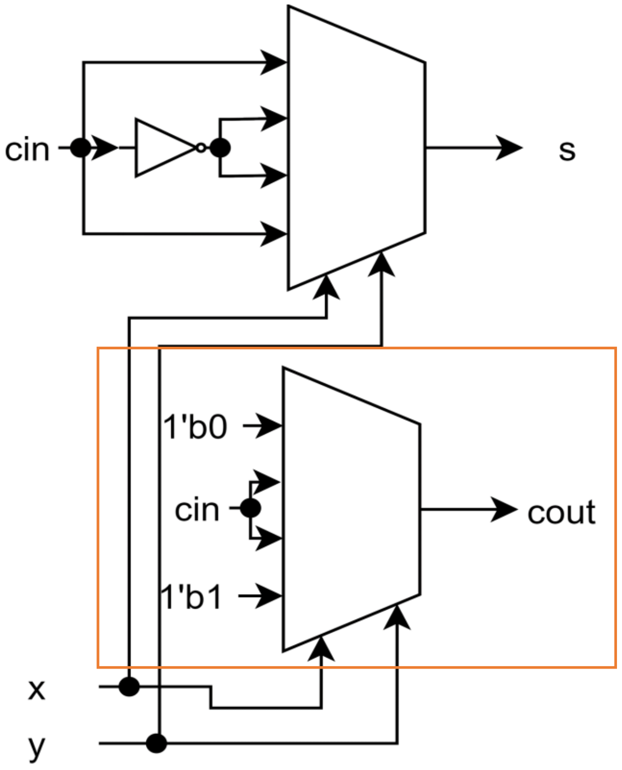


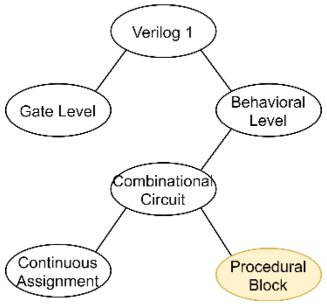
Case in Procedural Block - Default Case

- Default case is not required in this case. However, it’s a better practice to **write default case** because in a complex system, unexpected case might happen. Thus, having default case to handle them is better.

```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x, y, cin;
always @ * begin
    case({x, y})
    2'd0: begin
        cout = 1'b0;
    end
    2'd1: begin
        cout = cin;
    end
    2'd2: begin
        cout = cin;
    end
    2'd3: begin
        cout = 1'b1;
    end
    default: begin
        cout = 1'b0;
    end
    endcase

    if (x ^ y) begin
        s = !cin;
    end
    else if (!x && !y) begin
        s = cin;
    end
    else begin
        s = cin;
    end
end
endmodule
```


```
case({x, y})
2'd0: begin
    cout = 1'b0;
end
2'd1: begin
    cout = cin;
end
2'd2: begin
    cout = cin;
end
2'd3: begin
    cout = 1'b1;
end
default: begin
    cout = 1'b0;
end
endcase
```

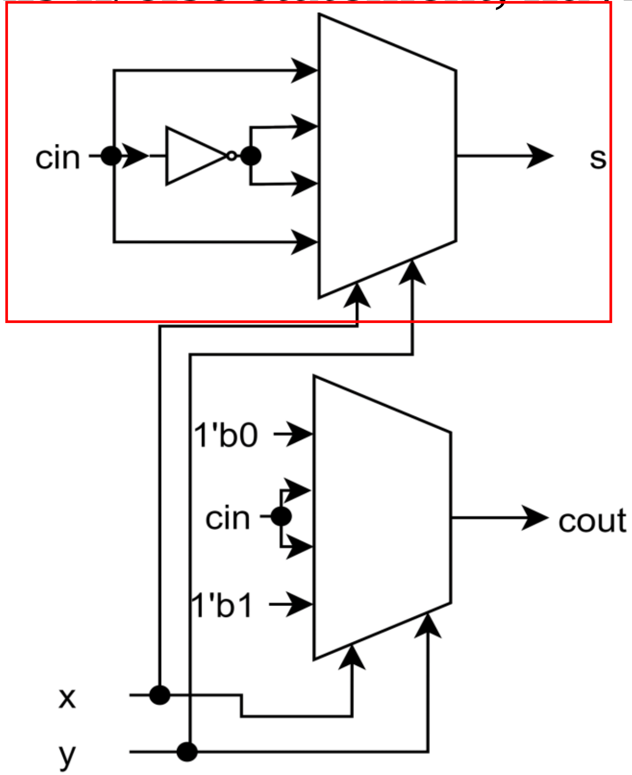


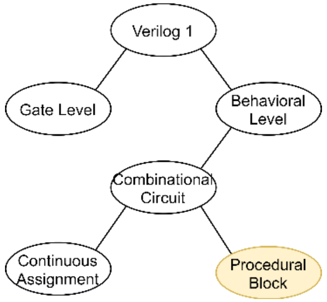
Case in Procedural Block - Default Case

- For usage of `if/else if/else`, it is quite similar to C.
- In case that you miss a case when designing this if/else statement, **having else statement is a better practice**.

```
module FA(cout, s, x, y, cin);
output reg cout, s;
input x. y, cin;
           * begin
    case({x, y})
    2'd0: begin
        cout = 1'b0;
    end
    2'd1: begin
        cout = cin;
    end
    2'd2: begin
        cout = cin;
    end
    2'd3: begin
        cout = 1'b1;
    end
    default: begin
        cout = 1'b0;
    end
    endcase

    if (x ^ y) begin
        s = !cin;
    end
    else if (!x && !y) begin
        s = cin;
    end
    else begin
        s = cin;
    end
end
endmodule
```

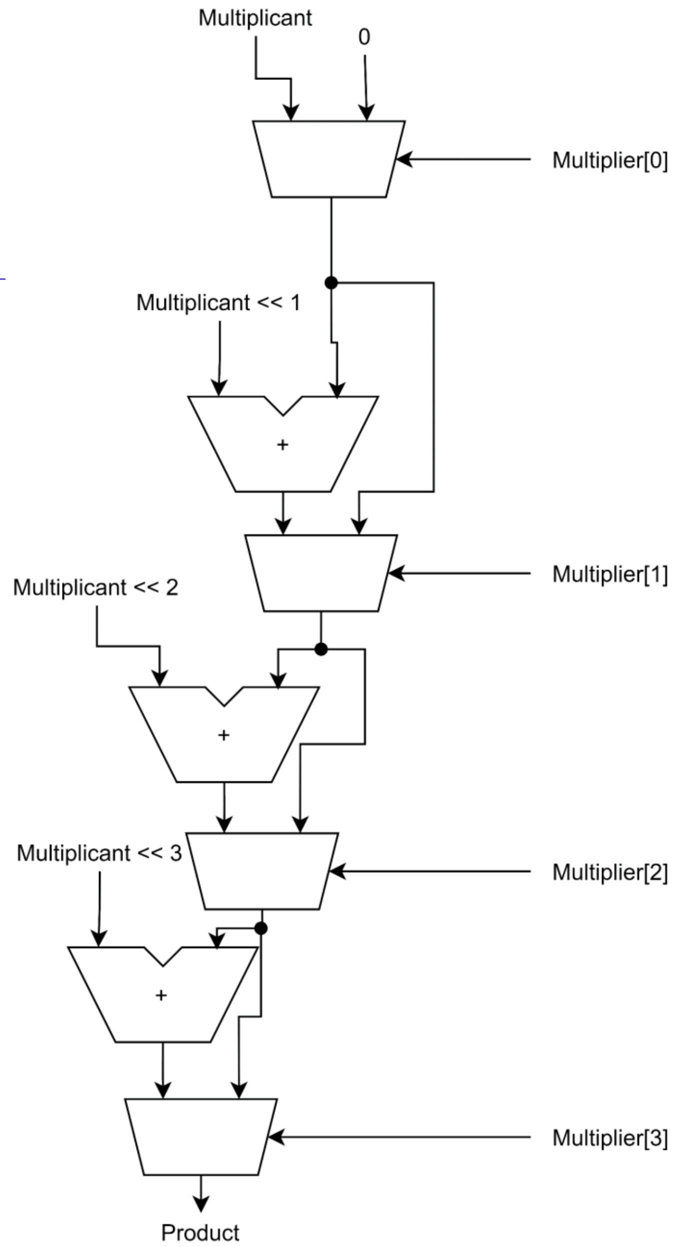
```
if (x ^ y) begin
    s = !cin;
end
else if (!x && !y) begin
    s = cin;
end
else begin
    s = cin;
end
```





Exercise 3 - Multiplier

- Multiplier in combinational
  - Write a module named **`Mult`** in **`Ex3/Mult.v`**
  - One 4-bit **`Multiplicand`** input
  - One 4-bit **`Multiplier`** input
  - One 8-bit **`Product`** output
  - Perform **`Multiplicand * Multiplier = Product`**



What Would be Taught in Coming Classes?

Expecting Materials

- Sequential Circuit with Procedural Blocks
- State Machines
- IC Contest 1999
- Pipelining

Reference

- [IEEE Standard 1364-2005]

Appendix - How to Run the Test Bench - 1

- After extracting Lab4, you should see three folders.

名稱	修改日期	類型
Ex1	2021/11/2 下午 10:25	檔案資料夾
Ex2	2021/11/2 下午 10:25	檔案資料夾
Ex3	2021/11/2 下午 10:25	檔案資料夾

- In which, taking Ex1 for example, you might see these files.

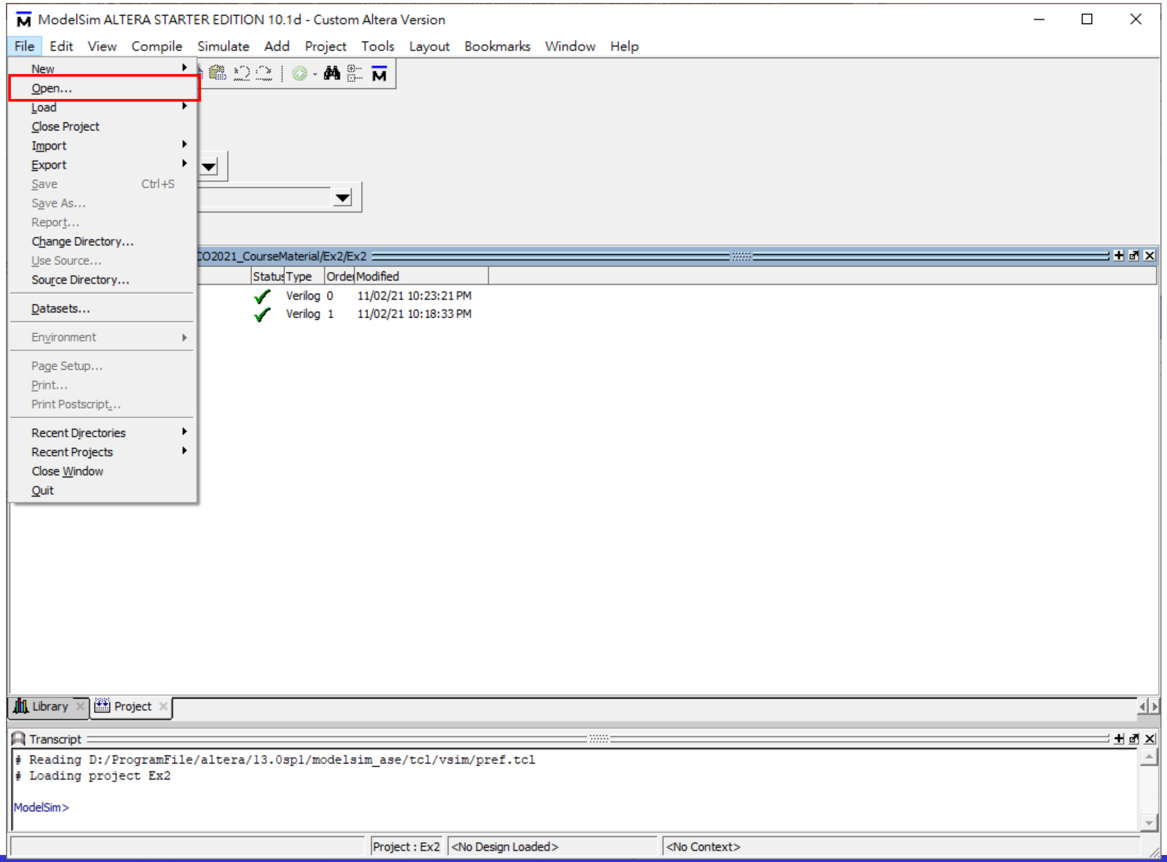
work	2021/11/2 下午 10:28	檔案資料夾	
Adder.v	2021/11/2 下午 10:28	V 檔案	0 KB
Ex1.cr.mti	2021/11/2 下午 10:21	MTI 檔案	1 KB
Ex1.mpf	2021/11/2 下午 10:21	MPF 檔案	20 KB
tb.v	2021/11/2 下午 10:18	V 檔案	3 KB

Appendix - How to Run the Test Bench - 2

- Start writing your code on **Adder.v** and **Mult.v**.

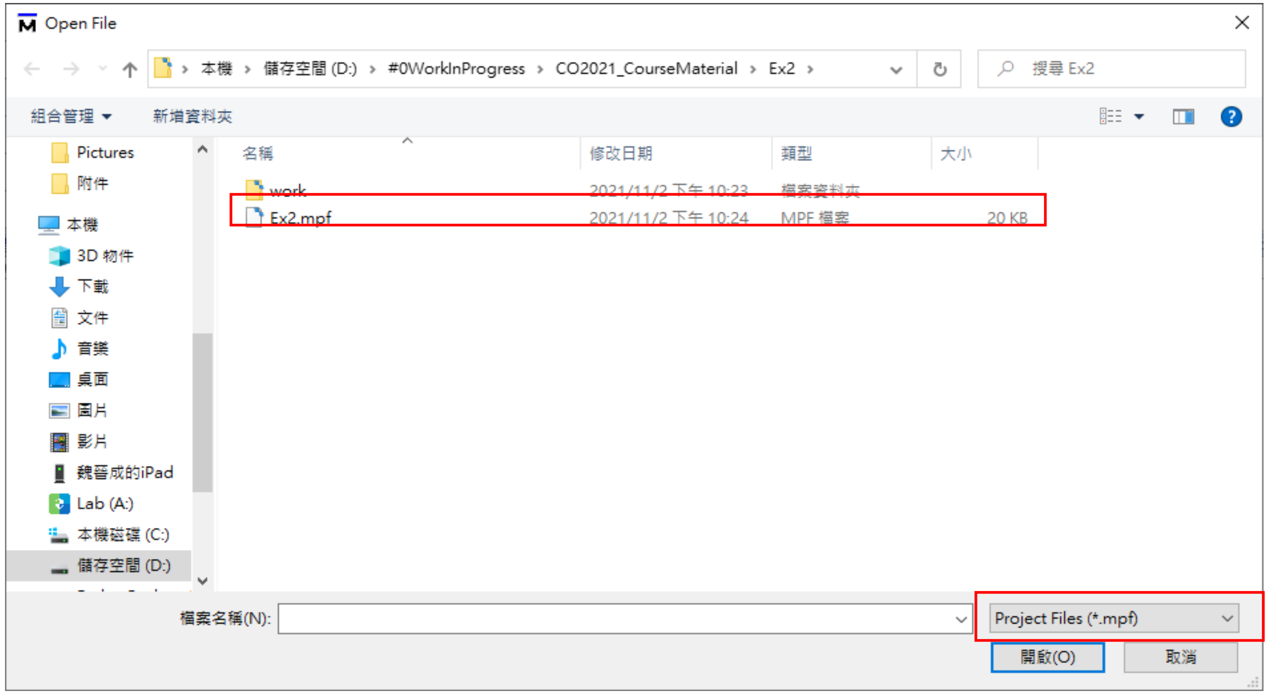
Appendix - How to Run the Test Bench - 3

- After finishing your modules, open ModelSim.
- Choose to open file.



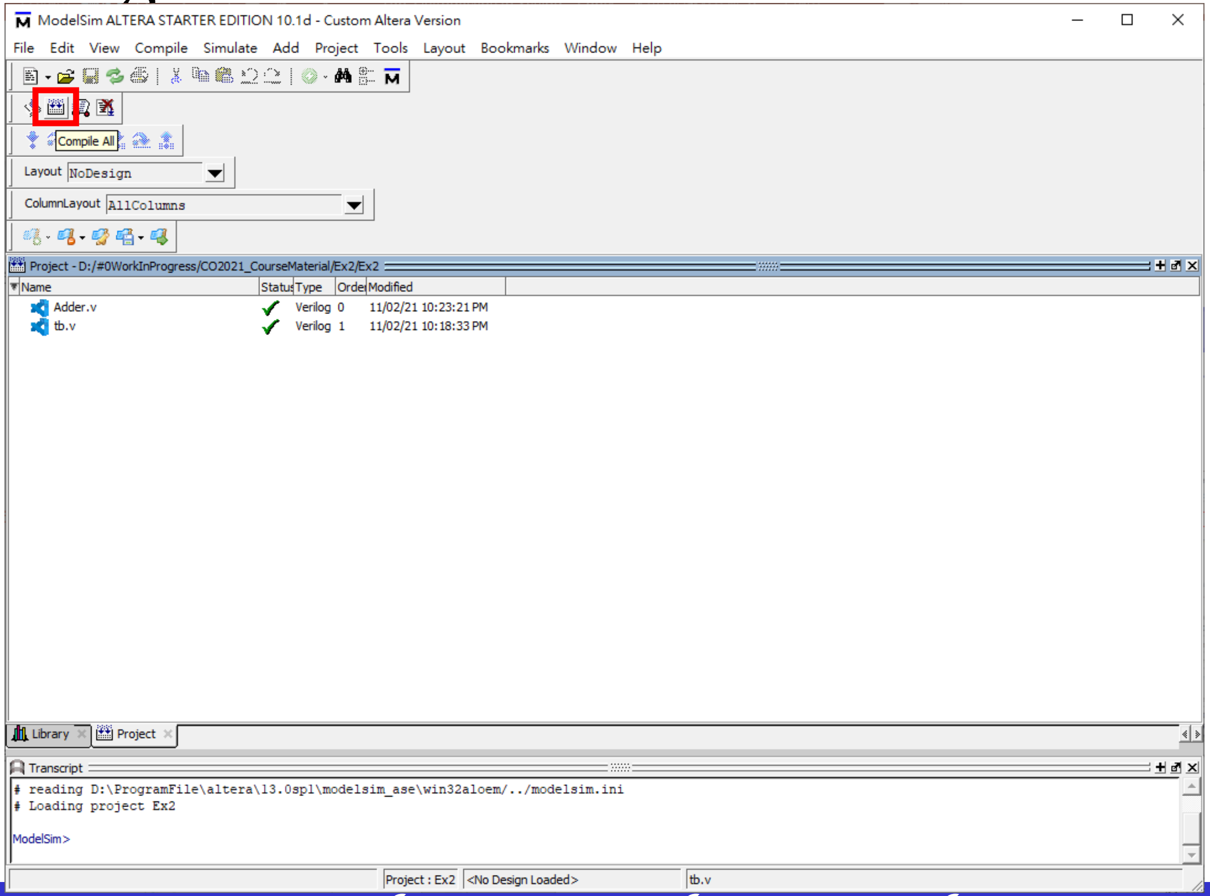
Appendix - How to Run the Test Bench - 4

- Select file type “Project File (*.mpf)” in lower right corner, you can see Ex1/2/3.mpf in corresponding folder. Click it.



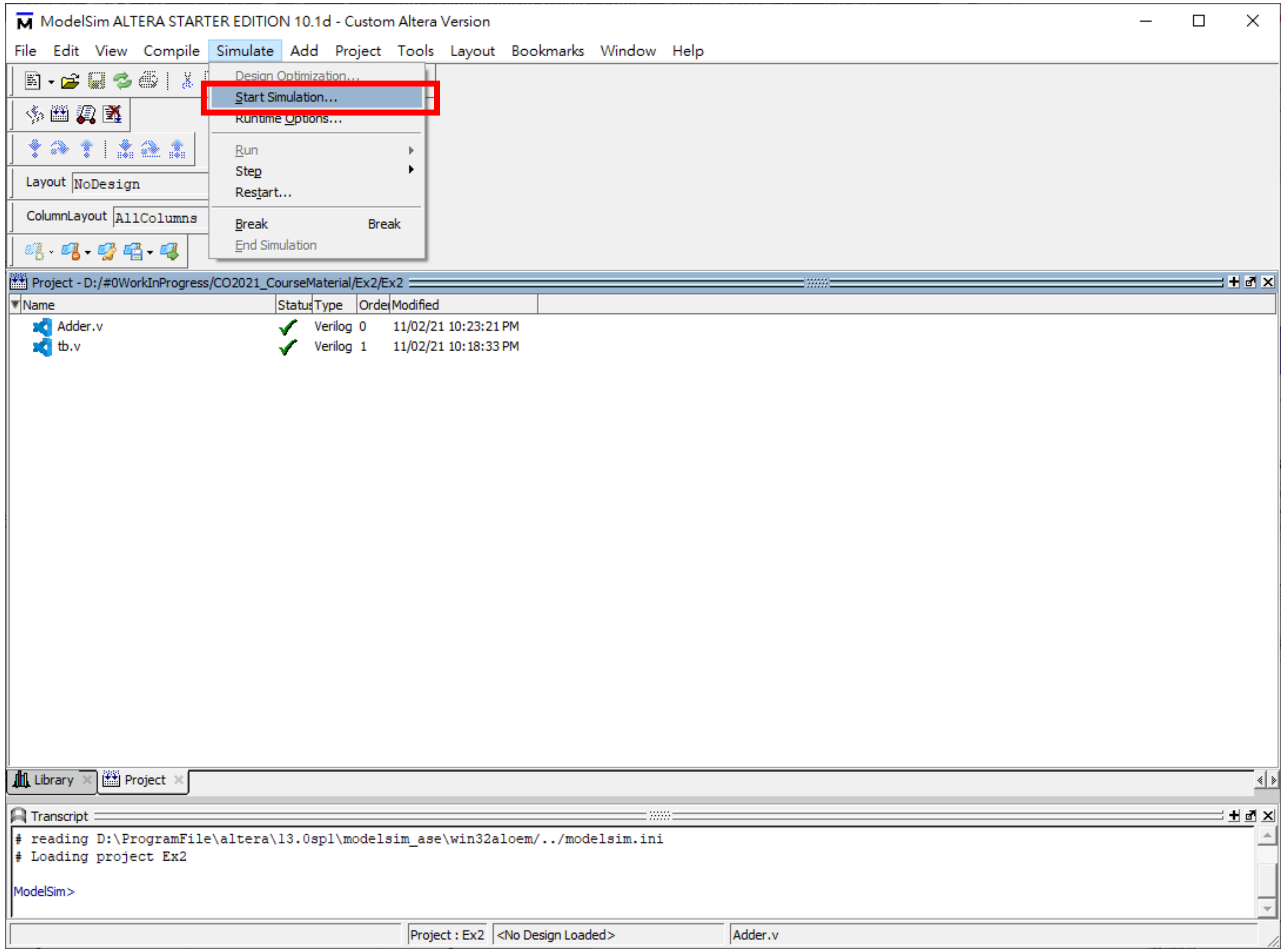
Appendix - How to Run the Test Bench - 5

- Compile your module along with test bench.



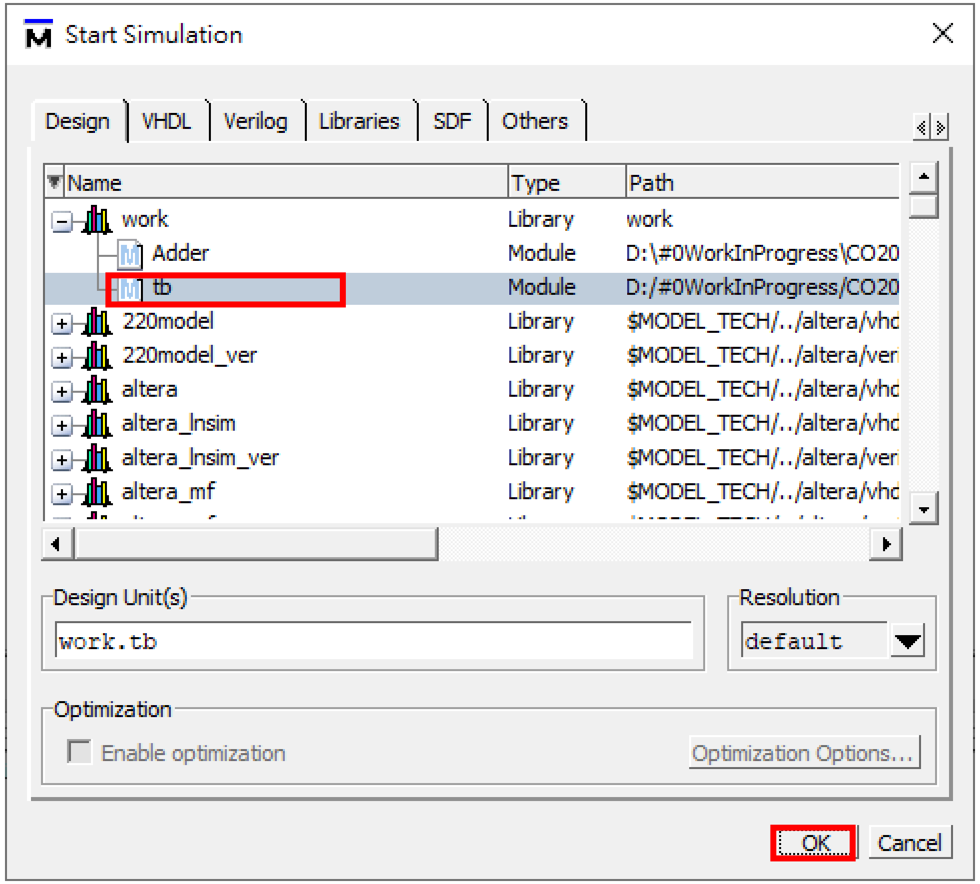
Appendix - How to Run the Test Bench - 6

- Start Simulation.



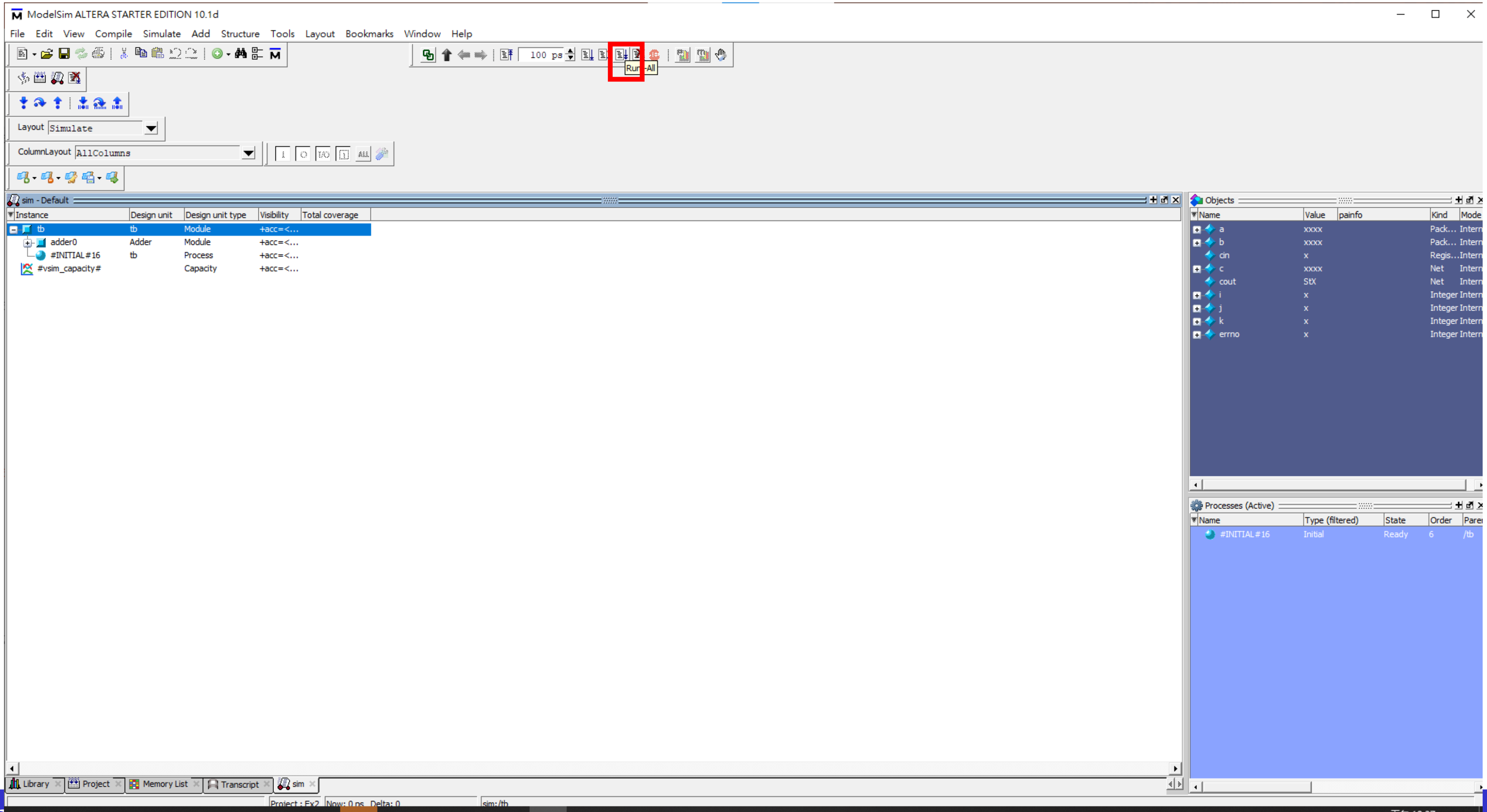
Appendix - How to Run the Test Bench - 7

- Choose tb in work.
- Click OK.



Appendix - How to Run the Test Bench - 8

- Run all~



Appendix - How to Run the Test Bench - 9

- If something went wrong, the input and golden would be displayed on “Transcript” tab.

```
#  instead of cout: 0, s: 12
# Error:          15 +         5 +          0 should be cout: 1, s:          4
#  instead of cout: 0, s: 10
# Error:          15 +         5 +          1 should be cout: 1, s:          5
#  instead of cout: 0, s: 11
# Error:          15 +         6 +          0 should be cout: 1, s:          5
#  instead of cout: 0, s:  9
# Error:          15 +         6 +          1 should be cout: 1, s:          6
#  instead of cout: 0, s: 10
# Error:          15 +         7 +          0 should be cout: 1, s:          6
#  instead of cout: 0, s:  8
# Error:          15 +         7 +          1 should be cout: 1, s:          7
#  instead of cout: 0, s:  9
# Error:          15 +         8 +          0 should be cout: 1, s:          7
#  instead of cout: 0, s:  7
# Error:          15 +         8 +          1 should be cout: 1, s:          8
#  instead of cout: 0, s:  8
# Error:          15 +         9 +          0 should be cout: 1, s:          8
#  instead of cout: 0, s:  6
# Error:          15 +         9 +          1 should be cout: 1, s:          9
#  instead of cout: 0, s:  7
# Error:          15 +        10 +          0 should be cout: 1, s:          9
#  instead of cout: 0, s:  5
# Error:          15 +        10 +          1 should be cout: 1, s:         10
#  instead of cout: 0, s:  6
# Error:          15 +        11 +          0 should be cout: 1, s:         10
#  instead of cout: 0, s:  4
# Error:          15 +        11 +          1 should be cout: 1, s:         11
#  instead of cout: 0, s:  5
# Error:          15 +        12 +          0 should be cout: 1, s:         11
#  instead of cout: 0, s:  3
# Error:          15 +        12 +          1 should be cout: 1, s:         12
#  instead of cout: 0, s:  4
# Error:          15 +        13 +          0 should be cout: 1, s:         12
#  instead of cout: 0, s:  2
# Error:          15 +        13 +          1 should be cout: 1, s:         13
#  instead of cout: 0, s:  3
# Error:          15 +        14 +          0 should be cout: 1, s:         13
#  instead of cout: 0, s:  1
# Error:          15 +        14 +          1 should be cout: 1, s:         14
#  instead of cout: 0, s:  2
# Error:          15 +        15 +          0 should be cout: 1, s:         14
#  instead of cout: 0, s:  0
# Error:          15 +        15 +          1 should be cout: 1, s:         15
#  instead of cout: 0, s:  1
# Total Error:          480
```

Appendix - How to Run the Test Bench - 10

- If all data are correct, the outcome would be like this in tab “Transcript”