Logic Design

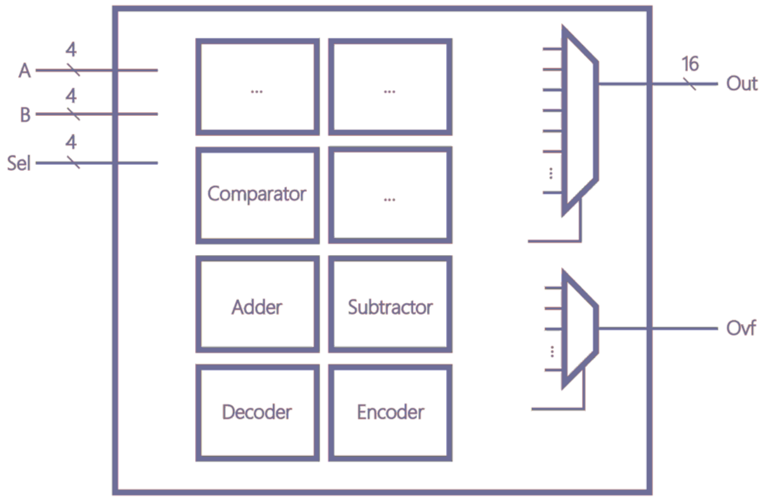
EECS101001

Lab 2: Applications of Multiplexer – Report

104021215 熊 磊

1. Foreword

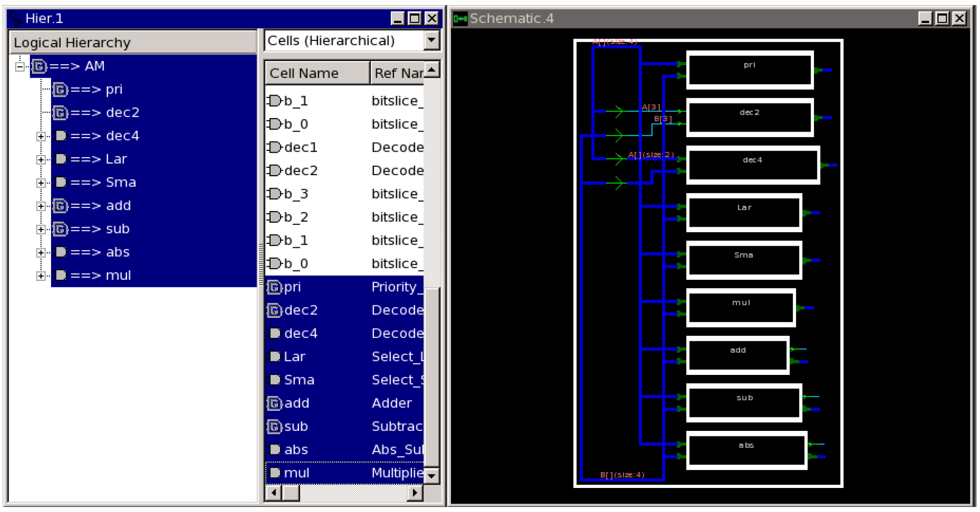
This Lab is trying to use Multiplexer (or Selector) to select the mudule by Sel with input A, B, and output is Out, Ovf.



2. Implementation Details and Designs

- Multiplexer

  I use always block to implement the Multiplexer. The code is too long to show here, one can check AM.v in line #25~#109. Below is the image created by Design Vision.



Hier.1

Logical Hierarchy

```
==> AM
  ==> pri
  ==> dec2
  ==> dec4
  ==> Lar
  ==> Sma
  ==> add
  ==> sub
  ==> abs
  ==> mul
```

Cells (Hierarchical)

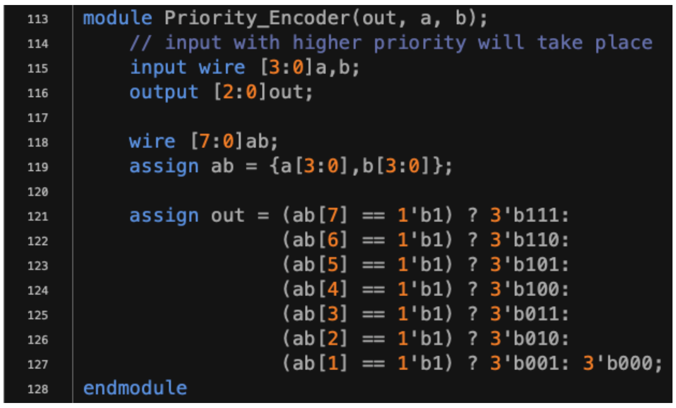
Cell Name	Ref Nar
b_1	bitslice_
b_0	bitslice_
dec1	Decode
dec2	Decode
b_3	bitslice_
b_2	bitslice_
b_1	bitslice_
b_0	bitslice_
pri	Priority_
dec2	Decode
dec4	Decode
Lar	Select_L
Sma	Select_S
add	Adder
sub	Subtrac
abs	Abs_Sul
mul	Multiplie

Hence, we need to implement each module for each module.

- [0000] Priority Encoder

  First I easily combine the two inputs a, b to wire[7:0]ab.

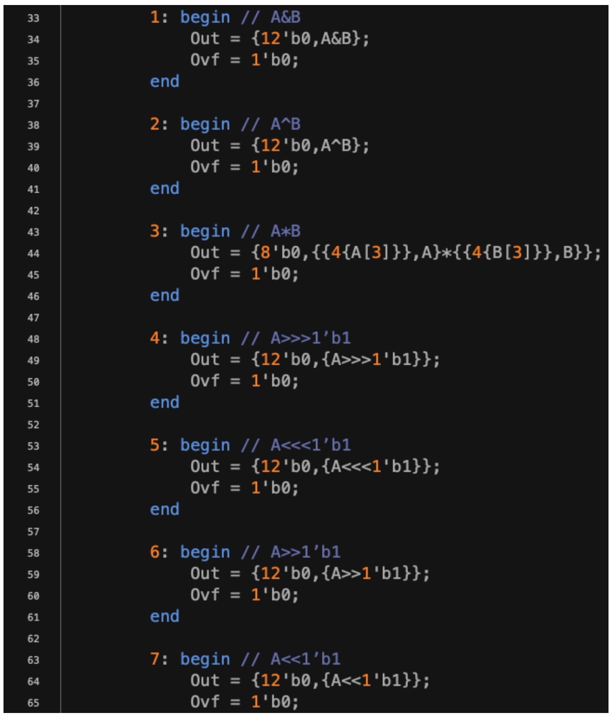
Then one can get the result of priority encoder by assign method check the bit form MSB to LSB one by one.



```verilog
module Priority_Encoder(out, a, b);
    // input with higher priority will take place
    input wire [3:0]a,b;
    output [2:0]out;

    wire [7:0]ab;
    assign ab = {a[3:0],b[3:0]};

    assign out = (ab[7] == 1'b1) ? 3'b111:
                 (ab[6] == 1'b1) ? 3'b110:
                 (ab[5] == 1'b1) ? 3'b101:
                 (ab[4] == 1'b1) ? 3'b100:
                 (ab[3] == 1'b1) ? 3'b011:
                 (ab[2] == 1'b1) ? 3'b010:
                 (ab[1] == 1'b1) ? 3'b001: 3'b000;
endmodule
```

- `[0001]~[0111]`A&B, A^B, A*B, A>>>1’ b1, …, A<<1’ b1

For these functions, without restrictions, I directly use Verilog built-in operation.

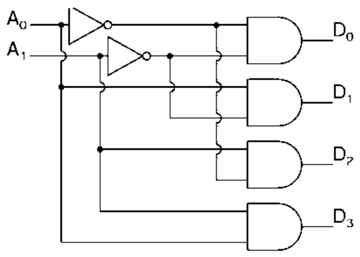


```verilog
        1: begin // A&B
            Out = {12'b0,A&B};
            Ovf = 1'b0;
        end

        2: begin // A^B
            Out = {12'b0,A^B};
            Ovf = 1'b0;
        end

        3: begin // A*B
            Out = {8'b0,{{4{A[3]}},A}*{{4{B[3]}},B}};
            Ovf = 1'b0;
        end

        4: begin // A>>>1'b1
            Out = {12'b0,{A>>>1'b1}};
            Ovf = 1'b0;
        end

        5: begin // A<<<1'b1
            Out = {12'b0,{A<<<1'b1}};
            Ovf = 1'b0;
        end

        6: begin // A>>1'b1
            Out = {12'b0,{A>>1'b1}};
            Ovf = 1'b0;
        end

        7: begin // A<<1'b1
            Out = {12'b0,{A<<1'b1}};
            Ovf = 1'b0;
```

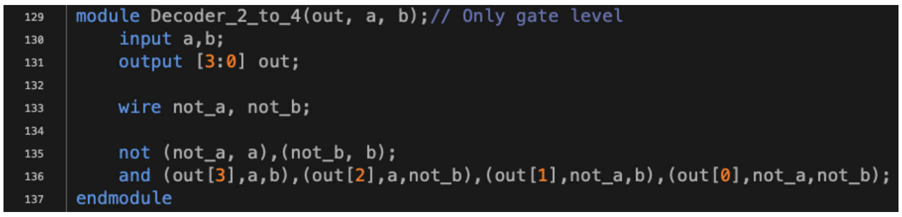
- `[1000]` 2-to-4 decoder

We need to use **only gate level** to implement 2-to-4 decoder, we first write down the truth table and draw a simple diagram below.

A_0	A_1	D_0	D_1	D_2	D_3
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



Therefore, we need four **and gate** and two **not gate**. The code is below.

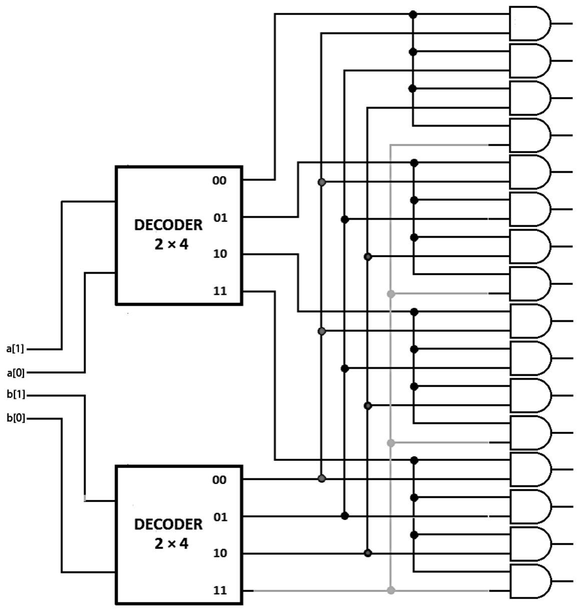


```
module Decoder_2_to_4(out, a, b);// Only gate level
    input a,b;
    output [3:0] out;

    wire not_a, not_b;

    not (not_a, a),(not_b, b);
    and (out[3],a,b),(out[2],a,not_b),(out[1],not_a,b),(out[0],not_a,not_b);
endmodule
```

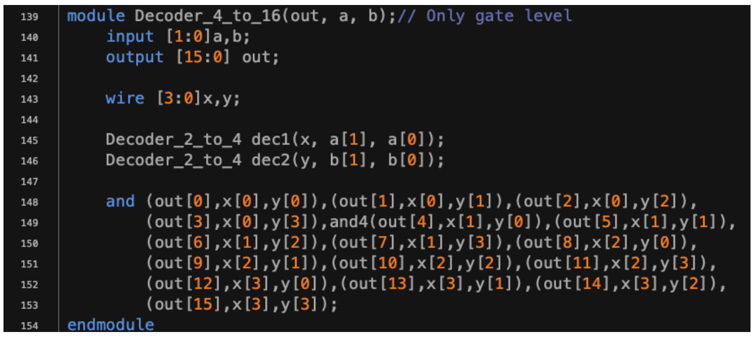
- [1001] 4-to-16 decoder

  By reusing 2-to-4 decoder, the idea is very simple, we can draw a diagram below.

A[3:2]	B[3:2]	Out
00	00	0000_0000_0000_0001
00	01	0000_0000_0000_0010
00	10	0000_0000_0000_0100
00	11	0000_0000_0000_1000
01	00	0000_0000_0001_0000
01	01	0000_0000_0010_0000
01	10	0000_0000_0100_0000
01	11	0000_0000_1000_0000
10	00	0000_0001_0000_0000
10	01	0000_0010_0000_0000
10	10	0000_0100_0000_0000
10	11	0000_1000_0000_0000
11	00	0001_0000_0000_0000
11	01	0010_0000_0000_0000
11	10	0100_0000_0000_0000
11	11	1000_0000_0000_0000



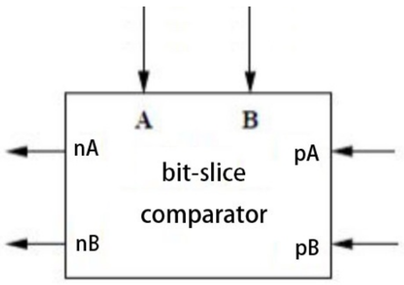
Therefore, we need sixteen **and gate** and two **2-4 Decoder**. The code is below.

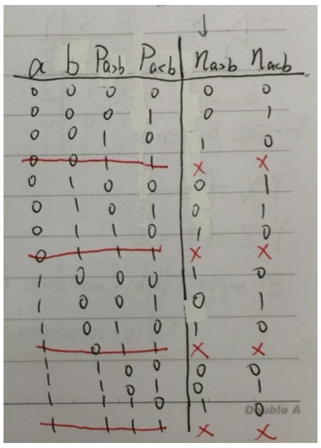


```verilog
module Decoder_4_to_16(out, a, b);// Only gate level
    input [1:0]a,b;
    output [15:0] out;

    wire [3:0]x,y;

    Decoder_2_to_4 dec1(x, a[1], a[0]);
    Decoder_2_to_4 dec2(y, b[1], b[0]);

    and (out[0],x[0],y[0]),(out[1],x[0],y[1]),(out[2],x[0],y[2]),
        (out[3],x[0],y[3]),and4(out[4],x[1],y[0]),(out[5],x[1],y[1]),
        (out[6],x[1],y[2]),(out[7],x[1],y[3]),(out[8],x[2],y[0]),
        (out[9],x[2],y[1]),(out[10],x[2],y[2]),(out[11],x[2],y[3]),
        (out[12],x[3],y[0]),(out[13],x[3],y[1]),(out[14],x[3],y[2]),
        (out[15],x[3],y[3]);
endmodule
```

- [1010]~[1011]Select the larger number, Select the smaller number

  [Design Bit Slice - Comparator]

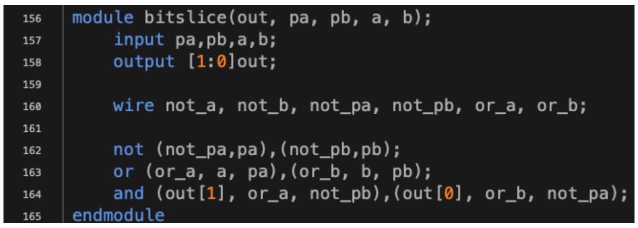




a	b	$P_{a>b}$	$P_{a<b}$	$n_{a>b}$	$n_{a<b}$
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
~~0~~	~~0~~	~~1~~	~~1~~	X	X
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	0
~~0~~	~~1~~	~~1~~	~~1~~	X	X
1	0	0	0	1	0
1	0	0	1	0	1
1	0	1	0	1	0
~~1~~	~~0~~	~~1~~	~~1~~	X	X
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
~~1~~	~~1~~	~~1~~	~~1~~	X	X

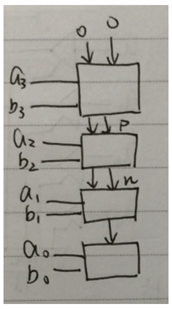
nA is equivalent to (A+pA)&pB’ nB is equivalent to (B+pB)&pA’

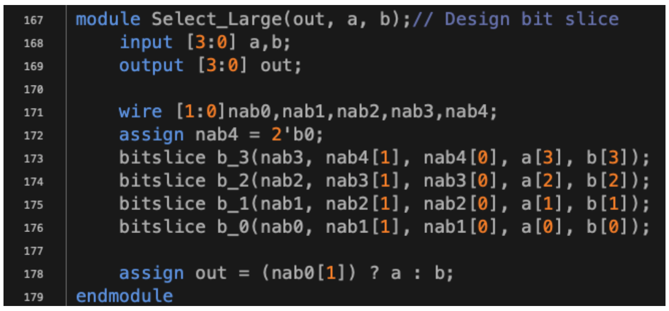
If nA is 1, A>B; if nB is 1, B>A.



```verilog
module bitslice(out, pa, pb, a, b);
    input pa,pb,a,b;
    output [1:0]out;

    wire not_a, not_b, not_pa, not_pb, or_a, or_b;

    not (not_pa,pa),(not_pb,pb);
    or (or_a, a, pa),(or_b, b, pb);
    and (out[1], or_a, not_pb),(out[0], or_b, not_pa);
endmodule
```

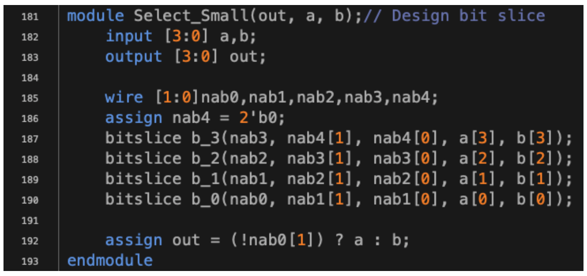
So, we can use Comparator repeatedly, like this. For example, **Select the larger number**, We just have to check if **nA** is 1.





```verilog
module Select_Large(out, a, b);// Design bit slice
    input [3:0] a,b;
    output [3:0] out;

    wire [1:0]nab0,nab1,nab2,nab3,nab4;
    assign nab4 = 2'b0;
    bitslice b_3(nab3, nab4[1], nab4[0], a[3], b[3]);
    bitslice b_2(nab2, nab3[1], nab3[0], a[2], b[2]);
    bitslice b_1(nab1, nab2[1], nab2[0], a[1], b[1]);
    bitslice b_0(nab0, nab1[1], nab1[0], a[0], b[0]);

    assign out = (nab0[1]) ? a : b;
endmodule
```

On the other hand, **Select the smaller number**, we can just choose the one not the large.

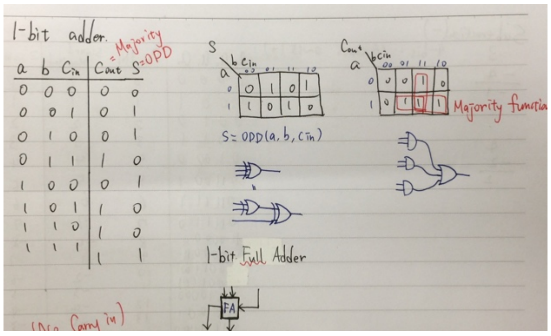


```verilog
module Select_Small(out, a, b);// Design bit slice
    input [3:0] a,b;
    output [3:0] out;

    wire [1:0]nab0,nab1,nab2,nab3,nab4;
    assign nab4 = 2'b0;
    bitslice b_3(nab3, nab4[1], nab4[0], a[3], b[3]);
    bitslice b_2(nab2, nab3[1], nab3[0], a[2], b[2]);
    bitslice b_1(nab1, nab2[1], nab2[0], a[1], b[1]);
    bitslice b_0(nab0, nab1[1], nab1[0], a[0], b[0]);

    assign out = (!nab0[1]) ? a : b;
endmodule
```

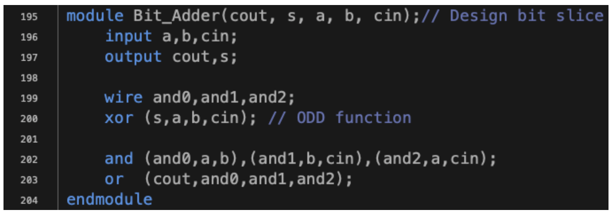
- `[1100]`A+B

  [Design Bit Slice – Full Adder]

  Here we first analyze **Cout** and **s**. By drawing a truth table and K-map, found that they are Majority function and ODD function respectively.

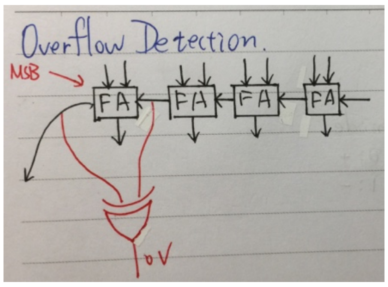


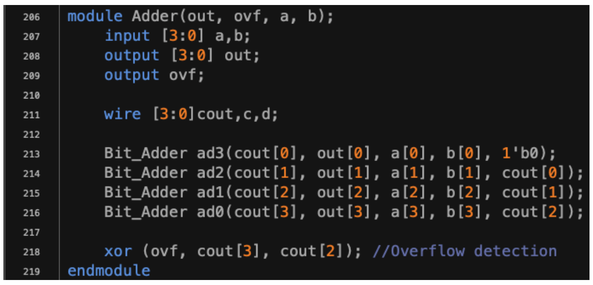
a	b	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



```verilog
module Bit_Adder(cout, s, a, b, cin);// Design bit slice
    input a,b,cin;
    output cout,s;

    wire and0,and1,and2;
    xor (s,a,b,cin); // ODD function

    and (and0,a,b),(and1,b,cin),(and2,a,cin);
    or  (cout,and0,and1,and2);
endmodule
```

  Now, we can use Full adder repeatedly for four-digits input to implement **Adder**. To detect overflow, using an **xor** gate at the last Full Adder.



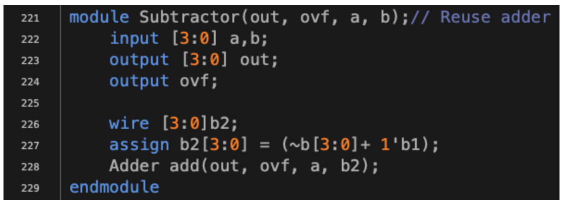


```verilog
module Adder(out, ovf, a, b);
    input [3:0] a,b;
    output [3:0] out;
    output ovf;

    wire [3:0]cout,c,d;

    Bit_Adder ad3(cout[0], out[0], a[0], b[0], 1'b0);
    Bit_Adder ad2(cout[1], out[1], a[1], b[1], cout[0]);
    Bit_Adder ad1(cout[2], out[2], a[2], b[2], cout[1]);
    Bit_Adder ad0(cout[3], out[3], a[3], b[3], cout[2]);

    xor (ovf, cout[3], cout[2]); //Overflow detection
endmodule
```

- `[1101]`A-B

  [Reuse Adder]

It’ s easy to implement Subtractor by reusing Adder. We just change the problem A-B to the problem A+(-B). Just like line #227 below.



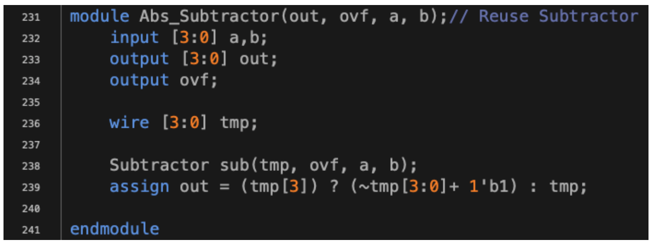
```
221 module Subtractor(out, ovf, a, b);// Reuse adder
222     input [3:0] a,b;
223     output [3:0] out;
224     output ovf;
225 
226     wire [3:0]b2;
227     assign b2[3:0] = (~b[3:0]+ 1'b1);
228     Adder add(out, ovf, a, b2);
229 endmodule
```

- [1110] | A-B |

  [Reuse Subtractor]

  If overflow, we don’ t care about the output.

  The only step we need to care is to make sure the output is positive. Therefore, convert to positive if it’ s negative. As show in Line #239.

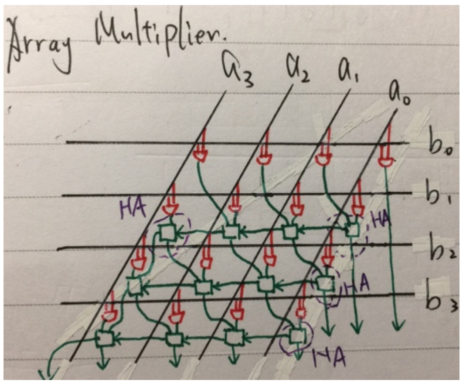


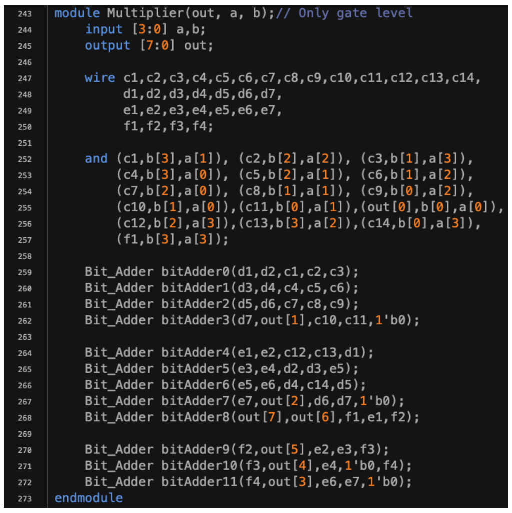
```
231 module Abs_Subtractor(out, ovf, a, b);// Reuse Subtractor
232     input [3:0] a,b;
233     output [3:0] out;
234     output ovf;
235 
236     wire [3:0] tmp;
237 
238     Subtractor sub(tmp, ovf, a, b);
239     assign out = (tmp[3]) ? (~tmp[3:0]+ 1'b1) : tmp;
240 
241 endmodule
```

- [1111] A*B

  [Only Gate Level]

  I use array Multiplier to implement. The diagram is below. It’ s hard to say in text, but the main idea is to reuse Full Adder(made by gate level) to control bit slice, and use and gate to do bit-multiplying.

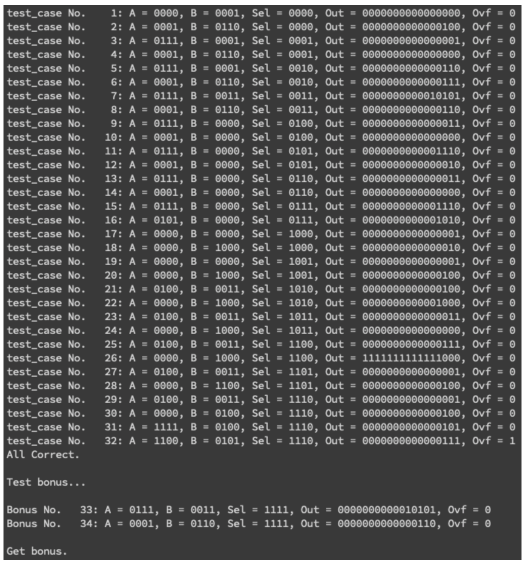




```
243 module Multiplier(out, a, b);// Only gate level
244     input [3:0] a,b;
245     output [7:0] out;
246 
247     wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,
248          d1,d2,d3,d4,d5,d6,d7,
249          e1,e2,e3,e4,e5,e6,e7,
250          f1,f2,f3,f4;
251 
252     and (c1,b[3],a[1]), (c2,b[2],a[2]), (c3,b[1],a[3]),
253         (c4,b[3],a[0]), (c5,b[2],a[1]), (c6,b[1],a[2]),
254         (c7,b[2],a[0]), (c8,b[1],a[1]), (c9,b[0],a[2]),
255         (c10,b[1],a[0]),(c11,b[0],a[1]),(out[0],b[0],a[0]),
256         (c12,b[2],a[3]),(c13,b[3],a[2]),(c14,b[0],a[3]),
257         (f1,b[3],a[3]);
258 
259     Bit_Adder bitAdder0(d1,d2,c1,c2,c3);
260     Bit_Adder bitAdder1(d3,d4,c4,c5,c6);
261     Bit_Adder bitAdder2(d5,d6,c7,c8,c9);
262     Bit_Adder bitAdder3(d7,out[1],c10,c11,1'b0);
263 
264     Bit_Adder bitAdder4(e1,e2,c12,c13,d1);
265     Bit_Adder bitAdder5(e3,e4,d2,d3,e5);
266     Bit_Adder bitAdder6(e5,e6,d4,c14,d5);
267     Bit_Adder bitAdder7(e7,out[2],d6,d7,1'b0);
268     Bit_Adder bitAdder8(out[7],out[6],f1,e1,f2);
269 
270     Bit_Adder bitAdder9(f2,out[5],e2,e3,f3);
271     Bit_Adder bitAdder10(f3,out[4],e4,1'b0,f4);
272     Bit_Adder bitAdder11(f4,out[3],e6,e7,1'b0);
273 endmodule
```

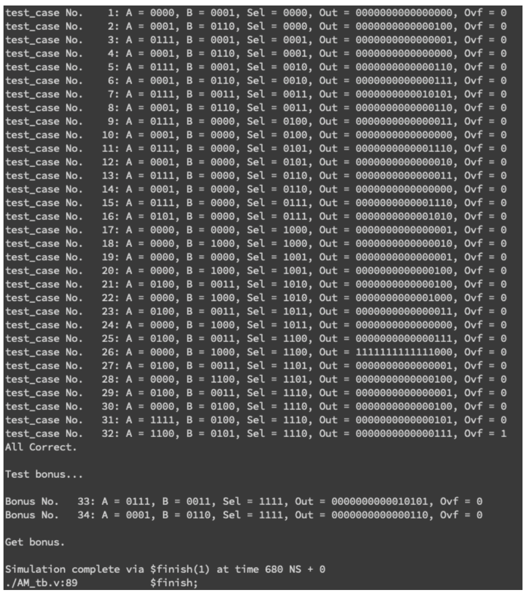
3. Simulation Result

[Before Synthesis]



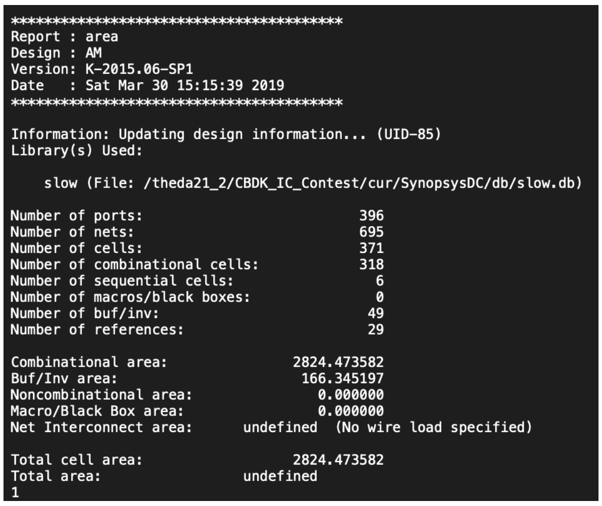
```
test_case No.    1: A = 0000, B = 0001, Sel = 0000, Out = 0000000000000000, Ovf = 0
test_case No.    2: A = 0001, B = 0110, Sel = 0000, Out = 0000000000000100, Ovf = 0
test_case No.    3: A = 0111, B = 0001, Sel = 0001, Out = 0000000000000001, Ovf = 0
test_case No.    4: A = 0001, B = 0110, Sel = 0001, Out = 0000000000000000, Ovf = 0
test_case No.    5: A = 0111, B = 0001, Sel = 0010, Out = 0000000000000110, Ovf = 0
test_case No.    6: A = 0001, B = 0110, Sel = 0010, Out = 0000000000000111, Ovf = 0
test_case No.    7: A = 0111, B = 0011, Sel = 0011, Out = 0000000000010101, Ovf = 0
test_case No.    8: A = 0001, B = 0110, Sel = 0011, Out = 0000000000000110, Ovf = 0
test_case No.    9: A = 0111, B = 0000, Sel = 0100, Out = 0000000000000011, Ovf = 0
test_case No.   10: A = 0001, B = 0000, Sel = 0100, Out = 0000000000000000, Ovf = 0
test_case No.   11: A = 0111, B = 0000, Sel = 0101, Out = 0000000000001110, Ovf = 0
test_case No.   12: A = 0001, B = 0000, Sel = 0101, Out = 0000000000000010, Ovf = 0
test_case No.   13: A = 0111, B = 0000, Sel = 0110, Out = 0000000000000011, Ovf = 0
test_case No.   14: A = 0001, B = 0000, Sel = 0110, Out = 0000000000000000, Ovf = 0
test_case No.   15: A = 0111, B = 0000, Sel = 0111, Out = 0000000000001110, Ovf = 0
test_case No.   16: A = 0101, B = 0000, Sel = 0111, Out = 0000000000001010, Ovf = 0
test_case No.   17: A = 0000, B = 0000, Sel = 1000, Out = 0000000000000001, Ovf = 0
test_case No.   18: A = 0000, B = 1000, Sel = 1000, Out = 0000000000000010, Ovf = 0
test_case No.   19: A = 0000, B = 0000, Sel = 1001, Out = 0000000000000001, Ovf = 0
test_case No.   20: A = 0000, B = 1000, Sel = 1001, Out = 0000000000000100, Ovf = 0
test_case No.   21: A = 0100, B = 0011, Sel = 1010, Out = 0000000000000100, Ovf = 0
test_case No.   22: A = 0000, B = 1000, Sel = 1010, Out = 0000000000001000, Ovf = 0
test_case No.   23: A = 0100, B = 0011, Sel = 1011, Out = 0000000000000011, Ovf = 0
test_case No.   24: A = 0000, B = 1000, Sel = 1011, Out = 0000000000000000, Ovf = 0
test_case No.   25: A = 0100, B = 0011, Sel = 1100, Out = 0000000000000111, Ovf = 0
test_case No.   26: A = 0000, B = 1000, Sel = 1100, Out = 1111111111111000, Ovf = 0
test_case No.   27: A = 0100, B = 0011, Sel = 1101, Out = 0000000000000001, Ovf = 0
test_case No.   28: A = 0000, B = 1100, Sel = 1101, Out = 0000000000000100, Ovf = 0
test_case No.   29: A = 0100, B = 0011, Sel = 1110, Out = 0000000000000001, Ovf = 0
test_case No.   30: A = 0000, B = 0100, Sel = 1110, Out = 0000000000000100, Ovf = 0
test_case No.   31: A = 1111, B = 0100, Sel = 1110, Out = 0000000000000101, Ovf = 0
test_case No.   32: A = 1100, B = 0101, Sel = 1110, Out = 0000000000000111, Ovf = 1
All Correct.

Test bonus...

Bonus No.   33: A = 0111, B = 0011, Sel = 1111, Out = 0000000000010101, Ovf = 0
Bonus No.   34: A = 0001, B = 0110, Sel = 1111, Out = 0000000000000110, Ovf = 0

Get bonus.
```

[Run Synthesis]



```
test_case No.    1: A = 0000, B = 0001, Sel = 0000, Out = 0000000000000000, Ovf = 0
test_case No.    2: A = 0001, B = 0110, Sel = 0000, Out = 0000000000000100, Ovf = 0
test_case No.    3: A = 0111, B = 0001, Sel = 0001, Out = 0000000000000001, Ovf = 0
test_case No.    4: A = 0001, B = 0110, Sel = 0001, Out = 0000000000000000, Ovf = 0
test_case No.    5: A = 0111, B = 0001, Sel = 0010, Out = 0000000000000110, Ovf = 0
test_case No.    6: A = 0001, B = 0110, Sel = 0010, Out = 0000000000000111, Ovf = 0
test_case No.    7: A = 0111, B = 0011, Sel = 0011, Out = 0000000000010101, Ovf = 0
test_case No.    8: A = 0001, B = 0110, Sel = 0011, Out = 0000000000000110, Ovf = 0
test_case No.    9: A = 0111, B = 0000, Sel = 0100, Out = 0000000000000011, Ovf = 0
test_case No.   10: A = 0001, B = 0000, Sel = 0100, Out = 0000000000000000, Ovf = 0
test_case No.   11: A = 0111, B = 0000, Sel = 0101, Out = 0000000000001110, Ovf = 0
test_case No.   12: A = 0001, B = 0000, Sel = 0101, Out = 0000000000000010, Ovf = 0
test_case No.   13: A = 0111, B = 0000, Sel = 0110, Out = 0000000000000011, Ovf = 0
test_case No.   14: A = 0001, B = 0000, Sel = 0110, Out = 0000000000000000, Ovf = 0
test_case No.   15: A = 0111, B = 0000, Sel = 0111, Out = 0000000000001110, Ovf = 0
test_case No.   16: A = 0101, B = 0000, Sel = 0111, Out = 0000000000001010, Ovf = 0
test_case No.   17: A = 0000, B = 0000, Sel = 1000, Out = 0000000000000001, Ovf = 0
test_case No.   18: A = 0000, B = 1000, Sel = 1000, Out = 0000000000000010, Ovf = 0
test_case No.   19: A = 0000, B = 0000, Sel = 1001, Out = 0000000000000001, Ovf = 0
test_case No.   20: A = 0000, B = 1000, Sel = 1001, Out = 0000000000000100, Ovf = 0
test_case No.   21: A = 0100, B = 0011, Sel = 1010, Out = 0000000000000100, Ovf = 0
test_case No.   22: A = 0000, B = 1000, Sel = 1010, Out = 0000000000001000, Ovf = 0
test_case No.   23: A = 0100, B = 0011, Sel = 1011, Out = 0000000000000011, Ovf = 0
test_case No.   24: A = 0000, B = 1000, Sel = 1011, Out = 0000000000000000, Ovf = 0
test_case No.   25: A = 0100, B = 0011, Sel = 1100, Out = 0000000000000111, Ovf = 0
test_case No.   26: A = 0000, B = 1000, Sel = 1100, Out = 1111111111111000, Ovf = 0
test_case No.   27: A = 0100, B = 0011, Sel = 1101, Out = 0000000000000001, Ovf = 0
test_case No.   28: A = 0000, B = 1100, Sel = 1101, Out = 0000000000000100, Ovf = 0
test_case No.   29: A = 0100, B = 0011, Sel = 1110, Out = 0000000000000001, Ovf = 0
test_case No.   30: A = 0000, B = 0100, Sel = 1110, Out = 0000000000000100, Ovf = 0
test_case No.   31: A = 1111, B = 0100, Sel = 1110, Out = 0000000000000101, Ovf = 0
test_case No.   32: A = 1100, B = 0101, Sel = 1110, Out = 0000000000000111, Ovf = 1
All Correct.

Test bonus...

Bonus No.   33: A = 0111, B = 0011, Sel = 1111, Out = 0000000000010101, Ovf = 0
Bonus No.   34: A = 0001, B = 0110, Sel = 1111, Out = 0000000000000110, Ovf = 0

Get bonus.

Simulation complete via $finish(1) at time 680 NS + 0
./AM_tb.v:89           $finish;
```

4. Report Area



```
****************************************
Report : area
Design : AM
Version: K-2015.06-SP1
Date   : Sat Mar 30 15:15:39 2019
****************************************

Information: Updating design information... (UID-85)
Library(s) Used:

    slow (File: /theda21_2/CBDK_IC_Contest/cur/SynopsysDC/db/slow.db)

Number of ports:                          396
Number of nets:                           695
Number of cells:                          371
Number of combinational cells:            318
Number of sequential cells:                 6
Number of macros/black boxes:               0
Number of buf/inv:                         49
Number of references:                      29

Combinational area:               2824.473582
Buf/Inv area:                      166.345197
Noncombinational area:               0.000000
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                  2824.473582
Total area:                 undefined
1
```

5. Encountered Problems and Solving

- Cannot using **assign** in **always block**.
- Cannot instantiate **module** in **always block**.
- Reg can only using in **always block**.
- We can use big parentheses {} to do sign extension or merge.
  e.g. {a, b}, {12'b0, a&b}, {8'b0, {{4{a[3]}}, a}*{{4{b[3]}},b}}…
  Amazing!!
- To reduce the coding time, drawing the diagram is really helpful!

6. Questions
   - Why we can ignore the function name to built-in module but cannot ignore in my own module?
     e.g.
     ```
     and        (and0,a,b);         legal!
     my_and     (and0,a,b);         illegal!
     my_and   and_0(and0,a,b);      legal!
     ```
   - What is meaning of **combinational "area"** in report_area.txt?

7. Impression and Experience
   Before I start this lab, I think this is a really tough, cause I have to implement totally 16 function. However, most of them are easily. By reusing the module, can reduce workload significantly!! I think, however, the most difficult one is bonus one, since there are lots of gate have to be connected…
   Thank TAs for helping us and teaching us. Take care:-)