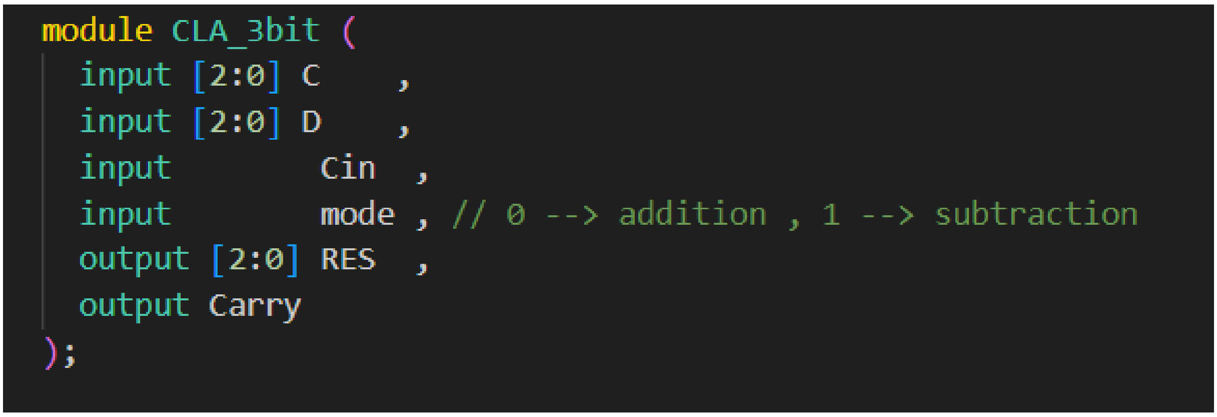
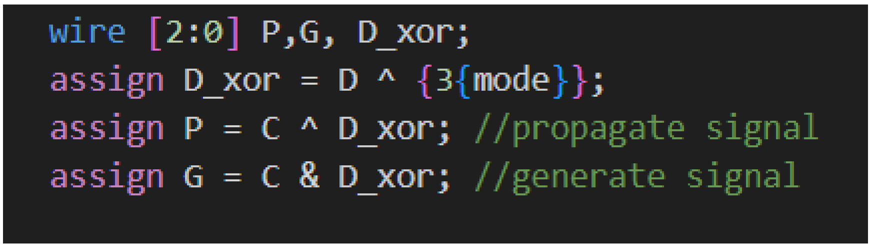
A 15-bit carry lookahead adder is designed using the 3-bit carry lookahead adder as a submodule. The 15-bit CLA can make addition and subtraction operations and gives the sum, detects the overflow and carry as result.

Implementation of CLA_3bit.v:



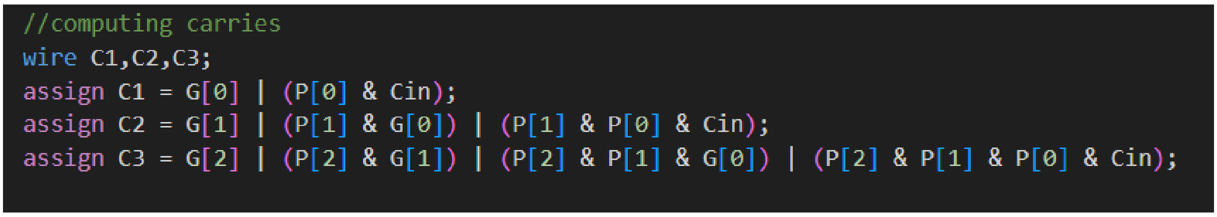
```
module CLA_3bit (
    input [2:0] C    ,
    input [2:0] D    ,
    input        Cin  ,
    input        mode , // 0 --> addition , 1 --> subtraction
    output [2:0] RES  ,
    output Carry
);
```

- **C** and **D** are two inputs, **Cin** the input for carry which comes from other 3-bit CLA, **mode** is the operation decider, **RES** is the sum and the **Carry** is the carry of the operation result.



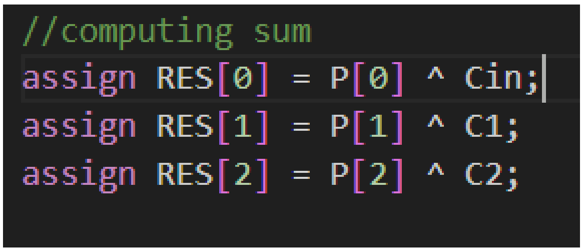
```
wire [2:0] P,G, D_xor;
assign D_xor = D ^ {3{mode}};
assign P = C ^ D_xor; //propagate signal
assign G = C & D_xor; //generate signal
```

- With extending the mode and making a xor operation with second input, the operation converted the summation by changing the sign of second input in case of a substraction. After that, the **P** and **G** are decided as propagation and generation signals.



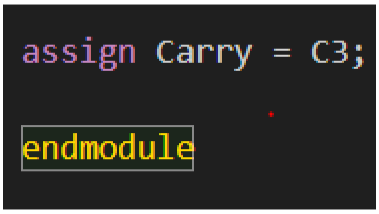
```
//computing carries
wire C1,C2,C3;
assign C1 = G[0] | (P[0] & Cin);
assign C2 = G[1] | (P[1] & G[0]) | (P[1] & P[0] & Cin);
assign C3 = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] & P[0] & Cin);
```

- Carries are decided using propagates, generates and Cin. C3 will be the final carry.



```
//computing sum
assign RES[0] = P[0] ^ Cin;
assign RES[1] = P[1] ^ C1;
assign RES[2] = P[2] ^ C2;
```

- Result's bits are computed using propagates and carries.

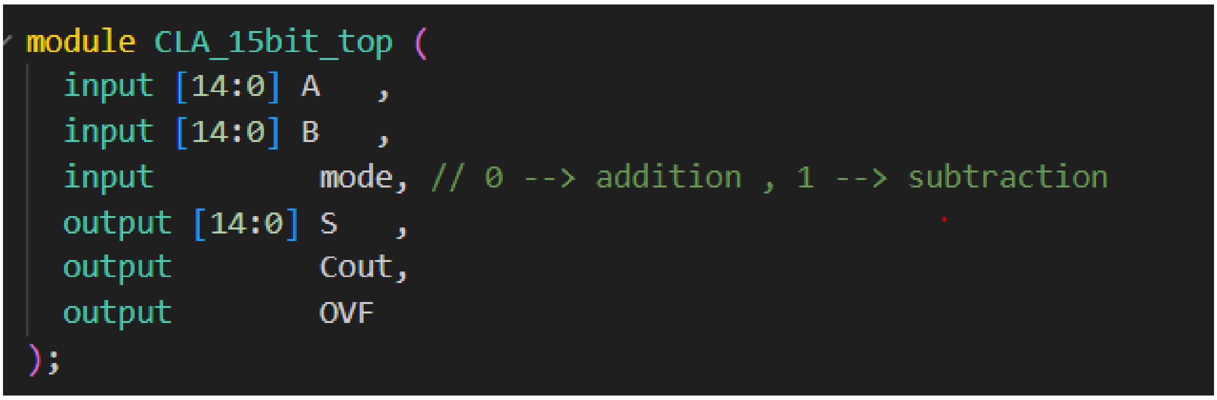


```
assign Carry = C3;

endmodule
```

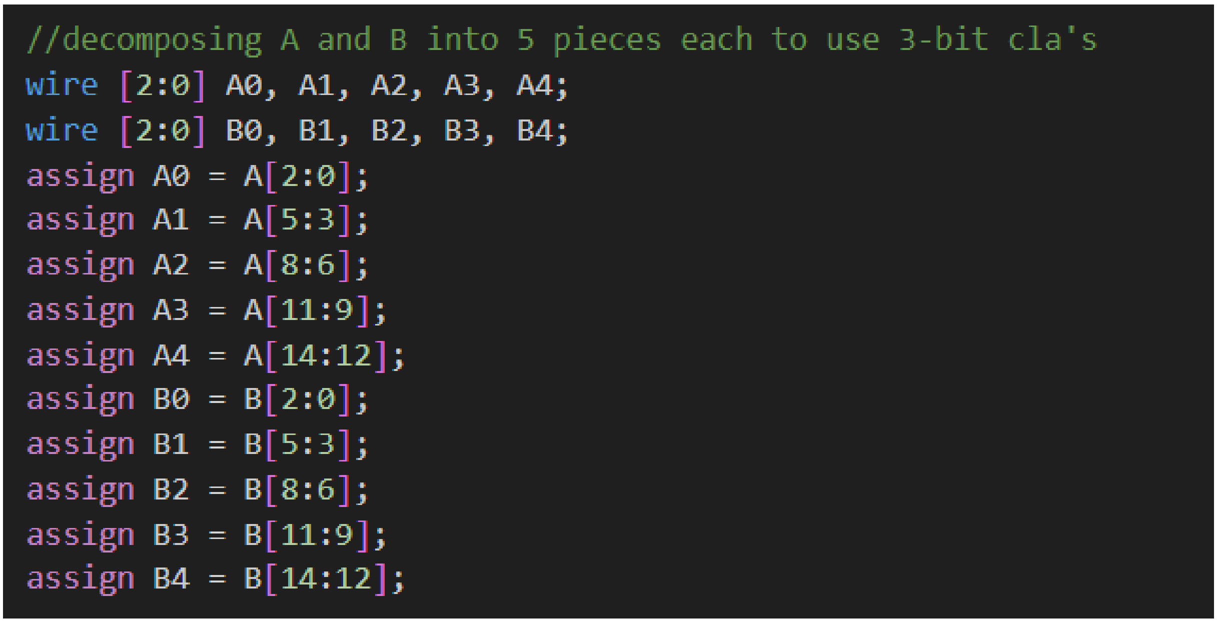
- The final carry is decided as C3 and the module ends.

Implementation of CLA_15bit_top.v:



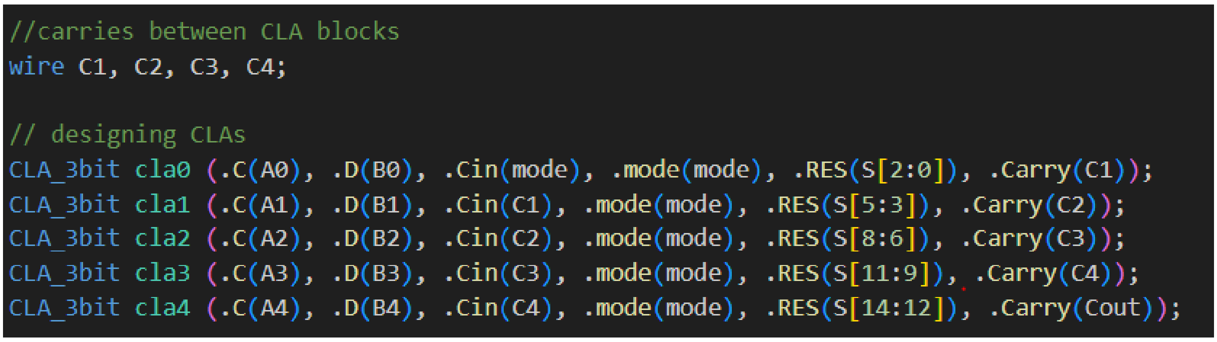
```
module CLA_15bit_top (
    input [14:0] A    ,
    input [14:0] B    ,
    input         mode, // 0 --> addition , 1 --> subtraction
    output [14:0] S    ,
    output        Cout,
    output        OVF
);
```

- **A** and **B** are the 15-bit inputs, **mode** is the operation decider, **S** is the resulting sum, **Cout** is carry and **OVF** is the overflow of the operation.



```
//decomposing A and B into 5 pieces each to use 3-bit cla's
wire [2:0] A0, A1, A2, A3, A4;
wire [2:0] B0, B1, B2, B3, B4;
assign A0 = A[2:0];
assign A1 = A[5:3];
assign A2 = A[8:6];
assign A3 = A[11:9];
assign A4 = A[14:12];
assign B0 = B[2:0];
assign B1 = B[5:3];
assign B2 = B[8:6];
assign B3 = B[11:9];
assign B4 = B[14:12];
```

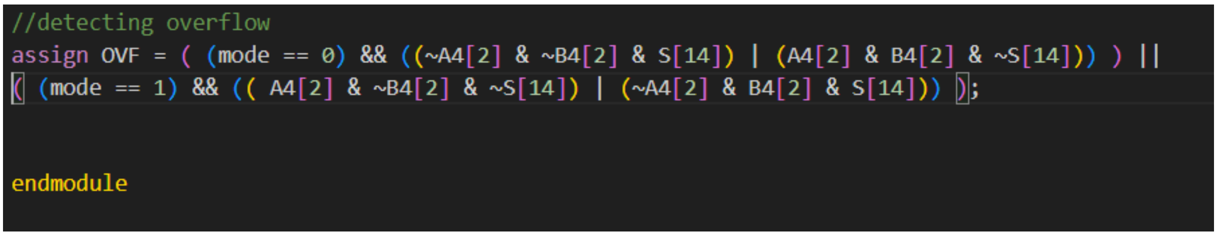
- Two inputs are parsed into groups consist of 3 bits to use 3-bit CLAs as submodules.



```
//carries between CLA blocks
wire C1, C2, C3, C4;

// designing CLAs
CLA_3bit cla0 (.C(A0), .D(B0), .Cin(mode), .mode(mode), .RES(S[2:0]), .Carry(C1));
CLA_3bit cla1 (.C(A1), .D(B1), .Cin(C1), .mode(mode), .RES(S[5:3]), .Carry(C2));
CLA_3bit cla2 (.C(A2), .D(B2), .Cin(C2), .mode(mode), .RES(S[8:6]), .Carry(C3));
CLA_3bit cla3 (.C(A3), .D(B3), .Cin(C3), .mode(mode), .RES(S[11:9]), .Carry(C4));
CLA_3bit cla4 (.C(A4), .D(B4), .Cin(C4), .mode(mode), .RES(S[14:12]), .Carry(Cout));
```

- Use of 3-bit CLAs with parsed inputs.



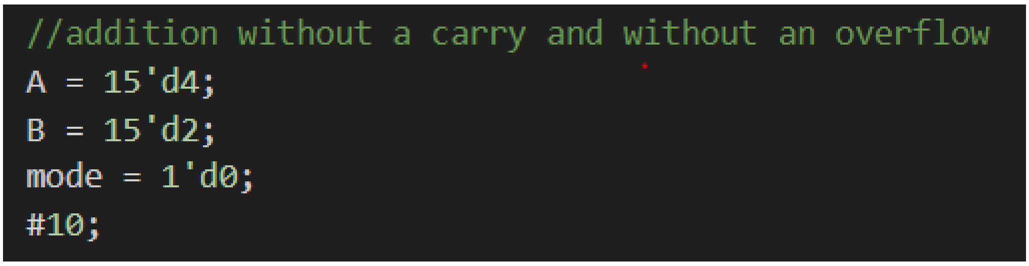
```
//detecting overflow
assign OVF = ( (mode == 0) && ((~A4[2] & ~B4[2] & S[14]) | (A4[2] & B4[2] & ~S[14])) ) ||
( (mode == 1) && (( A4[2] & ~B4[2] & ~S[14]) | (~A4[2] & B4[2] & S[14])) );


endmodule
```

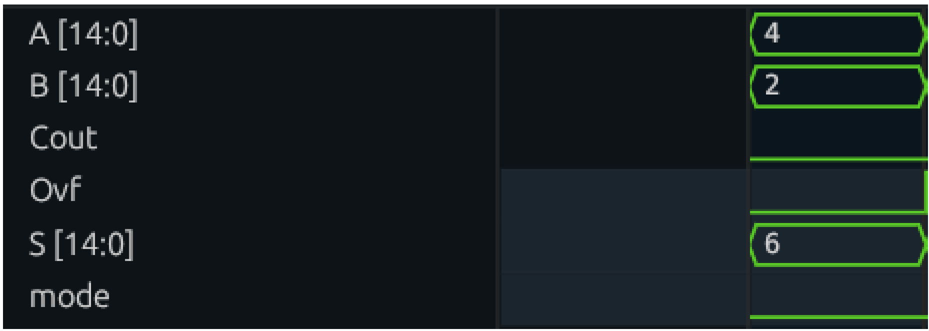
- Overflow detector is taking care of the addition and substraction operations’ overflows in different ways. After detecting the overflow, the module ends.

Test Cases Using CLA_15bit_tb.v:

1) Addition without a carry and without an overflow:

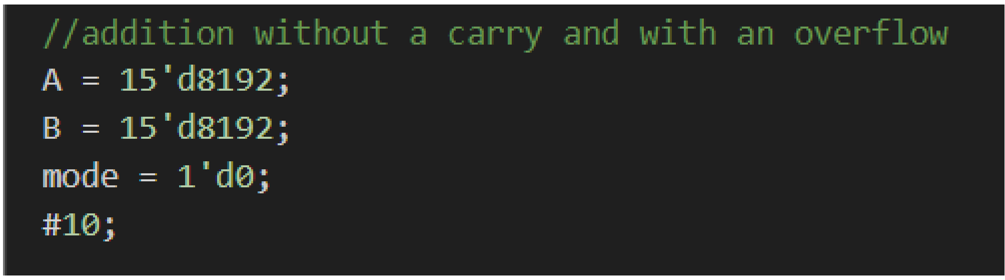


```
//addition without a carry and without an overflow
A = 15'd4;
B = 15'd2;
mode = 1'd0;
#10;
```

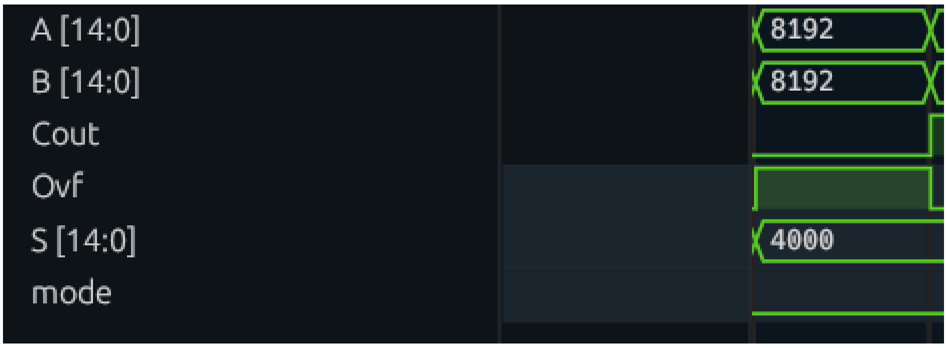


- Addition of 4 and 2 results in no carry and no overflow since the result is on the representable range.

2) Addition without a carry and with an overflow:

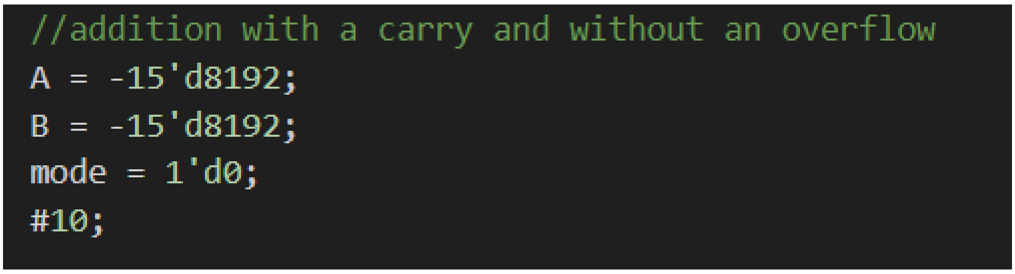


```
//addition without a carry and with an overflow
A = 15'd8192;
B = 15'd8192;
mode = 1'd0;
#10;
```

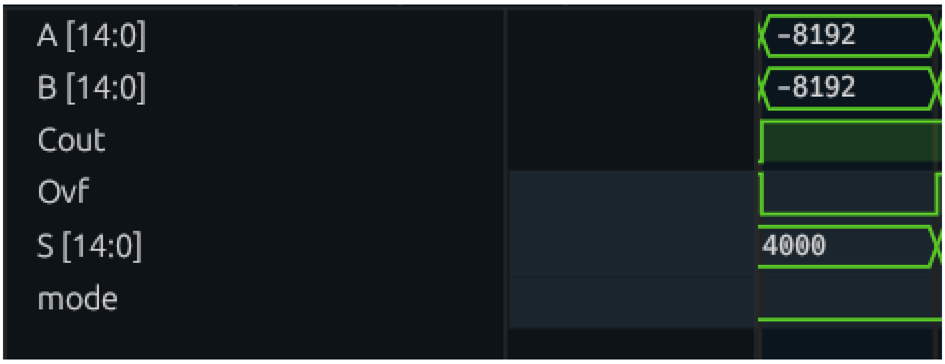


- This operation exceeds the representable range by 1, so there is an overflow but there is no carry.

3) Addition with a carry and without an overflow:

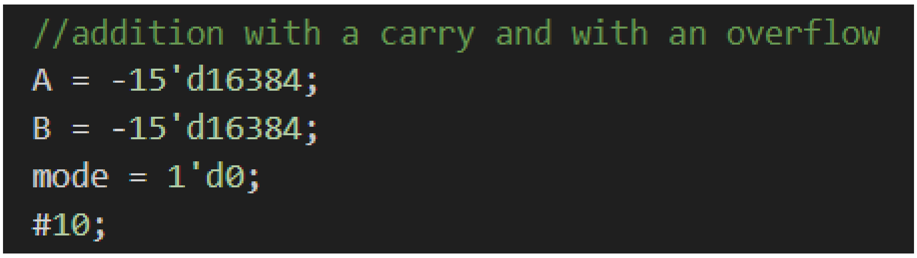


```
//addition with a carry and without an overflow
A = -15'd8192;
B = -15'd8192;
mode = 1'd0;
#10;
```

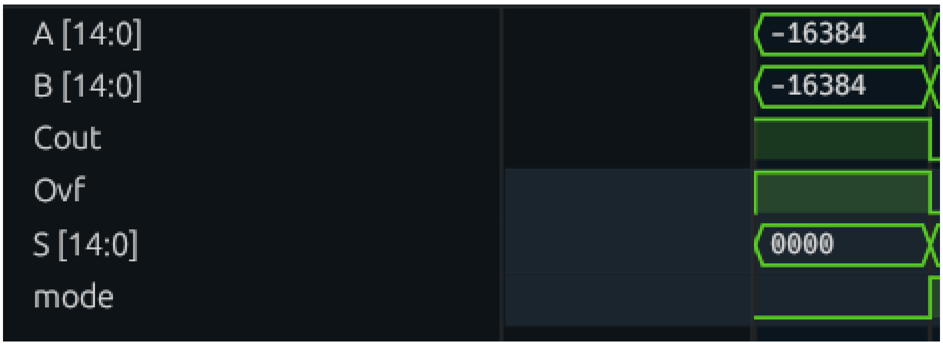


- This operation does not exceed the represantable range, indeed the result has the edge value, so there is no overflow. However, a carry is present as a result of the operation.

4) Addition with a carry and with an overflow:



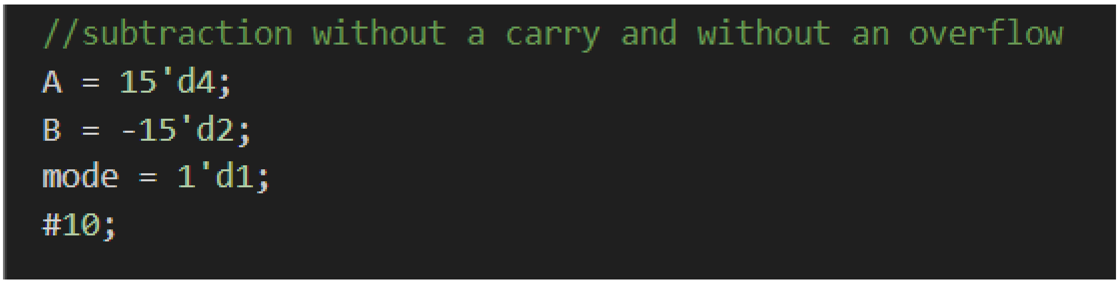
```
//addition with a carry and with an overflow
A = -15'd16384;
B = -15'd16384;
mode = 1'd0;
#10;
```



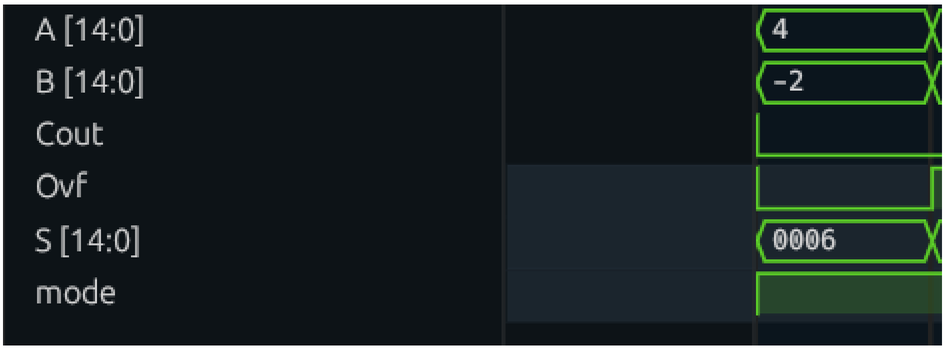
```
A [14:0]   -16384
B [14:0]   -16384
Cout
Ovf
S [14:0]   0000
mode
```

- This operation exceeds the represantable range and it generates a carry.

5) Subtraction without a carry and without an overflow:



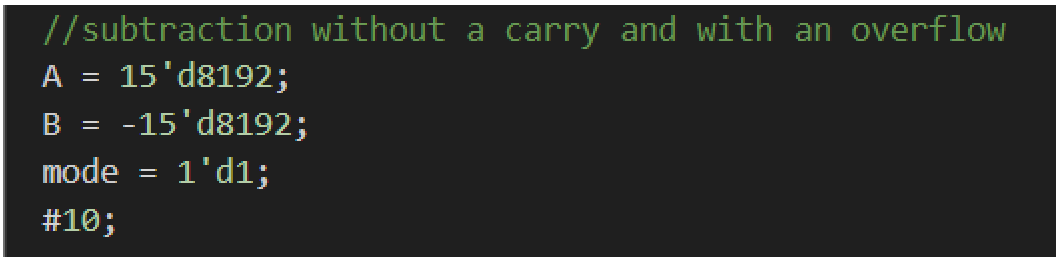
```
//subtraction without a carry and without an overflow
A = 15'd4;
B = -15'd2;
mode = 1'd1;
#10;
```



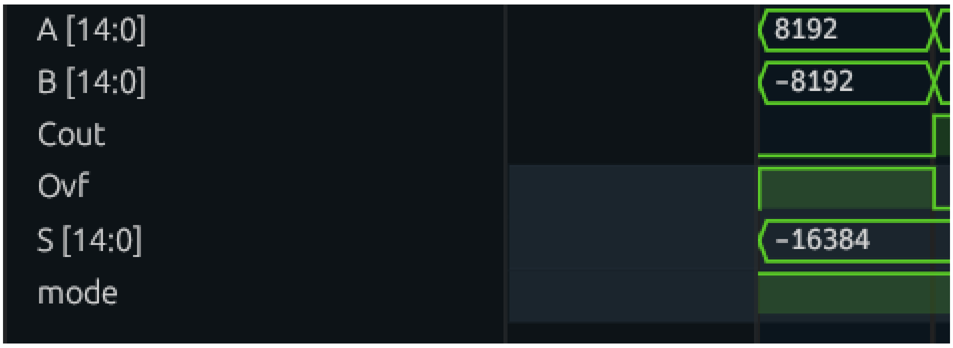
```
A [14:0]   4
B [14:0]   -2
Cout
Ovf
S [14:0]   0006
mode
```

- For the example cases of subtraction, the following logic that A+B and A-(-B) produces same results is used. 4-(-2) produces no carry and no overflow.

6) Subtraction without a carry and with an overflow:

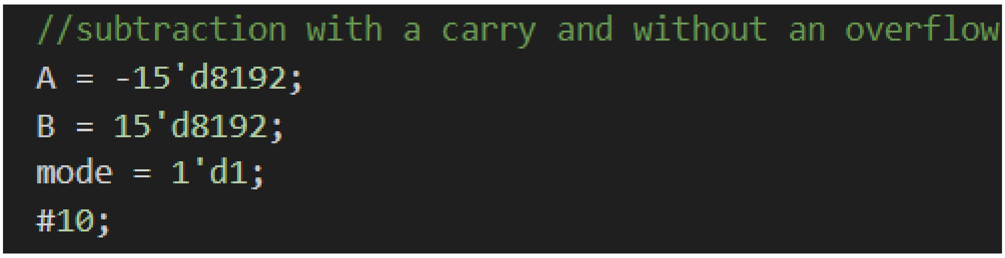


```
//subtraction without a carry and with an overflow
A = 15'd8192;
B = -15'd8192;
mode = 1'd1;
#10;
```

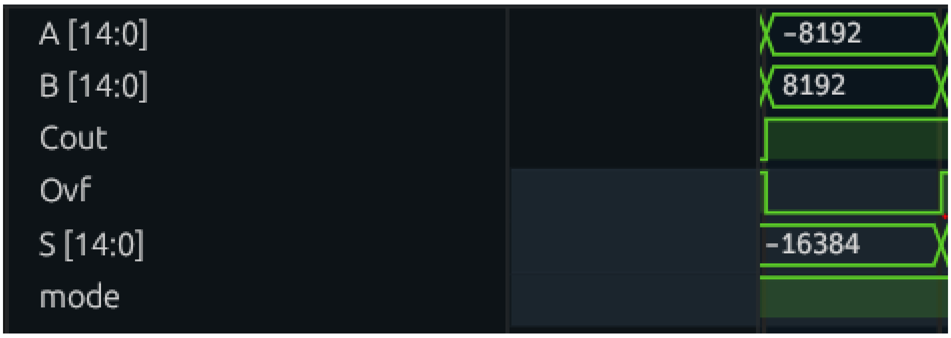


- This operation exceeds the representable range by 1, so there is an overflow. However, it does not produce a carry.

7) Subtraction with a carry and without an overflow:

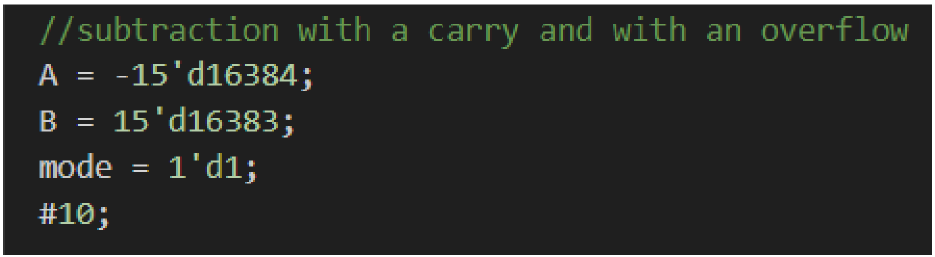


```
//subtraction with a carry and without an overflow
A = -15'd8192;
B = 15'd8192;
mode = 1'd1;
#10;
```

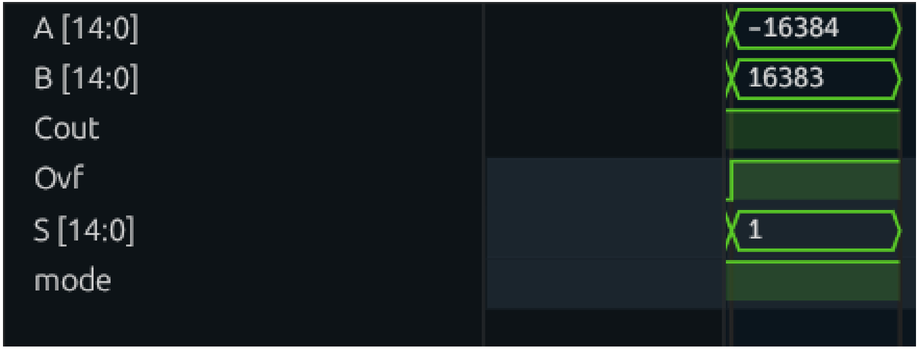


- The result of operation is just on the edge, so there isn't an overflow, but the operation produces a carry.

8) Subtraction with a carry and with an overflow:

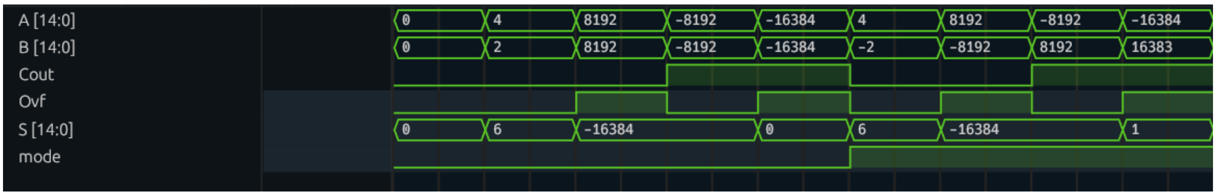


```
//subtraction with a carry and with an overflow
A = -15'd16384;
B = 15'd16383;
mode = 1'd1;
#10;
```



- The operation produces both carry and overflow.

The resulting simulation is below:


A [14:0]
0
4
8192
-8192
-16384
4
8192
-8192
-16384
B [14:0]
0
2
8192
-8192
-16384
-2
-8192
8192
16383
Cout
Ovf
S [14:0]
0
6
-16384
0
6
-16384
1
mode