ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 242E

DIGITAL CIRCUITS LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	3
EXPERIMENT DATE	:	02.04.2021
LAB SESSION	:	FRIDAY - 10.30
GROUP NO	:	G17

GROUP MEMBERS:

150180024 : ŞULE BEYZA KARADAĞ

150190710 : SENİHA SERRA BOZKURT

150190024 : AHMET FURKAN KAVRAZ

SPRING 2021

Contents

1 INTRODUCTION [10 points]

In the following experiment, we are going to implement a 16-bit Full adder-subtracter using the Verilog.

But we have steps until the 16-bit Full adder.

- First, we will implement our classical AND, OR, NOT, XOR gates.
- Then, in addition to them, we will follow the steps below:

1. We will implement a 1-bit Half-Adder.
2. By using it, we will implement a 1-bit Full-Adder.
3. By using it, we will implement a 4-bit Full-Adder.
4. By using it, we will implement a 16-bit Full-Adder.
5. By using it, we will implement a 16-bit Full-Adder-Subtracter.

2 MATERIALS AND METHODS [40 points]

2.1 Preliminary

2.1.1 Pre-Addition

• To recall signed and unsigned addition for binary numbers in 2's complement notation. Addition in 2's complement notation is done in two sub-parts:

1. Unsigned integers
2. Signed integers

To express these clearly, we are going to give some examples covering both the regular and the edge cases of these operations.

First, we do the unsigned integers part. Examples are:

1. a. To carry out A + B for regular case:

A = 0111 0101 : 117
B = + 0110 0011 : 99
Result = (A + B) = 1101 1000 : 216
carry bit ((n+1)th bit) = 0

In this example, we do not have a carry bit because we can represent our result "216" with using 8-bits.

The result of the addition is "256" and the 9^{th} bit is 1. "256" cannot be represented with using 8-bits. That's why we need a carry bit.

2. a. To carry out A + B for different values of A and B = edge case:

A = 1111 1111 : 255
B = + 0000 0001 : 1
Result = (A + B) = 1 0000 0000 : 256
carry bit ((n+1)th bit) = 1

CARRY explanation is that it can occur in addition of unsigned numbers. It shows that the result cannot be represented with n bits, and $(n+1)^{th}$ bit is necessary.

- Then, we do the signed integers part.

Representation of Negative Integers:

Negative integers are represented by 2's complement system. In this system, negative integers are represented by the two's complement of the positive number (absolute value). Getting the 2's complement:

• First invert (l's complement) the number. Change 0 to 1, 1 to 0.

• Then add 1 to the inverted number.

Example: -A = (1's complement of A) + 1 = A' + 1

2's complement system makes it easy to add or subtract two numbers without sign and magnitude checks.

2's complement operation changes the sign of a number. Applying 2's complement operation to a negative number makes it positive.

Normal cases' examples are: The operation is performed as with unsigned numbers, but the result is interpreted differently.

• Adding two's-complement numbers requires no special processing even if the operands have opposite signs.

• If (n+1)th bit arises by adding two n-bit signed numbers, this bit is ignored. Example: Addition of 8-bit signed numbers:

3. a. To carry out A + B for regular case:

A = 1111 1111 : -1
B = + 1111 1111 : -1
Result = (A + B) = 1 1111 1110 : -2
carry bit ((n+1)th bit) = 1 = Ignored
Sign bit = nth bit = 1 = Negative sign implicant.

As you can see next column, addition between two 8-bit -1 resulted in a 9-bit expression, so we have a carry bit, but it doesn't matter, because the n-th bit is always the sign bit.

Attention: While working with n-bit numbers the sign bit is always the most significant bit (counting from right to left) the nth bit, not the (n+1)th one (it is carry bit)

As you can see next column, addition between two 8-bit numbers resulted in a 9-bit expression, so we have a carry bit, but it doesn't matter, because the n-th bit is always the sign bit.

Overflow cases' examples are:

The operation has been done between positive integers. So the result should also be positive. So there has been an OVERFLOW, and the result cannot be represented with 8 bits.

4. a. To carry out A + B for edge case:

A = 1111 1111 : -1
B = + 0000 0001 : 1
Result = (A + B) = 1 0000 0000 : 0
carry bit ((n+1)th bit) = 1 = Ignored
Sign bit = nth bit = 0 = Positive sign implicant.

5. a. To carry out A + B for OVERFLOW case:

A = 0111 1111 : 127
B = + 0000 0010 : 2
Result = (A + B) = 1000 0001 : ???

Sign bit = nth bit = 1 = Negative sign implicant.
Result in decimal = 129

6. a. To carry out A + B for OVERFLOW case:

A = 1000 0000 : -128
B = + 1111 1111 : -1
Result = (A + B) = 1 0111 1111 : ???
carry bit ((n+1)th bit) = 1 = Ignored
Sign bit = nth bit = 0 = Positive sign implicant.

The operation has been done between negative integers. So the result should also be negative. So there has been an OVERFLOW, and the result cannot be represented with 8 bits.

Results: **OVERFLOW** explanation is that it can occur in addition operations only on signed numbers. It shows that the result cannot be represented with n bits. Overflow can be detected by checking and comparing the signs of operands and the result. There is an overflow in the following cases:

pos. + pos. → neg.

neg. + neg. → pos.

2.1.2 Pre-Subtraction

• Recall signed and unsigned subtraction for binary numbers in **2's complement** notation.

• Computers usually use the method of complements to implement subtraction.

• 2's complement of the second operand is added with the first number:

A - B = A + (-B)
= A + 2's complement (B)
= A + B' + 1

B = 0001 1001 : 25
1's complement of B = 1110 0110
One = + 1
Result = (- B) = 1110 0111 : -25

So, only one addition circuit is sufficient to perform both addition and subtraction. Like addition, also subtraction operations are performed on unsigned and signed numbers in the same way (because of 2's complement representation). But the interpretation of the result is different for unsigned and signed numbers.

If the result of the subtraction of two n-bit unsigned numbers, performed by 2's complement method, is a (n+1)-bit number, then there is not a borrow and the result is valid. If the (n+1) th bit of the result is zero, the first operand is smaller than the second and there is borrow. Example:

A =	0010 0000 : 32
2's complement of B =	+ 1110 0111 : -25
Result =	**1 0000 0111 : 7**

2.1.3 Pre-Terms

• Recall what carry, borrow and overflow mean, when they occur and how are they interpreted.

Addition of 2 n-bit numbers can give us an (n+1)-bit number. This bit is called carry. If we are dealing with signed numbers, this $(n+1)^{th}$ bit is ignored. It can only occur in the result of the sum of unsigned numbers. If the result of addition of unsigned numbers can not shown with n-bit, we use a carry.

When we do subtraction with unsigned numbers, borrow can occur. If the second operand is larger than the first one, there is no carry. That means there is a borrow, and the result of the subtraction cannot represented. If the result of the subtraction of 2 n-bit numbers is a (n+1)-bit number, there is a carry which means no borrow. Therefore the result is valid.

When adding or subtracting n-bit signed numbers, the result can be too large to represent with n-bits. There is a certain range that can be represented for each value of n. If the result of our operation is out of the range, an overflow occurs. After the operations, carry is ignored; then according the $(n)^{th}$ bit, we determine the sign of the result. Overflow depends on the signs of the result and operands. There are 4 cases where overflow can occur:

pos. + pos. → neg.

neg. + neg. → pos.

pos. - neg. → neg.

neg. - pos. → pos.

2.2 Experiment

2.2.1 Part 1

AND gates are created by assigning the result of applied "AND" operation between inputs to output.

```
module and_gate_3input(
    input wire A,
    input wire B,
    input wire C,
    output wire D
    );
    assign D = A & B & C;
endmodule
```

```
//and gate implemented that can AND 2 inputs
module and_gate_2input(
    input wire A,
    input wire B,
    output wire C
    );
    assign C = A & B;
endmodule
```

NOT gate is created by assigning result of applying "NOT" operation of input to output.

```
module not_gate(
    input wire A,
    output wire B
    );
    assign B = ~A;
endmodule
```

```
//or gate implemented that can OR 2 inputs
module or_gate_2input(
    input wire A,
    input wire B,
    output wire C
    );
    assign C = A | B;
endmodule
```

OR gate is created by assigning the result of applied "OR" operation between inputs to output.

```
module xor_gate_16_1input(
    input wire [15:0] A,
    input wire s,
    output wire [15:0] B
    );
    wire not_s;

    not_gate not1(s, not_s);

    assign B[0]  = (A[0] & not_s)  | (~A[0] & s);
    assign B[1]  = (A[1] & not_s)  | (~A[1] & s);
    assign B[2]  = (A[2] & not_s)  | (~A[2] & s);
    assign B[3]  = (A[3] & not_s)  | (~A[3] & s);
    assign B[4]  = (A[4] & not_s)  | (~A[4] & s);
    assign B[5]  = (A[5] & not_s)  | (~A[5] & s);
    assign B[6]  = (A[6] & not_s)  | (~A[6] & s);
    assign B[7]  = (A[7] & not_s)  | (~A[7] & s);
    assign B[8]  = (A[8] & not_s)  | (~A[8] & s);
    assign B[9]  = (A[9] & not_s)  | (~A[9] & s);
    assign B[10] = (A[10] & not_s) | (~A[10] & s);
    assign B[11] = (A[11] & not_s) | (~A[11] & s);
    assign B[12] = (A[12] & not_s) | (~A[12] & s);
    assign B[13] = (A[13] & not_s) | (~A[13] & s);
    assign B[14] = (A[14] & not_s) | (~A[14] & s);
    assign B[15] = (A[15] & not_s) | (~A[15] & s);
endmodule
```

```
//XOR gate implemented that can XOR 2 inputs
module xor_gate_2input(
    input wire A,
    input wire B,
    output wire C
    );
    wire not_A, not_B;

    not_gate not0(A, not_A);
    not_gate not1(B, not_B);

    assign C = (A & not_B) | (not_A & B);
endmodule
```

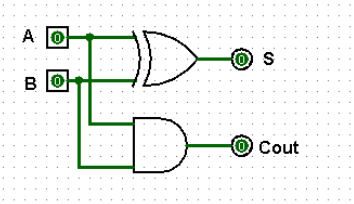
As you can see from the above and next column, the implementations of XOR gates are similar to each other, the working principle of an XOR gate is if it's inputs are different, it returns a TRUE value as 1, and if it's inputs are same, it returns a FALSE value as 0.

2.2.2 Part 2

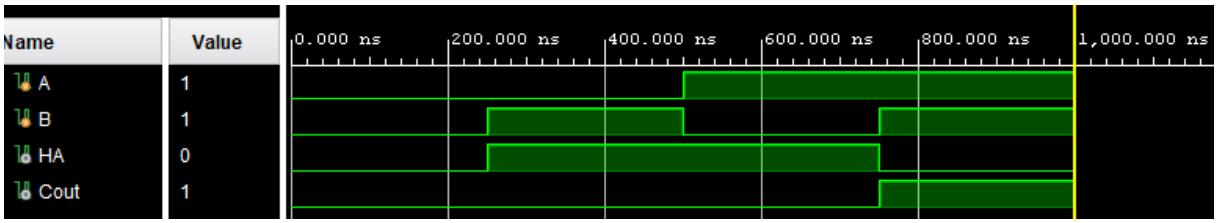
Half Adder adds two 1-bit numbers. Wires A, B, s, Cout, represent first number, second number, result and carry output respectively. As you can see in the truth table, our result column is same as the truth table of "XOR" operation. Therefore, we used "XOR" gate to connect wires A and B, and assigned it to the result s. The column of the carry out can be obtained by applying "AND" gate to A and B.

```
module half_adder_1bit(
    input wire A,
    input wire B,
    output wire s,
    output wire Cout
    );
    xor_gate_2input output1(A, B, s);
    and_gate_2input output2(A, B, Cout);
endmodule
```

	a	b	s	c
0	0	0	0	0
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1



Simulation results:

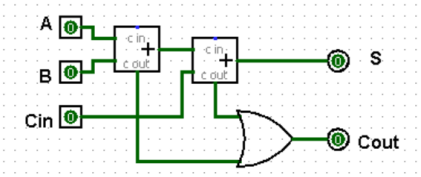


2.2.3 Part 3

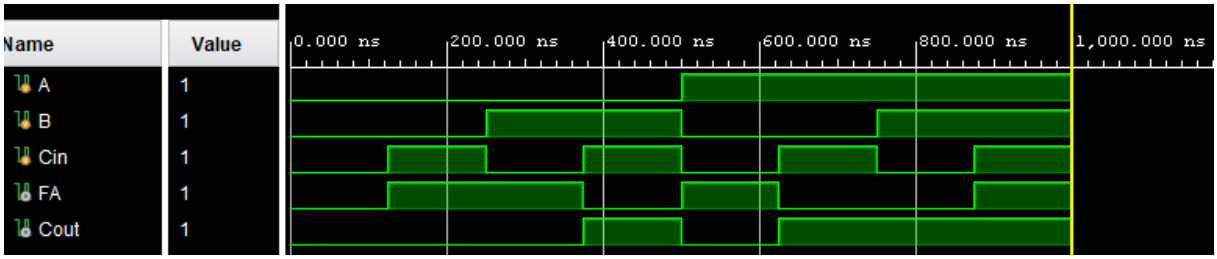
In this part, we implemented 1-Bit Full Adder by using half adder and OR modules which we designed in the previous parts. Wires A, B, Cin, s and Cout represent first number, second number, carry in, result and carry output respectively. As you can see in the circuit design, the result of the first half adder which is s1 and Cin are connected to another half adder. c1 and c2 ,which are the carry outputs of the half adders, are connected to an OR gate and the result of the operation is assigned to the Cout. Then, we simulated it for each different combination of inputs.

```
module full_adder_1bit(
    input wire A,
    input wire B,
    input wire Cin,
    output wire s,
    output wire Cout
    );

    wire c1, c2, s1;
    half_adder_1bit ha1(A, B, s1, c1);
    half_adder_1bit ha2(s1, Cin, s, c2);

    or_gate_2input or1(c1, c2, Cout);
endmodule
```

	a	b	c_{in}	c_{out}	s
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

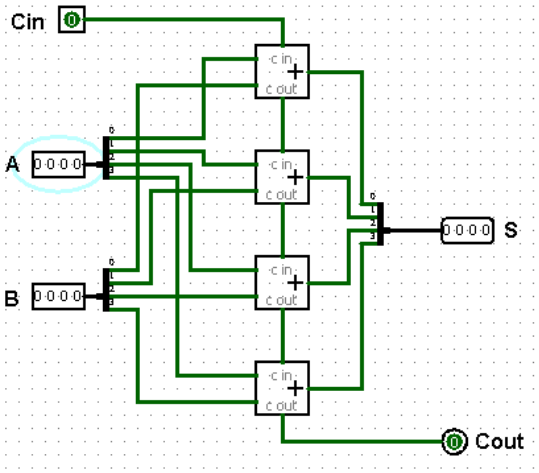


Simulation results:



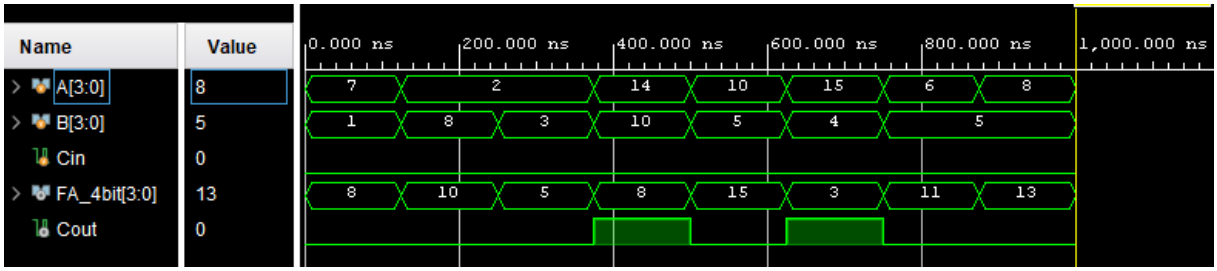
2.2.4 Part 4

In this part, we implemented a 4-Bit Full Adder by using 1-Bit Full Adder modules which we designed in the previous part. 4-Bit Full Adder adds two 4-bit binary numbers. Each bit of input A and B are connected to the 1-bit full adders one by one. Each *carry out* of the full adders are the *carry in* values of the next 1-bit full adder. Input wire Cin is used for the first 1-bit full adder. Then, we simulated it for '7+1', '2+8','2+3', '14+10', '10+5', '15+4', '6+5', and '8+5' operations.



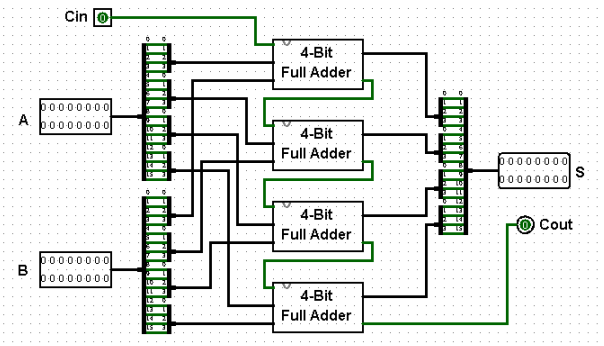

```
//4-Bit Full Adder implemented that can
//get 2 input values, 1 Carry Input, 1 Output and 1 Carry Output
// using with 1-Bit Full Adder
module full_adder_4bit(
    input wire [3:0] A,
    input wire [3:0] B,
    input wire Cin,
    output wire [3:0] s,
    output wire Cout
    );

    wire c1, c2, c3;

    full_adder_1bit fa0(A[0], B[0], Cin, s[0], c1);
    full_adder_1bit fa1(A[1], B[1], c1 , s[1], c2);
    full_adder_1bit fa2(A[2], B[2], c2 , s[2],  c3);
    full_adder_1bit fa3(A[3], B[3], c3 , s[3],  Cout);
endmodule
```

Simulation results:



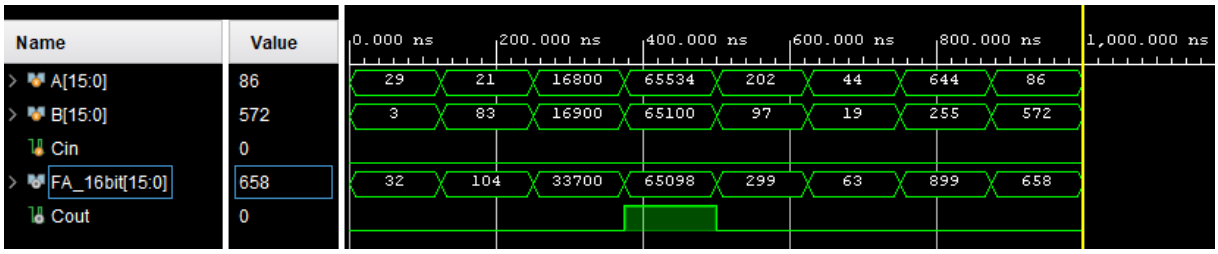
2.2.5 Part 5

In this part, we implemented a 16-Bit Full Adder by using 4-Bit Full Adder module which we designed in the previous part. Each 4-bit part of input A and B are connected to the 4-bit full adders in order. Each *carry out* of the full adders are the *carry in* values of the next 4-bit full adder. Input wire Cin is used for the first 4-bit full adder. Then, we simulated it for '29+3', '21+83', '16800+16900', '65534+65100', '202+97', '44+19', '644+255', and '86+572' operations.




```
//16-Bit Full Adder implemented that can get
//2 input values, 1 Carry Input, 1 Output and 1 Carry Output
// using with 4-Bit Full Adder
module full_adder_16bit(
    input wire [15:0] A,
    input wire [15:0] B,
    input wire Cin,
    output wire [15:0] s,
    output wire Cout
    );

    wire c1, c2, c3;

    full_adder_4bit fa0(A[3:0]  , B[3:0]  , Cin, s[3:0],    c1  );
    full_adder_4bit fa1(A[7:4]  , B[7:4]  , c1 , s[7:4],    c2  );
    full_adder_4bit fa2(A[11:8] , B[11:8] , c2 , s[11:8],   c3  );
    full_adder_4bit fa3(A[15:12], B[15:12], c3 , s[15:12],  Cout);
endmodule
```

Simulation results:



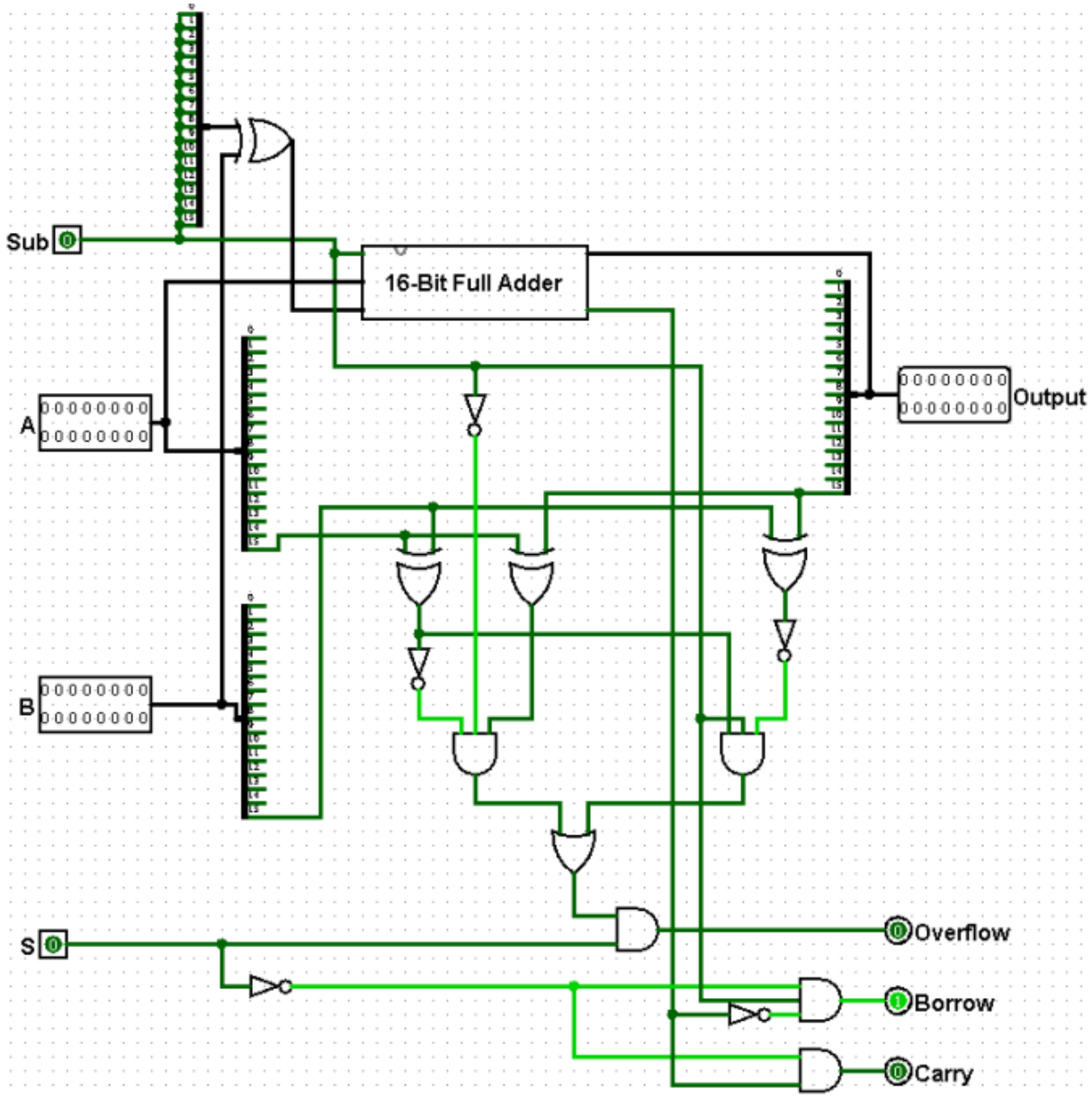
2.2.6 Part 6

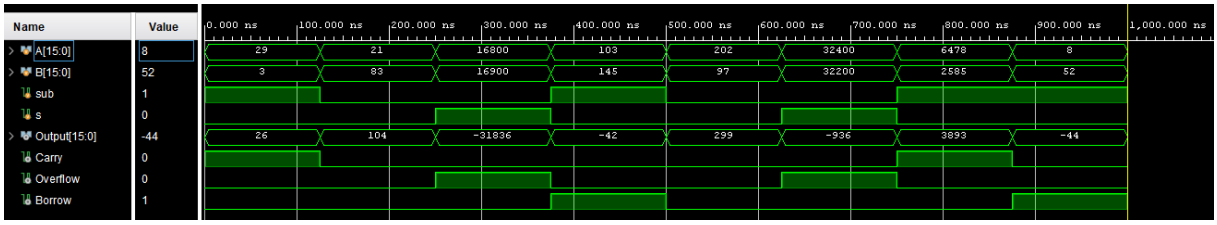
```
module full_adder_subtractor_16bit(
    input wire [15:0] A,
    input wire [15:0] B,
    input wire sub,
    input wire s,
    output wire [15:0] Output,
    output wire Carry,
    output wire Overflow,
    output wire Borrow
    );

    wire [15:0] c;
    wire cout;

    xor_gate_16_1input xor0(B[15:0]  , sub, c[15:0]);

    full_adder_16bit FA(A, c, sub, Output, Cout);

    //Overflow  checking
    wire sub_not;
    wire xor1, xor1_not, xor2, xor3, xor3_not;
    wire and1, and2, and3;
    wire or1;

    xor_gate_2input xorf1(A[15], B[15]     , xor1);
    xor_gate_2input xorf2(A[15], Output[15], xor2);
    xor_gate_2input xorf3(B[15], Output[15], xor3);

    not_gate not1(xor1, xor1_not);
    not_gate not2(xor3, xor3_not);
    not_gate not3(sub, sub_not);

    and_gate_3input andf1(xor1_not, sub_not, xor2, and1);
    and_gate_3input andf2(xor1    , sub    , xor3_not, and2);

    or_gate_2input  orf1 (and1, and2, or1);
    and_gate_2input andf3(s, or1, Overflow);

    //Borrow  checking
    wire s_not, Cout_not;

    not_gate not4(s, s_not);
    not_gate not5(Cout, Cout_not);

    and_gate_3input andf4(s_not, sub, Cout_not, Borrow);

    //Carry Checking
    and_gate_2input andf5(s_not, Cout, Carry);
endmodule
```

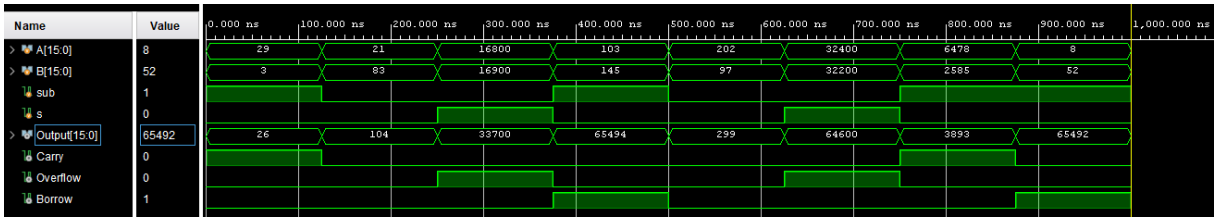
In this part, we implemented a 16-Bit Full Adder-Subtractor by using 16-Bit Full Adder, NOT, XOR, AND, and OR modules which we designed in the previous parts.

We received an additional input 'S' to indicate the numbers are signed or unsigned numbers. If S equals to 0 then the number is unsigned, if S equals to 1 then the number is signed. Also, we received an additional input 'sub' to indicate the operations are addition or subtraction. If sub equals to 0 then the operation is addition, if sub equals to 1 then the operation is subtraction. After that, we simulated it for '29-3, S=0', '21+83, S=0', '16800+16900, S=1', '103-145, S=0', '202+97, S=0', '32400+32200, S=1', '6478-2585, S=0', and '8-52, S=0' operations.

If there is an overflow or borrow occurs on operation above, then we showed these results with flags. Then, we showed that the operation above is whether valid or not.

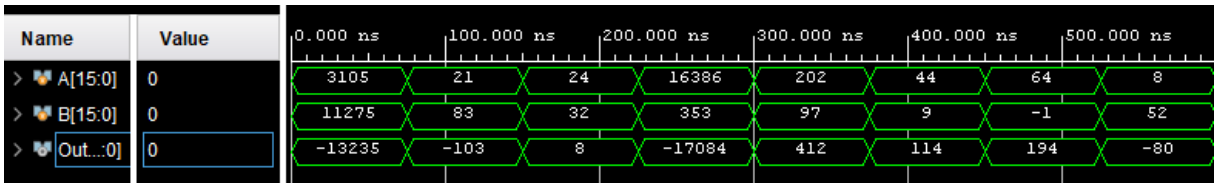

Sub
16-Bit Full Adder
A
Output
B
S
Overflow
Borrow
Carry

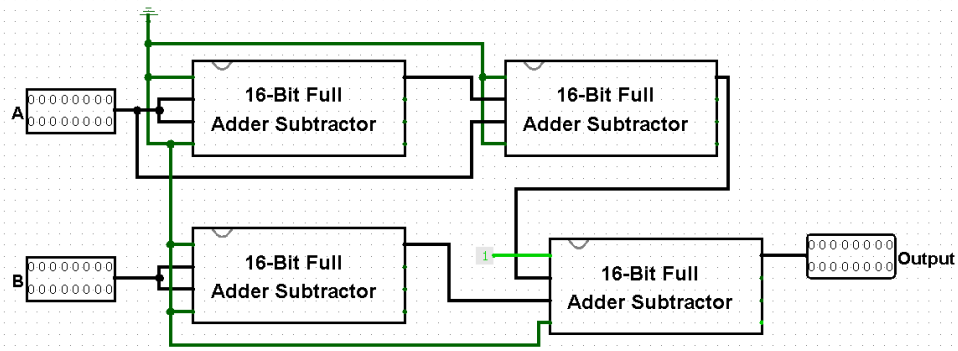

Name
Value
A[15:0]
8
B[15:0]
52
sub
1
s
0
Output[15:0]
-44
Carry
0
Overflow
0
Borrow
1


Name
Value
A[15:0]
8
B[15:0]
52
sub
1
s
0
Output[15:0]
65492
Carry
0
Overflow
0
Borrow
1

2.2.7 Part 7

In this part, we implemented a module which calculates the $3A - 2B$ by using 16- Bit Adder-Subtractor. Then, we simulated it for 'A=3105, B=11275', 'A=21, B= 83', 'A=24, B=32', 'A=16386, B=353', 'A=202, B=97', 'A=44, B=9', 'A=64, B=65535', and 'A=8, B=52' inputs.





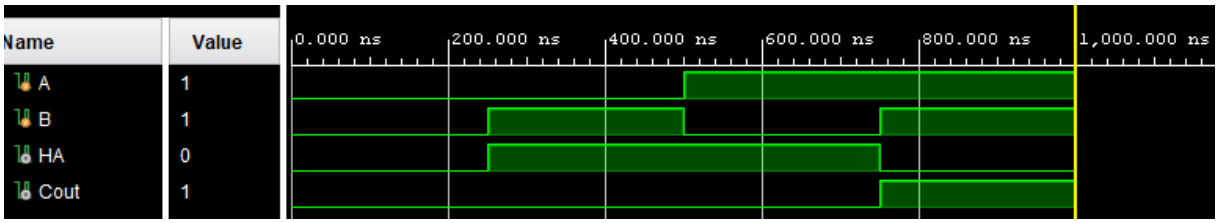

```
module threeA_minus_twoB(
    input wire [15:0] A,
    input wire [15:0] B,
    output wire [15:0] Output
    );
    wire [15:0] twoA;
    wire [15:0] threeA;
    wire [15:0] twoB;

    wire Carry, Overflow, Borrow;
    wire firstCout, secondCout, thirdCout ;

    full_adder_16bit fa1(A   , A,  0, twoA  , firstCout);
    full_adder_16bit fa2(twoA, A,  0, threeA, secondCout);
    full_adder_16bit fa3(B   , B,  0, twoB  , thirdCout);

    full_adder_subtractor_16bit fad_sub0
            (A, A, 0, 0, twoA, Carry, Overflow, Borrow);

    full_adder_subtractor_16bit fad_sub1
            (A, twoA, 0, 0, threeA, Carry, Overflow, Borrow);

    full_adder_subtractor_16bit fad_sub2
            (B, B, 0, 0, twoB, Carry, Overflow, Borrow);

    full_adder_subtractor_16bit fad_sub3
            (threeA, twoB, 1, 0, Output, Carry, Overflow, Borrow);
endmodule
```

3 RESULTS [15 points]

You can see the simulation inputs and their given times to complete each step of their part of the simulations below. To carry out each part of the experiment, we arranged inputs and the time slots of each simulation according to part which it belongs. The simulation results we obtained after completing each step of the experiments resulted as we expected. This concludes the fact that our experiment ended up being coherent.

```
//  Half Adder Simulation
reg A_1bit, B_1bit;
wire Half_Adder, Cout_HA;
half_adder_1bit  uut0(A_1bit , B_1bit , Half_Adder, Cout_HA);

initial begin
    A_1bit = 0;   B_1bit = 0; #250;
    A_1bit = 0;   B_1bit = 1; #250;
    A_1bit = 1;   B_1bit = 0; #250;
    A_1bit = 1;   B_1bit = 1; #250;
end
```

	a	b	s	c
0	0	0	0	0
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1

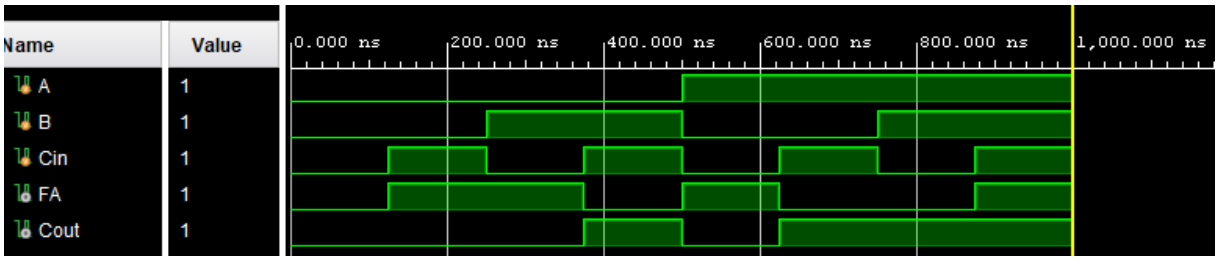


As you may see above, the truth table and simulation results give us the same values. Here, we implemented a 1 bit half adder, and in the code block above from *simulation.v* file, all the input combinations ($2^2 = 4$) exist.

```
//  1-Bit Full Adder Simulation
reg A_1bit, B_1bit, Cin;
wire Full_Adder, Cout_FA_1Bit;
full_adder_1bit  uut(A_1bit, B_1bit, Cin, Full_Adder, Cout_FA_1Bit);

initial begin
    A_1bit = 0;   B_1bit = 0;  Cin = 0;  #125;
    A_1bit = 0;   B_1bit = 0;  Cin = 1;  #125;
    A_1bit = 0;   B_1bit = 1;  Cin = 0;  #125;
    A_1bit = 0;   B_1bit = 1;  Cin = 1;  #125;
    A_1bit = 1;   B_1bit = 0;  Cin = 0;  #125;
    A_1bit = 1;   B_1bit = 0;  Cin = 1;  #125;
    A_1bit = 1;   B_1bit = 1;  Cin = 0;  #125;
    A_1bit = 1;   B_1bit = 1;  Cin = 1;  #125;
end
```

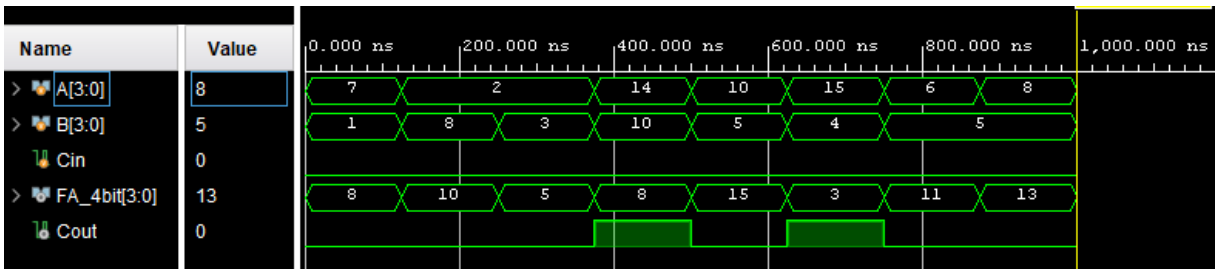
	a	b	c_{in}	c_{out}	s
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

As you can see from the truth table above and simulation results below, they give us the same values. Here, we implemented a 1 bit full adder, and in the code block above from *simulation.v* file, all the input combinations ($2^3 = 8$) exist.



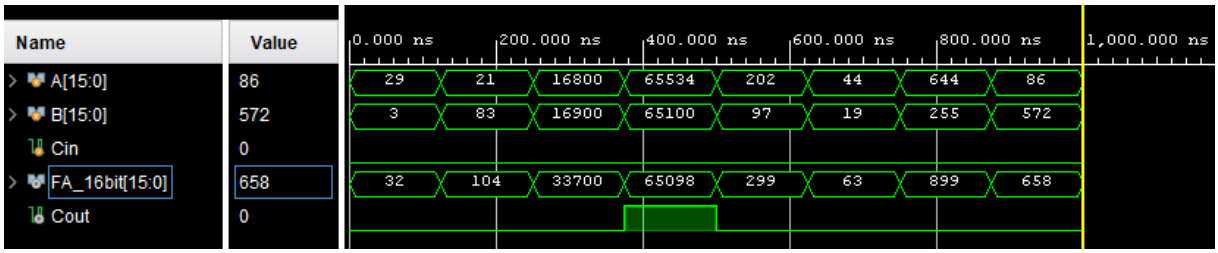
As you may see, the values are decimal numbers given to is from pdf. Here, we implemented a 4-bit full adder, and in the code block next column is from *simulation.v* file.

```
//  4-Bit Full Adder Simulation
reg [3:0] A_4bit;
reg [3:0] B_4bit;
wire [3:0] Full_Adder_4bit;
wire Cout_FA_4Bit;
full_adder_4bit  uut(A_4bit , B_4bit , 0, Full_Adder_4bit, Cout_FA_4Bit);
initial begin
    A_4bit = 4'd7;    B_4bit = 4'd1;    #125;
    A_4bit = 4'd2;    B_4bit = 4'd8;    #125;
    A_4bit = 4'd2;    B_4bit = 4'd3;    #125;
    A_4bit = 4'd14;   B_4bit = 4'd10;   #125;
    A_4bit = 4'd10;   B_4bit = 4'd5;    #125;
    A_4bit = 4'd15;   B_4bit = 4'd4;    #125;
    A_4bit = 4'd6;    B_4bit = 4'd5;    #125;
    A_4bit = 4'd8;    B_4bit = 4'd5;    #125;
end
```



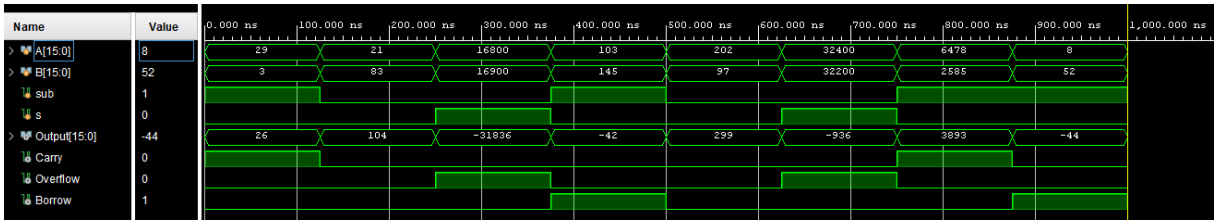
As you may see, the values are decimal numbers given to is from pdf. Here, we implemented a 16-bit full adder, and in the code block next column is from *simulation.v* file.

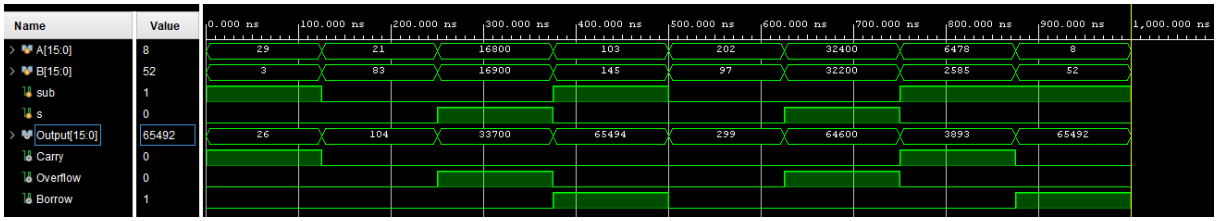
```
//  16-Bit Full Adder Simulation
reg [15:0] A_16bit;
reg [15:0] B_16bit;
wire [15:0] Full_Adder_16bit;
wire Cout_FA_16Bit;
full_adder_16bit uut(A_16bit, B_16bit, 0, Full_Adder_16bit, Cout_FA_16Bit);

initial begin
    A_16bit = 16'd29;     B_16bit = 16'd3;          #125;
    A_16bit = 16'd21;     B_16bit = 16'd83;         #125;
    A_16bit = 16'd16800;  B_16bit = 16'd16900;      #125;
    A_16bit = 16'd65534;  B_16bit = 16'd65100;      #125;
    A_16bit = 16'd202;    B_16bit = 16'd97;         #125;
    A_16bit = 16'd44;     B_16bit = 16'd19;         #125;
    A_16bit = 16'd644;    B_16bit = 16'd255;        #125;
    A_16bit = 16'd86;     B_16bit = 16'd572;        #125;
end
```



As you may see, the values are decimal numbers given to is from pdf. Here, we implemented a 16-bit full adder-subtracter, and in the code block next column is from *simulation.v* file.

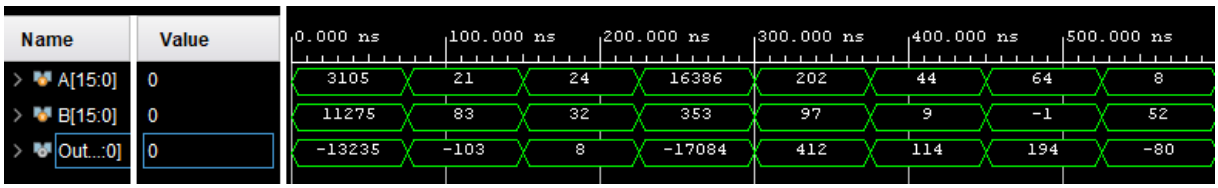
```
//  16-Bit Full Adder-Subtractor Simulation
reg [15:0] A_16bit;
reg [15:0] B_16bit;
reg sub;
reg s;
wire [15:0] Full_Adder_16bit;
wire Carry, Overflow, Borrow;

full_adder_subtractor_16bit fad_sub1(A_16bit, B_16bit, sub, s, Full_Adder_16bit,
                                     Carry, Overflow, Borrow);

 initial begin
    A_16bit = 16'd29;     B_16bit = 16'd3;      sub = 1;  s = 0; #125;
    A_16bit = 16'd21;     B_16bit = 16'd83;     sub = 1;  s = 0; #125;
    A_16bit = 16'd16800;  B_16bit = 16'd16900;  sub = 1;  s = 1; #125;
    A_16bit = 16'd103;    B_16bit = 16'd145;    sub = 1;  s = 0; #125;
    A_16bit = 16'd202;    B_16bit = 16'd97;     sub = 0;  s = 0; #125;
    A_16bit = 16'd32400;  B_16bit = 16'd32200;  sub = 0;  s = 1; #125;
    A_16bit = 16'd6478;   B_16bit = 16'd2585;   sub = 1;  s = 0; #125;
    A_16bit = 16'd8;      B_16bit = 16'd52;     sub = 1;  s = 0; #125;
end
```






```
//  3A - 2B
reg [15:0] A;
reg [15:0] B;
wire [15:0] threeA_Minus_twoB;

threeA_minus_twoB uut(A, B, threeA_Minus_twoB);
initial begin
    A = 16'd3105;    B = 16'd11275;    #125;
    A = 16'd21;      B = 16'd83;       #125;
    A = 16'd24;      B = 16'd32;       #125;
    A = 16'd16386;   B = 16'd353;      #125;
    A = 16'd202;     B = 16'd97;       #125;
    A = 16'd44;      B = 16'd9;        #125;
    A = 16'd64;      B = 16'd65535;    #125;
    A = 16'd8;       B = 16'd52;       #125;
    A = 16'd0;       B = 16'd0;        #125;
  end
```

As you may see, the values are decimal numbers given to us from pdf. Here, we did the $3 * A - 2 * B$ operation and in the code block next column is from *simulation.v* file. You can see that some of the values are not valid, that is because of the result cannot be represented using just 16-bits.



4 DISCUSSION [25 points]

In Part-1, we implemented AND, OR, NOT, XOR modules which we used in the following parts of the experiment. Then instead of using those operations, we used these modules we implemented in the following parts.
In Part-2, we implemented 1-Bit Half Adder module by using AND, OR, NOT, XOR modules which we designed in the first part. Then, we simulated it for each different combination of input.

For the Experiment part (Binary Arithmetic):
In Part-3, we implement 1-Bit Full Adder by using half adder and OR modules which you designed in the previous parts. Then, we simulated it for each different input combinations.
In Part-4, we implemented a 4-Bit Full Adder by using 1-Bit Full Adder modules which we designed in the previous part. Then, we simulated it for given inputs.
In Part-5, we implement a 16-Bit Full Adder by using 4-Bit Full Adder module which we designed in the previous part. Then, we simulated it for given inputs.
In Part-6, we implement a 16-Bit Full Adder-Subtractor by using 16-Bit Full Adder, NOT, XOR, AND, and OR modules which we designed in the previous parts. We received an additional input 'S' to indicate the numbers are signed or unsigned numbers.

```
If S == 0:
    the_number is Unsigned
If S == 1:
    the_number is Signed
```

After that, we simulated it for given inputs. If there is an overflow or borrow occurs on operation above, then we showed these results with flags. Then, we showed that the operation above is whether valid or not.
In Part-7, we implemented a module which calculates the 3A - 2B by using 16- Bit Adder-Subtractor, Adder, NOT, XOR, AND, and OR modules. Then, we simulated it for given inputs.

While implementing the circuits, we always tried to consider all the edge cases, like calculating $3.A$ when $A = 65535$ and $B = 0$. So, $3 * A = 196605$, so $log_2(196605) = 17.5849404868$ So we need 17+ bits (18 bits) to represent (We also saw that the binary representation of $196605 = 101111111111111101$ so it requires 18 bits to be represented).

5 CONCLUSION [10 points]

We learned how to implement big adders using more smaller adders. And we also learned how to do the calculations given to us by using our implementations.

We learned how to respect the calculators in our computers because the process on the back-end was so hard for us to implement, even for just 16 bit numbers.

We saw how to implement basic calculations, just the adding and subtracting. We didn't know how to implement multiplication and division, so when calculating $3 * A$ and $-2 * B$, we just connected them to the adder 3 times and 2 times. We don't know what to do if a complex operation like $100 * A - 789 * B$ comes in front of us, so we hope to learn them soon.

We think the explanations were not clear enough, we thought we should calculate all the expressions correctly (especially in Part-7) so we tried to consider all the edge cases, carries and borrows. And we didn't know that we are not supposed to do them all in the correct way, so we tried so hard and then we deleted all the efforts just to consider what's necessary and what's not. We also didn't learn this fact from any announcement, just one of our friends (from a different group and session) mailed the assistant and then informed all those who're taking this class (via WhatsApp) about how to handle edge cases. So we at least expect clearer explanations or announcements for a wider spread of this kind of necessary information.

Best regards.