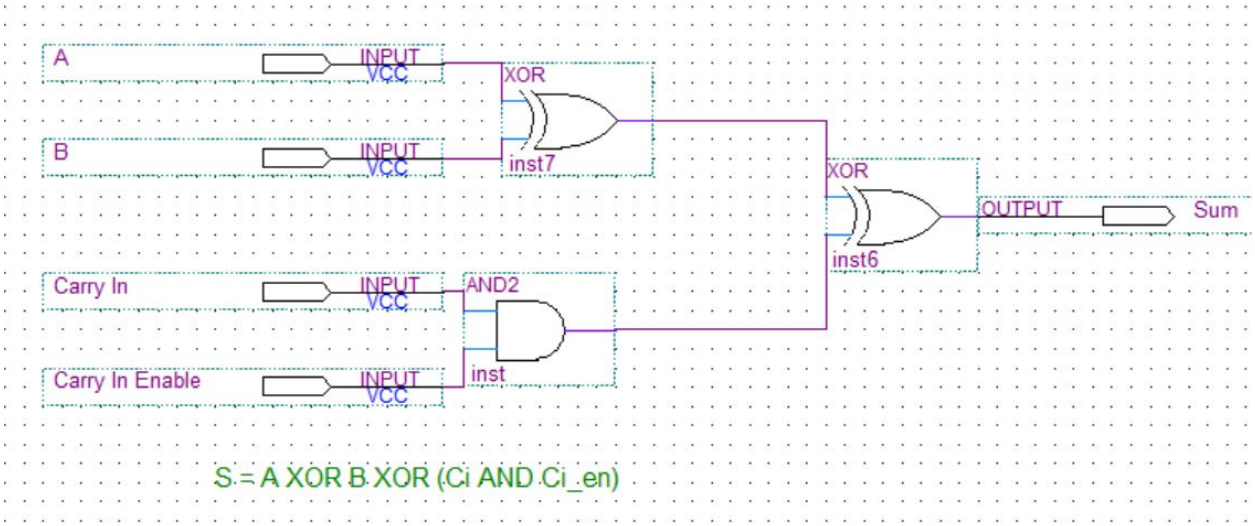
James Condie
EE421

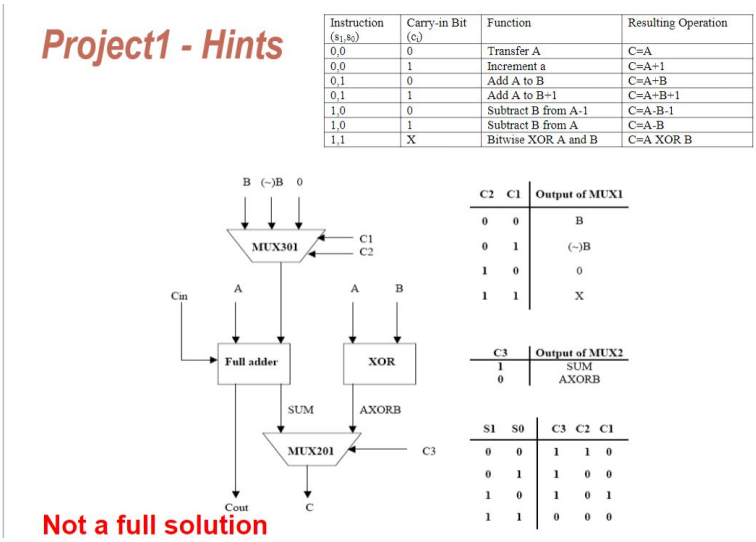
Project 1 report

I attempted to improve upon the project hints. Firstly I add an XOR function to the full adder module. My motivation is that the sum of the adder uses XOR gates. These XOR gates can be used for XOR operation if a carry in enable is added.

Sum bit for Adder with XOR



We see when Carry in is disabled, the sum output gives A XOR B.The Carry in enable bit becomes one control bit. This eliminates the need for an XOR module and the MUX201 module. Project one is shown so I can refer to improvements on this model.



Instruction (s_1,s_0)	Carry-in Bit (c_i)	Function	Resulting Operation
0,0	0	Transfer A	C=A
0,0	1	Increment a	C=A+1
0,1	0	Add A to B	C=A+B
0,1	1	Add A to B+1	C=A+B+1
1,0	0	Subtract B from A-1	C=A-B-1
1,0	1	Subtract B from A	C=A-B
1,1	X	Bitwise XOR A and B	C=A XOR B

C2	C1	Output of MUX1
0	0	B
0	1	(–)B
1	0	0
1	1	X

C3	Output of MUX2
1	SUM
0	AXORB

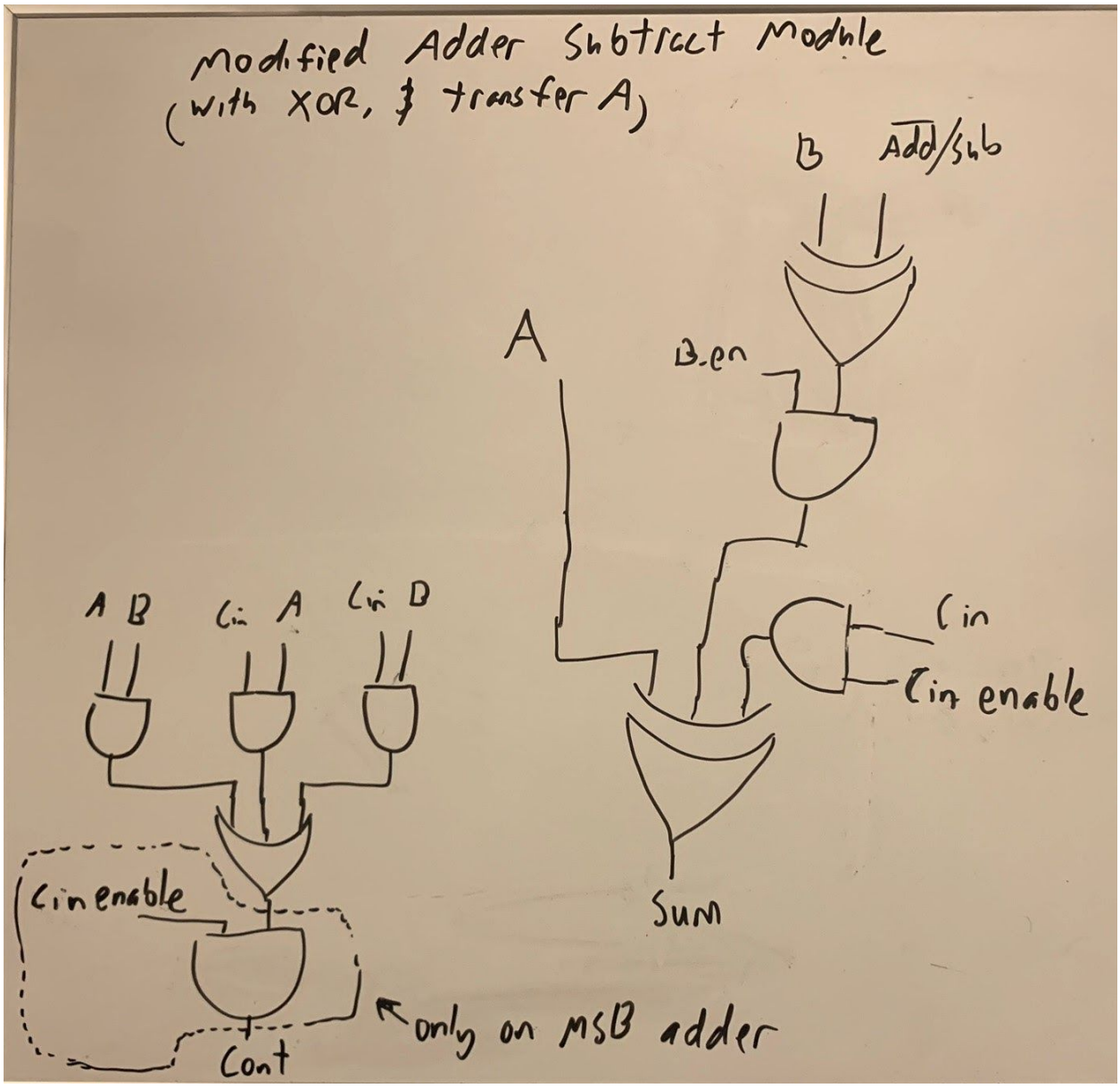
S1	S0	C3	C2	C1
0	0	1	1	0
0	1	1	0	0
1	0	1	0	1
1	1	0	0	0

. **The modified adder with XOR component VHDL code**

```
1   --full add with modified XOR
2
3   LIBRARY ieee ;
4   USE ieee.std_logic_1164.all ;
5
6   entity fulladd_w_xor is
7   PORT ( Cin, x, y , c_in_en: IN STD_LOGIC ;
8            s,Cout: OUT STD_LOGIC ) ;
9   end fulladd_w_xor;
10
11
12
13  ARCHITECTURE LogicFunc OF fulladd_w_xor IS
14  signal c_in_p:  std_logic;
15  BEGIN
16
17  c_in_p <=Cin AND c_in_en;
18  s <= x XOR y XOR c_in_p;
19  Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
20
21  END LogicFunc ;
```

With this design, an issue arises. If XOR is decided, there is still a Carry out signal when the bits are added. This won't affect output (except for the most significant bit) because the next higher significant bit will ignore its carry in anyway. But the carry out of the most significant bit needs to be disabled for proper XOR operation.

Full Adder with XOR for most significant bit

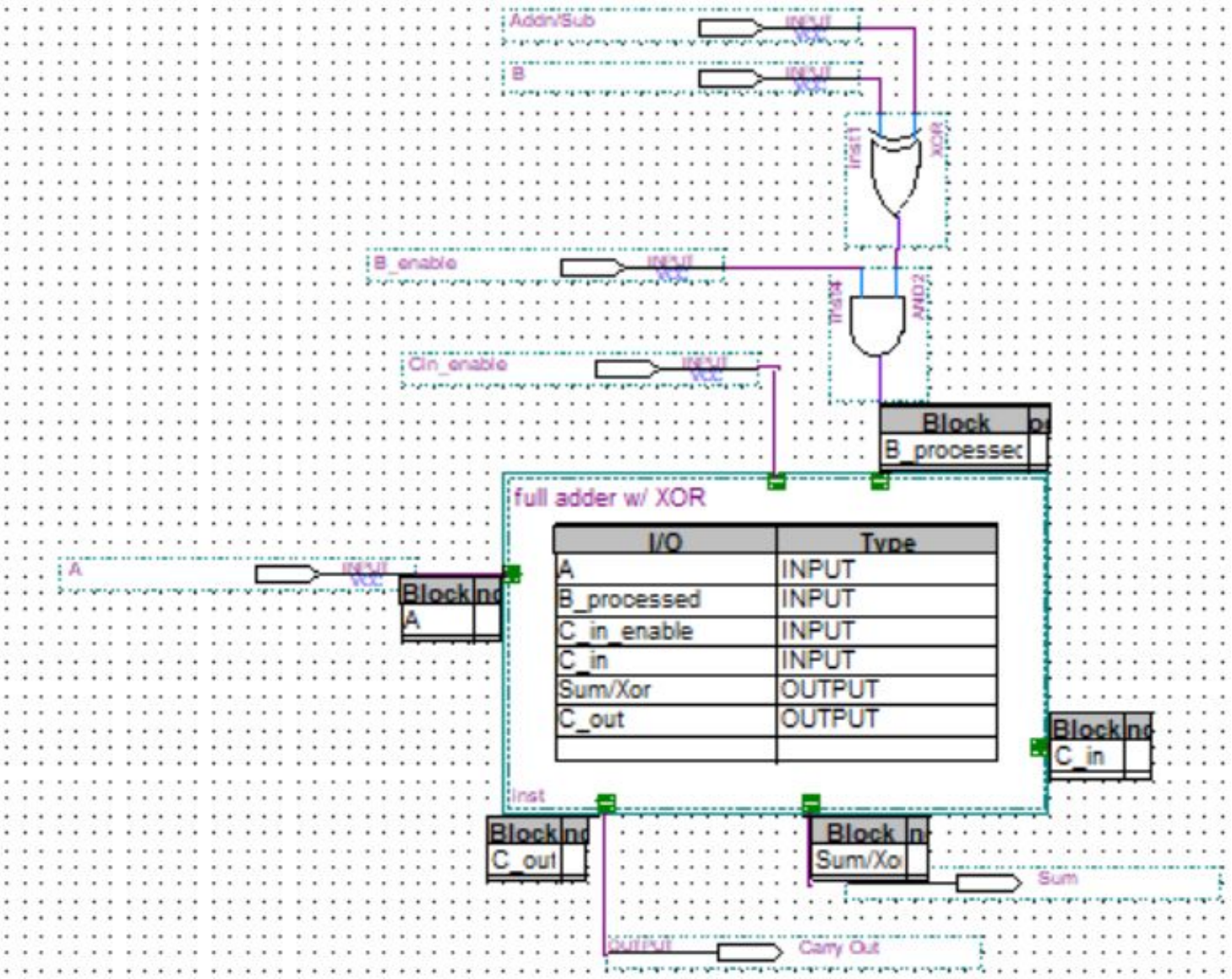


The full adders with XOR for all other bits are the same as the picture above, except they don't need the region within the dotted line.

Code for full adder with XOR (modified for Most significant bit)

```
1  --last bit uses processed cin (c_in_p) for last carry out,
2  -- so that if carry in is disabled there won't be a carry out.
3  --This is only done for last bit, as it adds propogation delay
4
5
6  LIBRARY ieee ;
7  USE ieee.std_logic_1164.all ;
8
9  entity fulladd_w_xor_last_bit is
10 PORT ( Cin, x, y , c_in_en: IN STD_LOGIC ;
11         s,Cout: OUT STD_LOGIC ) ;
12 end fulladd_w_xor_last_bit;
13
14
15
16 ARCHITECTURE LogicFunc OF fulladd_w_xor_last_bit IS
17 signal c_in_p:  std_logic;
18 BEGIN
19
20 c_in_p<=Cin AND c_in_en;
21 s <= x XOR y XOR c_in_p;
22 Cout <= ((x AND y) OR (Cin AND x) OR (Cin AND y)) AND c_in_en ;
23
24 END LogicFunc ;
```

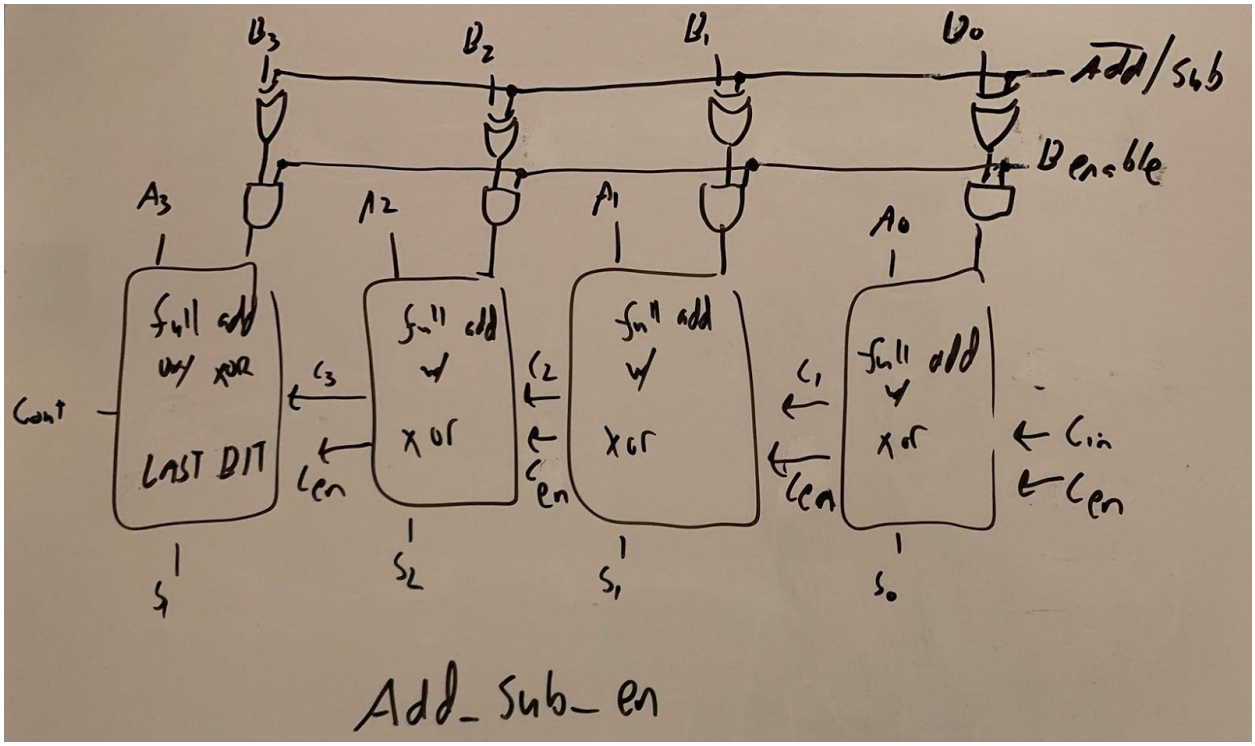
My second improvement is for the MUX301 for B input. Since the three options are B, it's one's complement, or 0. This can be realized with an adder/subtractor with an enable for the B input.



At the cost of one XOR and one AND gate per B input bit, there is no need for a MUX301. Since Our user controls carry-in, the carry-in is made independent of the add/subtract node.

All that's left is a single modified adder/subtractor module.

Connecting bits with adder/subtractor/xor/transfer modules



The code for the 4 bit module is as shown:

Date: October 20, 2019 ../first lecture tests/adder/add_sub_en.vhd Project: Proj_1_alu

```
1   --adder/subtractor modified for EE421 proj. 1
2   --modified in that cin is not connected to add/sub unit
3
4   LIBRARY ieee ;
5   USE ieee.std_logic_1164.all ;
6   USE work.proj_1_package.all ;
7
8   ENTITY add_sub_en IS
9   PORT ( Cin, B_En, Add_Sub, C_in_en : IN STD_LOGIC ;
10          A, B : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
11          S : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
12          Cout : OUT STD_LOGIC;
13          Overflow : out std_logic ) ;
14  END add_sub_en ;
15
16
17  ARCHITECTURE structure OF add_sub_en IS
18  SIGNAL C: STD_LOGIC_VECTOR(1 TO 3) ;
19  signal B_p: std_logic_vector (3 downto 0);
20  signal Cout_w: std_logic;
21  BEGIN
22  B_p(3)<=(Add_Sub XOR B(3)) AND B_En;
23  B_p(2)<=(Add_Sub XOR B(2)) AND B_En;
24  B_p(1)<=(Add_Sub XOR B(1)) AND B_En;
25  B_p(0)<=(Add_Sub XOR B(0)) AND B_En;
26
27
28  stage0: fulladd_w_xor PORT MAP ( Cin, A(0), B_p(0),C_in_en, S(0), C(
    1) ) ;
29  stage1: fulladd_w_xor PORT MAP ( C(1), A(1), B_p(1),C_in_en, S(1), C
    (2) ) ;
30  stage2: fulladd_w_xor PORT MAP ( C(2), A(2), B_p(2),C_in_en, S(2), C
    (3) ) ;
31  stage3: fulladd_w_xor_last_bit PORT MAP ( C(3), A(3), B_p(3),C_in_en
    , S(3), Cout_w ) ;
32
33
34  --discard carryout bit if subtraction
35  Cout <= not(Cout_w) NOR (Add_Sub);
36
37  --check for overflow
38  Overflow <=Cout_w XOR C(3);
39
40  END structure ;
```

Notice that the last carry out bit is discarded as it should be for subtraction. The only way carry-out is propagated is if the module is set to add.

The control logic for the instructions.
The 3 essential control bits are B enable which allows a transfer of A, Add/Subtract bit, and Carry in enable, which allows for XOR operation.

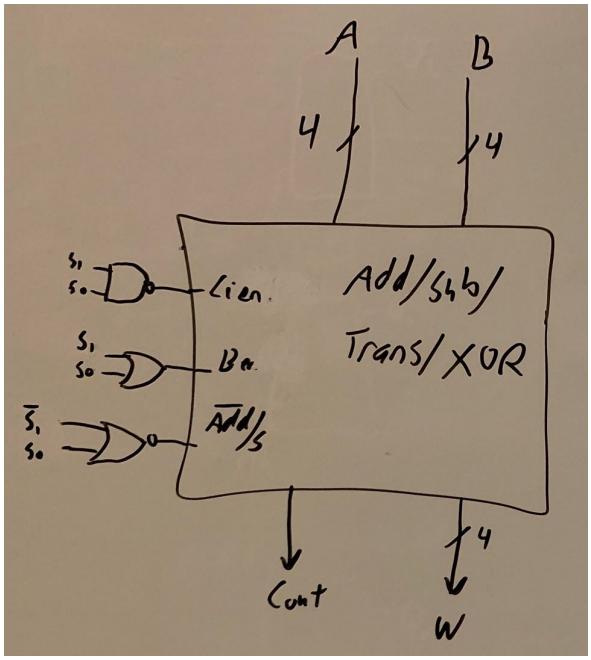
	S_1	S_0	C_i	Addn / Sub	B enable	C_i enable
Transfer A	0	0	0	X	0	X
Incr. A	0	0	1	0	0	1
A + B - 1	0	1	0	0	1	1
A + B	0	1	1	0	1	1
A - B-1	1	0	0	1	1	1
A - B	1	0	1	1	1	1
A XOR B	1	1	X	0	1	0

Control Logic:

Addn/Sub = not(S_1) NOR S_0

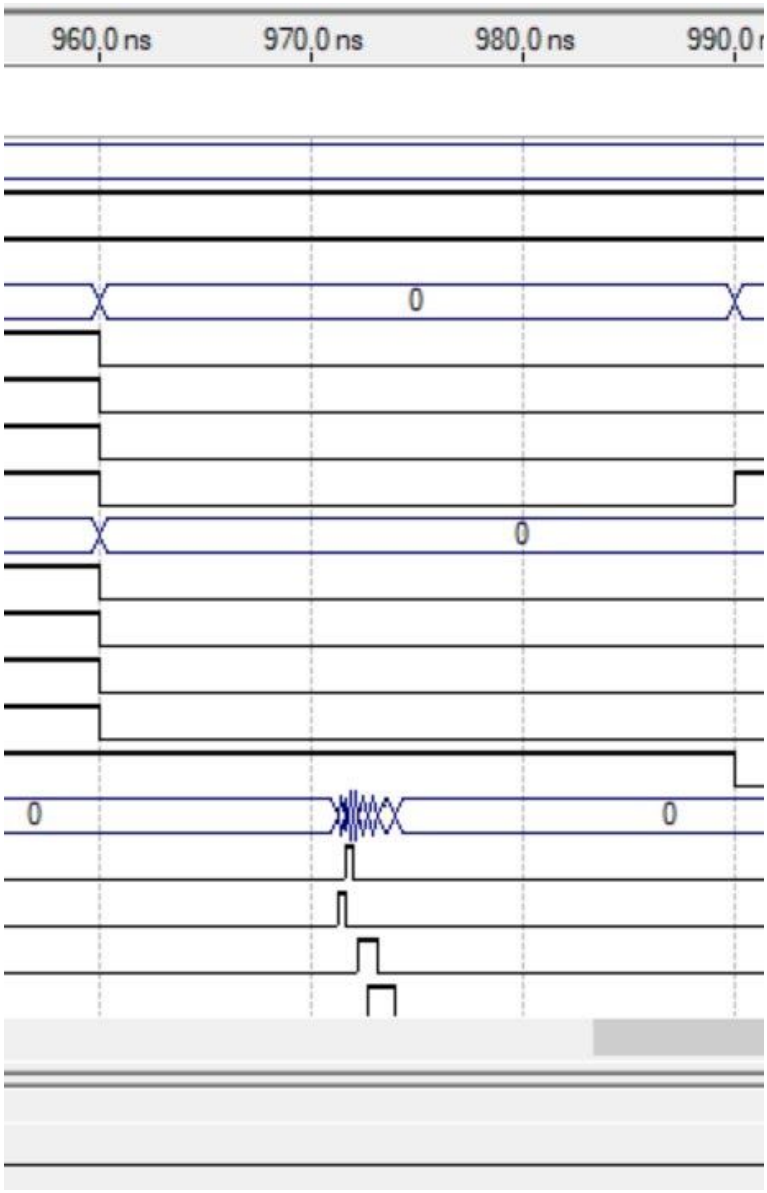
B enable = S_1 OR S_0

C_i enable = S_1 NAND S_0



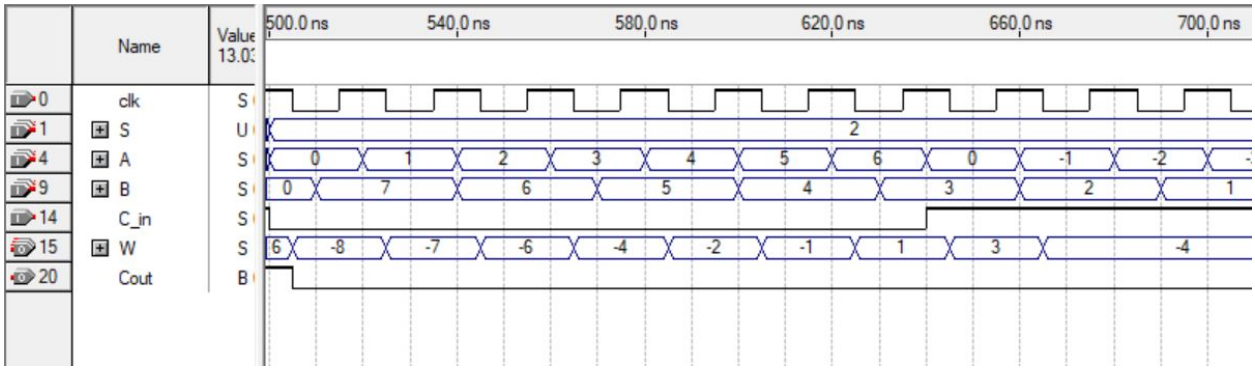
Control Logic set the the ALU module

The last step is to add input and output buffers to synchronize the data flow. For such I used D flip flops. A rising edge flip flop is used for the input. A falling edge flip flop is used for the output. The module will have ½ of a clock period to propagate results. A 50 gigahertz clock will give a 20 nanosecond period. The longest delay in combinational circuit is less than 10 nanoseconds, shown between 970 and 980 nanoseconds.



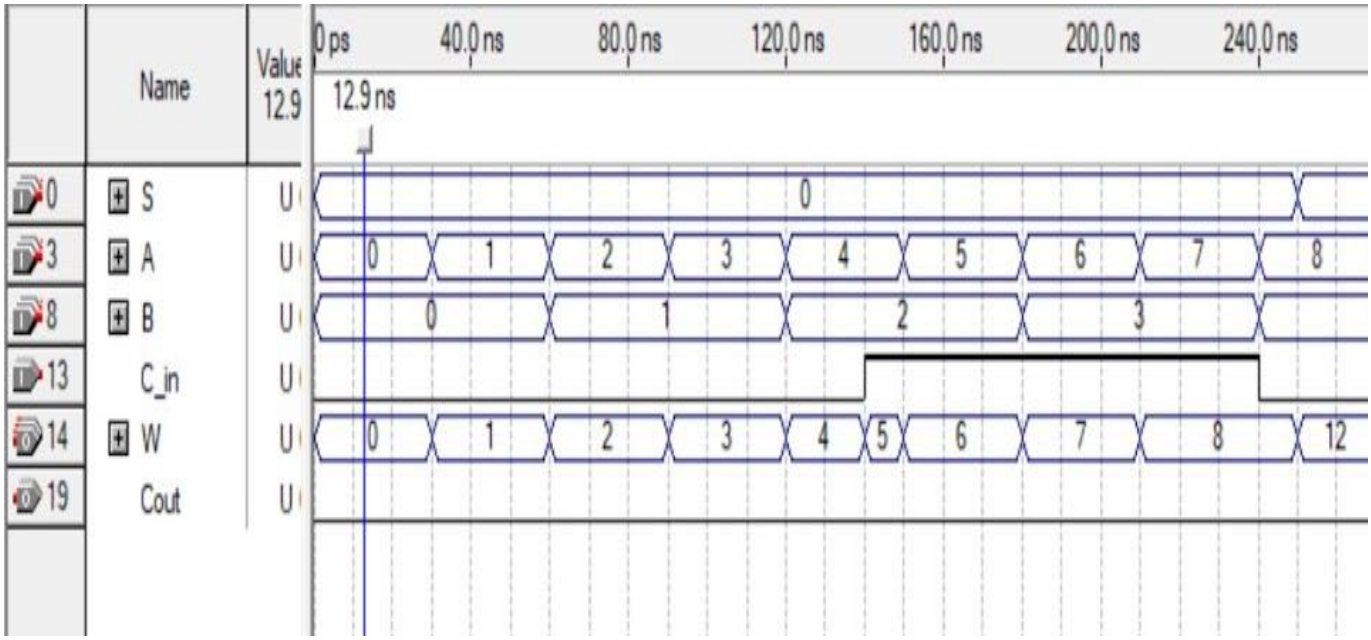
I connect the DFF's to the 50 GHz clock and find no error. The final top level code is the first page of the complete code printout.

Simulation highlights :



Subtraction

Note that C_in is active low. We see that when and when C_in button is depressed and S=2. A and B are stored on the rising edge, and on the falling edge, W gives A-B. the C_in is not depressed, the signal is high and we get A-B-1.

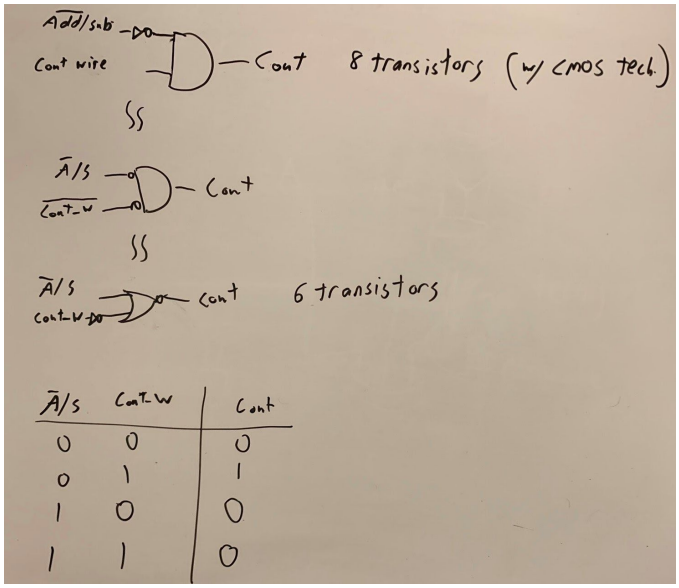


Addition

We see we get transfer when S is set to 0. A gets incremented when c_in is set.

What I learned from this project:

I learned that converting logic to NAND or NOR gates can help reduce transistor counts. For example, for the carry out disable, I was able to save two transistors by converting to NAND gate. As shown



Ā/s	Cout_w	Cout
0	0	0
0	1	1
1	0	0
1	1	0