1. **Consider the function with three inputs (A,B,C) and two outputs (X,Y) that works like this:**

A	B	C	X	Y
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

1. **Design two logic circuits for this function, one using AND, OR and NOT gates only, and one using NAND gates only. *<u>You DO NOT HAVE to draw the circuit</u>*, but it might be helpful to do that to visualize and trace the logic. However, for this question you are only required to write the two formulas — one for computing `X` and one for computing `Y`. They can take the form of a logical equation such as**

   `X := A and B` **or such as** `Y := not-B and (A or C)`**.**

Hw4

Wednesday, October 19, 2022 6:28 PM

A	B	C	X	Y
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

X ↓: $A'BC$, $AB'C'$, $AB'C$, ABC', ABC

Y ↓: $A'B'C'$, $A'B'C$, $A'BC'$, $A'BC$, $AB'C$, ABC

$$X = A'BC + AB'C' + AB'C + ABC' + ABC$$
$$= A'BC + ABC + AB'C' + AB'C + ABC'$$
$$= BC(A'+A) + AB'(C'+C) + ABC'$$
$$= BC + AB' + ABC'$$
$$= A + C(B+B') + ABC'$$
$$= A + C + ABC'$$
$$= A(C + BC')$$
$$= AB(C+C').$$

X = A and B

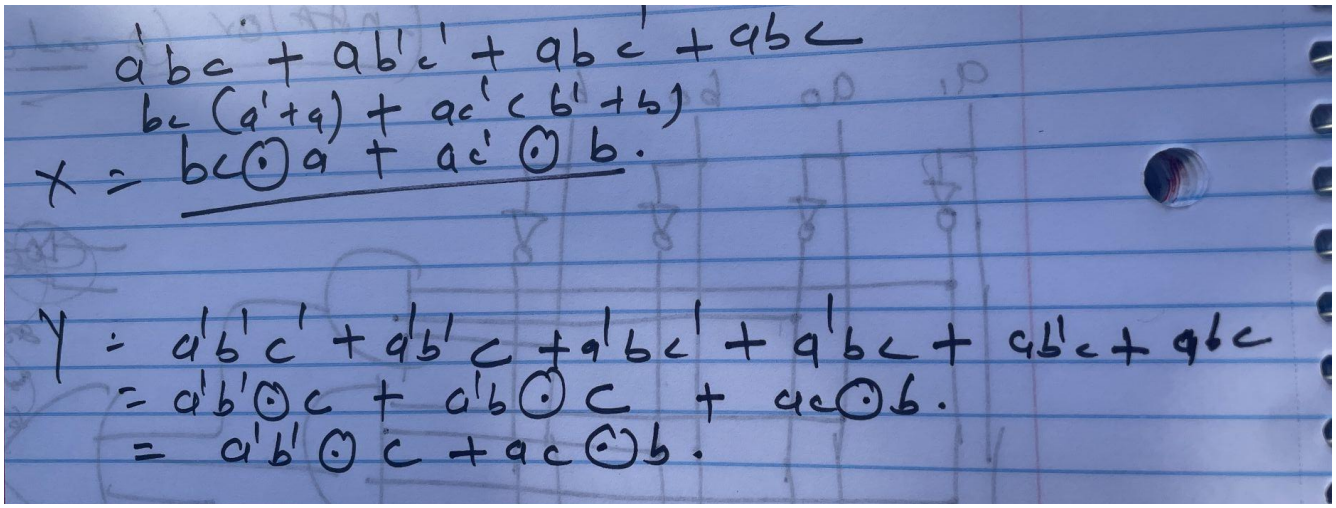
$$Y = A'B'C' + A'B'C + A'BC' + A'BC + AB'C + ABC$$
$$= A'B'(C+C') + A'B(C'+C) + AC(B+B')$$
$$= A'B' + A'B + AC$$
$$= A'(B'+B) + AC$$
$$= A' + AC$$

= NOT A OR (A AND C)

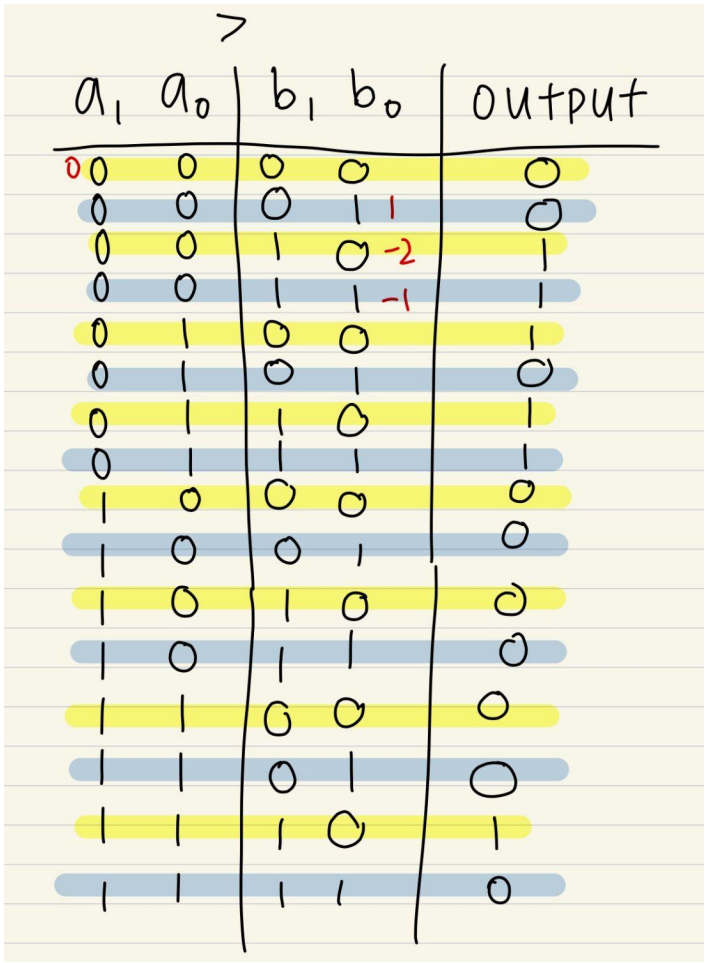
AND/OR/NOT:

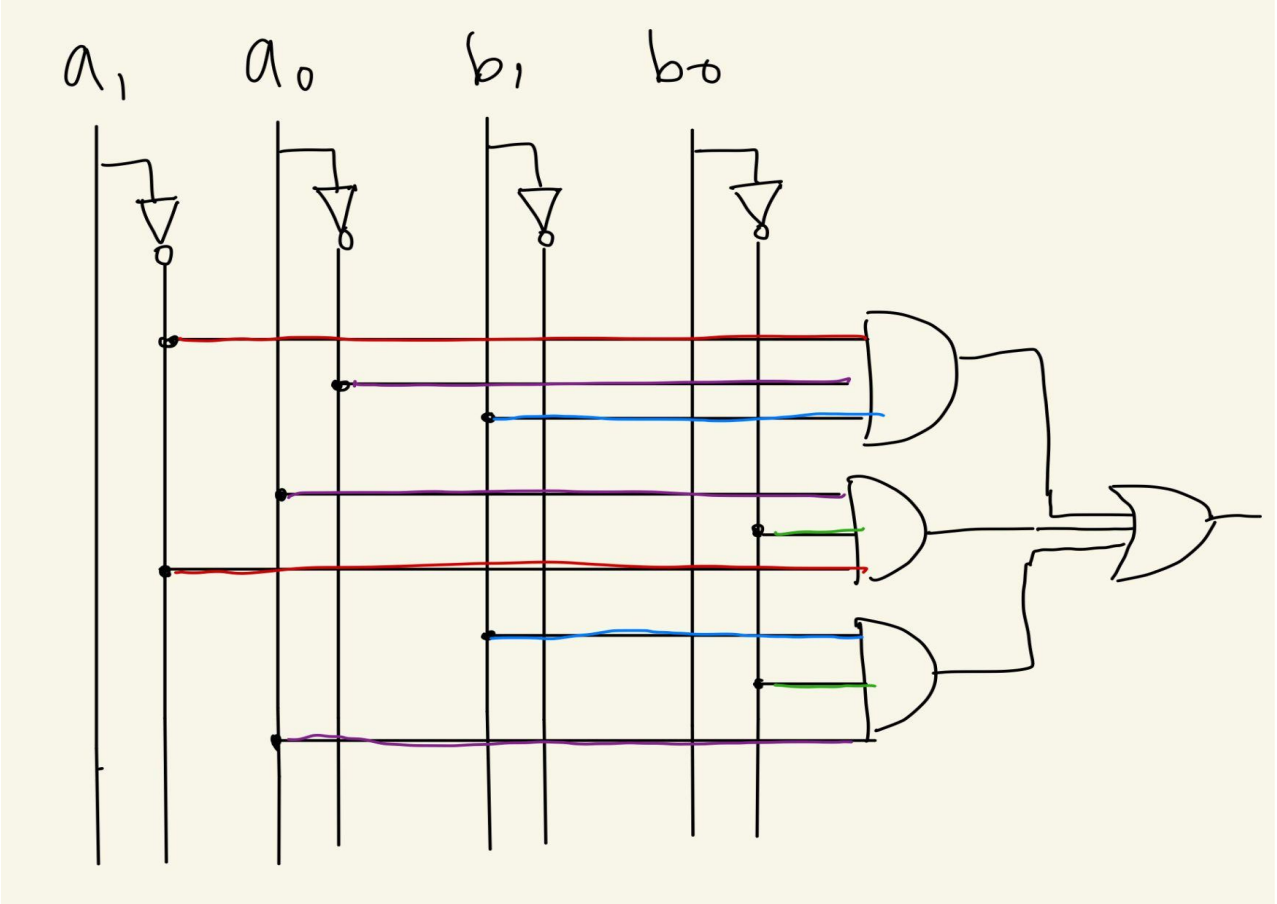
Y: (not A and not B xor C) OR (A and C xor B)

X: (B and C xor A) OR (A and not C xor B)



2. **Draw a logic circuit that compares two 2-bit signed numbers as follows. It should have four inputs a_1, a_0, b_1, and b_0. a_1a_0 is a 2-bit signed number (call it a) and b_1b_0 is a 2-bit signed number (call it b). The circuit has one output, c, which is 1 if a > b and 0 otherwise.**


>
a_1 a_0 b_1 b_0 output
0
1
-2
-1



$a_1'a_0'b_1b_0' + a_1a_0'b_1b_0 + a_1'a_0b_1'b_0' + a_1'a_0b_1b_0' + a_1a_0b_1'b_0'$

$a_1'a_0'b_1(b_0+b_0') + a_1'a_0b_0'(b_1+b_1') + a_0b_1b_0'(a_1+a_1')$

$a_1a_0'b_1 + a_1'a_0b_0' + b_1b_0'a_0$

$a_1a_0'b_1 + a_0b_0'(a_1'+b_0')$

$a_1a_0'b_1 + a_0b_0'(a_1+b_1)$

$a_1'a_0'b_1 + a_0b_0'a_1 + b_1b_0'a_0.$

$a_1'a_0b_0 + a_1'a_0B_0' + a_1a_0B_1B_0$

and.

A(not A) or b + (not A - not B) or A + not B

(a or b

(-not A) or (a and c)

3. **Given a 32-bit register, write logic instructions to perform the following operations. For parts (c) and (f) assume an unsigned interpretation; for part (d) assume a signed interpretation.**

    1. **Clear all even numbered bits1**
    2. **Set the last three bits.**
    3. **Compute the remainder when divided by 8.**
    4. **Make the value -1**
    5. **Complement the two highest order bits**
    6. **Compute the largest multiple of 8 less than or equal to the value itself**

s1	**0**	
s2		
s3		
s4		
s5		
s6		
s7		
s8		
value		
limit:	**0x0FFFF**	
start:	**LOAD**	**s1**
	AND	**s2**
	STORE	**a**
	LOAD	**s3**
	OR	**a**

	STORE	**a**
	LOAD	**s6**
	AND	**a**
	STORE	**s6**
	NOT	**s6**
	STORE	**s6**
	LOAD	**s7**
	LOAD	**a**
	OR	**s7**
	STORE	**s7**
	LOAD	**s6**
	AND	**s7**
	STORE	**a**
	LOAD	**a**
	MOD	**8**
	STORE	**s8**
	LOAD	**a**
	SUB	**s8**
	STORE	**a**
end:	**JMP**	**end**

a	**<-**	**S1 AND s2**
a	**<-**	**S3 OR a**

s6	**<-**	**S6 AND a**
s6	**<-**	**NOT s6**
s7	**<-**	**A OR s7**
a	**<-**	**S6 AND s7**
s8	**<-**	**A MOD 00001000**
a	**<-**	**A SUB s8**

4. **For the sample single-accumulator computer discussed in class, write a complete assembly language program in the `stanley/penguin` language that sends the values `0` through `255` out to port `0x8`. NOTE: the machine code for this will be written in the next problem.**

	JMP	**start**
val:	**0**	
limit:	**255**	
start:	**LOAD**	**val**
	WRITE	**8**
	ADD	**1**
	STORE	**val**
	SUB	**limit**
	JLZ	**start**
end:	**JMP**	**end**

5. **Translate your assembly language program in the previous problem to machine language.**

C0000003

00000001

00000FF0

00000001 load

30000008 write

40000001 add

10000001 store

50000002 sub

E0000003 jlz

C0000009 end

6. **For the sample single-accumulator computer discussed in class, write a complete assembly language program in the `stanley/penguin` language that computes a greatest common divisor. Assume the two inputs are read in from port `0x100`. Write the result to port `0x200`. You do not need to write machine code for this problem.**

	JMP	start	What is happening:
val1:	0		List variables
val2:	0		
tmp:	0		
remainder:	0		
start:	READ	256	Take in inputs
	STORE	val1	

	READ	**256**	
	STORE	**val2**	
compare:	**LOAD**	**val1**	
	SUB	**val2**	**See which is bigger**
	JZ	**write**	
	LOAD	**val1**	**Exchange val 1 and val 2**
	STORE	**tmp**	
	LOAD	**val2**	
	STORE	**val1**	
	LOAD	**tmp**	
	STORE	**val2**	
	LOAD	**val1**	
	MOD	**val2**	**Division**
	JZ	**write**	
	LOAD	**val1**	**Move remainder to val1**
	STORE	**remainder**	
	JMP	**compare**	
write:	**WRITE**	**512**	**Write GCD**
end:	**JMP**	**end**	**End program**

1. **Start program.**
2. **Load val1 and val 2**
3. **Check if they're equal. If yes, go to step 9, else go to the next step.**
4. **Is val1 greater than val2? If yes, go to step 6. Else, go to next step**
5. **Exchange val1 and val2 so that val1 is bigger**
6. **Do division (val1/ val2)**

7. Is there a remainder? If yes, go to next step, else go to step 9
8. Move the remainder into val1 and go to step 4
9. Save val2 as the greatest common divisor
10. Write the GCD to the port
11. End program.

7. **For the sample single-accumulator computer discussed in class, give a code fragment, in assembly language of the `stanley/penguin` language, that swaps the accumulator and memory address `0x30AA`. You do not need to write machine code for this problem.**

tmp1	0	
tmp2	0	
Start:	STORE	tmp1
	LOAD	0x30AA
	STORE	tmp2
	LOAD	tmp1
	STORE	0x30AA
	LOAD	tmp2

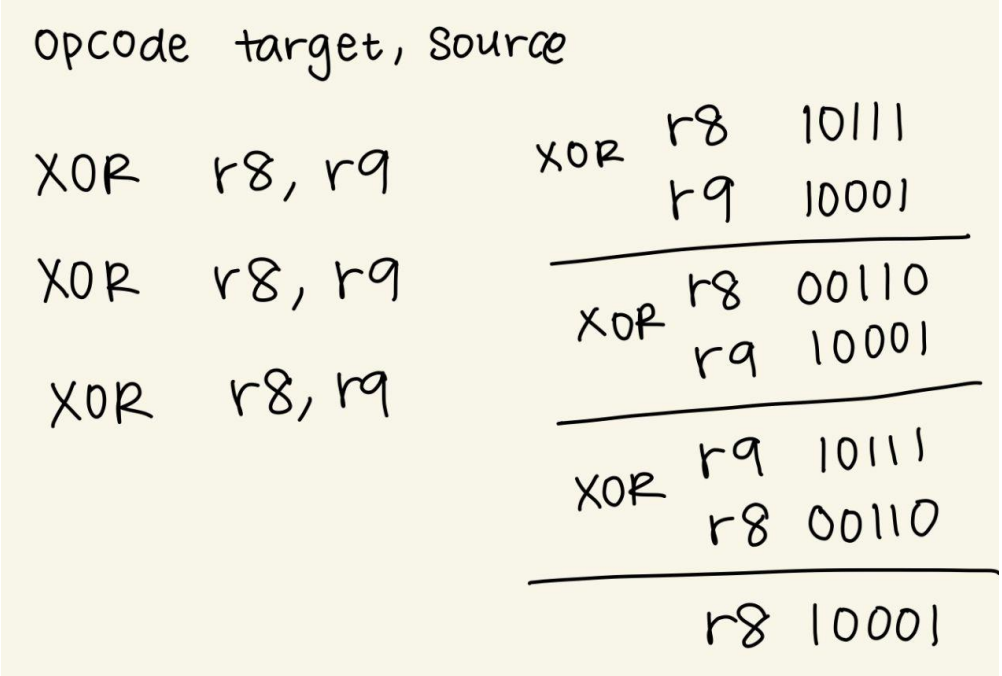
8. **For the sample single-accumulator computer discussed in class, give a code fragment, in assembly language of the `stanley/penguin` language that has the effect of jumping to the code at address `0x837BBE1` if the value in the accumulator is greater than or equal to `0`. You do not need to write machine code for this problem.**

val:	0	
start	LOAD	val
	JZ	0x837BBE1
	JGZ	0x837BBE1

9.

Part 1 of 2: Explain, at a high-level, what the following sequence of instructions does. In other words, suppose a programmer has stored data in `r8` and `r9`. After executing these instructions, what does the programmer notice about the data?

```
xor r8, r9
xor r9, r8
xor r8, r9
```



The sequence of instructions applies the exclusive or gate (XOR) in each line to r8 and r9. After the instructions have been executed, the bits in r8 and r9 will be set to 1 if the corresponding bits are different, and 0 if they are the same. So the original values in the registers will be swapped. The statements will swap the values in r8 and r9 after 3 consecutive exclusive or gates.

- ExclusiveOr (BITS AT r8) XOR (BITS AT r9)
- ExclusiveOr (BITS AT r9) XOR (BITS AT r8)
- ExclusiveOr (BITS AT r8) XOR (BITS AT r9)

Part 2 of 2: Also state as briefly as possible why that effect happens.

The exclusive or has the effect of changing the value of the individual, corresponding bits. After 3 consecutive exclusive or's , the data values in the registers will be swapped.