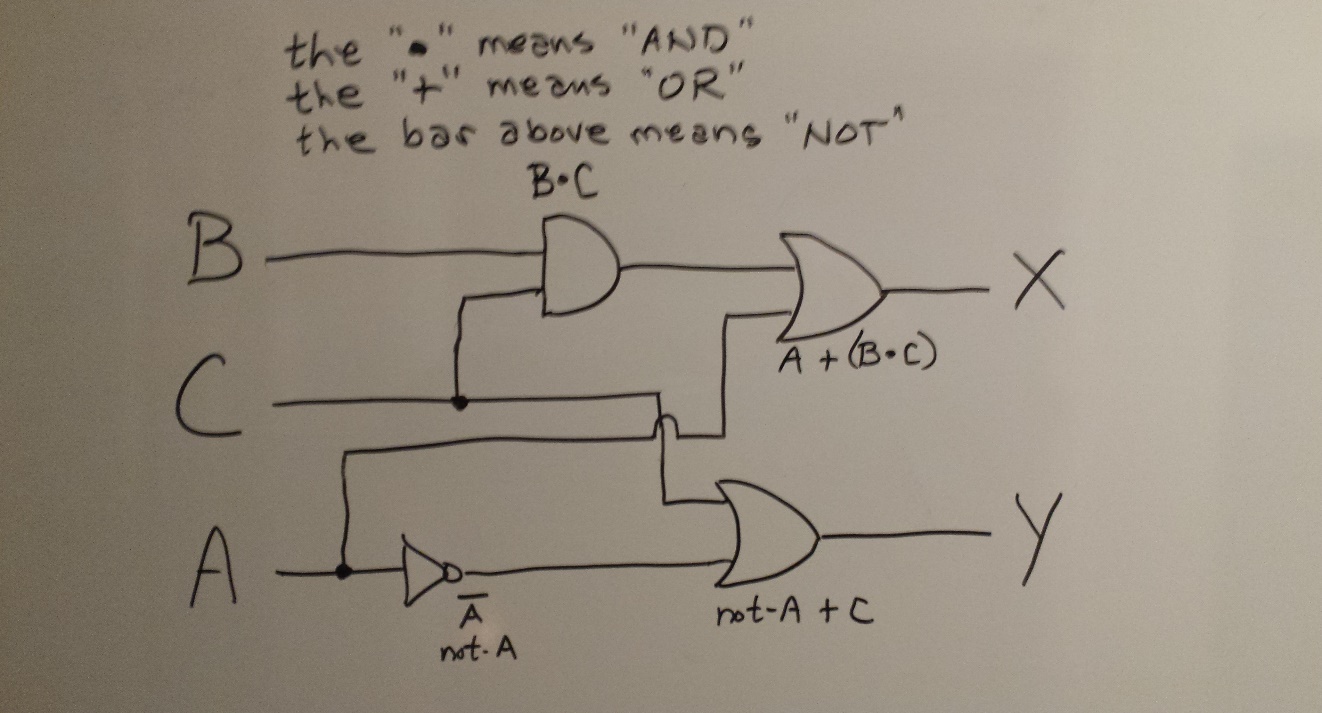
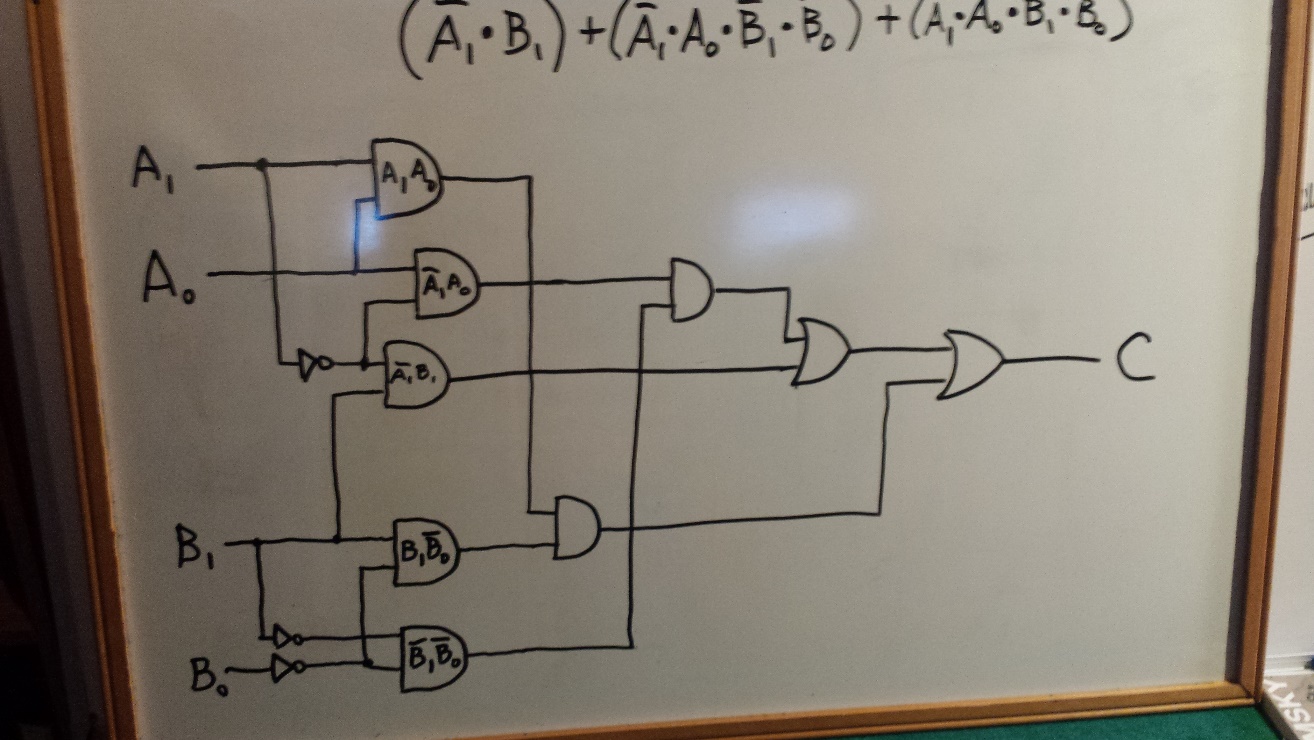
# Homework 4 solutions ~ CMSI 284 Spring 2023

1. It’s a little clearer if we use the ↑ notation for NAND, but here are both:

* X = A + BC = (AA)(BC) = (A ↑ A) ↑ (B ↑ C)
* Y = A + C = A(CC) = A ↑ (C ↑ C)
* To me, this looks like it should be:
  + X = A + (BC) -or- X = A or (B and C)
  + Y = ⌐A + C -or- Y = not-A or C
* The NAND version then looks like:
  + X = (A **↑** A) **↑** (B **↑** C) -or- X = (A nand A) nand (B nand C)
  + Y = A **↑** (C **↑** C) -or- Y = A nand (C nand C)



1. a1b1 + a1a0b1b0 +a1a0b1b0



1. a) AND the value with 0xAAAAAAAA ~   
    ANDing with 1010 1010 1010 1010 1010 1010 1010 1010 will set all even bits to zero

b) OR the value with 0x00000007 ~   
 ORing with a “1” produces a “1” no matter what is already in that   
 location

c) AND the value with 0x00000007 ~   
 division by 8 is the same as shifting the bits to the right by three; the  
 ANDing with 0111 will provide the remainder only

d) OR the value with 0xFFFFFFFF ~   
 ORing sets ALL bits to “1” which is -1

e) XOR the value with 0xC0000000 ~  
 XORing with “C” will change those two bits as follows:  
 11 → 00; 00 → 11; 01 →10; and 10 → 01

***ALSO***

NANDing with “C” will change those two bits as follows:  
 11 → 00; 00 → 11; 01 →10; and 10 → 01

f) AND with 0xFFFFFFF8 ~   
 masking off the last three bits will provide the multiple of 8 which  
 will be the largest value to not have a remainder

1. Program that sends the values 0 through 255 out to port 0x8. NOTE: this is just a re-working of the demonstration problem that outputs the powers of two.

JMP start ; skip over the data area

value: 0

one: 1

start: LOAD value ; get current value

WRITE 0x08 ; and write it out

ADD one ; this will be the next value

STORE value ; store it for next time through the loop

SUB 0x100 ; will be zero only when the next value is limit

JLZ start ; not yet zero means we have more to do

end: JMP end ; only way to stop the program

1. Machine language for the previous problem.

C0000003

00000000

00000001

00000001

30000008

40000002

10000001

50000100

E0000003

C0000009

1. Computes a greatest common divisor. Assume the two inputs are read in from port 0x100 in two successive read operations. Write the result to port 0x200. We can use Euclid's algorithm:

gcd(a, b) = a if b = 0 else gcd(b, a mod b)

JMP start

a: 0

b: 0

a\_mod\_b: 0 ; yes, we really need this

start: READ 0x100 ; read first value

STORE a ; and save it

READ 0x100 ; read next value

STORE b ; and save that, too

top: LOAD b ; load the value at y

JZ done ; if b == 0 go return a

STORE a\_mod\_b ; else store that as “a\_mod\_b”

LOAD a ; get value at a

MOD b ; mod it by value at b

STORE b ; store that for next b ~ a\_mod\_b := a % b

LOAD a\_mod\_b ; load previous b value

STORE a ; store that at a ~ next\_a := a\_mod\_b

JMP top ; the JMP is at the top here

done: LOAD a ; load the value at a

WRITE 0x200 ; This is the real "return a"

end: JMP end

1. Swaps the accumulator and memory address 0x30AA. Here’s an answer with comments explaining the approach. I'm calling the original value in the accumulator a, and the original value in 30AA x. Assume there's some other variable t lying around; we’re gonna need it.

STORE t1 ; t1 holds a

LOAD 0x30AA

STORE t2 ; t2 holds x

LOAD t1

STORE 0x30AA ; now 30AA holds original value of a

LOAD t2 ; now accumulator holds x

**There is an xor-style solution, too:**

XOR 0x30AA ; acc holds a xor x

STORE t ; t holds a xor x

XOR 0x30AA ; acc holds a xor x xor x, which is a

STORE 0x30AA ; 30AA holds a

XOR t ; now accumulator holds a xor a xor x, which is x, all done!

1. Jump to address 0x837BBE1 if the value in the accumulator is greater than or equal to 0.

JGZ 0x837BBE1

JZ 0x837BBE1

1. The values in r8 and r9 are swapped. Here’s how:

|  |  |  |
| --- | --- | --- |
| **Instruction** | **r8 [call this “x”]** | **r9 [call this “y”]** |
| **start:** | X = 1100 0111 [in r8] | Y = 0110 1111 [in r9] |
| xor r8, r9 | X = 1010 0110 [in r8]: x xor y | Y = 0110 1111 [in r9]: no change |
| xor r9, r8 | X = 1010 0110 [in r8]: no change | Y = 1100 0111 [in r9]:  y xor with (x xor y) |
| xor r8, r9 | X = 0110 1111 [in r8]:  x xor (y xor (x xor y))  which was original Y value in r9 | Y = 1100 0111 which was original X value in r8 |

This is an example of the XOR Swap Algorithm: <https://en.wikipedia.org/wiki/XOR_swap_algorithm>

Note that this is EXACTLY what happens when swapping two variables without a “temp” variable using subtraction/addition/subtraction, but this is slower since XORing is at the bit level and works with one clock cycle:

|  |  |  |
| --- | --- | --- |
| Operation | A | B |
| Start | 23 | 17 |
| A = A – B | 6 | 17 |
| B = A + B | 6 | 23 |
| A = B – A | 17 | 23 |

“The XOR Swap Algorithm eliminates the usage of the intermediate register which is a limited resource in machine language programming. It also eliminates two memory access cycles which would be expensive compared to a register operation.”

Source: <https://academickids.com/encyclopedia/index.php/Xor_swap_algorithm>