1. Design two logic circuits for this function, one using AND, OR and NOT gates only, and one using NAND gates only.

**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**

**A || BC**

**A or BC**

**A** because x is always on when A is on irrespective of B and C

**BC** because line 4 A is off but BC are on

No overlap in rules

***NAND gate produces an output which is false only if all its inputs are true***

***so if all inputs true, output is false***

**¬A and (B or C) ¬A \* (B || C)**

***// Above logic seems correct but not sure if NAND gates can be expressed this way?***

**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**

**¬A || C**

**¬A** because y is always on when ¬A irrespective of B and C

C because y is always on when C is on irrespective of A and B

Two rules overlap (both occur) on lines 2 and 4

***NAND Gate:***

**A and ¬C A \* ¬C**

***// Above logic seems correct but not sure if NAND gates can be expressed this way?***

2. Draw a logic circuit that compares two 2-bit signed numbers as follows. It should have four inputs a1, a0, b1, and b0. a1a0 is a 2-bit signed number (call it a) and b1b0 is a 2-bit signed number (call it b). The circuit has one output, c, which is 1 if a > b and 0 otherwise.

***// Question 2 is only question skipped***

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 bits.
  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

**eax is is "Extended Accumulator," a 32-bit general-purpose register in x86 architecture, or you can use “immediate” in place of “eax”**

a. **Clear all even number bits of their** values so use 0101…0101 HEX: 5…5

01010101010101010101010101010101 = 0x55555555

AND eax, 0x55555555

RATIONALE:

ODDS: If both corresponding bits are 1, the result is 1 (So odd numbers not affected, i.e. 0 AND 1 is still 0 while 1 AND 1 is still 1)

EVENS: If either bit is 0, the result is 0.

**b. Set last three bits**

Set the last three bits

OR operation with a mask that has the last three bits set to 1

00000000000000000000000000000111 = 0x00000007

OR eax, 0x00000007

RATIONALE:

ODDS: “1” and “OR” overrides everything to “1”

EVENS: Unaffected because if 0 OR 0 = 0, while 1 OR 0 = 1

**c. Compute the remainder when divided by 8 (unsigned):**

AND eax, 0x00000007

Keep the last three bits, which represent the remainder

**d. Make the value -1 (signed):**

NOT eax 0x00000000 Invert all bits to get -1 in two's complement representation

MOV eax, 0xFFFFFFFF Moves the value -1 (all bits set to 1) into the register.

**e. Complement the two highest order bits:**

XOR eax 0xC0000000

XOR with a mask that flips the two highest bits

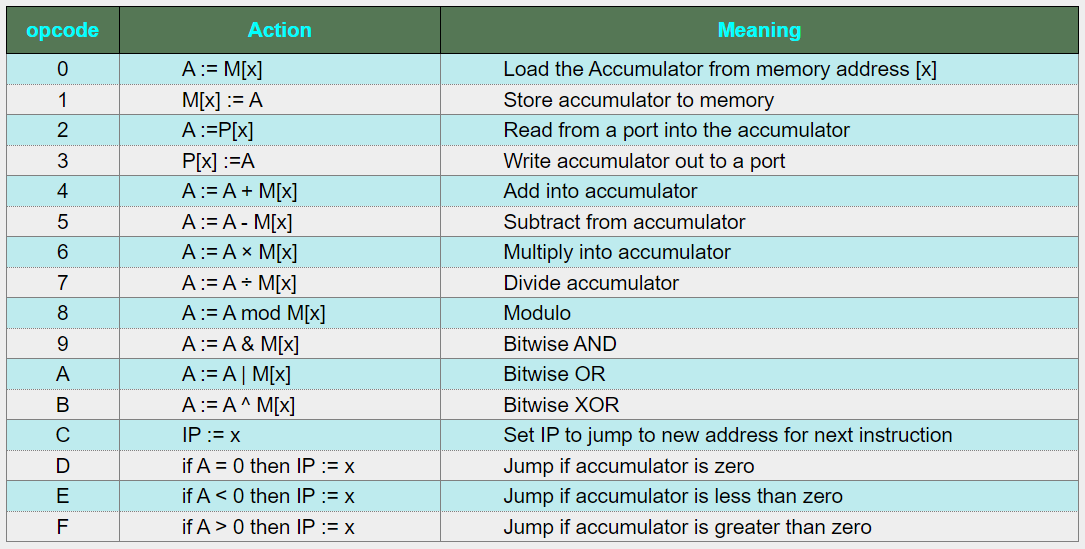
**f. Compute the largest multiple of 8 less than or equal to the value itself:**

AND eax 0xFFFFFFF8

Clear the last three bits to align to the nearest multiple of 8

**Stanley/Penguin explained in middle of week 6 lecture notes**

*https://bjohnson.lmu.build/cmsi2210web/week06.html*



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.

**0 A00 ; Load 0 into accumulator**

**loop: 3 08 ; Write accumulator to port 0x8**

**4 A01 ; Add 1 to accumulator**

**F 02 ; Jump to address 02 if accumulator > 0**

**C 00 ; Jump to address 00 (reset)**

LDAC 0 ; Initialize accumulator to 0

; Loop to send values 0 through 255

LOOP: ; start LOOP

OUT 0x8 ; Send accumulator value to port 0x8

ADDC 1 ; Adds 1 to the accumulator.

CMP 256 ; Compare accumulator with 256

BNE LOOP ; If not equal, jump back to LOOP

LDI 0x00 ; Initialize accumulator with 0

OUT 0x8 ; Output accumulator to port 0x8

LOOP:

ADD 0x01 ; Increment accumulator by 1

OUT 0x8 ; Output accumulator to port 0x8

CMP 0xFF ; Compare accumulator with 255

JLE LOOP ; Jump to LOOP if accumulator is less than or equal to 255

HLT ; Halt the program

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

**0x00000000 ; LDI 0x00**

**0x30000008 ; OUT 0x8**

**0x40000001 ; ADD 0x01**

**0x30000008 ; OUT 0x8**

**0x500000FF ; CMP 0xFF**

**0xD0000003 ; JLE 0x00000003**

**0x60000000 ; HLT**

Initialize accumulator with 0:

Assembly: LDI 0x00

Machine: 0x00000000

Output accumulator to port 0x8:

Assembly: OUT 0x8

Machine: 0x30000008

Increment accumulator:

Assembly: ADD 0x01

Machine: 0x40000001

Output accumulator to port 0x8:

Assembly: OUT 0x8

Machine: 0x30000008

Compare accumulator with 255:

Assembly: CMP 0xFF

Machine: 0x500000FF

Jump to start of loop if accumulator is less than or equal to 255:

Assembly: JLE 0x00000003

Machine: 0xD0000003

Halt:

Assembly: HLT

Machine: 0x60000000

|  |  |  |
| --- | --- | --- |
| Initialize accumulator with 0: | 0x00000000 | LDI 0x00 |
| Output accumulator to port 0x8: | 0x30000008 | OUT 0x8 |
| Increment accumulator: | 0x40000001 | ADD 0x01 |
| Output accumulator to port 0x8: | 0x30000008 | OUT 0x8 |
| Compare accumulator with 255: | 0x500000FF | CMP 0xFF |
| Jump to start of loop if accumulator is less than or equal to 255: | 0x00000003 | JLE 0x00000003 |
| Halt: | 0x60000000 | HLT |

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.

# Read inputs from port 0x100

2 100 # A := P[100]

1 30 # M[30] := A (store first input)

2 100 # A := P[100]

1 31 # M[31] := A (store second input)

# GCD calculation loop

LOOP:

0 30 # A := M[30]

8 31 # A := A mod M[31]

D END # if A = 0 then IP := END

1 32 # M[32] := A (store remainder)

0 31 # A := M[31]

1 30 # M[30] := A

0 32 # A := M[32]

1 31 # M[31] := A

C LOOP # IP := LOOP (continue loop)

END:

0 31 # A := M[31] (load GCD result)

3 200 # P[200] := A (output result to port 0x200)

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.

1 30AB # M[30AB] := A (temporary storage)

0 30AA # A := M[30AA]

1 30AA # M[30AA] := A (old accumulator value)

0 30AB # A := M[30AB] (restore original accumulator value)

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.

E SKIP # if A < 0 then IP := SKIP

C 837BBE1 # IP := 837BBE1 (jump if A >= 0)

SKIP:

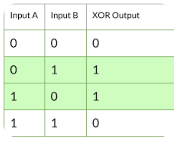
**9. Part 1 of 2**: Explain, at a high-level, what the following sequence of instructions does. In other words, suppose a programmer has stored data in **r8** and **r9**. After executing these instructions, what does the programmer notice about the data?

xor r8, r9

xor r9, r8

xor r8, r9

**xor r8, r9:** bitwise XOR operation between the values in r8 and r9, storing the result in r8. In essence, it combines the bits of both values using the XOR logic or checks for inequality (XOR)



**xor r9, r8:** another XOR operation between the current (original) value in r9 and the newly modified value in r8 (r8 and r9). This operation effectively swaps the values between the two registers.

**xor r8, r9:** the original value of r9 is restored to r8. By XORing r8 (which now contains the original value of r9) with the current value in r9 (which contains the original value of r8), the values have been swapped to their original position.

Thus values in r8 and r9 have been swapped. The original value that was in r8 is now in r9, and the original value that was in r9 is now in r8

**9. Part 2 of 2**: Briefly state why this happens

If you perform XOR on a value twice with the same other value, you end up with the original value. The values in r8 and r9 are swapped without needing to use a temporary variable.