1)

AND/OR/NOT formulas

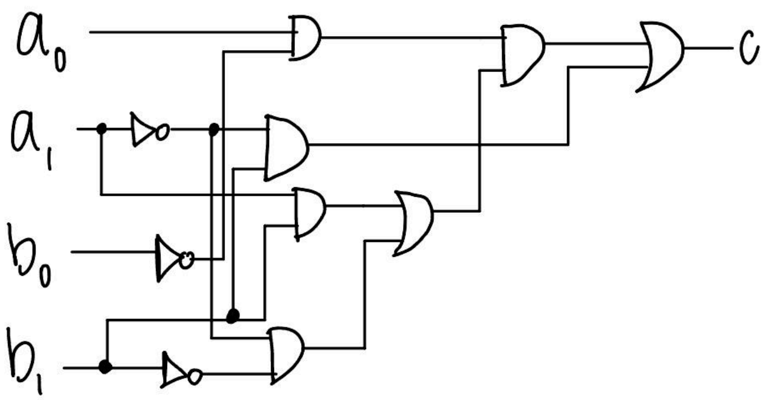
- X=A + (B · C)X = A \;+\; (B \cdot C)X=A+(B · C)

- Y=A‾ + CY = \overline{A} \;+\; CY=A+C

NAND-only versions

X = NAND(NAND(A,A),NAND(NAND(B,C),NAND(B,C)))
Y=NAND(NAND(C,C), A) = A * C = A + C

2)



if a>b: c=1

3)
32-bit register bit-twiddling (logic-only)

(a) Clear all even-numbered bits (bit 0,2,4,…)

AND R, R, 0xAAAAAAAA

(b) Set the last three bits (bits 0,1,2)

```
OR  R, R, 0x00000007
```

(c) Unsigned remainder mod 8

```
AND R, R, 0x00000007    ; R ← R % 8
```

(d) Make the value −1 (signed all-ones)

```
XOR R, R, R             ; R ← 0
NOT R, R                ; R ← 0xFFFFFFFF
```

(e) Complement the two highest-order bits (bits 31 and 30)

```
XOR R, R, 0xC0000000
```

(f) Largest unsigned multiple of 8 ≤ value

```
AND R, R, 0xFFFFFFF8
```

4)

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <limits.h>
#include <string.h>

int main(void) {
    char line[256];
    long n = 0;

    // Ask until we get a valid positive integer
    while (1) {
        printf("Enter a positive integer N: ");
        if (!fgets(line, sizeof line, stdin)) {
            fprintf(stderr, "Input error.\n");
            return 1;
        }

        // strip trailing newline
```

```c
        line[strcspn(line, "\n")] = '\0';

        // convert with error checking
        errno = 0;
        char *end = NULL;
        long value = strtol(line, &end, 10);

        // check: had digits, consumed all chars (no junk), no range error, positive
        if (end == line || *end != '\0' || errno == ERANGE || value <= 0) {
            printf("Please enter a valid positive integer.\n");
            continue;
        }

        n = value;
        break;
    }

    // Fizz-Buzz from 1 to N
    for (long i = 1; i <= n; ++i) {
        int by3 = (i % 3 == 0);
        int by5 = (i % 5 == 0);

        if (by3 && by5) {
            printf("fizz-buzz\n");
        } else if (by3) {
            printf("fizz\n");
        } else if (by5) {
            printf("buzz\n");
        } else {
            printf("%ld\n", i);
        }
    }

    return 0;
}
```

5)

```
Assembly (writes 0 → 255 to port 0x8)
; Opcodes:
; 0 LOAD   A := M[x]
; 1 STORE  M[x] := A
; 2 READ   A := P[x]
; 3 WRITE  P[x] := A
; 4 ADD    A := A + M[x]
; 5 SUB    A := A - M[x]
; C JMP    IP := x
; D JZ     if A = 0 then IP := x
```

```
        LOAD ZERO        ; A ← 0
        STORE COUNT       ; COUNT ← 0

LOOP:   LOAD COUNT        ; A ← COUNT
        WRITE 0x00000008  ; output A to port 8

        LOAD COUNT
        ADD  ONE         ; A ← COUNT + 1
        STORE COUNT       ; COUNT ← COUNT + 1

        LOAD COUNT
        SUB  TWO56       ; A ← COUNT - 256
        JZ   DONE        ; if COUNT == 256, stop
        JMP  LOOP

DONE:   JMP  DONE        ; halt by spinning

; ---- data ----
COUNT:  .DATA 0
ZERO:   .DATA 0
ONE:    .DATA 1
TWO56:  .DATA 256        ; 0x00000100
```

6)
Machine language (32-bit words)

Format: word = (opcode << 28) | x. Shown as 8 hex digits. The program starts at address 0x00000000.
Labels are resolved to the addresses in the left column.

Address	Assembly	Machine word
0x00000000	LOAD ZERO	0x0000000D
0x00000001	STORE COUNT	0x1000000C
0x00000002	LOAD COUNT	0x0000000C
0x00000003	WRITE 0x00000008	0x30000008
0x00000004	LOAD COUNT	0x0000000C
0x00000005	ADD ONE	0x4000000E
0x00000006	STORE COUNT	0x1000000C
0x00000007	LOAD COUNT	0x0000000C
0x00000008	SUB TWO56	0x5000000F
0x00000009	JZ DONE	0xD000000B
0x0000000A	JMP LOOP	0xC0000002
0x0000000B	DONE: JMP DONE	0xC000000B
0x0000000C	COUNT: .DATA 0	0x00000000
0x0000000D	ZERO: .DATA 0	0x00000000
0x0000000E	ONE: .DATA 1	0x00000001
0x0000000F	TWO56: .DATA 256	0x00000100

7)

```
; ===== Stanley/Penguin — GCD via repeated subtraction =====
; Reads two integers from P[0x00000100] and writes gcd to P[0x00000200]
; ISA:
; 0 LOAD   A := M[x]
; 1 STORE  M[x] := A
; 2 READ   A := P[x]
; 3 WRITE  P[x] := A
; 4 ADD    A := A + M[x]
; 5 SUB    A := A - M[x]
; C JMP    IP := x
; D JZ     if A = 0 then IP := x

      READ  0x00000100        ; A <- first input
      STORE A
      READ  0x00000100        ; A <- second input
      STORE B

      LOAD  A                 ; if A==0 -> gcd=B
      JZ    OUT_B
      LOAD  B                 ; if B==0 -> gcd=A
      JZ    OUT_A

GCD_LOOP:
      LOAD  A                 ; if A==B -> done
      SUB   B
      JZ    DONE

      ; Compare A vs B without sign flags:
      ; Copy to X,Y and count both down to find the smaller.
      LOAD  A
      STORE X
      LOAD  B
      STORE Y

CMP_LOOP:
      LOAD  X
      JZ    A_LT_B            ; A < B
      LOAD  Y
      JZ    B_LT_A            ; B < A
      LOAD  X
      SUB   ONE
      STORE X
      LOAD  Y
      SUB   ONE
      STORE Y
      JMP   CMP_LOOP
```

```
A_LT_B:                 ; B := B - A
    LOAD  B
    SUB   A
    STORE B
    LOAD  B
    JZ    OUT_A         ; if B==0 -> gcd=A
    JMP   GCD_LOOP

B_LT_A:                 ; A := A - B
    LOAD  A
    SUB   B
    STORE A
    LOAD  A
    JZ    OUT_B         ; if A==0 -> gcd=B
    JMP   GCD_LOOP

DONE:
    LOAD  A             ; A == B is the gcd
    WRITE 0x00000200
    JMP   HALT

OUT_A:
    LOAD  A
    WRITE 0x00000200
    JMP   HALT

OUT_B:
    LOAD  B
    WRITE 0x00000200

HALT:  JMP   HALT         ; halt by spinning

; ---- data ----
A:     .DATA 0
B:     .DATA 0
X:     .DATA 0
Y:     .DATA 0
ONE:   .DATA 1
```

8)

```
; Swap ACC <-> M[0x000030AA] (Stanley/Penguin ISA)

    STORE TMP_ACC       ; save old ACC
    LOAD  0x000030AA    ; ACC ← old M[0x000030AA]
    STORE TMP_MEM       ; save old MEM
    LOAD  TMP_ACC       ; ACC ← old ACC
```

```
        STORE 0x000030AA      ; M[0x000030AA] ← old ACC
        LOAD  TMP_MEM         ; ACC ← old MEM  (done)

; ---- data ----
TMP_ACC: .DATA 0
TMP_MEM: .DATA 0
```

9)

```
; Jump to 0x0837BBE1 if ACC >= 0
; Opcodes:
; 0 LOAD   A := M[x]
; C JMP    IP := x
; D JZ     if A = 0 then IP := x
; E JN     if A < 0 then IP := x   ; (jump if negative)

        JN   SKIP           ; if ACC < 0, skip the jump
        JMP  0x0837BBE1     ; if ACC >= 0, jump there
SKIP:
        ; execution continues here if ACC < 0
```

10)

Part 1:

After executing the sequence, the values in r8 and r9 are swapped. This happens because XOR is its own inverse and allows the two registers to exchange values without using a temporary register.

Part 2:

It happens because XOR is reversible — each operation cancels the previous one, resulting in the two registers swapping values.