Computer Design

Computer Science (2nd year B.Sc.)

Programming under hardware constraints

Seminar #3

1. Pre-Check

This section is designed as a check to help you determine whether you understand the concepts covered in class. Please answer "true/false" to the following questions and include an explanation.

1.1. Assembly language is portable and can be used across different processor architectures.
FALSE! Assembly language is architecture-specific and cannot be directly used across different processors without modification.

1.2. The Assembler tool converts each assembly instruction into exactly one machine code.
FALSE! While this is usually the case, some assembly instructions may be expanded into multiple machine code instructions, depending on the architecture.

1.3. In the decode phase, an instruction is converted from machine code into assembly language.
FALSE! In the decode phase, the instruction is interpreted by the control unit and the necessary signals are generated to execute it. It is not translated back into assembly language.

1.4. The compiler can check both syntax and logical errors
FALSE! The compiler can check for syntax errors (such as missing semicolons or incorrect structure) but cannot detect reasoning errors (i.e. flawed logic in the program). Logical errors occur when the program runs but produces unexpected results.

1.5. The "Instruction Set Architecture" (ISA) defines both the hardware and the software of a computer system.
FALSE! The ISA primarily defines the interface between software and hardware. That is, the assembly instruction (software), its associated machine code (hardware), and what it should do (task). The ISA does not tell how the hardware should be implemented.

2. Roll back the clock!

The Intel 4004, released in 1971, was the world's first commercially available microprocessor. It was a 4-bit processor designed for calculators, featuring a modest instruction set and limited capabilities. In fact, the Intel 4004 lacked basic instructions for doing logical operations like AND, OR, and XOR, which are now common it todays CPUs. These logical operations had to be implemented using subroutines, making even simple bitwise operations a challenging task in both code size and execution time.

2.1. Given two numbers A and B, write below the body of a C function that returns the bitwise AND operation between A and B without using any of the bitwise operators "&, | , ^, ~". **Hint**: Use a combination of "shifts" and arithmetic operations.

```
uint32_t AND(uint32_t A, uint32_t B) {
   uint32_t pow2 = 1, result = 0;
   for(int i=0 ; i < 32 ; i++) {
       int bitA = A % 2, bitB = B % 2;
       result += bitA * bitB * pow2;
       A /= 2;
       B /= 2;
       pow2 *= 2;
   }
   return result;
}
```

2.2. Given two numbers A and B, write below the body of a C function that returns the bitwise XOR operation between A and B without using any of the bitwise operators "&, | , ^, ~".

```
uint32_t XOR(uint32_t A, uint32_t B) {
   uint32_t pow2 = 1, result = 0;
   for(int i=0 ; i < 32 ; i++) {
       int bitA = A % 2, bitB = B % 2;
       if(bitA != bitB)
          result += pow2;
       A /= 2;
       B /= 2;
       pow2 *= 2;
   }
   return result;
}
```

3. Roll back the clock even further in time!

Fortran 1 (1957) is the first high level language designed to accomplish scientific and engineering calculations. It had Input/Output instructions, DO loops, GOTO and IF statements. There were no functions though or structured programming in general. Here is an example of code (given in the original Fortran manual):

```
C      PROGRAM FOR FINDING THE LARGEST VALUE
C    X    ATTAINED BY A SET OF NUMBERS
      DIMENSION A(999)
      READ 1 N, (A(I), I = 1,N)
   1  FORMAT (I3/(12F6.2))
      BIGA = A(1)
      DO 20 I = 2 , N
      IF (BIGA - A(I)) 10 , 20 , 20
  10  BIGA = A(I)
  20  CONTINUE
      PRINT 2 N, BIGA
   2  FORMAT (21HTHE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)
      STOP 0
```

A program written in an unstructured way uses jump instructions (like the `goto` statement in C) to labels or to instruction “numbers”. The lines of this program are usually numbered (as above) or may have “labels” (as in C) which allows the flow of execution to branch to any line of the program. This is in contrast to “structured programming” which uses block constructs (i.e. if/then/else statements) and loops (i.e. while, for, …).

3.1. Consider the C code below. Write an equivalent implementation in unstructured form. That is without using the block constructs (i.e. if/then/else) or loops (i.e. while, for, …)

Structured form	Unstructured form
`// num is an 8-bit unsigned integer`	`// num is an 8-bit unsigned integer`
`  int count=0;`	`  int count=0;`
`  for (int i=0; i<8; ++i) {`	`  int i=0;`
`     if( num & ( 1 << i ) )`	` WL:`
`        count++;`	`  if(i>=8) goto NEXT;`
`  }`	`  if( !( num & ( 1 << i ) ) ) goto ITER;`
	`  count++;`
	` ITER:`
	`  i++;`
	`  goto WL;`
	` NEXT:`

3.2. Using only “goto” and if statements, write below the body of a C function that computes the factorial of a number.

```
int factorial(int n) {
   int fact = 1;
   int i = 1;
LOOP:
   fact *= i;
   i++;
   if(i <= n) goto LOOP;
   return fact;
}
```

4. Fetch – Decode – Execute Cycle

To which execution phase do the steps below belong to: **F**etch, **D**ecode or **E**xecute?

- The instruction is read from memory and placed in the “instruction register”. [✓ F] / D / E
- The “control unit” interprets the instruction and prepares the necessary signals. F / [✓ D] / E
- The result of an arithmetic operation is written back into a register. F / D / [✓ E]
- The “program counter” is incremented to point to the next instruction. [✓ F] / D / E
- The operands for the instruction are retrieved from registers or memory. F / D / [✓ E]
- The “control unit” sends signals to the “arithmetic and logical unit” to perform an operation. F / D / [✓ E]
- The “control unit” checks the instruction to determine what actions to be taken. F / [✓ D] / E