# CSCI-UA.0201 – 007 Computer Systems Organization
## FALL 2022

# PROJECT 3
# WRITTEN ASSIGNMENT

## Due date: 11.11.2022, 11.59pm
### Late submissions due date: 11.14.2022, 11.59pm

---

**This is an individual work. No team work is allowed.**
**Similarity check will be applied to submitted codes.**

**QUESTION 1 - x86 assembly to C pseudo code**

According to the given assembly code on the left, write the updated values of ecx register.

| | |
|---|---|
| movl %eax, %ecx | ecx = eax |
| movl $0x13, %ecx | ecx = 0x13 |
| movl 0x224, %ecx | ecx = M[0x224] |
| movl(%ebx), %ecx | ecx = M[%ebx] |
| movl 2(%ebx), %ecx | ecx = M[%ebx+2] |

**QUESTION 2 - x86 assembly to C code**

Carefully read the given assembly code. Note that variables x, y and z are initially stored in %rdi, %rsi, and %rdx, respectively. Now, fill in the blanks in the given c code.

```
fun:                        x    y     t1
      leaq (%rdi,%rsi), %rax               t1 = x+y    } x+y+z
      addq %rdx, %rax                      t2 = t1+z   }
            z
      cmpq $5, %rdi
      jge .L2      x>=5
      cmpq %rsi, %rdx
      jge .L3      z>=y
      movq %rdi, %rax       t5 = x
      imulq %rsi, %rax      t6 = y*x.
      ret           y
.L3:
      movq %rsi, %rax       t7 = y
      imulq %rdx, %rax      val = z*y
      ret           z
.L2
      cmpq $4, %rdi
      jle .L4      x<=4
      movq %rdi, %rax       t3 = x
      imulq %rdx, %rax      t4 = z*t3 = z*x
.L4                 z
      rep; ret
```

x>=5

↓ L2   if x<5

```
long fun (long x, long y, long z)
{
      long val ---------- = x+y+z
      if(---x<5-----)
      {
            if(---z<y------)
                  val = ---x*y-------
            else
                  val = --y*z-------        L3
      }
      else if(---x>4------)
            val = ---x*z------

      return val;
}
```

**QUESTION 3 - C code to x86 assembly**

For the given for loop, fill in the blanks in the respective assembly code. Note that i is saved in edx and total is saved in ecx registers respectively.

```
        edx  ecx
for(i=0; i < total; i++)
      ct +=i;


xorl --%edx--, --%edx--- #here you need to initialize i to 0
cmpl --%ecx---, --%edx--- # now it is time to compare i with total
jge .L4                #jump to end if necessary
movl ct, --%eax---- #cache value of ct in eax register

.L6:

addl --%edx---, --%eax---- #ct +=i is here
incl --%edx---- # i++
cmpl %ecx, %edx
jl .L6
movl %eax, -----ct----  #store the value of ct back in memory

.L4:
```

**QUESTION 4 - C code to x86 assembly**

Check the given fun.s assembly file and try to interpret its purpose. Note that if you try to run this file, it may not run properly due to your system architecture. Therefore, try to guess its purpose.

**QUESTION 5 – Simple coding**

In this question, you will implement recursive Fibonacci. Save your code in Fibonacci.c file. Then, create your corresponding assembly file Fibonacci.s by following the instructions we covered in the classroom. Depending on your system architecture, it may give different results. Write your guess how it should look like in x86 assembly and now compare it with your fibonacci.s file result. Write down the differences.

**How to submit:** All questions, except the 5th one, are written questions and can be submitted in a single doc or pdf file. You can even scan and submit your hand-written notes in a single pdf file. For question 5, you need to submit your .c and .s files and an explanation. Please include your written explanation in the previous file of the first 4 solutions. Additionally, do not forget to submit your code files. Therefore, you will submit one .zip file which includes: 1 doc/pdf file, 1 fibonacci.c file and 1 fibonacci.s file.

# 4. fun.s

L1: if (x == 0)
goto L2
y = x * y
x--

```
.global main
main:
    movl $5 %eax          t1 = 5
    movl $1 %ebx          t2 = 1

recursive L1: cmpl $0, %eax       } jump if : t1 == 0
    je L2   jump if ZF=1          }
    imul %eax %ebx        t2 = t1 * t2
    decl %eax             t1 --
    jmp L1   uncondition jump back (loop)

                                       esp+4 = ecx
base L2: movl %ebx , 4(%esp)     store M[%esp+4] to ebx
    movl $.LC0 , (%esp)           push argument to printf,
         constant string          so that %esp points to a
    call printf                   word that contains argument
    movl $0 , %eax                t1 = 0
leave
ret        return
```

fun.s is a recursive programe that calculates the factorial, starting with x=5, y=1.

- Each time we get the product, we decrement x, so we have 5x4 → 3x(5x4) → 2x(3x5x4) → 1x(2x3x4x5).
- L2 is the base case, which is reached when x is decremented to 0, then the stack is push back up until the top.
- C code: (approximate)

```
main:
  long x = 5
  long y = 1
L1: while(x != 0)                L2:
  {                                  int* n = y.
      y = x*y;                       printf (n)
      x --;                          x=0.

  }
```

# 5. Assembly:

```
pushl  %rbp              } Callee reserved;  rbp = n
movl   %rsp  %rbp        } rbp, rsp point to same address
pushl  %rsi       } callers reserved,
pushl  %rdi       } rsi, rdi : arguments to hold value
cmpl   $1 , 8(%rbp)   - compare 1 with [%rbp +8],
                        pushing the stack one down.
jg   .L2     n>2 → goto L2.
movl   8(%rbp) , %rax     return value → pushed rbp.
jmp . L1     unconditioned jump go to L1

.L2:

movl  8(%rbp) , %rax    set return val to n(original rbp)
decl  %rax       return val - 1
pushl  %rax      push return value
call  fib        call function
movl  %rax, %rsi      use rsi to store fib(n-1)
movl  8(%rbp), %rax   re set return value to n
subl   $2 , %rax      return val -2 .

pushl  %rax      push return val
call   fib       call function
movl   %rax, %rdi      use rdi to store fib(n-2)
leal  (%rdi , %rsi) , %rax    return value
jmp. L1    goto L1            = rdi + rsi
                              = fib(n-2) + fib(n-1)
```

.L1:

```
leal -8(%rbp), %rsp
popl %rdi
popl %rsi

ret
```

pop rsp to rbp-8 : to bottom

rdi, rsi not used. so pop them.

return

C:

```
fib(n)
    if (n <= 1)
        return n;

    return fib(n-1) + fib(n-2)
```

There are lots of differences between the generated main.s and my guessed x86 assembly code, basically because of the version difference of assembly and processors. My system is IOS and my chip is M1, and people with different ones might get different .s files. For example, I encountered a few commands with .cfi_def_cfa and .cfi_offset, which is not covered in class, and I believe that it is more like commands that is within the system. I did some research and found that it is shorthanded for Call Frame Information, which is similar to a loose function. The %rsp seemed to be represented by [sp], and there seemed to be some difference with the processed bytes. Some commands seemed similar, like we have add, mov, cmp, etc. However, the types are discarded compared to my version of x86 assembly where we always have addl, movl, and cmpl. There are also some lines with str and ldp that I have not seen. After some research, they all seem to load multiple instances for the program to store.