## Problem 1 (Instruction Analysis)

max.s

```
        .section __TEXT,__text,regular,pure_instructions
        .build_version macos, 11, 0       sdk_version 11, 3
        .globl   _max1                          ## -- Begin function max1
        .p2align 4, 0x90
_max1:                                  ## @max1
        .cfi_startproc
## %bb.0:
        pushq    %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq     %rsp, %rbp
        .cfi_def_cfa_register %rbp
        movl     %edi, -4(%rbp)
        movl     %esi, -8(%rbp)
        movl     -4(%rbp), %eax
        cmpl     -8(%rbp), %eax
        jle      LBB0_2
## %bb.1:
        movl     -4(%rbp), %eax
        movl     %eax, -12(%rbp)                ## 4-byte Spill
        jmp      LBB0_3
LBB0_2:
        movl     -8(%rbp), %eax
        movl     %eax, -12(%rbp)                ## 4-byte Spill
LBB0_3:
        movl     -12(%rbp), %eax                ## 4-byte Reload
        popq     %rbp
        retq
        .cfi_endproc
                                        ## -- End function
        .globl   _max2                          ## -- Begin function max2
        .p2align 4, 0x90
_max2:                                  ## @max2
        .cfi_startproc
## %bb.0:
        pushq    %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq     %rsp, %rbp
        .cfi_def_cfa_register %rbp
        movl     %edi, -4(%rbp)
        movl     %esi, -8(%rbp)
        movl     -4(%rbp), %eax
        cmpl     -8(%rbp), %eax
        setg     %cl
        andb     $1, %cl
        movzbl   %cl, %eax
        movl     %eax, -12(%rbp)
        cmpl     $0, -12(%rbp)
        je       LBB1_2
## %bb.1:
```

```
52          movl    -4(%rbp), %eax
53          movl    %eax, -16(%rbp)
54          jmp     LBB1_3
55 LBB1_2:
56          movl    -8(%rbp), %eax
57          movl    %eax, -16(%rbp)
58 LBB1_3:
59          movl    -16(%rbp), %eax
60          popq    %rbp
61          retq
62          .cfi_endproc
63                                          ## -- End function
64 .subsections_via_symbols
```

1) What does the code hint about the kind of instruction set? (e.g. Accumulator, Register Memory, Memory Memory, Register Register) Please justify your answer.

**Ans:** Instuction Set Architecture เป็นแบบ Register-Memory Architecture เนื่องจากคำสั่งที่มีการใช้ ALU Operation ต่าง ๆ ในไฟล์ Assembly (max.s) จะรับ operand ด้วยกัน 2 ตัว โดยตัวแรกเป็นค่าที่มาจากใน register และอีกตัวหนึ่งเป็นค่าที่มาจาก Memory เช่น บรรทัดที่ 16 คำสั่ง `cmpl -8(%rbp), %eax` จะเป็นการ compare ระหว่างค่าที่เก็บใน Memory ที่ Address %rbp-8 (mem[%rbp-8]) กับค่าที่เก็บใน Register eax

2) Can you tell whether the architecture is either Restricted Alignment or Unrestricted Alignment? Please explain how you come up with your answer.

**Ans:** ทำการทดลองเขียนโปรแกรมภาษา c ง่าย ๆ ขึ้นมาดังนี้ เพื่อทดสอบ

*unrestricted_vs_restricted.c*

```c
void test() {
        int a = 1;
        char b = 'x';
        char c = 'y';
        int d = 2;
}
```

ทำการเปลี่ยนเป็น Assembly ได้เป็น ได้ดังนี้

*unrestricted_vs_restricted.s*

```
        .section __TEXT,__text,regular,pure_instructions
        .build_version macos, 11, 0       sdk_version 11, 3
        .globl  _test                           ## -- Begin function test
        .p2align 4, 0x90
_test:                                  ## @test
        .cfi_startproc
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
```

```
11          movq    %rsp, %rbp
12          .cfi_def_cfa_register %rbp
13          movl    $1, -4(%rbp)
14          movb    $120, -5(%rbp)
15          movb    $121, -6(%rbp)
16          movl    $2, -12(%rbp)
17          popq    %rbp
18          retq
19          .cfi_endproc
20                                      ## -- End function
21  .subsections_via_symbols
```

จากบรรทัดที่ 13-16 ใน *unrestricted_vs_restricted.s* จะเห็นได้ว่า ตัวแปร int a ที่จะใช้ 4 byte ในการเก็บจะเก็บที่ Mem address ตั้งแต่ %rbp-4 จนถึง %rbp-1 ส่วนต่อมาคือ char b ซึ่งใช้เนื้อที่ 1 byte จะเก็บที่ address %rbp-5 และ char c ซึ่งใช้เนื้อที่ 1 byte เช่นเดียวกัน ก็จะเก็บที่ address %rbp-6 พอมาถึงการเก็บตัวแปร int d ซึ่งใช้ 4 byte จะไปเก็บอยู่ที่ address %rbp-12 จนถึง %rbp-9

จะสังเกตได้ว่าการเก็บตัวแปร int d ซึ่งใช้เนื้อที่ 4 byte ใน memory จะไม่ได้เก็บต่อเนื่อง คือจะข้าม address ช่อง %rbp-7 และ %rbp-8 ไป ดังนั้นถือว่าเป็นการเก็บค่าแบบ **Restricted Alignment** เพราะการเลือกค่า address ที่จะวางจะต้องเลือกวาง ณ ตำแหน่งหมายเลข address ที่ mod กับขนาดของค่าที่จะเก็บนั้นจะต้องเท่ากับ 0

3. Create a new function (e.g. testMax) to call max1. Generate new assembly code. What does the result suggest regarding the register saving (caller save vs callee save)? Please provide your analysis.

**Ans:** เป็นแบบ **Callee Save** โดยสังเกตเห็นได้ว่าเมื่อมีการทำงานฟังก์ชัน max1 จะมีการ pushq %rbp ไปเก็บไว้ใน stack ก่อน (บรรทัดที่ 8 ใน max.s) ซึ่งค่าใน Register rbp นั้นจะใช้ในการทำ addressing mode ไปยังตำแหน่งต่าง ๆ บน memory ซึ่งได้ทำการ push ไปเก็บไว้ก่อนการทำงานอื่น ๆ ในฟังก์ชันจะเริ่มต้นขึ้น หลังจากนั้นเมื่อการทำงานของฟังก์ชันจบลงจะมีการ popq %rbp (บรรทัดที่ 27 ใน max.s) เพื่อนำค่าเดิมที่เคยเก็บไว้กลับคืน เพื่อที่จะได้พร้อมกลับไปทำงานต่อในฟังก์ชันที่เรียกเข้ามา

testMax.c

```c
#include "max.c"

int main() {
        int result = max1(2,3);
}
```

testMax.s

```
1  _main:                                  main## @main
2          .cfi_startproc
3  ## %bb.0:
4          pushq   %rbp
5          .cfi_def_cfa_offset 16
6          .cfi_offset %rbp, -16
7          movq    %rsp, %rbp
8          .cfi_def_cfa_register %rbp
9          subq    $16, %rsp
10         movl    $2, %edi
11         movl    $3, %esi
12         callq   _max1
13         xorl    %ecx, %ecx
14         movl    %eax, -4(%rbp)
15         movl    %ecx, %eax
16         addq    $16, %rsp
17         popq    %rbp
18         retq
19         .cfi_endproc
20                                         ## -- End function
21 .subsections_via_symbols
```

4. How do the arguments be passed and the return value returned from a function? Please explain the code.

**Ans:** ในการส่ง aregument ไปยัง function max1 จะมีการบันทึกค่าของ argument ตัวแรก (a) ไว้ที่ Register edi และบันทึกค่าของ argument ตัวที่สอง (b) ไว้ที่ Register esi (บรรทัดที่ 10-11 ใน testMax.s) ซึ่งเมื่อ function max1 ทำงานก็จะหยิบค่าจาก Register edi และ esi ไปใช้เป็นตัวแปร a และ b ตามลำดับ

ส่วนของการส่งค่า return value จากใน function max1 เมื่อทำงานเสร็จจะมีการเก็บค่า return value ไว้ใน Register eax (บรรทัดที่ 23 หรือ 26 ใน max.s)

5. Find the part of code (snippet) that does comparison and conditional branch. Explain how it works.

**Ans:**

**function max1**

```
1          .section __TEXT,__text,regular,pure_instructions
2          .build_version macos, 11, 0       sdk_version 11, 3
3          .globl   _max1                            ## -- Begin function max1
4          .p2align 4, 0x90
5  _max1:                                  ## @max1
6          .cfi_startproc
7  ## %bb.0:
8          pushq   %rbp
9          .cfi_def_cfa_offset 16
10         .cfi_offset %rbp, -16
11         movq    %rsp, %rbp
```

```
12          .cfi_def_cfa_register %rbp
13          movl    %edi, -4(%rbp)
14          movl    %esi, -8(%rbp)
15          movl    -4(%rbp), %eax
16          cmpl    -8(%rbp), %eax
17          jle     LBB0_2
18  ## %bb.1:
19          movl    -4(%rbp), %eax
20          movl    %eax, -12(%rbp)                 ## 4-byte Spill
21          jmp     LBB0_3
22  LBB0_2:
23          movl    -8(%rbp), %eax
24          movl    %eax, -12(%rbp)                 ## 4-byte Spill
25  LBB0_3:
26          movl    -12(%rbp), %eax                 ## 4-byte Reload
27          popq    %rbp
28          retq
29          .cfi_endproc
30                                          ## -- End function
```

ค่าของตัวแปร int a จะเก็บอยู่ที่ Mem[%rbp-4] และค่าของตัวแปร int b จะเก็บอยู่ที่ Mem[%rbp-8]

1. ทำการย้ายค่าตัวแปร a ที่เก็บใน Mem[%rbp-4] ไปไว้ที่ Register eax

2. เรียกใช้คำสั่ง cmpl ในการเปรียบเทียบค่าระหว่าง Mem[%rbp-8] (ค่าของ b) และ Register eax (ค่าของ a) โดยการเปรียบเทียบค่าระหว่าง a กับ b

   2.1 ถ้าค่าของ a น้อยกว่าหรือเท่ากับ b (b > a) ก็จะทำการ jump ไปที่ LBB0_2 เพื่อทำการนำค่าที่เก็บใน b ไปใส่ไว้ที่ Register eax ที่จะทำหน้าที่เป็นตัวเก็บ return value ของ function max1 แล้วจึงทำ LBB0_3 ต่อ

   2.2 ถ้าค่าของ a มากกว่า b (a > b) ก็จะทำส่วนของ LBB0_2 คือนำค่าของ a ไปเก็บใน Register eax ที่ทำหน้าที่เป็นตัวเก็บ return value ของ function max1 แล้วจึงทำการ jump ไปทำส่วน LBB0_3

3. ส่วนสุดท้าย (LBB0_3) ก็จะเป็นการ restore ค่า rbp เดิมออกมาจาก stack แล้วจึงจบการทำงานของ function

**function max2**

```
31          .globl  _max2                           ## -- Begin function max2
32          .p2align 4, 0x90
33  _max2:                                  ## @max2
34          .cfi_startproc
35  ## %bb.0:
36          pushq   %rbp
37          .cfi_def_cfa_offset 16
38          .cfi_offset %rbp, -16
39          movq    %rsp, %rbp
40          .cfi_def_cfa_register %rbp
41          movl    %edi, -4(%rbp)
42          movl    %esi, -8(%rbp)
43          movl    -4(%rbp), %eax
44          cmpl    -8(%rbp), %eax
45          setg    %cl
46          andb    $1, %cl
```

```
47          movzbl  %cl, %eax
48          movl    %eax, -12(%rbp)
49          cmpl    $0, -12(%rbp)
50          je      LBB1_2
51  ## %bb.1:
52          movl    -4(%rbp), %eax
53          movl    %eax, -16(%rbp)
54          jmp     LBB1_3
55  LBB1_2:
56          movl    -8(%rbp), %eax
57          movl    %eax, -16(%rbp)
58  LBB1_3:
59          movl    -16(%rbp), %eax
60          popq    %rbp
61          retq
62          .cfi_endproc
63                                          ## -- End function
64  .subsections_via_symbols
```

ค่าของตัวแปร int a จะเก็บอยู่ที่ Mem[%rbp-4] และค่าของตัวแปร int b จะเก็บอยู่ที่ Mem[%rbp-8]

1. ทำการย้ายค่าตัวแปร a ที่เก็บใน Mem[%rbp-4] ไปไว้ที่ Register eax

2. เรียกใช้คำสั่ง cmpl ในการเปรียบเทียบค่าระหว่าง Mem[%rbp-8] (ค่าของ b) และ Register eax (ค่าของ a) โดยการเปรียบเทียบค่าระหว่าง a กับ b

3. เมื่อเปรียบเทียบเสร็จหาก a > b จะทำให้ได้ค่า logic 1 แต่ถ้าหาก b >= a จะทำให้ได้ค่า logic 0 มาเก็บไว้ที่ Register cl แล้วจึงนำไปเก็บไว้ใน Mem[%rbp-12]

4. เปรียบเทียบค่าระหว่าง 0 กับค่า logic ที่เกิดจากการเปรียบเทียบค่าระหว่าง a กับ b โดยใช้คำสั่ง cmpl $0, -12(%rbp)

4.1 หากเปรียบเทียบแล้วเท่ากัน (Mem[%rbp-12] = logic 0) นั่นคือ a <= b จะมีการ jump ไปทำงานในส่วนของ LBB1_2 ที่เป็นการนำค่าตัวแปร b จาก Mem[%rbp-8] ไปเก็บไว้ที่ตัวแปร max ซึ่งจะเป็น return value ของ function นี้ที Mem[%rbp-16] แล้วจึงไปทำงานในส่วนของ LBB1_3 ต่อไป

4.2 หากเปรียบเทียบแล้วไม่เท่ากัน (Mem[%rbp-12] = logic 0) นั่นคือ a > b จะทำงานต่อทันทีโดยไม่ jump คือการนำค่าตัวแปร a จาก Mem[%rbp-4] ไปเก็บไว้ที่ตัวแปร max ซึ่งจะเป็น return value ของ function นี้ที Mem[%rbp-16] แล้วจึงมีการ jump ไปทำงานในส่วนของ LBB1_3 ต่อไป

5. ส่วนสุดท้ายของฟังก์ชัน (LBB1_3) ก็จะเป็นการนำค่าที่จะ return ซึ่งเก็บไว้ที่ Mem[%rbp-16] ไปเก็บไว้ที่ Register eax ซึ่งเป็น register ที่ทำหน้าที่เก็บ return value เมื่อจบทำการของฟังก์ชัน จากนั้นก็จะมีการ restore ค่า rbp ออกมาจาก stack ที่ได้ push ไว้ตั้งแต่ตอนต้น แล้วจึงจบการทำงานของฟังก์ชัน

6. If max.c is compiled with optimization turned on (using “gcc -O2 -S max.c”), what are the differences that you may observe from the result (as compare to that without optimization). Please provide youir analysis

max.s (using “gcc -O2 -S max.c”)

```
        .section __TEXT,__text,regular,pure_instructions
        .build_version macos, 11, 0       sdk_version 11, 3
        .globl  _max1                           ## -- Begin function max1
        .p2align 4, 0x90
_max1:                                  ## @max1
        .cfi_startproc
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        movl    %esi, %eax
        cmpl    %esi, %edi
        cmovgel %edi, %eax
        popq    %rbp
        retq
        .cfi_endproc
                                        ## -- End function
        .globl  _max2                           ## -- Begin function max2
        .p2align 4, 0x90
_max2:                                  ## @max2
        .cfi_startproc
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        movl    %esi, %eax
        cmpl    %esi, %edi
        cmovgel %edi, %eax
        popq    %rbp
        retq
        .cfi_endproc
                                        ## -- End function
.subsections_via_symbols
```

**Ans:** สิ่งที่สังเกตได้ชัดเจนมากที่สุดคือเมื่อ function max1 และ max2 ถูก compile แบบ optimize แล้ว เนื่องจากทั้งสองฟังก์ชันรับ input เหมือนกัน (รับ int a, b) และมีการส่งค่า output เหมือนกัน (คืนค่าที่มากกว่า) ทำให้เมื่อถูก compile แบบ optimize แล้ว machine code ที่ได้ของทั้งสองฟังก์ชันเหมือนกันทุกประการ รวมถึงขั้นตอนการทำงานลดน้อยลง รวมถึงการทำงานไม่มีการยุ่งเกี่ยวกับ memory เลย ซึ่งจะส่งผลให้การทำงานนั้นเร็วมากยิ่งขึ้น (ISA แบบ optimize แล้วเป็นแบบ Register-Register Architecture)

7. Please estimate the CPU Time Required by the max1 function (using the equation CPI = IC x CPI x Tc). If possible, create a main function to call max1 and use the time command to measure the performance. Compare the measure to your estimation. What do you think are the factors that cause the difference? Please provide your analysis.

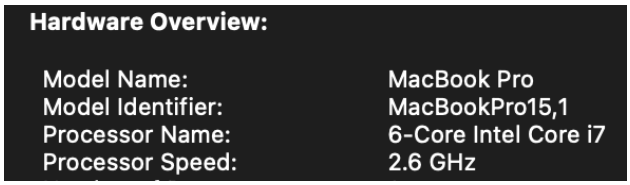
**Ans:**

กำหนดให้คำสั่งต่าง ๆ มีค่า CPI ดังต่อไปนี้ (พิจารณา code ที่ optimize แล้ว)

| Instruction | CPI |
|---|---|
| pushq | 1 |
| popq | 1 |
| movq | 1 |
| movl | 1 |
| cmpl | 1 |
| cmovgel | 1 |
| retq | 1 |

ดังนั้นจะได้ว่า Total CPI = 1*pushq + 1*movq + 1*movl + 1*cmpl + 1* cmovgel + 1*retq + 1*popq = 7

My CPU Rate (6-Core Intel Core i7) = 2.6 GHz




(ทำการทดลองโดยการ run function max1 จำนวน $10^9$ ครั้ง

$$\therefore CPU\ Time = IC \times CPI \times T_{clock}$$
$$= 10^9 \times 7 \times \frac{1}{2.6 \times 10^9}$$
$$= 2.69\ s$$

CPU Time ที่ได้จากการใช้ time command จับเวลา (เฉลี่ย 10 ครั้ง) = 2.05 s

ค่าที่ได้จากการคำนวณ กับค่าที่ได้จากการ run แล้วจับเวลาโดยใช้ time command อาจต่างกันเพราะความคลาดเคลื่อนในการประมาณ CPI ของแต่ละ instruction ย่อย และในทางปฏิบัติจริงอาจมีการ optimize อื่น ๆ เพิ่มเติมนอกจากการ optimize ตัว complier ทำให้การเวลาในการทำงานจริงนั้นเร็วกว่าเวลาที่คำนวณได้

Code for time command

```c
#include "max.c"
#include <stdio.h>
#include <time.h>

int main() {
        int times = 1e9;
        int N = 10;
        double sum_time_used = 0;
        int result;

        for (int i = 0; i < N; i++) {
                clock_t begin = clock();
                for (int j = 0; j < times; j++) {
                        result = max1(2,3);
                }
                clock_t end = clock();

                double time_used = (double)(end-begin)/CLOCKS_PER_SEC;
                sum_time_used += time_used;
        }

        printf("Average Time used: %lf\n", (double) sum_time_used/N);
        return 0;
}
```

---

## Problem 2 (Optimization)

Please measure the execution time (using the command) of this given program when compiling with optimization level 0 (no optimization), level 1, level 2 and level 3.

### Execution Time (s)

| Optimization Level | Level 0 | Level 1 | Level 2 | Level 3 |
|---|---|---|---|---|
| Round #1 | 10.5252 | 8.3078 | 6.7702 | 6.7446 |
| Round #2 | 10.4737 | 8.6236 | 6.8702 | 6.6771 |
| Round #3 | 10.5264 | 8.8793 | 6.7537 | 6.7802 |
| Round #4 | 10.4265 | 8.7683 | 6.7782 | 6.7312 |
| Round #5 | 10.4567 | 8.8562 | 6.7434 | 6.7556 |
| Average | **10.4817** | **8.6871** | **6.7832** | **6.7377** |

fibo.c

```c
#include <stdio.h>
#include <time.h>

long fibo(long a) {
        if (a <= 0L) {
                return 0L;
        }
        if (a == 1L) {
                return 1L;
        }
        return fibo(a-1L)+fibo(a-2L);
}

int main (int argc, char *argv[]) {
        int N = 5;
        double sum_time_used = 0;

        for (int r = 1; r <= N; r++) {
                printf("\n----------Round #%i----------\n", r);

                clock_t begin = clock();
                for (long i = 1L; i < 45L; i++) {
                        long f = fibo(i);
                        printf("fibo of %ld is %ld\n", i, f);
                }
                clock_t end = clock();

                double time_used = (double)(end-begin)/CLOCKS_PER_SEC;
                printf("\n====> Time used: %lf s \n", time_used);
                sum_time_used += time_used;
        }

        printf("\n\n<=== Average Time used: %lf s! ===>\n\n", (double)
sum_time_used/N);
```

---

## Problem 3 (Analysis)

As suggested by the results in Exercise 2, what kind of optimization are used by the compiler in each level in order to make the program faster?

**Ans:**

**Optimization Level 0:**

ลักษณะของ code ใน level 0 จะมีการแบ่ง conditional jump ในลักษณะคล้ายกับ source code ที่เขียนใน .c คือ แบ่งออกเป็นกรณี a = 0, a = 1 และค่า a อื่น ๆ ซึ่งในกรณีที่ a = 0 ก็จะมีการคืนค่า 0 ออกไป ส่วนกรณี a = 1 ก็จะคืนค่า 1 ออกไป แต่กรณี a อื่น ๆ จะต้องมีการเรียก fibo อีก 2 ครั้ง คือ fibo(a-1) และ fibo(a-2) สังเกตว่าสำหรับค่า a ที่

มากขึ้น จำนวนครั้งในการ call function fibo จะมากขึ้นตามโดยแปรผันกับค่า a แบบ exponential ทำให้การทำงานยังค่อนข้างช้า เมื่อเทียบกับ code ที่ถูก optimize โดย compiler

**Optimization Level 1:**

มีการรวบเงื่อนไข a = 0 และ a = 1 เข้าด้วยกัน เพื่อลดจำนวนครั้งในการ jump

**Optimization Level 2:**

พยายามลดการ call function fibo ในกรณี a > 1 อีก เช่น กรณีที่มีการเรียก fibo(x-1) + fibo(x-2) ถ้า x-2 == 0 หรือ 1 ก็จะไม่ต้องทำการเรียก fibo อีกรอบ

**Optimization Level 3:**

Function fibo จะเหมือนกับ level 2 เพียงแต่มีการเปลี่ยนการทำงานของ loop ให้เป็นการทำงานแบบ Sequence คือเรียก fibo(1) ไปจนถึง fibo(44) ในส่วนของ function main เพื่อลดการ jump (แต่ไม่ได้ช่วยร่นระยะเวลาในการทำงานได้มากนัก สังเกตว่า Execution Time ของ level 3 จะใกล้เคียงกับ level 2)

---

**Related Code**

https://github.com/jamestenni/comp-sys-arch-assignment1