Worralop Srichainont (6632200221)

Activity 01: Instruction Set and Compiler

1. (Instruction Analysis) This exercise will familiarize you with several aspects of the instruction set and the fundamentals of the compiler. Given max.c (below), please use gcc -S max.c to compile the code into assembly code. (The result will be in max.s) From the result, answer the following questions.

```c
#include<stdio.h>
int max1(int a, int b) {
    return (a > b) ? a : b;
}

int max2(int a, int b) {
    int isaGTb = a > b;
    int max;
    if(isaGTb)
        max = a;
    else
        max = b;
    return max;
}
```

ผลจากการใช้งานคำสั่ง เพื่อ compile code เป็น assembly code (max.s) ได้ออกมาเป็นดังนี้

```
    .file   "max.c"
    .text
    .globl  max1
    .type   max1, @function
max1:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -8(%rbp), %edx
    movl    -4(%rbp), %eax
    cmpl    %eax, %edx
    cmovge  %edx, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

```
.LFE0:
    .size   max1, .-max1
    .globl  max2
    .type   max2, @function
max2:
.LFB1:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    -20(%rbp), %eax
    cmpl    -24(%rbp), %eax
    setg    %al
    movzbl  %al, %eax
    movl    %eax, -4(%rbp)
    cmpl    $0, -4(%rbp)
    je  .L4
    movl    -20(%rbp), %eax
    movl    %eax, -8(%rbp)
    jmp .L5
.L4:
    movl    -24(%rbp), %eax
    movl    %eax, -8(%rbp)
.L5:
    movl    -8(%rbp), %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE1:
    .size   max2, .-max2
    .ident  "GCC: (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long   1f - 0f
    .long   4f - 1f
    .long   5
```

```
0:
    .string "GNU"
1:
    .align 8
    .long   0xc0000002
    .long   3f - 2f
2:
    .long   0x3
3:
    .align 8
4:
```

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

เมื่อพิจารณา assembly code จะเห็นได้ว่า มีการย้ายข้อมูลระหว่าง Register และ Memory เช่น

```
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    -20(%rbp), %eax
    cmpl    -24(%rbp), %eax
```

นอกจากนี้ยังมีการใช้ stack ซึ่งเป็น Memory เพื่อ push และ pop Register อีกด้วย

```
    pushq  %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -8(%rbp), %edx
    movl    -4(%rbp), %eax
    cmpl    %eax, %edx
    cmovge  %edx, %eax
    popq    %rbp
```

จากที่กล่าวไปข้างต้น จึงสรุปได้ว่าลักษณะของ Instruction Set ของ max.c เป็น Register Memory

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

พิจารณา assembly code ของฟังก์ชัน **max1**

```
max1:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -8(%rbp), %edx
    movl    -4(%rbp), %eax
    cmpl    %eax, %edx
    cmovge  %edx, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

เมื่อพิจารณาเลขหน้าคำสั่ง **%rbp** จะพบว่าเลขทั้งหมดเป็น offset ที่หารด้วย 4 ลงตัวทั้งหมด หมายความว่า code จะทำงานโดยเริ่มต้นที่ register ตามตำแหน่งของ แล้วบวกหรือลบไปเป็นจำนวนตัวเลขด้านหน้า นอกจากนี้ยังพบว่ามีการ allocate stack จำนวน 16 bytes อีกด้วย จากข้อมูลข้างต้นจึงสามารถสรุปได้ว่าเป็น Restricted Alignment

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.

จากโจทย์ เราได้สร้างฟังก์ชัน testMax ที่เรียกใช้งานฟังก์ชัน max1 ดังนี้

```
    int testMax(int a, int b) {

    return max1(a,b);
}
```

และใช้งานคำสั่ง gcc -S max.c จะได้ code assembly ในส่วนของฟังก์ชัน testMax ออกมาเป็นดังนี้

```
testMax:
.LFB2:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $8, %rsp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -8(%rbp), %edx
    movl    -4(%rbp), %eax
    movl    %edx, %esi
    movl    %eax, %edi
    call    max1
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE2:
    .size   testMax, .-testMax
    .ident  "GCC: (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long   1f - 0f
    .long   4f - 1f
    .long   5
```

และเมื่อพิจารณา assembly code ในส่วนของฟังก์ชัน **testMax** จะทราบได้ว่า
คำสั่ง **%eax %edx %edi** และ **%esi** ส่ง argument เข้า function ไม่ได้ถูกบันทึกค่าใน **testMax** ก่อนเรียก max1 ดังนั้น 4 ตัวนี้เป็น caller-save registers
แต่ **%rbp** ถูก push (บันทึก) และ pop (เรียกคืน) แสดงว่าเป็น callee-save registers

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

Argument **a** ส่งเข้า register **%edi** และ argument **b** ส่งเข้า **%esi** จากนั้นเอาทั้ง 2 ค่าเข้า stack frame แล้วเรียกใช้งาน **max1** และเอาผลลัพธ์ใส่เข้าไปใน register **%eax**

Find the code (snipplet) that does comparison and conditional branch. Explain how it works.

พิจารณาฟังก์ชัน max1

```
    movl    -8(%rbp), %edx
    movl    -4(%rbp), %eax
    cmpl    %eax, %edx
    cmovge  %edx, %eax
```

- คำสั่ง **cmpl** เป็นการเปรียบเทียบค่าระหว่าง **%eax (a)** และ **%edx (b)**
- คำสั่ง **cmovge** ถ้าหากเข้าเงื่อนไข a >= b จะย้ายค่าใน **%edx** เข้าไปยัง **%eax** แต่ถ้าหากไม่เข้าเงื่อนไข ค่าของ **%eax** จะยังคงเป็นค่าเดิม

พิจารณาฟังก์ชัน max2

```
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    -20(%rbp), %eax
    cmpl    -24(%rbp), %eax
    setg    %al
    movzbl  %al, %eax
    movl    %eax, -4(%rbp)
    cmpl    $0, -4(%rbp)
    je  .L4
    movl    -20(%rbp), %eax
    movl    %eax, -8(%rbp)
    jmp .L5
.L4:
    movl    -24(%rbp), %eax
    movl    %eax, -8(%rbp)
.L5:
    movl    -8(%rbp), %eax
```

- คำสั่ง **cmpl** เป็นการเปรียบเทียบค่าระหว่าง **%eax (a)** และ **%-24(%rbp) (b)**
- คำสั่ง **setg** ถ้าหากเข้าเงื่อนไข a > b จะให้ค่า **%al** = 1 แต่ถ้าไม่เข้าเงื่อนไข จะให้ค่า **%al** = 0
- คำสั่ง **cmovge** ถ้าหากเข้าเงื่อนไข a >= b จะย้ายค่าใน **%edx** เข้าไปยัง **%eax** แต่ถ้าหากไม่เข้าเงื่อนไข ค่าของ **%eax** จะยังคงเป็นค่าเดิม
- คำสั่ง **cmpl** เป็นการเปรียบเทียบค่าระหว่าง **0** กับ **%-4(%rbp)**
- คำสั่ง je .L4 คือการกระโดดไปยัง L4 ก็ต่อเมื่อผลลัพธ์การเปรียบเทียบเท่า 0 แต่ถ้าไม่เป็น 0 จะทำการเก็บค่า a เป็นค่า maximum
- ถ้าหาก a > b จะเก็บค่า a ลงใน **%-8(%rbp)** และกระโดดไปยัง L5 เพื่อ return ค่า
- ถ้าหาก a < b จะเก็บค่า b ลงใน **%-8(%rbp)**

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 compared to that without optimization). Please provide your analysis

ผลจากการใช้งานคำสั่ง เพื่อ compile code โดยมีการ optimization เป็น assembly code (max.s) ได้ออกมาเป็นดังนี้

```
    .file   "max.c"
    .text
    .p2align 4
    .globl  max1
    .type   max1, @function
max1:
.LFB23:
    .cfi_startproc
    endbr64
    cmpl    %edi, %esi
    movl    %edi, %eax
    cmovge  %esi, %eax
    ret
    .cfi_endproc
.LFE23:
    .size   max1, .-max1
    .p2align 4
    .globl  max2
    .type   max2, @function
max2:
.LFB24:
    .cfi_startproc
    endbr64
    cmpl    %esi, %edi
    movl    %esi, %eax
    cmovge  %edi, %eax
    ret
    .cfi_endproc
.LFE24:
    .size   max2, .-max2
    .ident  "GCC: (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long   1f - 0f
    .long   4f - 1f
    .long   5
```

```
0:
    .string "GNU"
1:
    .align 8
    .long   0xc0000002
    .long   3f - 2f
2:
    .long   0x3
3:
    .align 8
4:
```

ความแตกต่างที่พบเมื่อเปรียบเทียบกับ assembly code ที่ไม่ผ่านการ optimization

- ไม่มีการเก็บค่า base pointer และ stack pointer เพื่อลด overhead จาก function calls
- ใช้งาน register โดยตรง (ไม่มีการใช้งาน Memory) จึงลด number of instructions ลงไปได้บางส่วน
  ใช้งาน cgmove แทนการใช้งาน braching

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.
(You may find references online regarding the CPI of each instruction.)

ในข้อนี้ได้เขียนฟังก์ชัน main() ซึ่งทำงานโดยการเรียกใช้งานฟังก์ชัน max1 จำนวน 100 ล้านครั้ง

```c
#include<stdio.h>
int max1(int a, int b) {
    return (a > b) ? a : b;
}

int main() {
    int a = 1, b = 2;
    int result;
    for(int i = 0; i < 100000000; i++) {
        result = max1(a,b);
    }
    printf("Result: %d\n", result);
    return 0;
}
```

พิจารณา assembly code ของฟังก์ชัน **max1**

```
max1:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -8(%rbp), %edx
    movl    -4(%rbp), %eax
    cmpl    %eax, %edx
    cmovge  %edx, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
```

นับจำนวน Instruction (IC) และ Clock per Instruction (CPI)

อ้างอิงจาก https://agner.org/optimize/instruction_tables.pdf

Assembly Command	IC	CPI
`pushq %rbp`	1	1
`movq %rsp, %rbp`	1	0.5
`movl %edi, -4(%rbp)`	1	0.5
`movl %esi, -8(%rbp)`	1	0.5
`movl -8(%rbp), %edx`	1	0.5
`movl -4(%rbp), %eax`	1	0.5
`cmpl %eax, %edx`	1	0.5
`cmovge %edx, %eax`	1	0.5
`popq %rbp`	1	1
`ret`	1	2
IC (Total) / CPI (Average)	**10**	**0.75**

CPU ที่ใช้งานมีความถี่ 3.60 GHz

$$T_c = \frac{1}{3.60 \times 10^9\ Hz} = 3.6 \times 10^{-9}\ s$$

จากข้อมูลข้างต้น จึงสามารถคำนวณ CPU Time ได้ดังนี้

$$CPU = IC \times CPI \times T_c$$

$$CPU = 10 \times 0.75 \times (3.6 \times 10^{-9}\ s)$$

$$CPU = 2.07 \times 10^{-9}\ s$$

จากฟังก์ชัน main ที่เรียกใช้งานฟังก์ชัน max1 จำนวน 100,000,000 ครั้ง
ดังนั้นในทางทฤษฎี ฟังก์ชัน main ควรจะทำงานเสร็จภายในเวลา 0.207 วินาที

```
reisenx@reisenx-VirtualBox:~/Documents/GitHub/2110352-COM-SYS-ARCH/Activity01/program$ gcc -S main.c
reisenx@reisenx-VirtualBox:~/Documents/GitHub/2110352-COM-SYS-ARCH/Activity01/program$ gcc -o main main.c
reisenx@reisenx-VirtualBox:~/Documents/GitHub/2110352-COM-SYS-ARCH/Activity01/program$ time ./main
Result: 2

real    0m0.271s
user    0m0.264s
sys     0m0.002s
reisenx@reisenx-VirtualBox:~/Documents/GitHub/2110352-COM-SYS-ARCH/Activity01/program$
```

แต่พบว่าในทางปฏิบัติ ฟังก์ชัน main ทำงานเสร็จในเวลา 0.271 วินาที ซึ่งช้ากว่าที่คำนวณไว้ สาเหตุอาจจะเป็นเพราะว่า overhead จาก loop และ function call ไม่ได้ถูกพิจารณาในสูตรข้างต้น

2. (Optimization) We will use simple Fibonacci calculation as a benchmark. Please measure the execution time (using the time command) of this given program when compiling with optimization level 0 (no optimization), level 1, level 2 and level 3. (Note that some compilers do similar optimization for all level 1, level 2 and level 3. If that is the case, you will see no difference after 1.) You may want to run each program a few times and use the average value as a result

```c
#include<stdio.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[]) {
    for(long i = 1L; i < 45L; i++) {
        long f = fibo(i);
        printf("fibo of %ld is %ld\n", i, f);
    }
}
```

Level 0 (No optimization)

ผลของการรันคำสั่ง time จำนวน 3 ครั้ง จากการ compile ด้วย gcc -O0 -o fibo fibo.c

เวลาเฉลี่ย: 18.975 วินาที

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m18.542s
user    0m18.502s
sys     0m0.009s
```

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m19.638s
user    0m19.501s
sys     0m0.037s
```

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m18.744s
user    0m18.675s
sys     0m0.036s
```

Level 1 optimization

ผลของการรันคำสั่ง time จำนวน 3 ครั้ง จากการ compile ด้วย `gcc -O1 -o fibo_o1 fibo.c`

เวลาเฉลี่ย: 11.565 วินาที

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m10.702s
user    0m10.678s
sys     0m0.012s
```

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m11.773s
user    0m11.711s
sys     0m0.019s
```

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m12.221s
user    0m12.169s
sys     0m0.020s
```

Level 2 optimization

ผลของการรันคำสั่ง time จำนวน 3 ครั้ง จากการ compile ด้วย `gcc -O2 -o fibo_o2 fibo.c`

เวลาเฉลี่ย: 3.699 วินาที

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m3.761s
user    0m3.733s
sys     0m0.010s
```

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m3.686s
user    0m3.606s
sys     0m0.015s
```

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m3.651s
user    0m3.626s
sys     0m0.014s
```

Level 3 optimization

ผลของการรันคำสั่ง time จำนวน 3 ครั้ง จากการ compile ด้วย `gcc -O3 -o fibo_o3 fibo.c`

เวลาเฉลี่ย: 3.112 วินาที

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m3.096s
user    0m3.084s
sys     0m0.005s
```

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m3.091s
user    0m3.050s
sys     0m0.011s
```

```
fibo of 38 is 39088169
fibo of 39 is 63245986
fibo of 40 is 102334155
fibo of 41 is 165580141
fibo of 42 is 267914296
fibo of 43 is 433494437
fibo of 44 is 701408733

real    0m3.149s
user    0m3.116s
sys     0m0.011s
```

3. (Analysis) As suggested by the result in Exercise 2, what kinds of optimization are used by the compiler in each level in order to make the program faster? To answer this question, use gcc -S -O2 fibo.c and use this result as a basis for your analysis

Level 0 (No optimization)

- เขียน Assembly ทำงานเหมือนกับที่เขียนในภาษา C ทุกประการ

Level 1 optimization

- ลดความซ้ำในการใช้งาน stack operation
- Recursion ดีขึ้นโดยการใช้ leal

Level 2 optimization

- ใช้งาน register มากขึ้นเพื่อลดปริมาณ register ใน stack
- มีการใช้งาน memory ที่ดีขึ้น

Level 3 optimization

- มีการพยายาม optimize ให้ใช้เวลารันไวขึ้น โดยแลกกับการใช้งาน memory ที่มากขึ้น
- Stack มีความซับซ้อนมากขึ้นในแต่ละ register