Exercises

1. (Instruction Analysis) This exercise will familiarize you with several aspects of 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.

   1.1 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. Instruction set นี้ เป็นแบบ Register Memory เพราะเมื่อดูจาก max.s จะพบว่าในขั้นตอนการเปรียบเทียบ (cmpl) เป็นการเปรียบเทียบระหว่าง %eax ที่เป็นค่าของ register กับ 24(%rbp) ที่เป็นการดึงค่ามาจาก memory[rbp+24] จึงเป็นแบบ Register Memory และเมื่อเขียน function ดูการลบก็จะชัดเจนขึ้นว่าเป็น operation ระหว่างข้อมูลที่เรียกจาก register กับ memory

Code:

```
cmpl    %eax, 24(%rbp)
```

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

Ans. ยังไม่สามารถบอกได้ว่า Architecture นี้ เป็นแบบ Restricted Alignment เพราะข้อมูลที่มีอยู่มีเพียง int เท่านั้น แม้เมื่อดูจากการเก็บบน function max1 ที่เก็บ int ตัวแรกไว้ที่ rbp+16 (displacement) และ ตำแหน่งต่อไปที่เก็บข้อมูลคือ rbp+24 ซึ่ง int มีขนาด 4 bytes และ 24 mod 4 = 0 แต่ยังไม่สามารถบอกได้ เราสามารถระบุให้ชัดเจนขึ้นด้วยการสร้าง function restricted ที่มีข้อมูลประเภทอื่น ๆด้วย เช่น char, bool, double, short มาด้วย แล้วดูตำแหน่งการ access register จะสามารถบอกได้ชัดเจนว่าเป็นแบบ Restricted Alignment โดยเมื่อดูจาก function restricted() เมื่อ compile ออกมาแล้วพบว่ามีชี้การเก็บ char ไปที่ตำแหน่ง mem[rbp-5] และ mem[rbp-6] และชี้ตำแหน่งเก็บ int ถัดไปที่ตำแหน่ง mem[rbp-12] (ช่องที่ mem[rbp-9], mem[rbp-10], mem[rbp-11], mem[rbp-12] เพราะ int มีขนาด 4 bytes) ซึ่งจะข้ามช่องที่ mem[rbp-7] กับ mem[rbp-8] ไป เพื่อให้ตรงตามเงื่อนไขของ Restricted Alignment (-12 mod 4 = 0)

Code:

```
void restricted()
{
    int a = 10;
    char b = 'a';
    char c = 'c';
    int d = 8;
}
```

Compile:

```
restricted:
            pushq    %rbp
```

```
        .seh_pushreg    %rbp
        movq    %rsp, %rbp
        .seh_setframe   %rbp, 0
        subq    $16, %rsp
        .seh_stackalloc 16
        .seh_endprologue
        movl    $10, -4(%rbp)
        movb    $97, -5(%rbp)
        movb    $99, -6(%rbp)
        movl    $8, -12(%rbp)
        nop
        addq    $16, %rsp
        popq    %rbp
        ret
        .seh_endproc
        .def    __main; .scl    2;      .type   32;     .endef
        .section .rdata,"dr"
```

1.3 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. การ save register rbp เป็นแบบ Callee save เพราะว่ามีการ push และ pop ค่าออกมาเพื่อต้องการที่จะใช้งานในระยะยาวและ เป็นค่าที่ return จาก stack frame นี้ ส่วน register ตัวอื่นๆนั้นไม่ถูก pop ค่าออกมาเลย จึงเป็นแบบ Caller save สามารถพิจารณาเพิ่มเติมได้จาก function testMax() พบว่าเมื่อเรียกใช้ max1 2 ครั้ง พบว่า มีเพียง register rbp ที่มีการถูก pop ออกมาเมื่อเรียกใช้ function จบ

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

Ans. จาก max.s การรับส่งค่าบน register เป็นแบบ Stack frame กล่าวคือเปรียบเสมือน register เป็น stack ซ้อนกันอยู่ใน memory เวลาเรียกใช้จะไล่ลงไปเรื่อย ๆตาม stack ส่วนการ return ค่าจาก function นั้น จะคืนผ่าน register rbp ที่เป็น base pointer ที่มี eax เป็นองค์ประกอบย่อย

Code:

```
int testMax(int a, int b, int c)
{
    int i = max1(a, b);
    int j = max1(a, c);
    return i-j;
}
```

Compile:

```
testMax:
        pushq   %rbp
        .seh_pushreg    %rbp
        movq    %rsp, %rbp
        .seh_setframe   %rbp, 0
        subq    $48, %rsp
        .seh_stackalloc 48
        .seh_endprologue
        movl    %ecx, 16(%rbp)
        movl    %edx, 24(%rbp)
        movl    %r8d, 32(%rbp)
        movl    24(%rbp), %eax
        movl    %eax, %edx
        movl    16(%rbp), %ecx
        call    max1
        movl    %eax, -4(%rbp)
        movl    32(%rbp), %eax
        movl    %eax, %edx
        movl    16(%rbp), %ecx
        call    max1
        movl    %eax, -8(%rbp)
        movl    -4(%rbp), %eax
        subl    -8(%rbp), %eax
        addq    $48, %rsp
        popq    %rbp
        ret
        .seh_endproc
        .globl  sub
        .def    sub;    .scl    2;      .type   32;     .endef
        .seh_proc       sub
```

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

Ans. – Function max1 มีหลักการทำงานของ Comparison and conditional branch คือระบุตำแหน่งตัวเลข 2 ตัวที่ต้องการเปรียบเทียบบน rbp ด้วยการไล่ตาม memory แบบ displacement จนถึงตำแหน่งที่ mem[rbp+16] (a) กับ mem[rbp+24] (b) ให้ eax เก็บค่าบน mem[rbp+16] แล้วใช้คำสั่ง cmpl เปรียบเทียบค่าบน eax กับ ค่าบน mem[rbp+24] ถ้าค่าบน mem[rbp+24] >=eax ให้ใส่ค่าของ mem[rbp+24] ลงบน eax แล้วคืนตัว rbp ไป

- Function max2 มีหลักการทำงานของ Comparison and conditional branch คือระบุตำแหน่งตัวเลข 2 ตัวที่ต้องการเปรียบเทียบบน rbp ด้วยการไล่ตาม memory แบบ displacement จนถึงตำแหน่งที่ mem[rbp+16] (a) กับ mem[rbp+24] (b) ให้ eax เก็บค่า mem[rbp+16] แล้วใช้คำสั่ง cmpl ถามว่าค่าบน eax >= mem[rbp+24] หรือไม่ ให้เก็บค่า Boolean ไว้ที่ al แล้วส่งต่อไปที่ eax และ mem[rbp-8] จากนั้นเรียก cmpl เพื่อเทียบว่าค่า 0 กับค่าบน mem[rbp-8] ถ้าค่าเท่ากัน (mem[rbp-8]=0 or eax<mem[rbp+24]) ให้ jump ไปที่ L4 ซึ่งจะใส่ค่าบน mem[rbp+24] ให้กับ eax แล้วไปทำคำสั่งที่ L5 ต่อเลย ถ้าค่าไม่เท่ากัน (eax>=mem[rbp+24]) ให้ใส่ค่าบน mem[rbp+16] ไปที่ eax แล้ว jump ไปที่ L5 แล้วกลับไปทำตามคำสั่งเดิม

Code:

```
max1:
        pushq   %rbp
        .seh_pushreg    %rbp
        movq    %rsp, %rbp
        .seh_setframe   %rbp, 0
        .seh_endprologue
        movl    %ecx, 16(%rbp)
        movl    %edx, 24(%rbp)
        movl    16(%rbp), %eax
        cmpl    %eax, 24(%rbp)
        cmovge  24(%rbp), %eax
        popq    %rbp
        ret
        .seh_endproc
        .globl  max2
        .def    max2;   .scl    2;      .type   32;     .endef
        .seh_proc       max2
max2:
        pushq   %rbp
        .seh_pushreg    %rbp
        movq    %rsp, %rbp
        .seh_setframe   %rbp, 0
        subq    $16, %rsp
        .seh_stackalloc 16
        .seh_endprologue
        movl    %ecx, 16(%rbp)
        movl    %edx, 24(%rbp)
        movl    16(%rbp), %eax
        cmpl    24(%rbp), %eax
        setg    %al
        movzbl  %al, %eax
```

```
        movl    %eax, -8(%rbp)
        cmpl    $0, -8(%rbp)
        je      .L4
        movl    16(%rbp), %eax
        movl    %eax, -4(%rbp)
        jmp     .L5
.L4:
        movl    24(%rbp), %eax
        movl    %eax, -4(%rbp)
.L5:
        movl    -4(%rbp), %eax
        addq    $16, %rsp
        popq    %rbp
        ret
        .seh_endproc
        .globl  testMax
        .def    testMax; .scl   2;      .type   32;     .endef
        .seh_proc       testMax
```

1.6 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 compared to that without optimization). Please provide your analysis

Ans. การทำงานของ function max1 และ function max2 แทบจะไม่แตกต่างแล้วมีเพียงการสลับที่ของ register บางช่องเท่านั้น และไม่มีการ push หรือ pop ค่า rbp แล้ว จึงทำให้การ save rbp กลายเป็นแบบ Caller save เพื่อใช้งานในระยะสั้นแทน

Code:

```
max1:
        .seh_endprologue
        cmpl    %ecx, %edx
        movl    %ecx, %eax
        cmovge  %edx, %eax
        ret
        .seh_endproc
        .p2align 4
        .globl  max2
        .def    max2;   .scl    2;      .type   32;     .endef
        .seh_proc       max2
```

```
max2:
        .seh_endprologue
        cmpl    %edx, %ecx
        movl    %edx, %eax
        cmovge  %ecx, %eax
        ret
        .seh_endproc
        .p2align 4
        .globl  testMax
        .def    testMax; .scl   2;      .type   32;     .endef
        .seh_proc       testMax
```

1.7 Please estimate the CPU Time required by the max1 function (using the equation CPI=ICxCPIxTc). 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.)

Ans. จากการดู max.s พบว่า function max1 มีการใช้ movq 1 ครั้ง, movl 3 ครั้ง, cmovge 1 ครั้ง, pushq 1 ครั้ง, popq 1 ครั้ง, cmpl 1 ครั้ง ซึ่งจากการไปดูจาก Reference พบว่าเมื่อใช้ Instruction Table ของ Skylake (For Intel 6th gen core) แต่ละคำสั่งจะใช้ CPI = 1 cycle/instruction ยกเว้น popq cmovge กับ cmpl ที่ใช้ CPI = 0.5 cycle/instruction ดังนั้น CPI ทั้งหมด = 1*1+1*3+0.5*1+1*1+0.5*1+0.5*1 = 6.5 Cycle per instruction และ Clock Frequency = 3.40 GHz ซึ่งสามารถคำนวณหา CPU Time ได้จาก

$$\text{CPU Time} = \text{Instruction Count} * \text{CPI(avg)} * \text{Tc}$$

$$\text{CPU Time} = 600^3 * 6.5 * \frac{1}{3.4 * 10^9} = 0.413 \text{ s.}$$

เมื่อเปรียบเทียบจากการ Run code จะพบว่าได้ CPU Time average $= \frac{0.493+0.504+0.497+0.501+0.502}{5} = 0.499$ s. ซึ่งถือว่าคลาดเคลื่อนไป 17.24% ซึ่งอาจเกิดจากการ Run ในชีวิตจริงอาจมี clock per instruction ที่แตกต่างจากในทฤษฎีไปบ้างและขึ้นกับการทำงานของคอมพิวเตอร์ในขณะที่ run ด้วยว่าทำงานได้ประสิทธิภาพเท่าไหร่

Code:

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

```
    int a = 3;
    int b = 5;
    int c;
    int time = pow(600, 3);
    clock_t start = clock();
    for (int i = 0; i < time; i++)
    {
        c = max1(a, b);
    }
    clock_t stop = clock();
    double s = (double)(stop - start) / CLOCKS_PER_SEC;
    printf("Total CPU time: %lf", s);

}
```

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

Ans. จากการพิจารณา Execution time จะพบว่าที่ Optimization level ต่าง ๆ จะได้ผลลัพธ์ดังตารางด้านล่าง

Optimization Level	ครั้งที่ 1 (s.)	ครั้งที่ 2 (s.)	ครั้งที่ 3 (s.)	ครั้งที่ 4 (s.)	ครั้งที่ 5 (s.)	ค่าเฉลี่ย (s.)
Level 0	12.493	12.596	12.684	12.587	12.768	12.626
Level 1	11.122	11.131	11.160	11.219	11.177	11.162
Level 2	8.526	8.301	8.358	8.292	8.403	8.376
Level 3	8.241	8.476	8.104	8.350	8.386	8.311

Code:

```
#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[])
{
    clock_t start = clock();
    for (long i = 1L; i < 45L; i++)
    {
        long f = fibo(i);
        printf("fibo of %ld is %ld\n", i, f);
    }
    clock_t stop = clock();
    double s = (double)(stop - start) / CLOCKS_PER_SEC;
    printf("Total CPU Time: %lf", s);

}
```

3. As suggested by the results 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” to generate the assembly code for each level (e.g. “gcc -S -O2 fibo.c”) and use this result as a basis for your analysis. (Depending on your version of the compiler, the result may vary.)

Ans.

- ที่ Optimization Level 1 จะมีลดจำนวน jump ลงจาก 4 ครั้ง เป็น 3 ครั้ง โดยการนำเงื่อนไขที่ค่าที่รับมาต้องมากกว่า 1 ไปใส่ไว้ตั้งแต่แรก
- ที่ Optimization Level 2 จะเพิ่มจำนวนการ Jump ขึ้นแต่จะจัดลำดับการ Jump มีประสิทธิภาพมากขึ้น และพยายามลดจำนวนครั้งในการเรียก function ลง
- ที่ Optimization Level 3 จะมีจำนวนการ Jump ขึ้นอีกแต่แลกกับการที่จะมีเงื่อนไขเพิ่มคือ ให้ A เป็น Big Number โดยถ้า ผลลัพธ์ที่คำนวณได้มีค่าไม่เกิน A ก็ไม่ต้องเรียก function ซ้ำ และมีการตั้งชื่อ register ใหม่บางส่วน
