Kittipat Jusingha 6531304621

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.

Register-Register architecture

the code frequently moves values between registers (movl %ecx, 16(%rbp)), compares values in registers (cmpl %eax, %edx), and conditionally moves data between registers (cmovge %edx, %eax). This pattern is characteristic of a Register-Register architecture

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

Unrestricted Alignment.

The assembly code does not show any special instructions for handling misaligned memory access.

1.3.What does the result suggest regarding the register saving (caller save vs. callee save)? Please provide your analysis.

Callee-Save %rbp

after call function there are always pushq %rbp ,then popq %rbp before return.

Kittipat Jusingha 6531304621

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

passed:

The first argument to max1 is passed in the %ecx register.

The second argument to max1 is passed in the %edx register.

```
movl $2, %edx   ; Second argument is 2, stored in %edx

movl $1, %ecx   ; First argument is 1, stored in %ecx

call max1       ; Call the function max1 with arguments in %ecx and %edx
```

return:

passed back to the caller in the %rax register

```
cmpl %eax, %edx  ; Compare the values in %eax (first argument) and %edx (second argument)

cmovge %edx, %eax; If %edx >= %eax, move %edx into %eax (this handles the return value)

ret             ; Return from the function, with the result in %eax
```

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

```
cmpl %eax, %edx  ; Compare the values in %eax (first argument) and %edx (second argument)

cmovge %edx, %eax; If %edx >= %eax, move %edx into %eax (this handles the return value)
```

Kittipat Jusingha 6531304621

1.6.what are the differences that you may observe from the result (as compared to that without optimization). Please provide your analysis

- it remove

```
 pushq   %rbp
.seh_pushreg      %rbp
movq     %rsp, %rbp
.seh_setframe     %rbp, 0
```

- it remove redundant move command.

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.

```
max1:
   cmpl    %ecx, %edx    ; 1 instruction
   movl    %ecx, %eax    ; 1 instruction
   cmovge  %edx, %eax    ; 1 instruction
   ret              ; 1 instruction
```

Total IC = 4 instructions

Average CPI = 4/4 = 1 cycle/instruction

Tc= 1 / 2.3 × 10^9 = 0.4348 ns

CPU Time = 4 instructions × 1 cycle/instruction × 0.333 ns/cycle = 1.739 ns

Kittipat Jusingha 6531304621

```
Days              : 0

Hours             : 0

Minutes           : 0

Seconds           : 1

Milliseconds      : 412

Ticks             : 14123589

TotalDays         : 1.63467465277778E-05

TotalHours        : 0.000392321916666667

TotalMinutes      : 0.023539315

TotalSeconds      : 1.4123589

TotalMilliseconds : 1412.3589
```

1.412 ns

Modern CPUs can execute multiple instructions simultaneously due to features like pipelining so it faster

Kittipat Jusingha 6531304621

2. Measure Execution Time Using PowerShell:

Measurement	**O0**	**O1**	**O2**	**O3**
Days	0	0	0	0
Hours	0	0	0	0
Minutes	0	0	0	0
Seconds	10	9	3	2
Milliseconds	374	395	577	40
Ticks	103748073	93959893	35778975	20403954
TotalDays	0.000120078788194444	0.000108749876157407	4.14108506944444E-05	2.36156875E-05
TotalHours	0.00288189091666667	0.00260999702777778	0.000993860416666667	0.0005667765
TotalMinutes	0.172913455	0.156599821666667	0.059631625	0.03400659
TotalSeconds	10.3748073	9.3959893	3.5778975	2.0403954
TotalMilliseconds	10374.8073	9395.9893	3577.8975	2040.3954

Kittipat Jusingha 6531304621

3.(Analysis) 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.

O1:

- Uses leal (load effective address) to compute arguments for recursive calls

- Procedure Inlining

- Common Expression Elimination and Constant Propagation

- Stack Height Reduction

- Copy Propagation and Code Motion

O2:

- O1

- Code Motion

- Strength Reduction

- Branch Offset Optimization and Pipeline Scheduling

O3:

- O2

- Loop Unrolling

- Vectorization

- Advanced Dead Code Elimination

For the assembly code you can go see at https://github.com/CUknot/ComputerSystemArchitecture.git