- 1.1) Register-Memory because cmpl compare between register and memory. cmpl
- cmpl %eax, 24(%rbp)
- 1.2) Restricted Alignment because int is 4 bytes but to allocate double that is 8 bytes, it allocates at 24 instead of 20 mean that it's Restricted Alignment.

```
movl %ecx, 16(%rbp)
movsd %xmm1, 24(%rbp)
```

1.3) Callee saves as this picture.

```
pushq
       %rbp
               %rbp
.seh_pushreg
      %rsp, %rbp
movq
.seh_setframe %rbp, 0
.seh_endprologue
mov1
       %ecx, 16(%rbp)
movsd %xmm1, 24(%rbp)
cvtsi2sd
         16(%rbp), %xmm0
comisd 24(%rbp), %xmm0
jbe .L7
cvtsi2sd
           16(%rbp), %xmm0
cvttsd2si
jmp .L5
movsd 24(%rbp), %xmm0
cvttsd2si %xmm0, %eax
       %rbp
popq
ret
.seh endproc
.globl max2
.def
       max2:
               .scl
                      2; .type 32; .endef
```

1.4) Pass by register where %ecx and %edx for function argument and %eax for return value.

```
pushq
      %rbp
.seh_pushreg
               %rbp
       %rsp, %rbp
mova
.seh_setframe
               %rbp, 0
.seh_endprologue
       %ecx, 16(%rbp)
mov1
mov1
       %edx, 24(%rbp)
       16(%rbp), %eax
movl
       %eax, 24(%rbp)
cmp1
cmovge 24(%rbp), %eax
popq
ret
.seh_endproc
.globl max2
.def
      max2;
              .scl 2; .type 32; .endef
.seh_proc max2
```

max1: 1.5) %rbp pushq %rbp .seh pushreg %rsp, %rbp %rbp, 0 .seh_setframe .seh endprologue %ecx, 16(%rbp) mov1 %edx, 24(%rbp) movl mov1 16(%rbp), %eax %eax, 24(%rbp) cmp1 24(%rbp), %eax cmovge %rbp popq ret

Firstly, let's return value be a. Then compare a and b if b is higher than a then write return value to be b.

```
pusha
       %rbp
.seh pushreg
               %rbp
       %rsp, %rbp
movq
.seh setframe
               %rbp, 0
      $16, %rsp
suba
.seh stackalloc 16
.seh_endprologue
        %ecx, 16(%rbp)
mov1
        %edx, 24(%rbp)
mov1
        16(%rbp), %eax
mov1
        24(%rbp), %eax
cmp1
        %al
setg
movzbl %al, %eax
mov1
       %eax, -8(%rbp)
cmp1
       $0, -8(%rbp)
je .L4
movl
       16(%rbp), %eax
movl
       %eax, -4(%rbp)
jmp .L5
mov1
        24(%rbp), %eax
        %eax, -4(%rbp)
        -4(%rbp), %eax
addq
        $16, %rsp
        %rbp
popq
```

First cmpl use for comparing if b < a and set %al to 1 if it's true else 0.

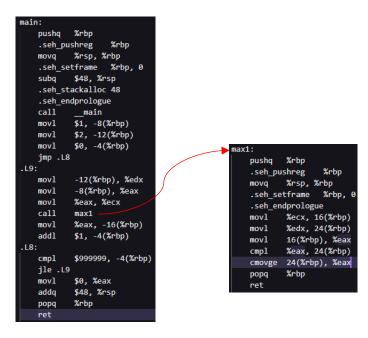
Second cmpl use to select which bracket to work. If a > b will not jump and do first bracket else will jump to L4 and do second bracket.

```
max1:
    .seh_endprologue
    cmpl %ecx, %edx
    movl %ecx, %eax
    cmovge %edx, %eax
    ret

max2:
    .seh_endprologue
    cmpl %edx, %ecx
    movl %edx, %eax
    cmovge %ecx, %eax
    ret
```

max1 and max2 are now same it just write first argument to return value then check if it must overwrite with second argument in case if b > a.

1.7) All of operation use 1 CPI except cmpl, cmovge, add and ret use 2 CPI. So, I use 12 clocks for each loop call max1 which is around 1 / 5.4G * 21 = 3.89 nanosecond but it's use 3.51 millisecond. After I make loop to call max1 10^6 times it uses 4.12 millisecond so in average it uses around 4 nanoseconds for each loop to call max1 function.



- 2) Based on average 10 time the result is:
 - Optimize 0: 6.8745 second.
 - Optimize 1: 6.9963 second.
 - Optimize 2: 3.7274 second.
 - Optimize 3: 3.7762 second.
- 3) On each optimize level:
 - Optimize 0: Normal Assembly.
 - Optimize 1: Optimize branching which can easily be seen in looping call fibo and remove many unnecessary Register-Memory commands such as movl and allocate less stack.
 - Optimize 2: Now, it's start using xor to optimize some trick like set register to 0 by xorl itself faster than movl 0 to its. And use less stack use more register which use less clock mean faster. Also remove redundant movl and change where loop call fibo is. As a result, run time is significantly reduced.
 - Optimize 3: Heavily optimize loop and logical flow which we can see that fibo part is still same but flow to loop call main fibo then print is now change to minimum redundant part that don't have to recheck.