

## inlab9.pdf

For Lab 9: Assembly, Part 2, I chose to answer the 2<sup>nd</sup> optional question for the in-lab, which said to compare code generated normally compared to optimized code. I first generated assembly code generated normally, without the `-O2` compiler flag. Then, to generate the optimized version of the assembly code, I compiled with the `-O2` compiler flag.

To explore some features of the optimized `-O2` compiler flag, I first tried a code with a simple C++ code that contained loops and function calls in main. I created a C++ code that multiplies the first parameter by the second parameter in my function `multiply` and called `multiply` in my main method. I used the intel version of the assembly code.

```
#include <iostream>
using namespace std;

int multiply(int x, int y) {
    int total;
    for (int i = 0; i < y; i++) {
        total += x;
    }
    return total;
}

int main() {
    int a = 25;
    int b = 5;
    int z = multiply(a, b);
    cout << z << endl;
    return 0;
}
```

The non-optimized version is on the left and optimized version is on the right.

```
_Z8multiplyii:                # @_Z8multiplyii
.cfi_startproc
# BB#0:
    mov     dword ptr [rsp - 4], edi
    mov     dword ptr [rsp - 8], esi
    mov     dword ptr [rsp - 12], 0
    mov     dword ptr [rsp - 16], 0

.LBB1_1:
    mov     eax, dword ptr [rsp - 16]
    cmp     eax, dword ptr [rsp - 8]
    jge     .LBB1_4
# BB#2:
    mov     eax, dword ptr [rsp - 4]
    add     eax, dword ptr [rsp - 12]
    mov     dword ptr [rsp - 12], eax
# BB#3:
    mov     eax, dword ptr [rsp - 16]
    add     eax, 1
    mov     dword ptr [rsp - 16], eax
    jmp     .LBB1_1
.LBB1_4:
    mov     eax, dword ptr [rsp - 12]
    ret
.Lfunc_end1:
    .size   _Z8multiplyii, .Lfunc_end1-_Z8multiplyii
    .cfi_endproc

    .globl  main
    .align  16, 0x90
    .type   main,@function
```

```
main:                # @main
.cfi_startproc
# BB#0:
    sub     rsp, 24

.Ltmp1:
    .cfi_def_cfa_offset 32
    mov     dword ptr [rsp + 20], 0
    mov     dword ptr [rsp + 16], 25
    mov     dword ptr [rsp + 12], 5
    mov     edi, dword ptr [rsp + 16]
    mov     esi, dword ptr [rsp + 12]
    call    _Z8multiplyii
    movabs  rdi, _ZSt4cout
    mov     dword ptr [rsp + 8], eax
    mov     esi, dword ptr [rsp + 8]
    call    _ZN5IostreamE10write<char, 1>
    movabs  rsi, _ZSt4endlC11char_traits<char>E13basic_ostreamIT_0_E6_
    mov     rdi, rax
    call    _ZN5IostreamE10write<char, 1>
    xor     ecx, ecx
    mov     qword ptr [rsp], rax    # 8-byte Spill
    add     rsp, 24
    ret
```

```
_Z8multiplyii:                # @_Z8multiplyii
.cfi_startproc
# BB#0:
    imul     edi, esi
    xor     eax, eax
    test     esi, esi
    cmovg    eax, edi
    ret
.Lfunc_end0:
    .size   _Z8multiplyii, .Lfunc_end0-_Z8multiplyii
    .cfi_endproc

    .globl  main
    .align  16, 0x90
    .type   main,@function
```

```
main:                # @main
.cfi_startproc
# BB#0:
    push     r14

.Ltmp0:
    .cfi_def_cfa_offset 16
    push     rbx

.Ltmp1:
    .cfi_def_cfa_offset 24
    push     rax

.Ltmp2:
    .cfi_def_cfa_offset 32

.Ltmp3:
    .cfi_offset rbx, -24

.Ltmp4:
    .cfi_offset r14, -16
    mov     edi, _ZSt4cout
    mov     esi, 125
    call    _ZN5IostreamE10write<char, 1>
    mov     r14, rax
    mov     rax, qword ptr [r14]
    mov     rax, qword ptr [rax - 24]
    mov     rbx, qword ptr [r14 + rax + 240]
    test    rbx, rbx
    je      .LBB1_5
```

As I generated a non-optimized version and an optimized version, I recognized that for my function multiply, which is the callee as it is called by main, the caller, I recognized the it was generally more compact and concise. I intentionally made the multiply function very inefficient. I could have not used a for loop and just multiplied the two values. When compiled without the optimized -O2 flag, it does a faithful but literal translation into assembly code, which causes longer time when it is executed. The optimized version gets rid of unnecessary code.

First, I recognized that in the callee function, multiply, there is a lot of copying of values going on in the non-optimized version. However, the callee function of the optimized version did only had one copying of value, the cmovg instruction opcode. As Professor Bloomfield mentioned in lecture about his max function, I noticed the cmovg instruction opcode generated in the optimized version, which is for “conditional move if greater than,” which will move the greater value into the first parameter. So, edi is moved into eax as edi is greater than eax. The two parameters (edi and esi) are multiplied while eax is zeroed out. Compared to the non-optimized version, there were numerous copying of values in order to finally move the final value into eax. This reduces the running time in the optimized version, as there are fewer instructions to copy values from one register to another.

Second, I recognized that there were local variables created onto the stack for copying and moving values in the non-optimized version. As there is an int total, a local variable, in my multiply function, which is totally unnecessary, if you only care about speed for running it faster. This creation of local variables and copying of values took more instructions compared to the optimized version. The optimized version just got rid of the unnecessary local variables such as int total since the optimized flag wants to reduce the running speed and wants to make it faster.

Third, as the optimized version of the assembly code does not need a local variable int total that causes a lot of copying of values and moving values in the non-optimized version, I

realized that in the optimized version, it simply just multiplies the first parameter and the second parameter with the `imul` opcode instruction, which multiplies the two parameters. It creates a more concise version and takes fewer opcodes to do the whole operation that is created in the very inefficiently in the non-optimized version: it uses a for loop to add the first parameter to the local variable, `int total`, which does the same thing as multiplying but has more instructions.

Fourth, I also recognized that there is memory access in the non-optimized version since it needs to copy and move values. And if we assume 3 cycles per memory access, there are about 33 cycles as there are 11 memory accesses in the non-optimized version ( $3 * 11$ ), which takes more time and is inefficient. For a cycle, the CPU has to stall and wait for it until it is over. However, the optimized version did not have any memory accesses, which reduces the time. The optimized version only takes 4 or 5 cycles since it doesn't have memory accesses.

In conclusion, the optimized version creates more compact and concise code compared to non-optimized version. Other than these differences that you can see in the assembly code, Bloomfield also mentioned in lecture that optimized code for very large programs would take longer time to compile since it wants to optimize and shorten the code in order to improve running time. Also, he mentioned that as there are no local variables created in the optimized code, it is very hard to debug when it is compiled with the `-O2` flag since it is hard to trace what is going on in each level. Thus, for speed of execution, optimized code wins but for debugging and compile time, non-optimized code is better.