Adarsh Solanki
As5nr
CS 2150: In-Lab 9
4/11/12

Optimized Code

For this section of the lab, I compiled a simple C file (tried C++, but the generated assembly was identical for both according to `diff` because no C++ specific features were used)

```c
int main() {

    int x = 5 + 4;   // 9
    int y = 3 - 2;   // 1

    int z = (x + y)/2;   // 5
    z = 4;
    return z;
}
~
```

```asm
    .section    __TEXT,__text,regular
    .globl  _main
    .align  4, 0x90
_main:
Leh_func_begin1:
    pushq   %rbp
Ltmp0:
    movq    %rsp, %rbp
Ltmp1:
    movl    $9, -12(%rbp)
    movl    $1, -16(%rbp)
    movl    -12(%rbp), %eax
    movl    -16(%rbp), %ecx
    addl    %ecx, %eax
    movl    %eax, %ecx
    shrl    $31, %ecx
    leal    (%rax,%rcx), %eax
    sarl    %eax
    movl    %eax, -20(%rbp)
    movl    $4, -20(%rbp)
    movl    $4, -8(%rbp)
    movl    $4, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    ret
Leh_func_end1:
```

Above: original C code
Right: basic g++ assembly output
Below: optimized (O2) g++ assembly output

```asm
    .section    __TEXT,__t
    .globl  _main
    .align  4, 0x90
_main:
Leh_func_begin1:
    pushq   %rbp
Ltmp0:
    movq    %rsp, %rbp
Ltmp1:
    movl    $4, %eax
    popq    %rbp
    ret
Leh_func_end1:
```

The difference with the –O2 flag is tremendous! The entire main function was a 17 instruction x86 segment that closely matched the source C code. There are definitely also some inefficiencies, for example at the very end, there are a lot of extraneous 4's being placed onto the stack and then eventually one is pushed into the return register and the function returns.

Conversely, on the optimized assembly, g++ did whatever it could to minimize the number of instructions required to run the output assembly, so it took the time to pre-process things like the return value. The main function is 5 instructions now, 4 of which are standard boilerplate like saving/restoring the base pointer and calling `ret` to return. The way I see it, the O2 flag gives g++ the directive to take extra time now to make it more efficient later, pre-compute anything that you already have enough information to compute and cut-out any unnecessary movement/storage of data.

Cout Function

The next type of optimization I explored was optimization of I/O using standard functions: namely `cout.` Although, I didn't expect to really be able to understand most of the generated assembly, as I imagine I/O to be complex on the lower levels, I did gain some valuable insights throughout reading through the assembled programs. I ended up reading a good deal into the differences between the standard x86-64 architecture and that of x86-64 Mac OS X computers.

The following are screen captures of the C++ source, the default assembly, and the optimized assembly (in that order):

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << 127 << endl;
}
```

Original Source

Truncated assembly

```
    .section    __TEXT,__text,regular,pure_instructions
    .globl  _main
    .align  4, 0x90
_main:
Leh_func_begin1:
    pushq   %rbp
Ltmp0:
    movq    %rsp, %rbp
Ltmp1:
    subq    $16, %rsp
Ltmp2:
    movq    __ZSt4cout@GOTPCREL(%rip), %rax
    leaq    (%rax), %rax
    movl    $127, %ecx
    movq    %rax, %rdi
    movl    %ecx, %esi
    callq   __ZNSolsEi
    movq    __ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GOTPCREL(%rip), %rcx
    leaq    (%rcx), %rcx
    movq    %rax, %rdi
    movq    %rcx, %rsi
    callq   __ZNSolsEPFRSoS_E
    movl    $0, -8(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    addq    $16, %rsp
    popq    %rbp
    ret
Leh_func_end1:

    .section    __TEXT,__StaticInit,regular,pure_instructions
    .align  4, 0x90
__GLOBAL__I_main:
Leh_func_begin2:
    pushq   %rbp
Ltmp3:
    movq    %rsp, %rbp
Ltmp4:
    movl    $1, %eax
    movl    $65535, %ecx
    movl    %eax, %edi
    movl    %ecx, %esi
    callq   __Z41__static_initialization_and_destruction_0ii
    popq    %rbp
    ret
Leh_func_end2:

    .align  4, 0x90
__Z41__static_initialization_and_destruction_0ii:
Leh_func_begin3:
    pushq   %rbp
Ltmp5:
    movq    %rsp, %rbp
Ltmp6:
```

Optimized Assembly

```
	.section	__TEXT,__text,regular,pure_instructions
	.globl	_main
	.align	4, 0x90
_main:
Leh_func_begin1:
	pushq	%rbp
Ltmp0:
	movq	%rsp, %rbp
Ltmp1:
	movq	__ZSt4cout@GOTPCREL(%rip), %rdi
	movl	$127, %esi
	callq	__ZNSolsEi
	movq	__ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GOTPCREL(%rip), %rsi
	movq	%rax, %rdi
	callq	__ZNSolsEPFRSoS_E
	xorl	%eax, %eax
	popq	%rbp
	ret
Leh_func_end1:

	.section	__TEXT,__StaticInit,regular,pure_instructions
	.align	4, 0x90
__GLOBAL__I_main:
Leh_func_begin2:
	pushq	%rbp
Ltmp2:
	movq	%rsp, %rbp
Ltmp3:
	leaq	__ZStL8__ioinit(%rip), %rdi
	callq	__ZNSt8ios_base4InitC1Ev
	leaq	___tcf_0(%rip), %rdi
	xorl	%esi, %esi
	movq	___dso_handle@GOTPCREL(%rip), %rdx
	popq	%rbp
	jmp	___cxa_atexit  # TAILCALL
Leh_func_end2:

	.section	__TEXT,__text,regular,pure_instructions
	.align	4, 0x90
___tcf_0:
Leh_func_begin3:
	pushq	%rbp
Ltmp4:
	movq	%rsp, %rbp
Ltmp5:
	leaq	__ZStL8__ioinit(%rip), %rdi
	popq	%rbp
	jmp	__ZNSt8ios_base4InitD1Ev  # TAILCALL
Leh_func_end3:

.zerofill __DATA,__bss,__ZStL8__ioinit,1,3
	.section	__DATA,__mod_init_func,mod_init_funcs
	.align	3
	.quad	__GLOBAL__I_main
	.section	__TEXT,__eh_frame,coalesced,no_toc+strip_static_syms+live_support
EH_frame0:
Lsection_eh_frame:
```

The first thing I noticed is that the optimized assembly code passes much less to the __ZNSolsEi function that is called. The original code passes in __ZSt4cout@GOTPCREL($rip) [via $RAX] to the function. This intrigued me, as it seemed like it was passing in a function to the function.

It turns out, the GOTPCREL($rip) is the Mac x86-64 environment's way of accessing local and small data. (Source: http://developer.apple.com/library/mac#documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/x86_64_code.html)

My interpretation of this is that the prefix __ZSt4cout is a label for one of the functions that exists in the iostream library (the cout function, perhaps a specific one to print a number vs. printing another data type) and is represented in the program space's "global offset table" (GOT) which uses "RIP-relative addressing" which means addressing relative to the instruction pointer.

The optimized code performs very similar things, but in fewer instructions. For example, the default code stores the ZSt4cout function pointer in $rax and then moves it from $rax to $rdi. This is probably due to convention. The optimized code on the other hand stores the information directly into $rdi. I don't understand why modern compilers don't always optimize little steps like these, as it doesn't even seem to be an optimization, but rather an improvement on the inefficient assembly produced by default. I'm sure there is a reason for this that involves things beyond the scope of my understanding.

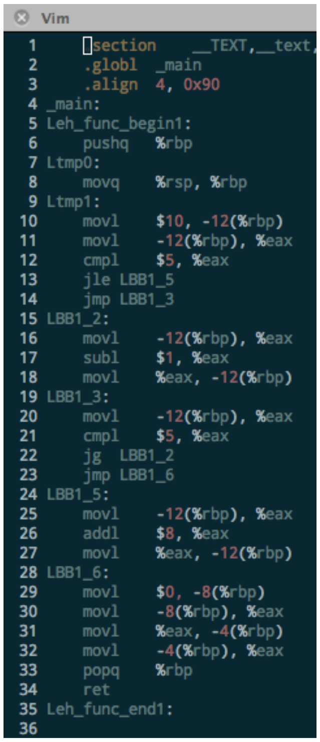
Control Flow

The final optimization that I tried to study was that of control flow. I made a C++ program that included a very primitive demonstration of if/else and a while loop.

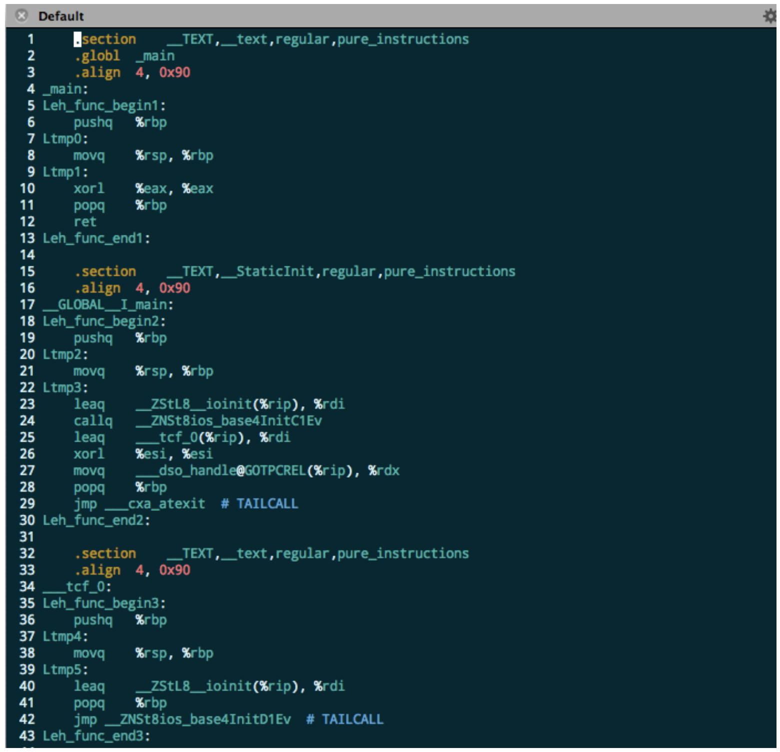
```
Vim
#include <iostream>

using namespace std;

int main() {
    int x = 10;
    if (x > 5) {
        while ( x > 5 ) {
            x--;
        }
    } else {
        x += 8;
    }
}
```



default



optimized

The loop iteration in the non-optimized code is pretty similar to how I would code it myself. It creates a few labels for each of the control flow “states” and then uses the basic `cmp, jle, j, jge` instructions that I would use, were I to code this myself.

The optimized code on the other hand, is very different. It is much harder to draw similarities between the C++ code and the optimized assembly code. I tried to search for some common instructions like `cmp` or any jump, but to no avail, so I decided to dig around in the code and see what I could find through the Internet.

The first thing that jumped out to me was the line

```
jmp __cxa_atexit # TAILCALL
```

This seemed odd, because I hadn’t yet seen g++ generate comments in assembled code. It turned out that this was actually present in the default assembly code, as well as every other assembled program I had generated with g++. It turned out to be g++’s way of doing constructors, so this must have been some sort of initialization of the main() object? This leads me to believe that this program simply does nothing, as the first (main) function simply xor's EAX with itself (zero's it) and then returns. The compiler rendered my function equivalent to "return 0;" and come to think of it, it technically was, as none of the variables or register values have any importance outside of the scope of the function itself.