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):

Original Source

```
#include <iostream>

using namespace std;

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

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;
    }
```

Vim

```
	section	__TEXT,__text,
	.globl	_main
	.align	4, 0x90
_main:
Leh_func_begin1:
	pushq	%rbp
Ltmp0:
	movq	%rsp, %rbp
Ltmp1:
	movl	$10, -12(%rbp)
	movl	-12(%rbp), %eax
	cmpl	$5, %eax
	jle LBB1_5
	jmp LBB1_3
LBB1_2:
	movl	-12(%rbp), %eax
	subl	$1, %eax
	movl	%eax, -12(%rbp)
LBB1_3:
	movl	-12(%rbp), %eax
	cmpl	$5, %eax
	jg  LBB1_2
	jmp LBB1_6
LBB1_5:
	movl	-12(%rbp), %eax
	addl	$8, %eax
	movl	%eax, -12(%rbp)
LBB1_6:
	movl	$0, -8(%rbp)
	movl	-8(%rbp), %eax
	movl	%eax, -4(%rbp)
	movl	-4(%rbp), %eax
	popq	%rbp
	ret
Leh_func_end1:
```

default

Default

```
	.section	__TEXT,__text,regular,pure_instructions
	.globl	_main
	.align	4, 0x90
_main:
Leh_func_begin1:
	pushq	%rbp
Ltmp0:
	movq	%rsp, %rbp
Ltmp1:
	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:
```

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.

Inheritance

In order to explore the assembly language that is created with classes, I created a simple "Numbers" class which holds three integer pointers, and then I created a subclass called "MoreNumbers" which holds an additional integer pointer. The code for the two classes are shown to the right and below:

```
class Numbers {
  public:
    Numbers( int *n1, int *n2, int *n3 );
    ~Numbers();
    int *x1;
    int *x2;
    int *x3;

};

Numbers::Numbers( int *n1, int *n2, int *n3 ) {
   x1 = n1;
   x2 = n2;
   x3 = n3;
}

Numbers::~Numbers() {
    delete x1;
    delete x2;
    delete x3;
}
```

```
class MoreNumbers : public Numbers {
    public:
      MoreNumbers( int *n1, int *n2, int *n3, int *n4 );
      ~MoreNumbers();
      int *x4;
};

MoreNumbers::MoreNumbers( int *n1, int *n2, int *n3, int *n4 ) : Numbers ( n1, n2, n3 ) {
    x4 = n4;
}

MoreNumbers::~MoreNumbers() {
    delete x4;
}

```

The main function of the program simply instantiates various integers, and creates one instance of Numbers and one of of MoreNumbers, and then deletes them in order to be able to properly examine the constructor and destructor of both a normal object and an object which inherits some of its data.

```cpp
int main() {
    int t1 = 0;
    int t2 = 2;
    int t3 = 3;
    Numbers *n = new Numbers ( &t1, &t2, &t3 );

    t1 = 5;
    n->x1 = &t1;

    t1 = 9;
    t2 = 10;
    t3 = 16;
    int t4 = 15;

    MoreNumbers *m = new MoreNumbers( &t1, &t2, &t3, &t4 );

    delete n;
    delete m;

}
```

To the left is a screen capture of the main function, showing its exact functionality, followed by the associated assembly code.

The first few lines in the snippet are showing the three arguments for the Numbers constructor (0, 2, 3) as written in the C++, but next there is an instruction that I have never seen before, `movabsq $24, %rax.` From an online search, I found that this is meant to move an absolute quadword, and I think it may be passing in the size of the arguments (24 = 8 * 3, 8 for each pointer). The function that is called is called (Znwm) doesn't seem to be in any part of the assembly file, and it is called again later in the code to call the constructor for MoreNumbers, this is probably to do the inherited functionality from the base-class's constructor. On second thought, it appears to be a function that is called before the constructor, which I believe to be the function on line 294.

Default

```
275 Ltmp29:
276     movl    $0, -44(%rbp)
277     movl    $2, -48(%rbp)
278     movl    $3, -52(%rbp)
279     movabsq $24, %rax
280     movq    %rax, %rdi
281     callq   __Znwm
282     movq    %rax, -40(%rbp)
283     movq    -40(%rbp), %rax
284     leaq    -44(%rbp), %rcx
285     leaq    -48(%rbp), %rdx
286     leaq    -52(%rbp), %rsi
287     movq    %rax, %rdi
288     movq    %rsi, -88(%rbp)
289     movq    %rcx, %rsi
290     movq    %rdx, -96(%rbp)
291     movq    -88(%rbp), %rax
292     movq    %rcx, -104(%rbp)
293     movq    %rax, %rcx
294     callq   __ZN7NumbersC1EPiS0_S0_
295     movq    -40(%rbp), %rax
296     movq    %rax, -64(%rbp)
297     movl    $5, -44(%rbp)
298     movq    -64(%rbp), %rax
299     movq    -104(%rbp), %rcx
300     movq    %rcx, (%rax)
301     movl    $9, -44(%rbp)
302     movl    $10, -48(%rbp)
303     movl    $16, -52(%rbp)
304     movl    $15, -68(%rbp)
305     movabsq $32, %rax
306     movq    %rax, %rdi
307     callq   __Znwm
308     movq    %rax, -32(%rbp)
309     movq    -32(%rbp), %rax
310     leaq    -68(%rbp), %rcx
311     movq    %rax, %rdi
312     movq    -104(%rbp), %rsi
313     movq    -96(%rbp), %rdx
314     movq    -88(%rbp), %rax
315     movq    %rcx, -112(%rbp)
316     movq    %rax, %rcx
317     movq    -112(%rbp), %r8
318     callq   __ZN11MoreNumbersC1EPiS0_S0_
319     movq    -32(%rbp), %rax
320     movq    %rax, -80(%rbp)
321     movq    -64(%rbp), %rax
322     movq    %rax, -24(%rbp)
323     movq    -24(%rbp), %rax
324     cmpq    $0, %rax
325     je  LBB11_2
326     movq    -24(%rbp), %rax
327     movq    %rax, %rdi
328     callq   __ZN7NumbersD1Ev
329     movq    -24(%rbp), %rax
330     movq    %rax, %rdi
331     callq   __ZdlPv
```

Next, it appears that the return value (the object itself?) is stored 40 bytes behind the base pointer. I know that 44 bytes behind the base pointer lies the t1 member variable, as line 297 seems to be setting it to five. This means that the first four bytes (40-44 behind the base) are probably the implicit "this", and the following four byte chunks are the data members.

```
1       .section    __TEXT,__text
2       .globl  __ZN7NumbersC2EPi
3       .align  1, 0x90
4 __ZN7NumbersC2EPiS0_S0_:
5 Leh_func_begin1:
6       pushq   %rbp
7 Ltmp0:
8       movq    %rsp, %rbp
9 Ltmp1:
10      movq    %rdi, -8(%rbp)
11      movq    %rsi, -16(%rbp)
12      movq    %rdx, -24(%rbp)
13      movq    %rcx, -32(%rbp)
14      movq    -8(%rbp), %rax
15      movq    -16(%rbp), %rcx
16      movq    %rcx, (%rax)
17      movq    -8(%rbp), %rax
18      movq    -24(%rbp), %rcx
19      movq    %rcx, 8(%rax)
20      movq    -8(%rbp), %rax
21      movq    -32(%rbp), %rcx
22      movq    %rcx, 16(%rax)
23      popq    %rbp
24      ret
25 Leh_func_end1:
```

Line 318 seems to be the call to the constructor of MoreNumbers, and in name alone, it seems to have a lot in common with the constructor of Numbers, except for the addition of "More" and one extra "S0_" suffix. This is probably a result of the inheritance. I was expecting the assembler to automatically call both constructors, perhaps calling the base-class's constructor as a part of the call to the constructor of the subclass, and I predicted correctly.

```
70      .globl  __ZN11MoreNumbersC2EPiS0_S0_S0_
71      .align  1, 0x90
72 __ZN11MoreNumbersC2EPiS0_S0_S0_:
73 Leh_func_begin4:
74      pushq   %rbp
75 Ltmp6:
76      movq    %rsp, %rbp
77 Ltmp7:
78      subq    $48, %rsp
79 Ltmp8:
80      movq    %rdi, -8(%rbp)
81      movq    %rsi, -16(%rbp)
82      movq    %rdx, -24(%rbp)
83      movq    %rcx, -32(%rbp)
84      movq    %r8, -40(%rbp)
85      movq    -8(%rbp), %rax
86      movq    -16(%rbp), %rcx
87      movq    -24(%rbp), %rdx
88      movq    -32(%rbp), %rsi
89      movq    %rax, %rdi
90      movq    %rsi, -48(%rbp)
91      movq    %rcx, %rsi
92      movq    -48(%rbp), %rcx
93      callq   __ZN7NumbersC2EPiS0_S0_
94      movq    -8(%rbp), %rax
95      movq    -40(%rbp), %rcx
96      movq    %rcx, 24(%rax)
97      addq    $48, %rsp
98      popq    %rbp
99      ret
100 Leh_func_end4:
```

Above is the body of the Numbers constructor, and to the right is the body of the MoreNumbers constructor. As is apparent on line 93, the MoreNumbers constructor calls the Numbers constructor within itself. I cannot exactly tell how the data is manipulated after the constructor is called though.

```
326     movq    -24(%rbp), %rax
327     movq    %rax, %rdi
328     callq   __ZN7NumbersD1Ev
329     movq    -24(%rbp), %rax
330     movq    %rax, %rdi
331     callq   __ZdlPv
332 LBB11_2:
333     movq    -80(%rbp), %rax
334     movq    %rax, -16(%rbp)
335     movq    -16(%rbp), %rax
336     cmpq    $0, %rax
337     je  LBB11_4
338     movq    -16(%rbp), %rax
339     movq    %rax, %rdi
340     callq   __ZN11MoreNumbersD1Ev
341     movq    -16(%rbp), %rax
342     movq    %rax, %rdi
343     callq   __ZdlPv
344 LBB11_4:
345     movl    $0, -8(%rbp)
346     movl    -8(%rbp), %eax
347     movl    %eax, -4(%rbp)
348     movl    -4(%rbp), %eax
349     addq    $112, %rsp
350     popq    %rbp
351     ret
352 Leh_func_end11:
```

I believe that per the x86 naming conventions of g++, the constructors are always the name of the class followed by a C, and the destructors with a D. That being said, the destructors are the next functions called in the main function (right, lines 328 and 340).

The prologue for the first destructor seems to be data passed in through the RDI register from behind the base pointer. This is probably the "this" pointer for the object that is allocated somewhere on the heap. The next function called at 331 "ZdlPv" is kind of confusing, but from purely the name it sounds like it **de**letes a **p**ointer **v**ariable, and perhaps it is responsible for deallocating the data members of the instance of the Numbers class. From searching through the Internet, I found references to this strange function on gnu.org and realized that it is a part of GCC and stands for "operator delete(void*)" (source: http://lists.gnu.org/archive/html/bug-binutils/2010-01/msg00047.html)meaning it deletes a void pointer (a point of any type). It seems to me like this is probably the function used to delete all pointers in every single destructor.

Predictably, the function is called again after the MoreNumbers destructor is called, leading me to believe that this Zdlpv is the function to actually free the memory itself, whereas the other functions that share the namesake of their respective classes exist simply to perform any other functionality that might need to exist for an objects destruction.