Introduction to X86-64 Assembly for Compiler Writers

by Prof. Douglas Thain

(Last updated 9-Nov-2015)

Note: This material has been incorporated into Chapter 10 of the free online textbook Introduction to Compilers and Language Design.

Overview

This is a brief introduction to X86-64 assembly language novice compiler writers using the GNU software tools. It is not an exhaustive description of the architecture, but it is enough to orient you toward the official manuals and write most of the backend of a C compiler for an undergraduate class.

X86-64 is a generic term that refers to 64-bit extensions to the industry standard X86 32-bit architecture. X86-64 is often used interchangeably with the names X64, AMD64, Intel-64, and EMT64, but note that it is *not* the same as the IA64 architecture. X86-64 at Wikipedia has a nice overview of some of the history and minor distinctions between these terms.

The X86-64 instruction set is described in complete detail in the Intel-64 and IA-32 Architectures Software Developer Manuals, available freely online. You will need to browse these manual and pick out key details as you go. I recommend that you download the PDFs to your laptop, keep them handy, and then read the following sections of the manual:

- Volume 1: Sections 2.1, 3.4, and 3.7
- Volume 2: Read instructions as needed.

Open Source Assembler Tools

For these examples, we will be using the GNU compiler and assembler, known as `gcc` and `as` (or sometimes `gas`.) A quick way to learn something about assembly is to view the assembler output of the compiler. To do this, run `gcc` with the `-S` flag, and the compiler will produce assembly output rather than a binary program. On Unix-like systems, assembly code is stored in files ending with `.s`. (The suffix "s" stands for "source" file, whereas the suffix "a" is used to indicate an "archive" (library) file.) So, `gcc -S hello.c` on this program:

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
        printf("hello %s\n","world");
        return 0;
}
```

will yield a file `hello.S` that looks something like this:

```
.file   "test.c"

.data
```

```
.LC0:
        .string "hello %s\n"
.LC1:
        .string "world"

.text
.globl main
.type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        movl    $.LC0, %eax
        movl    $.LC1, %esi
        movq    %rax, %rdi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-11)"
        .section        .note.GNU-stack,"",@progbits
```

Note that the assembly code has three different kinds of elements:

- **Directives** begin with a dot and indicate structural information useful to the assembler, linker, or debugger. but are not in and of themselves assembly instructions. For example, `.file` simply records the name of the original source file. `.data` indicates the start of the data section of the program, while `.text` indicates the start of the actual program code. `.string` indicates a string constant within the data section, and `.globl main` indicates that the label `main` is a global symbol that can be accessed by other code modules. (You can ignore most of the other directives.)
- **Labels** end with a colon and indicate by their position the association between names and locations. For example, the label `.LC0:` indicates that the immediately following string should be called `.LC0`. The label `main:` indicates that the instruction `pushq %rbp` is the first instruction of the main function. By convention, labels beginning with a dot are temporary local labels generated by the compiler, while other symbols are user-visible functions and global variables.
- **Instructions** are the actual assembly code (`pushq %rbp`), typically indented to visually distinguish them from directives and labels.

To take this assembly code and turn it into a runnable program, just run `gcc`, which will figure out that it is an assembly program, assemble it, and link it with the standard library:

```
% gcc hello.s -o hello
% ./hello
hello world
```

It is also interesting to compile the assembly code into object code, and then use the `nm` utility to display the symbols ("names") present in the code:

```
% gcc hello.s -c -o hello.o
```

```
% nm hello.o
0000000000000000 T main
                 U printf
```

This display the information available to the linker. `main` is present in the text (`T`) section of the object, at location zero, and `printf` is undefined (`U`), since it must be obtained from the standard library. But none of the labels like `LC0` because they were not declared as `.globl`.

Take advantage of the fact that GCC emits assembly code. If you don't know quite what instructions to generate with your compiler, see what GCC emits and then look up the details in the Intel manual.

Now that you know what tools to use, let's begin to look at the assembly instructions in detail.

Sidebar: AT&T Syntax versus Intel Syntax

Note that the GNU tools use the traditional AT&T syntax, which is used across many processors on Unix-like operating systems, as opposed to the Intel syntax typically used on DOS and Windows systems. The following instruction is given in AT&T syntax:

```
movl %esp, %ebp
```

`movl` is the name of the instruction, and the percent signs indicate that `esp` and `ebp` are registers. In the AT&T syntax, the source is always given first, and the destination is always given second.

In other places (such as the Intel manual), you will see the Intel syntax, which (among other things) dispenses with the percent signs and *reverses* the order of the arguments. For example, this is the same instruction in the Intel syntax:

```
MOVQ EBP, ESP
```

When reading manuals and web pages, be careful to determine whether you are looking at AT&T or Intel syntax: look for the percent signs!

Registers and Data Types

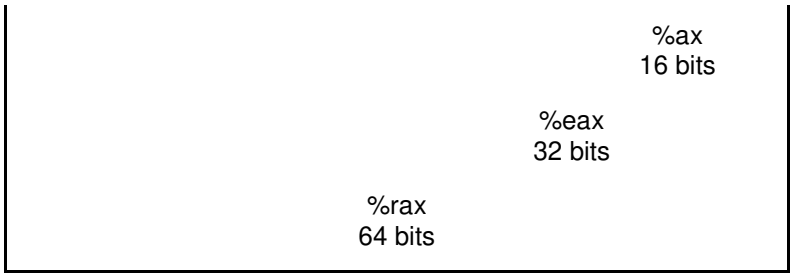
X86-64 has sixteen (almost) general purpose 64-bit integer registers:

%rax	%rbx	%rcx	%rdx	%rsi	%rdi	%rbp	%rsp	%r8	%r9	%r10	%r11	%r12	%r13	%r14	%r15

We say *almost* general purpose because earlier versions of the processors intended for each register to be used for a specific purpose, and not all instructions could be applied to every register. As the design developed, new instructions and addressing modes were added to make the various registers almost equal. A few remaining instructions, particularly related to string processing, require the use of %rsi and %rdi. In addition, two registers are reserved for use as the stack pointer (%rsp) and the base pointer (%rbp). The final eight registers are numbered and have no specific restrictions.

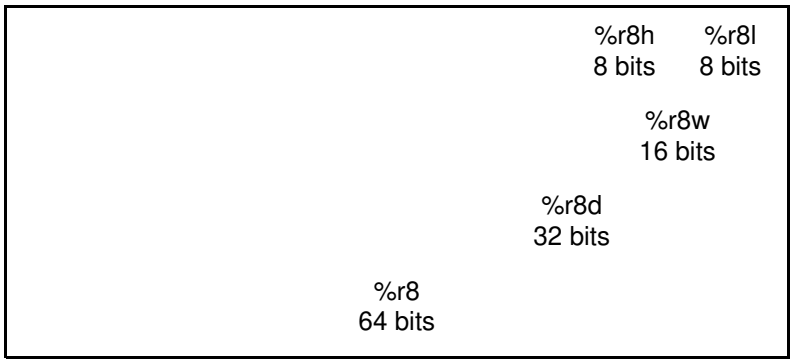
The architecture has expanded from 8 to 16 to 32 bits over the years, and so each register has some internal structure that you should know about:

	%ah 8 bits	%al 8 bits



The lowest 8 bits of the %rax register are an 8-bit register %al, and the next 8 bits are known as %ah. The low 16 bits are collectively known as %ax, the low 32-bits as %eax, and the whole 64 bits as %rax.

The numbered registers %r8-%r15 have the same structure, but a slightly different naming scheme:



To keep things simple, we will focus our attention on the 64-bit registers. (C-minor was designed explicitly to use 64-bit arithmetic to help you out.) However, most production compilers use a mix of modes: The 32-bit registers are generally used for integer arithmetic, since most programs don't need integer values above 2^32 (4.2 billion). The 64-bit registers are generally used to hold memory addresses (pointers), enabling addressing up to 16EB (exa-bytes) of virtual memory.

Addressing Modes

The first instruction that you should know about is the MOV instruction, which moves data between registers and to and from memory. X86-64 is a complex instruction set (CISC), so the MOV instruction has many different variants that move different types of data between different cells.

MOV, like most instructions, has a single letter suffix that determines the amount of data to be moved. The following names are used to describe data values of various sizes:

Suffix	Name	Size
B	BYTE	1 byte (8 bits)
W	WORD	2 bytes (16 bits)
L	LONG	4 bytes (32 bits)
Q	QUADWORD	8 bytes (64 bits)

So, MOVB moves a byte, MOVW moves a word, MOVL moves a long, MOVQ moves a quad-word. Generally, the size of the locations you are moving to and from must match the suffix. It is possible to leave off the suffix, and the assembler will attempt to choose the right size based on the arguments. However, this is not recommended, as it can have unexpected effects.

The arguments to MOV can have one of several addressing modes. A **global value** is simply referred to by an

unadorned name such as `x` or `printf` An **immediate value** is a constant value indicated by a dollar sign such as `$56` A **register value** is the name of a register such as `%rbx`. An **indirect** refers to a value by the address contained in a register. For example, `(%rsp)` refers to the value pointed to by %rsp. A **base-relative** value is given by adding a constant to the name of a register. For example, `-16(%rcx)` refers to the value at the memory location sixteen bytes below the address indicated by `%rcx`. This mode is important for manipulating stacks, local values, and function parameters. There are a variety of complex variations on base-relative, for example `-16(%rbx,%rcx,8)` refers to the value at the address `-16+%rbx+%rcx*8`. This mode is useful for accessing elements of unusual sizes arranged in arrays.

Here is an example of using each kind of addressing mode to load a 64-bit value into `%rax`:

Mode	**Example**
Global Symbol	MOVQ x, %rax
Immediate	MOVQ $56, %rax
Register	MOVQ %rbx, %rax
Indirect	MOVQ (%rsp), %rax
Base-Relative	MOVQ -8(%rbp), %rax
Offset-Scaled-Base-Relative	MOVQ -16(%rbx,%rcx,8), %rax

For the most part, the same addressing modes may be used to store data into registers and memory locations. However, not *all* modes are supported. For example, it is not possible to use base-relative for both arguments of MOV: `MOVQ -8(%rbx), -8(%rbx)`. To see exactly what combinations of addressing modes are supported, you must read the manual pages for the instruction in question.

Basic Arithmetic

You will need four basic arithmetic instructions for your compiler: ADD, SUB, IMUL, and IDIV. ADD and SUB have two operands: a source and a destructive target. For example, this instruction:

```
ADDQ %rbx, %rax
```

adds %rbx to %rax, and places the result in %rax, overwriting what might have been there before. This requires that you be a little careful in how you make use of registers. For example, suppose that you wish to translate `c = b*(b+a)`, where `a` and `b` are global integers. To do this, you must be careful not to clobber the value of `b` when performing the addition. Here is one possible translation:

```
MOVQ  a, %rax
MOVQ  b, %rbx
ADDQ  %rbx, %rax
IMULQ %rbx
MOVQ  %rax, c
```

The IMUL instruction is a little unusual: it takes its argument, multiplies it by the contents of %rax, and then places the low 64 bits of the result in %rax and the high 64 bits in %rdx. (Multiplying two 64-bit numbers yields a 128-bit number, after all.)

The IDIV instruction does the same thing, except backwards: it starts with a 128 bit integer value whose low 64 bits are in %rax and high 64 bits in %rdx, and divides it by the value give in the instruction. (The CDQO instruction serves the very specific purpose of sign-extending %rax into %rdx, to handle negative values correctly.) The quotient is placed in %rax and the remainder in %rdx. For example, to divide a by five:

```
MOVQ a,  %rax    # set the low 64 bits of the dividend
```

```
CDQO            # sign-extend %rax into %rdx
IDIVQ $5        # divide %rdx:%rax by 5, leaving result in %eax
```

(Note that the modulus instruction found in most languages simply exploits the remainder left in %rdx.)

The instructions INC and DEC increment and decrement a register destructively. For example, the statement `a = ++b` could be translated as:

```
MOVQ b, %rax
INCQ %rax
MOVQ %rax, a
```

Boolean operations work in a very similar manner: AND, OR, and XOR perform destructive boolean operations on two operands, while NOT performs a destructive boolean-not on one operand.

Like the MOV instruction, the various arithmetic instructions can work on a variety of addressing modes. However, for your compiler project, you will likely find it most convenient to use MOV to load values in and out of registers, and then use only registers to perform arithmetic.

Sidebar: Floating Point

While we won't cover floating point operations in detail, you should at least know that they are handled by a different set of instructions and a different set of registers. In older machines, floating point instructions were handled by an optional (external) chip known as the 8087 FPU, hence these capabilities are often described as X87 operations, even though the capability is now integerated into the CPU itself.

The X87 FPU contains eight 80-bit registers (R0-R7) arranged in a stack. To perform floating-point arithmetic, code must push data onto the FPU stack, issue instructions that implicitly manipulate the top of the stack, and then write the items back into memory. In memory, double-precision numbers are stored in 64-bit values.

An oddity of this architecture is that the internal representation (80 bits) has higher precision than the representation in memory (64 bits). As a result, the values of floating point computations can change, depending on the precise ordering of when values are moved between registers and memory!

Floating point math is much more subtle than it first appears, the following is recommended reading:

- Chapter 8-1 of the Intel manual.
- David Goldberg, What Every Computer Scientist Should Know About Floating Point Arithmetic, ACM Computing Surveys, volume 23, issue 1, March 1991.
- A more informal guide: What Every Programmer Should Know about Floating Point

Comparisons and Jumps

Using the JMP instruction, we may create a simple infinite loop that counts up from zero using the %eax register:

```
        MOVQ $0, %rax
loop:
        INCQ %rax
        JMP loop
```

To define more useful structures such as *terminating* loops and if-then statements, we must have a mechanism for evaluating values and changing program flow. In most assembly languages, these are handled by two different

kinds of instructions: compares and jumps.

All comparisons are done with the CMP instruction. CMP compares two different registers and then sets a few bits in an internal EFLAGS registers, recording whether the values are the same, greater, or lesser. You don't need to look at the EFLAGS register directly. Instead a selection of conditional jumps examine the EFLAGS register and jump appropriately:

Instruction	Meaning
JE	Jump If Equal
JNE	Jump If Not Equal
JL	Jump If Less Than
JLE	Jump If Less or Equal
JG	Jump if Greater Than
JGE	Jump If Greater or Equal

For example, here is a loop to count %rax from zero to five:

```
        MOVQ $0, %rax
loop:
        INCQ %rax
        CMPQ $5, %rax
        JLE  loop
```

And here is a conditional assignment: if global variable x is greater than zero, then global variable y gets ten, else twenty:

```
        MOVQ x, %rax
        CMPQ $0, %rax
        JLE  twenty
ten:
        MOVQ $10, %rbx
        JMP  done
twenty:
        MOVQ $20, %rbx
        JMP  done
done:
        MOVQ %ebx, y
```

Note that jumps require the compiler to define target labels. These labels must be unique and private within one assembly file, but cannot be seen outside the file unless a `.globl` directive is given. In C parlance, a plain assembly label is `static`, while a `.globl` label is `extern`.

The Stack

The stack is an auxiliary data structure used primarily to record the function call history of the program along with local variables that do not fit in registers.

By convention, the stack grows **downward** from high values to low values. The %rsp register is known as the "stack pointer" and keeps track of the bottom-most item on the stack. So, to push %rax onto the stack, we must subtract 8 (the size of %rax in bytes) from %rsp and then write to the location pointed to by %rsp:

```
SUBQ $8, %rsp
MOVQ %rax, (%rsp)
```

Popping a value from the stack involves the opposite:

```
MOVQ (%rsp), %rax
ADDQ $8, %rsp
```

To discard the most recent value from the stack, just move the stack pointer:

```
ADDQ $8, %rsp
```

Of course, pushing to and popping from the stack referred to by %rsp is so common, that the two operations have their own instructions that behave exactly as above:

```
PUSHQ %rax
POPQ  %rax
```

Calling Other Functions

All of the functions available in the C standard library are also available in an assembly-language program. The are invoked in a standard way known as a "calling convention" so that code written in multiple languages can all be linked together.

In **most** assembly languages (but not X86-64) the calling convention is simply to push each argument on to the stack, and then invoke the function. The called function looks for the arguments on the stack, does its work, and returns the result in a single register. The caller then pops the arguments off the stack. (In fact, this is exactly the calling convention for X86 32-bit code and you could continue to use it, if you were only working with your own code.)

The calling convention used by X86-64 on Linux is somewhat different and is known as the System V ABI. The complete convention is rather complicated, but following is a simplified explanation that will be sufficient for us:

- Integer arguments (including pointers) are placed in the registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9, in that order.
- Floating point arguments are placed in the registers %xmm0-%xmm7, in that order.
- Arguments in excess of the available registers are pushed onto the stack.
- If the function takes a variable number of arguments (like `printf`) then the %eax register must be set to the number of floating point arguments.
- The called function may use any registers, but it must restore the values of the registers %rbx, %rbp, %rsp, and %r12-%r15, if it changes them.
- The return value of the function is placed in %eax.

This table summarizes what you need to know:

Register	Purpose	Saved?
%rax	result	not saved
%rbx	scratch	callee saves
%rcx	argument 4	not saved
%rdx	argument 3	not saved
%rsi	argument 2	not saved
%rdi	argument 1	not saved
%rbp	base pointer	callee saves
%rsp	stack pointer	callee saves

%r8	argument 5	not saved
%r9	argument 6	not saved
%r10	scratch	CALLER saves
%r11	scratch	CALLER saves
%r12	scratch	callee saves
%r13	scratch	callee saves
%r14	scratch	callee saves
%r15	scratch	callee saves

To invoke a function, we must first compute the arguments and place them in the desired registers. Then, we must push the two caller-saved registers (`%r10` and `%r11`) on the stack, to save their values. We then issue the `CALL` instruction, which pushes the current instruction pointer on to the stack then jumps to the code location of the function. Upon return from the function, we pop the two caller-saved registers off of the stack, and look for the return value of the function in the `%eax` register.

For example, the following C code fragment:

```
long x=0;
long y=10;

int main()
{
        x = printf("value: %d",y);
}
```

could be translated to this:

```
.data
x:
        .quad 0
y:
        .quad 10
str:
        .string "value: %d\n"

.text
.globl main
main:
        MOVQ  $str, %rdi  # first argument in %rdi: string pointer
        MOVQ  y,    %rsi  # second argument in %rsi: value of y
        MOVQ  $0,   %rax  # there are zero floating point args

        PUSHQ %r10        # save the caller-saved registers
        PUSHQ %r11

        CALL  printf      # invoke printf

        POPQ %r11         # restore the caller-saved registers
        POPQ %r10

        MOVQ  %rax, x     # save the result in x

        RET               # return from main function
```

Defining a Simple Leaf Function

Because function arguments are passed in registers, it is easy to write simple mathematical functions that compute a value and then return it. For example, code for the following function:

```
long square( long x )
{
        return x*x;
}
```

Can be as simple as this:

```
.global square
square:
        MOVQ  %rdi, %rax  # copy first argument to %rax
        IMULQ %rdi, %rax  # multiply it by itself
                          # result is already in %rax
        RET               # return to caller
```

Unfortunately, this only works for a **leaf function** that calls no other functions, and can complete its work within the set of registers already passed. Generalized functions emitted by a compiler must instead save and restore state to the stack.

Defining a Complex Function

A fully-featured function must be able to invoke other functions and compute expressions of arbitrary complexity, and then return to the caller with the original state intact. Consider the following recipe for a function that accepts three arguments and uses two local variables:

```
.globl func
func:
        pushq %rbp          # save the base pointer
        movq  %rsp, %rbp    # set new base pointer

        pushq %rdi          # save first argument on the stack
        pushq %rsi          # save second argument on the stack
        pushq %rdx          # save third argument on the stack

        subq  $16, %rsp     # allocate two more local variables

        pushq %rbx          # save callee-saved registers
        pushq %r12
        pushq %r13
        pushq %r14
        pushq %r15

        ### body of function goes here ###

        popq %r15            # restore callee-saved registers
        popq %r14
        popq %r13
        popq %r12
        popq %rbx

        movq   %rbp, %rsp    # reset stack to previous base pointer
        popq   %rbp          # recover previous base pointer
        ret                  # return to the caller
```

There is a lot to keep track of here: the arguments given to the function, the information necessary to return, and space for local computations. For this purpose, we use the base register pointer %rbp. Whereas the stack pointer %rsp points to the end of the stack where new data will be pushed, the base pointer %rbp points to the start of the values used by this function. The space between %rbp and %rsp is known as the "stack frame" or the "activation record" of the function call.

There is one more complication: each function needs to use a selection of registers to perform computations. However, what happens when one function is called in the middle of another? We do not want any registers currently in use by the caller to be clobbered by the called function. To prevent this, each function must save and restore all of the registers that it uses by pushing them onto the stack at the beginning, and popping them off of the stack before returning. According to the System V ABI, each function must preserve the values of %rsp, %rbp, %rbx, and %r12-%r15 when it completes.

Consider the stack layout for `func`, defined above:

Contents	**Address**	
old %rip register	8(%rbp)	
old %rbp register	(%rbp)	<-- %rbp points here
argument 0	-8(%rbp)	
argument 1	-16(%rbp)	
argument 2	-24(%rbp)	
local variable 0	-32(%rbp)	
local variable 1	-40(%rbp)	
saved register %rbx	-48(%rbp)	
saved register %r12	-56(%rbp)	
saved register %r13	-64(%rbp)	
saved register %r14	-72(%rbp)	
saved register %r15	-80(%rbp)	<-- %rsp points here
("top" of the stack)		

Note that the base pointer (`%rbp`) locates the start of the stack frame. So, within the body of the function, we may use base-relative addressing against the base pointer to refer to both arguments and locals. The arguments to the function follow the base pointer, so argument zero is at -8(%rbp), argument one at -16(%rbp), and so forth. Past those are local variables to the function at -32(%rbp) and then saved registers at -48(%rbp). The stack pointer points to the last item on the stack. If we use the stack for additional purposes, data will be pushed to further negative values. (Note that we have assumed all arugments and variables that are 8 bytes large: different types would result in different offsets.)

Here is a complete example that puts it all together. Suppose that you have a C function defined as follows:

```
int func( int a, int b, int c )
{
        int x, y;
        x = a+b+c;
        y = x*5;
        return y;
}
```

Here is a conservative translation of the function:

```
.globl func
func:
        ##################### preamble of function sets up stack

        pushq %rbp          # save the base pointer
```

```
        movq  %rsp, %rbp    # set new base pointer to esp

        pushq %rdi          # save first argument (a) on the stack
        pushq %rsi          # save second argument (b) on the stack
        pushq %rdx          # save third argument (c) on the stack

        subq  $16, %rsp     # allocate two more local variables

        pushq %rbx          # save callee-saved registers
        pushq %r12
        pushq %r13
        pushq %r14
        pushq %r15

        ######################## body of function starts here

        movq  -8(%rbp),  %rbx   # load each arg into a scratch register
        movq  -16(%rbp), %rcx
        movq  -24(%rbp), %rdx

        addq  %rdx, %rcx       # add the args together
        addq  %rcx, %rbx
        movq  %rbx, -32(%rbp)   # store the result into local 0 (x)

        movq  -32(%rbp), %rbx   # load local 0 (x) into a scratch register.
        imulq  $5, %rbx         # multiply it by 5
        movl  %rbx, -40(%ebp)   # store the result in local 1 (y)

        movl  -20(%ebp), %eax   # move local 1 (y) into the result register

        ################### epilogue of function restores the stack

        popq %r15          # restore callee-saved registers
        popq %r14
        popq %r13
        popq %r12
        popq %rbx

        movq %rbp, %rsp    # reset stack to base pointer.
        popq %rbp          # restore the old base pointer

        ret                # return to caller
```

Thoughts on Optimization

In retrospect, there are many ways in which the example above could be improved. As it turned out, this particular code didn't need to use registers `%rbx-%r15`, so we didn't have to save them. With care, we could have kept the arguments in registers without saving them to the stack. The result could have been computed directly into `%eax` rather than saving it to a local variable. These optimizations are easy to make when writing code by hand, but not so easy when writing a compiler.

For your first attempt at building a compiler, your code created will not be very efficient, because each statement is translated independently. The preamble to a function must save all the registers, because it does not know *a priori* which registers will be used later. A statement that computes a value must save it back to a local variable, because it does not know *a priori* that the local will be used as a return value.

That said, a clever compiler can reduce things significantly. Try compiling the C source for `func` with the gcc compiler, with and without the `-O` flag. Examine the generated assembly carefully to see how it was optimzed.

At later points in the semester, we will discuss a variety of strategies by which generated code may be optimized by a compiler. For now, worry about making your code conservatively correct, rather than efficient.

Further Reading

This document gives you the rudiments of X86-64 assembly, but there is much more to learn. To complete your compiler, you will almost certainly find that you need a few more instructions not listed here. Recommend that you peruse the list of instructions found in section 5.1 of Volume I of the Intel manual. Once you have identified the desired instruction, look up its details in Volume II.

Good luck, and now get to work!