Introduction to Compilers and Language Design

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at this website:
`http://compilerbook.org`

Draft version: December 12, 2017

Chapter 10 – Assembly Language

10.1 Introduction

In order to build a compiler, you must have a working knowledge of at least one kind of assembly language. And, it helps to see two or more variations of assembly, so as to fully appreciate the distinctions between architectures.

We have observed that many students seem to think that assembly language is rather obscure and complicated. Well, it is true that the complete manual for a CPU is extraordinarily thick, and may document hundreds of instructions and obscure addressing modes. However, it's been our experience that it is really only necessary to learn a small subset of a given assembly language (perhaps 30 instructions) in order to write a functional compiler. Many of the additional instructions and features exist to handle special cases for operating systems, floating point math, and multi-media computing. You can do almost everything needed with the basic subset.

We will look at two different CPU architectures that are in wide use today: X86 and ARM. The Intel X86 is a CISC architecture that has evolved since the 1970s from 8-bit to 64-bit and is now the dominant chip in personal computers, laptops, and high performance servers. The ARM processor is a RISC architecture began life as a 32-bit chip for the personal computer market, and is now the dominant chip for low-power and embedded devices such as mobile phones and tablets.

This chapter will give you a working knowledge of the basics of each architecture, but you will need a good reference to look up more details. We recommend that you consult the Intel Software Developer Manual [2] and the ARM Architecture Reference Manual [1] for the complete details.

10.2 Open Source Assembler Tools

A given assembly language can have multiple dialects for the same CPU, depending on whether one uses the assembler provided by the chip vendor, or other open source tools. For consistency, we will give examples in the assembly dialect supported by the GNU compiler and assembler, which are known as `gcc` and `as` (or sometimes `gas`.)

A good way to get started is to view the assembler output of the compiler for a C program. 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`, which indicates "source" file.

If you run `gcc -S hello.c -o hello.s` on this C program:

```
#include <stdio.h>

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

then you should see output similar to this in `hello.s`

```
.file   "test.c"
.data
.LC0:
        .string "hello %s\n"
.LC1:
        .string "world"
.text
.globl main
main:
        PUSHQ   %rbp
        MOVQ    %rsp, %rbp
        SUBQ    $16, %rsp
        MOVQ    %rdi, -8(%rbp)
        MOVQ    %rsi, -16(%rbp)
        MOVQ    $.LC0, %rax
        MOVQ    $.LC1, %rsi
        MOVQ    %rax, %rdi
        MOVQ    $0, %rax
        CALL    printf
        MOVQ    $0, %rax
        LEAVE
        RET
```

(There are many valid ways to compile `hello.c` and so the output of your compiler may be somewhat different.)

Regardless of the CPU architecture, 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 to assist the debugger. `.data` indicates the start of the data segment of the program, while `.text` indicates the start of the program segment. `.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.

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. The labels *per se* do not become part of the resulting machine code, they are just a means of referring to an address in the code.

Instructions are the actual assembly code like (`PUSHQ %rbp`), typically indented to visually distinguish them from directives and labels. Instructions in GNU assembly are not case sensitive, but we will generally uppercase them, for consistency.

To take this `hello.s` 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` appear because they were not declared as `.globl`.

As you are learning assembly language, take advantage of an existing compiler: write some simple functions to see what `gcc` generates. This can give you some starting points to identify new instructions and techniques to use.

10.3 X86 Assembly Language

X86 is a generic term that refers to the series of microprocessors descended from (or compatible with) the Intel 8088 processor used in the original IBM PC, including the 8086, 80286, '386, '486, and many others. Each generation of CPUs added new instructions and addressing modes from 8-bit to 16-bit to 32-bit, all while retaining backwards compatibility with old code. A variety of competitors (such as AMD) produced compatible chips that implemented the same instruction set.

However, Intel broke with tradition in the 64-bit generation by introducing a new brand (Itanium) and architecture (IA64) that was *not* backwards compatible with old code. Instead, it implemented a new concept known as Very Long Instruction Word (VLIW) [7] in which multiple concurrent operations were encoded into a single word. This had the potential for significant speedups due to instruction-level parallelism but represented a break with the past.

AMD stuck with the old ways and produced a 64-bit architecture (AMD64) that *was* backwards compatible with both Intel and AMD chips. While the technical merits of both approaches were debatable, the AMD approach won in the marketplace, and Intel followed by producing its own 64-bit architecture (Intel64) that was compatible with AMD64 and its own previous generation of chips. X86-64 is the generic name that covers both AMD64 and Intel64 architectures.

X86-64 is a fine example of CISC (complex instruction set computing). There are a very large number of instructions with many different submodes, some of them designed for very narrow tasks. However, a small subset of instructions will let us accomplish a lot.

10.3.1 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

These registers are *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. The names of the lower eight registers indicate the purpose for which each was originally intended: for example, %rax is the accumulator.

A Note on 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:

```
MOVQ %RSP, %RBP
```

`MOVQ` is the name of the instruction, and the percent signs indicate that `RSP` and `RBP` 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!

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 64 bits over the years, and so each register has some internal structure. 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.

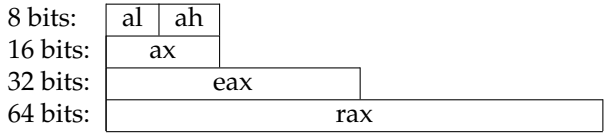

Figure 10.1: X86 Register Structure

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. However, most production compilers use a mix of modes: a byte can represent a boolean; a longword is usually sufficient for integer arithmetic,

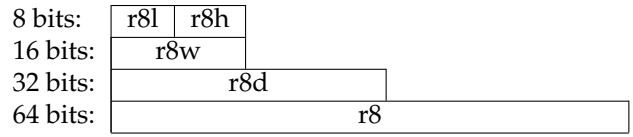

Figure 10.2: X86 Register Structure

since most programs don't need integer values above 2^{32}; and a quadword is needed to represent a memory address, enabling up to 16EB (exa-bytes) of virtual memory.

10.3.2 Addressing Modes

The `MOV` instruction moves data between registers and to and from memory in a variety of different modes. A single letter suffix determines the size of data to be moved:

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)

`MOVB` moves a byte, `MOVW` moves a word, `MOVL` moves a long, `MOVQ` moves a quad-word.[1] Generally, the size of the locations you are moving to and from must match the suffix. In some cases, you can leave off the suffix, and the assembler will infer the right size. However, this can have unexpected consequences, so we will make a habit of using the suffix.

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`, which the assembler translates into an absolute address or an address computation.
- An **immediate value** is a constant value indicated by a dollar sign such as `$56`, and has a limited range, depending on the instruction in use.
- A **register value** is the name of a register such as `%rbx`.
- An **indirect value** 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

[1] Careful: These terms are not portable. A *word* has a different size on different machines.

mode is important for manipulating stacks, local values, and function parameters, where the start of an object is given by a register.

- A **complex** address is of the form $D(R_A, R_B, C)$ which refers to the value at address $R_A + R_B * C + D$. Both R_A and R_B are general purpose registers, while C can have the value 1, 2, 4, or 8, and D can be any integer displacement. This mode is used to select an item within an array, where R_A gives the base of the array, R_B gives the index into the array, C gives the size of the items in the array, and D is an offset relative to that item.

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`
Complex	`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. There are some exceptions. 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.

In some cases, you may want to load the *address* of a variable instead of its value. This is handy when working with strings or arrays. For this purpose, use the `LEA` (**Load Effective Address**) instruction, which can perform the same address computations as `MOV`:

Mode	Example
Global Symbol	`LEAQ x, %rax`
Base-Relative	`LEAQ -8(%rbp), %rax`
Complex	`LEAQ -16(%rbx,%rcx,8), %rax`

10.3.3 Basic Arithmetic

You will need four basic arithmetic instructions for your compiler: integer addition, subtraction, multiplication, and division.

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 a little care, so that you don't accidentally clobber a value that you might want to use later. For example, you could translate `c = a+b+b;` like this:

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

The `IMUL` instruction is a little unusual, because multiplying two 64-bit integers results in a 128-bit integer, in the general case. `IMUL` 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. (This is implicit: %rdx is not mentioned in the instruction.)

For example, suppose that you wish to translate `c = b*(b+a);`, where `a`, and `b`, and `c` are global integers. Here is one possible translation:

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

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 given in the instruction. The quotient is placed in %rax and the remainder in %rdx. (If you want to implement the modulus instruction instead of division, just use the value of %rdx.)

To set up a division, you must make sure that both registers have the necessary sign-extended value. If the dividend fits in the lower 64 bits, but is negative, then the upper 64 bits must all be ones to complete the twos-complement representation. The CQO instruction serves the very specific purpose of sign-extending %rax into %rdx for divsion.

For example, to divide a by five:

```
MOVQ a,  %rax     # set the low 64 bits of the dividend
CQO               # sign-extend %rax into %rdx
IDIVQ $5          # divide %rdx:%rax by 5,
                  #          leaving result in %rax
```

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,b
MOVQ %rax, a
```

The instructions AND, OR, and XOR perform destructive *bitwise* boolean operations on two values. Bitwise means that the operation is applied to each individual bit in the operands, and stored in the result.
So, `AND $0101B $0110B` would yield the result `$0100B`. In a similar way, the NOT instruction inverts each bit in the operand. For example, the bitwise C expression `c = (a & ~b);` could be translated like this:

```
MOVQ a, %rax
MOVQ b, %rbx
NOTQ %rbx
ANDQ %rax, %rbx
MOVQ %rbx, c
```

Be careful here: these instructions *do not* implement logical boolean operations according to the C representation that you are probably familiar with. For example, if you define "false" to be the integer zero, and true to be any non-zero value. In that case, `$0001` is true, but `NOT $0001B` is `$1110B`, which is also true! To implement that correctly, you need to use CMP with conditionals described below.[2]

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.

[2] Alternatively, you could could use the bitwise operators as logical operators if you give `true` the integer value -1 (all ones) and `false` the integer value zero.

10.3.4 Comparisons and Jumps

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

JE	Jump if Equal
JNE	Jump if Not Equal
JL	Jump if Less
JLE	Jump if Less or Equal
JG	Jump if Greater
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  .L1
.L0:
        MOVQ $10, $rbx
        JMP  .L2
.L1:
        MOVQ $20, $rbx
.L2:
        MOVQ %rbx, 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. Labels like `.L0`, `.L1`, etc, can be generated by the compiler on demand.

10.3.5 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.

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 the appropriate number of bytes :

```
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
```

Note that, in 64-bit code, `PUSH` and `POP` are limited to working with 64-bit values, so a manual `MOV` and `ADD` must be used if it is necessary to move smaller items to/from the stack.

10.3.6 Calling a Function

Prior to the 64-bit architecture described here, a simple stack calling convention was used: arguments were pushed on the stack in reverse order, then the function was invoked with `CALL`. The called function looked for the arguments on the stack, did its work, and returned the result in `%eax`. The caller then removed the arguments from the stack.

However, 64-bit code uses a register calling convention, in order to exploit the larger number of available registers in the X86-64 architecture. [3] This convention is known as the **System V ABI** [3] and is written out in a lengthy technical document. The complete convention is quite complicated, but this summary handles the basic cases:

Figure 10.3: Summary of System V ABI Calling Convention

- The first six integer arguments (including pointers and other types that can be stored as integers) are placed in the registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9, in that order.
- The first eight floating point arguments are placed in the registers %xmm0-%xmm7, in that order.
- Arguments in excess of those registers are pushed onto the stack.
- If the function takes a variable number of arguments (like `printf`) then the %rax register must be set to the number of floating point arguments.
- The return value of the function is placed in %rax.

In addition, we also need to know how the remaining registers are handled. A few are **caller saved**, meaning that the calling function must save those values before invoking another function. Others are **callee saved**, meaning that a function, when called, must save the values of those registers, and restore them on return. The argument and result registers need not be saved at all. Figure 10.4 shows these requirements.

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 `%rax` register.

[3] Note that there is nothing *stopping* you from writing a compiler that uses a stack calling convention. But if you want to invoke functions compiled by others (like the standard library) then you need to stick to the convention already in use.

Here is an example. The following C program:

```
int x=0;
int 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
      MOVQ  y,    %rsi  # second argument in %rsi: y
      MOVQ  $0,   %rax  # there are zero float args

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

      CALL  printf      # invoke printf

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

      MOVQ  %rax, x     # save the result in x

      RET               # return from main function
```

Figure 10.4: System V ABI Register Assignments

Register	Purpose	Who Saves?
%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

10.3.7 Defining a Leaf Function

Because function arguments are passed in registers, it is easy to write a **leaf function** that computes a value without calling any other functions. For example, code for the following function:

```
square: function integer ( x: integer ) =
{
    return x*x;
}
```

Could be as simple as this:

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

Unfortunately, this won't work for a function that wants to invoke other functions, because we haven't set up the stack properly. A more complex approach is needed for the general case.

10.3.8 Defining a Complex Function

A complex 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 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** for this 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 Figure 10.4, each function must preserve the values of %rsp, %rbp, %rbx, and %r12-%r15 when it completes.

Here is 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

Figure 10.5: Example X86-64 Stack Layout

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 arguments and variables 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-minor function defined as follows:

```
compute: function integer
         ( a: integer, b: integer, c: integer )
{
    int x, y;
    x = a+b+c;
    y = x*5;
    return y;
}
```

A complete translation of the function is on the next page. The code given is correct, but rather conservative. As it turned out, this particular function didn't need to use registers `%rbx`–`%r15`, so it wasn't necessary to save and restore them. In a similar way, we could have kept the arguments in registers without saving them to the stack. The result could have been computed directly into `%rax` 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 (probably) not be very efficient if 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. Likewise, a statement that computes a value must save it back to a local variable, because it does not know beforehand whether the local will be used as a return value. We will explore these issues later in Chapter 12 on optimization.

10.4 ARM Assembly

Coming in second edition!

```
.globl compute
compute:
#################### preamble of function sets up stack
pushq %rbp          # save the base pointer
movq  %rsp, %rbp    # set new base pointer to rsp

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 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 register.
movq  $5, %rcx          # load 5 into a register
movq  %rbx, %rax        # move argument in rax
imulq %rcx              # multiply them together
movq  %rax, -40(%rbp)   # store the result in local 1 (y)

movq  -20(%rbp), %rax   # move local 1 (y) into the result

#################### 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
```