CSE351 A, Grossman, Midterm Exam

Taylor Blau[1]

Spring, 2018

[1]Material is compiled from the lecture notes, the readings in *Computer Systems* (3rd edition), Bryant, O'Hallaron, and my own notes.

Contents

List of Figures

Chapter 1

Memory & Data

1.1 Numeric Bases

Numbers can be written in an arbitrary choice of base. Our numbers are written as decimals, meaning that they are written in base 10. Given a number $n_{b=10}$, we denote the i-th digit, n_i, and express n as follows:

$$n = \sum_{i=0}^{j-1} n_i 10^i \tag{1.1}$$

1.1.1 Common Bases

Binary A binary number, subscripted n_b, is written with the digit set 0, 1. The number is expressed similarly as to the above, rewriting the 10 as a 2.

Octal Similarly, an octal number is written as $n_{b=8}$ and uses digits 0...7. Again, the number is written according to powers of 8, as opposed to 2 or 10.

Hexadecimal Hexadecimal is the most interesting case of the above, since it requires additional digits beyond 0...9. As such, we add the following additional characters to 0...9 to create sixteen unique digits: $0, \ldots, 9, A, \ldots F$.

1.1.2 Base Conversion

To convert from one base a to another base b, we apply the following process:

1. Convert n_a to a commonly understood base, for convenience.

2. Find the largest power of base b that does not exceed the number n_a.

3. Mark the maximum number of bits in that position. For example, in binary, this would always be 1, but in decimal, hexadecimal, or octal this may be a number from $0 - 9$, $0 - f$, or similar.

4. Decrease the target number n_a by the number of bits added, denoting it n'_a.

5. See step (2).

1.1.3 Special Conversion

An interesting conversion occurs between the two most commonly used bases in computer software: base 16 (hexadecimal), and base 2 (binary).

Notably, we may use the following conversion table to rewrite between the two:

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Binary to Hexadecimal

When converting from binary to hexadecimal, note that each hexadecimal digit occupies four bits of binary. As such, pad the left-hand side of the binary digit with zeros, and convert four-bit chunks of the binary sequence, replacing it with matches in hexadecimal.

Hexadecimal to Binary

When converting from hexadecimal to binary, apply the same tactic in reverse, replacing each single-character hexadecimal digit with the four binary digits that it is encoded as in the table above.

1.1.4 Meaningful Bits

What do a particular sequence of bits refer to? By themselves, bits do not mean anything: they can have an arbitrary meaning given an arbitrary encoding. We derive meaning from a sequence of bits by examining them with a known encoding. More later.

1.2 Memory

1.2.1 Memory Organization

Conceptually, one of the tasks of an operating system is to organize memory into a *virtual address space* that is consume-able by a program running in user-space. The operating system creates the effect of memory being a continuous array of bytes, but this is only an effect.

Each space in memory is referred to by a *memory address*. The domain of all possible addresses is referred to as an *address space*.

1.2.2 Bounding the Address Space

To bound the size of an address space, we introduce a new concept, called the *word size*. The word size is the maxmimum number of bits available to a single memory address. Though it was the case that a common address size was 32-bits when the course materials were published, most personal computers have a word size of 64-bits.

$$\text{Word Size} \quad \equiv \quad \text{Address Size} \quad \equiv \quad \text{Register Size}$$

So, given a word size of w-bits, our address space contains 2^w addresses. general it is true that w bits produces)

1.2.3 Memory Access

Notably, memory addresses refer to *bytes* in memory, not the individual bits. We obtain information from individual bits by accessing a `byte`, and then performing bitwise operations on that bit to access individual bits.

The address of a word is the address of the first byte of that word. So a 64-bit integer (1 word) is addressed by the address referring to the very first byte in that integer (more on this later).

Memory Alignment

Modern CPU architectures make optimizations for reading specific sections of memory at a time. For example, reading address `0x4` will be slower than reading address `0x8`. Why? This is because the address `0x4` is *misaligned*, meaning that its first bit does not occur at a multiple of the word-size.

In other terms, a primitive data structure of K bytes must be aligned to an address that is a multiple of K.

1.2.4 Data Representations

In the C programming language, there are several built-in datatypes that have a given size. They are listed as follows:

Java Type	C Type	32-bit	64-bit
`boolean`	`bool`	1	1
`byte`	`char`	1	1
`char`		2	2
`short`	`short int`	2	2
`int`	`int`	4	4
`float`	`float`	4	4
	`long int`	4	8
`double`	`double`	4	8
`long`	`long`	8	8
	`long double`	8	16
(reference)	`void *`	4	8

1.2.5 Byte Ordering

How should multiple-byte data structures be represented in memory? Consider an `int` that has 4 bytes. Which bytes should come first? The 8 bits within a byte are in a fixed ordering (most to least significant from left to right), but the bytes themselves are arbitrary. There are two schools of thought, both puns of Swift's *Gulliver's Travels*:

Big-endian Consume the big end first. This means that the most significant *byte* will appear at the lowest memory address, and the higher memory addresses yield bytes of lesser significance. Consider the number `0xbaddecaf`, starting at memory address $0x0$. Then:

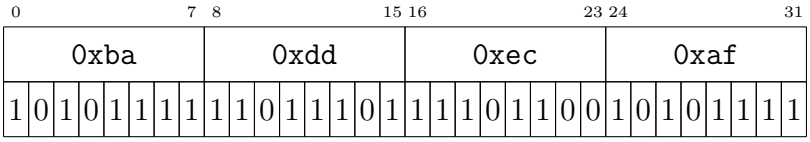


Figure 1.1: `0xbaddecaf` in big-endian byte order.

Little-endian Consume the little end first. This means that the most significant *byte* will appear at the highest (right-most) memory address, and the lower memory addresses yield bytes of higher significance. Consider the number `0x0decaf`, starting at memory address $0x0$. Then:

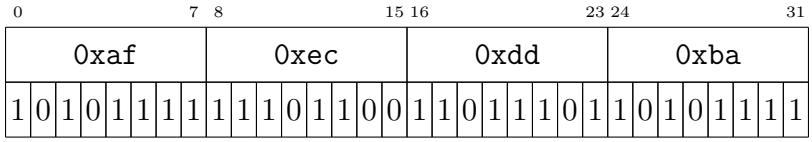


Figure 1.2: `0xbaddecaf` in little-endian byte order.

An advantage to little-endian byte ordering is that if you cast to a larger datatype (read: wider), the underlying bit-level representation need not change provided that the encoding remains the same (i.e., `int` to `long int`, but not `float` to `double`).

1.2.6 Memory Manipulation in C

There are two key operators for dealing with memory and memory locations in C:

`&` Addressing operator. Takes the address of its operand.

`*` Dereferencing operator. Loads from memory at the address of its operand.

Additionally, `*` can be used in the type declaration of a variable. Consider the following program and its meaning:

```
// Declare a new variable of type 'int *' (pointer to int) called 'ptr'.
int *ptr;

// Declare two variables, x and y and the values 5 and 2 to them,
// respectively.
int x = 5;
int y = 2;

// Assign ptr the address of variable x in memory.
ptr = &x;

// De-reference ptr and add 1 to it.
y = 1 + *ptr;
```

1.2.7 Pointer Arithmetic

Pointer arithmetic works like normal arithmetic with two key differences:

1. Its operands can be optionally treated as memory locations (which are integers, see previous sections).

2. Its operands are *scaled* by the size of the data structure of which they are referring to.

Considering (2), we see that this means the following:

```
int arr[2];

arr[1] = 5;
// Add "1" to the address of "arr", where 1 is scaled by sizeof(int), making
// this expression in-effect equivalent, "arr + 1*sizeof(int)".
*(arr+1) = 6;
```

1.2.8 Arrays in C

Arrays in C are contiguous blocks of memory and each element is aligned one after another. For example, the type `int [4]` is a 4-wide array of integers, meaning that 16 total bytes are allocated.

Array access (done using the `[]` operator) is not bounds-checked. In practice, this means that for an array of length n, you can access not only elements $0 \ldots (n-1)$, but also n, $n+1$, -1, and so on. Each value is taken as a relative offset from the starting address of the memory.

1.2.9 Strings in C

Why do we spell "string" as "`char *`"? In fact, a string *is* an array of characters. A string is allocated as a contiguous block of 1-byte characters, and the address of a string is the address of the first character contained within it.

Notably, strings in C are `NUL` terminated, which means they end with a byte `0x00`.

1.3 Bit-level Manipulations

1.3.1 Boolean Algebra

We introduce the notion of Boolean Algebra, as coined by George Boole in the 19th century. Operands are either 0 or 1, and the following operations are defined:

Operator	C-Spelling	Logical Symbol	Meaning
AND	&	$\wedge$	1 when both operands are 1.
OR	|	$\vee$	1 when either operand is 1.
NOT	!	$\neg$	1 when operand is 0, vice-versa.
XOR	^	$\oplus$	1 when operands are un-equal.

Operations can occur on arbitrary bit vectors, $\vec{b}$, provide that each operand is of the same length. The operators above are defined on a bit-vector of length 1, but operation of multi-bit $\vec{b}$'s is the pair-wise application of that operator.

1.3.2 Bit-level operations in C

Bit-level operations can occur on any integer-like data type in C, including: `long`, `int`, `short`, and `unsigned`.

1.3.3 Logical operators in C

In C, logical operators can be applied to *any* data type, not just integers. The boolean truth value of an arbitrary data-type is defined as follows:

$$\texttt{bool}(x) = \begin{cases} \texttt{false} & x = 0 \\ \texttt{true} & x \neq 0 \end{cases}$$

1.3.4 Bitwise encoding

In many programming languages, a sum type or structure would be encouraged instead of a single integer. C prefers the later, and so we study it: there are several ways that we can encode data into an integer by modifying the individual bits of that integer.

"One-hot" encoding Turn one unique bit on per unique encoding of a particular value.

Packed encoding Pack "sections" of data throughout the integer, and apply encoding techniques recursively.

Chapter 2

Integers

2.1 Encoding Styles

There are two types of integers that hardware supports, and we begin our study there. They are outlined in the subsequent sections, and enumerated here:

Unsigned Represents positive values from $0 \ldots 2^w - 1$.

Signed Represents positive and negative values $-(2^{w-1}) \ldots 2^{w-1} - 1$.

2.1.1 Unsigned

We encode unsigned integers as normal binary numbers, and apply addition rules as previously.

2.1.2 Signed

How do we encode numbers that have a sign? There are several approaches, which we enumerate below:

Sign and Magnitude In this approach, we reserve the most significant bit (M.S.B.) to encode whether the number is negative or not, i.e., 1 for negative, and 0 for positive. This approach, while reasonable, presents problems for implementation: there are now two encodings of zero (± 0), and addition breaks in strange ways (i.e., two negative numbers added together produce a positive number).

Two's Complement By far the most common encoding of signed integers is called "Two's Complement". This approach takes the most significant bit, normally weighted 2^{32}, and instead weights it -2^{32}, leaving *all other encodings unchanged*. This alone produces the correct effect and allows us to encode integers in the range listed above without introducing the problems that the "Sign and Magnitude" approach does.

2.2 Notable Values, Functions

$$B2U_w(\vec{x}) \equiv \sum_{i=0}^{w-1} x_i 2^i$$

$$B2T_w(\vec{x}) \equiv -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

$$T2U_w(\vec{x}) \equiv \begin{cases} x + 2^w & x < 0 \\ x & x \geq 0 \end{cases}$$

$TMin_w$ A single 1 in the most-significant bit (-2^w).

$TMax_w$ All 1s, except in the most-significant bit ($2^{w-1} - 1$).

$UMax_w$ All 1s ($2^w - 1$)

$UMin_w$ All 0s (0).

2.2.1 Logical, Arithmetic Shifts

We introduce two new operators for dealing with integer-like datatypes in C: `<<` and `>>`.

`<<` shifts all bits n positions to the left, filling the right-most "empty" bits with 0, and discarding the left-most bits that are shifted off.

`>>` shifts all bits n positions to the right, filling the left-most "empty" bits with either the (1) sign of the integer, if the datatype is signed, or (2) zeros otherwise.

2.2.2 C Peculiarities

By default, integer literals are treated as signed integers, unless they are suffixed with `u`. A mixed expression is evaluated by casting all signed values to unsigned. This can cause surprising behavior for comparison between signed and unsigned numbers.

2.2.3 Fast Multiplication

Note that `x << n` is the same as `x = x *` 2^n. We obtain two forms for fast multiplication by another number, p, written in binary.

1. `(x << n) + (x << (n-1)) + ... + (x << m)` Where m and n are the ending and starting indices of the 1s in a number.
2. `(x<<(n+1)) - (x<<m)`

Chapter 3

Floating Point

We now have knowledge of how to represent "large" numbers, like integers, longs, and etc. But, the question remains: how do we represent very small number precisely? To answer this, we turn the IEEE 754 specification of Floating Point numerals.

3.1 General Form

A floating point number is made up, generally, of either 32- or 64-bits. They are categorized as follows:

Sign bit The most significant bit is 1 for negative numbers, and 0 for positive numbers.

(Weighted) exponent The weighted exponent E is multiplied as $2^{E-\text{bias}}$.

Mantissa (significand) The number x, where x is used as $1.x$. Occupied in remaining bits.

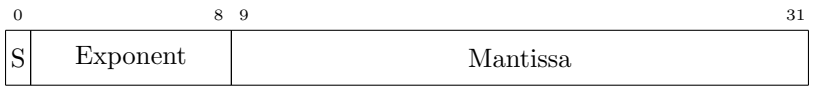


Figure 3.1: Generalized form of a IEE 754 single-precision float

Together, a floating point F is written as:

$$F = (-1)^S \cdot 1.M \cdot w^{E-\text{Bias}}$$

3.1.1 Biased Exponent

In general, the bias of an exponent is written as $b = 2^{w-1} - 1$, where w is the width of the exponent field, which is (as above) 8 for single-precision floating points. Therefore, if the bit pattern is `0xfe`, the b would therefore be `0x7f` (equal to 127), instead of `0xfe`, since we subtract 127 before including it in F.

Therefore, to represent an exponent 0, we use the bit-vector `0x7f`.

3.2 Forms of Floating Point

3.2.1 Normalized Form

As described above. The smallest normalized IEE 574 single-precision floating point is $\pm 1 \cdot 2^{-126}$.

3.2.2 Denormalized Form

Floating point numbers have a "gap" around values *very* close to zero. How do we close that gap? The IEE 574 standard defines several specialized values which are meant to create additional values very close to zero. They are defined as follows:

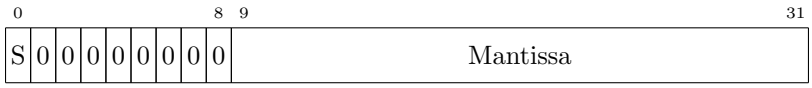


Figure 3.2: Generalized form of a IEE 754 single-precision float (denormalized)

There are two cases for the Mantissa:

All zeros The signed value ± 0.

Anything else A float $F = (-1)^S \cdot 0.M \cdot 2^{-126}$ (Note that the exponent is -126 *even though* the $\vec{E} =$ `0000 0000`.)

Therefore, the smallest denormalized IEE 574 single-precision floating point is $\pm 2^{-149}$.

There are some other special cases for (denormalized) floating point numbers, if $\vec{E} =$ `0xffff ffff`:

$M = \vec{0}$ $\pm\infty$.

$M \neq \vec{0}$ `NaN`.

3.3 Floating Point Arithmetic

3.3.1 Addition

To add two floating point numbers, apply the following technique:

1. For the sign S and the mantissa M, align the two numbers with $F1$ on the left and $F2$ on the right separated by $E_1 - E_2$, then add.
2. The exponent E is taken as E_1.
3. Make final adjustments:

(a) If $M \geq 2$, shift M right and increment E.

(b) If $M < 1$, shift M left k positions and $E' = E - k$.

(c) Overflow or underflow if E is out-of-range.

(d) Round M to fit the precision.

3.3.2 Multiplication

The exact result is given as (for $F_1 = (-1)^{S_1} \cdot M_1 \cdot 2^{E_1}$ and $F_2 = (-1)^{S_2} \cdot M_2 \cdot 2^{E_2}$):

$S = S_1 \hat{} S_2$.

$M = M_1 \times M_2$.

$E = E_1 + E_2$.

Then, make adjustments as during addition:

1. If $M \geq 2$, shift M right and increment E.
2. Overflow or underflow if E is out-of-range.
3. Round M to fit the precision.

3.4 Floating Point in Practice

A few surprising things can occur in practice when doing floating point computations.

3.4.1 `NaN`, $\pm\infty$

For one, if the exponent overflows, a value of $\pm\infty$ will occur. Once `NaN` or $\pm\infty$ is introduced, they are considered "sticky", which is to say that any other computations will evaluate to $\pm\infty$ or `NaN` if either of those values are involved in computations up to that point.

3.4.2 Rounding Issues

There are three common rounding issues that can occur during floating point operations:

No exact representation of fractions Fractions that are of infinite length (with non-zero values at the end) cannot be accurately represented in floating point. For example, the following code can occur in practice:

```
assert((1.0 / 3) * 3 != 0);
// OK
```

Limited Mantissa A limited mantissa leads to a loss of precision. In general, a very small number will be rounded away when being added to a very large number, so it is best to do small computations first. Consider these two examples:

```
static void broken() {
  float x = huge_number;
  for (int i = 0; i < large; i++) {
    x += small_number;
  }
}
```

```
static void maybe_broken() {
  float x = 0;
  for (int i = 0; i < large; i++) {
    x += small_number;
  }
  x += huge_number;
}
```

No Distributivity A natural consequence of the limited mantissa is that distributivity is broken for a similar reason.

3.5 Floating Point in C

C offers several values of precision:

`float` single-precision (32-bit).

`double` double-precision (64-bit).

`long double` "double double"-precision (128-bit).

Including the header `#include <math.h>` provides the `INFINITY` and `NAN` constants. Equality (`==`) is allowed, but should not be used.

Several conversions are possible, and some do and do not overflow:

`int` → `float` Might be rounded (only 23-bit), but overflow is impossible.

`int`**(or** `float`**)** → `double` Exact conversion in memory.

`long` → `double` Depends on the word-size, (32-bit is exact, but 64-bit is rounded.)

`double` → `float`**(or** `int`**)** Truncates the fractional part, and is undefined when a value is out-of-range or is `NaN`.

Chapter 4

x86_64 Assembly

4.1 Instruction Set Architecture

What is an Instruction Set Architecture? The Instruction Set Architecture (abbrev. "I.S.A.") defines the following:

1. The system state (e.g., registers, memory, program counter, etc).
2. The instructions that the CPU can execute.
3. The semantic meaning of those instructions on the state of the system.

There are two schools of thought on how an I.S.A. should be defined:

Complex Instruction Set Computing (CISC) Should add more elaborate and specialized instructions as needed. There are a lot of tools to be able to use, but the compiler and CPU must support them. (x86_64 for example is a CISC-like I.S.A., but mostly only a small subset of instructions are encountered with most Linux programs.)

Reduced Instruction Set Computing (RISC) Should keep instruction set small and regular, meaning that it is easier to build fast hardware, and software will do the complicated operations by composing many simple ones. May be slower in effect, but that's an OK trade-off.

So, an architecture is the parts of a processor that one needs to understand in order to write assembly code. An *micro-architecture*, on the other hand, is the implementation of that architecture, and is out-of-scope for our present discussion.

4.1.1 A Programmer's view of x86_64

From the programmer's perspective, there are several visible pieces of state worth understanding:

The Program Counter (abbrev. "PC", or `%rip`) The address of the next instruction.

Registers Used to store program data, and as operands for other assembly instructions.

Condition codes Stores status about the most recent arithmetic operation.

Memory Byte-addressable array, contains the code and user-data, as well as the "stack".

4.2 x86_64 "Data Types"

There are no aggregate data-types such as arrays or structures in assembly, only contiguously allocated bytes in memory. There are integer-like data types of 1, 2, 4, and 8 bytes which can be data values, or addresses.

Also of note, is that there are two common sets of syntax, the "AT&T syntax", as well as the "Intel syntax". For our discussion, we'll mostly stick with Intel, but it's important to know which you are reading.

4.3 x86_64 Registers

A register is a location in the CPU (notably, *not* in memory) that stores a small amount of data, and can be access once per clock cycle. Registers have names that start with a %.

Here is the layout of a single register:

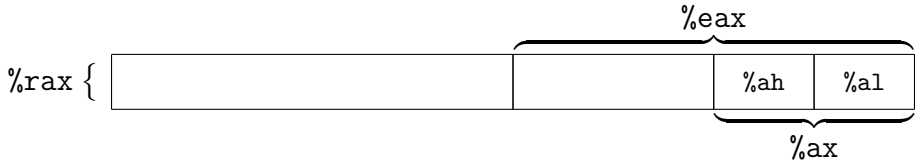
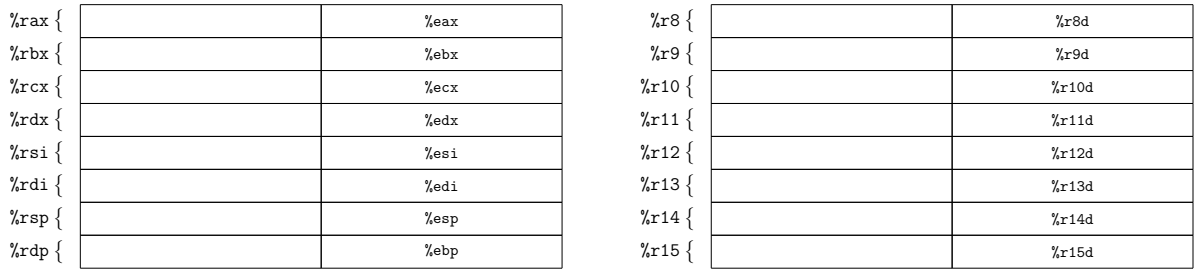


Figure 4.1: A single 64-bit wide register

Here is a layout of all registers:



4.4 x86_64 Instructions

4.4.1 Instruction Types

There are three basic kinds of instructions in x86_64 assembly:

Data Transfer Two kinds:

1. Load: load a value from memory into a register.
2. Store: store a value from a register into memory.

Ariethmetic Operations Add two operands together, logical shift, etc.

Control Flow Jump to a location, conditionally jump, etc.

4.4.2 Operand Types

Similarly, there are three kinds of operand types:

Immediate A constant integer, written in the same style as C, but prefixed with a literal `$`, (e.g., `$0x400`).

Register A named register, written as `%rax`.

Memory An address in memory, written with "()" surrounding. There are various addressing modes, as described below.

4.4.3 Addressing Modes

Indirect `(R)`, where the data in register "R" specifies the memory address.

Displacement `D(R)`, where the data in register "R" is *offset* from location "D" in memory.

General Form `D(`R_b`,`R_i`,`S`)`, where R_b is the base register, R_i is the index register, S is the scale factor (either 1, 2, 4, or 8), and D is the constant displacement factor. As a closed form, this is $R_b + R_i * S + D$. There are a few special cases, listed below:

- $D(R_b, R_i)$ (implicit $S = 1$).
- (R_b, R_i, S) (implicit $D = 0$).
- (R_b, R_i) (implicit $D = 0$, $S = 1$).
- $(, R_i, S)$ (implicit $R_b = 0$, $D = 0$).

4.5 Common Instructions

Several instructions are suffixed with an operand size, the available ones of which we define here:

suffix	name	size (bytes)	size (bits)
b	byte	1	8
w	word	2	16
l	long	4	32
q	quad-word	8	64

4.5.1 Arithmetic Instructions

There are several unary arithmetic instructions, defined as follows:

`inc_` Increment target by 1.

`dec_` Decrement target by 1.

`neg_` Negate target (`-`).

`not_` Bitwise complement target (`~`).

There are several binary arithmetic instructions, defined as follows:

`add_` Add source to destination.

`sub_` Subtract source from destination.

`imul_` Multiply destination by source.

`shl_` Logical left-shift.

`shr_` Logical right-shift.

`sar_` Arithmetic right-shift.

`xor_` XOR into destination.

`and_` AND into destination.

`or_` OR into destination.

Each of these sets condition codes as side effects of computation. There are four condition codes, `CF`, equal to the carry out from the most significant bit (unsigned overflow), `ZF`, equal to 1 if the computation is equal to zero, `SF`, equal to 1 if the computation is less than zero (undefined if unsigned), and `OF`, equal to 1 if there is a two's complement (signed) overflow.

4.5.2 Movement Instructions

`movq` The `mov_` instruction takes data from a source to a destination. Source and destinations can be any combination of immediate, register, or memory address *except* memory to memory. An intermediate register must be used instead.

As an aside, there are two other related movement instructions, `movz` and `movs`, which zero- or sign- extend from *source* (register, or memory) to a *destination* (memory only).

`leaq` The `lea_` instruction takes an address expression as its source, and another register as its destination. It computes the address in source and assigns it to the destination register, but *does not* load that address from memory.

4.5.3 Explicit Condition Codes

One way previously mentioned to set condition codes is by treating them as a side-effect of doing arithmetic computation. Another way to set them is to run any arbitrary instruction that explicitly sets their value. For example:

`cmp_ s1, s2` Sets condition flags based on the computation $s_2 - s_1$, as follows:

`CF=1` If the carry out from the MSB (unsigned comparison).

`ZF=1` If $a = b$.

`SF=1` If $(b - a) < 0$, under a signed comparison.

`OF=1` If there's a two's complement overflow.

`test_ s2, s1` Sets condition flags based on the computation `s2&s1`, as follows:

`ZF=1` If `a&b=0`.

`SF=1` If `a&b<0`, under a signed comparison.

4.5.4 Jump Operation(s)

Based on the condition codes, there are several jump instructions that can occur, and they are listed below as follows:

Instruction	Condition	Description
`jmp .label`	-	Unconditional jump.
`je .label`	ZF	Jump if equal or zero.
`jne .label`	$\tilde{\text{Z}}$F	Jump if not equal or not zero.
`js .label`	SF	Jump if negative.
`jns .label`	$\tilde{\text{S}}$F	Jump if non-negative.
`jg .label`	...	Jump if (signed) greater.
`jge .label`	...	Jump if (signed) greater-than or equal-to.
`jl .label`	...	Jump if (signed) less.
`jle .label`	...	Jump if (signed) less-than or equal-to.
`ja .label`	...	Jump if above (unsigned "¿").
`jb .label`	...	Jump if below (unsigned "¡").

4.5.5 Set Operation(s)

Based on the condition codes, there are several "`set*`" operations that set the low-order byte of `dst` to either 0 or 1 based on condition codes.

Instruction	Condition	Description
`sete .label`	`ZF`	Set if equal or zero.
`setne .label`	`~ZF`	Set if not equal or not zero.
`sets .label`	`SF`	Set if negative.
`setns .label`	`~SF`	Set if non-negative.
`setg .label`	...	Set if (signed) greater.
`setge .label`	...	Set if (signed) greater-than or equal-to.
`setl .label`	...	Set if (signed) less.
`setle .label`	...	Set if (signed) less-than or equal-to.
`seta .label`	...	Set if above (unsigned "¿").
`setb .label`	...	Set if below (unsigned "¡").

4.5.6 Choosing conditionals

	`cmp a, b`	`test a,b`
`je .label`	`b == a`	`b&a == 0`
`jne .label`	`b != a`	`b&a != 0`
`js .label`	`b-a < 0`	`b&a < 0`
`jns .label`	`b-a >= 0`	`b&a >= 0`
`jg .label`	`b > a`	`b&a > 0`
`jge .label`	`b >= a`	`b&a >= 0`
`jl .label`	`b < a`	`b&a < 0`
`jle .label`	`b <= a`	`b&a <= 0`
`ja .label`	`b > a`	`b&a > 0U`
`jb .label`	`b < a`	`b&a < 0U`

4.6 Loops

4.6.1 Compiling `while` loops

Consider the following C code and its assembly analogue:

4.6.2 Compiling `do ... while` loops

Consider the following C code and its assembly analogue:

```
while (i != 0) {
  // body
}
// done
```

(a) Traditional `while` loop in C.

```
loopTop:
  testq %rax, %rax
  je    loopDone
  ...
  jmp   loopTop
loopDone:
  ...
```

(b) `while` loop compiled to assembly.

```
do {
  // body
} while (i != 0);
// done
```

(a) Traditional `do...while` loop in C.

```
loopTop:
  ...
  testq %rax, %rax
  jne   loopTop
loopDone:
  ...
```

(b) `do...while` loop compiled to assembly.

4.6.3 Compiling `for` loops

To compile `for` loops, we first compile them to `while` loops, and then apply the translation as above.

```
for (Init; Test; Update)
  Body
```

(a) Traditional `for` loop in C.

```
Init;
while (Test) {
  Body;
  Update;
}
```

(b) Traditional `for` loop written as `while`-equivalent in C.

```
loopInit:
  ...
loopTop:
  testq %rax, %rax
  je    .loopDone
  ... # (body, update)
loopDone:
  ...
```

(c) Traditional `for` loop compiled to assembly.

4.7 `switch` statement

Switches are a complicated structure in C. They have multiple branches, can have fall-through's, empty case(s), and etc. The way that they are implemented in assembly is as follows:

1. Construct a "jump table" referring to the location in the executable binary where each case "arm" goes.
2. Perform any initialization steps (setting up the stack, declaring any variables, etc.)
3. Ensure that the value being switched upon is within bounds.
4. Lookup the location of that item within the jump table.
5. Jump to that location, and continue execution.

Here is a switch statement (with details elided) in C:

```
long switch_ex(long x, long y, long z) {
  long w = 1;
  switch (x) {
    // ...
  }
  return w;
}
```

and the jump table generated:

```
.section .rodata
  .align 8
.L4:
  .quad .L8 # x = 0, .L8 is default
  .quad .L3 # x = 1
  .quad .L5 # x = 2
  .quad .L9 # x = 3
  .quad .L8 # x = 4, missing case
  .quad .L7 # x = 5, fallthrough/empty
  .quad .L7 # x = 6
```

and then the assembly translation of step (3) and (4):

```
.switch_eq
  movq %rdx, %rcx
  cmpq $6, %rdi
  ja   .L8            # jump above to catch negative cases, goto default
  jmp  *.L4(,%rdi,8) # indirect jump
```

Chapter 5

Procedures

5.1 Introduction

Procedures are how humans organize code. But to effectively implement procedures, we need to compile several important constructs from C to Assembly. Namely, those constructs are:

Passing control How do we send the running execution of a program to a different function, and then back to where we left off when that function returns?

Passing data How do we send arguments to a different function, and receive its return value?

Memory management How do we implement lexical scoping rules?

5.2 Memory Layout

Here is a simplified view of how memory is laid out when an executable is loaded from disk into memory:

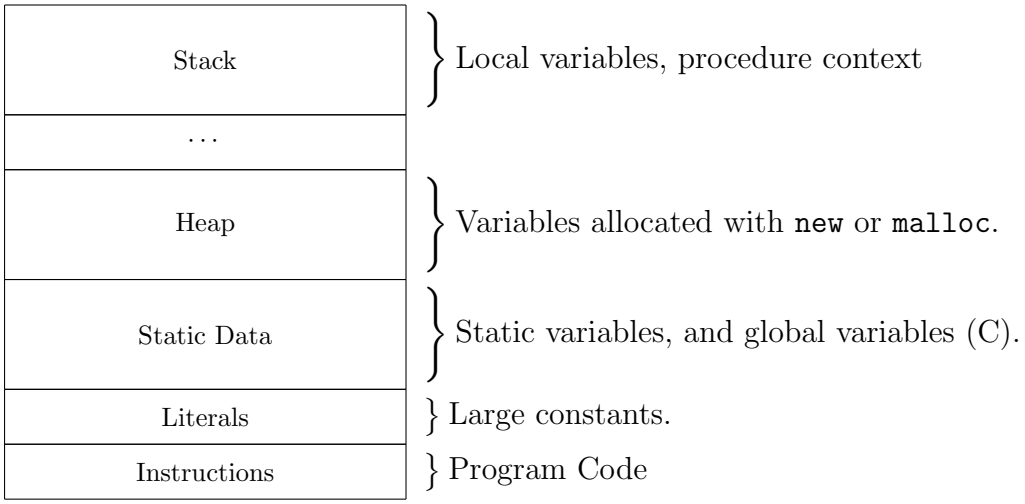


Figure 5.1: Simplified view of memory layout

5.3 Stack Structure

5.3.1 Stack Discipline

In an ELF x86_64 binary, the stack "grows downward", meaning that the "last" item on the stack is at its highest address, and new items are added to the "top" of the stack, which is at lower memory addresses than the "bottom". Programmers, right?

Ugh. Anyhow, the register `%rsp` contains the address of the top element on the stack.

5.3.2 `pushq`, `popq`

When wanting to temporarily discard–and later recall–data from registers you, can use the `pushq` and `popq` operators to do so.

For example:

`pushq`

`pushq dst` loads the value given at `%rsp`, stores that value into `dst`, and then increments the stack pointer by 8 (`addq 8, %rsp`).

`popq`

`popq dst` stores the value given as `dst` at `%rsp` in memory, and then decrements the stack pointer by 8 (`subq 8, %rsp`).

5.4 Calling Conventions

5.4.1 Passing Control

Recall that one of our goals was "Passing Control" (as described below):

Passing control How do we send the running execution of a program to a different function, and then back to where we left off when that function returns?

How do we do this? What does it mean? We have to provide two key pieces of information: where to jump *to*, and where to *return* to once the sub-procedure is done. In practice, this involves two non-surprising instructions:

`call label` First, push the return address onto the stack. The return address is the *next address* after the `jump`. Then, resume execution of the program at `label`.

`ret` First, pop the return address off of the stack, then jump to that address.

5.4.2 Passing Data

x86_64 calling conventions dictate that the first six arguments to a function are given as registers. Those registers are, in-order: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and, `%r9`. A pneumonic for remembering this is "**D**ianne's **s**ilk **d**ress **c**ost $**89** dollars."

By convention, the result of a procedure call is stored in `%rax`. This choice is arbitrary, but has some implications: the caller must know to save the value of `%rbx` if it will be needed later, since the callee knows that it can overwrite whatever is already in there.

5.4.3 Managing Local Data

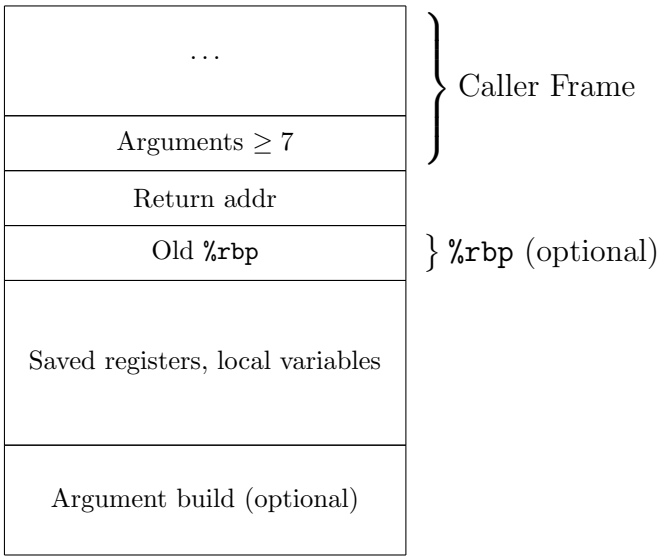


Figure 5.2: An x86_64 stack frame

5.5 Register Saving Conventions

x86_64 has a limited number of registers, which means that if one function is occupying space in many of those registers, and then wishes to call another function that wishes to operate over those registers, it will be unable to do so. As such, x86 defines a number of *register saving conventions* that are useful in determining who should save or not save certain registers.

5.5.1 Caller-saved registers

It is the caller's responsibility to save any data in that register before the control flow is relinquished. The callee can freely use any caller-saved registers. The caller will later restore values from that register if they need them later.

Here is a list of caller-saved registers:

`%rax` The return value, is caller-saved, and can be modified.

`%rdi` ... `%r9` Argument registers, which are caller saved and restored, and can be modified by the procedure.

`%r10`, `%r11` Caller-saved and restored, can be modified by the procedure.

5.5.2 Callee-saved registers

It is the callee's responsibility to save any data in that register before assigning into them. The caller can freely assume that any callee-saved registers will persist their data before and after a control flow transfer. The callee must save the data in their stack frame before changing the value in a register, and will necessarily have to `popq` from the stack before returning at the end of the function.

Here is a list of callee-saved registers:

`%rbx`, `%r12`, `%r13`, `%r14` Callee-saved; the caller must save and restore these registers.

`%rbp` Callee-saved; the caller must save and restore this register, and it can (optionally) be used as the frame pointer.

`%rsp` Special form of callee-saved; the caller does *not* have to restore this register, as it is done so automatically by the CPU.

5.5.3 Why?

Why have both options? It is because that both are appopriate choices dependent upon the situation and how often values are being read into and out of a register. For instance:

1. If a caller isn't using a register, caller-saved is more efficient (requires less operations).
2. If a callee doesn't need a register, callee-saved is more efficient in the same fashion.