Notes on x86-64 programming

Andrew Tolmach, September 2022

This document gives a brief summary of the x86-64 architecture and instruction set. It concentrates on features likely to be useful to compiler writing. It makes no aims at completeness; current versions of this architecture contain over 1000 distinct instructions! Fortunately, relatively few of these are needed in practice.

For a fuller treatment of the material in this document, see Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective,* Prentice Hall, 2nd ed., Chapter 3. (Alternatively, use the first edition, which covers ordinary 32-bit x86 programming, and augment it with the on-line draft update for the second edition covering x86-64 topics, available at `http://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf`. Note that there are a few errors in the on-line draft.)

In this document, we adopt "AT&T" style assembler syntax and opcode names, as used by the GNU assembler.

x86-64

Most x86 processors manufactured by Intel and AMD for the past twenty years support a 64-bit mode that changes the register set and instruction set of the machine. When we choose to program using the "x86-64" model, it means both using this mode *and* adopting a particular Application Binary Interface (ABI) that dictates things like function calling conventions.

For those familiar with 32-bit x86 programming, the main differences are these:

- Addresses are 64 bits.
- There is direct hardware support for arithmetic and logical operations on 64-bit integers.
- There are 16 64-bit general purpose registers (instead of 8 32-bit ones).
- We use a different calling convention that makes heavy use of registers to pass arguments (rather than passing them on the stack in memory).
- For floating point, we use the `%xmm` register set provided by the SSE extensions, rather than the old x87 floating instructions.

Data Types

The x86-64 registers, memory and operations use the following data types (among others):

data type	suffix	size (bytes)
byte	b	1
word	w	2
double (or long) word	l	4
quad word	q	8
single precision float	s	4
double precision float	d	8

The "suffix" column above shows the letter used by the GNU assembler to specify appropriately-sized variants of instructions.

The machine is byte-addressed. It is a "little endian" machine, i.e., the least significant byte in a word has the lowest address. Data should be aligned in memory; that is, an n-byte item should start at an address divisible by n.

Addresses are 64 bits. In practice, no current hardware implements a 16 exabyte address space; the current norm is 48 bits (256 terabytes).

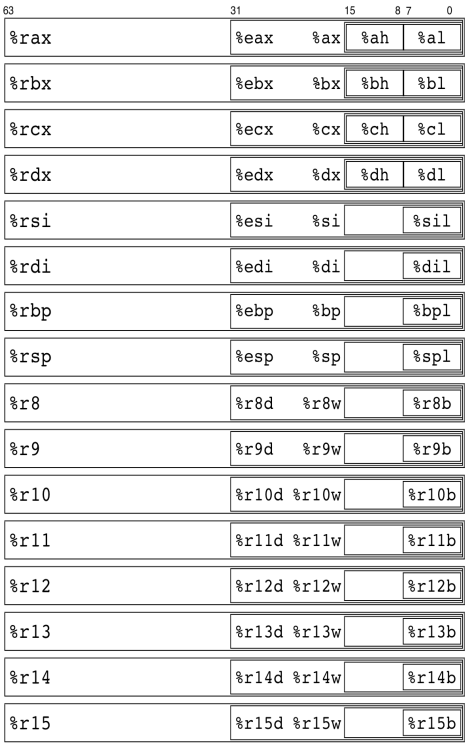

Figure 1: x86-64 registers (from Bryant ad O'Hallaron)

Registers and Stack

There are 16 64-bit "general-purpose" registers; the low-order 32, 16, and 8 bits of each register can be accessed independently under other names, as shown in Figure 1.

In principle, almost any register can be used to hold operands for almost any logical and arithmetic operation, but some have special or restricted uses.

By convention, and because several instructions (e.g., `push`,`pop`,`call`) make implicit use of it, `%rsp` is reserved as the stack pointer. The stack grows *down* in memory; `%rsp` points to the lowest occupied stack location (*not* to the next one to use).

Register `%rbp` is sometimes used as a frame pointer, i.e., the base of the current stack frame. The `enter` and `leave` instructions make implicit reference to it. It is common to do without a frame pointer, however, allowing `%rbp` to be used for other purposes. This decision can be made on a per-function basis.

A few other instructions make implicit use of certain registers; for example, the integer divide instructions require `%rax` and `%rdx`.

The instruction pointer register (`%rip`) points to the next instruction to execute; it cannot be directly accessed by the programmer, but is heavily used in assembly code as the base for position-independent code addressing.

For floating point, it is best to use the registers that are provided by the SSE extensions available in all recent processors. (SSE has nothing directly to do with 64 bit support, but the use of SSE is part of the X86-64 ABI. The older "x87" floating point instructions, which use an inconvenient register stack, are best avoided.) These registers are named `%xmm0` through `%xmm15` (*not* to be confused with the `%mmx` registers, which are something else entirely!) Each `%xmm` register can be used to hold either a single-precision (32 bit) or a double-precision (64 bit) floating value.

Addressing Modes

Operands can be immediate values, registers, or memory values.

Immediates are specified by a `$` followed by an integer in standard C notation. In nearly all cases, immediates are limited to 32 bits.

For all but a few special instructions, memory addresses are specified as

`offset(base,index,scale)`

where `base` and `index` are registers, `scale` is a constant 1,2,4, or 8, and `offset` is a constant or symbolic label. The *effective address* corresponding to this specification is (`base` + `index` × `scale` + `offset`). Any of the various fields may be omitted if not wanted; in effect, the omitted field contributes 0 to the effective address (except that `scale` defaults to 1). Most instructions (e.g., `mov`) permit at most one operand to be a memory value.

Instructions are byte-aligned, with a variable number of bytes. The size of an instruction depends mostly on the complexity of its addressing mode. The performance tradeoff between using shorter, simpler instructions and longer, more powerful ones is complex.

Offsets are limited to 32 bits. This means that only a 4GB window into the potential 64-bit address space can be accessed from a given base value. This is mainly an issue when accessing static global data. It is standard to access this data using *PC-relative addressing* (using `%rip` as the `base`). For example, we would write the address of a global value stored at location labeled `a` as `a(%rip)`, meaning that the assembler and linker should cooperate to compute the offset of `a` from the ultimate location of the current instruction.

Data transfer instructions

In the instruction specifications that follow, s is an immediate, register, or memory address, d is a register or memory address, and r denotes a register.

Most transfers use the `mov` instruction, which works between two registers or between registers and memory (but not memory-to-memory).

`mov[b|w|l|q]` s`,`d	move s to d
`movs[bw|bl|bq|wl|wq|lq]` s`,`r	move with sign extension
`movz[bw|bl|bq|wl|wq]` s`,`r	move with zero extension
`movabsq` imm`,`r	move absolute quad word (imm is 64-bit)
`pushq` s	push onto stack
`popq` d	pop from stack

When writing a byte or word into the lower part of a register, `mov` (and the arithmetic operations) only affect the lower byte or word. This is seldom what you want; use the `movs` or `movz` instruction instead to fill the higher-order bits appropriately. Inconsistently, `mov` (and the arithmetic operations) operations that write a *longword* into the lower half of a register cause the uppper half of the register to be set to zero.

Recall that immediates are normally restricted to 32 bits. To load a larger constant into a quad register, you can use `movabsq`, which takes a full 64-bit immediate as its source. Many assemblers automatically generate generate `movabsq` in place of `movq` from an immediate that does not fit in 32 bits.

The `pushq` and `popq` combine a move with an adjustment to `%rsp`. Note that the stack should stay 8-byte aligned at all times.

There are also various specialized instructions, not shown here, that move multiple bytes directly from memory to memory. Depending on the processor implementation, these may be quite efficient, but they are typically not very useful to a compiler (as opposed to hand-written library code).

Integer Arithmetic and Logical Operations

`lea[b|wl|q]` m`,`r	load effective address of m into r
`inc[b|w|l|q]` d	$d = d + 1$
`dec[b|w|l|q]` d	$d = d - 1$
`neg[b|w|l|q]` d	$d = -d$
`not[b|w|l|q]` d	$d = \sim d$ (bitwise complement)
`add[b|w|l|q]` s`,`d	$d = d + s$
`sub[b|w|l|q]` s`,`d	$d = d - s$
`imul[w|l|q]` s`,`d	$d = d * s$ (throws away high-order half of result; d must be a register)
`xor[b|w|l|q]` s`,`d	$d = d \wedge s$ (bitwise)
`or[b|w|l|q]` s`,`d	$d = d \mid s$ (bitwise)
`and[b|w|l|q]` s`,`d	$d = d \& s$ (bitwise)
`idivl` d	signed divide of `%edx::%eax` by d; quotient in `%eax`, remainder in `%edx`
`divl` d	unsigned divide of `%edx::%eax` by d; quotient in `%eax`, remainder in `%edx`
`cltd`	sign extend `%eax` into `%edx::%eax`
`idivq` s	signed divide of `%rdx::%rax` by s; quotient in `%rax`, remainder in `%rdx`
`divq` s	unsigned divide `%rdx::%rax` by s; quotient in `%rax`, remainder in `%rdx`
`cqto`	sign extend `%rax` into `%rdx::%rax`
`sal[b|w|l|q]` imm`,`d	$d = d << imm$ (left shift)
`sar[b|w|l|q]` imm`,`d	$d = d >> imm$ (arithmetic right shift)
`shr[b|w|l|q]` imm`,`d	$d = d >> imm$ (logical right shift)

The `lea` instruction loads the effective address of its source operand (rather than the datum at that address) into its destination register. It can also be used to perform arithmetic that has nothing to do with addressing.

A very common trick is to zero a register by `xor`ing it with itself.

Recall that when an instruction targets the low-order byte or word of a register, the higher-order portion of the register is unchanged, but if it targets the low-order longword, the higher-order longword is zeroed. In practice, it is usually easiest to do all arithmetic on full quadwords, by sign or zero extending at loads and ignoring high-order parts at stores.

Multiplication of two n-byte values yields a potentially $2n$-byte result. The `imul` instruction simply discards the high-order half of the result, so the result still fits in n bytes; this is the normal semantics for muliply in most programming languages. Note that signed and unsigned multiplication are equivalent in this case. (There is another version of `imul` that preserves the high-order information, but we won't need it.) This form of `imul` requires the destination to be a register, not a memory address.

Division requires special arrangements: `idiv` (signed) and `div` (unsigned) operate on a $2n$-byte dividend and an n-byte divisor to produce an n-byte quotient and n-byte remainder. The dividend always lives in a fixed pair of registers (`%edx` and `%eax` for the 32-bit case; `%rdx` and `%rax` for the 64-bit case); the divisor is specified as the source operand (indicated above as d because it must not be an immediate value) in the instruction. The quotient goes in `%eax` (resp. `%rax`); the remainder in `%edx` (resp. `%rdx`). For signed division, the `cltd` (resp. `ctqo`) instruction is used to prepare `%edx` (resp. `%rdx`) with the sign extension of `%eax` (resp. `%rax`). For example, if `a`,`b`, `c` are memory locations holding quad words, then we could set `c = a/b` using the sequence: `movq a(%rip), %rax; ctqo; idivq b(%rip); movq %rax, c(%rip).`

Condition Codes

Nearly all arithmetic instructions (notably excluding `lea`) set processor condition codes based on their result. The codes are

ZF	result was Zero
CF	result caused Carry out of most significant bit
SF	result was negative (Sign bit was set)
OF	result caused (signed) Overflow

In practice, compilers usually set the condition codes using one of the following instructions, which do not change any registers:

`cmp[b|w|l|q]` s_2`,`d_1	set flags based on $d_1 - s_2$
`test[b|w|l|q]` s_2`,`d_1	set flags based on $d_1 \& s_2$ (logical and)

Various combinations of condition codes correspond to intersting relationships between compared values. The following standard condition suffixes *cc* are defined:

cc	condition tested	meaning after `cmp`
`e`	ZF	equal to zero
`ne`	~ ZF	not equal to zero
`s`	SF	negative
`ns`	~ SF	non-negative
`g`	~ (SF xor OF) & ~ ZF	greater (signed $>$)
`ge`	~ (SF xor OF)	greater or equal (signed $>=$)
`l`	SF xor OF	less (signed $<$)
`le`	(SF xor OF) | ZF	less or equal (signed $<=$)
`a`	~ CF & ~ ZF	above (unsigned $>$)
`ae`	~ CF	above or equal (unsigned $>=$)
`b`	CF	below (unsigned $<$)
`be`	CF | ZF	below or equal (unsigned $<=$)

These suffixes modify three different kinds of instructions:

- `j`*cc* l transfers control to label l if the specified *cc* holds.
- `set`*cc* d sets the single byte destination d to 1 or 0 according to whether the specified *cc* holds or not.
- `cmov`*cc* instructions perform moves only if the specified *cc* holds. (We won't discuss these further.)

In AT&T syntax, the order of operands to comparisons is backwards from what you might expect. For example, the sequence `cmpq %rax, %rbx; jl foo` transfers control to `foo` if the value of `%rbx` is less than that of `%rax`.

Control Transfers

The basic unconditional transfer is `jmp`, which has a direct form (e.g., `jmp fred` transfers control to label `fred`) and an indirect form (e.g., `jmp *%eax` transfers control to the address in `%eax`; any valid addressing mode can be used here).

Conditional transfers have the form `j`*cc*, where are based on the condition codes given above; they exist only in direct form.

The `call` instruction acts like `jmp`, except that it first pushes `%rip` (the address of the instruction following the `call`) onto the stack, for later use by a `ret` instruction. `ret` pops the top of stack into `%rip`, thus resuming execution in the calling routine. The target of a `call` should be 16-byte aligned; this can be guaranteed by using the assembler pseudo-op `.p2align 4,0x90`, which pads to a 16-byte boundary filling with no-op instructions.

Sometimes `%rbp` is used as a frame pointer (i.e. pointer to where the top of stack was at entry to the function). The `leave` instruction sets `%rsp` to `%rbp` and then pops the stack into `%rbp`, effectively popping the entire current stack frame. It is nominally intended to reverse the action of a previous `enter` instruction, but you probably won't want to use `enter`, whose spec is complicated.

Floating Point Arithmetic

This section describes operations on the SSE3 scalar floating point registers. Here x means a register in the range `%xmm0`,...,`%xmm15`. Each register can contain either a 4-byte single precision or 8-byte double precision float. (In fact, the registers can contain 16-byte long doubles, but we'll ignore these.) In this section, a source operand s is either a memory location or an x register; a destination operand d is always an x register.

`movs[s|d]` s`,`d	move [single|double]-precision s to d
`movs[s|d]` x`,`m	move [single|double]-precision x to m
`cvtss2sd` s`,`d	convert single s to double precision d
`cvtsd2ss` s`,`d	convert double s to single precision d
`cvtsi2s[s|d]` $[m \mid r]$`,`d	convert longword integer to [single|double] precision d
`cvtsi2s[s|d]q` $[m \mid r]$`,`d	convert quadword integer to [single|double] precision d
`cvtts[s|d]2si` s`,`r	convert with truncation [single|double] precision s to longword integer
`cvtts[s|d]2siq` s`,`r	convert with truncation [single|double] precision s to quadword integer
`adds[s|d]` s`,`d	$d = d + s$
`subs[s|d]` s`,`d	$d = d - s$
`muls[s|d]` s`,`d	$d = d * s$
`divs[s|d]` s`,`d	$d = d/s$
`maxs[s|d]` s`,`d	$d = max(d, s)$ (maximum)
`mins[s|d]` s`,`d	$d = min(d, s)$ (minimum)
`sqrts[s|d]` s`,`d	$d = \sqrt{s}$ (square root)
`ucomis[s|d]` s_2`,`s_1	set condition codes based on comparison $s_1 - s_2$

Note that the `ucomis[s|d]` instructions perform signed comparison (floats are always signed), but the result should be tested using the *unsigned* condition suffixes (`a`,`ae`,`b`,`be`).

For further information on floating point arithmetic operations using the SSE extensions and `%xmm` registers, see the Web Aside ASM:SSE for Bryant and O'Hallaron's book, available at `http://csapp.cs.cmu.edu/public/waside/waside-sse.pdf`.

Calling Conventions

This section describes the C calling convention adopted by linux and GNU tools for the x86-64. Although we could, in principle, use any calling convention we like for calling non-C functions, in practice it is easiest to stick to this one.

Integer arguments (up to the first six) are passed in registers, namely: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`. Additional arguments, if needed, are passed in stack slots immmediately above the return address.

An integer-valued function returns its result in `%rax`.

Registers `%rbx`,`%rbp`,`%r12`,`%r13`,`%r14`,`%r15` are callee-save; that is, the callee is responsible for making sure that the values in these registers are the same at exit as they were on entry. Use of `%rbp` as a frame pointer by the callee is optional. The remaining registers are caller-save; that is, the callee can do what it likes with them, so if the caller wants to preserve their values across calls, it must do so itself.

Floating arguments (up to 8) are passed in SSE registers `%xmm0`, `%xmm1`, ..., `%xmm7`. Additional arguments, if needed, are passed in stack slots. When calling a function that takes a variable number of arguments (notably `printf`) or lacks a prototype, byte register `%al` must be set before the call to indicate how many of the `%xmm` registers are used. A floating point return value is returned in `%xmm0`. All the `%xmm` registers are caller-save.

Any arguments passed on the stack are pushed in reverse (right-to-left) order. Each argument is 8-byte aligned. Moreover, the value (`%rsp`+8) must always be 16-byte aligned when control is transfered to a function entry point. (System libraries rely on this invariant, and calls to them may crash if it is not obeyed.)

Normally, functions enlarge their stack frame to contain local data and saved registers by adjusting `%rsp`, either all at once or by a sequence of `pushq` operations. However, it is always permissable to write to the 128-byte area immediately *below* `%rsp`, the so-called "red zone." This permits functions to write a small amount of local data without having to adjust `%rsp` at all. All values in stack frames should be aligned to their size (e.g. 8-byte values on 8-byte boundaries).