

# AMD64 overview

COMP 40

Fall 2009

## 1 Key locations

### 1.1 Integer unit

The 64-bit registers by number are `%rax`, `%rcx`, `%rdx`, `%rbx`, `%rsp`, `%rbp`, `%rsi`, `%rdi`, and `%r8` to `%r15`. Figure 1 shows the various sub-registers. You are quite likely to encounter such registers as `%eax` or `%edi`, especially when dealing with functions that take 32-bit parameters.

The integer status register includes the typical flags OF (overflow flag), SF (sign flag), ZF (zero flag), and CF (carry flag). Flags unique to the Intel family include PF (parity flag), AF (auxiliary carry flag), and DF (direction flag for string operations). Flags are set by most arithmetic operations and tested by the “jump conditional” instructions.

### 1.2 128-bit multimedia unit

This unit includes sixteen 128-bit registers numbered `%xmm0` to `%xmm15`. This unit provides a variety of vector-parallel instructions (Streaming SIMD Extensions, or SSE) including vector-parallel floating-point operations on either 32-bit or 64-bit IEEE floating-point numbers (single and double precision).

### 1.3 IEEE Floating-point unit

The IEEE floating-point unit has eight 80-bit registers numbered `%fpr0` to `%fpr7`. It provides floating-point operations on 80-bit IEEE floating-point numbers (double extended precision).

### 1.4 Parameter registers

Integer parameters are passed in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`. Single-precision and double-precision floating-point parameters (`float` and `double`) are passed in registers `%xmm0` through `%xmm7`. Structure parameters, extended-precision floating-point numbers (`long double`), and parameters too numerous to fit in registers are passed on the stack.

### 1.5 Result registers

An integer result is normally returned in `%rax`. If an integer result is too large to fit in a 64-bit register, it will be returned in the `%rax:%rdx` register pair. A single-precision or double-precision floating-point result is returned in `%xmm0`; an extended-precision floating-point result is returned on top of the floating-point stack in `%st0`. Complex numbers return their imaginary parts in `%xmm1` or `%st1`.

### 1.6 Registers preserved across calls

Most registers are overwritten by a procedure call, but the values in the following registers must be preserved:

`%rbx` `%rsp` `%rbp` `%r12` `%r13` `%r14` `%r15`

In addition, the contents of the x87 floating-point control word, which controls rounding modes and other behavior, must be preserved across calls.

A typical procedure arranges preservation with a prolog that pushes `%rbp` and `%rbx` and subtracts a constant  $k$  from `%rsp`. The body of the procedure usually avoids `%r12–%r15` entirely. Finally, before returning, the procedure then adds  $k$  to `%rsp`, then pops `%rbx` and `%rbp`. But there are many other ways to achieve the same goal, which is that on exit, the nonvolatile registers have the same values they had on entry.

## 2 Assembly-language reference to operands and results

A reference to an operand or result is called an *effective address*. The value of an operand may be coded into the instruction as a literal or *immediate* operand, or it may be stored in a container. A result is always stored in a container.

Immediate operands begin with `$` and are followed by C syntax for a decimal or hexadecimal literal:

```
$0x408ba
$12
$-4
$0xffffffffffffffffc0
```

In DDD, literals are written as in C, without the `$` sign. As in C, hexadecimal literals must have a leading `0x`.

The machine can refer to two kinds of containers: registers and memory. Registers are referred to by name, with a `%` sign in the assembler and in `objdump`:

```
%rax    %xmm0
```

In DDD, registers are referred to with a `$` sign.

Memory locations are always referred to by the address of the first byte; the assembly-language syntax is arcane:

```
(%rax)           The address is the value stored in %rax, which we'll refer to simply as %rax.
0x10(%rax)       The address is %rax + 16.
-0x8(%ebx)       The address is %ebx - 8.
$0x4089a0(,%rax,8) The address is 0x4089a0 + 8 * %rax. This form of reference can be used for
                  very fast array indexing, provided the elements of the array are 8 bytes in size, as in
                  an array of pointers. Only multipliers 1, 2, 4, and 8 are supported.
(%ebx,%ecx,1)    The address is %ebx + 1 * %ecx, i.e., the sum of the values in %ebx and %ecx.
12(%ebx,%ecx,1)  The address is %12 + %ebx + %ecx.
```

Here are some example instructions:

```
mov  -0x8(%rbx),%edx    Take the 32-bit word whose first byte is stored at memory address %rbx-8
                        and put it into the least significant 32 bits of %rax.

mov  0x8(%rsp),%rbx     Take from the stack the 64-bit word whose first byte is located at address
                        %rsp+8, and put it into register %rbx.

mov  $0x5,%edx         Store the literal 5 into %rdx.

add  $0x1,%rsi         Add 1 to the contents of register %rsi.

addq $0x1,0x8(%rsp)    Add 1 to the 64-bit word whose first byte is located at address %rsp+8.
                        The q suffix is needed on the add because the literal 1 could represent an
                        integer of any size, and the address %rsp+8 could point to an integer of
                        any size. The q means "64 bits." (l means 32 bits, w means 16 bits, and
                        b means 8 bits). A suffix is normally unnecessary, because the way the
                        register is named indicates the size (examples include %rax, %eax, %ax,
                        and %al).

lea  -0x30(%edx,%esi,8),%esi Compute the address %edx+8*%esi-48, but don't refer to the contents of
                        memory. Instead, store the address itself into register %esi. This is the
                        "load effective address" instruction: its binary coding is short, it doesn't
                        tie up the integer unit, and it doesn't set the flags.
```

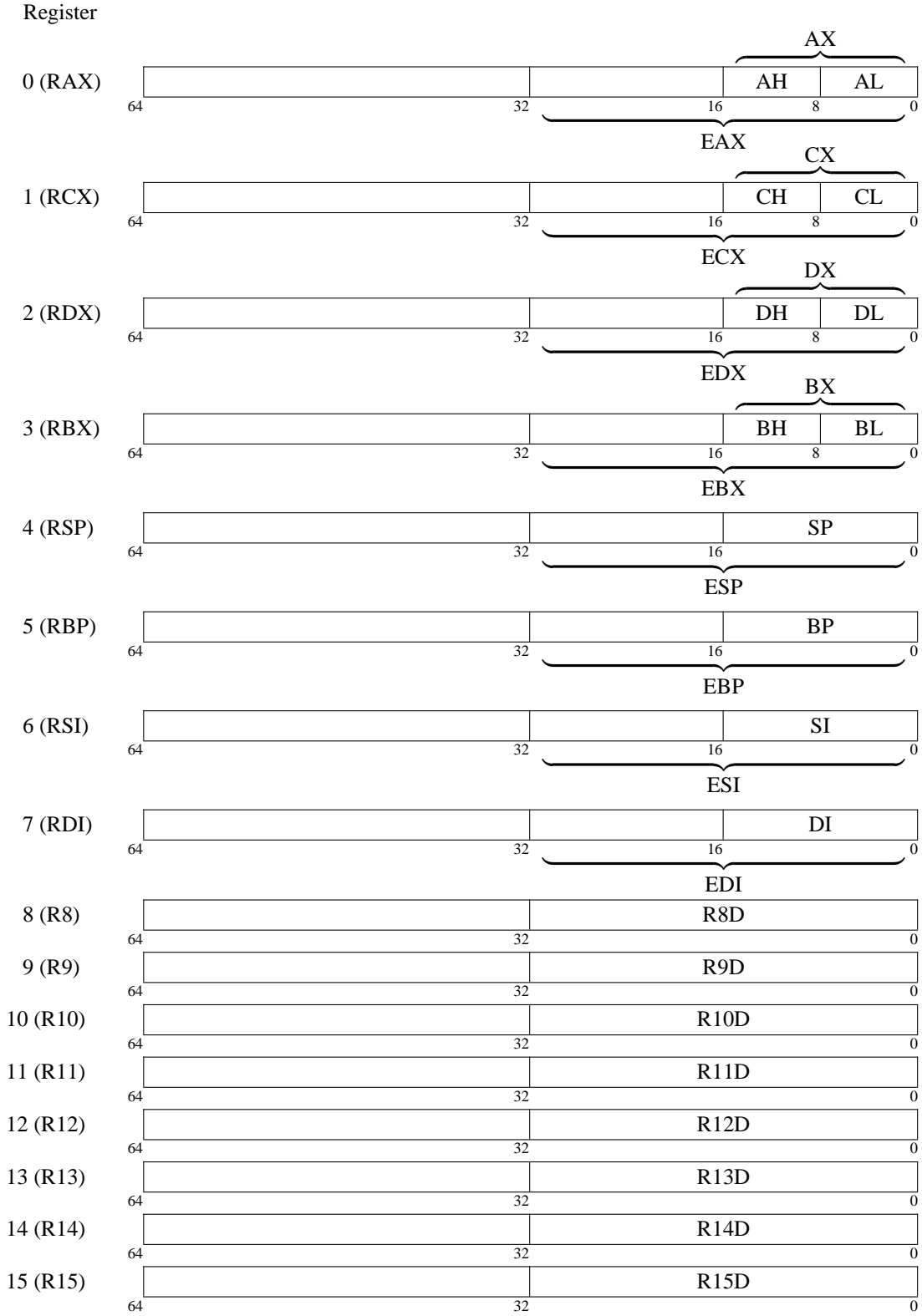


Figure 1: AMD64 Integer Registers

### 3 Selected integer instructions

<i>Opcode</i>	<i>Examples</i>	<i>RTL</i>	
add	add \$0x18,%rsp add 0x8(%rcx),%rdx	$\%rsp := \%rsp + 24$   touch flags $\%rdx := \$(\%rcx + 8)$   touch flags	
sub	sub \$0x18,%rsp sub %rax,0x8(%rdx) sub %rdx,%rax	$\%rsp := \%rsp - 24$   touch flags $\$(\%rdx + 8) := \$(\%rdx + 8) - \%rax$   touch flags $\%rax := \%rax - \%rdx$   touch flags	
lea	lea 0x10(%rsp),%rax lea (%rbx,%rax,8),%rax	$\%rax := \%rsp + 16$ $\%rax := \%rbx + \%rax \times_u 8$	load effective address (flags unchanged)
adc	adc \$0x0,%ecx adc \$0xffffffffffffffff,%r12	$\%rcx := \$(\%rcx + 0 + CF)$   touch flags $\%r12 := \$(\%r12 - 1 + CF)$   touch flags	add with carry
sbb	sbb %eax,%eax sbb \$0x3,%rdi	$\%eax := \%eax - (\%eax + CF)$   touch flags $\%rdi := \%rdi - (3 + CF)$   touch flags	subtract with borrow
neg	neg %edx negq 0x28(%rsp)	$\%edx := -\%edx$   touch flags $\$(\%rsp + 40)_{32} := -\$(\%rsp + 40)_{32}$   touch flags	two's-complement negate
mul	mul %rcx mul %ecx	$\%rdx:\%rax := \%rax \times_u \%rcx$   touch flags $\%edx:\%eax := \%eax \times_u \%ecx$   touch flags	unsigned multiply
imul	imul 0x10(%rbx),%rbp	$\%rbp := \text{lobits}_{64}(\%rbp \times_s \$(\%rbx + 16))$   touch flags	signed multiply
div	div %esi	$\%rdx := \%rdx:\%rax \div_u \%esi$   $\%rax := \%rdx:\%rax \text{ rem}_u \%esi$   undef flags	unsigned divide
idiv	idiv %r8	$\%rdx := \%rdx:\%rax \div_s \%r8$   $\%rax := \%rdx:\%rax \text{ rem}_s \%esi$   undef flags	signed divide
shl	shl %cl,%rax	$\%rax := \%rax \ll (\%cl \bmod 64)$   touch flags	shift left
sar	sar %cl,%rdx	$\%rdx := \%rdx \gg_s (\%cl \bmod 64)$   touch flags	shift arithmetic right (signed)
shr	shr %cl,%rax shrl \$0x8,0x8c(%rsp)	$\%rax := \%rax \gg_z (\%cl \bmod 64)$   touch flags $\$(\%rsp + 140)_{32} := \$(\%rsp + 140)_{32} \gg_z (8 \bmod 32)$   touch flags	shift right (unsigned)
and	and %r11,%rcx	$\%rcx := \%rcx \wedge \%r11$   touch flags	bitwise and
or	or %ebx,0x10(%rsp)	$\$(\%rsp + 16) := \$(\%rsp + 16) \vee \%ebx$   touch flags	bitwise or
xor	xorb \$0x36, (%rax,%r12,1)	$\$(\%rax + \%r12)_8 := \$(\%rax + \%r12)_8 \text{ xor } 54$   touch flags	bitwise exclusive or
not	not %ebp	$\%ebp := \neg \%ebp$	one's complement
mov	mov \$0x7fffffffffffffff,%rax mov %rax, (%r9,%rsi,8) mov 0x8(%rsp),%rdi	$\%rax := 2^{63} - 1$   undef flags $\$(\%r9 + \%rsi \times 8)_{64} := \%rax$   undef flags $\%rdi := \$(\%rsp + 8)_{64}$   undef flags	load immediate store load
movs	movsbq (%rbx),%rdx movslq %edi,%rax	$\%rdx := \text{sx}_{8 \rightarrow 64} \$(\%rbx)$ $\%rax := \text{sx}_{32 \rightarrow 64} \$(\%edi)$	sign-extending load sign-extending move
movz	movzbl 0x10(%rdi),%esi movzbl 0x2(%r12,%rax,1),%eax	$\%esi := \text{zx}_{8 \rightarrow 32} \$(\%rdi + 16)$ $\%eax := \text{zx}_{8 \rightarrow 32} \$(\%r12 + \%rax + 2)$	zero-extending load
pop	pop %rbx	$\%rbx := \$(\%rsp)$   $\%rsp := \%rsp + 8$	(flags unchanged)
push	push %r14	$\$(\%rsp - 8) := \%r14$   $\%rsp := \%rsp - 8$	(flags unchanged)

### 3.1 Comparisons and control flow

<i>Opcode</i>	<i>Examples</i>	<i>Meaning</i>	
jmp	jmp <i>L</i>	start executing program at label <i>L</i>	jump
cmp	cmp %r13,%r12	set flags as if for sub %r13,%r12 (but leave %r12 unchanged)	compare
test	testb \$0x10,(%rsi)	set flags as if for andb \$0x10,(%rsi) (but leave memory unchanged)	test bit(s)
	test %eax,%eax	$ZF := (\%eax \wedge \%eax = 0)$ , and set other flags also	
ja	ja <i>L</i>	if comparison showed $>_u$ , jump to label <i>L</i>	jump if above
jae	jae <i>L</i>	if comparison showed $\geq_u$ , jump to label <i>L</i>	jump if above or equal
jb	jb <i>L</i>	if comparison showed $<_u$ , jump to label <i>L</i>	jump if below
jbe	jbe <i>L</i>	if comparison showed $\leq_u$ , jump to label <i>L</i>	jump if below or equal
jc	jc <i>L</i>	if $CF \neq 0$ , jump to label <i>L</i>	jump if carry
je	je <i>L</i>	if comparison showed equal ( $ZF = 0$ ), jump to label <i>L</i>	jump if equal
jg	ja <i>L</i>	if comparison showed $>_s$ , jump to label <i>L</i>	jump if greater
jge	ja <i>L</i>	if comparison showed $\geq_s$ , jump to label <i>L</i>	jump if greater or equal
jl	ja <i>L</i>	if comparison showed $<_s$ , jump to label <i>L</i>	jump if less
jle	ja <i>L</i>	if comparison showed $\leq_s$ , jump to label <i>L</i>	jump if less or equal
:			
jz	jz <i>L</i>	if last result was zero, jump to label <i>L</i> (same as je)	jump if zero
call	call printf	push address of next instruction and go to printf	call
	callq *%rax	push address of next instruction and go to instruction at address found in %rax	
	callq *0x10(%rcx)	push address of next instruction and go to instruction at address found in $\$m[\%rcx + 16]$	
ret	retq	pop an address from the stack and go to that address	return

There are many more conditional comparison instructions to be found in the architecture manual. Most notably, every conditional jump comes in both positive and negative versions; for example, the negative version of ja is jna, i.e., “jump if not above.”

SASL library	Firefox binary
75222	mov
11881	test
11073	callq
10887	je
9267	lea
7567	xor
7531	jne
5818	jmpq
5180	add
4397	cmp
2908	movq
2791	movl
2633	sub
2292	nopl
2285	pop
1944	testb
1804	and
1782	push
1732	retq
1560	jmp
1528	movzwl
1422	movzbl
1180	cmpq
931	nopw
649	shl
524	cmpl
499	xchg
499	nop
496	ja
445	or
439	jbe
414	cmovl
406	cmpb
373	orl
331	sar
326	ror
299	shr
285	movb
269	sete
258	movslq
257	sbb
230	addl
3364	mov
693	call
569	lea
507	pop
505	push
435	add
405	nop
367	test
318	je
301	sub
271	jmp
267	ret
226	movl
212	cmp
126	jne
108	xor
89	movzbl
42	movzwl
41	jbe
35	jae
33	js
33	ja
31	xchg
27	shr
24	jb
24	cmpb
23	leave
21	movsbl
19	and
18	movb
13	shl
13	addl
12	sete
12	fxch
12	fstp
11	imul
10	setne
10	sar
10	movswl
9	cmpl
8	ror
8	flds

Figure 2: Popular instructions by mnemonic and suffix