AMD64 overview

COMP 40

Fall 2010

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 am 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
$0xffffffffffffffc0
```

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.

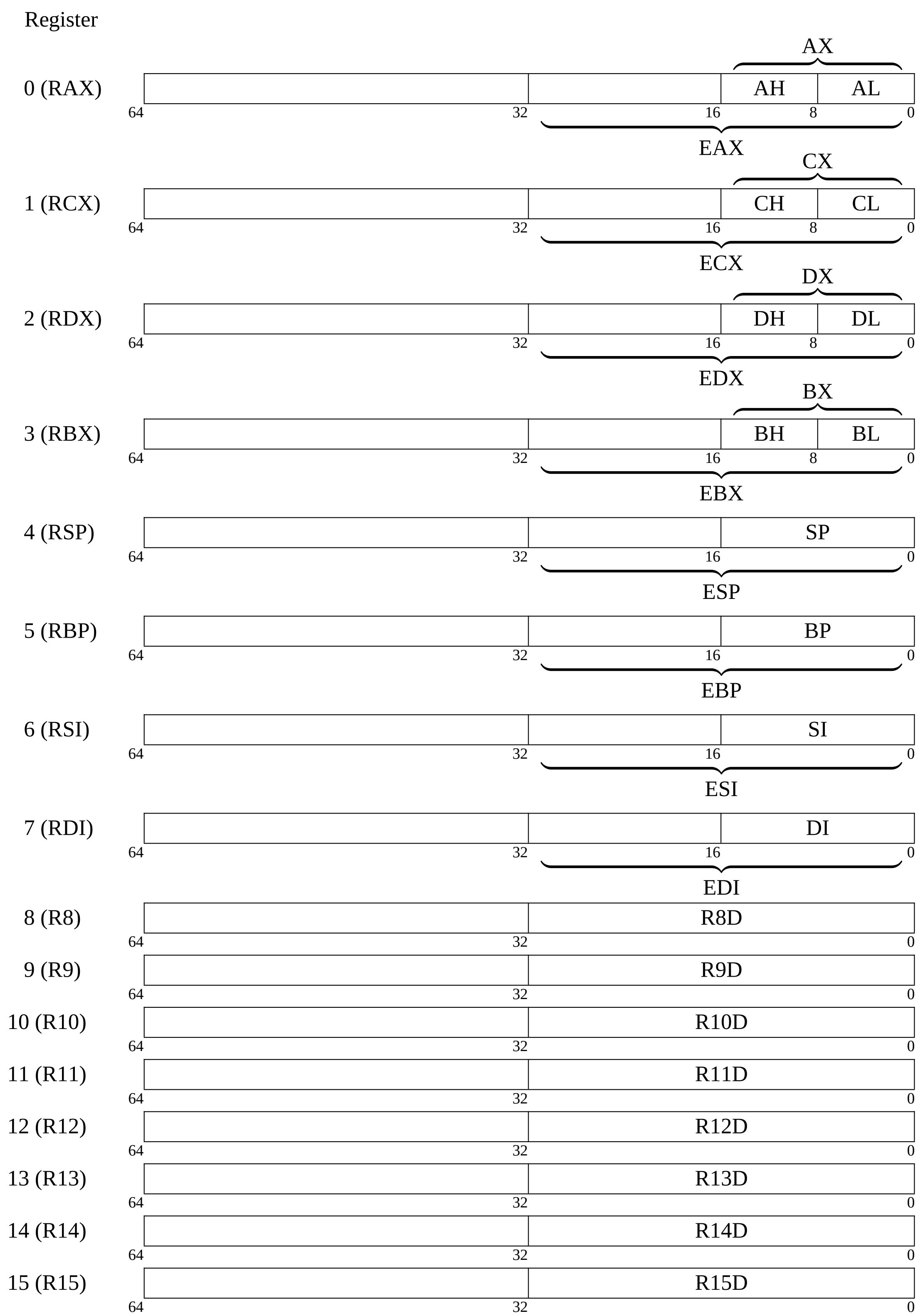

Figure 1: AMD64 Integer Registers

3 Selected integer instructions

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

3.1 Comparisons and control flow

Opcode	*Examples*	*Meaning*	
`jmp`	`jmp` L	start executing program at label L	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`	$ZF := (\texttt{\%eax} \wedge \texttt{\%eax} = 0)$, and set other flags also	
`ja`	`ja` L	if comparison showed $>_u$, jump to label L	jump if above
`jae`	`ja` L	if comparison showed $\geq_u$, jump to label L	jump if above or equal
`jb`	`jb` L	if comparison showed $<_u$, jump to label L	jump if below
`jbe`	`jb` L	if comparison showed $\leq_u$, jump to label L	jump if below or equal
`jc`	`jc` L	if $CF \neq 0$, jump to label L	jump if carry
`je`	`je` L	if comparison showed equal ($ZF = 0$), jump to label L	jump if equal
`jg`	`ja` L	if comparison showed $>_s$, jump to label L	jump if greater
`jge`	`ja` L	if comparison showed $\geq_s$, jump to label L	jump if greater or equal
`jl`	`ja` L	if comparison showed $<_s$, jump to label L	jump if less
`jle`	`ja` L	if comparison showed $\leq_s$, jump to label L	jump if less or equal
⋮			
`jz`	`jz` L	if last result was zero, jump to label L (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 $\texttt{\$m}[\texttt{\%rcx} + 16]$	
`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`	3364	`mov`
11881	`test`	693	`call`
11073	`callq`	569	`lea`
10887	`je`	507	`pop`
9267	`lea`	505	`push`
7567	`xor`	435	`add`
7531	`jne`	405	`nop`
5818	`jmpq`	367	`test`
5180	`add`	318	`je`
4397	`cmp`	301	`sub`
2908	`movq`	271	`jmp`
2791	`movl`	267	`ret`
2633	`sub`	226	`movl`
2292	`nopl`	212	`cmp`
2285	`pop`	126	`jne`
1944	`testb`	108	`xor`
1804	`and`	89	`movzbl`
1782	`push`	42	`movzwl`
1732	`retq`	41	`jbe`
1560	`jmp`	35	`jae`
1528	`movzwl`	33	`js`
1422	`movzbl`	33	`ja`
1180	`cmpq`	31	`xchg`
931	`nopw`	27	`shr`
649	`shl`	24	`jb`
524	`cmpl`	24	`cmpb`
499	`xchg`	23	`leave`
499	`nop`	21	`movsbl`
496	`ja`	19	`and`
445	`or`	18	`movb`
439	`jbe`	13	`shl`
414	`cmove`	13	`addl`
406	`cmpb`	12	`sete`
373	`orl`	12	`fxch`
331	`sar`	12	`fstp`
326	`ror`	11	`imul`
299	`shr`	10	`setne`
285	`movb`	10	`sar`
269	`sete`	10	`movswl`
258	`movslq`	9	`cmpl`
257	`sbb`	8	`ror`
230	`addl`	8	`flds`

Figure 2: Popular instructions by mnemonic and suffix