X86-64

A Brief Introduction: New Things Compared to Y86-64

CS 224

Dr. Archibald

X86-64 vs Y86-64

- Same basic ideas
- Many more instructions
- Many more variants
- More complex
- We won’t write, but we need to be able to read
- You will get there, with your Y86 experience!
- Lot’s of practice!
  - Textbook has tons of examples in Chapter 3! Use them!

Data Type Sizes and X86 suffixes

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8

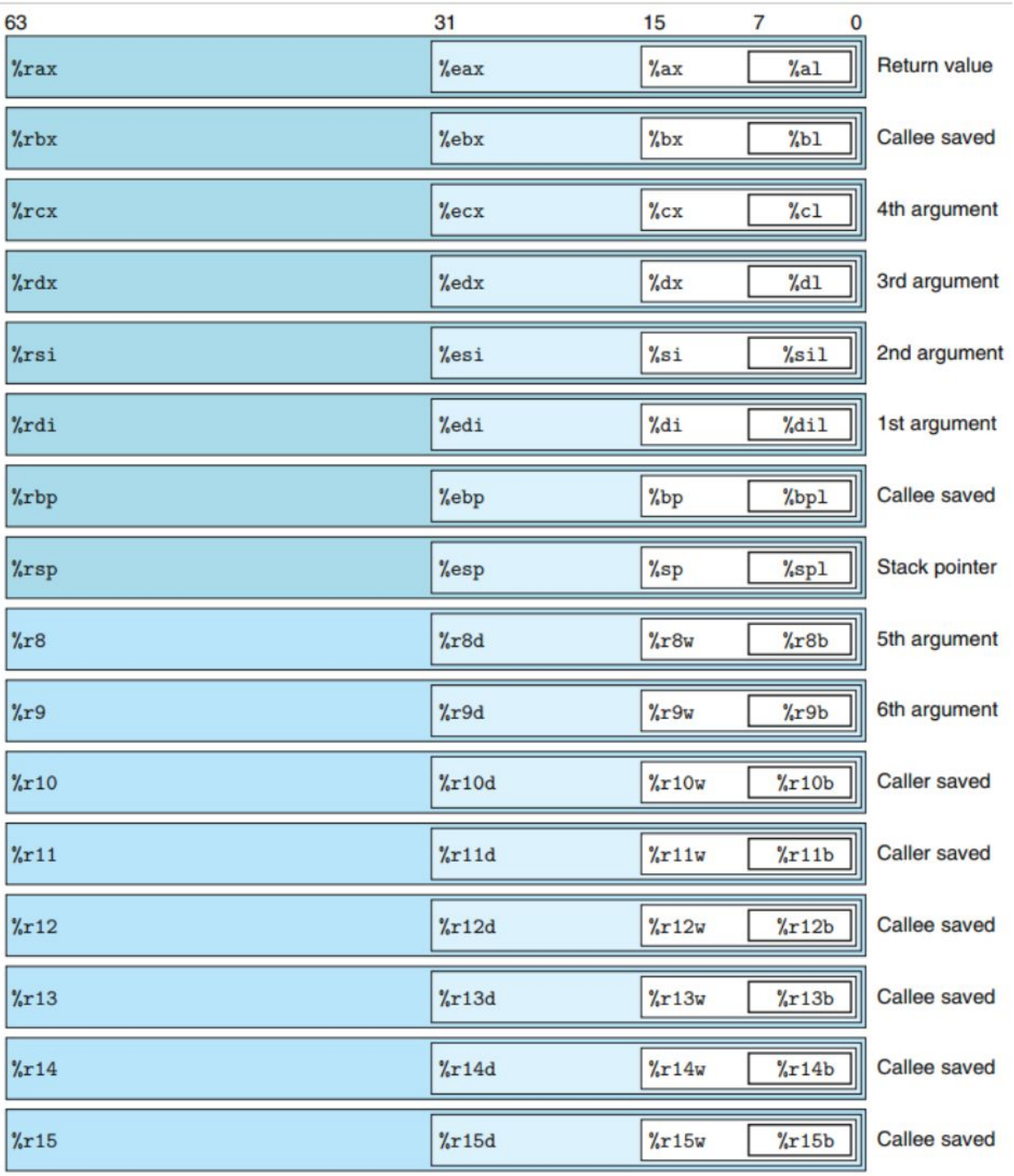
Figure 3.1 Sizes of C data types in x86-64. With a 64-bit machine, pointers are 8 bytes long.

Implications: Many versions of instructions. Example:

- `movb`
- `movw`
- `movl`
- `movq`

Registers

- Y86-64 names correspond to full 64-bit registers (plus %r15)
- In X86-64, the 32/16/8 bit registers can still be accessed
  - “Inside” the 64-bit registers
  - Naming ensures backwards compatibility
- Register size and instruction postfix need to match (where applicable)



Operand forms

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathtt{r}_a$	$\mathsf{R}[\mathtt{r}_a]$	Register
Memory	Imm	$\mathsf{M}[Imm]$	Absolute
Memory	$(\mathtt{r}_a)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_a]]$	Indirect
Memory	$Imm(\mathtt{r}_b)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b]]$	Base + displacement
Memory	$(\mathtt{r}_b,\mathtt{r}_i)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$	Indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$	Indexed
Memory	$(,\mathtt{r}_i,s)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(,\mathtt{r}_i,s)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed

Figure 3.3 **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

Operand forms: implications

- No more variants of mov
  - (`irmov, rrmov, mrmov, rmmov`)
- Instead these are written as `mov` with different operands
  - `movq $5, %rax` = `irmovq $5, %rax`
  - `movq %rax, %rbx` = `rrmovq %rax, %rbx`
  - `movq 0(%rdx), %rax` = `mrmovq 0(%rdx), %rax`
  - `movq %rax, 0(%rdx)` = `rmmovq %rax, 0(%rdx)`
- Now many more variants!
  - Still can't do memory to memory in one instruction!
  - Registers used to address memory (inside ()) must be whole 64-bit registers

Practice Problem 3.1 (solution page 325)

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Fill in the following table showing the values for the indicated operands:

Operand	Value
%rax	_____
0x104	_____
$0x108	_____
(%rax)	_____
4(%rax)	_____
9(%rax,%rdx)	_____
260(%rcx,%rdx)	_____
0xFC(,%rcx,4)	_____
(%rax,%rdx,4)	_____

New instructions: Setting Condition Codes

Instruction		Based on	Description
CMP	S_1, S_2	$S_2 - S_1$	Compare
`cmpb`			Compare byte
`cmpw`			Compare word
`cmpl`			Compare double word
`cmpq`			Compare quad word
TEST	S_1, S_2	S_1 & S_2	Test
`testb`			Test byte
`testw`			Test word
`testl`			Test double word
`testq`			Test quad word

Figure 3.13 Comparison and test instructions. These instructions set the condition codes without updating any other registers.

New ALU Instructions (`+b,w,l,q`)

- `inc` – increment
- `dec` – decrement
- `neg` – negate
- `not` - complement
- `imul` – multiply
- `or` – or
- `sal` – left shift
- `shl` – left shift
- `sar` – arithmetic right shift
- `shr` – logical right shift
- `leaq` – load effective address

`leaq` – load effective address

- This computes the address of the first operand, but doesn't access memory, it just puts the computed address in the destination location
- Used by compilers to do a lot of math

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	$Imm(r_b,r_i,s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Practice Problem 3.6 (solution page 327)

Suppose register %rax holds value x and %rcx holds value y. Fill in the table below with formulas indicating the value that will be stored in register %rdx for each of the given assembly-code instructions:

Instruction	Result
`leaq 6(%rax), %rdx`	_____
`leaq (%rax,%rcx), %rdx`	_____
`leaq (%rax,%rcx,4), %rdx`	_____
`leaq 7(%rax,%rax,8), %rdx`	_____
`leaq 0xA(,%rcx,4), %rdx`	_____
`leaq 9(%rax,%rcx,2), %rdx`	_____

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	$Imm(r_b,r_i,s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Practice Problem 3.7 (solution page 328)

Consider the following code, in which we have omitted the expression being computed:

```
long scale2(long x, long y, long z) {
    long t = ____________________;
    return t;
}
```

Compiling the actual function with GCC yields the following assembly code:

```
  long scale2(long x, long y, long z)
  x in %rdi, y in %rsi, z in %rdx
scale2:
  leaq    (%rdi,%rdi,4), %rax
  leaq    (%rax,%rsi,2), %rax
  leaq    (%rax,%rdx,8), %rax
  ret
```

Fill in the missing expression in the C code.

Practice Problem 3.8 (solution page 328)

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x108	0xAB	%rcx	0x1
0x110	0x13	%rdx	0x3
0x118	0x11		

Fill in the following table showing the effects of the following instructions, in terms of both the register or memory location that will be updated and the resulting value:

Instruction	Destination	Value
`addq %rcx,(%rax)`		
`subq %rdx,8(%rax)`		
`imulq $16,(%rax,%rdx,8)`		
`incq 16(%rax)`		
`decq %rcx`		
`subq %rdx,%rax`		

Jumps

new →

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

Indirect jumps are written using '*' followed by an operand specifier using one of the memory operand formats described in Figure 3.3. As examples, the instruction

```
jmp *%rax
```

uses the value in register %rax as the jump target, and the instruction

```
jmp *(%rax)
```

reads the jump target from memory, using the value in %rax as the read address.

Indirect Jumps

What will the PC be set to after each of the following indirect jump instructions?

- jmp *%rax
- jmp *(%rax)
- jmp *4(%rax, %rbx, 4)
- jmp *8(%rax, %rcx, 4)

Register	Value
%rax	0x100
%rbx	0x1
%rcx	0x2

Address	Value
0x100	0x104
0x108	0x100
0x110	0x108
0x118	0x63
0x120	0x32
0x128	0x

Indirect Jumps

What will the PC be set to after each of the following indirect jump instructions?
ANSWERS

- jmp *%rax
  - 0x100
- jmp *(%rax)
  - 0x104
- jmp *4(%rax, %rbx, 4)
  - 0x100
- jmp *8(%rax, %rcx, 4)
  - 0x108

Register	Value
%rax	0x100
%rbx	0x1
%rcx	0x2

Address	Value
0x100	0x104
0x108	0x100
0x110	0x108
0x118	0x63
0x120	0x32
0x128	0x

Summary

- Brief intro: X86 vs Y86
- Future weeks: practice, practice, practice
- We can’t cover it all in lecture
- Read Chapter 3!
- Lab 11 is all problems from the book
  - For each problem, we point you to most relevant section in book
  - But you should read it all!