CS356: Discussion #4

Assembly Instructions

Marco Paolieri (paolieri@usc.edu)

USC University of Southern California

What about programs that operate on data?

Integer and Floating-Point Formats

- Two’s Complement
- IEEE 754

Machine-Level Programs

- Operand specifiers
- Data movement
- Arithmetic and logic operations
- Stack manipulation
- Control structures
- Procedures

High-Level Programs

- C/C++
- Java
- Python

Why learning assembly?

Understanding the machine

- Reverse engineering
- Security analysis
- Performance tuning (rarely)

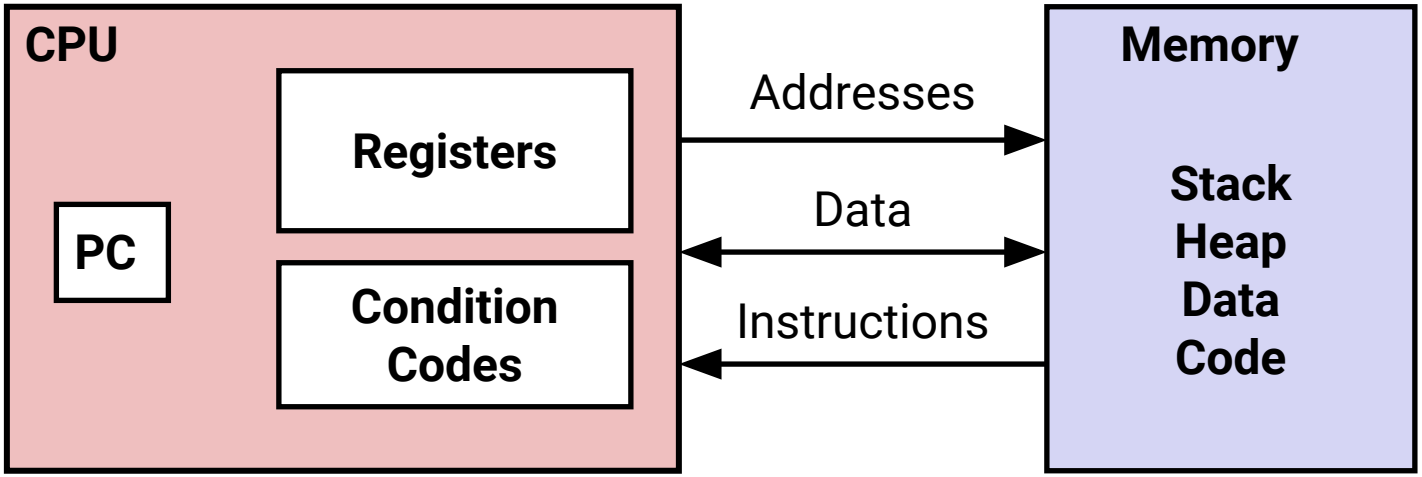
Beware: the compiler applies several optimizations!

- Rearrange execution order.
- Eliminate unneeded computations.
- Replace slow operations with faster ones.
- Change recursive operations with iterative ones.

Compilation

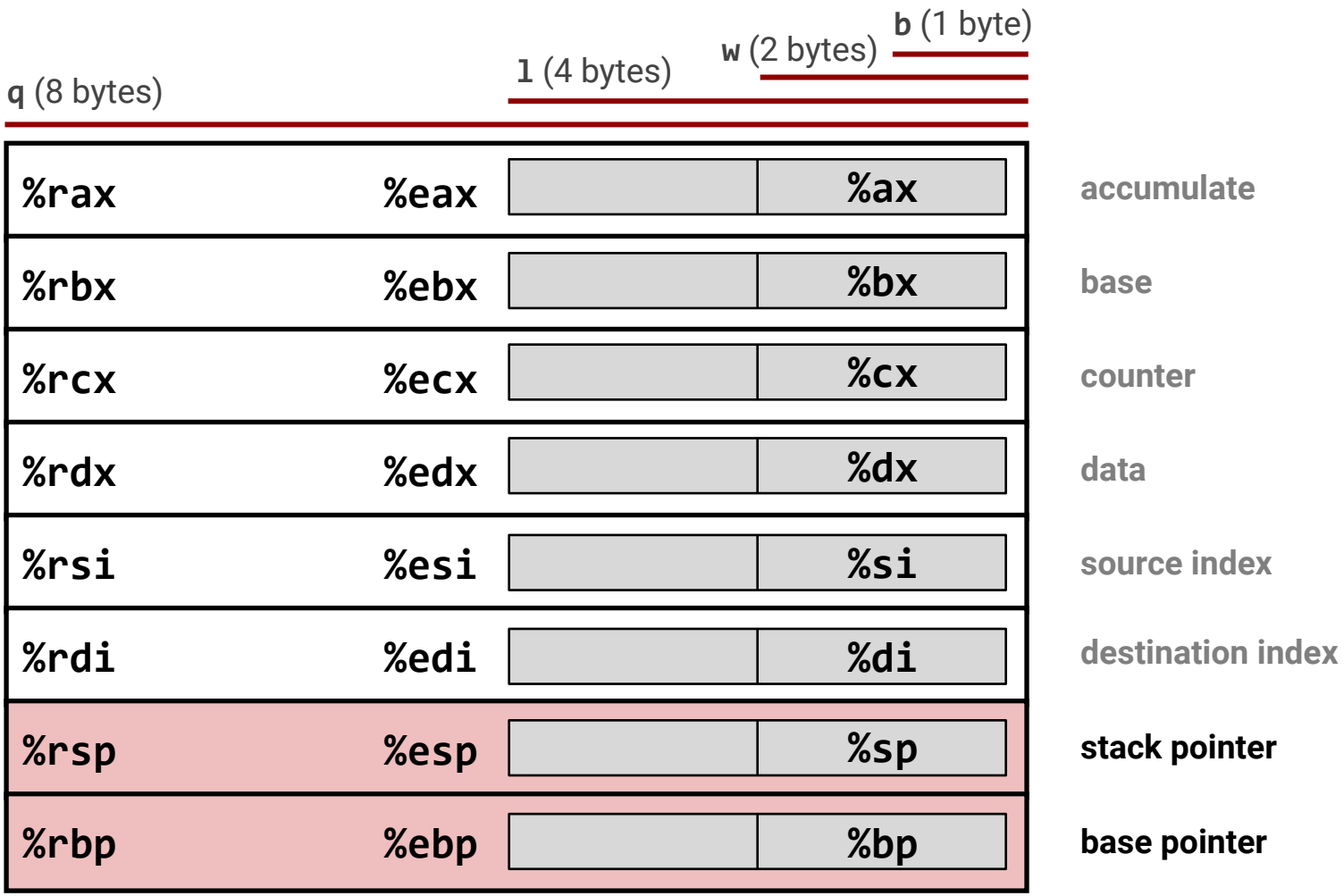
- **C** → **ASM** (compiler) → **object program** (assembler) → **executable** (linker)
- "`gcc -Og -S input.c`" produces the **assembly** of the input program

Programmer-Visible State



- **Instructions** and **data** must be read from main memory.
- Instructions are executed on **registers**.

16 × 64-bit registers



- In addition: `%r8` to `%r15` (`%r8d` / `%r8w` / `%r8b` for lower 4 / 2 / 1 bytes)

Operand Forms

Different ways to specify source values and output location.

Immediate: `$imm` to use a constant input value, e.g., `$0xFF`.

Register: `%reg` to use the value contained in a register, e.g., `%rax` .

Memory reference

- **Absolute**: `addr`, e.g., `0x1122334455667788` [use a fixed address]
- **Indirect**: `(%reg)`, e.g., `(%rax)` [use the address contained in a **q register**]
- **Base+displacement**: `imm(%reg)`, e.g., `16(%rax)` [add a displacement]
- **Indexed:** `(%reg1,%reg2)`, e.g., `(%rax,%rbx)` [add another register]
- **Indexed+displacement:** `imm(%reg1,%reg2)` [add both]
- **Scaled indexed:** `imm(%reg1,%reg2,c)` [use address: `imm+reg1+reg2*c`]
  Variants: omit `imm` or `reg1` or **both**. E.g., `(,%rax,10)`

(A memory reference selects the first byte.)

Operand Forms: Examples

Values at each memory address:

- `0x100: 0xFF`
- `0x104: 0xAB`
- `0x108: 0x13`
- `0x10C: 0x11`

Values in registers:

- `%rax: 0x100`
- `%rcx: 0x1`
- `%rdx: 0x3`

Operand value?

- `%rax`
- `0x104`
- `$0x108`
- `(%rax)`
- `4(%rax)`
- `9(%rax,%rdx)`
- `260(%rcx,%rdx)`
- `0xFC(,%rcx,4)`
- `(%rax,%rdx,4)`

Solutions:

`0x100`
`0xAB`
`0x108`
`0xFF`
`0XAB`
`0x11`
`0x13`
`0xFF`
`0x11`

Data Movement: Instructions

Move to register/memory (register operands must match size codes)

- **`movb`** `src, dst` (1 byte)
- **`movw`** `src, dst` (2 bytes)
- **`movl`** `src, dst` (4 bytes / with register destination, the others are set to 0)
- **`movq`** `src, dst` (8 bytes)
- **`movabsq`** `imm, reg` (8 bytes / 64-bit source value allowed into register)

(Either **`src`** or **`dst`** can refer to a memory location, not both; no **`imm`** as **`dst`**.)

Move from register/memory to register (zero extension)

- **`movzbw`** `src, reg` (byte to word)
- **`movzbl`** `src, reg` (byte to double word)
- **`movzbq`** `src, reg` (byte to quad word)
- **`movzwl`** `src, reg` (word to double word)
- **`movzwq`** `src, reg` (word to quad word)

Same, but with sign extension (replicate MSB):

- **`movsbw`**, **`movsbl`**, **`movsbq`**, **`movswl`**, **`movswq`**, **`movslq`**, **`cltq`** (%eax to %rax)

Data Movement: Examples

Mistakes

- `movb $0xF, (%ebx)`
- `movl %rax, (%rsp)`
- `movw (%rax), 4(%rsp)`
- `movb %al, %sl`
- `movq %rax, $0x123`
- `movl %eax, %rdx`
- `movb %si, 8(%rbp)`

Valid

- ```
  movq (%rdi), %rax     // read 8 bytes from address in %rdi
  movq %rax, (%rsi)     // save 8 bytes to address in %rsi
  ```
- ```
  movsbw (%rdi), %ax    // like char to short cast in C
  movw %ax, (%rsi)
  ```
- ```
  movzbl (%rdi), %eax   // like unsigned char to long cast in C
  movq %rax, (%rsi)
  ```
  ```

Data Movement: Zero and Sign Extension

A sequence of instructions

```
movabsq $0x011223344556677, %rax   // %rax = 0x011223344556677
movb $0xAA, %dl                    // %dl  = 0xAA
movb   %dl, %al                    // %rax = 0x011223344556AA
movsbq %dl, %rax                   // %rax = 0xFFFFFFFFFFFFFFAA
movzbq %dl, %rax                   // %rax = 0x00000000000000AA
```

Another sequence of instructions

```
movabsq $0x011223344556677, %rax   // %rax = 0x011223344556677
movb $-1, %al                      // %dl  = 0x011223344556FF
movw $-1, %ax                      // %rax = 0x0112233445FFFF
movl $-1, %eax                     // %rax = 0x00000000FFFFFFFF
movq $-1, %rax                     // %rax = 0xFFFFFFFFFFFFFFFF
// note: “movq $-1, %rax” extends $0xFFFFFFFF to 8 bytes
movq $0xFF, %rax                   // %rax = 0x00000000000000FF
```

Arithmetic Instructions

Unary (with `q` / `l` / `w` / `b` variants)

- `incq x` is equivalent to `x++`
- `decq x` is equivalent to `x--`
- `negq x` is equivalent to `x = -x`
- `notq x` is equivalent to `x = ~x`

Any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to `0`.

Binary (with `q` / `l` / `w` / `b` variants)

- `addq x,y` is equivalent to `y += x`
- `subq x,y` is equivalent to `y -= x`
- `imulq x,y` is equivalent to `y *= x`
- `andq x,y` is equivalent to `y &= x`
- `orq x,y` is equivalent to `y |= x`
- `xorq x,y` is equivalent to `y ^= x`
- `salq n,y` is equivalent to `y = y << n`
- `sarq n,y` is equivalent to `y = y >> n`
- `shrq n,y` is equivalent to `y = y >> n`

Except for right shift, all instructions are the same for signed/unsigned values (thanks to 2's-complement)

n is $imm or `%cl` (mod 32 or 64)
arithmetic: fill in sign bit from left
logical: fill in zeros from left

Arithmetic Instructions: Examples

Values at each memory address:

- `0x100: 0xFF`
- `0x108: 0xAB`
- `0x110: 0x13`
- `0x118: 0x11`

Values in registers:

- `%rax: 0x100`
- `%rcx: 0x1`
- `%rdx: 0x3`

Effect?

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

Solutions:

Write `0x100` at `0x100`
Write `0xA8` at `0x108`
Write `0x110` at `0x118`
Write `0x14` at `0x110`
Write `0x0` inside `%rcx`
Write `0xFD` inside `%rax`

Bonus: `leaq` (Load Effective Address)

`leaq src, reg`

- Saves the first **parameter** into an 8-byte register
- The first parameter can be any displaced / indexed / scaled **address**

Useful for:

- Saving an address for later use.
- Performing simple additions and constant multiplication:
  `leaq imm(reg1,reg2,c), reg3` saves `imm+reg1+reg2*c` into `reg3`
- Only one instruction is used: efficient!

Examples (`%rax` = x, `%rcx` = y)

- `leaq 6(%rax),%rdx` saves (6+x) in `%rdx`
- `leaq (%rax,%rcx),%rdx` saves (x+y) in `%rdx`
- `leaq (%rax,%rcx,4),%rdx` saves (x+4*y) in `%rdx`
- `leaq 7(%rax,%rax,8),%rdx` saves (7+9*x) in `%rdx`
- `leaq 0xA(,%rcx,4),%rdx` saves (10+4*y) in `%rdx`

Fill In the Missing C Expression

The assembly code on the right is produced by the compiler. What is a corresponding C expression for the input code?

```c
long scale(long x, long y, long z) {
// x in %rdi, y in %rsi,  z in %rdx
// output saved in %rax
    return 5*x + 2*y + 8*z;
}
```

```
scale:
    leaq (%rdi,%rdi,4), %rax
    leaq (%rax,%rsi,2), %rax
    leaq (%rax,%rdx,8), %rax
    ret
```

Try yourself! Save the C code in **`scale.c`** and compile: **`gcc -O3 -c scale.c`**
Then, disassemble the binary object **`scale.o`** generated by the compiler:

```
$ objdump -M suffix -d scale.o
scale.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <scale>:
   0:    48 8d 04 bf             leaq   (%rdi,%rdi,4),%rax
   4:    48 8d 04 70             leaq   (%rax,%rsi,2),%rax
   8:    48 8d 04 d0             leaq   (%rax,%rdx,8),%rax
   c:    c3                      retq
```

Case study: a stack

Pushing a value

- Decrement stack pointer `%rsp`
- Store new value at address pointed by `%rsp`

Example: `pushq %rax` is equivalent to

```
subq $8, %rsp
movq %rax, (%rsp)
```

Popping a value

- Read value at address pointed by `%rsp`
- Increment `%rsp`

Example: `popq %rax` is equivalent to

```
movq (%rsp), %rax
addq $8, %rsp
```

Note: Any stack element can be accessed with `%rsp`