The Hardware/Software Interface
x86-64 Programming II

Instructors:

Justin Hsia, Amber Hu

Teaching Assistants:

Anthony Mangus	Divya Ramu
Grace Zhou	Jessie Sun
Jiuyang Lyu	Kanishka Singh
Kurt Gu	Liander Rainbolt
Mendel Carroll	Ming Yan
Naama Amiel	Pollux Chen
Rose Maresh	Soham Bhosale
Violet Monserate	

http://xkcd.com/99/

Relevant Course Information

- Early Course Reflection due tonight!
- HW6 due tonight, HW7 due Monday, HW8 due Wednesday

- Lab 1b due Monday (10/13) at 11:59 pm
  - No major programming restrictions, but should avoid magic numbers by using C macros (`#define`)
  - For debugging, can use provided utility functions `print_binary_short()` and `print_binary_long()`
  - Pay attention to the output of `aisle_test` and `store_test` – failed tests will show you actual vs. expected
  - You have *late day tokens* available

Lecture Outline (1/3)

- **Memory Addressing Modes**
- Address Computation Instruction
- Extension Instructions

Memory Addressing Modes (Review)

- General memory operand: `D(Rb,Ri,S)`
  - D: Constant displacement value (like an immediate, but no '$')
  - Rb: Base register (any register)
  - `Ri`: Index register (any register except `%rsp`)
  - S: Scale factor on `Ri` – 1, 2, 4, or 8 for fixed widths of data

- Computed address is Reg[Rb]+Reg[`Ri`]*S+D
  - Usually dereferenced by instruction: Mem[Reg[Rb]+Reg[`Ri`]*S+D]
    - Example: **`movb`** `8(%rax,%rbx,2), %cl` will copy the byte of data stored *in memory* at address Reg[`rax`] + Reg[`rbx`]*2 + 8 into the lowest byte of register `%rcx`
  - Useful to encompass data structure memory accesses
    - Example: `ar[i]` ↔ `*(ar+i)` ↔ Mem[`ar+i*sizeof(`data type`)`]

Memory Addressing Special Cases (Review)

- Default values when omitted from: `D(Rb,Ri,S)`
  - D: 0
  - Rb: Reg[Rb] = 0
  - `Ri`: Reg[`Ri`] = 0
  - S: 1

- Examples:
  - `(%rsi)` → only Rb → Mem[Reg[`rsi`]]
  - `4(%r10, %r11)` → no S → Mem[Reg[`r10`] + Reg[`r11`] + 4]
  - `(,%rdx,2)` → `Ri` but no Rb → Mem[Reg[`rdx`]`*2`]

Polling Questions (1/2)

- `D(Rb,Ri,S)` computes address `Reg[Rb]+Reg[Ri]*S+D`
  - Likely will get dereferenced, but that's up to the instruction
  - Default values: `D` = `0`, `Reg[Rb]` = `0`, `Reg[Ri]` = `0`, `S` = `1`

- Assuming `%rdx` contains `0xF000` and `%rcx` contains `0x100`, what addresses are computed by the following memory operands?
  - `0x8(%rdx)`
  - `(%rdx,%rcx,4)`
  - `0x80(,%rdx,2)`

Lecture Outline (2/3)

- Memory Addressing Modes
- **Address Computation Instruction**
- Extension Instructions

Address Computation Instruction (Review, 1/2)

- Load effective address: `leaq src, dst`
  - `src` is "Mem" operand, `dst` is a Reg operand
    - **`lea`** `D(Rb,Ri,S), R`
  - Sets `dst` to the *address* computed by the `src` expression
    - Stores Reg[Rb]+Reg[Ri]*S+D in Reg[R]; does not go to memory! – IT JUST DOES MATH

- Used to compute addresses (without a memory reference)
  - Example: **`int*`** `p = &ar[i];` could be **`lea`** `(%rdi,%rsi,4), %rax`
  - Example: `p = p + 1;` could be **`lea`** `4(%rax), %rax`

Address Computation Instruction (Review, 2/2)

- Load effective address: `leaq src, dst`
  - `src` is "Mem" operand, `dst` is a Reg operand
    - **`lea`** `D(Rb,Ri,S), R`
  - Sets `dst` to the *address* computed by the `src` expression
    - Stores Reg[Rb]+Reg[Ri]*S+D in Reg[R]; does not go to memory! – IT JUST DOES MATH

- Also used to compute arithmetic expressions of the form `rb+ri*s+d`
  - Recall that `s` can only be 1, 2, 4, or 8
  - Example: **`lea`** `(%rdi,%rsi,4), %rax` with `x` in `%rdi`, `y` in `%rsi`, and `z` in `%rax`
    - If **`int*`** `x` and **`long`** `y`, then this is equivalent to **`int*`** `z = &x[y];` (compute address)
    - If **`long`** `x` and **`long`** `y`, then this is equivalent to **`long`** `z = x + 4*y;` (arithmetic)

Example: `lea` vs. `mov`

Registers

%rax	
%rbx	
%rcx	0x4
%rdx	0x100
%rdi	
%rsi	

Memory	**Word Address**
0x400	0x120
0xF	0x118
0x8	0x110
0x10	0x108
0x1	0x100

```
leaq (%rdx), %rdi
movq (%rdx), %rsi
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
```

Example: `lea` Arithmetic (1/2)

```
long arith(long x, long y, long z) {
  long t1 = x + y;
  long t2 = z + t1;
  long t3 = x + 4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Variable	Register
x	%rdi
y	%rsi
z	%rdx
return value	%rax

Compiler Explorer:
https://godbolt.org/z/dn37nxG7h

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rdx
  imulq   %rdx, %rax
  ret
```

Example: `lea` Arithmetic (2/2)

```
long arith(long x, long y, long z) {
  long t1 = x + y;
  long t2 = z + t1;
  long t3 = x + 4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

Register	Use(s)
%rdi	x
%rsi	y
%rdx	z, t4, t5
%rax	t1, t2, rval

```
arith:
  leaq    (%rdi,%rsi), %rax     # rax/t1   = x + y
  addq    %rdx, %rax            # rax/t2   = t1 + z
  leaq    (%rsi,%rsi,2), %rdx   # rdx      = 3 * y
  salq    $4, %rdx              # rdx/t4   = (3*y) * 16
  leaq    4(%rdi,%rdx), %rdx    # rdx/t5   = x + t4 + 4
  imulq   %rdx, %rax            # rax/rval = t5 * t2
  ret
```

Polling Questions (2/2)

- Which of the following x86-64 instructions correctly calculates `%rax=9*%rdi`?

  A. `leaq (,%rdi,9), %rax`

  B. `movq (,%rdi,9), %rax`

  C. `leaq (%rdi,%rdi,8), %rax`

  D. `movq (%rdi,%rdi,8), %rax`

Lecture Outline (3/3)

- Memory Addressing Modes
- Address Computation Instruction
- **Extension Instructions**

Extension Instructions: `movz`

- `movz__ src, dst  # Move with zero extension`
  `movs__ src, dst  # Move with sign extension`
  - Copy from a smaller source value to a larger destination
    - First suffix letter is size of source, second suffix letter is size of destination
    - Recall: zero-extension always fills with `0`, sign-extension fills with copy of the sign bit
  - `src` can be Mem or Reg; `dst` must be Reg

- Example: data shown in hex
  - `movzbq %al, %rbx`

??	??	??	??	??	??	??	FF	←%rax
00	00	00	00	00	00	00	FF	←%rbx

Extension Instructions: `movs`

- `movz__ src, dst  # Move with zero extension`
  `movs__ src, dst  # Move with sign extension`
  - Copy from a smaller source value to a larger destination
    - First suffix letter is size of source, second suffix letter is size of destination
    - Recall: zero-extension always fills with `0`, sign-extension fills with copy of the sign bit
  - `src` can be Mem or Reg; `dst` must be Reg

- Example: data shown in hex
  - `movsbl (%rax), %ebx`

00	00	7F	FF	C6	1F	A4	E8	←%rax

...	??	??	80	??	??	??	...	← MEM

00	00	00	00	FF	FF	FF	80	←%rbx

Recall, *any x86-64 instruction* that stores into a 32-bit (suffix `l`) register zeros out the upper 4 bytes of the register.

GDB Demo #1

- The `movz` and `movs` examples on a real machine!
  - `movzbq %al, %rbx`
  - `movsbl (%rax), %ebx`
- You will need to use GDB to get through Lab 2
  - Useful debugger in this class and beyond!
- Pay attention to:
  - Setting breakpoints (`break`)
  - Stepping through code (`step`/`next` and `stepi`/`nexti`)
  - Printing out expressions (`print` – works with regs & vars)
  - Examining memory (`x`)

Summary (1/2)

- **Memory Addressing Modes:** Memory operands specify an address in several different forms
  - `D(Rb,Ri,S)` with *base register*, *index register*, *scale factor*, and *displacement* compute the address `Reg[Rb]+Reg[Ri]*S+D` and is usually dereferenced (`Mem[]`) by instructions
    - Defaults when omitted: `Reg[Rb]=0`, `Reg[Ri]=0`, `S=1`, `D=0`
  - These map well to pointer arithmetic operations (S = size of data type)

- **Load effective address** (`lea`) instruction used to compute addresses and perform basic arithmetic
  - *Doesn't* dereference the source memory operand, unlike all other instructions!
  - Useful for computing an address (*e.g.*, `&a[2]`) or basic arithmetic (*e.g.*, `x+4*y+7`)

Summary (2/2)

- **Extension instructions** (`movz`, `movs`) allow us to zero and sign extend data into longer widths
  - Require two size suffixes for source (smaller) and destination (larger)