Lecture 7: ALU operations, arithmetic

David Hovemeyer

February 10, 2020

601.229 Computer Systems Fundamentals

ALU operations

- ALU = "Arithmetic Logic Unit"
- An ALU is a hardware component within the CPU that does computations (of various kinds) on data values
  - Addition/subtraction
  - Logical operations (shifts, bitwise and/or/negation), etc.
- So, ALU instructions are the ones that do computations on values
  - Typically, ALU operates only on integer values
  - CPU will typically have floating-point unit(s) for operations on FP values

lea instruction

- `lea` stands for "Load Effective Address"
- Instructions that allow a memory reference as an operand generally do an *address computation*
  - E.g., `movl 12(%rdx,%rsi,4), %eax`
  - Computed address (for source memory location) is `%rdx`+(`%rsi`×4)+12
- The `lea` instruction computes a memory address, but does *not* access a memory location
  - E.g., `leaq 12(%rdx,%rsi,4), %rdi`
  - Keep in mind we're not obligated to use the computed address as an address — we can just use it as an integer
- In general, `lea` can do integer computations of the form $p + (qS) + r$ where S is 0, 1, 2, 4, or 8
- `lea` does not set condition codes (e.g., on overflow)

Addition, subtraction

- `add` and `sub` instructions add and subtract integer values
- Two operands, second operand modified to store the result
- E.g.,

```
movq $1, %r9
movq $2, %r10
addq %r9, %r10
/* %r10 now contains the value 3 */
```

- Overflow is possible!
  - Can detect using condition codes

Increment, decrement

- `inc` and `dec` instructions increment or decrement by 1
- One operand, can be either register or memory
- Examples:

```
incq %rax      /* increment %rax by 1 */
incl 4(%rbp)   /* increment 32 bit value at addr %rbp+4 */
decq %rdi      /* decrement %rdi */
```

- Overflow is possible, check condition codes

Shifts

- Left shift: `sal`
- Right shift: `sar` (arithmetic), `shr` (logical)
  - `sar` shifts in the value of the sign bit, `shr` shifts in zeroes
- Examples:

```
movl $0xFFFF0000, %eax
sall $1, %eax            /* %eax set to 0xFFFE0000 */
movl $0xFFFF0000, %eax
sarl $1, %eax            /* %eax set to 0xFFFF8000 */
movl $0xFFFF0000, %eax
shrl $1, %eax            /* %eax set to 0x7FFF8000 */
```

Bitwise logical operations

- Two-operand logical operations: `and`, `or`, `xor`
- Unary logical operation: `not`
- Examples:

```
/* Note: 0x30 = 00110000b,
         0x50 = 01010000b */
movb $0x30, %al; movb $0x50, %bl
andb %bl, %al      /* set %al=0x10 (00010000b) */
movb $0x30, %al; movb $0x50, %bl
orb %bl, %al       /* set %al=0x70 (01110000b) */
movb $0x30, %al; movb $0x50, %bl
xorb %bl, %al      /* set %al=0x60 (01100000b) */
movb $0x30, %al
notb %al           /* set %al=0xCF (11001111b) */
```

Multiplication

- Two forms of `imul` instruction
- Two operand: multiply operands and truncate
  - Example:

```
imulq %rdi, %rsi  /* set %rsi to %rdi * %rsi,
                     truncated to 64 bits */
```

- One operand: multiply 64 bit operand and value in `%rax`, 128-bit result in `%rdx:%rax`
  - Signed (`imulq`) and unsigned (`mulq`) variants
  - Example:

```
mulq %rdi         /* set %rdx:%rax to unsigned product
                     %rax * %rsi */
```

Division