

CMSC216: Assembly Basics and x86-64

Chris Kauffman

*Last Updated:
Tue Mar 11 09:22:56 AM EDT 2025*

Logistics

Reading Bryant/O'Hallaron

- ▶ Skim Ch 2.7-8: Floating Point Layout
- ▶ Now Ch 3.1-7: Assembly, Arithmetic, Control
- ▶ Later Ch 3.8-11: Arrays, Structs, Floats
- ▶ Any overview guide to x86-64 assembly instructions such as [Brown University's x64 Cheat Sheet](#)

Assignments

- ▶ P2: Due Fri 14-Mar-2025
- ▶ Lab06: Assembly Coding
- ▶ HW06: Assembly Debugging

Both relevant to P3

Goals

- ▶ Floating Point Layout (20min)
- ▶ Assembly Basics
- ▶ x86-64 Overview

Announcements

P1 / Exam 1 Grades Posted

Regrade requests open through Mon 24-Mar (after spring break)

Configure VS Code for x86-64 Assembly Code

This video installs a good plugin, has a few other tips

<https://youtu.be/AgmXUF0EgIw>

GDB Reverse Execution

Material added in the GDB Quick Guide, works for both C and Assembly programs on most platforms (including x86-64)

<https://kauffman77.github.io/tutorials/gdb.html#advanced>

The **Many** Assembly Languages

- ▶ Most **microprocessors** are created to understand a **binary machine language**
- ▶ Machine Language provides means to manipulate internal memory, perform arithmetic, etc.
- ▶ The Machine Language of one processor is **not understood** by other processors

MOS Technology 6502

- ▶ 8-bit operations, limited addressable memory, **1 general purpose register**, powered notable gaming systems in the 1980s
- ▶ Apple IIe, Atari 2600, Commodore
- ▶ Nintendo Entertainment System / Famicom

IBM Cell Microprocessor

- ▶ Developed in early 2000s, 64-bit, many cores (execution elements), many registers (32 on the PPE), large addressable space, fast multimedia performance, is a **pain** to program
- ▶ Playstation 3 and Blue Gene Supercomputer

Assemblers and Compilers



- ▶ **Compiler**: chain of tools that translate high level languages to lower ones, may perform optimizations
- ▶ **Assembler**: translates text description of the machine code to binary, formats for execution by processor, late compiler stage
- ▶ **Consequence**: The compiler can **generate assembly code**
- ▶ Generated assembly is a pain to read but is often quite fast
- ▶ **Consequence**: A compiler on an Intel chip can generate assembly code for a different processor, **cross compiling**

Our focus: The x86-64 Assembly Language

- ▶ x86-64 Targets Intel/AMD chips with 64-bit word size
Reminder: 64-bit “word size” \approx size of pointers/addresses
- ▶ Lineage of x86 family
 - ▶ 1970s: 16-bit systems like Intel 8086
 - ▶ 1990s: IA32 (Intel 32-bit systems like 80386 and 80486)
 - ▶ 2000s: x86-64 (64-bit extension by AMD)
- ▶ x86-64 is backwards compatibility, consequently much cruft
 - ▶ Can run compiled code from the 70's / 80's on modern processors without much trouble BUT means 50-year-old instructions must be preserved
 - ▶ x86-64 is not the assembly language you would design from scratch today, it's the assembly you have to code against
 - ▶ RISC-V is a new assembly language that is “clean” as it has no history to support (and few CPUs run it)
- ▶ **Warning:** Lots of information available on the web for Intel assembly programming **BUT** some of it is dated, IA32 info which may not work on 64-bit systems

x86-64 Assembly Language Syntax(es)

- ▶ Different assemblers understand different syntaxes for the same assembly language
- ▶ GCC use the GNU Assembler (GAS, command 'as file.s')
- ▶ GAS and Textbook favor AT&T syntax so **we will too**
- ▶ NASM assembler favors Intel, may see this online

AT&T Syntax (Our Focus)

```
multstore:
    pushq    %rbx
    movq     %rdx, %rbx
    call     mult2@PLT
    movq     %rax, (%rbx)
    popq     %rbx
    ret
```

- ▶ Use of % to indicate registers
- ▶ Use of q/l/w/b to indicate 64 / 32 / 16 / 8-bit operands

Intel Syntax

```
multstore:
    push     rbx
    mov      rbx, rdx
    call     mult2@PLT
    mov      QWORD PTR [rbx], rax
    pop      rbx
    ret
```

- ▶ Register names are bare
- ▶ Use of QWORD etc. to indicate operand size

Generating Assembly from C Code

- ▶ `gcc -S file.c` will stop compilation at assembly generation
- ▶ Leaves assembly code in `file.s`
 - ▶ `file.s` and `file.S` conventionally assembly code though sometimes `file.asm` is used
- ▶ By default, compiler generates code that is often difficult for humans to interpret, may include re-arrangements, “conservative” compatibility assembly, etc. increasing size of assembly considerably
- ▶ `gcc -Og file.c`: optimize for debugging, generally makes it easier to read generated assembly, aligns somewhat more closely to C code

Example of Generating Assembly from C

```
>> cat exchange.c                                     # show C file to be translated
// exchange.c: sample C function
// to compile to assembly
long exchange(long *xp, long y){
    long x = *xp;
    *xp = y;
    return x;
}

>> gcc -Og -S exchange.c                             # Compile to show assembly
                                                    # -Og: debugging level optimization
                                                    # -S: only output assembly

>> cat exchange.s                                     # show assembly output
    .file      "exchange.c"
    .text
    .globl    exchange
    .type     exchange, @function

exchange:
.LFB0:
    .cfi_startproc
    movq      (%rdi), %rax                # pointer derefs in assembly
    movq      %rsi, (%rdi)              # uses registers
    ret
    .cfi_endproc

.LFE0:
    .size     exchange, .-exchange
    .ident    "GCC: (GNU) 11.1.0"
    .section   .note.GNU-stack,"",@progbits
```

gcc -Og -S mstore.c

```
> cat mstore.c                                # show a C file
long mult2(long a, long b);
void multstore(long x, long y, long *dest){
    long t = mult2(x, y);
    *dest = t;
}

> gcc -Og -S mstore.c                          # Compile to show assembly
                                              # -Og: debugging level optimization
                                              # -S: only output assembly

> cat mstore.s                                # show assembly output
    .file      "mstore.c"
    .text
    .globl    multstore
    .type     multstore, @function
multstore:
.LFB0:
    .cfi_startproc
    pushq    %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq     %rdx, %rbx
    call     mult2@PLT
    movq     %rax, (%rbx)
    popq     %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc

# assembler directives
# assembly instruction
# directives

# assembly instructions
# function call

# function return
```

Every Programming Language

Look for the following as it should almost always be there

- ☐ Comments
- ☐ Statements/Expressions
- ☐ Variable Types
- ☐ Assignment
- ☐ Basic Input/Output
- ☐ Function Declarations
- ☐ Conditionals (if-else)
- ☐ Iteration (loops)
- ☐ Aggregate data (arrays, structs, objects, etc)
- ☐ Library System

Exercise: Examine col_simple_asm.s

Take a simple sample problem to demonstrate assembly:

Computes Collatz Sequence starting at $n=10$:

*if n is ODD $n=n*3+1$; else $n=n/2$.*

*Return the number of steps to converge to 1 as the **return code** from `main()`*

The following codes solve this problem

| Code | Notes |
|------------------|---|
| col_simple_asm.s | Hand-coded assembly for obvious algorithm Straight-forward reading |
| col_unsigned.c | Unsigned C version Generated assembly is reasonably readable |
| col_signed.c | Signed C version Generated assembly is ... interesting |

- ▶ Kauffman will Compile/Run code
- ▶ Students should study the code and predict what lines do
- ▶ Illustrate tricks associated with gdb and assembly

Exercise: col_simple_asm.s

```
1  ### Compute Collatz sequence starting at 10 in assembly.
2  .section .text
3  .globl main
4  main:
5      movl    $0, %r8d      # int steps = 0;
6      movl    $10, %ecx     # int n = 10;
7  .LOOP:
8      cmpl    $1, %ecx      # while(n > 1){ // immediate must be first
9      jle     .END          #   n <= 1 exit loop
10     movl    $2, %esi      #   divisor in esi
11     movl    %ecx,%eax     #   prep for division: must use edx:eax
12     cqto    %ecx          #   extend sign from eax to edx
13     idivl   %esi          #   divide edx:eax by esi
14             #   eax has quotient, edx remainder
15     cmpl    $1,%edx       #   if(n % 2 == 1) {
16     jne     .EVEN         #       not equal, go to even case
17  .ODD:
18     imull   $3, %ecx      #       n = n * 3
19     incl    %ecx          #       n = n + 1 OR n++
20     jmp     .UPDATE       #   }
21  .EVEN:
22             #   else{
23     sarl    $1,%ecx       #       n = n / 2; via right shift
24             #   }
25  .UPDATE:
26     incl    %r8d          #   steps++;
27     jmp     .LOOP        # }
28  .END:
29     movl    %r8d, %eax    # r8d is steps, move to eax for return value
30     ret
```

Answers: x86-64 Assembly Basics for AT&T Syntax

- ▶ *Comments* are one-liners starting with #
- ▶ *Statements*: each line does ONE thing, frequently text representation of an assembly instruction

```
movq    %rdx, %rbx    # move rdx register to rbx
```

- ▶ Assembler directives and labels are also possible:

```
.global  multstore    # notify linker of location multstore
multstore:            # label beginning of multstore section
    blah blah blah    # instructions in this this section
```

- ▶ *Variables*: mainly **registers**, also memory ref'd by registers maybe some named global locations
- ▶ *Assignment*: instructions like movX that put bits into registers and memory
- ▶ *Conditionals/Iteration*: assembly instructions that jump to code locations
- ▶ *Functions*: code locations that are **labeled** and global
- ▶ *Aggregate data*: none, use the stack/multiple registers
- ▶ *Library System*: link to other code

So what *are* these Registers?

- ▶ Memory locations directly wired to the CPU
- ▶ Usually very fast to access, faster than **main memory**
- ▶ Most instructions involve registers, access or change reg val

Example: Adding Together Integers

- ▶ Ensure registers have desired values in them
- ▶ Issue an addX instruction involving the two registers
- ▶ Result will be stored in a register

```
addl %eax, %ebx  
# add ints in eax and ebx, store result in ebx
```

```
addq %rcx, %rdx  
# add longs in rcx and rdx, store result in rdx
```

- ▶ Note instruction and register names indicate whether 32-bit int or 64-bit long are being added

x86-64 “General Purpose” Registers

Many “general purpose” registers have special purposes and conventions associated such as

- ▶ Return Value:
%rax / %eax / %ax
- ▶ Function Args 1 to 6:
%rdi, %rsi, %rdx,
%rcx, %r8, %r9
- ▶ Stack Pointer (top of stack): %rsp
- ▶ Old Code Base Pointer:
%rbp, **historically start of current stack frame** but is not used that way in modern codes

Note: There are also Special Registers like %rip and %eflags which we will discuss later.

| 64-bit | 32-bit | 16-bit | 8-bit | Notes |
|---------------------|--------|----------------------------|-------|------------|
| %rax | %eax | %ax | %al | Return Val |
| %rbx | %ebx | %bx | %bl | |
| %rcx | %ecx | %cx | %cl | Arg 4 |
| %rdx | %edx | %dx | %dl | Arg 3 |
| %rsi | %esi | %si | %sil | Arg 2 |
| %rdi | %edi | %di | %dil | Arg 1 |
| %rsp | %esp | %sp | %spl | Stack Ptr |
| %rbp | %ebp | %bp | %bpl | Base Ptr? |
| %r8 | %r8d | %r8w | %r8b | Arg 5 |
| %r9 | %r9d | %r9w | %r9b | Arg 6 |
| %r10 | %r10d | %r10w | %r10b | |
| %r11 | %r11d | %r11w | %r11b | |
| %r12 | %r12d | %r12w | %r12b | |
| %r13 | %r13d | %r13w | %r13b | |
| %r14 | %r14d | %r14w | %r14b | |
| %r15 | %r15d | %r15w | %r15b | |
| Caller Save: | | Restore after calling func | | |
| Callee Save: | | Restore before returning | | |

Register Naming Conventions

- ▶ AT&T syntax identifies registers with prefix %
- ▶ Naming convention is a historical artifact
- ▶ Originally 16-bit architectures in x86 had
 - ▶ General registers `ax, bx, cx, dx,`
 - ▶ Special Registers `si, di, sp, bp`
- ▶ *Extended* to 32-bit: `eax, ebx, ..., esi, edi, ...`
- ▶ Grew again to 64-bit: `rax, rbx, ..., rsi, rdi, ...`
- ▶ Added Eight 64-bit regs `r8, r9, ..., r14, r15` with 32-bit portion `r8d, r9d, ..., 16-bit r8w, r9w, ..., etc.`
- ▶ Instructions must match registers sizes:
 - `addw %ax, %bx # word (16-bit)`
 - `addl %eax, %ebx # long word (32-bit)`
 - `addq %rax, %rbx # quad-word (64-bit)`
- ▶ When hand-coding assembly, easy to mess this up, assembler will error out

Hello World in x86-64 Assembly : Not that Easy

- ▶ Non-trivial in assembly because **output is involved**
 - ▶ Try writing `helloworld.c` without `printf()`
- ▶ Output is the business of the **operating system**, always a request to the almighty OS to put something somewhere
 - ▶ **Library call**: `printf("hello");` mangles some bits but eventually results with a ...
 - ▶ **System call**: Unix system call directly implemented in the OS **kernel**, puts bytes into files / onto screen as in
`write(1, buf, 5); // file 1 is screen output`

This gives us several options for hello world in assembly:

1. `hello_printf64.s`: via calling `printf()` which means the C standard library must be (painfully) linked
2. `hello64.s` via direct system `write()` call which means no external libraries are needed: OS knows how to write to files/screen. Use the 64-bit Linux calling convention.
3. `hello32.s` via direct system call using the older 32 bit Linux calling convention which “traps” to the operating system.

(Optional): The OS Privilege: System Calls

- ▶ Most interactions with the outside world happen via Operating System Calls (or just “system calls”)
- ▶ User programs indicate what service they want performed by the OS via making system calls
- ▶ System Calls differ for each language/OS combination
 - ▶ x86-64 Linux: set `%rax` to system call number, set other args in registers, issue `syscall`
 - ▶ IA32 Linux: set `%eax` to system call number, set other args in registers, issue an **interrupt**
 - ▶ C Code on Unix: make system calls via `write()`, `read()` and others (studied in CSCI 4061)
 - ▶ Tables of Linux System Call Numbers
 - ▶ 64-bit (335 calls)
 - ▶ 32-bit (190 calls)
 - ▶ Mac OS X: very similar to the above (it's a Unix)
 - ▶ Windows: use OS wrapper functions
- ▶ OS executes **privileged** code that can manipulate any part of memory, touch internal data structures corresponding to files, do other fun stuff discussed in CSCI 4061 / 5103

Basic Instruction Classes

- ▶ **Remember:** Goal is to understand assembly as a *target* for higher languages, not become expert “assemblists”
- ▶ Means we won't hit all 4,834 pages of the [Intel x86-64 Manual](#)
- ▶ [Brown University's x64 Cheat Sheet](#) has a good overview
- ▶ [x86 Assembly Guide from Yale](#) is also good but is limited to 32-bit coverage

| Kind | Assembly Instructions |
|---------------------------|-------------------------|
| <i>Fundamentals</i> | |
| - Memory Movement | mov |
| - Stack manipulation | push,pop |
| - Addressing modes | (%eax),12(%eax,%ebx)... |
| <i>Arithmetic/Logic</i> | |
| - Arithmetic | add,sub,mul,div,lea |
| - Bitwise Logical | and,or,xor,not |
| - Bitwise Shifts | sal,sar,shr |
| <i>Control Flow</i> | |
| - Compare / Test | cmp,test |
| - Set on result | set |
| - Jumps (Un)Conditional | jmp,je,jne,jl,jg,... |
| - Conditional Movement | cmov,cmovg,... |
| <i>Procedure Calls</i> | |
| - Stack manipulation | push,pop |
| - Call/Return | call,ret |
| - System Calls | syscall |
| <i>Floating Point Ops</i> | |
| - FP Reg Movement | vmov |
| - Conversions | vcvts |
| - Arithmetic | vadd,vsub,vmul,vdiv |
| - Extras | vmins,vmaxs,sqrts |

Data Movement: movX instruction

movX SOURCE, DEST # move/copy source value to dest

Overview

- ▶ Moves data...
 - ▶ Reg to Reg
 - ▶ Mem to Reg
 - ▶ Reg to Mem
 - ▶ Imm to ...
- ▶ Reg: register
- ▶ Mem: main memory
- ▶ Imm: "immediate" value (constant) specified like
 - ▶ \$21 : decimal
 - ▶ \$0x2f9a : hexadecimal
 - ▶ **NOT** 1234 (mem adder)
- ▶ More info on operands next

Examples

64-bit quadword moves

```
movq $4, %rbx    # rbx = 4;
movq %rbx,%rax   # rax = rbx;
movq $10, (%rcx) # *rcx = 10;
```

32-bit longword moves

```
movl $4, %ebx    # ebx = 4;
movl %ebx,%eax   # eax = ebx;
movl $10, (%rcx) # *rcx = 10;
```

Note variations

- ▶ movq for 64-bit (8-byte)
- ▶ movl for 32-bit (4-byte)
- ▶ movw for 16-bit (2-byte)
- ▶ movb for 8-bit (1-byte)

Operands and Addressing Modes

In many instructions like `movX`, operands can have a variety of forms called **addressing modes**, may include constants and memory addresses

| Style | Address Mode | C-like | Notes |
|--------------------------------|--------------|-----------------------------|---|
| \$21 \$0xD2 | immediate | 21 | value of constant like 21 or 0xD2 = 210 |
| %rax (%rax) | register | rax | to/from register contents |
| 8(%rax) | indirect | *rax | reg holds memory address, deref |
| 4(%rdx) | displaced | *(rax+2) rdx->field | base plus constant offset, often used for struct field derefs |
| (%rax,%rbx) | indexed | *(rax+rbx) char_arr[rbx] | base plus offset in given reg actual value of rbx is used, NOT multiplied by sizeof() |
| (%rax,%rbx,4) (%rax,%rbx,8) | scaled index | rax[rbx] rax[rbx] | like array access with sizeof(..)=4 "" with sizeof(..)=8 |
| 1024 | absolute | ... | Absolute address #1024 Rarely used |

Exercise: Show movX Instruction Execution

Code movX_exercise.s

```
movl $16, %eax
movl $20, %ebx
movq $24, %rbx
## POS A

movl %eax,%ebx
movq %rcx,%rax
## POS B

movq $45, (%rdx)
movl $55, 16(%rdx)
## POS C

movq $65, (%rcx,%rbx)
movq $3,%rbx
movq $75, (%rcx,%rbx,8)
## POS D
```

Registers/Memory

INITIAL

| -----+-----+----- | | | |
|-------------------|-------|--|-------|
| REG | %rax | | 0 |
| | %rbx | | 0 |
| | %rcx | | #1024 |
| | %rdx | | #1032 |
| -----+-----+----- | | | |
| MEM | #1024 | | 35 |
| | #1032 | | 25 |
| | #1040 | | 15 |
| | #1048 | | 5 |
| -----+-----+----- | | | |

Lookup...

May need to look up addressing conventions for things like...

```
movX %y,%x    # reg y to reg x
movX $5, (%x)  # 5 to address in %x
```

Answers Part 1/2: movX Instruction Execution

| | | | | | |
|--------------|--------------|---|--------------|--|--------------|
| INITIAL | | movl \$16, %eax movl \$20, %ebx movq \$24, %rbx ## POS A | | movl %eax,%ebx movq %rcx,%rax #WARNING! ## POS B | |
| -----+----- | -----+----- | -----+----- | -----+----- | -----+----- | -----+----- |
| REG VALUE | REG VALUE | REG VALUE | REG VALUE | REG VALUE | REG VALUE |
| %rax 0 | %rax 16 | %rax #1024 | %rax 16 | %rax #1024 | %rax #1024 |
| %rbx 0 | %rbx 24 | %rbx #1024 | %rbx 16 | %rbx #1024 | %rbx #1024 |
| %rcx #1024 | %rcx #1024 | %rcx #1024 | %rcx #1024 | %rcx #1024 | %rcx #1024 |
| %rdx #1032 | %rdx #1032 | %rdx #1032 | %rdx #1032 | %rdx #1032 | %rdx #1032 |
| -----+----- | -----+----- | -----+----- | -----+----- | -----+----- | -----+----- |
| MEM VALUE | MEM VALUE | MEM VALUE | MEM VALUE | MEM VALUE | MEM VALUE |
| #1024 35 | #1024 35 | #1024 35 | #1024 35 | #1024 35 | #1024 35 |
| #1032 25 | #1032 25 | #1032 25 | #1032 25 | #1032 25 | #1032 25 |
| #1040 15 | #1040 15 | #1040 15 | #1040 15 | #1040 15 | #1040 15 |
| #1048 5 | #1048 5 | #1048 5 | #1048 5 | #1048 5 | #1048 5 |
| -----+----- | -----+----- | -----+----- | -----+----- | -----+----- | -----+----- |

#WARNING!: On 64-bit systems, ALWAYS use a 64-bit reg name like %rdx and movq to copy memory addresses; using smaller name like %edx will miss half the memory addressing leading to major memory problems

Answers Part 2/2: movX Instruction Execution

```
movl %eax,%ebx
movq %rcx,%rax #!
```

POS B

| REG | VALUE |
|-------|-------|
| %rax | #1024 |
| %rbx | 16 |
| %rcx | #1024 |
| %rdx | #1032 |
| MEM | VALUE |
| #1024 | 35 |
| #1032 | 25 |
| #1040 | 15 |
| #1048 | 5 |

```
movq $45, (%rdx)
               #1032
movq $55, 16(%rdx)
               16+#1032=#1048
```

POS C

| REG | VALUE |
|-------|-------|
| %rax | #1024 |
| %rbx | 16 |
| %rcx | #1024 |
| %rdx | #1032 |
| MEM | VALUE |
| #1024 | 35 |
| #1032 | 45 |
| #1040 | 15 |
| #1048 | 55 |

```
movq $65, (%rcx,%rbx)
               #1024+16 = #1040
movq $3,%rbx
movq $75, (%rcx,%rbx,8)
               #1024 + 3*8 = #1048
```

POS D

| REG | VALUE |
|-------|-------|
| %rax | #1024 |
| %rbx | 3 |
| %rcx | #1024 |
| %rdx | #1032 |
| MEM | VALUE |
| #1024 | 35 |
| #1032 | 45 |
| #1040 | 65 |
| #1048 | 75 |

`gdb` Assembly: Examining Memory

`gdb` commands `print` and `x` allow one to print/examine memory of interest. Try on `movX_exercises.s`

```
(gdb) tui enable           # TUI mode
(gdb) layout asm           # assembly mode
(gdb) layout reg           # show registers
(gdb) stepi                # step forward by single Instruction
(gdb) print $rax           # print register rax
(gdb) print *($rdx)        # print memory pointed to by rdx
(gdb) print (char *) $rdx  # print as a string (null terminated)
(gdb) x $r8                # examine memory at address in r8
(gdb) x/3d $r8             # same but print as 3 4-byte decimals
(gdb) x/6g $r8             # same but print as 6 8-byte decimals
(gdb) x/s $r8              # print as a string (null terminated)
(gdb) print *((int*) $rsp)  # print top int on stack (4 bytes)
(gdb) x/4d $rsp            # print top 4 stack vars as ints
(gdb) x/4x $rsp            # print top 4 stack vars as ints in hex
```

Many of these tricks are needed to debug assembly.

Register Size and Movement

- ▶ Recall %rax is 64-bit register, %eax is lower 32 bits of it
- ▶ Data movement involving small registers **may NOT overwrite** higher bits in extended register

- ▶ Moving data to low 32-bit regs automatically zeros high 32-bits

```
movabsq $0x1122334455667788, %rax # 8 bytes to %rax
movl $0xAABBCCDD, %eax # 4 bytes to %eax
## %rax is now 0x00000000AABBCCDD
```

- ▶ Moving data to other small regs DOES NOT ALTER high bits

```
movabsq $0x1122334455667788, %rax # 8 bytes to %rax
movw $0xAABB, %ax # 2 bytes to %ax
## %rax is now 0x112233445566AABB
```

- ▶ Gives rise to two other families of movement instructions for moving little registers (X) to big (Y) registers, see movz_examples.s

```
## movzXY move zero extend, movsXY move sign extend
movabsq $0x112233445566AABB,%rdx
movzwl %dx,%rax # %rax is 0x000000000000AABB
movswl %dx,%rax # %rax is 0xFFFFFFFFFFFFAABB
```

Exercise: movX differences in Main Memory

| Instr | # bytes |
|-------|---------|
| movb | 1 byte |
| movw | 2 bytes |
| movl | 4 bytes |
| movq | 8 bytes |

Show the result of each of the following copies to main memory in sequence.

```
movl    %eax,    (%rsi) #1
movq    %rax,    (%rsi) #2
movb    %cl,     (%rsi) #3
movw    %cx,     2(%rsi) #4
movl    %ecx,    4(%rsi) #5
movw    4(%rsi), %ax    #6
```

INITIAL

| | |
|-------|--------------------|
| REG | |
| rax | 0x00000000DDCCBBAA |
| rcx | 0x000000000000FFEE |
| rsi | #1024 |
| MEM | |
| #1024 | 0x00 |
| #1025 | 0x11 |
| #1026 | 0x22 |
| #1027 | 0x33 |
| #1028 | 0x44 |
| #1029 | 0x55 |
| #1030 | 0x66 |
| #1031 | 0x77 |
| #1032 | 0x88 |
| #1033 | 0x99 |

Answers: movX to Main Memory 1/2

| REG | | |
|-----|--------------------|--|
| rax | 0x00000000DDCCBBAA | |
| rcx | 0x000000000000FFEE | |
| rsi | #1024 | |

| movl | %eax, (%rsi) | #1 4 bytes rax -> #1024 |
|------|---------------|-------------------------|
| movq | %rax, (%rsi) | #2 8 bytes rax -> #1024 |
| movb | %cl, (%rsi) | #3 1 byte rcx -> #1024 |
| movw | %cx, 2(%rsi) | #4 2 bytes rcx -> #1026 |
| movl | %ecx, 4(%rsi) | #5 4 bytes rcx -> #1028 |
| movw | 4(%rsi), %ax | #6 2 bytes #1024 -> rax |

| INITIAL | #1 movl %eax, (%rsi) | #2 movq %rax, (%rsi) | #3 movb %cl, (%rsi) |
|--------------|-------------------------|-------------------------|------------------------|
| MEM | MEM | MEM | MEM |
| #1024 0x00 | #1024 0xAA | #1024 0xAA | #1024 0xEE |
| #1025 0x11 | #1025 0xBB | #1025 0xBB | #1025 0xBB |
| #1026 0x22 | #1026 0xCC | #1026 0xCC | #1026 0xCC |
| #1027 0x33 | #1027 0xDD | #1027 0xDD | #1027 0xDD |
| #1028 0x44 | #1028 0x44 | #1028 0x00 | #1028 0x00 |
| #1029 0x55 | #1029 0x55 | #1029 0x00 | #1029 0x00 |
| #1030 0x66 | #1030 0x66 | #1030 0x00 | #1030 0x00 |
| #1031 0x77 | #1031 0x77 | #1031 0x00 | #1031 0x00 |
| #1032 0x88 | #1032 0x88 | #1032 0x88 | #1032 0x88 |
| #1033 0x99 | #1033 0x99 | #1033 0x99 | #1033 0x99 |

Answers: movX to Main Memory 2/2

| | | | |
|-----|--------------------|------|---------------------------------------|
| REG | | movl | %eax, (%rsi) #1 4 bytes rax -> #1024 |
| rax | 0x00000000DDCCBBAA | movq | %rax, (%rsi) #2 8 bytes rax -> #1024 |
| rcx | 0x000000000000FFEE | movb | %cl, (%rsi) #3 1 byte rcx -> #1024 |
| rsi | #1024 | movw | %cx, 2(%rsi) #4 2 bytes rcx -> #1026 |
| | | movl | %ecx, 4(%rsi) #5 4 bytes rcx -> #1028 |
| | | movw | 4(%rsi), %ax #6 2 bytes #1024 -> rax |

| #3 | #4 | #5 | #6 |
|------------------|-------------------|--------------------|-------------------|
| movb %cl, (%rsi) | movw %cx, 2(%rsi) | movl %ecx, 4(%rsi) | movw 4(%rsi), %ax |
| MEM | MEM | MEM | MEM |
| #1024 0xEE | #1024 0xEE | #1024 0xEE | #1024 0xEE |
| #1025 0xBB | #1025 0xBB | #1025 0xBB | #1025 0xBB |
| #1026 0xCC | #1026 0xEE | #1026 0xEE | #1026 0xEE |
| #1027 0xDD | #1027 0xFF | #1027 0xFF | #1027 0xFF |
| #1028 0x00 | #1028 0x00 | #1028 0xEE | #1028 0xEE |
| #1029 0x00 | #1029 0x00 | #1029 0xFF | #1029 0xFF |
| #1030 0x00 | #1030 0x00 | #1030 0x00 | #1030 0x00 |
| #1031 0x00 | #1031 0x00 | #1031 0x00 | #1031 0x00 |
| #1032 0x88 | #1032 0x88 | #1032 0x88 | #1032 0x88 |
| #1033 0x99 | #1033 0x99 | #1033 0x99 | #1033 0x99 |

| rax | 0x00000000DDCCFFEE |

addX : A Quintessential ALU Instruction

`addX B, A` $\# A = A+B$ ► Addition represents most 2-operand ALU instructions well

OPERANDS:

`addX %reg, %reg`
`addX (%mem), %reg`
`addX %reg, (%mem)`
`addX $con, %reg`
`addX $con, (%mem)`

- Second operand A is modified by first operand B, No change to B
- Variety of register, memory, constant combinations honored
- `addX` has variants for each register size: `addq`, `addl`, `addw`, `addb`

$\#$ No `mem+mem` or `con+con`

EXAMPLES:

| | |
|--------------------------------|-----------------------|
| <code>addq %rdx, %rcx</code> | $\# rcx = rcx + rdx$ |
| <code>addl %eax, %ebx</code> | $\# ebx = ebx + eax$ |
| <code>addq \$42, %rdx</code> | $\# rdx = rdx + 42$ |
| <code>addl (%rsi), %edi</code> | $\# edi = edi + *rsi$ |
| <code>addw %ax, (%rbx)</code> | $\# *rbx = *rbx + ax$ |
| <code>addq \$55, (%rbx)</code> | $\# *rbx = *rbx + 55$ |

`addl (%rsi,%rax,4), %edi` $\# edi = edi + rsi[rax] \text{ (int)}$

Optional Exercise: Addition

Show the results of the following addX/movX ops at each of the specified positions

```
addq $1,%rcx      # con + reg
addq %rbx,%rax     # reg + reg
## POS A

addq (%rdx),%rcx   # mem + reg
addq %rbx,(%rdx)   # reg + mem
addq $3,(%rdx)     # con + mem
## POS B

addl $1,(%r8,%r9,4) # con + mem
addl $1,%r9d       # con + reg
addl %eax,(%r8,%r9,4) # reg + mem
addl $1,%r9d       # con + reg
addl (%r8,%r9,4),%eax # mem + reg
## POS C
```

| INITIAL | | |
|-------------|--|-------|
| -----+----- | | |
| REGS | | |
| %rax | | 15 |
| %rbx | | 20 |
| %rcx | | 25 |
| %rdx | | #1024 |
| %r8 | | #2048 |
| %r9 | | 0 |
| -----+----- | | |
| MEM | | |
| #1024 | | 100 |
| ... | | ... |
| #2048 | | 200 |
| #2052 | | 300 |
| #2056 | | 400 |
| -----+----- | | |

Answers: Addition

| INITIAL | | POS A | | POS B | | POS C | |
|-------------|-------|-------------|-------|-------------|-------|-------------|-------|
| REG | | REG | | REG | | REG | |
| %rax | 15 | %rax | 35 | %rax | 35 | %rax | 435 |
| %rbx | 20 | %rbx | 20 | %rbx | 20 | %rbx | 20 |
| %rcx | 25 | %rcx | 26 | %rcx | 126 | %rcx | 126 |
| %rdx | #1024 | %rdx | #1024 | %rdx | #1024 | %rdx | #1024 |
| %r8 | #2048 | %r8 | #2048 | %r8 | #2048 | %r8 | #2048 |
| %r9 | 0 | %r9 | 0 | %r9 | 0 | %r9 | 2 |
| -----+----- | | -----+----- | | -----+----- | | -----+----- | |
| MEM | | MEM | | MEM | | MEM | |
| #1024 | 100 | #1024 | 100 | #1024 | 123 | #1024 | 123 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| #2048 | 200 | #2048 | 200 | #2048 | 200 | #2048 | 201 |
| #2052 | 300 | #2052 | 300 | #2052 | 300 | #2052 | 335 |
| #2056 | 400 | #2056 | 400 | #2056 | 400 | #2056 | 400 |
| -----+----- | | -----+----- | | -----+----- | | -----+----- | |

```
addq $1,%rcx
addq %rbx,%rax
```

```
addq (%rdx),%rcx
addq %rbx,(%rdx)
addq $3,(%rdx)
```

```
addl $1,(%r8,%r9,4)
addl $1,%r9d
addl %eax,(%r8,%r9,4)
addl $1,%r9d
addl (%r8,%r9,4),%eax
```

The Other ALU Instructions

- ▶ Most ALU instructions follow the same pattern as `addX`: two operands, second gets changed.
- ▶ Some one operand instructions as well.

| Instruction | Name | Effect | Notes |
|-------------------------|-------------|------------------|---------------------------------------|
| <code>addX B, A</code> | Add | $A = A + B$ | Two Operand Instructions |
| <code>subX B, A</code> | Subtract | $A = A - B$ | |
| <code>imulX B, A</code> | Multiply | $A = A * B$ | <i>Has a limited 3-arg variant</i> |
| <code>andX B, A</code> | And | $A = A \& B$ | |
| <code>orX B, A</code> | Or | $A = A B$ | |
| <code>xorX B, A</code> | Xor | $A = A \wedge B$ | |
| <code>salX B, A</code> | Shift Left | $A = A \ll B$ | B is constant or <code>%cl</code> reg |
| <code>shlX B, A</code> | | $A = A \ll B$ | |
| <code>sarX B, A</code> | Shift Right | $A = A \gg B$ | Arithmetic: Sign carry |
| <code>shrX B, A</code> | | $A = A \gg B$ | Logical: Zero carry |
| <code>incX A</code> | Increment | $A = A + 1$ | One Operand Instructions |
| <code>decX A</code> | Decrement | $A = A - 1$ | |
| <code>negX A</code> | Negate | $A = -A$ | |
| <code>notX A</code> | Complement | $A = \sim A$ | |

leax: Load Effective Address

- ▶ Memory addresses must often be loaded into registers
- ▶ Often done with a leax, usually leaq in 64-bit platforms
- ▶ Sort of like “address-of” op & in C but a bit more general

INITIAL

| -----+----- | |
|-------------|-------|
| REG | VAL |
| rax | 0 |
| rcx | 2 |
| rdx | #1024 |
| rsi | #2048 |
| -----+----- | |
| MEM | |
| #1024 | 15 |
| #1032 | 25 |
| ... | |
| #2048 | 200 |
| #2052 | 300 |
| #2056 | 400 |
| -----+----- | |

leax_examples.s:

```
movq 8(%rdx),%rax      # rax = *(rdx+1)      = 25
leaq 8(%rdx),%rax      # rax = rdx+1         = #1032
movl (%rsi,%rcx,4),%eax # rax = rsi[rcx]      = 400
leaq (%rsi,%rcx,4),%rax # rax = &(rsi[rcx]) = #2056
```

Compiler sometimes uses leax for multiplication as it is usually faster than imulX but less readable.

```
# Odd Collatz update n = 3*n+1
#READABLE with imulX      #OPTIMIZED with leax:
imul $3,%eax              leal 1(%eax,%eax,2),%eax
addl $1,%eax
# eax = eax*3 + 1         # eax = eax + 2*eax + 1,
# 3-4 cycles              # 1 cycle
```



Division: It's a Pain (1/2)

- ▶ `idivX` operation has some special rules
- ▶ Dividend must be in the `rax` / `eax` / `ax` register
- ▶ Sign extend to `rdx` / `edx` / `dx` register with `cqto`
- ▶ `idivX` takes one **register** argument which is the divisor
- ▶ At completion
 - ▶ `rax` / `eax` / `ax` holds quotient (integer part)
 - ▶ `rdx` / `edx` / `dx` holds the remainder (leftover)

```
### division.s:
movl $15, %eax          # set eax to int 15
cqto                    # extends 0 sign bit (positive) to edx
## combined 64-bit register %edx:%eax is
## eax: 0x00000000 0000000F = 15
## edx: 0x00000000 00000000 = 0
movl $2, %esi           # set esi to 2
idivl %esi              # divide combined register by 2
## 15 div 2 = 7 rem 1
## %eax == 7, quotient
## %edx == 1, remainder

# answer in eax, return
ret
```

Compiler avoids division whenever possible: compile `col_unsigned.c` and `col_signed.c` to see some tricks.

Division: It's a Pain (2/2)

- ▶ When performing division on 8-bit or 16-bit quantities, use instructions to sign extend small reg to all rax register

```
### division with 16-bit shorts from division.s
movq $0,%rax      # set rax to all 0's
movq $0,%rdx      # set rdx to all 0's
                  # rax = 0x00000000 00000000
                  # rdx = 0x00000000 00000000
movw $-17, %ax    # set ax to short -17
                  # rax = 0x00000000 0000FFEF
                  # rdx = 0x00000000 00000000
cwtl              # "convert word to long" sign extend ax to eax
                  # rax = 0x00000000 FFFFFFFF
                  # rdx = 0x00000000 00000000
cltq              # "convert long to quad" sign extend eax to rax
                  # rax = 0xFFFFFFFF FFFFFFFF
                  # rdx = 0x00000000 00000000
cqto              # sign extend rax to rdx
                  # rax = 0xFFFFFFFF FFFFFFFF
                  # rdx = 0xFFFFFFFF FFFFFFFF
movq $3, %rcx     # set rcx to long 3
idivq %rcx        # divide combined rax/rdx register by 3
                  # rax = 0xFFFFFFFF FFFFFFFB = -5 (quotient)
                  # rdx = 0xFFFFFFFF FFFFFFFE = -2 (remainder)
```