CSCI 2021: Assembly Basics and x86-64

Chris Kauffman

Last Updated:
Tue Oct 19 11:08:09 AM CDT 2021
Logistics

Reading Bryant/O’Hallaron

- 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

Goals

- Assembly Basics
- x86-64 Overview

Lab / HW

- Lab06: GDB Basics
- HW06: Assembly Basics
- HW07: Assembly+GDB
- Lab07: Assembly Functions

Project 2: Due Wed 10/20

- Problem 1: Bit shift operations (50%)
- Problem 2: Puzzlebox via debugger (50% + makeup)

NOTE: Line Count Limits
GDB: The GNU Debugger

- Overview for C and Assembly Programs here: [https://www-users.cs.umn.edu/~kauffman/2021/gdb](https://www-users.cs.umn.edu/~kauffman/2021/gdb)
- Most programming environments feature a Debugger
  - Java, Python, OCaml, etc.
- GDB works well C and Assembly programs
- Features in P2 (C programs) and P3 (Assembly Programs)
- P2 Demo has some basics for C programs including
  - TUI Mode
  - Breakpoint / Continue
  - Next / Step
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, many cores (execution elements), many registers, 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” ≈ size of pointers/addresses*
- Descended from IA32: Intel Architecture 32-bit systems
- IA32 descended from earlier 16-bit systems like Intel 8086
- There is a **LOT** of cruft in x86-64 for backwards compatibility
  - Can run compiled code from the 70’s / 80’s on modern processors without much trouble
  - x86-64 is not the assembly language you would design from scratch today
- Will touch on evolution of Intel Assembly as we move forward
- **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
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)

```assembly
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

```assembly
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 performs lots of optimizations to code
- `gcc -Og file.c`: disable optimizations to make it easier to debug, generated assembly is slightly more readable assembly
gcc -Og -S mstore.c

> cat mstore.c
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
.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
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

<table>
<thead>
<tr>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>col_simple_asm.s</code></td>
<td>Hand-coded assembly for obvious algorithm</td>
</tr>
<tr>
<td></td>
<td>Straight-forward reading</td>
</tr>
<tr>
<td><code>col_unsigned.c</code></td>
<td>Unsigned C version</td>
</tr>
<tr>
<td></td>
<td>Generated assembly is reasonably readable</td>
</tr>
<tr>
<td><code>col_signed.c</code></td>
<td>Signed C version</td>
</tr>
<tr>
<td></td>
<td>Generated assembly is … interesting</td>
</tr>
</tbody>
</table>

- 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

```assembly
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 # 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: # else{
22         sarl $1,%ecx # n = n / 2; via right shift
23          .UPDATE:
24         incl %r8d # steps++;
25         jmp .LOOP # }
26     .END:
27       movl %r8d, %eax # r8d is steps, move to eax for return value
28     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
  
  ```assembly
  movq %rdx, %rbx  # move rdx register to rbx
  ```

- Assembler directives and labels are also possible:
  
  ```assembly
  .globl multstore  # notify linker of location multstore
  multstore:        # beginning of multstore section
  blah blah blah
  ```

- **Variables**: mainly **registers**, also memory ref’d by registers maybe some named global locations

- **Assignment**: instructions like `movX` that put move 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

```c
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
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 additional 64-bit regs `r8, r9, ..., r14, r15` with 32-bit `r8d, r9d, ...` and 16-bit `r8w, r8w...`
- Instructions must match registers sizes:
  - `addw %ax, %bx` # words (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
x86-64 “General Purpose” Registers

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

- `%rax | %eax | %ax` contains return value from functions
- `%rdi, %rsi, %rdx, %rcx, %r8, %r9` contain first 6 arguments in function calls
- `%rsp` is top of the stack
- `%rbp` (base pointer) may be the beginning of current stack but is often optimized away by the compiler

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

**Caller Save:** Restore after calling func
**Callee Save:** Restore before returning
Hello World in x86-64 Assembly

- 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.
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 (328 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

- **x86 Assembly Guide from Yale** summarizes well though is 32-bit only, function calls different

- **Remember**: Goal is to understand assembly as a *target* for higher languages, not become expert “assemblists”

- Means we won’t hit all 5,038 pages of the Intel x86-64 Manual

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

\[
\text{movX SOURCE, DEST} \quad \# \text{move source value to destination}
\]

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
  - $21\ : \text{decimal}$
  - $0x2f9a\ : \text{hexadecimal}$
  - NOT 1234 (mem adder)
- More info on operands next

Examples

## 64-bit quadword moves

\[
\begin{align*}
\text{movq} & \quad $4, \%\text{rbx} \quad \# \text{rbx} = 4; \\
\text{movq} & \quad \%\text{rbx},\%\text{rax} \quad \# \text{rax} = \text{rbx}; \\
\text{movq} & \quad $10, (\%\text{rcx}) \quad \# \text{rcx} = 10;
\end{align*}
\]

## 32-bit longword moves

\[
\begin{align*}
\text{movl} & \quad $4, \%\text{ebx} \quad \# \text{ebx} = 4; \\
\text{movl} & \quad \%\text{ebx},\%\text{eax} \quad \# \text{eax} = \text{ebx}; \\
\text{movl} & \quad $10, (\%\text{ecx}) \quad \# \text{ecx} = 10; >:-(
\end{align*}
\]

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 `mov X`, operands can have a variety of forms called **addressing modes**, may include constants and memory addresses

<table>
<thead>
<tr>
<th>Style</th>
<th>Address Mode</th>
<th>C-like</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$21</td>
<td>immediate</td>
<td>21</td>
<td>value of constant like 21 or 0xD2 = 210</td>
</tr>
<tr>
<td>$0x2D</td>
<td>immediate</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>%rax</td>
<td>register</td>
<td>rax</td>
<td>to/from register contents</td>
</tr>
<tr>
<td>(%rax)</td>
<td>indirect</td>
<td>*rax</td>
<td>reg holds memory address, deref</td>
</tr>
<tr>
<td>8(%rax)</td>
<td>displaced</td>
<td>*(rax+2)</td>
<td>base plus constant offset,</td>
</tr>
<tr>
<td>-4(%rax)</td>
<td>displaced</td>
<td>*(rax-1)</td>
<td>C examples presume sizeof(..)=4</td>
</tr>
<tr>
<td>(%rax,%rbx)</td>
<td>indexed</td>
<td>*(rax+rbx)</td>
<td>base plus offset in given reg</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>actual value of rbx is used, NOT multi</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>plied by sizeof()</td>
</tr>
<tr>
<td>(%rax,%rbx,4)</td>
<td>scaled index</td>
<td>rax[rbx]</td>
<td>like array access with sizeof(..)=4</td>
</tr>
<tr>
<td>(%rax,%rbx,8)</td>
<td>scaled index</td>
<td>rax[rbx]</td>
<td>“” with sizeof(..)=8</td>
</tr>
<tr>
<td>1024</td>
<td>absolute</td>
<td>...</td>
<td>Absolute address #1024</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rarely used</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>REG</th>
<th>%rax</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>#1024</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>#1032</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEM</th>
<th>#1024</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#1032</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>#1040</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>#1048</td>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>INITIAL</th>
<th># POS A</th>
<th># POS B</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG</td>
<td>VALUE</td>
<td>REG</td>
</tr>
<tr>
<td>%rax</td>
<td>0</td>
<td>%rax</td>
</tr>
<tr>
<td>%rbx</td>
<td>0</td>
<td>%rbx</td>
</tr>
<tr>
<td>%rcx</td>
<td>#1024</td>
<td>%rcx</td>
</tr>
<tr>
<td>%rdx</td>
<td>#1032</td>
<td>%rdx</td>
</tr>
<tr>
<td>MEM</td>
<td>VALUE</td>
<td>MEM</td>
</tr>
<tr>
<td>#1024</td>
<td>35</td>
<td>#1024</td>
</tr>
<tr>
<td>#1032</td>
<td>25</td>
<td>#1032</td>
</tr>
<tr>
<td>#1040</td>
<td>15</td>
<td>#1040</td>
</tr>
<tr>
<td>#1048</td>
<td>5</td>
<td>#1048</td>
</tr>
</tbody>
</table>

#!: 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 ## POS C ## POS D
|-------+-------| |-------+-------| |-------+-------|
| REG   | VALUE   | | REG   | VALUE   | | REG   | VALUE   |
| %rax  | #1024   | | %rax  | #1024   | | %rax  | #1024   |
| %rbx  | 16      | | %rbx  | 16      | | %rbx  | 3      |
| %rcx  | #1024   | | %rcx  | #1024   | | %rcx  | #1024   |
| %rdx  | #1032   | | %rdx  | #1032   | | %rdx  | #1032   |
|-------+-------| |-------+-------| |-------+-------|
| MEM   | VALUE   | | MEM   | VALUE   | | MEM   | VALUE   |
| #1024 | 35      | | #1024 | 35      | | #1024 | 35      |
| #1032 | 25      | | #1032 | 45      | | #1032 | 45      |
| #1040 | 15      | | #1040 | 15      | | #1040 | 65      |
| #1048 | 5       | | #1048 | 55      | | #1048 | 75      |
|-------+-------| |-------+-------| |-------+-------|

movq $45,(%rdx)

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

movq $65,(%rcx,%rbx)

movq $3,%rbx
movq $75,(%rcx,%rbx,8)

#1024+16 = #1040
#1024 + 3*8 = #1048
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) stepei # 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
  movzwq  %dx,%rax  # %rax is 0x000000000000AABB
  movswq  %dx,%rax  # %rax is 0xFFFFFFFFFFFFAABB
  ```
Exercise: movX differences in Memory

<table>
<thead>
<tr>
<th>Instr</th>
<th># bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb</td>
<td>1 byte</td>
</tr>
<tr>
<td>movw</td>
<td>2 bytes</td>
</tr>
<tr>
<td>movl</td>
<td>4 bytes</td>
</tr>
<tr>
<td>movq</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

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
### Answers: movX to Main Memory 1/2

<table>
<thead>
<tr>
<th>REG</th>
<th>movl %eax, (%rsi) #1 4 bytes rax -&gt; #1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>movq %rax, (%rsi) #2 8 bytes rax -&gt; #1024</td>
</tr>
<tr>
<td>rcx</td>
<td>movb %cl, (%rsi) #3 1 byte rcx -&gt; #1024</td>
</tr>
<tr>
<td>rsi</td>
<td>movw %cx, 2(%rsi) #4 2 bytes rcx -&gt; #1026</td>
</tr>
<tr>
<td>-----</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>rsi</td>
<td>movl %ecx, 4(%rsi) #5 4 bytes rcx -&gt; #1028</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INITIAL</th>
<th>#1 movl %eax, (%rsi)</th>
<th>#2 movq %rax, (%rsi)</th>
<th>#3 movb %cl, (%rsi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
</tr>
<tr>
<td>#1024</td>
<td>0x00</td>
<td>#1024</td>
<td>0xAA</td>
</tr>
<tr>
<td>#1025</td>
<td>0x11</td>
<td>#1025</td>
<td>0xBB</td>
</tr>
<tr>
<td>#1026</td>
<td>0x22</td>
<td>#1026</td>
<td>0xCC</td>
</tr>
<tr>
<td>#1027</td>
<td>0x33</td>
<td>#1027</td>
<td>0xDD</td>
</tr>
<tr>
<td>#1028</td>
<td>0x44</td>
<td>#1028</td>
<td>0x00</td>
</tr>
<tr>
<td>#1029</td>
<td>0x55</td>
<td>#1029</td>
<td>0x00</td>
</tr>
<tr>
<td>#1030</td>
<td>0x66</td>
<td>#1030</td>
<td>0x00</td>
</tr>
<tr>
<td>#1031</td>
<td>0x77</td>
<td>#1031</td>
<td>0x00</td>
</tr>
<tr>
<td>#1032</td>
<td>0x88</td>
<td>#1032</td>
<td>0x00</td>
</tr>
<tr>
<td>#1033</td>
<td>0x99</td>
<td>#1033</td>
<td>0x00</td>
</tr>
</tbody>
</table>
### Answers: movX to Main Memory 2/2

<table>
<thead>
<tr>
<th>REG</th>
<th>movl %eax, (%rsi) #1 4 bytes rax -&gt; #1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>movq %rax, (%rsi) #2 8 bytes rax -&gt; #1024</td>
</tr>
<tr>
<td>rcx</td>
<td>movb %cl, (%rsi) #3 1 byte rcx -&gt; #1024</td>
</tr>
<tr>
<td>rsi</td>
<td>movw %cx, 2(%rsi) #4 2 bytes rcx -&gt; #1026</td>
</tr>
<tr>
<td>-----</td>
<td>------------------------------------------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOV</th>
<th>MEM</th>
<th>MOV</th>
<th>MEM</th>
<th>MOV</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1024</td>
<td>0xEE</td>
<td>#1024</td>
<td>0xEE</td>
<td>#1024</td>
<td>0xEE</td>
</tr>
<tr>
<td>#1025</td>
<td>0xBB</td>
<td>#1025</td>
<td>0xBB</td>
<td>#1025</td>
<td>0xBB</td>
</tr>
<tr>
<td>#1026</td>
<td>0xCC</td>
<td>#1026</td>
<td>0xEE</td>
<td>#1026</td>
<td>0xEE</td>
</tr>
<tr>
<td>#1027</td>
<td>0xDD</td>
<td>#1027</td>
<td>0xFF</td>
<td>#1027</td>
<td>0xFF</td>
</tr>
<tr>
<td>#1028</td>
<td>0x00</td>
<td>#1028</td>
<td>0x00</td>
<td>#1028</td>
<td>0x00</td>
</tr>
<tr>
<td>#1029</td>
<td>0x00</td>
<td>#1029</td>
<td>0x00</td>
<td>#1029</td>
<td>0x00</td>
</tr>
<tr>
<td>#1030</td>
<td>0x00</td>
<td>#1030</td>
<td>0x00</td>
<td>#1030</td>
<td>0x00</td>
</tr>
<tr>
<td>#1031</td>
<td>0x00</td>
<td>#1031</td>
<td>0x00</td>
<td>#1031</td>
<td>0x00</td>
</tr>
<tr>
<td>#1032</td>
<td>0x88</td>
<td>#1032</td>
<td>0x88</td>
<td>#1032</td>
<td>0x88</td>
</tr>
<tr>
<td>#1033</td>
<td>0x99</td>
<td>#1033</td>
<td>0x99</td>
<td>#1033</td>
<td>0x99</td>
</tr>
</tbody>
</table>

#3

```
movb %cl, (%rsi)  #3 1 byte rcx -> #1024
```

#4

```
movw %cx, 2(%rsi)  #4 2 bytes rcx -> #1026
```

#5

```
movl %ecx, 4(%rsi)  #5 4 bytes rcx -> #1028
```
addX : A Quintessential ALU Instruction

addX B, A # A = A+B

OPERANDS
addX <reg>, <reg>
addX <mem>, <reg>
addX <reg>, <mem>
addX <con>, <reg>
addX <con>, <mem>

No mem+mem or con+con

EXAMPLES
addq %rdx, %rcx # rcx = rcx + rdx
addl %eax, %ebx # ebx = ebx + eax
addq $42, %rdx # rdx = rdx + 42
addl (%rsi),%edi # edi = edi + *rsi
addw %ax, (%rbx) # *rbx = *rbx + ax
addq $55, (%rbx) # *rbx = *rbx + 55

addl (%rsi,%rax,4),%edi # edi = edi+rsi[rax] (int)
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
## POS C

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

INITIAL

|-------+-------|
| REGS  |       |
| %rax  |  15   |
| %rbx  |  20   |
| %rcx  |  25   |
| %rdx  | #1024 |
| %r8   | #2048 |
| %r9   |   0   |

|-------+-------|
| MEM   |       |
| ...   |   ... |
| ...   |   ... |
| #1024 |  100  |
| #2048 |  200  |
| #2052 |  300  |
| #2056 |  400  |

31
## Answers: Addition

<table>
<thead>
<tr>
<th>INITIAL</th>
<th>POS A</th>
<th>POS B</th>
<th>POS C</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG</td>
<td>REG</td>
<td>REG</td>
<td>REG</td>
</tr>
<tr>
<td>%rax</td>
<td>15</td>
<td>%rax</td>
<td>35</td>
</tr>
<tr>
<td>%rbx</td>
<td>20</td>
<td>%rbx</td>
<td>20</td>
</tr>
<tr>
<td>%rcx</td>
<td>25</td>
<td>%rcx</td>
<td>26</td>
</tr>
<tr>
<td>%rdx</td>
<td>#1024</td>
<td>%rdx</td>
<td>#1024</td>
</tr>
<tr>
<td>%r8</td>
<td>#2048</td>
<td>%r8</td>
<td>#2048</td>
</tr>
<tr>
<td>%r9</td>
<td>0</td>
<td>%r9</td>
<td>0</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
</tr>
<tr>
<td>#1024</td>
<td>100</td>
<td>#1024</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>#2048</td>
<td>200</td>
<td>#2048</td>
<td>200</td>
</tr>
<tr>
<td>#2052</td>
<td>300</td>
<td>#2052</td>
<td>300</td>
</tr>
<tr>
<td>#2056</td>
<td>400</td>
<td>#2056</td>
<td>400</td>
</tr>
</tbody>
</table>

```
addq $1,%rcx
addq %rbx,%rax
addq (%rdx),%rcx
addl $1,(%r8,%r9,4)
addl $1,%r9d
addq $3,%rdx
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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Name</th>
<th>Effect</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>addX B, A</td>
<td>Add</td>
<td>A = A + B</td>
<td>Two Operand Instructions</td>
</tr>
<tr>
<td>subX B, A</td>
<td>Subtract</td>
<td>A = A - B</td>
<td></td>
</tr>
<tr>
<td>imulX B, A</td>
<td>Multiply</td>
<td>A = A * B</td>
<td>Has a limited 3-arg variant</td>
</tr>
<tr>
<td>andX B, A</td>
<td>And</td>
<td>A = A &amp; B</td>
<td></td>
</tr>
<tr>
<td>orX B, A</td>
<td>Or</td>
<td>A = A</td>
<td>B</td>
</tr>
<tr>
<td>xorX B, A</td>
<td>Xor</td>
<td>A = A ^ B</td>
<td></td>
</tr>
<tr>
<td>salX B, A</td>
<td>Shift Left</td>
<td>A = A &lt;&lt; B</td>
<td></td>
</tr>
<tr>
<td>shlX B, A</td>
<td>Shift Left</td>
<td>A = A &lt;&lt; B</td>
<td></td>
</tr>
<tr>
<td>sarX B, A</td>
<td>Shift Right</td>
<td>A = A &gt;&gt; B</td>
<td>Arithmetic: Sign carry</td>
</tr>
<tr>
<td>shrX B, A</td>
<td>Shift Right</td>
<td>A = A &gt;&gt; B</td>
<td>Logical: Zero carry</td>
</tr>
<tr>
<td>incX A</td>
<td>Increment</td>
<td>A = A + 1</td>
<td>One Operand Instructions</td>
</tr>
<tr>
<td>decX A</td>
<td>Decrement</td>
<td>A = A - 1</td>
<td></td>
</tr>
<tr>
<td>negX A</td>
<td>Negate</td>
<td>A = -A</td>
<td></td>
</tr>
<tr>
<td>notX A</td>
<td>Complement</td>
<td>A = ~A</td>
<td></td>
</tr>
</tbody>
</table>
**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:

```
# 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
imul $3,%eax
addl $1,%eax
# eax = eax*3 + 1
# 3-4 cycles

#OPTIMIZED with leaX:
leal 1(%eax,%eax,2),%eax
addl $1,%eax
# eax = eax + 2*eax + 1,
# 1 cycle

# gcc, you are so clever...
```
Division: It’s a Pain (1/2)

- Unlike other ALU operations, 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)

```c
### division.s:
movl $15, %eax  # set eax to int 15
cqto             # extend sign of eax to edx
## combined 64-bit register %edx:%eax is
## now 0x00000000 0000000F = 15
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
```

Compiler avoids division whenever possible: compile col_unsigned.c and col_signed.c to see some tricks.
When performing division on 8-bit or 16-bit quantities, use instructions to sign extend small reg to all rax register

```assembly
### 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 FFFFFFEF
    # rdx = 0x00000000 00000000
cwtl # "convert word to long" sign extend ax to eax
    # rax = 0x00000000 FFFFFFFE
    # rdx = 0x00000000 00000000
cltq # "convert long to quad" sign extend eax to rax
    # rax = 0xFFFFFFFF FFFFFFFE
    # rdx = 0x00000000 00000000
cqto # sign extend rax to rdx
    # rax = 0xFFFFFFFF FFFFFFFE
    # 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)
```