CSCI 2021: Assembly Basics and x86-64

Chris Kauffman

Last Updated:
Mon Feb 25 17:36:07 CST 2019

Logistics

Reading Bryant/O'Hallaron

- Ch 3.1-7: Assembly, Arithmetic, Control
- Ch 3.8-11: Arrays, Structs, Floats

Goals

- Assembly Basics
- x86-64 Overview

Assignment 2: Questions?

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

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

- Binary languages are hard for humans to read
- An **assembler** is a software tool that translates text description of the machine code to binary and formats it for execution by a processor
- A **compiler** is a chain of tools that translate high level languages to lower ones, perform optimizations, link library code, etc.
- Assembly generation is a late stage in compilers like `gcc`: C code is transformed to an internal data structure, optimized, then assembly is produced
- 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

- Targets Intel/AMD compatible chips with 64-bit word size (addresses)
- Descended from Intel Architecture (IA32) assembly for 32-bit systems
- IA32 descended from earlier 16-bit systems
- There is a **LOT** of cruft in x86-64 for backwards compatibility: it is not the assembly language you would design from scratch today
- Will touch on evolution of as we move forward
- Warning: Much information is available on assembly programming for Intel chips on the web **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

```
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 operand size

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 performs lots of optimizations to code
- `gcc -Og file.c`: disable optimizations to help with debugging
- Usually do this to generate slightly more readable assembly

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

> cat mstore.s                      # show assembly output
        .file   "mstore.c"
        .text
        .globl  multstore           # function symbol for linking
        .type   multstore, @function
multstore:                          # beginning of mulstore function
.LFB0:
        .cfi_startproc              # assembler directives
        pushq   %rbx                # assembly instruction
        .cfi_def_cfa_offset 16      # directives
        .cfi_offset 3, -16
        movq    %rdx, %rbx          # assembly instructions
        call    mult2@PLT           # function call
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret                         # function return
        .cfi_endproc
```

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

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:

```
        .globl  multstore   # notify linker of location multstore
  multstore:                # beginning of multstore section
        blah blah blah
```

- *Variables*: registers and memory, maybe some named locations
- *Assignment*: instructions that put bits in registers/memory
- *Functions*: code locations that are **labeled** and global
- *Conditionals/Iteration*: assembly instructions that jump to code locations
- *Aggregate data*: none, use the stack/multiple registers
- *Library System*: link to other code

So what *are* these Registers?

- Memory locations that are directly manipulated by the CPU
- Usually *very* fast memory directly on the CPU (not RAM)
- Most instructions involve changes to registers

Example: Adding Together Integers

- Ensure registers have desired values in them
- Issue an `add` 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

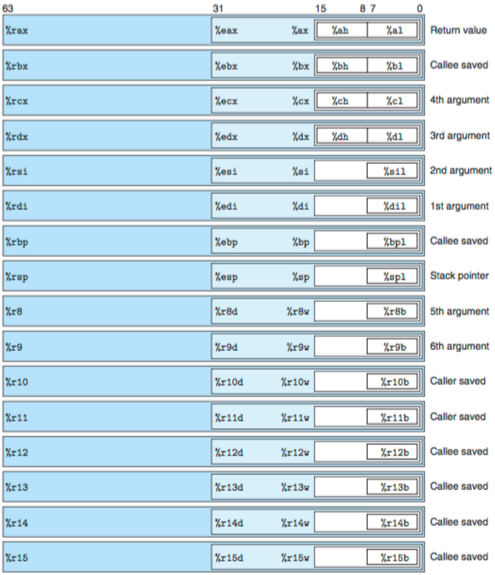
Register Naming Conventions

- AT&T syntax identifies registers with preceding %
- 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 64-bit regs `r8,r9,...,r14,r15` with 32-bit `r8d,r9d,...` and 16-bit `r8w,r8w...`
- Instructions must match 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

x86-64 "General Purpose" Registers



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

- `%rax | %eax | %ax` contains the return value from functions depending on type.
- `%rdi,%rsi,%rdx, %rcx,%r8, %r9` contain first six 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

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** to but bytes into files, in this case the screen, likely `write(1, buf, 5);`

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 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: ???
- OS executes **priveleged** code that can manipulate any part of memory, touch internal data structures corresponding to files, do other fun stuff discussed in CSCI 4061 / 5103

A Complete Example: `col_simple_asm.s`

- The following codes solve the problem:

  *Computes Collatz seq starting at 10. Return the number of steps to converge to 1 as the **return code** from `main()`*

- Examine source code, produce assembly with `gcc`
- Compile/Run, show in the debugger
- Illustrate tricks associated with `gdb` and assembly

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 vesion
	Generated assembly is ... interesting

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 4,810 pages of the x86-64 manual

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

Data Movement: `movX` instruction

```
movX SOURCE, DEST    # 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
- Moving data to
- 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, (%ecx)    # *ecx = 10; >:-(
```

Note variations

- `movq` for 64-bit
- `movl` for 32-bit
- `movw` for 16-bit
- `movb` for 8-bit

Operands and Addressing Modes

In many instructions, operands can have a variety of forms including constants and memory addresses

Style	Type	C-like	Notes
`$21`	immediate	`21`	value of constant like 21
`$0xD2`			or `0xD2 = 210`
`%rax`	register	`rax`	to/from register contents
`(%rax)`	indirect	`*rax`	reg holds memory address, deref
`8(%rax)`	displaced	`*(rax+2)`	base plus constant offset,
`-4(%rax)`		`*(rax-1)`	C examples presume `sizeof(..)=4`
`(%rax,%rbx)`	indexed	`*(rax+rbx)`	base plus offset in given reg actual value of `rbx` is used, NOT multiplied by `sizeof()`
`(%rax,%rbx,4)`	scaled index	`rax[rbx]`	like array access with `sizeof(..)=4`
`(%rax,%rbx,8)`		`rax[rbx]`	"" with `sizeof(..)=8`
`1024`	absolute	...	Absolute address #1024 Almost never used

Exercise: Show movX Instruction Execution

Code movX_exercise.s

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

movq %rax,%rbx
movl %ecx,%eax
## POS B

movq $45,(%rdx)
movq $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 x to reg y
movX $5,(%x)   # address in %x to 5
```

Answers Part 1/2: `movX` Instruction Execution

```
                    movl $16, %eax
                    movl $20, %ebx       movq %rax,%rbx
                    movq $24, %rbx       movl %ecx,%eax #DANGER!
INITIAL             ## POS A             ## POS B
|-------+-------|   |-------+-------|    |-------+-------|
| REG   | VALUE |   | REG   | VALUE |    | REG   | VALUE |
| %rax  |     0 |   | %rax  |    16 |    | %rax  | #1024 |
| %rbx  |     0 |   | %rbx  |    24 |    | %rbx  |    16 |
| %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 |    25 |    | #1032 |    25 |
| #1040 |    15 |   | #1040 |    15 |    | #1040 |    15 |
| #1048 |     5 |   | #1048 |     5 |    | #1048 |     5 |
|-------+-------|   |-------+-------|    |-------+-------|
```

`#!`: Moving `ecx` to `eax` may miss half the memory address, observe effects in gdb (next topic).

Answers Part 2/2: `movX` Instruction Execution

```
                                                 movq $65,(%rcx,%rbx)
 movq %rax,%rbx       movq $45,(%rdx)            movq $3,%rbx
 movl %ecx,%eax #!    movq $55,16(%rdx)          movq $75,(%rcx,%rbx,8)
 ## 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 |
|-------+-------|    |-------+-------|          |-------+-------|
```

gdb Assembly: Examining Memory

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

```
(gdb) tui enable             # TUI mode
(gdb) layout asm             # assembly mode
(gdb) layout reg             # show registers
(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 0x0x112233445566AABB
```

- Gives rise to two other families of movement instructions for moving little registers (`X`) to big (`Y`) registers:

```
## movzXY move zero extend
movzwq $0xAABB,%rax  # %rax is 0x0x000000000000AABB
## movsXY move sign extend
movswq $-1,%rax      # %rax is 0x0xFFFFFFFFFFFFFFFF
```

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
```

- Addition represents most 2-operand ALU instructions well
- 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`

```
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

 addq (%rsi,%rax,4),%rdi    # rdi = rdi+rsi[eax] (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
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 (%rdx),%rcx   addl $1,(%r8,%r9,4)
addq %rbx,%rax     addq %rbx,(%rdx)   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 patter as `addX`: two operands, second gets changed.
- Some one operand instructions as well.

Instruction	Name	Effect	Notes
`addX B, A`	Add	`A = A + B`	Two Operand Instructions
`subX B, A`	Subtract	`A = A - B`	
`imulX B, A`	Multiply	`A = A * B`	
`andX B, A`	And	`A = A & B`	
`orX B, A`	Or	`A = A | B`	
`xorX B, A`	Xor	`A = A ^ B`	
`salX B, A`	Shift Left	`A = A << B`	
`shlX B, A`		`A = A << B`	
`sarX B, A`	Shift Right	`A = A >> B`	Arithmetic: Sign carry
`shrX B, A`		`A = A >> B`	Logical: Zero carry
`incX A`	Increment	`A = A + 1`	One Operand Instructions
`decX A`	Decrement	`A = A - 1`	
`negX A`	Negate	`A = -A`	
`notX A`	Complement	`A = ~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 often uses `leaq` for multiplication as it is usually faster than `imul` but less readable

```
# Odd Collatz update n = 3*n+1
# with imulX          # 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
                      # gcc, you are so clever...
```

Division: It's a Pain

- 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 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                 # 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.