CMSC216: Assembly Basics and x86-64

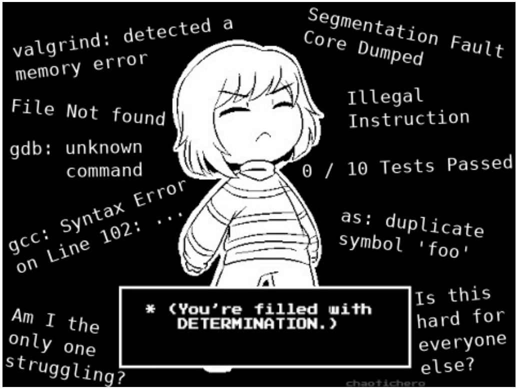
Chris Kauffman

Last Updated:
Mon Feb 23 06:02:21 PM EST 2026

Logistics

Announcements

Don't Give Up, Stay Determined!


valgrind: detected a memory error
Segmentation Fault Core Dumped
File Not found
Illegal Instruction
gdb: unknown command
0 / 10 Tests Passed
gcc: Syntax Error on Line 102: ...
as: duplicate symbol 'foo'
Am I the only one struggling?
* (You're filled with DETERMINATION.)
Is this hard for everyone else?
chaotichero

- If Project 1 / Exam 1 went awesome, count yourself lucky
- If things did not go well, Don't Give Up
- Spend some time contemplating why things didn't go well, talk to course staff about it, learn from mistakes
- There is a LOT of semester left and time to recover from a bad start

Wrapping Up Integer Affairs

Pre-exam one slides contain info on byte-ordering (little / big endian) and bit-wise operations to be discussed here prior to assembly.

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

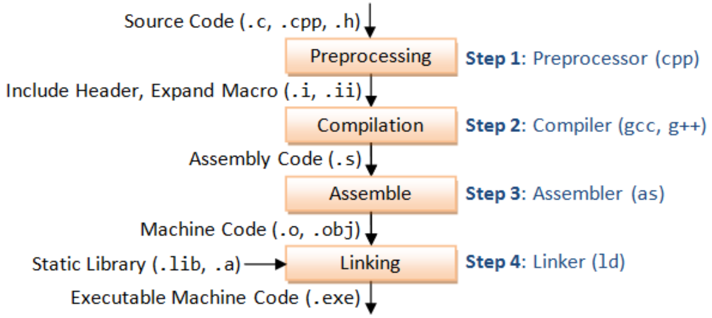
MOS Technology 6502

- 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


Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

- **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
- 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)

<MINTED>

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

Intel Syntax

<MINTED>

- 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

<MINTED>

gcc -Og -S mstore.c

<MINTED>

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 vesion
	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`

```
<MINTED>
```

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

  <MINTED>
- Assembler directives and labels are also possible:

  <MINTED>
- *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
  `<MINTED>`
- 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:

  <MINTED>
- 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 **priveleged** code that can manipulate any part of memory, touch internal data structures corresponding to files, do other fun stuff discussed in OS courses

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
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/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

<MINTED>

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`	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, often
`4(%rdx)`		`rdx->field`	used for strcut field derefs
`(%rax,%rbx)`	indexed	`*(rax+rbx)`	base plus offset in given reg
		`char_arr[rbx]`	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
			Rarely used

Exercise: Show movX Instruction Execution

Code movX_exercise.s

<MINTED>

Registers/Memory

<MINTED>

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

<MINTED>

`#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

<MINTED>

gdb Assembly: Examining Memory

gdb commands print and x allow one to print/examine memory memory of interest. Try on movX_exercises.s
<MINTED>
Many of these tricks are needed to debug assembly.

Register Size and Data Movement

- `%rax` is 64-bit register, `%eax` is its lower 32 bits
- 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

  <MINTED>
- Moving data to other small regs DOES NOT ALTER high bits

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

  <MINTED>

Exercise: movX differences in Main Memory

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

<MINTED>

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

<MINTED>

Answers: movX to Main Memory 1/2

```
<MINTED>
```

```
<MINTED>
```

Answers: movX to Main Memory 2/2

```
<MINTED>
```

```
<MINTED>
```

addX : A Quintessential ALU Instruction

<MINTED>

- 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

Optional Exercise: Addition

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

```
<MINTED>
```

```
<MINTED>
```

Answers: Addition

<MINTED>

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`	*Has a limited 3-arg variant*
`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`	`B` is constant or `%cl` reg
`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

<MINTED>

<MINTED>

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

<MINTED>

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)

<MINTED>

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

<MINTED>