Introduction to x86 asm

7

CS/COE 0449
Introduction to Systems Software

wilkie
(with content borrowed from Vinicius Petrucci and Jarrett Billingsley)

Spring 2019/2020

Assembly Refresher

What is forgotten... is art.

What is "Assembly"

- **Assembly: Human-readable representation of machine code.**
- **Machine code: what a computer *actually* runs.**
- **The "atoms" that make up a program.**
  - CPUs are actually fairly simple in concept.
  - (Yet we have an *entire semester* to fill, hmm)
- **Each CPU chooses its own machine code (and therefore its own style of assembly language)**
- **We used MIPS in CS 447.**
  - A RISC processor.
- **We will compare that to x86 today!**
  - A CISC processor. 😰

What is “Assembly”

- **Involves very simple commands.**

```
mov a0, t0
```

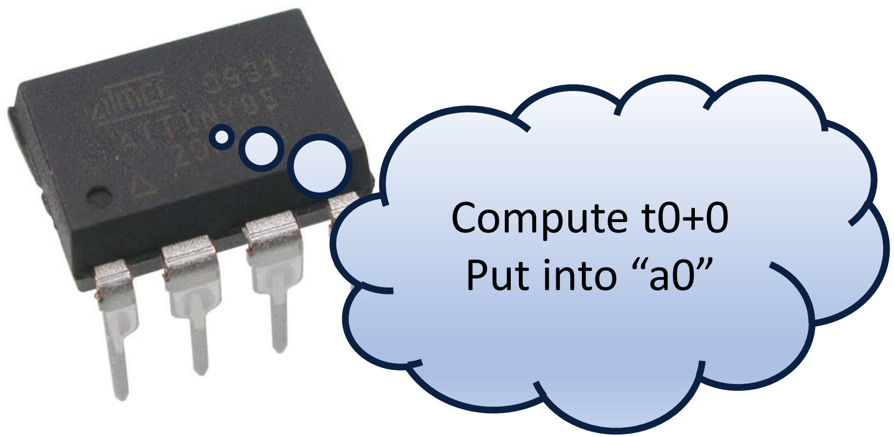
- **This command copies data from one place to another.**
  - Despite being called “move”, ugh!

```
-> add a0, t0, zero
```

- **Surprise! It’s actually shorthand for a different set of instructions.**
  - The processor can be made simpler.

```
-> 00000001000000000010000000100000
```

- **This command gets transformed into a numerical representation.**



- **The processor then interprets the binary representation.**
  - That’s essentially all a computer does!
  - CS 447 looks at this in much greater detail.

Assembly vs. Machine Language

- **Machine language instructions** are the patterns of bits that a processor reads to know what to do
- **Assembly language** (or "asm") is a human-readable (mostly), textual representation of machine language.

MIPS asm	MIPS machine language
`lw t0, 1200(t1)`	`100011 01001 01000 0000010010110000` `lw t1 t0 1200`
`add t2, s2, t0`	`000000 10010 01000 01010 00000 100000` `<math> s2 t0 t2 n/a add`
`sw t2, 1200(t1)`	`101011 01001 01010 0000010010110000` `sw t1 t2 1200`

Is Assembly Useful?

- Short answer: *YES*

- Assembly is “fast”, so we should use it for everything!
  --- NO!!! ---
- No type-checking, no control structures, very few abstractions.
  --- Fairly impractical for large things ---
- Tied to a particular CPU.
  - So, large programs have to be rewritten (usually) to work on new things.
- Yet: good for specialized stuff.
  - Critical paths and “boot” code in Kernels / Operating Systems
  - HPC (simulators, supercomputer stuff)
  - Real-time programs (video games; tho increasingly less / abstracted away)
  - And…

Practical Applications of Assembly: Modification

- Modifying programs after-the-fact. (Or reverse-engineering them)
- Legal "gray-area," / "confusing-mess" but generally modification/reverse engineering is allowed. Kinda? (Section 1201, US Code 17 § 108, etc)
  - Removing copy protection in order to preserve/backup.
  - **Librarians** and **preservationists** and "**pirates**" alike may all use/view/write assembly for this!
- I patched (the freely distributed) Lost Vikings so it would avoid copy protection and use a different sound configuration (so I could run it in a browser emulator)

x86 (NASM / Intel Syntax, MS-DOS)

```
; patching some bytes
; assembled with: `nasm -fbin -o patch.com patch.asm`
    org   0x100             ; .com files always start 256 by

    mov   ax, 0x00

    mov   dx, msg           ; the address of or message in d
    mov   ah, 9             ; ah=9 - "print string" sub-func
    int   0x21              ; call dos services

    mov   dx, fname         ; open file to patch
          …
```

Practical Applications of Assembly: Debugging

- Programs written in C, etc are generally translated into assembly.
  - And then into machine code.
- Or you can look at the machine code of programs and get an assembly code listing.
  - And step through the program one instruction at a time.
- When programs crash (sometimes programs you don’t have the code for) you can look at the assembly code and assess.
- Programs exist to help you (gdb, IDA Pro, radare, etc)
- We will apply this knowledge (using gdb) in a future assignment!

Basics of x86 Assembly

x86 really puts the… you know what… in Assembly

Instruction Set Architecture (ISA)

- An **ISA** is the **interface that a CPU presents to the programmer**.
  - When we say "architecture," *this* is what we mean.

- The ISA defines:
  - What the CPU **can do** (add, subtract, call functions, etc.)
  - What **registers** it has (we'll get to those)
  - The **machine language**
    - That is, the bit patterns used to encode instructions.

- The ISA **does not** define:
  - How to design the hardware!
    - ...if there's any hardware at all (think of Java, etc: virtual/hypothetical ISAs)

Types of ISAs: RISC

- **RISC**: "**R**educed **I**nstruction **S**et **C**omputer"
- ISA designed to make it easy to:
  - **build the CPU hardware**
  - make that hardware **run fast**
  - **write *compilers*** that make machine code
- A **small number** of instructions.
- Instructions are **very simple**

- MIPS (and RISC-V) is *very* RISCy

Types of ISAs: CISC

- **CISC**: "**C**omplex **I**nstruction **S**et **C**omputer"
- ISA designed for humans to write asm.
  - From the days *before compilers!*
- **Lots** of instructions and ways to use them
- **Complex** (multi-step) instructions to shorten and simplify programs.
  - "search a string for a character"
  - "copy memory blocks"
  - "check the bounds of an array access“
- Without these, you’d just write your programs to use the simpler instructions to build the complex behavior itself.
- x86 is *very* CISCy

Types of ISAs: Overview

- **CISC**: Complex Instruction Set Computer (does a whole lot)
- **RISC**: Reduced Instruction Set Computer (does enough)
- **Both**: **Equivalent!!** (RISC programs might be longer)

"Hackers" (1995) – Of course, they are talking about a Pentium x86 chip…
which thanks to its backwards compatibility, is *CISC*. Oh well!

Then again… x86 is so complex, modern designs translate the CISC instructions into RISC microcode on the fly… so it's RISC?? It can get complicated.

x86

- Descended from 16-bit 8086 CPU from ***1978***.
- Extended to 32 bits, then 64.
- Each version can **run most programs** from the previous version.
  - You can (mostly) run programs written in ‘78 on your brand new x86 CPU!

- This ISA is complex!
  - 30 years of backwards-compatibility… yikes.
  - We won’t exhaustively go over it.
  - There are, however, many very common idioms and instructions.
    - We will focus on these.
    - And we will focus on READING x86, not writing it.

x86 Registers (general)

- **Like MIPS, there are a set of general-purpose registers.**
  - There are 16; 64-bits in size and hold integer values in binary form.
- **Unlike MIPS, you can refer to parts of each register.**
  - Called **partial registers**.

%rax	%eax	%ax	%r8	%r8d	%r8w
%rbx	%ebx	%bx	%r9	%r9d	%r9w
%rcx	%ecx	%cx	%r10	%r10d	%r10w
%rdx	%edx	%dx	%r11	%r11d	%r11w
%rsi	%esi	%si	%r12	%r12d	%r12w
%rdi	%edi	%di	%r13	%r13d	%r13w
%rsp	%esp	%sp	%r14	%r14d	%r14w
%rbp	%ebp	%bp	%r15	%r15d	%r15w

Denoted by %

Stack Pointer

16-bit register names

32-bit register names

x86 Registers (specialized)

- There are also registers that you cannot directly interact with.
- Like MIPS, x86 has a **program counter** (`%rip`)
  - Also like MIPS, it cannot be read directly.

- There is also a `FLAGS` status register, which has information about the CPU state after an instruction is completed.
  - Stuff like a carry flag (CF) that denotes if an addition has a final carry.
  - Overflow detection (OF) denoting if an operation overflowed.

- And some extra registers for vector math, floating point math, and for OS usage we won't go over.

x86 Instruction Types

- In MIPS, you had R-type, I-type and J-type instructions.
- In x86 (CISC) you generally can have any instruction refer to data anywhere it is:
  - Registers, Immediates, Memory addresses, etc
  - Cannot refer to memory twice! (not possible: `mov (ptr), (ptr2)`)

x86-64 (gas / AT&T syntax)

```
mov %rbx, %rax        # rax = rbx
```

Immediates (prefixed by $)

```
mov $0x100, %rax      # rax = 0x100
```

Memory load (within parens)

```
mov (ptr), rax        # rax = *ptr
```

Memory store

```
mov %rax, (ptr)       # *ptr = rax

lea (ptr), %rax
mov 4(%rax), %rax     # *(ptr + 4) = rax
```

Displacement (can be -4, etc)

MIPS

```
add  t0, zero, t1          # t0 = t1

addi t1, zero, 0x100  # t1 = 0x100

la t0, ptr            # t0 = ptr
lw t1, 0(t0)          # t1 = *ptr

la t0, ptr            # t0 = ptr
sw t1, 0(t0)          # *ptr = t1

la t0, ptr            # t0 = ptr
sw t1, 4(t0)          # *(ptr + 4) = t1
```

Complex Addressing

- In MIPS, you would carefully craft the set of instructions necessary to interface with an array. (RISC)
- In x86, you can do a lot with just a single instruction. (CISC)
  - `(Rb, Ri, S)`: Base + (Index * Scalar) where Scalar must be 1, 2, 4 or 8
    - The fields are all optional; i.e., `(,Ri, S)` does just Index * Scalar

x86-64 (gas / AT&T syntax)

```
.data
arr: .int 1, -2, 6, -4, 11

.text

.global _start

_start:
  lea     (arr), %rbx      # rbx = addr to arr
  mov     $2, %rdi         # rdi = 2
  mov     (%rbx, %rdi, 4), %rdi # rdi = arr[2]

  lea     (%rbx, %rdi, 4), %rdi # rdi = &arr[2]
```

"Load Effective Address"

LEA simply computes address (no memory access)

MIPS

```
.data
arr: .word 1, -2, 6, -4, 11

.text
.globl main

main:
  la  t0, arr      # t0 = address to arr
  li  t1, 2        # t1 = 2
  mul t1, t1, 4    # t1 = t1 * 4
  add t0, t0, t1   # t0 = t0 + t1
  lw  s0, 0(t0)    # s0 = arr[2]
                   # ( t0 = &arr[2] )
```

Complex Addressing: CISC Strikes Again!!

- **When we say you can do a lot with just a single instruction, we mean it!**
  - `(Rb, Ri, S)`: Base + (Index * Scalar) where Scalar must be 1, 2, 4 or 8
  - What does the following do?
    - `%rbx = %rdi + %rdi * 8`

x86-64 (gas / AT&T syntax)

```
mov     $7, %rdi             # rdi = 7
```

"Load Effective Address" ???

```
lea     (%rdi, %rdi, 8), %rbx      # rbx = ???
```

LEA simply computes address… it's just very specific math.

x86 Instruction Qualifiers

- In MIPS, you sometimes had instructions varying on bitsize.
- In x86 (CISC) you can operate on any part of a register.
  - 64-bits, 32-bits, 16-bits… even 8-bit sections sometimes.
- The assembler can assume usually, but explicit names also work:

x86-64 (gas / AT&T syntax)

The assembler "figures it out"

```
mov  (ptr), %rax      # rax = *ptr
```

"quad word" which is 64-bits.

```
movq $0xfe, (ptr)    # *ptr = 0x100
```

"long word" which is 32-bits. ☹

```
movl $0xfe, (ptr)    # *ptr = 0x100
```

Ugh. In x86 a "word" here is 16-bits

```
movw $0xfe, (ptr)    # *ptr = 0x100
```

MIPS64

```
la t0, ptr      # t0 = ptr
lq t1, 0(t0)    # t1 = *ptr

la t0, ptr
li t1, 0x100
sq t1, 0(t0)    # *(long int*)ptr = 0x100

la t0, ptr
li t1, 0x100
sw t1, 0(t0)    # *(int*)ptr = 0x100

la t0, ptr
li t1, 0x100
sh t1, 0(t0)    # *(short*)ptr = 0x100
```

Hello World! (x86 vs. MIPS)

x86-64 (gas / AT&T syntax)

```
# Assumes Linux system calls

.data
db: .asciz  "Hello, world!\n"

.text

.global _start

_start:
  # write(1, db, 14)
  mov     $1, %rax     # system call 1 is write
  mov     $1, %rdi     # file handle 1 is stdout
  lea     (db), %rsi   # address of string
  mov     $14, %rdx    # number of bytes
  syscall              # invoke OS to print

  # exit(0)
  mov     $60, %rax    # system call 60 is exit
  xor     %rdi, %rdi   # we want return code 0
  syscall              # invoke OS to exit
```

MIPS (MARS)

```
# Run with MARS 4.5

.data

Hello:  .asciiz "Hello, world!\n"

.text

.globl main

main:
  li v0, 4         # print syscall
  la a0, Hello     # a0 = address
  syscall

  li v0, 17        # exit syscall
  move a0, zero    # a0 = 0
  syscall
```

Doing some x86 maths

- x86 and MIPS have, essentially, the same mathematical instructions.

x86-64 (gas / AT&T syntax)

```
add  $5, %rax    # rax += 5
add  %rbx, %rax  # rax += rbx
sub  $5, %rax    # rax -= 5
sub  %rbx, %rax  # rax -= rbx
sar  $5, %rax    # rax >>= 5
sar  %rbx, %rax  # rax >>= rbx
shr  $5, %rax    # rax >>= 5
shr  %rbx, %rax  # rax >>= rbx
shl  $5, %rax    # rax <<= 5
shl  %rbx, %rax  # rax <<= rbx
xor  $5, %rax    # rax ^= 5
xor  %rbx, %rax  # rax ^= rbx
```

MIPS

```
addi t0, t0, 5   # t0 += 5
add  t0, t0, t1  # t0 += t1
subi t0, t0, 5   # t0 -= 5
sub  t0, t0, t1  # t0 -= t1
sra  t0, t0, 5   # t0 >>= 5
sra  t0, t0, t1  # t0 >>= t1
srl  t0, t0, 5   # t0 >>= 5
srl  t0, t0, t1  # t0 >>= t1
sll  t0, t0, 5   # t0 <<= 5
sll  t0, t0, t1  # t0 <<= t1
xori t0, t0, 5   # t0 ^= 5
xor  t0, t0, t1  # t0 ^= t1
```

However, x86 lets you slice and dice

- **Each math instruction in x86 has variants based on the bitsize.**
  - `addq` (64-bit), `addl` (32-bit), `addw` (16-bit), `addb` (8-bit) <u>(rest of field zero extended!!)</u>

x86-64 (gas / AT&T syntax)

```
addq  $5, %rax    # rax += 5
addq  %rbx, %rax # rax += rbx

subl  $5, %eax    # eax -= 5
subl  %ebx, %eax # eax -= ebx
```

Arithmetic shift (sign extends)

```
sarw  $5, %ax     # ax >>= 5
sarw  %bx, %ax    # ax >>= bx
```

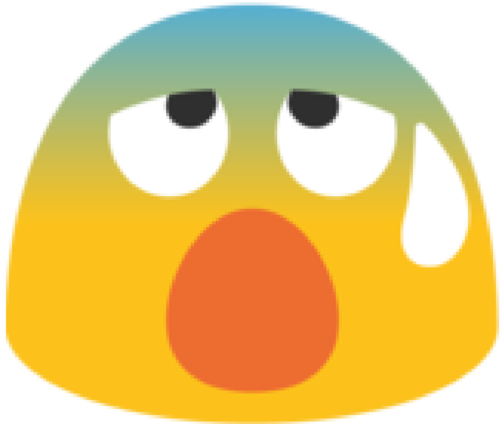
Logical shift (zero extends)

```
shrb  $5, %al     # al >>= 5
shrb  %bl, %al    # al >>= bl
```

8-bit register aliases are not commonly used

MIPS

```
# Only operates on words!!
```



Assembly Interlude

Here, we take a break, and look at some existing code.

Why write assembly? When you can write C

- You can take any of your C programs and emit the assembly.
- The compiler can do this for you:

```
gcc -S my_code.c
```

- This will create a file called `my_code.s` which looks… messy.
  - It has a ton of messy specific stuff wedged in there.
  - But you can generally pull apart some meaning from it.

Looking at C compilers...

- The messy output of the gcc compilation to assembly:

x86-64 (gas / AT&T syntax, `gcc -Og -S`)

```
.globl  abs
.type abs, @function
abs:
.LFB0:
  .cfi_startproc
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq  %rsp, %rbp
  .cfi_def_cfa_register 6
  movl  %edi, -4(%rbp)
  cmpl  $0, -4(%rbp)
  jns .L2
  negl  -4(%rbp)
.L2:
  movl  -4(%rbp), %eax
  popq  %rbp
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
```

main hasn't even shown up yet...

C

```c
/* Returns the absolute value of the
   given integer. */
int abs(int x) {
  if (x < 0) {
    x = -x;
  }

  return x;
}

int main() {
  printf("|%d| = %d\n", -5, abs(-5));
}
```

Disassembly – See how the sausage is made...

- So, that’s not very useful. And often we don’t have the code!
  - How do we go backward?
- You can take any compiled program and emit the assembly.
  - Many tools can help you do this (radare, objdump, gdb)
- Using a tool called objdump (only disassembles code section):

```
objdump -d my_program > my_program.asm
```

- This will create a file called `my_program.asm`.
  - You can glance at it and notice that it does not have names.
  - And labels are a bit, well, nonexistent.

And... here we are...

- An objdump disassembly is slightly lacking context.

x86-64 (gas / AT&T syntax, `objdump -d`)

```
0000000000001139 <abs>:
 1139: 55            push   %rbp
 113a: 48 89 e5      mov    %rsp,%rbp
 113d: 89 7d fc      mov    %edi,-0x4(%rbp)
 1140: 83 7d fc 00   cmpl   $0x0,-0x4(%rbp)
 1144: 79 03         jns    1149 <abs+0x10>
 1146: f7 5d fc      negl   -0x4(%rbp)
 1149: 8b 45 fc      mov    -0x4(%rbp),%eax
 114c: 5d            pop    %rbp
 114d: c3            retq
```

Machine code (in bytes)

Instruction address

C

```c
/* Returns the absolute value of the
   given integer. */
int abs(int x) {
  if (x < 0) {
    x = -x;
  }

  return x;
}

int main() {
  printf("|%d| = %d\n", -5, abs(-5));
}
```

Looking deeper

- Now we are starting to read the code… It does what we tell it to do!

x86-64 (gas / AT&T syntax, `objdump -d`)

```
0000000000001139 <abs>:
 1139: 55               push   %rbp
 113a: 48 89 e5         mov    %rsp,%rbp
 113d: 89 7d fc         mov    %edi,-0x4(%rbp)
 1140: 83 7d fc 00      cmpl   $0x0,-0x4(%rbp)
 1144: 79 03            jns    1149 <abs+0x10>
 1146: f7 5d fc         negl   -0x4(%rbp)
 1149: 8b 45 fc         mov    -0x4(%rbp),%eax
 114c: 5d               pop    %rbp
 114d: c3               retq
```

- ← **Preserves `%rbp` (caller activation frame)**
- **Allocates “`x`” on stack (`-4` from top)**
- ← **Move argument to `x`**
- ← **Compares `x` to `0` and sets `FLAGS`**
- ← **Jumps if `FLAGS[SF]` is 0 (`x` is positive)**
- ← `x = -x`
- ← **Sets `%eax` to `x`**
- ← **Resets caller activation frame**
- ← **Returns (return value is in `%rax`)**

`0x1139 + 0x10`
`= 0x1149`

Instructions have varying size

So, the next instruction address is irregular. Compare with MIPS / RISC-V.

Brought to you by the letters: C ABI

- The **C Application Binary Interface** (ABI) are assembly conventions
  - Like MIPS, certain registers are typically used for returns values, args, etc
  - It is not defined by the language, but rather the OS.
    - Windows and Linux (UNIX/System V) have a different C ABI ☹

- **In our x86-64 Linux C ABI, registers are used to pass arguments:**
  - `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` (First, second, etc) (Like MIPS `a0` – `a3`)
  - Remaining arguments go on the stack.
  - Callee must preserve `%rbp`, `%rbx`, `%r12`, `%r13`, `%r14`, `%r15` (Like MIPS `s0` – `s7`)
  - Return value: `%rax` (overflows into `%rdx` for 128-bits) (MIPS `v0` – `v1`)
  - Lots of other small things not worth going over.

- **For reference:** https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf