Recitation 06

Assembly

Preface

Preparing for Recitation 06

Pulling the sixth assignment

- Inside your recitations repository in your VM
  - Run **git pull upstream master**
  - Make sure you have an r06 directory
  - You should have an in-class directory and a for-homework directory inside r06

Today's agenda

- We will discuss in recitation
  - Assembly programming
- What you will do tonight
  - R06
    - Write some assembly code
      - You are given a C function to translate
    - Write some C code
      - You are given some assembly code to translate back to C

Assembly

C is for people

Why Assembly

- In the real world, computers don’t “understand” code
- They only “understand” a set of instructions
- To run code
  1. The CPU fetches an instruction from the memory at the PC
  2. The CPU decodes that instruction
  3. If needed, the CPU fetches data from memory
  4. The CPU performs computations
  5. If needed, the CPU writes data to memory
  6. The CPU increments the PC to the next instruction

Why Assembly

- Computers don’t “understand” assembly either, but assembly maps much more closely to machine instructions than C code
- Assembly code involves instruction nmeonics
  - That was a hard word to mess up spelling. I meant mnemonics.
  - For x86_64, These are things like addq, movq, jne

Registers

- Accessing memory is very, very slow compared to the rest of what a CPU can do
- Registers are fast temporary storage
- Originally there were 8, all 16-bits large
  - %ax, %bx, %cx, %dx, %si, %di, %bp, %sp
  - These have 32-bit counterparts – add an e, eg %eax, %esp
  - These also have 64-bit counterparts – add an r, eg %rax, %rsp
- With 64 bits came 8 more registers, %r8 to %r15
  - These have 32-bit counterparts - add a D to the start, eg %r8d
  - These have 16-bit counterparts – add a W to the start, eg %r8w
- All registers also allow you to access their lowest 8 bits
- %ax, %bx, %cx, and %dx, allow you to access their upper 8 bits

Important Instructions

Instruction	What it does
mov src, dest	dest = src
add src, dest	dest = dest + src
sub src, dest	dest = dest - src
imul src, dest	dest = dest * src
inc dest	dest = dest + 1

Instruction operands

- src and dest can be one of three things
  1. An **immediate**
     1. A constant value, prefaced with $
     2. Eg. $0, or $0xdeadbeef
     3. dest cannot be an immediate – why?
  2. A **register**
     1. One of the general purpose registers
     2. Eg. %eax
  3. A location in **memory**
     1. Consider registers as pointers, and get the value at an address after some simple calculations
     2. You cannot perform a mov from memory into memory
     3. How big is what you are getting from memory, in bytes?

Instruction Suffixes

Suffix	Name	Size (bytes)
b	byte	1
w	word	2
l	long	4
q	quadword	8

Memory Addressing Modes

- Direct
  - Given a register, use the value located at the memory address contained in the register
  - Register name in parens
  - Eg mov (%rax), %rbx
- With displacement
  - Use the value in memory located at the register value plus a constant displacement
  - Have the constant appear before the parens
  - Eg mov 10(%rax), %rbx

Memory Addressing Modes

- Complete
  - We have a constant displacement, a starting point, an offset, and a constant to scale the offset by…
  - D(Rb, Ri, S)
    - The address at Rb + Ri*S + D, where S and D are constant and Rb and Ri are registers
    - Why might this ever be useful?!?
  - Eg mov 10(%rax, %rbx, 4), %rcx
  - If the displacement is 0 or the scale is 1, you may leave them out

Lea src, dest

- Load Effective Address
- Take the address expression from src, and save it to dest
- Do not access memory, just compute the address from the offsets, index, base, and scale, and then save the computed address in dest
- Can also be used to quickly add registers and store the result in a third register

EFLAGS

- A special register that stores some status about the executed instructions
- Different bits tell us different things
- Instructions may set those bits depending on what has happened
  - These include arithmetic instructions like add or sub, as well as instructions like cmp

EFLAGS

Flag	Meaning
ZF	Result was 0
SF	The most significant bit of the result
CF	Set if the result borrowed from or carried out of the most significant bit
OF	Overflow for signed arithmetic

- The CPU doesn’t know if operands are signed or unsigned
- So, it calculates both the signed overflow (OF) and the unsigned overflow (CF) for each instruction
  - That is, OF is set assuming both are signed
  - CF is set assuming both are unsigned

cmp

- Same as sub, except it doesn’t store the result in dest
- It does, however, still change the EFLAGS I just mentioned
- This makes it useful for comparisons and conditions

jmp

- **jmp** label
  - Continues executing from the label, unconditionally
  - label is where to jump to
  - It acts like goto in C

Conditional Jumps

- **je** label
  - Jump if ZF is set
- **jne** label
  - Jump if ZF is not set
- **jg** label
  - Jump if ZF is not set and SF and OF are the same
- **jl** label
  - Jump if SF and OF are not the same
- **ja** label
  - Jump if CF and ZF are both not set

Calling conventions for x86_64

- The first six arguments are stored in this order:
  - %rdi, %rsi, %rdx, %rcx, %r8, %r9
- The return value is stored in %rax
- Functions may feel free to use the argument registers and the return value register, as well as %r10, and %r11
- If a function wants to use the other 7 registers, they must save them then restore them before returning