Machine and Assembly Code

x86-64 Introduction
Topics

- Architecture/assembly intro
- Operands
- Basic opcodes
Let's focus for now on the single-CPU components
von Neumann architecture

1. Fetch
2. Decode
3. Execute
(repeat)
Machine code

- **Machine code instruction**
  - Variable-length binary encoding of **opcodes** and **operands**
  - Program (instructions) stored in memory along with data
  - Specific to a particular CPU architecture (e.g., x86-64)
  - Looks very different than the original C code!

```c
int add (int num1, int num2)
{
    return num1 + num2;
}
```

0000000000400606 <add>:
0000000000400606 55
0000000000400607 48 89 e5
000000000040060a 89 7d fc
000000000040060d 89 75 f8
0000000000400610 8b 55 fc
0000000000400613 8b 45 f8
0000000000400616 01 d0
0000000000400618 5d
0000000000400619 c3
Machine code

- Instructions are specified by an instruction set architecture (ISA)
  - x86-64 (x64) is the current dominant workstation/server architecture
    - Enormous and complex; lots of legacy features and support for previous ISAs
    - We'll learn a bit of it now, then later focus on a simplified form called Y86
  - ARM is used in embedded and mobile markets
  - POWER is used in the high-performance market (supercomputers!)
  - RISC-V is used in CPU research (and is growing in the industrial market)

```
000000000000400606 <add>:
  400606:   55
  400607:   48 89 e5
  40060a:   89 7d fc
  40060d:   89 75 f8
  400610:   8b 55 fc
  400613:   8b 45 f8
  400616:   01 d0
  400618:   5d
  400619:   c3
```
Assembly code

- **Assembly code**: human-readable form of machine code
  - Each indented line of text represents a single machine code instruction
    - Two main x86-64 formats: Intel and AT&T (we'll use the latter)
    - Use "#" to denote comments (extends to end of line)
  - Generated from C code by compiler (not a simple process!)
  - Disassemblers like `objdump` can extract assembly from an executable
  - Understanding assembly helps you to debug, optimize, and secure your programs

```
0000000000400606 <add>:
  400606:   55  push  %rbp
  400607:   48 89 e5  mov  %rsp,%rbp
  40060a:   89 7d fc  mov  %edi,-0x4(%rbp)
  40060d:   89 75 f8  mov  %esi,-0x8(%rbp)
  400610:   8b 55 fc  mov  -0x4(%rbp),%edx
  400613:   8b 45 f8  mov  -0x8(%rbp),%eax
  400616:   01  d0  add  %edx,%eax
  400618:   5d  pop  %rbp
  400619:   c3  retq
```
Assembly code

- Assembly provides low-level access to machine
  - Program counter (PC) tracks current instruction
    - Like a bookmark; also referred to as the instruction pointer (IP)
  - Arithmetic logic unit (ALU) executes opcode of instructions
    - Today, we'll focus on some very basic opcodes
  - Register file & main memory store operands
    - Registers are faster but main memory is larger

```
0000000000400606 <add>:
  400606: 55  push  %rbp
  400607: 48 89 e5  mov  %rsp,%rbp
  400609: 89 7d fc  mov  %edi,-0x4(%rbp)
  40060a: 89 75 f8  mov  %esi,-0x8(%rbp)
  40060d: 8b 55 fc  mov  -0x4(%rbp),%edx
  400610: 8b 45 f8  mov  -0x8(%rbp),%eax
  400613: 01 d0  add  %edx,%eax
  400615: 5d  pop  %rbp
  400616: c3  retq
```
Operand types

- **Immediate**
  - Operand value embedded in instruction itself
  - Extends the size of the instruction by the width of the value
  - Written in assembly using “$” prefix (e.g., $42 or $0x1234)

- **Register**
  - Operand stored in register file
  - Accessed by register number
  - Written in assembly using name and “%” prefix (e.g., %eax or %rsp)

- **Memory**
  - Operand stored in main memory
  - Accessed by effective address calculated from instruction components
  - Written in assembly using a variety of addressing modes
Registers

- **General-purpose**
  - %rax, %rbx, %rcx, and %rdx
  - %rsi and %rdi
  - Legacy name meanings (e.g., “%rax” as the accumulator) are less important for us
    - But for now, note that %rax is also used to store the return value of a function

- **Special**
  - %rip: instruction pointer
    - This is the PC on x86-64
  - %flags: status info
    - "Condition codes" in CS:APP
  - %rbp: base pointer
  - %rsp: stack pointer

| %rax | (contents of %rax) |
| %rbx | (contents of %rbx) |
| %rcx | (contents of %rcx) |
| %rdx | (contents of %rdx) |
| %rsi | (contents of %rsi) |
| %rdi | (contents of %rdi) | ... |

| %rip | (contents of %rip) |
| %rflags | (contents of %rflags) | ... |

Register File
Memory addressing modes

- Absolute: $addr$
  - Effective address: $addr$

- Indirect: $(reg)$
  - Effective address: $R[reg]$

- Base + displacement: $offset(reg)$
  - Effective address: $offset + R[reg]$

- Indexed: $offset(reg_{base}, reg_{index})$
  - Effective address: $offset + R[reg_{base}] + R[reg_{index}]$

- Scaled indexed: $offset(reg_{base}, reg_{index}, s)$
  - Effective address: $offset + R[reg_{base}] + R[reg_{index}] \cdot s$
  - Scale ($s$) must be 1, 2, 4, or 8

$R[reg] = value$ of register $reg$

useful for pointers!

useful for arrays!

(Also, note that $offset$ and $reg_{base}$ are optional here)
Exercise

Given the following machine status, what is the value of the following assembly operands? (assume 32-bit memory locations)

- $42
- $0x10
- %rax
- 0x104
- (%rax)
- 4(%rax)
- 2(%rax, %rdx)
- (%rax, %rdx, 4)

<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
</tr>
<tr>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x108</td>
</tr>
</tbody>
</table>
Exercise

- Given the following machine status, what is the value of the following assembly operands? (assume 32-bit memory locations)

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Registers</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x100</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>0x2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
<td>0x13</td>
</tr>
</tbody>
</table>
In x86-64, assume the %rax register stores the address of the data you want to access. Which of the following operand specifiers could NOT be used to access the data?

- A) %rax
- B) (%rax)
- C) @(%rax)
- D) (,%rax,1)
- E) @(,%rax,1)
Basic x86-64 instructions

- **Data movement**: "mov"
  - Copies data from first operand to second operand
    - E.g., `mov $1, %rax` will set the value of %rax to 1

- **Arithmetic**: "add", "sub", "imul"
  - Performs operation, saving result in **second** operand
    - E.g., `add %rcx, %rax` will add the value of %rcx to the value of %rax
    - (Note lack of division)

- **Bitwise**: "and", "or", "xor"
  - Performs operation, saving result in **second** operand
    - E.g., `xor %rcx, %rax` will XOR the values of %rcx and %rax, saving the result in %rax
Basic x86-64 instructions

- **Control flow**: change the PC with `jmp` (%rip cannot be set directly)
  - Label (name followed by `:`) marks a location in code that can be “jumped to”
    - E.g., “foo:”
  - `jmp`: Jump to a given label
    - E.g., `jmp foo` will “jump to” label “foo”

- **Conditionals**: "cmp" followed immediately by "je" or "jne"
  - `cmp`: Compares operand values
  - `je`: If the values were **equal**, jump to a label
    - E.g., `cmp %rax, $0` followed by `je foo` will jump to label “foo” if the value of %rax was zero
  - `jne`: If the values were **not equal**, jump to a label
    - E.g., `cmp %rax, $0` followed by `jne foo` will jump to label “foo” if the value of %rax was NOT zero
What is the value of `%rax` after these instructions execute?

```assembly
mov $5, %rcx
and $0, %rax
cmp $0, %rcx
je skip
add %rcx, %rax
skip:
sub $1, %rax
```

- A) 0
- B) 1
- C) 4
- D) 5
- E) Cannot be determined
Hand-writing x86_64 assembly

• Minimal template (returns 0; known to work on stu):

```assembly
.globl main # makes “main” a global symbol
main: # execution will start here

    mov $0, %rax # your code goes here

    ret # “return from “main”
```

• Save in .s file and build with gcc as usual (don’t use “-c” flag)
  
  - Run program and view return value (final value of %rax) in bash with “echo $?”

• Use gdb to trace execution
  
  - `start`: begin execution and pause at main
  - `disas`: print disassembly of current function
  - `ni`: next instruction (step over function calls)
  - `si`: step instruction (step into function calls)
  - `p/x $rax`: print value of RAX (note “$” instead of “%”)
  - `info registers`: print values of all registers