Machine and Assembly Code

Data Movement and Arithmetic
Topics

- Architecture/assembly intro
- Data formats
- Data movement
- Arithmetic and logical operations
Let's focus for now on the single-CPU components
von Neumann architecture

1. Fetch
2. Decode
3. Execute

Main Memory
- Machine code
  - Variable-length binary encoding of **opcodes** and **operands**
  - Program is 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;
}
```

```
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
```
Machine code

- Machine instructions are specified by an instruction set architecture (ISA)
  - x86-64 (x64) is the current dominant workstation/server architecture
    - 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)
  - x86-64 has an enormous, complex instruction set
    - 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

000000000000400606 <add>:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>400606</td>
<td>55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400607</td>
<td>48</td>
<td>89</td>
<td>e5</td>
</tr>
<tr>
<td>40060a</td>
<td>89</td>
<td>7d</td>
<td>fc</td>
</tr>
<tr>
<td>40060d</td>
<td>89</td>
<td>75</td>
<td>f8</td>
</tr>
<tr>
<td>400610</td>
<td>8b</td>
<td>55</td>
<td>fc</td>
</tr>
<tr>
<td>400613</td>
<td>8b</td>
<td>45</td>
<td>f8</td>
</tr>
<tr>
<td>400616</td>
<td>01</td>
<td>d0</td>
<td></td>
</tr>
<tr>
<td>400618</td>
<td>5d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400619</td>
<td>c3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 ATT (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 %ebp
  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 data movement and arithmetic opcodes
  - Register file & main memory store operands
    - Registers are faster but main memory is larger

```
000000000000400606 <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
```
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**
  - AX: accumulator
  - BX: base
  - CX: counter
  - DX: address
  - SI: source index
  - DI: dest index

- **Special**
  - BP: base pointer
  - SP: stack pointer
  - FLAGS: status info
    - "Condition codes" in CS:APP
  - IP: instruction pointer
    - This is the PC on x86-64

\[
e_{xx} = \text{lower 32-bits (e.g., eax)} \\
r_{xx} = \text{full 64 bits (e.g., rax)}
\]
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

\( R[reg] = \text{value of register } reg \)

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

- Historical artifact: "word" in x86 is 16 bits
  - 1 byte (8 bits) = "byte" (b suffix)
  - 2 bytes (16 bits) = "word" (w suffix)
  - 4 bytes (32 bits) = "double word" (l suffix)
  - 8 bytes (64 bits) = "quad word" (q suffix)

- Often, a “class” of instructions will perform similar jobs, but on different sizes of data
  - There are no “types” in assembly code
  - Thus, instruction suffixes and operand sizes must match!
  - E.g., movq $1, %rax is valid but movq $1, %eax is not
Data movement

- Primary data movement instruction: "mov"
  - **Copies** data from first operand to second operand
    - E.g., `movq $1, %rax` will set the value of RAX to 1
    - `movb`, `movw`, `movl`, `movq`, `movabsq`
    - `movabsq` is the only form that takes a 64-bit immediate

- Zero-extension variant: "movz"
  - `movzbw`, `movzbl`, `movzwl`, `movzbq`, `movzwq`
  - Note lack of `movzlq`; just use `movl`, which sets higher 32-bits to zero

- Sign-extension variant: "movs"
  - `movsbw`, `movsbl`, `movswl`, `movsbq`, `movswq`, `movslq`
  - byte-to-word
x86-64 addresses

- Addresses in x86-64 are always 32 or 64 bits
  - Thus, the registers used to calculate the effective address of a memory operand must be 32 or 64 bits
    - E.g., `movw %ax, (%ebp)` is valid
    - E.g., `movw %ax, (%rbp)` is valid
    - E.g., `movw %ax, (%bp)` is not valid!
    - E.g., `movw %ax, %rbp` is not valid!
  - This does NOT mean that the instruction will load or store 32/64 bits from/to memory
    - The size of data moved is determined by the instruction suffix
    - Memory locations have no “type” in assembly/machine code
Validity summary

• Is an instruction valid?
  - Is the opcode valid?
  - Are all of the operands valid?
    • For immediate operands, is it a source register?
      - (cannot write to immediates!)
    • For register operands, is it a valid register?
      - (and does it match the width suffix?)
    • For memory operands, is it a valid addressing mode?
      - (and are all registers used 32- or 64-bits?)
Stack management

- The **system stack** holds 8-byte (quadword) slots, growing downward from high addresses to low addresses
  - **Stack Pointer** (SP) register stores address of "top" of stack
    - i.e., a pointer to the last value pushed (lowest address)
    - On x86-64, it is `%rsp` b/c addresses are 64 bits
  - **pushq <reg>** instruction
    - Subtract 8 from stack pointer
    - Store value of <reg> at (%rsp)
  - **popq <reg>** instruction
    - Retrieve value at (%rsp)
      - Save value in the given register
    - Increment stack pointer by 8
Exercise

• Given the following register state, what will the values of the registers be after the following instruction sequence?
  - pushq %rax
  - pushq %rcx
  - pushq %rbx
  - pushq %rdx
  - popq %rax
  - popq %rbx
  - popq %rcx
  - popq %rdx

<table>
<thead>
<tr>
<th>Registers</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%rax</td>
<td>0xAA</td>
</tr>
<tr>
<td></td>
<td>%rbx</td>
<td>0xBB</td>
</tr>
<tr>
<td></td>
<td>%rcx</td>
<td>0xCC</td>
</tr>
<tr>
<td></td>
<td>%rdx</td>
<td>0xDD</td>
</tr>
</tbody>
</table>
### Arithmetic operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaq</td>
<td>$D \leftarrow &amp;S$</td>
<td>Load effective address</td>
</tr>
<tr>
<td>INC</td>
<td>$D \leftarrow D+1$</td>
<td>Increment</td>
</tr>
<tr>
<td>DEC</td>
<td>$D \leftarrow D-1$</td>
<td>Decrement</td>
</tr>
<tr>
<td>NEG</td>
<td>$D \leftarrow -D$</td>
<td>Negate</td>
</tr>
<tr>
<td>NOT</td>
<td>$D \leftarrow \neg D$</td>
<td>Complement</td>
</tr>
<tr>
<td>ADD</td>
<td>$D \leftarrow D + S$</td>
<td>Add</td>
</tr>
<tr>
<td>SUB</td>
<td>$D \leftarrow D - S$</td>
<td>Subtract</td>
</tr>
<tr>
<td>IMUL</td>
<td>$D \leftarrow D \times S$</td>
<td>Multiply</td>
</tr>
<tr>
<td>XOR</td>
<td>$D \leftarrow D \oplus S$</td>
<td>Exclusive-or</td>
</tr>
<tr>
<td>OR</td>
<td>$D \leftarrow D</td>
<td>S$</td>
</tr>
<tr>
<td>AND</td>
<td>$D \leftarrow D &amp; S$</td>
<td>And</td>
</tr>
<tr>
<td>SAL</td>
<td>$D \leftarrow D \ll k$</td>
<td>Left shift</td>
</tr>
<tr>
<td>SHL</td>
<td>$D \leftarrow D \ll k$</td>
<td>Left shift (same as SAL)</td>
</tr>
<tr>
<td>SAR</td>
<td>$D \leftarrow D \gg_A k$</td>
<td>Arithmetic right shift</td>
</tr>
<tr>
<td>SHR</td>
<td>$D \leftarrow D \gg_L k$</td>
<td>Logical right shift</td>
</tr>
</tbody>
</table>

**Figure 3.10** Integer arithmetic operations. The load effective address (leaq) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation $\gg_A$ and $\gg_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.
Exercise

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaq S, D</td>
<td>D ← &amp;S</td>
<td>Load effective address</td>
</tr>
<tr>
<td>INC D</td>
<td>D ← D+1</td>
<td>Increment</td>
</tr>
<tr>
<td>DEC D</td>
<td>D ← D−1</td>
<td>Decrement</td>
</tr>
<tr>
<td>NEG D</td>
<td>D ← −D</td>
<td>Negate</td>
</tr>
<tr>
<td>NOT D</td>
<td>D ← ~D</td>
<td>Complement</td>
</tr>
<tr>
<td>ADD S, D</td>
<td>D ← D + S</td>
<td>Add</td>
</tr>
<tr>
<td>SUB S, D</td>
<td>D ← D − S</td>
<td>Subtract</td>
</tr>
<tr>
<td>IMUL S, D</td>
<td>D ← D * S</td>
<td>Multiply</td>
</tr>
<tr>
<td>XOR S, D</td>
<td>D ← D ^ S</td>
<td>Exclusive-or</td>
</tr>
<tr>
<td>OR S, D</td>
<td>D ← D</td>
<td>Or</td>
</tr>
<tr>
<td>AND S, D</td>
<td>D ← D &amp; S</td>
<td>And</td>
</tr>
<tr>
<td>SAL k, D</td>
<td>D ← D &lt;&lt; k</td>
<td>Left shift</td>
</tr>
<tr>
<td>SHL k, D</td>
<td>D ← D &lt;&lt; k</td>
<td>Left shift (same as SAL)</td>
</tr>
<tr>
<td>SAR k, D</td>
<td>D ← D &gt;&gt; A k</td>
<td>Arithmetic right shift</td>
</tr>
<tr>
<td>SHR k, D</td>
<td>D ← D &gt;&gt; L k</td>
<td>Logical right shift</td>
</tr>
</tbody>
</table>

What are the values of the destination registers after each of the following instructions executes in sequence?

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x12</td>
</tr>
<tr>
<td>%rbx</td>
<td>0x56</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x02</td>
</tr>
<tr>
<td>%rdx</td>
<td>0xF0</td>
</tr>
</tbody>
</table>

**Figure 3.10** Integer arithmetic operations. The load effective address (lea q) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation >> A and >> L to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.
Hand-writing x86_64 assembly

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

  .globl main
  main:

  movq $0, %rax  # your code goes here

  ret

- Save in .s file and build with gcc as usual (don’t use “-c” flag)
  - Run program and view return value 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