Assembly Language (Machine Programming)  
Introduction

• Learning Objectives
  • Explain what assembly language is
  • Define
    • Registers
    • Instruction
    • Operands
  • Produce assembly from C
  • Figure out how the following are expressed in assembly
    • Arithmetic operations
    • Logical operations
  • Figure out how arguments are passed to functions
What is assembly?

• Yet another layer of abstraction!
• When you strip away C, the assembly language is a human readable representation that more closely matches the hardware.
• Typically each assembly instruction corresponds to a machine instruction.
• Assembly doesn’t really manipulate variables; it expresses computation in terms of:
  • Registers
  • Memory
  • Instructions
A Note on our Assembly

- We are using Intel x86-64.
  - This means that we are using Intel’s 64-bit architecture
- The 3rd edition of the book uses this architecture in most sections, but still has some remnants of the 32-bit architecture in places.
- The two are quite similar, but you want to be sure to understand the 64-bit architecture.
- Three ways to see assembly output:
  - cc –S x.c
  - objdump –d x.o
  - In gdb: disas(semble) <address>
Screen Capture
Example

.file "f00.c"
.text
.globl f
.type f,@function

f:
.LFB0:
  rep ret
.LFE0:
  .size f, .-f
.ident "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3") 4.8.4"
.section .note.GNU-stack,"",@progbits
Registers

- Registers are fast memory in the processor.
  - Processors execute many instructions in a single cycle; accessing memory can take 10s or 100s of cycles; placing data in registers allows the processor to execute things more quickly.
  - Most processors have a few tens of registers.
  - The Intel x86-64 has 16 64-bit general purpose registers:
    - %rax, %rbx, %rcx, %rdx, %rbp, %rsi, %rdi, %rsp, %r8-%r15
    - Some conventions for how some of the registers are used.
      - For example:
        - %rbp is the frame pointer
        - %rax is used to return values from procedure calls
        - %rdi, %rsi, %rdx, %rcx, %r8, %r9 are used to pass argument to procedures
Parts of Registers

• In assembly language, we don’t really have types like we do in C, but we do operate upon data in different sized units:
  • Double Quad word: 128 bits
  • Quad word: 64 bits (q: 8 bytes)
  • Double word: 32 bits (l: 4 bytes)
  • Word: 16 bits (w: 2 bytes)
  • Byte: 8 bits (b: 1 byte)

• While registers are quad words, we can access smaller items in registers, using different names for the register. Consider %rax:
  • %eax references the low order 32 bits of %rax (a double word)
  • %ax references the low order 16 bits of %rax (a word)
  • %al references the low order 8 bits of %rax (a byte)
  • %ah references bits 8-16 of %rax (also a byte)
  • These conventions apply to %rbx, %rcx, etc.
  • However, for registers %r8 - %r16, we use:
    • %r8d, %r8w, %r8b
Kinds of instructions

• Move data around
• Perform arithmetic operations
• Perform logical operations
• Compare things (sets condition flags)
• Flow control
Screen capture
Checkpoint 1

• Registers are referenced with %
• When we see an `imull` operation like:
  
  \[ \text{OP } \text{operand1, operand2, operand3} \]

  It means
  
  \[ \text{operand3} = \text{operand1} \times \text{operand2} \]

• When we see an `add` operation like:
  
  \[ \text{OP } \text{operand1, operand2} \]

  It means:
  
  \[ \text{operand2} = \text{operand2} \text{ OP } \text{operand1} \]

• The first argument was in `%edi`.
• The second argument was in `%esi`.
• We returned the result in `%eax`. 
What questions should we ask?

- Registers are referenced with %
- When we see an `imull` operation like:
  ```
  OP operand1, operand2, operand3
  It means
  operand3 = operand1 * operand2
  ```
- When we see an `add` operation like:
  ```
  OP operand1, operand2
  It means:
  operand2 = operand2 OP operand1
  ```
- The first argument was in `%edi`.
- The second argument was in `%esi`.
- We returned the result in `%eax`.

What other operations act like `imull` and which ones act like `add`?

What happens if we use longs instead of ints?

What if we have more than 2 arguments?
Screen capture
What questions should we ask?

- Registers are referenced with %
- When we see an `imull` operation like:
  ```plaintext
  OP operand1, operand2, operand3
  It means
  operand3 = operand1 * operand2
  ```
- When we see an `add` operation like:
  ```plaintext
  OP operand1, operand2
  It means:
  operand2 = operand2 OP operand1
  ```
- The first argument was in `%edi`.
- The second argument was in `%esi`.
- We returned the result in `%eax`.

Add, sub, and, or, xor all seem to have the same structure.

What happens if we use longs instead of ints?

What if we have more than 2 arguments?
Screen Capture
What questions should we ask?

- Registers are referenced with %
- When we see an **imull** operation like:
  
  \[ \text{OP } \text{operand1, operand2, operand3} \]
  
  It means
  
  \[ \text{operand3} = \text{operand1} \times \text{operand2} \]

- When we see an **add** operation like:
  
  \[ \text{OP } \text{operand1, operand2} \]
  
  It means:
  
  \[ \text{operand2} = \text{operand2} \text{ OP } \text{operand1} \]

- The first argument was in %edi.
- The second argument was in %esi.
- We returned the result in %eax.

---

Add, sub, and, or, xor all seem to have the same structure. Only **imull** and **idiv** seem to have the 3-op versions…

What happens if we use longs instead of ints?

What if we have more than 2 arguments?
Screen Capture
What questions should we ask?

- Registers are referenced with %
- When we see an `imull` operation like:
  
  ```
  OP operand1, operand2, operand3
  ```
  It means
  
  ```
  operand3 = operand1 * operand2
  ```
- When we see an `add` operation like:
  
  ```
  OP operand1, operand2
  ```
  It means:
  
  ```
  operand2 = operand2 OP operand1
  ```
- The first argument was in `%edi`.
- The second argument was in `%esi`.
- We returned the result in `%eax`.

Add, sub, and, or, xor all seem to have the same structure. Only `imull` and `idiv` seem to have the 3-op versions…

We change “types” in assembly by using instructions and registers of different sizes.

What if we have more than 2 arguments?
Screen Capture
What questions should we ask?

- Registers are referenced with %
- When we see an imull operation like:
  \[ \text{OP operand1, operand2, operand3} \]
  It means
  \[ \text{operand3} = \text{operand1} \times \text{operand2} \]
- When we see an add operation like:
  \[ \text{OP operand1, operand2} \]
  It means:
  \[ \text{operand2} = \text{operand2 OP operand1} \]
- The first argument was in %edi.
- The second argument was in %esi.
- We returned the result in %eax.

Add, sub, and, or, xor all seem to have the same structure.
Only imull and idiv seem to have the 3-op versions…

We change “types” in assembly by using instructions and registers of different sizes.

Parameters are passed in:
- %rdi (arg1)
- %rsi (arg2)
- %rdx (arg3)
- %rcx (arg4)
- %r8 (arg5)
- %r9 (arg6)
- … and then on the stack