Assembly Language: Part 1
Jennifer Rexford

Context of this Lecture
First half lectures: “Programming in the large”
Second half lectures: “Under the hood”

Goals of this Lecture
Help you learn:
• Language levels
• The basics of x86-64 architecture
  • Enough to understand x86-64 assembly language
• The basics of x86-64 assembly language
  • Instructions to define global data
  • Instructions to transfer data and perform arithmetic

Lectures vs. Precepts
Approach to studying assembly language:

<table>
<thead>
<tr>
<th>Precepts</th>
<th>Lectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Study complete pgms</td>
<td>Study partial pgms</td>
</tr>
<tr>
<td>Begin with small pgms; proceed to large ones</td>
<td>Begin with simple constructs; proceed to complex ones</td>
</tr>
<tr>
<td>Emphasis on writing code</td>
<td>Emphasis on reading code</td>
</tr>
</tbody>
</table>

Agenda
Language Levels
Architecture
Assembly Language: Defining Global Data
Assembly Language: Performing Arithmetic

High-Level Languages
Characteristics
• Portable
• To varying degrees
• Complex
  • One statement can do much work
• Expressive
  • To varying degrees
  • Good (code functionality / code size) ratio
• Human readable

```c
count = 0;
while (n>1) {
  count++;
  if (n&1) {
    n = n*3+1;
  } else {
    n = n/2;
  }
} 
```
Machine Languages

Characteristics
• Not portable
• Specific to hardware
• Simple
  • Each instruction does a simple task
• Not expressive
  • Each instruction performs little work
• Poor (code functionality / code size) ratio
• Not human readable
  • Requires lots of effort!
  • Requires tool support

Assembly Languages

Characteristics
• Not portable
• Each assembly language instruction maps to one machine language instruction
• Simple
  • Each instruction does a simple task
• Not expressive
  • Poor (code functionality / code size) ratio
• Human readable!!!

Why Learn Assembly Language?

Q: Why learn assembly language?
A: Knowing assembly language helps you:
• Write faster code
• In assembly language
• In a high-level language!
• Understand what’s happening “under the hood”
  • Someone needs to develop future computer systems
  • Maybe that will be you!

Why Learn x86-64 Assembly Language?

Why learn x86-64 assembly language?
Pros
• X86-64 is popular
• FC010 computers are x86-64 computers
  • Program natively on FC010 instead of using an emulator
Cons
• X86-64 assembly language is big
  • Each instruction is simple, but...
  • There are many instructions
  • Instructions differ widely

x86-64 Assembly Lang Subset

We’ll study a popular subset
• As defined by precept x86-64 Assembly Language document

We’ll study programs define functions that:
• Do not use floating point values
• Have parameters that are integers or addresses (but not structures)
• Have return values that are integers or addresses (but not structures)
• Have no more than 6 parameters

Claim: a reasonable subset

Agenda

Language Levels
Architecture
Assembly Language: Defining Global Data
Assembly Language: Performing Arithmetic
John Von Neumann (1903-1957)

In computing
- Stored program computers
- Cellular automata
- Self-replication

Other interests
- Mathematics
- Nuclear physics (hydrogen bomb)

Princeton connection
- Princeton Univ & IAS, 1930-death

Known for “Von Neumann architecture”
- In contrast to less successful “Harvard architecture”

Von Neumann Architecture

RAM (Random Access Memory)
- Conceptually: large array of bytes

Registers
- Small amount of storage on the CPU
- Much faster than RAM
- Top of the storage hierarchy
- Above RAM, disk, ...

General purpose registers:

<table>
<thead>
<tr>
<th>Register</th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RSP is unique; see upcoming slide
### Registers

**General purpose registers (cont.):**

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
<td>R8D</td>
<td>R8W</td>
<td>R8B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td>R9D</td>
<td>R9W</td>
<td>R9B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td>R10D</td>
<td>R10W</td>
<td>R10B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td>R11D</td>
<td>R11W</td>
<td>R11B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td>R12D</td>
<td>R12W</td>
<td>R12B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R13</td>
<td>R13D</td>
<td>R13W</td>
<td>R13B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R14</td>
<td>R14D</td>
<td>R14W</td>
<td>R14B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R15</td>
<td>R15D</td>
<td>R15W</td>
<td>R15B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### RSP Register

**RSP (Stack Pointer) register**
- Contains address of top (low address) of current function’s stack frame

- Allows use of the STACK section of memory
  
  (See Assembly Language: Function Calls lecture)

### EFLAGS Register

**EFLAGS (Flags) register**
- Contains CC (Condition Code) bits
- Affected by compare (cmp) instruction
- And many others
- Used by conditional jump instructions
  - je, jne, jl, jg, jle, jge, jb, jbe, ja, jae, jb

(See Assembly Language: Part 2 lecture)

### RIP Register

**Special-purpose register...**

**RIP (Instruction Pointer) register**
- Stores the location of the next instruction
  - Address (in TEXT section) of machine-language instructions to be executed next
- Value changed:
  - Automatically to implement sequential control flow
  - By jump instructions to implement selection, repetition

### Registers and RAM

**Typical pattern:**
- **Load** data from RAM to registers
- **Manipulate** data in registers
- **Store** data from registers to RAM

Many instructions combine steps
**ALU (Arithmetic Logic Unit)**
- Performs arithmetic and logic operations

**CPU (Central Processing Unit)**
- Control unit
- Fetch, decode, and execute
- ALU
- Execute low-level operations
- Registers
  - High-speed temporary storage

**Control Unit**
- Fetches and decodes each machine-language instruction
- Sends proper data to ALU

**Defining Data: DATA Section 1**
```
static char c = 'a';
static short s = 12;
static int i = 345;
static long l = 6789;
```

**Defining Data: DATA Section 2**
```
char c = 'a';
short s = 12;
int i = 345;
long l = 6789;
```

**Note:**
- Use `.globl` to mark a spot in RAM
- Use `.section` to announce DATA section
- Use `.byte`, `.word`, `.long`, `.quad` to specify data size
- Best to avoid "word" (2 byte) data

**Agenda**
- Language Levels
- Architecture
- Assembly Language: Defining Global Data
- Assembly Language: Performing Arithmetic
Defining Data: BSS Section

```c
static char c;
static short s;
static int i;
static long l;
```

Note:
- `.section` instruction (to announce BSS section)
- `.skip` instruction

---

Defining Data: RODATA Section

```c
"hello\n";
```

Note:
- `.section` instruction (to announce RODATA section)
- `.string` instruction

---

Agenda

- Language Levels
- Architecture
- Assembly Language: Defining Global Data
- Assembly Language: Performing Arithmetic

---

Instruction Format

Many instructions have this format:

```
name{b,w,l,q} src, dest
```

- `name`: name of the instruction (mov, add, sub, and, etc.)
- `byte` => operands are one-byte entities
- `word` => operands are two-byte entities
- `long` => operands are four-byte entities
- `quad` => operands are eight-byte entities

---

Instruction Format

Many instructions have this format:

```
name{b,w,l,q} src, dest
```

- `src`: source operand
  - The source of data
  - Can be
    - Register operand: `%rax, %ebx, etc.
    - Memory operand: `% (legal but silly), someLabel`
    - Immediate operand: `$5, $someLabel`

---

Instruction Format

Many instructions have this format:

```
name{b,w,l,q} src, dest
```

- `dest`: destination operand
  - The destination of data
  - Can be
    - Register operand: `%rax, %ebx, etc.
    - Memory operand: `% (legal but silly), someLabel`
    - Cannot be
    - Immediate operand
Performing Arithmetic: Long Data

static int length;
static int width;
static int perim;

perim = (length + width) * 2;

Note:
- movl instruction
- addl instruction
- sall instruction
- Immediate operand
- Memory operand

Performing Arithmetic: Byte Data

static char grade = 'B';

grade--;

Note:
- movb instruction
- subb instruction
- decb instruction

What would happen if we use movl instead of movb?

Generalization: Operands

Immediate operands
- $5 => use the number 5 (i.e. the number that is available immediately within the instruction)
- $i => use the address denoted by i (i.e. the address that is available immediately within the instruction)
- Can be source operand; cannot be destination operand

Register operands
- %rax => read from (or write to) register RAX
- Can be source or destination operand

Memory operands
- $5 => load from (or store to) memory at address 5 (silly; seg fault)
- i => load from (or store to) memory at the address denoted by i
- Can be source or destination operand (but not both)
- There’s more to memory operands; see next lecture

Generalization: Data Transfer

Data transfer instructions

movb(q, l, w) srcRM, destRM dest = src
movwb(q, l, w) srcRM, destRM dest = src (sign extend)
movwb(q, l, w) srcRM, destRM dest = src (zero fill)

movl(q, l, w) srcRM, destRM dest = src
movlw(q, l, w) srcRM, destRM dest = src (zero fill)

cqto reg[EDR:RAX] = reg[RAK] (sign extend)
cldt reg[EDR:RAK] = reg[RAK] (sign extend)
cwcl reg[RAK] = reg[RAK] (sign extend)
cwtl reg[AX] = reg[AX] (sign extend)
cbtw reg[AX] = reg[AL] (sign extend)

mov is used often; others less so

Generalization: Arithmetic

Arithmetic instructions

addb(q, l, w) srcRM, destRM dest += src
subb(q, l, w) srcRM, destRM dest -= src
incb(q, l, w) destRM dest++
decb(q, l, w) destRM dest--

movl(q, l, w) srcRM, destRM dest = src
subl(q, l, w) srcRM, destRM dest = src (sign extend)
incl(q, l, w) destRM dest++
dcl(q, l, w) destRM dest--

Note:
- # Option 1
- # Option 2
- # Option 3
### Generalization: Signed Mult & Div

**Signed multiplication and division instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imulq srcRM</td>
<td>reg[RD:RA] = reg[RA]*src</td>
</tr>
<tr>
<td>imull srcRM</td>
<td>reg[ED:EA] = reg[EA]*src</td>
</tr>
<tr>
<td>imulw srcRM</td>
<td>reg[DX:AX] = reg[AX]*src</td>
</tr>
<tr>
<td>idivw srcRM</td>
<td>reg[DX:AX] = reg[AX]%src</td>
</tr>
<tr>
<td>idivb srcRM</td>
<td>reg[AX] = reg[AL]%src</td>
</tr>
</tbody>
</table>

See Bryant & O’Hallaron book for description of signed vs. unsigned multiplication and division

### Generalization: Unsigned Mult & Div

**Unsigned multiplication and division instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulq srcRM</td>
<td>reg[RD:RA] = reg[RA]*src</td>
</tr>
<tr>
<td>mull srcRM</td>
<td>reg[ED:EA] = reg[EA]*src</td>
</tr>
<tr>
<td>mulw srcRM</td>
<td>reg[DX:AX] = reg[AX]*src</td>
</tr>
<tr>
<td>divw srcRM</td>
<td>reg[DX:AX] = reg[AX]%src</td>
</tr>
<tr>
<td>divb srcRM</td>
<td>reg[AX] = reg[AL]%src</td>
</tr>
</tbody>
</table>

See Bryant & O’Hallaron book for description of signed vs. unsigned multiplication and division

### Generalization: Bit Manipulation

**Bitwise instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and{q,l,w,b} srcIR, destRM</td>
<td>dest = src &amp; dest</td>
</tr>
<tr>
<td>or{q,l,w,b} srcIR, destRM</td>
<td>dest = src</td>
</tr>
<tr>
<td>xor{q,l,w,b} srcIR, destRM</td>
<td>dest = src ^ dest</td>
</tr>
<tr>
<td>not{q,l,w,b} destRM</td>
<td>dest = ~dest</td>
</tr>
<tr>
<td>sal{q,l,w,b} srcIR, destRM</td>
<td>dest = dest &lt;&lt; src</td>
</tr>
<tr>
<td>sar{q,l,w,b} srcIR, destRM</td>
<td>dest = dest &gt;&gt; src (sign extend)</td>
</tr>
<tr>
<td>shl{q,l,w,b} srcIR, destRM</td>
<td>(Same as sal)</td>
</tr>
<tr>
<td>shr{q,l,w,b} srcIR, destRM</td>
<td>dest = dest &gt;&gt; src (zero fill)</td>
</tr>
</tbody>
</table>

### Summary

**Language levels**

- The basics of computer architecture
  - Enough to understand x86-64 assembly language
- The basics of x86-64 assembly language
  - Instructions to define global data
  - Instructions to perform data transfer and arithmetic

**To learn more**

- Study more assembly language examples
- Chapter 3 of Bryant and O’Hallaron book
- Study compiler-generated assembly language code
  - gcc217 -S somefile.c

### Appendix

**Big-endian vs. little-endian byte order**

<table>
<thead>
<tr>
<th>Intel is a little endian architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least significant byte of multi-byte entity is stored at lowest memory address</td>
<td>&quot;Little end goes first&quot;</td>
</tr>
<tr>
<td>The int 5 at address 1000:</td>
<td>1000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>Some other systems use big endian</td>
<td>Description</td>
</tr>
<tr>
<td>Most significant byte of multi-byte entity is stored at lowest memory address</td>
<td>&quot;Big end goes first&quot;</td>
</tr>
<tr>
<td>The int 5 at address 1000:</td>
<td>1000 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>
Byte Order Example 1

```c
#include <stdio.h>
int main(void)
{
    unsigned int i = 0x003377ff;
    unsigned char *p;
    int j;
    p = (unsigned char *)&i;
    for (j=0; j<4; j++)
        printf("Byte %d: %2x\n", j, p[j]);
}
```

Output on a little-endian machine:
- Byte 0: ff
- Byte 1: 77
- Byte 2: 33
- Byte 3: 00

Output on a big-endian machine:
- Byte 0: 00
- Byte 1: 33
- Byte 2: 77
- Byte 3: ff

---

Byte Order Example 2

Note:
Flawed code; uses "b" instructions to manipulate a four-byte memory area.

Intel is little endian, so what will be the value of grade?

What would be the value of grade if Intel were big endian?

---

Byte Order Example 3

Note:
Flawed code; uses "l" instructions to manipulate a one-byte memory area.

What would happen?