Assembly Language:
Part 1
Context of this Lecture

First half lectures: “Programming in the large”
Second half lectures: “Under the hood”

Starting Now

- C Language
- Assembly Language
- Machine Language

language levels tour

Afterward

- Application Program
- Operating System
- Hardware

service levels tour
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 <strong>complete</strong> pgms</td>
<td>Study <strong>partial</strong> pgms</td>
</tr>
<tr>
<td>Begin with <strong>small</strong> pgms; proceed to <strong>large</strong> ones</td>
<td>Begin with <strong>simple</strong> constructs; proceed to <strong>complex</strong> ones</td>
</tr>
<tr>
<td>Emphasis on <strong>writing</strong> code</td>
<td>Emphasis on <strong>reading</strong> 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 lang instruction maps to one machine lang instruction

• Simple
  • Each instruction does a simple task

• Not expressive
  • Poor (code functionality / code size) ratio

• Human readable!!!
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 Lang?

Why learn x86-64 assembly language?

Pros

• X86-64 is popular
• CourseLab computers are x86-64 computers
  • Program natively on CourseLab 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

- Control Unit
- ALU
- Registers
- CPU
- Data bus
- RAM
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, …
**Registers**

**General purpose registers:**

<table>
<thead>
<tr>
<th>Register</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>64-bit</td>
</tr>
<tr>
<td>EAX</td>
<td>32-bit</td>
</tr>
<tr>
<td>AX</td>
<td>16-bit</td>
</tr>
<tr>
<td>AL</td>
<td>8-bit</td>
</tr>
<tr>
<td>RBX</td>
<td>64-bit</td>
</tr>
<tr>
<td>EBX</td>
<td>32-bit</td>
</tr>
<tr>
<td>BX</td>
<td>16-bit</td>
</tr>
<tr>
<td>BL</td>
<td>8-bit</td>
</tr>
<tr>
<td>RCX</td>
<td>64-bit</td>
</tr>
<tr>
<td>ECX</td>
<td>32-bit</td>
</tr>
<tr>
<td>CX</td>
<td>16-bit</td>
</tr>
<tr>
<td>CL</td>
<td>8-bit</td>
</tr>
<tr>
<td>RDX</td>
<td>64-bit</td>
</tr>
<tr>
<td>EDX</td>
<td>32-bit</td>
</tr>
<tr>
<td>DX</td>
<td>16-bit</td>
</tr>
<tr>
<td>DL</td>
<td>8-bit</td>
</tr>
</tbody>
</table>
General purpose registers (cont.):

RSP is unique; see upcoming slide
Registers

General purpose registers (cont.):

<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>R8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Registers

### General purpose registers (cont.):

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R12</td>
<td>R12D</td>
<td>R12W</td>
<td>R12B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R13</td>
<td>R13D</td>
<td>R13W</td>
<td>R13B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R14</td>
<td>R14D</td>
<td>R14W</td>
<td>R14B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R15</td>
<td>R15D</td>
<td>R15W</td>
<td>R15B</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

Special-purpose 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
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

![ALU Diagram]

- src1
- src2
- dest
- operation
- EFLAGS

Control Unit

Registers

ALU

CPU

RAM

Data bus
Control Unit

• Fetches and decodes each machine-language instruction
• Sends proper data to ALU
CPU (Central Processing Unit)

- Control unit
  - Fetch, decode, and execute
- ALU
  - Execute low-level operations
- Registers
  - High-speed temporary storage
Agenda

Language Levels
Architecture

Assembly Language: Defining Global Data
Assembly Language: Performing Arithmetic
Defining Data: DATA Section 1

```c
static char c = 'a';
static short s = 12;
static int i = 345;
static long l = 6789;
```

Note:
- `.section` instruction (to announce DATA section)
- label definition (marks a spot in RAM)
- `.byte` instruction (1 byte)
- `.word` instruction (2 bytes)
- `.long` instruction (4 bytes)
- `.quad` instruction (8 bytes)

Note:
Best to avoid “word” (2 byte) data
Defining Data: DATA Section 2

char c = 'a';
short s = 12;
int i = 345;
long l = 6789;

Note:
Can place label on same line as next instruction
.globl instruction
Defining Data: BSS Section

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

.section ".bss"
  c:  .skip 1
  s:  .skip 2
  i:  .skip 4
  l:  .skip 8

Note:
  .section instruction (to announce BSS section)
  .skip instruction
Defining Data: RODATA Section

..."hello\n"...;
...

.section "\rodata"
helloLabel:
.string "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:

\[
\text{name}\{\text{b, w, l, q}\} \text{ src, dest}
\]

- **name**: name of the instruction (\text{mov}, \text{add}, \text{sub}, \text{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
Many instructions have this format:

\[
\text{name\{b,w,l,q\} src, dest}
\]

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

Many instructions have this format:

\[ \text{name}\{b,w,l,q\} \ src, \ dest \]

- **dest**: destination operand
  - The destination of data
  - Can be
    - **Register operand**: %rax, %ebx, etc.
    - **Memory operand**: 5 (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
Register operand
Immediate operand
Memory operand

.section " .bss"
length: .skip 4
width: .skip 4
perim: .skip 4
...

[section " .text"
...

movl length, %eax
addl width, %eax
sall $1, %eax
movl %eax, perim

.section instruction (to announce TEXT section)
```c
static char grade = 'B';
...
grade--;
```

```plaintext
Note:
Comment
movb instruction
subb instruction
decb instruction
```

What would happen if we use `movl` instead of `movb`?
Generalization: Operands

Immediate operands
- \$5 \rightarrow use the number 5 (i.e. the number that is available immediately within the instruction)
- \$i \rightarrow 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 \rightarrow read from (or write to) register RAX
- Can be source or destination operand

Memory operands
- 5 \rightarrow load from (or store to) memory at address 5 (silly; seg fault)
- i \rightarrow 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: Notation

Instruction notation:
- q => quad (8 bytes); l => long (4 bytes);
- w => word (2 bytes); b => byte (1 byte)

Operand notation:
- src => source; dest => destination
- R => register; I => immediate; M => memory
### Data transfer instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov(q,l,w,b)</code> srcIRM, destRM</td>
<td>dest = src</td>
</tr>
<tr>
<td><code>movsb(q,l,w)</code> srcRM, destR</td>
<td>dest = src (sign extend)</td>
</tr>
<tr>
<td><code>movsw(q,l)</code> srcRM, destR</td>
<td>dest = src (sign extend)</td>
</tr>
<tr>
<td><code>movslq</code> srcRM, destR</td>
<td>dest = src (sign extend)</td>
</tr>
<tr>
<td><code>movzb(q,l,w)</code> srcRM, destR</td>
<td>dest = src (zero fill)</td>
</tr>
<tr>
<td><code>movzw(q,l)</code> srcRM, destR</td>
<td>dest = src (zero fill)</td>
</tr>
<tr>
<td><code>movzlq</code> srcRM, destR</td>
<td>dest = src (zero fill)</td>
</tr>
<tr>
<td><code>cqto</code></td>
<td>reg[RDX:RAX] = reg[RAX] (sign extend)</td>
</tr>
<tr>
<td><code>cltd</code></td>
<td>reg[EDX:EAX] = reg[EAX] (sign extend)</td>
</tr>
<tr>
<td><code>cwtl</code></td>
<td>reg[EAX] = reg[AX] (sign extend)</td>
</tr>
<tr>
<td><code>cbtw</code></td>
<td>reg[AX] = reg[AL] (sign extend)</td>
</tr>
</tbody>
</table>

*mov* is used often; others less so
Generalization: Arithmetic

Arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add{q,l,w,b}</td>
<td>srcIRM</td>
<td>destRM</td>
<td>dest += src</td>
</tr>
<tr>
<td>sub{q,l,w,b}</td>
<td>srcIRM</td>
<td>destRM</td>
<td>dest -= src</td>
</tr>
<tr>
<td>inc{q,l,w,b}</td>
<td>destRM</td>
<td></td>
<td>dest++</td>
</tr>
<tr>
<td>dec{q,l,w,b}</td>
<td>destRM</td>
<td></td>
<td>dest--</td>
</tr>
<tr>
<td>neg{q,l,w,b}</td>
<td>destRM</td>
<td></td>
<td>dest = -dest</td>
</tr>
</tbody>
</table>
Signed multiplication and division instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>imulq srcRM</td>
<td>reg[RDX:RAX] = reg[RAX]*src</td>
</tr>
<tr>
<td>imull srcRM</td>
<td>reg[EDX:EAX] = reg[EAX]*src</td>
</tr>
<tr>
<td>imulw srcRM</td>
<td>reg[DX:AX] = reg[AX]*src</td>
</tr>
<tr>
<td>imulb srcRM</td>
<td>reg[AX] = reg[AL]*src</td>
</tr>
<tr>
<td>idivq srcRM</td>
<td>reg[RAX] = reg[RDX:RAX]/src</td>
</tr>
<tr>
<td></td>
<td>reg[RDX] = reg[RDX:RAX]%src</td>
</tr>
<tr>
<td>idivl srcRM</td>
<td>reg[EAX] = reg[EDX:EAX]/src</td>
</tr>
<tr>
<td></td>
<td>reg[EDX] = reg[EDX:EAX]%src</td>
</tr>
<tr>
<td>idivw srcRM</td>
<td>reg[AX] = reg[DX:AX]/src</td>
</tr>
<tr>
<td></td>
<td>reg[DX] = reg[DX:AX]%src</td>
</tr>
<tr>
<td>idivb srcRM</td>
<td>reg[AL] = reg[AX]/src</td>
</tr>
<tr>
<td></td>
<td>reg[AH] = reg[AX]%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>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mulq srcRM</code></td>
<td><code>reg[RDX:RAX] = reg[RAX]*src</code></td>
</tr>
<tr>
<td><code>mull srcRM</code></td>
<td><code>reg[EDX:EAX] = reg[EAX]*src</code></td>
</tr>
<tr>
<td><code>mulw srcRM</code></td>
<td><code>reg[DX:AX] = reg[AX]*src</code></td>
</tr>
<tr>
<td><code>mulb srcRM</code></td>
<td><code>reg[AX] = reg[AL]*src</code></td>
</tr>
<tr>
<td><code>divq srcRM</code></td>
<td><code>reg[RAX] = reg[RDX:RAX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[RDX] = reg[RDX:RAX]%src</code></td>
</tr>
<tr>
<td><code>divl srcRM</code></td>
<td><code>reg[EAX] = reg[EDX:EAX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[EDX] = reg[EDX:EAX]%src</code></td>
</tr>
<tr>
<td><code>divw srcRM</code></td>
<td><code>reg[AX] = reg[DX:AX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[DX] = reg[DX:AX]%src</code></td>
</tr>
<tr>
<td><code>divb srcRM</code></td>
<td><code>reg[AL] = reg[AX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[AH] = reg[AX]%src</code></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>Operation</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>and{q,l,w,b}</td>
<td>srcIRM, destRM</td>
<td>dest = src &amp; dest</td>
</tr>
<tr>
<td>or{q,l,w,b}</td>
<td>srcIRM, destRM</td>
<td>dest = src</td>
</tr>
<tr>
<td>xor{q,l,w,b}</td>
<td>srcIRM, destRM</td>
<td>dest = src ^ dest</td>
</tr>
<tr>
<td>not{q,l,w,b}</td>
<td>destRM</td>
<td>dest = ~dest</td>
</tr>
<tr>
<td>sal{q,l,w,b}</td>
<td>srcIR, destRM</td>
<td>dest = dest &lt;&lt; src</td>
</tr>
<tr>
<td>sar{q,l,w,b}</td>
<td>srcIR, destRM</td>
<td>dest = dest &gt;&gt; src (sign extend)</td>
</tr>
<tr>
<td>shl{q,l,w,b}</td>
<td>srcIR, destRM</td>
<td>(Same as sal)</td>
</tr>
<tr>
<td>shr{q,l,w,b}</td>
<td>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
Byte Order

x86-64 is a **little endian** architecture

- **Least** significant byte of multi-byte entity is stored at lowest memory address
- “Little end goes first”

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>00000101</td>
</tr>
<tr>
<td>1001</td>
<td>00000000</td>
</tr>
<tr>
<td>1002</td>
<td>00000000</td>
</tr>
<tr>
<td>1003</td>
<td>00000000</td>
</tr>
</tbody>
</table>

The int 5 at address 1000:

Some other systems use **big endian**

- **Most** significant byte of multi-byte entity is stored at lowest memory address
- “Big end goes first”

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>00000000</td>
</tr>
<tr>
<td>1001</td>
<td>00000000</td>
</tr>
<tr>
<td>1002</td>
<td>00000000</td>
</tr>
<tr>
<td>1003</td>
<td>00000101</td>
</tr>
</tbody>
</table>

The int 5 at address 1000:
#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]);
}
Note:
Flawed code; uses "b" instructions to manipulate a four-byte memory area

x86-64 is **little** endian, so what will be the value of grade?

What would be the value of grade if x86-64 were **big** endian?

```
.section "\data"
grade: .long 'B'
...

.section "\text"
...
# Option 1
movb grade, %al
subb $1, %al
movb %al, grade
...
# Option 2
subb $1, grade
```
Note:
Flawed code; uses “I” instructions to manipulate a one-byte memory area

What would happen?

```
.section ".data"
grade: .byte 'B'
...

.section ".text"
...
# Option 1
movl grade, %eax
subl $1, %eax
movl %eax, grade
...
# Option 2
subl $1, grade
```