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
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!!!

```
loop:
  movl $0, %r10d
  cmpl $1, %r11d
  jle endloop
  addl $1, %r10d
  movl %r11d, %eax
  andl $1, %eax
  je else
  movl %r11d, %eax
  addl %eax, %r11d
  addl $1, %r11d
  jmp endif
else:
  sarl $1, %r11d
endif:
endloop:
```
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
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
• Inventor of game theory
• Nuclear physics (hydrogen bomb)

Princeton connection
• Princeton Univ & IAS, 1930-1957

Known for “Von Neumann architecture (1950)”
• In which programs are just data in the memory
• Contrast to the now-obsolete “Harvard architecture”
Von Neumann Architecture

- Control Unit
- ALU
- Registers
- CPU
- RAM
- Data bus
RAM (Random Access Memory)
Conceptually: large array of bytes

- Contains data
  (program variables, structs, arrays)
- and the program!

Instructions are fetched from RAM
So is data

Von Neumann Architecture
Von Neumann Architecture

Registers
- Small amount of storage on the CPU
- Much faster than RAM
- Top of the storage hierarchy
  - Above RAM, disk, …
## Registers (x86-64 architecture)

### General purpose registers:

<table>
<thead>
<tr>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>EAX</td>
<td>AX</td>
<td>AL</td>
<td></td>
</tr>
<tr>
<td>RBX</td>
<td>EBX</td>
<td>BX</td>
<td>BL</td>
<td></td>
</tr>
<tr>
<td>RCX</td>
<td>ECX</td>
<td>CX</td>
<td>CL</td>
<td></td>
</tr>
<tr>
<td>RDX</td>
<td>EDX</td>
<td>DX</td>
<td>DL</td>
<td></td>
</tr>
</tbody>
</table>
Registers (x86-64 architecture)

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>RSI</td>
<td></td>
<td>ESI</td>
<td>SI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDI</td>
<td></td>
<td>EDI</td>
<td>DI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBP</td>
<td></td>
<td>EBP</td>
<td>BP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSP</td>
<td></td>
<td>ESP</td>
<td>SP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RSP is unique; see upcoming slide
# Registers (x86-64 architecture)

## 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>R8</td>
<td>R8D</td>
<td>R8W</td>
<td>R8B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td>R9D</td>
<td>R9W</td>
<td>R9B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td>R10D</td>
<td>R10W</td>
<td>R10B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td>R11D</td>
<td>R11W</td>
<td>R11B</td>
<td></td>
<td></td>
</tr>
<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, and special-purpose stack manipulation instructions

(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, ...**

(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 summary

16 general-purpose 64-bit pointer/long-integer registers, many with stupid names:
rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15

- sometimes used as a “frame pointer” or “base pointer”
- “stack pointer”

If you’re operating on 32-bit “int” data, use these stupid names instead:
eax, ebx, ecx, edx, esi, edi, ebp, rsp, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d

- it doesn’t really make sense to put 32-bit ints in the stack pointer

2 special-purpose registers:
eflags, rip
- “condition codes”
- “program counter”
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](image)
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)
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"
  .skip 1
  c:
  .skip 2
  s:
  .skip 4
  i:
  .skip 8
  l:

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:

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

- **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:

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

- **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:

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

- **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 instruction (to announce TEXT section)
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
Performing Arithmetic: Byte Data

```c
static char grade = 'B';
...
grade--; 
```

Note:

Comment

```c
movb instruction
subb instruction
decb instruction
```

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

.section "\.text"
...

# Option 1
movb grade, %al
subb $1, %al
movb %al, grade
...

# Option 2
subb $1, grade
...

# Option 3
decb grade
```
Q: What would happen if we used `movl` instead of `movb`?

A. Would always work correctly
B. Would always work incorrectly
C. Would sometimes work correctly
D. *This* code would work, but something else might go wrong that would cause you sleepless nights of painful debugging
Q: What would happen if we used `subl` instead of `subb`?

A. Would always work correctly
B. Would always work incorrectly
C. Would sometimes work correctly
D. This code would work, but something else might go wrong that would cause you sleepless nights of painful debugging
More Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add{q,l,w,b} srcIRM, destRM</td>
<td>dest += src</td>
</tr>
<tr>
<td>sub{q,l,w,b} srcIRM, destRM</td>
<td>dest -= src</td>
</tr>
<tr>
<td>inc{q,l,w,b} destRM</td>
<td>dest++</td>
</tr>
<tr>
<td>dec{q,l,w,b} destRM</td>
<td>dest--</td>
</tr>
<tr>
<td>neg{q,l,w,b} destRM</td>
<td>dest = -dest</td>
</tr>
</tbody>
</table>

Operand notation:
- src ⇒ source; dest ⇒ destination
- R ⇒ register; I ⇒ immediate; M ⇒ memory
Data Transfer Instructions

```plaintext
mov{q,l,w,b} srcIRM, destRM    dest = src
movsb{q,l,w} srcRM, destR     dest = src (sign extend)
movsw{q,l} srcRM, destR       dest = src (sign extend)
movslq srcRM, destR           dest = src (sign extend)
movzb{q,l,w} srcRM, destR     dest = src (zero fill)
movzw{q,l} srcRM, destR       dest = src (zero fill)
movzlq srcRM, destR           dest = src (zero fill)
cqto                           reg[RDX:RAX] = reg[RAX] (sign extend)
cld                            reg[EDX:EAX] = reg[EAX] (sign extend)
cwtl                           reg[EAX] = reg[AX] (sign extend)
cbtw                           reg[AX] = reg[AL] (sign extend)
```
## Multiplication and Division

### Signed multiplication and division instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>imulq srcRM</code></td>
<td><code>reg[RDX:RAX] = reg[RAX]*src</code></td>
</tr>
<tr>
<td><code>imull srcRM</code></td>
<td><code>reg[EDX:EAX] = reg[EAX]*src</code></td>
</tr>
<tr>
<td><code>imulw srcRM</code></td>
<td><code>reg[DX:AX] = reg[AX]*src</code></td>
</tr>
<tr>
<td><code>imulb srcRM</code></td>
<td><code>reg[AX] = reg[AL]*src</code></td>
</tr>
<tr>
<td><code>idivq 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>idivl 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>idivw 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>idivb 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.
### 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
Bit Manipulation

Bitwise instructions

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

Some other systems use **big endian**

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

The int 5 at address 1000:

<table>
<thead>
<tr>
<th>1000</th>
<th>00000101</th>
</tr>
</thead>
<tbody>
<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:

<table>
<thead>
<tr>
<th>1000</th>
<th>00000101</th>
</tr>
</thead>
<tbody>
<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>
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
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?

```assembly
.section ".data"
grade: .long 'B'
...
.section ".text"
...
# Option 1
movb grade, %al
subb $1, %al
movb %al, grade
...
# Option 2
subb $1, grade
```
Byte Order Example 3

Note:
Flawed code; uses “l” 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
```