Assembly Language: Part 1
## Context of this Lecture

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

<table>
<thead>
<tr>
<th>Starting Now</th>
<th>C Language</th>
<th>Assembly Language</th>
<th>Machine Language</th>
<th>Language levels tour</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>afterward</th>
<th>Application Program</th>
<th>Operating System</th>
<th>Hardware</th>
<th>Service levels tour</th>
</tr>
</thead>
</table>
Goals of this Lecture

Help you learn:

• Language levels
• The basics of IA-32 architecture
  • Enough to understand IA-32 assembly language
• The basics of IA-32 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

| 0000 0000 0000 0000 0000 0000 0000 0000 |
| 0000 0000 0000 0000 0000 0000 0000 0000 |
| 9222 9120 1121 A120 1121 A121 7211 0000 |
| 0000 0001 0002 0003 0004 0005 0006 0007 |
| 0008 0009 000A 000B 000C 000D 000E 000F |
| 0000 0000 0000 FE10 FACE CAFE ACED CEDE |

| 1234 5678 9ABC DEF0 0000 0000 F00D 0000 |
| 0000 0000 EEEE 1111 EEEE 1111 0000 0000 |
| B1B2 F1F5 0000 0000 0000 0000 0000 0000 |
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:
  cmpl $1, %edx
  jle endloop
  addl $1, %ecx
  movl %edx, %eax
  addl $1, %eax
  je else
  movl %edx, %eax
  addl %eax, %edx
  addl %eax, %edx
  addl $1, %edx
  jmp endif
else:
  sarl $1, %edx
endif:
  jmp loop
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 IA-32 Assembly Lang?

Why learn **IA-32** assembly language?

**Pros**
- IA-32 is the most popular processor
- Nobel computers are IA-32 computers
  - Program natively on nobel instead of using an emulator

**Cons**
- IA-32 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 Bryant & O’Hallaron Ch 3 and precept **IA-32 Assembly Language** document
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, …
## Registers

### General purpose registers

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>AH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SI</td>
<td>ESI</td>
</tr>
<tr>
<td>BH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DI</td>
<td>EDI</td>
</tr>
<tr>
<td>CH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BP</td>
<td>EBP</td>
</tr>
<tr>
<td>DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SP</td>
<td>ESP</td>
</tr>
</tbody>
</table>
ESP and EBP Registers

ESP (Stack Pointer) register
• Contains address of top (low address) of current function’s stack frame

EBP (Base Pointer) register
• Contains address of bottom (high address) of current function’s stack frame

Allow effective 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 (\texttt{cmp}) instruction
  • And many others
• Used by conditional jump instructions
  • \texttt{je}, \texttt{jne}, \texttt{jl}, \texttt{jg}, \texttt{jle}, \texttt{jge}, \texttt{jb}, \texttt{jbe}, \texttt{ja}, \texttt{jae}, \texttt{jb}

(See \textit{Assembly Language: Part 2} lecture)
EIP Register

Special-purpose register…

EIP (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

\[ \text{TEXT section} \]
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

![Diagram of ALU with inputs src1 and src2, operations, and outputs dest and EFLAGS]
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
static char c = 'a';
static short s = 12;
static int i = 345;

.section "\.data"
  c:
    .byte 'a'
  s:
    .word 12
  i:
    .long 345

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)

Note:
Best to avoid “word” (2 byte) data
char c = 'a';
short s = 12;
int i = 345;

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;

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
Many instructions have this format:

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

- **name**: name of the instruction (\textit{mov}, \textit{add}, \textit{sub}, \textit{and}, etc.)
- **byte** => operands are one-byte entities
- **word** => operands are two-byte entities
- **long** => operands are four-byte entities
Instruction Format

Many instructions have this format:

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

- **src**: source operand
  - The source of data
  - Can be
    - **Register operand**: `%eax`, `%ebx`, etc.
    - **Memory operand**: 5 (legal but silly), `someLabel`
    - **Immediate operand**: `$5$, `$someLabel$
Many instructions have this format:

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

- **dest**: destination operand
  - The destination of data
  - Can be
    - Register operand: \%eax, \%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)
Performing Arithmetic: Byte Data

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

```
.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
```

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
• \%eax \Rightarrow$ read from (or write to) register EAX
• 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:
• \( l \) => long (4 bytes); \( w \) => word (2 bytes); \( b \) => byte (1 byte)

Operand notation:
• \( \text{src} \) => source; \( \text{dest} \) => destination
• \( R \) => register; \( I \) => immediate; \( M \) => memory
Generalization: Data Transfer

Data transfer instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov{l,w,b} srcIRM, destRM</code></td>
<td><code>dest = src</code></td>
</tr>
<tr>
<td><code>movsb{l,w} srcRM, destR</code></td>
<td><code>dest = src (sign extend)</code></td>
</tr>
<tr>
<td><code>movswl srcRM, destR</code></td>
<td><code>dest = src (sign extend)</code></td>
</tr>
<tr>
<td><code>movzb{l,w} srcRM, destR</code></td>
<td><code>dest = src (zero fill)</code></td>
</tr>
<tr>
<td><code>movzwl srcRM, destR</code></td>
<td><code>dest = src (zero fill)</code></td>
</tr>
<tr>
<td><code>cltd</code></td>
<td><code>reg[EDX:EAX] = reg[EAX]</code></td>
</tr>
<tr>
<td></td>
<td>(sign extend)</td>
</tr>
<tr>
<td><code>cwtd</code></td>
<td><code>reg[DX:AX] = reg[AX]</code></td>
</tr>
<tr>
<td></td>
<td>(sign extend)</td>
</tr>
<tr>
<td><code>cbtw</code></td>
<td><code>reg[AX] = reg[AL]</code></td>
</tr>
<tr>
<td></td>
<td>(sign extend)</td>
</tr>
</tbody>
</table>

**`mov`** is used often; others rarely
Generalization: Arithmetic

Arithmetic instructions

```
add{l,w,b} srcIRM, destRM dest += src
sub{l,w,b} srcIRM, destRM dest -= src
inc{l,w,b} destRM dest++
dec{l,w,b} destRM dest--
neg{l,w,b} destRM dest = -dest
```
Signed multiplication and division instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>imull srcRM</td>
<td>\text{reg}[EDX:EAX] = \text{reg}[EAX] \times \text{src}</td>
</tr>
<tr>
<td>imulw srcRM</td>
<td>\text{reg}[DX:AX] = \text{reg}[AX] \times \text{src}</td>
</tr>
<tr>
<td>imulb srcRM</td>
<td>\text{reg}[AX] = \text{reg}[AL] \times \text{src}</td>
</tr>
</tbody>
</table>
| idivl srcRM  | \text{reg}[EAX] = \text{reg}[EDX:EAX] / \text{src}    \\
|              | \text{reg}[EDX] = \text{reg}[EDX:EAX] \% \text{src}  |
| idivw srcRM  | \text{reg}[AX] = \text{reg}[DX:AX] / \text{src}      \\
|              | \text{reg}[DX] = \text{reg}[DX:AX] \% \text{src}     |
| idivb srcRM  | \text{reg}[AL] = \text{reg}[AX] / \text{src}         \\
|              | \text{reg}[AH] = \text{reg}[AX] \% \text{src}        |

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>mull srcRM</td>
<td>( \text{reg[EDX:EAX]} = \text{reg[EAX]} \times \text{src} )</td>
</tr>
<tr>
<td>mulw srcRM</td>
<td>( \text{reg[DX:AX]} = \text{reg[AX]} \times \text{src} )</td>
</tr>
<tr>
<td>mulb srcRM</td>
<td>( \text{reg[AX]} = \text{reg[AL]} \times \text{src} )</td>
</tr>
<tr>
<td>divl srcRM</td>
<td>( \text{reg[EAX]} = \frac{\text{reg[EDX:EAX]}}{\text{src}} )</td>
</tr>
<tr>
<td></td>
<td>( \text{reg[EDX]} = \text{reg[EDX:EAX]} \mod \text{src} )</td>
</tr>
<tr>
<td>divw srcRM</td>
<td>( \text{reg[AX]} = \frac{\text{reg[DX:AX]}}{\text{src}} )</td>
</tr>
<tr>
<td></td>
<td>( \text{reg[DX]} = \text{reg[DX:AX]} \mod \text{src} )</td>
</tr>
<tr>
<td>divb srcRM</td>
<td>( \text{reg[AL]} = \frac{\text{reg[AX]}}{\text{src}} )</td>
</tr>
<tr>
<td></td>
<td>( \text{reg[AH]} = \text{reg[AX]} \mod \text{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>Operator</th>
<th>Description</th>
<th>Source Registers</th>
<th>Destination Registers</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>and{l,w,b}</td>
<td>AND</td>
<td>srcIRM</td>
<td>destRM</td>
<td>dest = src &amp; dest</td>
</tr>
<tr>
<td>or{l,w,b}</td>
<td>OR</td>
<td>srcIRM</td>
<td>destRM</td>
<td>dest = src</td>
</tr>
<tr>
<td>xor{l,w,b}</td>
<td>XOR</td>
<td>srcIRM</td>
<td>destRM</td>
<td>dest = src ^ dest</td>
</tr>
<tr>
<td>not{l,w,b}</td>
<td>NOT</td>
<td>destRM</td>
<td></td>
<td>dest = ~dest</td>
</tr>
<tr>
<td>sal{l,w,b}</td>
<td>SHL (Same as sal)</td>
<td>srcIR</td>
<td>destRM</td>
<td>dest = dest &lt;&lt; src</td>
</tr>
<tr>
<td>sar{l,w,b}</td>
<td>SHR (Sign extend)</td>
<td>srcIR</td>
<td>destRM</td>
<td>dest = dest &gt;&gt;&gt; src</td>
</tr>
<tr>
<td>shr{l,w,b}</td>
<td>SHR (Zero fill)</td>
<td>srcIR</td>
<td>destRM</td>
<td>dest = dest &gt;&gt;&gt; src</td>
</tr>
</tbody>
</table>

[41]
Summary

Language levels

The basics of computer architecture
  • Enough to understand IA-32 assembly language

The basics of IA-32 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

Intel 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>Address</th>
<th>Value</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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</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>
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**

<table>
<thead>
<tr>
<th>Byte 0:</th>
<th>ff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 1:</td>
<td>77</td>
</tr>
<tr>
<td>Byte 2:</td>
<td>33</td>
</tr>
<tr>
<td>Byte 3:</td>
<td>00</td>
</tr>
</tbody>
</table>

**Output on a big-endian machine**

<table>
<thead>
<tr>
<th>Byte 0:</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 1:</td>
<td>33</td>
</tr>
<tr>
<td>Byte 2:</td>
<td>77</td>
</tr>
<tr>
<td>Byte 3:</td>
<td>ff</td>
</tr>
</tbody>
</table>
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?

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?

```
[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?

```assembly
.section ".data"
grade: .byte 'B'
...
.section ".text"
...
# Option 1
movl grade, %eax
subl $1, %eax
movl %eax, grade
...
# Option 2
subl $1, grade
```