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

Afterward

Application Program
Operating System
Hardware

language levels tour

service levels tour
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 <em>complete</em> pgms</td>
<td>Study <em>partial</em> pgms</td>
</tr>
<tr>
<td>Begin with <em>small</em> pgms;</td>
<td>Begin with <em>simple</em></td>
</tr>
<tr>
<td>proceed to <em>large</em> ones</td>
<td>constructs; proceed to</td>
</tr>
<tr>
<td></td>
<td><em>complex</em> ones</td>
</tr>
<tr>
<td>Emphasis on <em>writing</em> code</td>
<td>Emphasis on <em>reading</em> 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!!!

```assembly
loop:
  cmpl $1, %edx
  jle endloop
  addl $1, %ecx
  movl %edx, %eax
  addl %eax, %edx
  addl $1, %edx
  addl %eax, %edx
  jmp endloop
else:
  movl %edx, %eax
  addl %eax, %edx
  addl %eax, %edx
  addl $1, %edx
  jmp endif
endif:
  jmp loop
endloop:
  movl $0, %ecx
```
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

- Control Unit
- ALU
- Registers
- CPU
- RAM
- Data bus
RAM (Random Access Memory)

- Conceptually: large array of bytes

```
00000000
  ...
  TEXT
  RODATA
  DATA
  BSS
  HEAP
  ↓
  STACK
  FFFFFFFF
  ...
```

- Control Unit
- ALU
- Registers
- Data bus
- RAM
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>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>AH</td>
<td>AL</td>
<td></td>
<td></td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td></td>
<td>BH</td>
<td>BL</td>
<td></td>
<td></td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td></td>
<td>CH</td>
<td>CL</td>
<td></td>
<td></td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td></td>
<td>DH</td>
<td>DL</td>
<td></td>
<td></td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td></td>
<td>SI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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)
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)*
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
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
 src1  src2
operation  ALU  EFLAGS
 dest

RAM

Control Unit
ALU
Registers

CPU

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

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;

.section ".data"
.globl c
c: .byte 'a'
.globl s
s: .word 12
.globl i
i: .long 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"...;
...

Note:
.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 (\text{mov, add, sub, 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:

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

- **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:

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

- **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

```c
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

```assembly
.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
```
Performing Arithmetic: Byte Data

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

Note:

Comment

- `movb` instruction
- `subb` instruction
- `decb` instruction

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

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

Memory operands
- \textbf{5} \implies \text{load from (or store to) memory at address 5 (silly; seg fault)}
- \textbf{i} \implies \text{load from (or store to) memory at the address denoted by i}
- Can be source or destination operand (\textbf{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:
  • src => source; dest => destination
  • R => register; l => 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> (sign extend)</td>
</tr>
<tr>
<td><code>cwtd</code></td>
<td><code>reg[DX:AX] = reg[AX]</code> (sign extend)</td>
</tr>
<tr>
<td><code>cbtw</code></td>
<td><code>reg[AX] = reg[AL]</code> (sign extend)</td>
</tr>
</tbody>
</table>

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

Arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>add{l,w,b}</td>
<td>srcIRM, destRM</td>
<td>dest += src</td>
</tr>
<tr>
<td>sub{l,w,b}</td>
<td>srcIRM, destRM</td>
<td>dest -= src</td>
</tr>
<tr>
<td>inc{l,w,b}</td>
<td>destRM</td>
<td>dest++</td>
</tr>
<tr>
<td>dec{l,w,b}</td>
<td>destRM</td>
<td>dest--</td>
</tr>
<tr>
<td>neg{l,w,b}</td>
<td>destRM</td>
<td>dest = -dest</td>
</tr>
</tbody>
</table>
### Generalization: Signed Mult & Div

**Signed multiplication and division instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>imull srcRM</code></td>
<td>( \text{reg}[EDX:EAX] = \text{reg}[EAX] \times \text{src} )</td>
</tr>
<tr>
<td><code>imulw srcRM</code></td>
<td>( \text{reg}[DX:AX] = \text{reg}[AX] \times \text{src} )</td>
</tr>
<tr>
<td><code>imulb srcRM</code></td>
<td>( \text{reg}[AX] = \text{reg}[AL] \times \text{src} )</td>
</tr>
</tbody>
</table>
| `idivl srcRM` | \( \text{reg}[EAX] = \text{reg}[EDX:EAX] \div \text{src} \)  
\( \text{reg}[EDX] = \text{reg}[EDX:EAX] \mod \text{src} \) |
| `idivw srcRM` | \( \text{reg}[AX] = \text{reg}[DX:AX] \div \text{src} \)  
\( \text{reg}[DX] = \text{reg}[DX:AX] \mod \text{src} \) |
| `idivb srcRM` | \( \text{reg}[AL] = \text{reg}[AX] \div \text{src} \)  
\( \text{reg}[AH] = \text{reg}[AX] \mod \text{src} \) |

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>Equation</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
| `divl srcRM` | `reg[EAX] = reg[EDX:EAX]/src`  
          | `reg[EDX] = reg[EDX:EAX]%src` |
| `divw srcRM` | `reg[AX] = reg[DX:AX]/src`  
          | `reg[DX] = reg[DX:AX]%src` |
| `divb srcRM` | `reg[AL] = reg[AX]/src`  
          | `reg[AH] = reg[AX]%src` |

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>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and{l,w,b} srcIRM, destRM</code></td>
<td>dest = src &amp; dest</td>
<td><code>and{l,w,b} srcIRM, destRM</code> dest = src &amp; dest</td>
</tr>
<tr>
<td><code>or{l,w,b} srcIRM, destRM</code></td>
<td>dest = src</td>
<td><code>or{l,w,b} srcIRM, destRM</code> dest = src</td>
</tr>
<tr>
<td><code>xor{l,w,b} srcIRM, destRM</code></td>
<td>dest = src ^ dest</td>
<td><code>xor{l,w,b} srcIRM, destRM</code> dest = src ^ dest</td>
</tr>
<tr>
<td><code>not{l,w,b} destRM</code></td>
<td>dest = ~dest</td>
<td><code>not{l,w,b} destRM</code> dest = ~dest</td>
</tr>
<tr>
<td><code>sal{l,w,b} srcIR, destRM</code></td>
<td>dest = dest &lt;&lt; src</td>
<td><code>sal{l,w,b} srcIR, destRM</code> dest = dest &lt;&lt; src</td>
</tr>
<tr>
<td><code>sar{l,w,b} srcIR, destRM</code></td>
<td>dest = dest &gt;&gt; src (sign extend)</td>
<td><code>sar{l,w,b} srcIR, destRM</code> dest = dest &gt;&gt; src (sign extend)</td>
</tr>
<tr>
<td><code>shl{l,w,b} srcIR, destRM</code></td>
<td>(Same as sal)</td>
<td><code>shl{l,w,b} srcIR, destRM</code> (Same as sal)</td>
</tr>
<tr>
<td><code>shr{l,w,b} srcIR, destRM</code></td>
<td>dest = dest &gt;&gt; src (zero fill)</td>
<td><code>shr{l,w,b} srcIR, destRM</code> dest = dest &gt;&gt; src (zero fill)</td>
</tr>
</tbody>
</table>
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 = 0x00377ff;
    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: 33
- Byte 2: 77
- 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?

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 “I” 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
```