Assembly Language: Part 1

Context of this Lecture

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

Starting Now

Afterward

C Language
Assembly Language
Machine Language

Application Program
Operating System
Hardware

Von Neumann Architecture

Instructions are fetched from RAM
- (encoded as bits)
Control unit interprets instructions
- to shuffle data between registers and RAM
- to move data from registers through ALU (arithmetic+logic unit) where operations are performed

CPU
Control Unit
ALU
RAM
Registers
Data bus

Agenda

Language Levels
Instruction-Set Architecture (ISA)
Assembly Language: Performing Arithmetic
Assembly Language: Control-flow instructions

High-Level Languages

Characteristics
- Portable
  - To varying degrees
- Complex
  - One statement can do much work
- Structured
  - while (...) (...) if () ... else ...
- Human readable

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
- Unstructured
- 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
- Unstructured
- Human readable!!!
  (well, in the same sense that Hungarian is human readable, if you know Hungarian).

```
7
movl $0, %r10d
loop:
  cmpl $1, %r11d
  jle endloop
  addl %eax, %r11d
  je else
  movl %r11d, %eax
  andl $1, %eax
  jmp endif
else:
  sarl $1, %r11d
endif:
  jmp loop
endloop:
```

```
8
```

```
9
```

```
10
```

```
11
```

```
12
```
RAM

RAM (Random Access Memory)
Conceptually: large array of bytes
- Contains data
  (program variables, structs, arrays)
- and the program!

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

RAM (Random Access Memory)
Conceptually: large array of bytes
- Instructions are fetched from RAM
- Registers
  - Small amount of storage on the CPU
  - Much faster than RAM
  - Top of the storage hierarchy
    - Above RAM, disk, ...

Registers

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></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>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>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>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>
<tr>
<td>RDX</td>
<td></td>
<td></td>
<td></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>RSI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RBP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

- **R8**
  - `R8D`
  - `R8W`
  - `R8B`
- **R9**
  - `R9D`
  - `R9W`
  - `R9B`
- **R10**
  - `R10D`
  - `R10W`
  - `R10B`
- **R11**
  - `R11D`
  - `R11W`
  - `R11B`
- **R12**
  - `R12D`
  - `R12W`
  - `R12B`
- **R13**
  - `R13D`
  - `R13W`
  - `R13B`
- **R14**
  - `R14D`
  - `R14W`
  - `R14B`
- **R15**
  - `R15D`
  - `R15W`
  - `R15B`

#### Registers summary

16 general-purpose 64-bit pointer/long-integer registers, many with stupid names:
- `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`
- `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`, `r8d`, `r9d`, `r10d`, `r11d`, `r12d`, `r13d`, `r14d`, `r15d`

**RSP Register**

- **RSP (Stack Pointer) register**
  - Contains address of top (low address) of current function’s stack frame
  - Sometimes used as a “frame pointer” or “base pointer”

**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, ja
  - Sometimes used as a “frame pointer” or “base pointer”

**RIP 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**

2 special-purpose registers:
- **EFLAGS**
- **RIP**

If you’re operating on 32-bit “int” data, use these stupid names instead:
- `r8d`, `r9d`, `r10d`, `r11d`, `r12d`, `r13d`, `r14d`, `r15d`

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

Control Unit

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

CPU

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: Performing Arithmetic
Assembly Language: Control-flow instructions

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:

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

- **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

- **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
  - (to announce TEXT section)

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX  14</td>
<td>length  5</td>
</tr>
<tr>
<td>R10</td>
<td>width  2</td>
</tr>
<tr>
<td></td>
<td>perim  14</td>
</tr>
</tbody>
</table>

```
.name "bss"
length: .skip 4
width: .skip 4
perim: .skip 4

.name "text"

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

```
# Option 1
movb grade, %al
subb $1, %al
movb %al, grade
# Option 2
subb $1, grade
# Option 3
dech grade
```

What would happen if we use movl instead of movb?

Performing Arithmetic: Byte Data

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

```
.name "data"
grade: .byte 'B'
.byte 'A'
.byte 'D'
.byte 0

.name "text"

Registers Memory

<table>
<thead>
<tr>
<th>EAX A</th>
<th>grade AAD0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>movb instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>subb instruction</td>
</tr>
<tr>
<td>decb instruction</td>
</tr>
</tbody>
</table>

Note:
- Comment

```
# Option 1
movb grade, %al
subb $1, %al
movb %al, grade
# Option 2
subb $1, grade
# Option 3
dech grade
```

Operands

### Immediate operands
- $5 \Rightarrow \text{use the number 5 (i.e. the number that is available immediately within the instruction)}$
- $i \Rightarrow \text{use the address denoted by i (i.e. the address that is available immediately within the instruction)}$
- \text{Can be source operand; cannot be destination operand}

### Register operands
- %rax \Rightarrow \text{read from (or write to) register RAX}$
- \text{Can be source or destination operand}

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

Notation

### Instruction notation:
- q \Rightarrow \text{quad (8 bytes); l} \Rightarrow \text{long (4 bytes)}$
- w \Rightarrow \text{word (2 bytes); b} \Rightarrow \text{byte (1 byte)}$

### Operand notation:
- src \Rightarrow \text{source; dest} \Rightarrow \text{destination}$
- R \Rightarrow \text{register}; I \Rightarrow \text{immediate; M} \Rightarrow \text{memory}$
Generalization: Data Transfer

Data transfer instructions

- `mov(q,l,w,b) srcRM, 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)
- `cwtl`: reg[EAX] = reg[AX] (sign extend)
- `cbtw`: reg[AX] = reg[AL] (sign extend)


`mov` is used often; others less so.

Generalization: Arithmetic

Arithmetic instructions

- `add(q,l,w,b) srcIRM, destRM`: dest += src
- `sub(q,l,w,b) srcIRM, destRM`: dest -= src
- `inc(q,l,w,b) destRM`: dest++
- `dec(q,l,w,b) destRM`: dest--
- `neg(q,l,w,b) destRM`: dest = -dest
- `ashl`: reg[RDX:RAX] = reg[RAX]*src
- `divl`: reg[EAX] = reg[EDX:EAX]/src
- `divb`: reg[AL] = reg[AX]/src

Q: Is this adding signed numbers or unsigned? A: Yes! [remember properties of 2's complement]

Q: Is this adding signed numbers or unsigned? A: Yes! [remember properties of 2's complement]

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

Generalization: Bit Manipulation

Bitwise instructions

- `and(q,l,w,b) srcIRM, destRM`: dest = src & dest
- `or(q,l,w,b) srcIRM, destRM`: dest = src | dest
- `xor(q,l,w,b) srcIRM, destRM`: dest = src ^ dest
- `not(q,l,w,b) destRM`: dest = ~dest
- `sal(q,l,w,b) srcIR, destRM`: dest = dest << src
- `salh(q,l,w,b) srcIR, destRM`: dest = dest >> src (zero fill)
- `shl(q,l,w,b) srcIR, destRM`: (Same as sal)
- `shr(q,l,w,b) srcIR, destRM`: dest = dest >> src (sign extend)
- `sbtrl(q,l,w,b) srcIR, destRM`: dest = dest >> src (sign extend)
- `sarl(q,l,w,b) srcIR, destRM`: dest = dest >> src (zero fill)

Signed (arithmetic right shift)
- 44 / 2: 000101100 = 11
- -44 / 2: 111010100 = -11

 Unsigned (logical right shift)
- 44 / 2: 000101100 = 11
- 468 / 2: 111010100 = 117

Translation: C to x86-64

```assembly
movl %r11d, %eax
andl $1, %eax
je loop
addl $1, %r10d
movl %r10d, %eax
addl %eax, %r10d
addl $1, %r10d
jmp endif
movl $0, %r10d
loop:
cmpl $1, %r10d
jle endloop
addl %r10d, %eax
addl %eax, %r10d
addl $1, %r10d
jmp endif
jmp loop
endif:
addl %eax, %r10d
```

Agenda

Language Levels

Architecture

Assembly Language: Performing Arithmetic

Assembly Language: Control-flow instructions
Control Flow with Signed Integers

Comparing (signed or unsigned) integers

Sets condition-code bits in the EFLAGS register
• Beware: operands are in counterintuitive order
• Beware: many other instructions set condition-code bits
• Conditional jump should immediately follow \texttt{cmp}


<table>
<thead>
<tr>
<th>movl $0, %r10d</th>
<th>movl $0, %r10d</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop:</td>
<td>loop:</td>
</tr>
<tr>
<td>cmpl $1, %r11d</td>
<td>cmpl $1, %r11d</td>
</tr>
<tr>
<td>jle endloop</td>
<td>jle endloop</td>
</tr>
<tr>
<td>addl $1, %r10d</td>
<td>addl $1, %r10d</td>
</tr>
<tr>
<td>movl %r11d, %eax</td>
<td>movl %r11d, %eax</td>
</tr>
<tr>
<td>andl $1, %eax</td>
<td>andl $1, %eax</td>
</tr>
<tr>
<td>je else</td>
<td>je else</td>
</tr>
<tr>
<td>movl %r11d, %eax</td>
<td>movl %r11d, %eax</td>
</tr>
<tr>
<td>addl %eax, %r11d</td>
<td>addl %eax, %r11d</td>
</tr>
<tr>
<td>addl %eax, %r11d</td>
<td>addl %eax, %r11d</td>
</tr>
<tr>
<td>addl $1, %r11d</td>
<td>addl $1, %r11d</td>
</tr>
<tr>
<td>jmp endif</td>
<td>jmp endif</td>
</tr>
<tr>
<td>sarl $1, %r11d</td>
<td>sarl $1, %r11d</td>
</tr>
<tr>
<td>endif: jmp loop</td>
<td>endif: jmp loop</td>
</tr>
<tr>
<td>endloop:</td>
<td>endloop:</td>
</tr>
</tbody>
</table>

Unconditional jump

\texttt{jmp X} Jump to address X

Conditional jumps after comparing signed integers

\texttt{je X} Jump to X if equal
\texttt{jne X} Jump to X if not equal
\texttt{jl X} Jump to X if less
\texttt{jle X} Jump to X if less or equal
\texttt{jg X} Jump to X if greater
\texttt{jge X} Jump to X if greater or equal

• Examine condition-code bits in EFLAGS register

Summary

Language levels
• The basics of computer architecture
  • Enough to understand x86-64 assembly language
The basics of x86-64 assembly language
• Registers
• Arithmetic
• Control flow
To learn more
• Study more assembly language examples
  • Chapter 3 of Bryant and O’Hallaron book
• Study compiler-generated assembly language code
  • \texttt{gcc} \texttt{217 -S} \texttt{somefile.c}