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

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

RAM
Control Unit
ALU
Registers
Data bus

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

High-Level Languages
Characteristics
- Portable
- Complex
- One statement can do much work
- Structured
- 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).

```
movl $0, %r10d
```

```
loop:
    cmp $1, %r10d
    jle endloop
    addl $1, %r10d
    movl %r11d, %eax
    addl %eax, %r10d
    addl %eax, %r11d
    addl $1, %r11d
    jmp loop
endloop:
```

### Translation: C to x86-64

```
count = 0;
while (n>1) {
    count++;
    if (n&1)
        n = n*3+1;
    else
        n = n/2;
}
```

### Why Learn Assembly Language?

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 widely used
- CourseLab computers are x86-64 computers
- Program natively on CourseLab instead of using an emulator

**Cons**
- X86-64 assembly language is big and ugly
- There are many instructions
- Instructions differ widely

### Agenda

- Language Levels
- Architecture
- Assembly Language: Defining global data
- Assembly Language: Performing Arithmetic
- Assembly Language: Control-flow instructions
**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 (x86-64 architecture)**

General purpose registers:

- RAX
- RBX
- RCX
- RDX

---

**Registers (x86-64 architecture)**

General purpose registers (cont.):

- RSI
- RDI
- RSP

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

General purpose registers (cont.):

<table>
<thead>
<tr>
<th>Register</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

If you're operating on 32-bit "int" data, use these names instead:

<table>
<thead>
<tr>
<th>Register</th>
<th>RAX</th>
<th>RBX</th>
<th>RCX</th>
<th>RDX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</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

EFLAGS Register

Special-purpose register...

EFLAGS (Flag) register
- Contains CC (Condition Code) bits
- Affected by compare (cmp) instruction
- Used by conditional jump instructions

Registers summary

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

If you're operating on 32-bit "int" data, use these names instead:

eax, ebx, ecx, edx, esi, edi, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d

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

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

If you're operating on 32-bit "int" data, use these names instead:

eax, ebx, ecx, edx, esi, edi, 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"
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

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

CPU

- 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
Assembly Language: Control-flow instructions
Defining Data: DATA Section 1

\[
\begin{align*}
\text{static char } c &= 'a' ; \\
\text{static short } s &= 12; \\
\text{static int } i &= 345; \\
\text{static long } l &= 6789;
\end{align*}
\]

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

\[
\begin{align*}
\text{char } c &= 'a' ; \\
\text{short } s &= 12; \\
\text{int } i &= 345; \\
\text{long } l &= 6789;
\end{align*}
\]

Note:
- `.section` instruction (to announce DATA section)
- `.globl` instruction

Defining Data: BSS Section

\[
\begin{align*}
\text{static char } c; \\
\text{static short } s; \\
\text{static int } i; \\
\text{static long } l;
\end{align*}
\]

Note:
- `.section` instruction (to announce BSS section)
- `.skip` instruction

Defining Data: RODATA Section

\[
\begin{align*}
\text{\textquotedbl} \text{hello\textbackslash n}\text{\textquotedbl};
\end{align*}
\]

Note:
- `.section` instruction (to announce RODATA section)
- `.string` instruction

Agenda

Language Levels
Architecture
Assembly Language: Defining global data
Assembly Language: Performing Arithmetic
Assembly Language: Control-flow instructions

Instruction Format

Many instructions have this format:

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

- `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:

```
name{b,w,l,q} src, dest
```

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

- **dest**: destination operand
  - The destination of data
  - Can be
    - Register operand: `%rax`, `%ebx`, etc.
    - Immediate operand
    - Cannot be
      - Memory operand: 5 (legally but silly), someLabel

Performing Arithmetic: Long Data

```
static int length;
static int width;
static int perim;
```

```
perim = (length + width) * 2;
```

```
.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
```

**Note:**
- `movl` instruction
- `addl` instruction
- `sall` instruction
- `movl` instruction

```
Registers
EAX 14
R10

Memory
length 5
width 2
perim 14
```

Performing Arithmetic: Byte Data

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

```
grade --;
```

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

```
section .text
...
```

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

```
# Option 2
subb $1, grade
...
```

```
# Option 3
decb grade
```

**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

**Operands**

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

- **Memory operands**
  - `$5$ ⇒ load from (or store to) memory at address 5 (silly; seg fault*)
  - `$i$ ⇒ 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: Data Transfer

Data transfer instructions

- **mov**(q,l,w,b) srcRM, destRM
  - dest = src
- **movb**(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)
- **cltd**(reg[EDX:EAX] = reg[EAX] (sign extend)
- **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

A: Yes! (Remember properties of 2's complement)

3 0011
4 + 1100
----
-1 1111

B: 12 + 1100
---
15 1111

Signed 2’s complement

unsigned

Translation: C to x86-64

```c
count = 0;
while (n>1) {
    if (n%2)
        n = n*3+1;
    else
        n = n/2;
}
```

```assembly
movl $0, %e1d
```

Loop:

```assembly
cmp %e1d, %e0d
je endloop
addl $1, %e1d
movl %e1d, %e0d
addl $1, %e1d
je else
```

Else:

```assembly
jmp endif
```

Endif:

```assembly
addl $1, %e1d
```

Endloop:

```assembly
jmp loop
```

Agenda

Language Levels
Architecture
Assembly Language: Defining global data
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 cmp

```
cmp {q,l,w,b} srcIRM, destRM
```

Conditional jumps after comparing signed integers

- Examine condition-code bits in EFLAGS register

```
j{e,n,l,g} X
```


```
movl %r11d, %eax
andl $1, %eax
je else
jmp endif
else:
sarl $1, %r11d
movl %r11d, %eax
addl %eax, %r11d
addl %eax, %r11d
addl $1, %r11d
addl $1, %r10d
```

Label stands for an address

```
movl $0, %r11d
loop: cmpl $1, %r11d
jle endloop
jmp loop
```

Translation: C to x86-64

```
count = 0;
while (n>1) { count++; if (n&1) n = n*3+1; else n = n/2; }
```

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
- 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”

```
  1000 1001 1002 1003
  1000 1001 1002 1003
```

Some other systems use big endian
- Most significant byte of multi-byte entity is stored at lowest memory address
- “Big end goes first”

```
  1000 1001 1002 1003
  1000 1001 1002 1003
```

Byte Order Example 1

```
#include <stdio.h>

int main(void)
{
  unsigned int i = 0x003377ff;
  unsigned char *p;
  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

```
.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 "b" instructions to manipulate a four-byte memory area
- x86-64 is little endian, so what will be the value of grade if x86-64 were big endian?

Byte Order Example 3

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

.section .text
...

# Option 1
movl grade, %eax
subl $1, %eax
movl %eax, grade
...

# Option 2
subl $1, grade
```

Note:
- Flawed code; uses "l" instructions to manipulate a one-byte memory area
- What would happen?