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

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;
}
## 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).

### Assembly Code

```
movl $0, %r10d
loop:
  cmpl $1, %r10d
  jle endloop
  addl %eax, %r10d
  jmp loop
endloop:
```

## Computer: CPU + RAM

- **Control Unit**
- **ALU**
- **Data bus**
- **RAM**
- **Registers**

## Translation: C to x86-64

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

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

## 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: Performing Arithmetic
- Assembly Language: Control-flow instructions
**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

---

**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>r31</th>
<th>r15</th>
<th>r7</th>
<th>r0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>RX</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RBX</td>
<td>BX</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>RCX</td>
<td>CX</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>RDX</td>
<td>DX</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

---

**Registers (x86-64 architecture) (cont.)**

General purpose registers (cont.):

<table>
<thead>
<tr>
<th>r31</th>
<th>r15</th>
<th>r7</th>
<th>r0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSI</td>
<td>SI</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>RDI</td>
<td>DI</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>RBP</td>
<td>BP</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>RSP</td>
<td>SP</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

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

General purpose registers (cont.):

<table>
<thead>
<tr>
<th>Register</th>
<th>R8</th>
<th>R8D</th>
<th>R8W</th>
<th>R8B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>43</td>
<td>31</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>R9</td>
<td>39</td>
<td>28</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>R10</td>
<td>160</td>
<td>128</td>
<td>63</td>
<td>31</td>
</tr>
<tr>
<td>R11</td>
<td>172</td>
<td>139</td>
<td>63</td>
<td>31</td>
</tr>
<tr>
<td>R12</td>
<td>185</td>
<td>152</td>
<td>63</td>
<td>31</td>
</tr>
<tr>
<td>R13</td>
<td>222</td>
<td>190</td>
<td>63</td>
<td>31</td>
</tr>
<tr>
<td>R14</td>
<td>235</td>
<td>202</td>
<td>63</td>
<td>31</td>
</tr>
<tr>
<td>R15</td>
<td>248</td>
<td>215</td>
<td>63</td>
<td>31</td>
</tr>
</tbody>
</table>

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`

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`

Sometimes used as a "frame pointer" or "base pointer".

RSP Register

RSP (Stack Pointer) register
- Contains address of top (low address) of current function’s stack frame.

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`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`

Sometimes used as a "frame pointer" or "base 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.

EFLAGS, RIP
**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**

**ALU (Arithmetic Logic Unit)**
- Performs arithmetic and logic operations

**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:

```
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

- **dest**: destination operand
  - The destination of data
  - Can be
    - Register operand: %rax, %ebx, etc.
    - Memory operand: 5 (legal but silly), someLabel
    - Immediate operand

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"
...
```

Note:
- movl instruction
- addl instruction
- sall instruction
- Register operand
- Immediate operand
- Memory operand

Operator notation:
- \( * \text{src} \) \( \Rightarrow \text{dest} \)
- \( \text{src} \) \( \Rightarrow \text{dest} \)
- \( \text{src} \) \( \Rightarrow \text{dest} \)
- \( \text{src} \) \( \Rightarrow \text{dest} \)

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 (\textit{but not both})
- There’s more to memory operands; see next lecture

Notation

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

Operand notation:
- src \( \Rightarrow \) source;
- dest \( \Rightarrow \) destination
- R \( \Rightarrow \) register;
- I \( \Rightarrow \) immediate;
- M \( \Rightarrow \) memory

* if you’re lucky

Performing Arithmetic: Byte Data

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

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

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

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

```
# Option 3
dech grade
```

Note:
- Comment
- movb instruction
- subb instruction
- dech instruction

What would happen if we use movl instead of movb?
Generalization: Data Transfer

Data transfer instructions

- `mov(q,l,w,b) srcIRM, destRM` dest = src
- `movsw(q,l,w) 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)
- `ctbw` 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

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

Signed 2's complement

| 3      | 0011B |
| -3     | 1111B |

Unsigned

| 3      | 0011B |
| 3      | 1111B |

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
- `sar(q,l,w,b) srcIR, destRM` dest = dest >> src (sign extend)
- `shr(q,l,w,b) srcIR, destRM` dest = dest >> src (zero fill)

Multiplication & Division

Signed

| 44 / 22 | 000101100B |
| -44 / 22 | 111010100B |

Unsigned

| 44 / 22 | 000101100B |
| 468 / 22 | 111010100B |

Translation: C to x86-64

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

```
count--%16
n--%16
```

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

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

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 cmp

Unconditional jump
jmp X Jump to address X

Conditional jumps after comparing signed integers
je X Jump to X if equal
jne X Jump to X if not equal
jl X Jump to X if less
jle X Jump to X if less or equal
jg X Jump to X if greater
jge X Jump to X if greater or equal


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

Translation: C to x86-64

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