Assembly Language: Part 1
Context of this Lecture

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

Starting Now

- C Language
  - Assembly Language
    - Machine Language

Afterward

- Application Program
  - Operating System
    - Hardware

language levels tour
service levels tour
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
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

```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
• 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:
cmpl $1, %r11d
jle endloop
addl $1, %r10d
movl %r11d, %eax
andl $1, %eax
je else

cmpl %r11d, %eax
addl %eax, %r11d
addl $1, %r11d
jmp endif
else:
sarl $1, %r11d
endif:
jmp loop
endloop:
```
Computer: CPU + RAM

```
loop:
    cmpl $1, %r11d
    jle endloop
    addl $1, %r10d
    movl %r1ld, %eax
    andl $1, %eax
    je else
    movl %r1ld, %eax
    addl %eax, %r1ld
    addl %eax, %r1ld
else:
    jmp endif
endif:
    jmp loop
endloop:
```

movl $0, %r10d
sarl $1, %r11d
movl %r11d, %eax
addl %eax, %r11d
addl %eax, %r11d
addl $1, %r11d
addl $1, %r10d
```
Translation: C to x86-64

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

```assembly
movl $0, %r10d

loop:
    cmpl $1, %r11d
    jle endloop

    addl $1, %r10d
    movl %r11d, %eax
    andl $1, %eax
    je else

    movl %r11d, %eax
    addl %eax, %r11d
    addl $1, %r11d

else:
    jmp endif

endif:
    sarl $1, %r11d

endloop:
    jmp loop
```

count ↔ r10d
n ↔ r11d
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

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:

<table>
<thead>
<tr>
<th>Register</th>
<th>EAX</th>
<th>AX</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>EBX</th>
<th>BX</th>
<th>BL</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>ECX</th>
<th>CX</th>
<th>CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>EDX</th>
<th>DX</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Registers (x86-64 architecture)

General purpose registers (cont.):

<table>
<thead>
<tr>
<th>Register</th>
<th>Lower 32 bits</th>
<th>Upper 32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSI</td>
<td>ESI</td>
<td>SI</td>
</tr>
<tr>
<td>RDI</td>
<td>EDI</td>
<td>DI</td>
</tr>
<tr>
<td>RBP</td>
<td>EBP</td>
<td>BP</td>
</tr>
<tr>
<td>RSP</td>
<td>ESP</td>
<td>SP</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>Bits</th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
<td></td>
<td></td>
<td>R8D</td>
<td>R8W</td>
<td>R8B</td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td></td>
<td>R9D</td>
<td>R9W</td>
<td>R9B</td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td></td>
<td></td>
<td>R10D</td>
<td>R10W</td>
<td>R10B</td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td></td>
<td></td>
<td>R11D</td>
<td>R11W</td>
<td>R11B</td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td></td>
<td></td>
<td>R12D</td>
<td>R12W</td>
<td>R12B</td>
<td></td>
</tr>
<tr>
<td>R13</td>
<td></td>
<td></td>
<td>R13D</td>
<td>R13W</td>
<td>R13B</td>
<td></td>
</tr>
<tr>
<td>R14</td>
<td></td>
<td></td>
<td>R14D</td>
<td>R14W</td>
<td>R14B</td>
<td></td>
</tr>
<tr>
<td>R15</td>
<td></td>
<td></td>
<td>R15D</td>
<td>R15W</td>
<td>R15B</td>
<td></td>
</tr>
</tbody>
</table>
16 general-purpose 64-bit pointer/long-integer registers, many with stupid names:

rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15

sometimes used as a “frame pointer” or “base pointer”

“stack 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
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

(See Assembly Language: Function Calls lecture)
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, jb**

(See *Assembly Language: Part 2* lecture)
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, rbp, rsp, r8, r9, r10, r11, r12, r13, r14, r15

- Sometimes used as a “frame pointer” or “base pointer”
- "stack pointer"

If you’re operating on 32-bit “int” data, use these stupid names instead:
eax, ebx, ecx, edx, esi, edi, ebp, rbp, 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"
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

```
src1  src2
operation  ALU
dest  EFLAGS
```

- CPU
- Control Unit
- Registers
- Data bus
- RAM
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: Performing Arithmetic
Assembly Language: Control-flow instructions
Instruction Format

Many instructions have this format:

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

- **name**: name of the instruction (\text{mov}, \text{add}, \text{sub}, \text{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
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 (legal but silly), someLabel
    - **Immediate operand**: $5, $someLabel
Many instructions have this format:

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

- **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

```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

```
.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
```

<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>
Performing Arithmetic: Byte Data

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

### Registers

- **EAX**: A

### Memory

- **grade**: A A D 0

### Note:

- **Comment**: movb instruction
- **subb** instruction
- **decb** instruction

What would happen if we use **movl** instead of **movb**?

```assembly
.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
decb grade
```
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
• %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 (but not both)
• There’s more to memory operands; see next lecture

*if you’re lucky
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
Generalization: Data Transfer

Data transfer instructions

\[
\begin{align*}
\text{mov}\{q,l,w,b\} \ srcRM, \ destRM & \quad \text{dest} = \ src \\
\text{movsb}\{q,l,w\} \ srcRM, \ destR & \quad \text{dest} = \ src \ (\text{sign extend}) \\
\text{movsw}\{q,l\} \ srcRM, \ destR & \quad \text{dest} = \ src \ (\text{sign extend}) \\
\text{movslq} \ srcRM, \ destR & \quad \text{dest} = \ src \ (\text{sign extend}) \\
\text{movzb}\{q,l,w\} \ srcRM, \ destR & \quad \text{dest} = \ src \ (\text{zero fill}) \\
\text{movzw}\{q,l\} \ srcRM, \ destR & \quad \text{dest} = \ src \ (\text{zero fill}) \\
\text{movzlq} \ srcRM, \ destR & \quad \text{dest} = \ src \ (\text{zero fill}) \\
\text{cqto} & \quad \text{reg}[RDX:RAX] = \text{reg}[RAX] \ (\text{sign extend}) \\
\text{cltd} & \quad \text{reg}[EDX:EAX] = \text{reg}[EAX] \ (\text{sign extend}) \\
\text{cwtl} & \quad \text{reg}[EAX] = \text{reg}[AX] \ (\text{sign extend}) \\
\text{cbtw} & \quad \text{reg}[AX] = \text{reg}[AL] \ (\text{sign extend})
\end{align*}
\]

\textbf{mov} is used often; others less so
Generalization: Arithmetic

Arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add{q,l,w,b}</code></td>
<td><code>srcIRM, destRM</code></td>
</tr>
<tr>
<td><code>sub{q,l,w,b}</code></td>
<td><code>srcIRM, destRM</code></td>
</tr>
<tr>
<td><code>inc{q,l,w,b}</code></td>
<td><code>destRM</code></td>
</tr>
<tr>
<td><code>dec{q,l,w,b}</code></td>
<td><code>destRM</code></td>
</tr>
<tr>
<td><code>neg{q,l,w,b}</code></td>
<td><code>destRM</code></td>
</tr>
</tbody>
</table>

Q: Is this adding signed numbers or unsigned?

A: Yes! [remember properties of 2’s complement]

<table>
<thead>
<tr>
<th>Signed 2’s complement</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0011&lt;sub&gt;B&lt;/sub&gt;</td>
</tr>
<tr>
<td>+ -4</td>
<td>+ 1100&lt;sub&gt;B&lt;/sub&gt;</td>
</tr>
<tr>
<td>--</td>
<td>----</td>
</tr>
<tr>
<td>-1</td>
<td>1111&lt;sub&gt;B&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

<p>| 3                     | 0011&lt;sub&gt;B&lt;/sub&gt; |</p>
<table>
<thead>
<tr>
<th>+ 12</th>
<th>+ 1100&lt;sub&gt;B&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>1111&lt;sub&gt;B&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
Generalization: Bit Manipulation

Bitwise instructions

- **and**\{q,l,w,b\} srcIRM, destRM: $\text{dest} = \text{src} \& \text{dest}$
- **or**\{q,l,w,b\} srcIRM, destRM: $\text{dest} = \text{src} \mid \text{dest}$
- **xor**\{q,l,w,b\} srcIRM, destRM: $\text{dest} = \text{src} \oplus \text{dest}$
- **not**\{q,l,w,b\} destRM: $\text{dest} = \overline{\text{dest}}$
- **sal**\{q,l,w,b\} srcIR, destRM: $\text{dest} = \text{dest} \ll \text{src}$
- **sar**\{q,l,w,b\} srcIR, destRM: $\text{dest} = \text{dest} \gg \text{src}$ (sign extend)
- **shl**\{q,l,w,b\} srcIR, destRM: (Same as sal)
- **shr**\{q,l,w,b\} srcIR, destRM: $\text{dest} = \text{dest} \gg \text{src}$ (zero fill)

**signed (arithmetic right shift):**
- $44 / 2^2 = 11$
- $-44 / 2^2 = -11$

**unsigned (logical right shift):**
- $44 / 2^2 = 11$
- $468 / 2^2 = 117$
### Multiplication & Division

#### Signed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>imulq srcRM</code></td>
<td><code>reg[RDX:RAX] = reg[RAX]*src</code></td>
</tr>
<tr>
<td><code>imull srcRM</code></td>
<td><code>reg[EDX:EAX] = reg[EAX]*src</code></td>
</tr>
<tr>
<td><code>imulw srcRM</code></td>
<td><code>reg[DX:AX] = reg[AX]*src</code></td>
</tr>
<tr>
<td><code>imulb srcRM</code></td>
<td><code>reg[AX] = reg[AL]*src</code></td>
</tr>
<tr>
<td><code>idivq srcRM</code></td>
<td><code>reg[RAX] = reg[RDX:RAX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[RDX] = reg[RDX:RAX]%src</code></td>
</tr>
<tr>
<td><code>idivl srcRM</code></td>
<td><code>reg[EAX] = reg[EDX:EAX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[EDX] = reg[EDX:EAX]%src</code></td>
</tr>
<tr>
<td><code>idivw srcRM</code></td>
<td><code>reg[AX] = reg[DX:AX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[DX] = reg[DX:AX]%src</code></td>
</tr>
<tr>
<td><code>idivb srcRM</code></td>
<td><code>reg[AL] = reg[AX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[AH] = reg[AX]%src</code></td>
</tr>
</tbody>
</table>

#### Unsigned

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mulq srcRM</code></td>
<td><code>reg[RDX:RAX] = reg[RAX]*src</code></td>
</tr>
<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>
<tr>
<td><code>divq srcRM</code></td>
<td><code>reg[RAX] = reg[RDX:RAX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[RDX] = reg[RDX:RAX]%src</code></td>
</tr>
<tr>
<td><code>divl srcRM</code></td>
<td><code>reg[EAX] = reg[EDX:EAX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[EDX] = reg[EDX:EAX]%src</code></td>
</tr>
<tr>
<td><code>divw srcRM</code></td>
<td><code>reg[AX] = reg[DX:AX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[DX] = reg[DX:AX]%src</code></td>
</tr>
<tr>
<td><code>divb srcRM</code></td>
<td><code>reg[AL] = reg[AX]/src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[AH] = reg[AX]%src</code></td>
</tr>
</tbody>
</table>

See Bryant & O’Hallaron book for description of signed vs. unsigned multiplication and division.
count = 0;
while (n>1) {
    count++;
    if (n&1)
        n = n*3+1;
    else
        n = n/2;
}

movl $0, %r10d

loop:
cmpl $1, %r11d
jle endloop
addl $1, %r10d
movl %r11d, %eax
addl $1, %eax
je else
movl %r11d, %eax
addl %eax, %r11d
addl %eax, %r11d
addl $1, %r11d

else:
    jmp endif
    sarl $1, %r11d
endif:
    jmp loop
endloop:
Agenda

Language Levels
Architecture
Assembly Language: Performing Arithmetic
Assembly Language: Control-flow instructions
Control Flow with Signed Integers

Comparing (signed or unsigned) integers

\[
\text{cmp\{q,l,w,b\} srcIRM, destRM} \quad \text{Compare dest with src}
\]

- 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 \textit{immediately} follow \texttt{cmp}
Control Flow with Signed Integers

Unconditional jump

\[
\text{jmp } X \quad \text{Jump to address } X
\]

Conditional jumps after comparing signed integers

\[
\begin{align*}
\text{je } X & \quad \text{Jump to } X \text{ if equal} \\
\text{jne } X & \quad \text{Jump to } X \text{ if not equal} \\
\text{jl } X & \quad \text{Jump to } X \text{ if less} \\
\text{jle } X & \quad \text{Jump to } X \text{ if less or equal} \\
\text{jg } X & \quad \text{Jump to } X \text{ if greater} \\
\text{jge } X & \quad \text{Jump to } X \text{ if greater or equal}
\end{align*}
\]

- Examine condition-code bits in EFLAGS register
Assembly lang.  

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

Machine lang.  

<table>
<thead>
<tr>
<th>address</th>
<th>contents (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>41ba00000000</td>
</tr>
<tr>
<td>1006</td>
<td>4183fb01</td>
</tr>
<tr>
<td>100a</td>
<td>7e25</td>
</tr>
<tr>
<td>100c</td>
<td>4183c201</td>
</tr>
<tr>
<td>1010</td>
<td>4489d8</td>
</tr>
<tr>
<td>1013</td>
<td>83242500000000001</td>
</tr>
<tr>
<td>101b</td>
<td>740f</td>
</tr>
<tr>
<td>101d</td>
<td>4489d8</td>
</tr>
<tr>
<td>1020</td>
<td>4101c3</td>
</tr>
<tr>
<td>1023</td>
<td>4101c3</td>
</tr>
<tr>
<td>1026</td>
<td>4183c301</td>
</tr>
<tr>
<td>102a</td>
<td>eb03</td>
</tr>
<tr>
<td>102c</td>
<td>41d1fb</td>
</tr>
<tr>
<td>102f</td>
<td>83c301</td>
</tr>
<tr>
<td>1031</td>
<td></td>
</tr>
</tbody>
</table>

$25 = 2f−0a$ (hex)
Label \textit{stands for an address}

\begin{align*}
\text{movl} & \quad 0, %r10d \\
\text{loop:} & \\
\text{cmp} & \quad 1, %r11d \\
\text{jle} & \quad \text{endloop} \\
\text{addl} & \quad 1, %r10d \\
\text{movl} & \quad %r11d, %eax \\
\text{andl} & \quad 1, %eax \\
\text{je} & \quad \text{else} \\
\text{movl} & \quad %r11d, %eax \\
\text{addl} & \quad %eax, %r11d \\
\text{addl} & \quad %eax, %r11d \\
\text{addl} & \quad 1, %r11d \\
\text{jmp} & \quad \text{endif} \\
\text{else:} & \\
\text{sarl} & \quad 1, %r11d \\
\text{endif:} & \\
\text{jmp} & \quad \text{loop} \\
\text{endloop:} &
\end{align*}

\begin{align*}
\text{address: contents (in hex)} \\
1000: & \quad 41ba00000000 \\
1006: & \quad 4183fb01 \\
100a: & \quad \text{7e25} \quad 25 = 31−0c \text{ (hex)} \\
100c: & \quad 4183c201 \\
1010: & \quad 4489d8 \\
1013: & \quad 8324250000000001 \\
101b: & \quad 740f \\
101d: & \quad 4489d8 \\
1020: & \quad 4101c3 \\
1023: & \quad 4101c3 \\
1026: & \quad 4183c301 \\
102a: & \quad \text{eb03} \\
102c: & \quad 41d1fb \\
102f: & \quad 83c301 \\
1031: &
\end{align*}
Translation: C to x86-64

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

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