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

language levels tour

Afterward

Application Program

Operating System

Hardware

service levels tour
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: Defining global data
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

```cpp
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

```
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
9222 9120 1121 A120 1121 A121 7211 0000
0000 0001 0002 0003 0004 0005 0006 0007
0008 0009 000A 000B 000C 000D 000E 000F
0000 0000 0000 FE10 FACE CAFE ACED CEDE
0000 0000 0000 0000 0000 0000 0000 0000
1234 5678 9ABC DEF0 0000 0000 F00D 0000
0000 0000 EEEE 1111 EEEE 1111 0000 0000
B1B2 F1F5 0000 0000 0000 0000 0000 0000
```
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
movl $0, %r10d
lop:
    cmp1 $1, %r11d
    jle endloop
    add1 $1, %r10d
    mov1 %r11d, %eak
    add1 $1, %eak
    je else
else:
    mov1 %r11d, %eak
    add1 %eak, %r11d
    add1 %eak, %r11d
    add1 %r11d
    jmp endif
endif:
    sar1 $1, %r11d
endloop:
    jmp loop
endloop:
```
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:
Translation: C to x86-64

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

count → r10d
n → r11d
```

```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
    jmp   endif
else:
    jmp   endif
endif:
    sarl  $1, %r11d
endloop:
```
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 (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>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>EAX</td>
<td>AX</td>
<td>AL</td>
<td></td>
</tr>
<tr>
<td>RBX</td>
<td>EBX</td>
<td>BX</td>
<td>BL</td>
<td></td>
</tr>
<tr>
<td>RCX</td>
<td>ECX</td>
<td>CX</td>
<td>CL</td>
<td></td>
</tr>
<tr>
<td>RDX</td>
<td>EDX</td>
<td>DX</td>
<td>DL</td>
<td></td>
</tr>
</tbody>
</table>
Registers (x86-64 architecture)

General purpose registers (cont.):

<table>
<thead>
<tr>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSI</td>
<td>ESI</td>
<td>SI</td>
<td>SIL</td>
<td></td>
</tr>
<tr>
<td>RDI</td>
<td>EDI</td>
<td>DI</td>
<td>DIL</td>
<td></td>
</tr>
<tr>
<td>RBP</td>
<td>EBP</td>
<td>BP</td>
<td>BPL</td>
<td></td>
</tr>
<tr>
<td>RSP</td>
<td>ESP</td>
<td>SP</td>
<td>SPL</td>
<td></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>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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Registers summary

16 general-purpose 64-bit pointer/long-integer registers:

rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, 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, ebp, rsp, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d

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

“stack pointer”

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)*
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

![Diagram of RIP Register](image)
Registers summary

16 general-purpose 64-bit pointer/long-integer registers:

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 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

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
ALU (Arithmetic Logic Unit)

- Performs arithmetic and logic operations

```
src1 \downarrow \text{operation} \downarrow \text{dest} \downarrow \text{src2}
```

- ALU
- EFLAGS
- RAM
- Data bus
- Registers
- Control Unit
- CPU
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

![](image_url)
Agenda

Language Levels
Architecture

Assembly Language: Defining global data
Assembly Language: Performing Arithmetic
Assembly Language: Control-flow instructions
RAM (Random Access Memory)

Control Unit

Registers

CPU

Data bus

RAM

0000000000000000

... TEXT

RODATA

DATA

BSS

HEAP

STACK

...
Defining Data: DATA Section 1

```
static char c = 'a';
static short s = 12;
static int i = 345;
static long l = 6789;
```

```
.section " .data"

  c:
    .byte 'a'

  s:
    .word 12

  i:
    .long 345

  l:
    .quad 6789
```

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

```c
char c = 'a';
short s = 12;
int i = 345;
long l = 6789;
```

```
.globl c
.c: .byte 'a'
.globl s
.s: .word 12
.globl i
.i: .long 345
.globl l
.l: .quad 6789
```

Note:
Can place label on same line as next instruction
.globl instruction
static char c;
static short s;
static int i;
static long l;

.section ".bss"

.c:
   .skip 1

.s:
   .skip 2

.i:
   .skip 4

.l:
   .skip 8

Note:

[section] instruction (to announce BSS section)
[skip] instruction
Defining Data: RODATA Section

```assembly
... "hello\n"...;
...

.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**
Assembly Language: Control-flow instructions
Many instructions have this format:

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

- **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:

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

- **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
Instruction Format

Many instructions have this format:

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

- **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;
```

```
.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
- Register operand
- Immediate operand
- Memory operand
- `.section` instruction
  (to announce TEXT section)

**Registers**

<table>
<thead>
<tr>
<th>EAX</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10</td>
<td></td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th></th>
<th>length</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>width</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>perim</td>
<td>14</td>
</tr>
</tbody>
</table>
Performing Arithmetic: Byte Data

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

### Registers

<table>
<thead>
<tr>
<th>EAX</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A A D 0</td>
</tr>
</tbody>
</table>

### Memory

```plaintext
.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
```

Note:
- Comment
- `movb` instruction
- `subb` instruction
- `decb` instruction

What would happen if we use `movl` instead of `movb`?
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; l ⇒ immediate; M ⇒ memory
Generalization: Data Transfer

Data transfer instructions

```plaintext
mov{q,l,w,b} srcIRM, 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)

cqto       reg[RDX:RAX] = reg[RAX] (sign extend)
cltd       reg[EDX:EAX] = reg[EAX] (sign extend)
cwsl       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 \hspace{1cm} \text{dest} += \text{src}
- sub\{q,l,w,b\} srcIRM, destRM \hspace{1cm} \text{dest} -= \text{src}
- inc\{q,l,w,b\} destRM \hspace{1cm} \text{dest}++
- dec\{q,l,w,b\} destRM \hspace{1cm} \text{dest}--
- neg\{q,l,w,b\} destRM \hspace{1cm} \text{dest} = -\text{dest}

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 \hspace{1cm} 0011_B</td>
<td>3 \hspace{1cm} 0011_B</td>
</tr>
<tr>
<td>+ -4 \hspace{1cm} + 1100_B</td>
<td>+ 12 \hspace{1cm} + 1100_B</td>
</tr>
<tr>
<td>-- \hspace{1cm} ----</td>
<td>-- \hspace{1cm} ----</td>
</tr>
<tr>
<td>-1 \hspace{1cm} 1111_B</td>
<td>15 \hspace{1cm} 1111_B</td>
</tr>
</tbody>
</table>
### Generalization: Bit Manipulation

#### Bitwise instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and{q,l,w,b}</code></td>
<td>srcIRM, destRM</td>
<td>dest = src &amp; dest</td>
</tr>
<tr>
<td><code>or{q,l,w,b}</code></td>
<td>srcIRM, destRM</td>
<td>dest = src</td>
</tr>
<tr>
<td><code>xor{q,l,w,b}</code></td>
<td>srcIRM, destRM</td>
<td>dest = src ^ dest</td>
</tr>
<tr>
<td><code>not{q,l,w,b}</code></td>
<td>destRM</td>
<td>dest = ~dest</td>
</tr>
<tr>
<td><code>sal{q,l,w,b}</code></td>
<td>srcIR, destRM</td>
<td>dest = dest &lt;&lt; src</td>
</tr>
<tr>
<td><code>sar{q,l,w,b}</code></td>
<td>srcIR, destRM</td>
<td>dest = dest &gt;&gt; src (sign extend)</td>
</tr>
<tr>
<td><code>shl{q,l,w,b}</code></td>
<td>srcIR, destRM</td>
<td>(Same as sal)</td>
</tr>
<tr>
<td><code>shr{q,l,w,b}</code></td>
<td>srcIR, destRM</td>
<td>dest = dest &gt;&gt; src (zero fill)</td>
</tr>
</tbody>
</table>

#### Signed (arithmetic right shift)

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>44 / 2^2</td>
<td>000101100B</td>
<td>11</td>
</tr>
<tr>
<td>-44 / 2^2</td>
<td>111010100B</td>
<td>-11</td>
</tr>
</tbody>
</table>

#### Unsigned (logical right shift)

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>44 / 2^2</td>
<td>000101100B</td>
<td>11</td>
</tr>
<tr>
<td>468 / 2^2</td>
<td>11101010100B</td>
<td>117</td>
</tr>
</tbody>
</table>

- Copies of sign bit
- Zeros
Multiplication & Division

### Signed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equation</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>imulb srcRM</code></td>
<td><code>reg[AX] = reg[AL] * src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[RDX] = reg[RDX:RAX] % src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[EDX] = reg[EDX:EAX] % 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>Equation</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>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></td>
<td><code>reg[RDX] = reg[RDX:RAX] % src</code></td>
</tr>
<tr>
<td></td>
<td><code>reg[EDX] = reg[EDX:EAX] % 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
Translation: C to x86-64

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

count \leftrightarrow r10d
n \leftrightarrow r11d

```
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:
```
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

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

- 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

- \[ \text{je } X \quad \text{Jump to } X \text{ if equal} \]
- \[ \text{jne } X \quad \text{Jump to } X \text{ if not equal} \]
- \[ \text{jl } X \quad \text{Jump to } X \text{ if less} \]
- \[ \text{jle } X \quad \text{Jump to } X \text{ if less or equal} \]
- \[ \text{jg } X \quad \text{Jump to } X \text{ if greater} \]
- \[ \text{jge } X \quad \text{Jump to } X \text{ if greater or equal} \]

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

```
loop:
  movl $0, %r10d
  cmp $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.

```
address: contents (in hex)
1000: 41ba00000000
1006: 4183fb01
100a: 7e20
100c: 4183c201
1010: 4489d8
1013: 83e001
1016: 740f
1018: 4489d8
101b: 4101c3
101e: 4101c3
1021: 4183c301
1025: eb03
1027: 41d1fb
102a: ebda
102c: ...
```

\[20 = 102c - 100c \text{ (hex)}\]
Label stands for an address

```
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:
```

address: contents (in hex)

```
1000: 41ba00000000
1006: 4183fb01
100a: 7e20  20 = 102c - 100c (hex)
100c: 4183c201
1010: 4489d8
1013: 83e001
1016: 740f
1018: 4489d8
101b: 4101c3
101e: 4101c3
1021: 4183c301
1025: eb03
1027: 41d1fb
102a: ebda
102c:
```
Translation: C to x86-64

```c
int 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 %eax, %r11d
    addl $1, %r11d
    jmp endif
else:
    sarl $1, %r11d
endif:
    jmp loop
endloop:
```
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
Big-endian vs little-endian 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”

Some other systems use big endian

- Most significant byte of multi-byte entity is stored at lowest memory address
- “Big end goes first”
Byte Order Example 1

```c
#include <stdio.h>
int main(void)
{
    unsigned int i = 0x003377ff;
    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: 77
- Byte 2: 33
- 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

x86-64 is little endian, so what will be the value of grade?

What would be the value of grade if x86-64 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

```
.section ".data"
grade: .byte 'B'
...
.section ".text"
...
  # Option 1
  movl grade, %eax
  subl $1, %eax
  movl %eax, grade
  ...
  # Option 2
  subl $1, grade
```