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
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

```
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
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).

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

else:
  jmp endif
  sarl $1, %r1ld
endif:
  jmp loop
endloop:
```
movl $0, %r10d
loop:
cmpl $1, %r11d
je endloop
addl $1, %r10d
movl %r11d, %eax
andl $1, %eax
je else
movl %r11d, %eax
addl %eax, %r11d
addl %eax, %r11d
addl $1, %r11d
else:
jmp endif
endif:
sarl $1, %r11d
endloop:
jmp loop
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 %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

![Diagram showing the Von Neumann Architecture](image-url)
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>High</th>
<th>Medium</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>EAX</td>
<td>AX</td>
<td>AL</td>
</tr>
<tr>
<td>RBX</td>
<td>EBX</td>
<td>BX</td>
<td>BL</td>
</tr>
<tr>
<td>RCX</td>
<td>ECX</td>
<td>CX</td>
<td>CL</td>
</tr>
<tr>
<td>RDX</td>
<td>EDX</td>
<td>DX</td>
<td>DL</td>
</tr>
</tbody>
</table>
Registers (x86-64 architecture)

General purpose registers (cont.):

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</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>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIL</td>
<td></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>
<td></td>
</tr>
<tr>
<td>EDI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIL</td>
<td></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>
<td></td>
</tr>
<tr>
<td>EBP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPL</td>
<td></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>
<td></td>
</tr>
<tr>
<td>ESP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPL</td>
<td></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>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>R8D</td>
<td>R8W</td>
<td>R8B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td>R9D</td>
<td>R9W</td>
<td>R9B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td>R10D</td>
<td>R10W</td>
<td>R10B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td>R11D</td>
<td>R11W</td>
<td>R11B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td>R12D</td>
<td>R12W</td>
<td>R12B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R13</td>
<td>R13D</td>
<td>R13W</td>
<td>R13B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R14</td>
<td>R14D</td>
<td>R14W</td>
<td>R14B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R15</td>
<td>R15D</td>
<td>R15W</td>
<td>R15B</td>
<td></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)
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, 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, 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”
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

![Diagram of ALU](image)

- ALU (Arithmetic Logic Unit)
  - src1
  - src2
  - operation
  - 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

![CPU Diagram]

RAM

Control Unit

Registers

ALU

Data bus

CPU
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 (\texttt{mov}, \texttt{add}, \texttt{sub}, \texttt{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:

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

- **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`
Instruction Format

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

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** instruction
  (to announce TEXT section)

```
    .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
```

**Registers**

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

**Memory**

| length | 5 |
| width  | 2 |
| perim  | 14 |
Performing Arithmetic: Byte Data

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

.registers
EAX | A | grade | A | A | D | 0 |

.memory
Note:
Comment
movb  instruction
subb  instruction
decb  instruction

What would happen if we use movl instead of movb?

Note:
Comment
movb  instruction
subb  instruction
decb  instruction

What would happen if we use movl instead of movb?

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

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

Memory operands
• 5 $\Rightarrow \text{load from (or store to) memory at address 5 (silly; seg fault*)}$
• i $\Rightarrow \text{load from (or store to) memory at the address denoted by } i$
• 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
### Data Transfer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov{q,l,w,b} srcIRM, destRM</code></td>
<td>dest = src</td>
</tr>
<tr>
<td><code>movsb{q,l,w} srcRM, destR</code></td>
<td>dest = src (sign extend)</td>
</tr>
<tr>
<td><code>movsw{q,l} srcRM, destR</code></td>
<td>dest = src (sign extend)</td>
</tr>
<tr>
<td><code>movslq srcRM, destR</code></td>
<td>dest = src (sign extend)</td>
</tr>
<tr>
<td><code>movzb{q,l,w} srcRM, destR</code></td>
<td>dest = src (zero fill)</td>
</tr>
<tr>
<td><code>movzw{q,l} srcRM, destR</code></td>
<td>dest = src (zero fill)</td>
</tr>
<tr>
<td><code>movzlq srcRM, destR</code></td>
<td>dest = src (zero fill)</td>
</tr>
<tr>
<td><code>cqto</code></td>
<td><code>reg[RDX:RAX] = reg[RAX]</code> (sign extend)</td>
</tr>
<tr>
<td><code>cltd</code></td>
<td><code>reg[EDX:EAX] = reg[EAX]</code> (sign extend)</td>
</tr>
<tr>
<td><code>cwtl</code></td>
<td><code>reg[EAX] = reg[AX]</code> (sign extend)</td>
</tr>
<tr>
<td><code>cbtw</code></td>
<td><code>reg[AX] = reg[AL]</code> (sign extend)</td>
</tr>
</tbody>
</table>

**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]

<table>
<thead>
<tr>
<th>signed 2’s complement</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0011\text{\textsubscript{B}}</td>
</tr>
<tr>
<td>+ -4</td>
<td>+ 1100\text{\textsubscript{B}}</td>
</tr>
<tr>
<td>--</td>
<td>----</td>
</tr>
<tr>
<td>-1</td>
<td>1111\text{\textsubscript{B}}</td>
</tr>
<tr>
<td>3</td>
<td>0011\text{\textsubscript{B}}</td>
</tr>
<tr>
<td>+ 12</td>
<td>+ 1100\text{\textsubscript{B}}</td>
</tr>
<tr>
<td>--</td>
<td>----</td>
</tr>
<tr>
<td>15</td>
<td>1111\text{\textsubscript{B}}</td>
</tr>
</tbody>
</table>
Generalization: Bit Manipulation

Bitwise instructions

\[
\begin{align*}
\text{and\{q,l,w,b\} srcIRM, destRM} & \quad \text{dest} = \text{src} \& \text{dest} \\
\text{or\{q,l,w,b\} srcIRM, destRM} & \quad \text{dest} = \text{src} \lor \text{dest} \\
\text{xor\{q,l,w,b\} srcIRM, destRM} & \quad \text{dest} = \text{src} \oplus \text{dest} \\
\text{not\{q,l,w,b\} destRM} & \quad \text{dest} = \neg \text{dest} \\
\text{sal\{q,l,w,b\} srcIR, destRM} & \quad \text{dest} = \text{dest} \ll \text{src} \\
\text{sar\{q,l,w,b\} srcIR, destRM} & \quad \text{dest} = \text{dest} \gg \text{src} \text{ (sign extend)} \\
\text{shl\{q,l,w,b\} srcIR, destRM} & \quad \text{dest} = \text{dest} \gg \text{src} \text{ (zero fill)} \\
\text{shr\{q,l,w,b\} srcIR, destRM} & \quad \text{dest} = \text{dest} \gg \text{src} \text{ (zero fill)} \\
\end{align*}
\]

**signed (arithmetic right shift)**

\[
\begin{align*}
44 / 2^2 & \quad 000101100_B \\
& \quad 000001011_B \\
-44 / 2^2 & \quad 111010100_B \\
& \quad 111110101_B \\
\end{align*}
\]

**unsigned (logical right shift)**

\[
\begin{align*}
44 / 2^2 & \quad 000101100_B \\
& \quad 000001011_B \\
468 / 2^2 & \quad 111010100_B \\
& \quad 001110101_B \\
\end{align*}
\]

- copies of sign bit
- zeros
### Multiplication & Division

#### Signed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>imulq srcRM</td>
<td>reg[RDX:RAX] = reg[RAX]*src</td>
</tr>
<tr>
<td>imull srcRM</td>
<td>reg[EDX:EAX] = reg[EAX]*src</td>
</tr>
<tr>
<td>imulw srcRM</td>
<td>reg[DX:AX] = reg[AX]*src</td>
</tr>
<tr>
<td>imulb srcRM</td>
<td>reg[AX] = reg[AL]*src</td>
</tr>
</tbody>
</table>

#### Unsigned

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulq srcRM</td>
<td>reg[RDX:RAX] = reg[RAX]*src</td>
</tr>
<tr>
<td>mull srcRM</td>
<td>reg[EDX:EAX] = reg[EAX]*src</td>
</tr>
<tr>
<td>mulw srcRM</td>
<td>reg[DX:AX] = reg[AX]*src</td>
</tr>
<tr>
<td>mulb srcRM</td>
<td>reg[AX] = reg[AL]*src</td>
</tr>
<tr>
<td>divb srcRM</td>
<td>reg[AL] = reg[AX]/src, reg[AH] = reg[AX]%src</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
andl $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 immediately follow \text{cmp}
Control Flow with Signed Integers

Unconditional jump

\texttt{jmp X} \quad \text{Jump to address X}

Conditional jumps after comparing signed integers

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

• Examine condition-code bits in EFLAGS register
movl  $0, %r10d

loop:
    cmp1  $1, %r11d
    jle   endloop
    add1  $1, %r10d
    movl  %r11d, %eax
    and1  $1, %eax
    je    else

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

    else:
        sarl  $1, %r11d

endif:
    jmp   endif

else:
    endif:
        jmp   loop

endloop:

address:  contents (in hex)

1000: 41ba00000000
1006: 4183fb01
100a: 7e25  \(25 = 2f–0a\) (hex)
100c: 4183c201
1010: 4489d8
1013: 8324250000000001
101b: 740f
101d: 4489d8
1020: 4101c3
1023: 4101c3
1026: 4183c301
102a: eb03
102c: 41d1fb
102f: 83c301
1031:
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
else:
  jmp endif
endif:
  jmp loop
endloop:

address: contents (in hex)

1000: 41ba00000000
1006: 4183fb01
100a: 7e25 25 = 31–0c (hex)
100c: 4183c201
1010: 4489d8
1013: 8324250000000001
101b: 740f
101d: 4489d8
1020: 4101c3
1023: 4101c3
1026: 4183c301
102a: eb03
102c: 41d1fb
102f: 83c301
1031:
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
else:
    jmp endif
endif:
    sarl $1, %r11d
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