Assembly Language: Overview

Jennifer Rexford
Goals of this Lecture

• Help you learn:
  • The basics of computer architecture
  • The relationship between C and assembly language
  • IA-32 assembly language, through an example
Context of this Lecture

Second half of the course

Starting Now

- C Language
- Assembly Language
- Machine Language

language levels tour

Afterward

- Application Program
- Operating System
- Hardware

service levels tour
Three Levels of Languages
High-Level Language

- Make programming easier by describing operations in a semi-natural language
- Increase the portability of the code
- One line may involve many low-level operations
- Examples: C, C++, Java, Pascal, …

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

- Tied to the specifics of the underlying machine
- Commands and names to make the code readable and writeable by humans
- Hand-coded assembly code may be more efficient
- E.g., IA-32 from Intel

```
movl  $0, %ecx

loop:
    cmpl  $1, %edx
    jle   endloop
    addl  $1, %ecx
    movl  %edx, %eax
    andl  $1, %eax
    je    else
    movl  %edx, %eax
    addl  %eax, %edx
    addl  %eax, %edx
    addl  $1, %edx
    jmp   endif
else:
    sarl  $1, %edx
endif:
    jmp   loop
endloop:
```
Machine Language

- Also tied to the underlying machine
- What the computer sees and deals with
- Every instruction is a sequence of one or more numbers
- All stored in memory on the computer, and read and executed
- Unreadable by humans

```
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
```
Why Learn Assembly Language?

• Write faster code (even in high-level language)
  • By understanding which high-level constructs are better
  • … in terms of how efficient they are at the machine level

• Understand how things work underneath
  • Learn the basic organization of the underlying machine
  • Learn how the computer actually runs a program
  • Design better computers in the future

• Some software is still written in assembly language
  • Code that really needs to run quickly
  • Code for embedded systems, network processors, etc.
Why Learn Intel IA-32 Assembly?

• Program natively on our computing platform
  • Rather than using an emulator to mimic another machine

• Learn instruction set for the most popular platform
  • Most likely to work with Intel platforms in the future

• But, this comes at some cost in complexity
  • IA-32 has a large and varied set of instructions
  • More instructions than are really useful in practice

• Fortunately, you won’t need to use everything
Computer Architecture
Von Neumann Architecture

- Central Processing Unit
  - Control unit
    - Fetch, decode, and execute
  - Arithmetic and logic unit
    - Execution of low-level operations
  - General-purpose registers
    - High-speed temporary storage
  - Data bus
    - Provide access to memory
Von Neumann Architecture

- Memory
  - Store executable machine-language instructions (text section)
  - Store data (rodata, data, bss, heap, and stack sections)

```
<table>
<thead>
<tr>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>RODATA</td>
</tr>
<tr>
<td>DATA</td>
</tr>
<tr>
<td>BSS</td>
</tr>
<tr>
<td>HEAP</td>
</tr>
<tr>
<td>STACK</td>
</tr>
</tbody>
</table>
```

Diagram:
- CPU
- Control Unit
- ALU
- Registers
- Random Access Memory (RAM)
- Data bus

Random Access Memory (RAM)
Control Unit: Instruction Pointer

- Stores the location of the next instruction
  - Address to use when reading machine-language instructions from memory (i.e., in the text section)

- Changing the instruction pointer (EIP)
  - Increment to go to the next instruction
  - Or, load a new value to “jump” to a new location
Control Unit: Instruction Decoder

- Determines what operations need to take place
  - Translate the machine-language instruction
- Control what operations are done on what data
  - E.g., control what data are fed to the ALU
  - E.g., enable the ALU to do multiplication or addition
  - E.g., read from a particular address in memory

![Diagram of ALU with inputs and outputs]
Registers

• Small amount of storage on the CPU
  • Can be accessed more quickly than main memory

• Instructions move data in and out of registers
  • Loading registers from main memory
  • Storing registers to main memory

• Instructions manipulate the register contents
  • Registers essentially act as temporary variables
  • For efficient manipulation of the data

• Registers are the top of the memory hierarchy
  • Ahead of main memory, disk, tape, …
Keeping it Simple: All 32-bit Words

- Simplifying assumption: all data in four-byte units
  - Memory is 32 bits wide
  - Registers are 32 bits wide

- In practice, can manipulate different sizes of data
C Code vs. Assembly Code
Kinds of Instructions

- Reading and writing data
  - count = 0
  - n
- Arithmetic and logic operations
  - Increment: count++
  - Multiply: n * 3
  - Divide: n/2
  - Logical AND: n & 1
- Checking results of comparisons
  - Is (n > 1) true or false?
  - Is (n & 1) non-zero or zero?
- Changing the flow of control
  - To the end of the while loop (if “n > 1”)
  - Back to the beginning of the loop
  - To the else clause (if “n & 1” is 0)

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

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

Referring to a register: percent sign ("%")

**Registers**

\[ n \quad \%edx \]
\[ count \quad \%ecx \]
```c
count=0;
while (n>1) {
    count++;
    if (n&1)
        n = n*3+1;
    else
        n = n/2;
}
```

Immediate and Register Addressing

Referring to a immediate operand: dollar sign ("$")

Read directly from the instruction

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

Computing intermediate value in register EAX
count=0;
while (n>1) {
    count++;
    if (n&1)
        n = n*3+1;
    else
        n = n/2;
}

Adding n twice is cheaper than multiplication!
Immediate and Register Addressing

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

Shifting right by 1 bit is cheaper than division!
Changing Program Flow

- Cannot simply run next instruction
  - Check result of a previous operation
  - Jump to appropriate next instruction

- Flags register (EFLAGS)
  - Stores the status of operations, such as comparisons, as a side effect
  - E.g., last result was positive, negative, zero, etc.

- Jump instructions
  - Load new address in instruction pointer

- Example jump instructions
  - Jump unconditionally (e.g., “}”)
  - Jump if zero (e.g., “n&1”)
  - Jump if greater/less (e.g., “n>1”)

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

- **Comparison `cmp1` compares two integers**
  - Done by subtracting the first number from the second
  - Discarding the results, but setting flags as a side effect
  - Example:
    - `cmp1 $1, %edx` (computes `%edx – 1`)
    - `jle endloop` (checks whether result was 0 or negative)

- **Logical operation `andl` compares two integers**
  - Example:
    - `andl $1, %eax` (bit-wise AND of `%eax` with 1)
    - `je else` (checks whether result was 0)

- Also, can do an unconditional branch `jmp`
  - Example:
    - `jmp endif and jmp loop`
Jump and Labels: While Loop

while (n>1) {
    cmpl $1, %edx
    jle endloop
    ...
}
endloop:

Checking if EDX is less than or equal to 1.

Jump and Labels: While Loop

while (n>1) {
    cmpl $1, %edx
    jle endloop
    ...
}
endloop:
jmp loop
Jump and Labels: While Loop

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

movl $0, %ecx
cmp $1, %edx
jle endloop
addl $1, %ecx
movl %edx, %eax
andl $1, %eax
je else
movl %edx, %eax
addl %eax, %edx
addl %eax, %edx
addl $1, %edx
jmp endif
else:
sarl $1, %edx
endif:
jmp loop
endloop:
Jump and Labels: If-Then-Else

```
if (n&1)
  ...
else
  ...
```

```
movl %edx, %eax
andl $1, %eax
je else
endif
```

```
else:
  ...
endif:
```

```
else:
  ...
endif:
```

```
\textbf{“then” block}\]
```

```
\textbf{“else” block}\]
```
Jump and Labels: If-Then-Else

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

Jump and Labels: If-Then-Else

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;
}

Replace with “jmp loop”

Making the Code More Efficient…

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

movl $0, %ecx
loop:
cmpl $1, %edx
jle endloop
addl $1, %ecx
movl %edx, %eax
andl $1, %eax
je else
movl %edx, %eax
addl %eax, %edx
addl %eax, %edx
addl $1, %edx
addl $1, %ecx
jmp endif
else:
sarh $1, %edx
endif:
jmp loop
endloop:
```c
count=0;
while (n>1) {
    count++;
    if (n&1)
        n = n*3+1;
    else
        n = n/2;
}
```

```
    movl $0, %ecx

loop:
    cmpl $1, %edx
    jle endloop

    addl $1, %ecx

    movl %edx, %eax
    andl $1, %eax
    je else

    movl %edx, %eax
    addl %eax, %edx
    addl %eax, %edx
    addl $1, %edx

else:
    jmp endif

endif:
    sarl $1, %edx

endloop:
```
Referring to a register: percent sign ("%")
  • E.g., "%ecx" or "%eip"

Referring to immediate operand: dollar sign ("$")
  • E.g., "$1" for the number 1

Storing result: typically in the second argument
  • E.g. “addl $1, %ecx” increments register ECX
  • E.g., “movl %edx, %eax” moves EDX to EAX

Assembler directives: starting with a period (".")
  • E.g., “.section .text” to start the text section of memory

Comment: pound sign ("#")
  • E.g., “# Purpose: Convert lower to upper case”
Conclusions

- **Assembly language**
  - In between high-level language and machine code
  - Programming the “bare metal” of the hardware
  - Loading and storing data, arithmetic and logic operations, checking results, and changing control flow

- **To get more familiar with IA-32 assembly**
  - Read more assembly-language examples
    - Chapter 3 of Bryant and O’Hallaron book
  - Generate your own assembly-language code
    - `gcc217 –S –O2 code.c`