Assembly Language: Overview

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

Three Levels of Languages

High-Level Language

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

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

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

```assembly
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 hardware 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

Why Learn Assembly Language?

- Write faster code (even in high-level language)
  - By understanding which high-level constructs are more efficient 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 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)
Control Unit: Instruction Pointer

- EIP: 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
  - Increment it to go to the next instruction
  - Or, load a new value into EIP to "jump" to a new location

Control Unit: Instruction Decoder

- Determines what operations need to take place
  - Translates 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

Registers

- Small amount of storage on the CPU
  - Can be accessed more quickly than main memory
  - Each register has a name, which assembly code uses
- Instructions move data in and out of registers
  - Loading (into) registers from main memory
  - Storing (from) 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 caches, 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
  - Bitwise 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)

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

Assign Registers to key Variables

<table>
<thead>
<tr>
<th>Registers</th>
<th>Addressing Registers and Immediates</th>
</tr>
</thead>
<tbody>
<tr>
<td>n %edx</td>
<td>count = 0;</td>
</tr>
<tr>
<td>count %ecx</td>
<td>while (n &gt; 1) {</td>
</tr>
<tr>
<td></td>
<td>count++;</td>
</tr>
<tr>
<td></td>
<td>if (n &amp; 1)</td>
</tr>
<tr>
<td></td>
<td>n = n*3 + 1;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>n = n/2;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>movl $0, %ecx</td>
</tr>
<tr>
<td></td>
<td>addl $1, %ecx</td>
</tr>
</tbody>
</table>

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

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

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

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

Computing intermediate value in register EAX

```
movl %edx, %eax
andi $1, %eax
```

Addressing Registers and Immediates

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

Update %edx this time, since changing value of n

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

Addressing Registers and Immediates

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

```
sarl $1, %edx
```

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., ”j”)
  - Jump if zero (e.g., ”jz”)
  - Jump if greater/less (e.g., ”jg”/”jl”)
Conditional and Unconditional Jumps

- **Comparison** `cmpl` compares two integers
  - Done by subtracting the first number from the second
  - Discarding the results, but setting flags as a side effect
  - Example:
    - `cmpl $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

```assembly
loop: cmpl $1, %edx
     jle endloop
while (n>1) {
  count++; 
  if (n&1) 
    n = n*3+1;
  else 
    n = n/2;
}
```

Jump and Labels: If-Then-Else

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

Jump and Labels: If-Then-Else

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

Making the Code More Efficient...

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

Complete Example

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

Reading IA-32 Assembly Language

- Referring to a register: percent sign (“%”)  
  - E.g., “%ecx” or “%ebp”
- 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, but mnemonically and not just with bits (machine language)
  - 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