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
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, ...

```c
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
    addl $1, %eax
    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

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
A Typical Computer

John Von Neumann (1903-1957)

- In computing
  - Stored program computers
  - Cellular automata
  - Self-replication

- Other interests
  - Mathematics (set theory, ergodic theory)
  - Nuclear physics (hydrogen bomb)

- Princeton connection
  - Princeton Univ & IAS, 1930-death

- Known for “Von Neumann architecture”
  - In contrast to less-successful “Harvard 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

- 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
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)

```c
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;
}
```

Registers

n %edx
count %ecx

Referring to a register: percent sign (“%”)

Immediate and Register Addressing

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

movl $0, %ecx
addl $1, %ecx

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

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

```c
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

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

• 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”)
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

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

Checking if EDX is less than or equal to 1.

```
jmp loop
endloop:
```
Jump and Labels: While Loop

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

Jump and Labels: If-Then-Else

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

```
Jump and Labels: While Loop

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:
endif:
jmp loop
endloop:

Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

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

```
Jump and Labels: If-Then-Else

if (n&1)
    ...
else
    ...
```
### Jump and Labels: If-Then-Else

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

```
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...

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

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

```
Replace with “jmp loop”
```
**Complete Example**

```
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
else:
    movl %edx, %eax
    addl %eax, %edx
    addl %eax, %edx
    addl $1, %edx
    jmp endif
endif:
sar $1, %edx
endloop:
jmp loop
```

**Reading IA-32 Assembly Language**

- **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