Computer Systems  CEN591(502)
Fall 2011

Sandeep K. S. Gupta
Arizona State University
6th lecture
Machine-Level Programming I: Basics
(Slides adapted from CSAPP)
Announcements

- HW1 due date:
  - tonight 11:59 pm
- Next class Quiz will be graded
Previous class topics

- Machine int and float representation
  - int operations
  - float operations
  - C data types, and operations

- This class: Machine-level programming
Agenda

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move, operations and condition statements
- Introduction to x86-64
Intel x86 Processors

- Totally dominate laptop/desktop/server market

- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on

- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.
CISC

- Complex Instruction Set Computer
  - Large number of complex instructions
  - Low level
  - Facilitate the extensive manipulation of low-level computational elements and events such as memory, binary arithmetic, and addressing.
  - [emphasize on hardware]

- Examples: Intel x86 architecture based processors
RISC

- **Reduced Instruction Set Computer**
  - Small number of instructions
  - instruction size constant
  - bans the indirect addressing mode
  - retains only those instructions that can be overlapped and made to execute in one machine cycle or less.
  - [Emphasize on software]

- **Examples:** Apple iPods, and iPhone, Some Nokia and Sony Ericsson mobile phones

# Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Intel x86 Processors: Overview

<table>
<thead>
<tr>
<th>Architectures</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>X86-16</td>
<td>8086</td>
</tr>
<tr>
<td>X86-32/IA32</td>
<td>286</td>
</tr>
<tr>
<td>MMX</td>
<td>386</td>
</tr>
<tr>
<td>SSE</td>
<td>486</td>
</tr>
<tr>
<td>SSE2</td>
<td>Pentium</td>
</tr>
<tr>
<td>SSE3</td>
<td>Pentium MMX</td>
</tr>
<tr>
<td>X86-64/EM64t</td>
<td>Pentium III</td>
</tr>
<tr>
<td>SSE4</td>
<td>Pentium 4</td>
</tr>
<tr>
<td></td>
<td>Pentium 4E</td>
</tr>
<tr>
<td></td>
<td>Pentium 4F</td>
</tr>
<tr>
<td></td>
<td>Core 2 Duo</td>
</tr>
<tr>
<td></td>
<td>Core i7</td>
</tr>
</tbody>
</table>

IA: often redefined as latest Intel architecture
Intel x86 Processors, contd.

- **Machine Evolution**
  - 386 1985 0.3M
  - Pentium 1993 3.1M
  - Pentium/MMX 1997 4.5M
  - PentiumPro 1995 6.5M
  - Pentium III 1999 8.2M
  - Pentium 4 2001 42M
  - Core 2 Duo 2006 291M
  - Core i7 2008 731M

- **Added Features**
  - Instructions to support multimedia operations
    - Parallel operations on 1, 2, and 4-byte data, both integer & FP
  - Instructions to enable more efficient conditional operations

- **Linux/GCC Evolution**
  - Two major steps: 1) support 32-bit 386. 2) support 64-bit x86-64
x86 Clones: Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper

- Then
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits
Intel’s 64-Bit

- Intel Attempted Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  -Executes IA32 code only as legacy
  - Performance disappointing

- AMD Stepped in with Evolutionary Solution
  - x86-64 (now called “AMD64”)

- Intel Felt Obligated to Focus on IA64
  - Hard to admit mistake or that AMD is better

- 2004: Intel Announces EM64T extension to IA32
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!

- All but low-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode
Our Coverage

- **IA32**
  - The traditional x86

- **x86-64/EM64T**
  - The emerging standard

- **Machine programming style: ATT style**

- **Presentation**
  - Book presents IA32 in Sections 3.1—3.12
  - Covers x86-64 in 3.13
  - We will cover both simultaneously
Definitions

- **Architecture**: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
  - Examples: instruction set specification, registers.

- **Microarchitecture**: Implementation of the architecture.
  - Examples: cache sizes and core frequency.

- **Example ISAs (Intel)**: x86, IA, IPF
Assembly Programmer’s View

- Programmer-Visible State
  - PC: Program counter
    - Address of next instruction
    - Called “EIP” (IA32) or “RIP” (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- Memory
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures
Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (`-O1`)
  - Put resulting binary in file `p`

```
text
C program (p1.c p2.c)

Compiler (gcc -S)

Asm program (p1.s p2.s)

Assembler (gcc or as)

Object program (p1.o p2.o)

Linker (gcc or ld)

Executable program (p)
```

Static libraries (.a)
Compiling Into Assembly

C Code

```c
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

Generated IA32 Assembly

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    popl %ebp
    ret
```

The suffix “l” points to the size of operands

Obtain with command

```
gcc -01 -S code.c
```

Produces file `code.s`

-01: to compile at the first optimization level (higher levels are available)
-S: to generate the assembly code
# C Data types in IA32

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>Assembly code suffix</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>Byte</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>Short</td>
<td>Word</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>Int</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>Long int</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>Long long int</td>
<td>- (needs generating sequence of operations)</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>Char *</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>Single precision</td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>Double</td>
<td>Double precision</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>Long double</td>
<td>Extended precision</td>
<td>t</td>
<td>10/12</td>
</tr>
</tbody>
</table>

No aggregate types such as arrays or structures!
Perform arithmetic function on register or memory data

Transfer data between memory and register
- Load data from memory into register
- Store register data into memory

Transfer control
- Unconditional jumps to/from procedures
- Conditional branches
Machine Instruction Example

- **C Code**
  - Add two signed integers

- **Assembly**
  - Add 2 4-byte integers
    - “Long” words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:
    - \(x\): Register \(\%eax\)
    - \(y\): Memory \(M[\%ebp+8]\)
    - \(t\): Register \(\%eax\)
      - Return function value in \(\%eax\)

- **Object Code**
  - 3-byte instruction
  - Stored at address \(0x80483ca\)

```c
int t = x+y;

addl 8(%ebp),%eax
```

Similar to expression:
\[x += y\]

More precisely:
```c
int eax;
int *ebp;
eax += ebp[2]
```

```
0x80483ca:  03 45 08
```

CEN591 Fall 2011
Disassembling Object Code

Disassembled

080483c4 <sum>:

80483c4:  55 push %ebp
80483c5:  89 e5 mov %esp,%ebp
80483c7:  8b 45 0c mov 0xc(%ebp),%eax
80483ca:  03 45 08 add 0x8(%ebp),%eax
80483cd:  5d pop %ebp
80483ce:  c3 ret

Disassembler

`objdump -d prog`

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a.out (complete executable) or .o file
Assembly Basics: Registers, operands, move
Integer Registers (IA32)

16-bit virtual registers (backwards compatibility)

- %eax
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp

- %ax
- %dx
- %bx
- %si
- %di
- %sp

- %ah
- %dh
- %bh
- %bl

- %cl
- %dl
- %bh
- %bl

General purpose

- %ecx
- %edx
- %ebx
- %esi
- %edi

Origin (mostly obsolete)

- %eax
- %ax
- %ah
- %al
- %ecx
- %cx
- %ch
- %cl
- %edx
- %dx
- %dh
- %dl
- %ebx
- %bx
- %bh
- %bl
- %esi
- %si
- %edi
- %di

- accumulate
- counter
- data
- base
- Source index
- destination index
- Stack pointer
- Base pointer

CEN591 Fall 2011
Moving Data: IA32

- Moving Data

  `movl Source, Dest`:

- Operand Types

  - **Immediate**: Constant integer data
    - Example: `$0x400, $-533`
    - Like C constant, but prefixed with `$`
    - Encoded with 1, 2, or 4 bytes
  
  - **Register**: One of 8 integer registers
    - Example: `%eax, %edx`
    - But `%esp` and `%ebp` reserved for special use
    - Others have special uses for particular instructions
  
  - **Memory**: 4 consecutive bytes of memory at address given by register
    - Simplest example: `(eax)`
    - Various other “address modes”
## movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax,(%edx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction*
What is next?

- IA32 control and loop operations, procedures
  - Read chapters 3.5, 3.6, and 3.7 of the CSAPP book
Extra slides
**Alternate Disassembly**

<table>
<thead>
<tr>
<th>Object</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040:</td>
<td></td>
</tr>
<tr>
<td>0x55</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xe5</td>
<td></td>
</tr>
<tr>
<td>0x8b</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x0c</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x5d</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></td>
</tr>
</tbody>
</table>

Dump of assembler code for function `sum`:

```
0x080483c4 <sum+0>:    push   %ebp
0x080483c5 <sum+1>:    mov    %esp,%ebp
0x080483c7 <sum+3>:    mov    0xc(%ebp),%eax
0x080483ca <sum+6>:    add    0x8(%ebp),%eax
0x080483cd <sum+9>:    pop     %ebp
0x080483ce <sum+10>:   ret
```

- **Within gdb Debugger**
  - `gdb p` 
  - `disassemble sum`
    - Disassemble procedure
  - `x/11xb sum`
    - Examine the 11 bytes starting at `sum`
What Can be Disassembled?

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source