Lecture 4: Machine Basics

Computer Architecture and
Systems Programming

(252-0061-00)

Timothy Roscoe
Herbstsemester 2012
Last Time: Floating Point

- Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
Today: Basic machine language programming

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move

What is an “Architecture”?
- Machine viewed by software (machine code)
- Assembly Instructions, registers, trap handling, etc.
Intel x86 Processors

• The x86 Architecture dominates the computer 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!
## 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>First 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>1MB address space</td>
<td></td>
</tr>
<tr>
<td>80386</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>First 32 bit processor, referred to as IA32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>Added “flat addressing”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>Capable of running Unix</td>
<td></td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>32-bit Linux/gcc uses no instructions introduced in later models</td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>2005</td>
<td>230M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>First 64-bit [x86] processor</td>
<td></td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>Meanwhile, Pentium 4s (Netburst arch.) phased out in favor of “Core” line</td>
<td></td>
</tr>
</tbody>
</table>
Intel x86 Processors: Overview

Architectures

- X86-16
- X86-32/IA32
- IA: often redefined as latest Intel architecture

Processors

- 8086
- 286
- 386
- 486
- Pentium
- Pentium MMX
- Pentium III
- Pentium 4
- Pentium 4E
- Pentium 4F
- Core 2 Duo
- Core i7

SSE

SSE2

SSE3

SSE4

time
Intel x86 Processors, contd.

• Machine Evolution
  486  1989  1.9M
  Pentium  1993  3.1M
  Pentium/MMX ‘97  74.5M
  PentiumPro 1995  6.5M
  Pentium III  1999  8.2M
  Pentium 4  2001  42M
  Core 2 Duo  2006  291M

• 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
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

• Recently
  – Intel much quicker with dual core design
  – Intel currently far ahead in performance
  – em64t backwards compatible to x86-64
Intel’s 64-Bit  
(partially true...)

• 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!
Intel Nehalem-EX

• Current leader (for the next few weeks)
  – 2.3 billion transistors/die
  – 8 or 10 cores per die
  – 2 threads per core
  – Up to 8 packages (= 128 contexts!)
  – 4 memory channels per package
  – Virtualization support
  – etc.

• Good illustration of why it is hard to teach state-of-the-art processor design!
Intel Single-Chip Cloud Computer - 2010

• Experimental processor (only a few 100 made)
  – Designed for research
  – Working version in our Lab

• 48 old-style Pentium cores

• Very fast interconnection network
  – Hardware support for messaging between cores
  – Variable speed of network

• Non-cache coherent
  – Sharing memory between cores won’t work with a conventional OS!
There are many architectures...

- You’ve already seen MIPS 2000 → MIPS 3000 → ...
  - Workstations, minicomputers, now mostly embedded networking
- IBM S/360 → S/370 → ... → zSeries
  - First to separate architecture from (many) implementations
- ARM (several variants)
  - Very common in embedded systems, basis for Advanced OS course at ETHZ
- IBM POWER → PowerPC (→ Cell, sort of)
  - Basis for all 3 current games console systems
- DEC Alpha
  - Personal favorite; killed by Compaq, team left for Intel to work on...
- Intel Itanium
  - First 64-bit Intel product; very fast (esp. FP), hot, and expensive
  - Mostly overtaken by 64-bit x86 designs
- etc.
Our Coverage

• ia32
  – The traditional x86

• x86-64/EM64T
  – The emerging standard

• Presentation
  – Book has ia32
  – Handout has x86-64
  – Lecture will cover both
Comparison with MIPS
(remember Digital Design?)

• MIPS is RISC: Reduced Instruction Set
  – Motivation: simpler is faster
    • Fewer gates ⇒ higher frequency
    • Fewer gates ⇒ more transistors left for cache
  – Seemed like a really good idea
• x86 is CISC: Complex Instruction Set
  – More complex instructions, addressing modes
    • Intel turned out to be way too good at manufacturing
    • Difference in gate count became too small to make a difference
    • x86 inside is mostly RISC anyway, decode logic is small
  – ⇒ Argument is mostly irrelevant these days
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
Definitions

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

• **Microarchitecture**: Implementation of the architecture.

• **Architecture examples**: instruction set specification, registers.

• **Microarchitecture examples**: cache sizes and core frequency.

• Example ISAs: x86, MIPS, ia64, VAX, Alpha, ARM, etc.
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 \texttt{p1.c \ p2.c}
- Compile with command: \texttt{gcc -O p1.c p2.c -o p}
  - Use optimizations (-O)
  - Put resulting binary in file \texttt{p}

\begin{itemize}
  \item \textbf{text}
    \begin{itemize}
      \item C program (\texttt{p1.c \ p2.c})
    \end{itemize}
  \item \textbf{text}
    \begin{itemize}
      \item Asm program (\texttt{p1.s \ p2.s})
    \end{itemize}
  \item \textbf{binary}
    \begin{itemize}
      \item Object program (\texttt{p1.o \ p2.o})
    \end{itemize}
  \item \textbf{binary}
    \begin{itemize}
      \item Executable program (\texttt{p})
    \end{itemize}
\end{itemize}

\begin{itemize}
  \item Compiler (\texttt{gcc -S})
  \item Assembler (\texttt{gcc \ or \ as})
  \item Linker (\texttt{gcc \ or \ ld})
  \item Static libraries (\texttt{.a})
\end{itemize}
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
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`

Some compilers use single instruction “leave”
Assembly Characteristics: Data Types

• “Integer” data of 1, 2, or 4 bytes
  – Data values
  – Addresses (untyped pointers)

• Floating point data of 4, 8, or 10 bytes

• No aggregate types such as arrays or structures
  – Just contiguously allocated bytes in memory
Assembly Characteristics: Operations

- 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
Object Code

Code for `sum`

```
0x401040  <sum>:
  0x55
  0x89
  0xe5
  0x8b
  0x0c
  0x03
  0x45
  0x08
  0x89
  0xec
  0x5d
  0xc3
```

- **Assembler**
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files

- **Linker**
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for `malloc`, `printf`
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address `0x401040`
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 **0x401046**

```c
int t = x+y;
addl 8(%ebp),%eax
```

Similar to expression:

```c
x += y
```

More precisely:

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

**0x401046:** 03 45 08

Similar to expression:

```c
x += y
```

More precisely:

```c
int eax;
int *ebp;
eax += ebp[2]
```
Disassembling Object Code

Disassembled

```
00401040  <_sum>:
  0:  55              push   %ebp
  1:  89 e5           mov    %esp,%ebp
  3:  8b 45 0c        mov    0xc(%ebp),%eax
  6:  03 45 08        add    0x8(%ebp),%eax
  9:  89 ec           mov    %ebp,%esp
 b:  5d              pop    %ebp
 c:  c3              ret
 d:  8d 76 00        lea    0x0(%esi),%esi
```

• Disassembler
  – `objdump -d p`
  – 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
Alternate Disassembly

Object

Disassembled

Within **gdb** Debugger

- **gdb** *p*
- disassemble sum
- Disassemble procedure
- **x/13b** sum
- Examine the 13 bytes starting at **sum**
What Can be Disassembled?

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

% objdump -d WINWORD.EXE

WINWORD.EXE: file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000: 55 push %ebp
30001001: 8b ec mov %esp,%ebp
30001003: 6a ff push $0xffffffff
30001005: 68 90 10 00 30 push $0x30001090
3000100a: 68 91 dc 4c 30 push $0x304cdc91
Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
### Integer Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>General Purpose</th>
<th>16-bit Virtual Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax, %ah, %al</td>
<td>(backwards compatibility)</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx, %ch, %cl</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>%dx, %dh, %dl</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx, %bh, %bl</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
</tr>
</tbody>
</table>

- `%eax` to `%ebp` are general purpose registers.
- `%esp` and `% ebp` are used for stack and base pointer respectively.
- `%esp` is not used in recent architectures.

Origin (mostly obsolete)
Moving Data: IA32

• Moving Data
  – `movx Source, Dest`
  – `x` in `{b, w, l}`
  
  – `movl Source, Dest:`
    Move 4-byte “long word”
  – `movw Source, Dest:`
    Move 2-byte “word”
  – `movb Source, Dest:`
    Move 1-byte “byte”

• Lots of these in typical code
Moving Data: IA32

- Moving Data
  \texttt{movl Source, Dest:}

- Operand Types
  - \textit{Immediate}: Constant integer data
    - Example: $0x400, $-533$
    - Like C constant, but prefixed with `$$`
    - Encoded with 1, 2, or 4 bytes
  - \textit{Register}: One of 8 integer registers
    - Example: \%eax, \%edx
    - But \%esp and \%ebp reserved for special use
    - Others have special uses for particular instructions
  - \textit{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>Reg</td>
<td>Movl $0x4, %eax</td>
<td>temp = 0x4;</td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>Movl $-147, (%eax)</td>
<td>*p = -147;</td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>Movl %eax, %edx</td>
<td>temp2 = temp1;</td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>Movl (%eax), %edx</td>
<td>*p = temp;</td>
<td></td>
</tr>
</tbody>
</table>

**Cannot do memory-memory transfer with a single instruction**
Simple Memory Addressing Modes

• Normal $\text{(R)}$ $\text{Mem[Reg[R]]}$
  
  – Register R specifies memory address

  \[
  \text{movl } (\%\text{ecx}),\%\text{eax}
  \]

• Displacement D(R) $\text{Mem[Reg[R]+D]}$
  
  – Register R specifies start of memory region
  – Constant displacement D specifies offset

  \[
  \text{movl } 8(\%\text{ebp}),\%\text{edx}
  \]
Using Simple Addressing Modes

```c
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
Using Simple Addressing Modes

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```assembly
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

movl 12(%ebp),%ecx           # ecx = yp
movl 8(%ebp),%edx            # edx = xp
movl (%ecx),%eax             # eax = *yp (t1)
movl (%edx),%ebx             # ebx = *xp (t0)
movl %eax,(%edx)             # *xp = eax
movl %ebx,(%ecx)             # *yp = ebx
Understanding Swap

### Register file

<table>
<thead>
<tr>
<th>Register</th>
<th>Value (Offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td>0x120</td>
</tr>
<tr>
<td>0x114</td>
<td>0x118</td>
</tr>
<tr>
<td>0x118</td>
<td>0x11c</td>
</tr>
<tr>
<td>0x120</td>
<td>0x124</td>
</tr>
<tr>
<td>0x124</td>
<td>0x128</td>
</tr>
</tbody>
</table>

### Code

```asm
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx   # edx = xp
movl (%ecx),%eax    # eax = *yp (t1)
movl (%edx),%ebx    # ebx = *xp (t0)
movl %eax,(%edx)   # *xp = eax
movl %ebx,(%ecx)   # *yp = ebx
```

---

According to the diagram and the code, the swap operation is performed as follows:

1. Set `ecx` to the value of `yp`:
   ```asm
   movl 12(%ebp),%ecx
   ```

2. Set `edx` to the value of `xp`:
   ```asm
   movl 8(%ebp),%edx
   ```

3. Load the value from `yp` into `eax`:
   ```asm
   movl (%ecx),%eax
   ```

4. Load the value from `xp` into `ebx`:
   ```asm
   movl (%edx),%ebx
   ```

5. Store `eax` back into `xp`:
   ```asm
   movl %eax,(%edx)
   ```

6. Store `ebx` back into `yp`:
   ```asm
   movl %ebx,(%ecx)
   ```

This sequence of instructions effectively swaps the values of `yp` and `xp`.
Understanding Swap

Register file

| %eax  | 456 |
| %edx  | 0x124 |
| %ecx  | 0x120 |
| %ebx  |    |
| %esi  |    |
| %edi  |    |
| %esp  |    |
| %ebp  | 0x104 |

Memory

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-4</td>
</tr>
</tbody>
</table>

movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx   # edx = xp
movl (%ecx),%eax    # eax = *yp (t1)
movl (%edx),%ebx    # ebx = *xp (t0)
movl %eax,(%edx)   # *xp = eax
movl %ebx,(%ecx)    # *yp = ebx
## Understanding Swap

### Register file

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>%epb → 0</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

### Instruction Explanation

- `movl 12(%ebp),%ecx`  
  # ecx = yp

- `movl 8(%ebp),%edx`  
  # edx = xp

- `movl (%ecx),%eax`  
  # eax = *yp (t1)

- `movl (%edx),%ebx`  
  # ebx = *xp (t0)

- `movl %eax,(%edx)`  
  # *xp = eax

- `movl %ebx,(%ecx)`  
  # *yp = ebx
Complete Memory Addressing Modes

• Most General Form
  \[ \text{D}(R_b, R_i, S) \quad \text{Mem}[\text{Reg}[R_b]+S\times\text{Reg}[R_i]+D] \]
  – D: Constant “displacement” 1, 2, or 4 bytes
  – Rb: Base register: Any of 8 integer registers
  – Ri: Index register: Any, except for \%esp
    • Unlikely you’d use \%ebp, either
  – S: Scale: 1, 2, 4, or 8 (why these numbers?)

• Special Cases
  \( (R_b, R_i) \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]] \)
  \( D(R_b, R_i) \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]+D] \)
  \( (R_b, R_i, S) \quad \text{Mem}[\text{Reg}[R_b]+S\times\text{Reg}[R_i]] \)
Next time

- Complete addressing mode, address computation (**lea**)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops