Machine-Level Programming I: Memory and Instructions  
Lecture 5

Instructor:  
Alvin R. Lebeck

Slides based on those from Randy Bryant and Dave O’Hallaron

Administrivia

- Homework #1 Due Sept 12
  - Three puzzles are extra credit
- Homework #2 will be up tomorrow (C Program w/ pointers)

Outline

- Instruction Set Architectures
- Brief History of Intel Architecture
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64

Reading

Chapter 3
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       |          |     text       |
+-----------------+          |     text       |
| C program (p1.c p2.c) |  Compiler (gcc -S) | Asm program (p1.s p2.s) |
|              |          |                |
| binary        |          | binary         |
| Object program (p1.o p2.o) | Linker (gcc or ld) | Static libraries (.a) |
|              |          | Executable program (p) |
```

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.
# Instruction Sets

```c
#include <stdio.h>

main()
{
    int a[100];
    int *p;
    int k;

    p = &a;
    for (k = 0; k < 100; k++)
    {
        *p = k;
        p++;
    }

    printf("entry 3 = %d\n", a[3]);
}
```

What primitive operations do we need?
(i.e., What should be implemented in hardware?)

---

## Design Space of Instruction Sets

- **Five Primary Dimensions**
  - **Operations**
    - add, sub, mul, ...
    - How is it specified?
  - **Number of explicit operands**
    - (0, 1, 2, 3)
  - **Operand Storage**
    - Where besides memory?
  - **Memory Address**
    - How is memory location specified?
  - **Type & Size of Operands**
    - byte, int, float, vector, ...
    - How is it specified?

- **Other Aspects**
  - **Successor instruction**
    - How is it specified?
  - **Conditions**
    - How are they determined?
  - **Encodings**
    - Fixed or variable? Wide?
Basic ISA Classes

**Accumulator:**
- 1 address: add A  \[ \text{acc } \leftarrow \text{acc + mem[A]} \]
- 1+x address: addx A  \[ \text{acc } \leftarrow \text{acc + mem[A + x]} \]

**Stack:**
- 0 address: add  \[ \text{tos } \leftarrow \text{tos + next (JAVA VM)} \]

**General Purpose Register:**
- 2 address: add A B  \[ B \leftarrow B + A \text{ (Intel x86)} \]
- 3 address: add A B C  \[ C \leftarrow B + A \]

**Load/Store:** (MIPS, ARM)
- 3 address: add Ra Rb Rc  \[ \text{Ra } \leftarrow \text{Ra} + \text{Rc} \]
  - load Ra Rb  \[ \text{Ra } \leftarrow \text{mem[Rb]} \]
  - store Ra Rb  \[ \text{mem[Rb] } \leftarrow \text{Ra} \]

Adding Registers to an ISA

- A place to hold values that can be named within the instruction
- Like memory, but much smaller
  - 8-64 locations
- Intel operands in either memory or register
  - add A B
Intel x86 Processors

- Totally dominate desktop, laptop, & server computer market
  - Will it last?
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
  - Many quirks because of backward compatibility
- 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.

### 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>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>Pentium 4F</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
</tbody>
</table>
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

Presentation

- Book presents IA32 in Sections 3.1—3.12
- Covers x86-64 in 3.13
- We will cover both simultaneously, but mostly IA32
- Most labs will be based on IA32, may on occasion be x86-64
Machine Programming I: Basics

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

Stored Program Computer

- **Instructions**: a fixed set of built-in operations
- Instructions and data are stored in the (same) computer memory
- Fetch-Execute Cycle
  ```java
  while (!done)
    fetch instruction
    execute instruction
  ```
- This is usually done by the hardware for speed
- This is what the Java Virtual Machine does
What Must be Specified?

- Instruction Format
  - how do we tell what operation to perform?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches
- \textit{fetch-decode-execute is implicit!}

Turning C into Object Code

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

\begin{align*}
\text{text} & \quad \text{C program (p1.c p2.c)} \\
\text{text} & \quad \text{Compiler (gcc -S)} \\
\text{text} & \quad \text{Asm program (p1.s p2.s)} \\
\text{binary} & \quad \text{Assembler (gcc or as)} \\
\text{binary} & \quad \text{Object program (p1.o p2.o)} \\
\text{binary} & \quad \text{Linker (gcc or ld)} \\
\text{binary} & \quad \text{Executable program (p)} \\
\end{align*}

Static libraries (.a)
Assembly Programmer’s View

- **Programmer-Visible State**
  - PC: Program counter
    - Address of next instruction
    - Called “EIP” (IA32) or “RIP” (x86-64)
  - Register file
    - Frequently 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

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
  - Intel has some 80-bit floats

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

- Arithmetic & Logical
  - Perform arithmetic function on register or memory data

- Data Movement: 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

Compiling Into Assembly

C Code

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

Generated IA32 Assembly

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

Some compilers use instruction "leave"

Obtain with command

```
/usr/bin/gcc -O1 -S code.c
```

Produces file `code.s`
# Object (Machine) Code

**Code for sum**

```c
0x401040 <sum>:
  0x55
  0x89
  0xe5
  0x8b
  0x45
  0x0c
  0x03
  0x45
  0x08
  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

- **Machine Instruction Example**

```c
int t = x+y;
```

**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`
Disassembling Object Code

**Disassembled**

```
080483c4 <sum>:
  080483c4:  55        push   %ebp
  080483c5:  89 e5      mov    %esp,%ebp
  080483c7:  8b 45 0c    mov    0xc(%ebp),%eax
  080483ca:  03 45 08    add    0x8(%ebp),%eax
  080483cd:  5d        pop    %ebp
  080483ce:  c3        ret
```

- **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**

```
0x401040:  0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45 0x08 0x5d 0xc3
```

**Disassembled**

```
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?

% 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

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

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>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>source</td>
</tr>
<tr>
<td>%edi</td>
<td>index</td>
</tr>
<tr>
<td>%esp</td>
<td>destination</td>
</tr>
<tr>
<td>%ebp</td>
<td>base</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)

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><em>Reg</em></td>
<td><em>Reg</em></td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td><em>Mem</em></td>
<td><em>Reg</em></td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><em>Reg</em></td>
<td><em>Mem</em></td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td><em>Mem</em></td>
<td><em>Reg</em></td>
<td>movl (%eax),%edx</td>
<td>*p = temp;</td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction*

---

**Simple Memory Addressing Modes**

- **Normal (R)** \( \text{Mem[Reg[R]]} \)
  - Register \( R \) contains memory address

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

- **Displacement \( D(R) \)** \( \text{Mem[Reg[R]+D]} \)
  - Register \( R \) contains start of memory region
  - Constant displacement \( D \) specifies offset

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

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

Swap:

```assembly
pushl %ebp
movl %esp,%ebp
pushl %ebx

movl 8(%ebp), %edx
movl 12(%ebp), %ecx
movl (%edx), %ebx
movl (%ecx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)

popl %ebx
popl %ebp
ret
```

Understanding Swap

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>xp</td>
</tr>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Stack (in memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
<td>0x120</td>
<td>0x100</td>
<td></td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>0x124</td>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td>4</td>
<td>0x108</td>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x11c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

Understanding Swap

```
%eax  456
%edx  0x124
%ecx  0x120
%ebx  123
%esi  
%edi  
%esp  
%ebp  0x104

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

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

Complete Memory Addressing Modes

- **Most General Form**
  
  $$ D(Rb,Ri,S) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]+D] $$
  
  - **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**
  
  
  $(Rb,Ri) \rightarrow Mem[Reg[Rb]+Reg[Ri]]$
  
  $D(Rb,Ri) \rightarrow Mem[Reg[Rb]+Reg[Ri]+D]$
  
  $(Rb,Ri,S) \rightarrow Mem[Reg[Rb]+S*Reg[Ri]]$
Data Representations: IA32 + x86-64

- Sizes of C Objects (in Bytes)
  - C Data Type | Generic 32-bit | Intel IA32 | x86-64
  - unsigned     | 4            | 4          | 4
  - int          | 4            | 4          | 4
  - long int     | 4            | 4          | 8
  - char         | 1            | 1          | 1
  - short        | 2            | 2          | 2
  - float        | 4            | 4          | 4
  - double       | 8            | 8          | 8
  - long double  | 8            | 10/12      | 16
  - char *       | 4            | 4          | 8
    - Or any other pointer

x86-64 Integer Registers

- %rax       %eax
- %rbx       %ebx
- %rcx       %ecx
- %rdx       %edx
- %rsi       %esi
- %rdi       %edi
- %rsp       %esp
- %rbp       %ebp
- %r8        %r8d
- %r9        %r9d
- %r10       %r10d
- %r11       %r11d
- %r12       %r12d
- %r13       %r13d
- %r14       %r14d
- %r15       %r15d

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose
Instructions

- Long word $l$ (4 Bytes) $\leftrightarrow$ Quad word $q$ (8 Bytes)

- New instructions:
  - movl $\leftrightarrow$ movq
  - addl $\leftrightarrow$ addq
  - sall $\leftrightarrow$ salq
  - etc.

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: addl

32-bit code for swap

```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 8(%ebp), %edx
    movl 12(%ebp), %ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)

    popl %ebx
    popl %ebp
    ret
```
64-bit code for swap

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

- **Operands passed in registers (why useful?)**
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- **No stack operations required**
- **32-bit data**
  - Data held in registers %eax and %edx
  - `movl operation`

Body:
- `movl (%rdi), %edx`
- `movl (%rsi), %eax`
- `movl %eax, (%rdi)`
- `movl %edx, (%rsi)`

Finish:
- `ret`

---

64-bit code for long int swap

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

- **64-bit data**
  - Data held in registers %rax and %rdx
  - `movq operation`
    - “q” stands for quad-word
Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86 move instructions cover wide range of data movement forms
- Intro to x86-64
  - A major departure from the style of code seen in IA32