Machine-Level Programming I: Basics
Lecture 5

Instructor:
Alvin R. Lebeck

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

Admin
- Homework #2 Due Wednesday 11:59pm
- No Lebeck office hours today
- Reading: Chapter 3

Machine Programming Basics
- Instruction Set Architectures
- Brief History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Intro to x86-64
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`

```
  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)
```

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
”, 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  acc ← acc + mem[A]
1+x address addx A  acc ← acc + mem[A + x]

Stack:

0 address  add  tos ← tos + next (JAVA VM)

General Purpose Register:

2 address  add A B  B ← B + A (Intel x86)
3 address  add A B C  C ← B + A

Load/Store:  (MIPS, ARM)

3 address  add Ra Rb Rc  Ra ← Rb + Rc
load Ra Rb  Ra ← mem[Rb]
store Ra Rb  mem[Rb] ← 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

- 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>
</tbody>
</table>
  - First 16-bit processor. Basis for IBM PC & DOS
  - 1MB address space
| 386      | 1985 | 275K        | 16-33|
  - First 32 bit processor, referred to as IA32
  - Added “flat addressing”
  - Capable of running Unix
  - 32-bit Linux/gcc uses no instructions introduced in later models
| Pentium 4F | 2004 | 125M        | 2800-3800|
  - First 64-bit processor, referred to as x86-64
| Core i7   | 2008 | 731M        | 2667-3333|
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
- Some labs will be based on x86-64, others on IA32
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**
  
  ```
  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
- *fetch-decode-execute is implicit*

### 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`

```
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)
```
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

- 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`

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

- Total of 11 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040

- 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

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

- C Code
  - Add two signed integers

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

- 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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>080483c4</td>
<td>push %ebp</td>
</tr>
<tr>
<td>080483c5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>080483c7</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>080483ca</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>080483cd</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>080483ce</td>
<td>ret</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x401044</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>0x401047</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x40104a</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>0x40104d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x40104e</td>
<td>ret</td>
</tr>
</tbody>
</table>

Within gdb Debugger

```
gdb p
```

- 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)

- **%eax, %ax**: Accumulate
- **%ecx, %cx**: Counter
- **%edx, %dx**: Data
- **%ebx, %bx**: Base
- **%esi, %si**: Source index
- **%edi, %di**: Destination index
- **%esp, %sp**: Stack pointer
- **%ebp, %bp**: Base pointer

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>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>temp = *p;</td>
</tr>
</tbody>
</table>

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

---

**Simple Memory Addressing Modes**

- **Normal (R)**: Mem[Reg[R]]
  - Register R specifies memory address
    
    \[
    \text{movl} \thinspace (%ecx),%eax
    \]

- **Displacement D(R)**: Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
    
    \[
    \text{movl} \thinspace 8(%ebp),%edx
    \]
Using Simple Addressing Modes

\begin{verbatim}
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

Stack (in memory)

Register Value
%edx   xp
%ecx   yp
%ebx   t0
%eax   t1

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

Address
0x120
0x11c
0x118
0x114
0x110
0x10c
0x108
0x104
0x100
0x124
0x12c
0x130
0x134
0x138
0x13c
0x140
0x144
0x148
0x14c
0x150
0x154
0x158
0x15c
0x160
0x164
0x168
0x16c
0x170
0x174
0x178
0x17c
0x180
0x184
0x188
0x18c
0x190
0x194
0x198
0x19c
0x1a0
0x1a4
0x1a8
0x1ac
0x1b0
0x1b4
0x1b8
0x1bc
0x1c0
0x1c4
0x1c8
0x1cc
0x1d0
0x1d4
0x1d8
0x1dc
0x1e0
0x1e4
0x1e8
0x1ec
0x1f0
0x1f4
0x1f8
0x1fc
0x200
Understanding Swap

\[
\begin{align*}
%eax & \quad \text{Address} \\
%edx & 0x124 \\
%ecx & 0x120 \\
%ebx & \quad \text{Offset} \\
%esi & \\
%edi & \\
%esp & \\
%ebp & 0x104 \\
movl & 8(\%ebp), \%edx \quad \# \%edx = xp \\
movl & 12(\%ebp), \%ecx \quad \# \%ecx = yp \\
movl & (\%edx), \%ebx \quad \# \%ebx = *xp (t0) \\
movl & (\%ecx), \%eax \quad \# \%eax = *yp (t1) \\
movl & \%eax, (\%edx) \quad \# *xp = t1 \\
movl & \%ebx, (\%ecx) \quad \# *yp = t0
\end{align*}
\]
Understanding Swap

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

```
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
```

Address

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
</tr>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
</tbody>
</table>

%ebp

%edx 0x124
%ecx 0x120
%ebx 123
%esi
%edi
%esp
%ebp 0x104
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
  \[
  \begin{align*}
  (Rb, Ri) & \rightarrow Mem[Reg[Rb]+Reg[Ri]] \\
  D(Rb, Ri) & \rightarrow Mem[Reg[Rb]+Reg[Ri]+D] \\
  (Rb, Ri, S) & \rightarrow Mem[Reg[Rb]+S*Reg[Ri]]
  \end{align*}
  \]

Data Representations: IA32 + x86-64

- Sizes of C Objects (in Bytes)
<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Generic 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
x86-64 Integer Registers

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

Instructions

- Long word 1 (4 Bytes) ↔ Quad word q (8 Bytes)

- New instructions:
  - movl → movq
  - addl → addq
  - sall → 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;
}
```

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`
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