MACHINE-LEVEL PROGRAMMING I: BASICS

COMPUTER ARCHITECTURE AND ORGANIZATION
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
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
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
Today: 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.
  • 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
Program to Process

- We write a program in e.g., C.
- A compiler turns that program into an instruction list.
- The CPU interprets the instruction list (which is more a graph of basic blocks).

```c
void X (int b) {
    if(b == 1) {
        ...
    }
    int main() {
        int a = 2;
        X(a);
    }
```
Process in Memory

- Program to process.
  - What you wrote
    ```c
    void X (int b) {
        if(b == 1) {
            ...
        }
        int main() {
            int a = 2;
            X(a);
        }
    }
    ```
  - What must the OS track for a process?
  - What is in memory.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
<th>Code</th>
</tr>
</thead>
</table>
| main; a = 2     |                               | void X (int b) {
|                 |                               |   if(b == 1) {
|                 | X; b = 2                      |       ...
|                 |                               | int main() {
|                 |                               |   int a = 2;
|                 |                               |   X(a);
|                 |                               | }
```
A shell forks and execs a calculator

```c
int pid = fork();
if(pid == 0) {
    close(".history");
    exec("/bin/calc");
} else {
    wait(pid);
}
```
A shell forks and then execs a calculator

```c
int shell_main() {
    int a = 2;
    ...
}
```

```c
int calc_main() {
    int q = 7;
    ...
}
```

pid = 128
open files = ".history"
lst_cpu = 0

Process Control Blocks (PCBs)
Anatomy of an address space

Process's address space

Executable File

Header

Code

Initialized data

mapped segments

DLL's

Stack

Heap

Initialized data

Code

Inaccessible
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)

binary
  Object program (p1.o  p2.o)
  Linker (gcc or ld)

binary
  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

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

Produces file code.s
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  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 11 bytes**
- **Each instruction 1, 2, or 3 bytes**
- **Starts at address 0x401040**
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 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

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

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
disable 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

```bash
% 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
```
Today: 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>General Purpose</th>
<th>Origin (mostly obsolete)</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 index</td>
</tr>
<tr>
<td>%edi</td>
<td>destination index</td>
</tr>
<tr>
<td>%esp</td>
<td>stack</td>
</tr>
<tr>
<td>%ebp</td>
<td>pointer</td>
</tr>
<tr>
<td>%esp</td>
<td>base</td>
</tr>
<tr>
<td>%ebp</td>
<td>pointer</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)
Simple Memory Addressing Modes

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

  \texttt{movl \%ecx,\%eax}

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

  \texttt{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:
    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
```

Set Up

Body

Finish
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 8(%ebp), %edx
movl 12(%ebp), %ecx
movl (%edx), %ebx
movl (%ecx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)
popl %ebx
popp %ebp
ret
```
Understanding Swap

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

### Register Value

<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>

### Stack (in memory)

```
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
%edx
%ecx
%ebx
%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

Offset

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x10c</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x114</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x124</td>
</tr>
</tbody>
</table>

Address

123
456

%ebp 0

yp 12
xp 8
Rtn adr

%ebp → 0

-4
Understanding Swap

%eax 0x124
%edx 0x104
%ecx 0x120
%ebx 0x11c
%esi 0x118
%edi 0x114
%esp 0x110
%ebp 0x10c

Address 0x120 0x124 0x114 0x118 0x10c 0x104 0x100
Offset 0x120 0x124 0x114 0x110 0x10c 0x104 0x100

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12 0x120</td>
</tr>
<tr>
<td>xp</td>
<td>8 0x124</td>
</tr>
<tr>
<td></td>
<td>4 0x10c</td>
</tr>
<tr>
<td></td>
<td>-4 0x100</td>
</tr>
</tbody>
</table>

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

| %eax   |       | %edx   | 0x124 |
| %edx   | 0x120 | %ecx   |       |
| %ecx   |       | %ebx   | 123   |
| %ebx   |       | %esi   |       |
| %esi   |       | %edi   |       |
| %edi   |       | %esp   |       |
| %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

<table>
<thead>
<tr>
<th>%eax</th>
<th>456</th>
</tr>
</thead>
<tbody>
<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>

<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>0x110</td>
</tr>
<tr>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td>-4</td>
<td>0x104</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

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 |

### Offset

| yp  | 12 | 0x120 |
| xp  | 8  | 0x124 |
|     | 4  | Rtn adr |
| %ebp | 0  | 0x108 |
|      | -4 | 0x104 |

### Address

0x124
0x120
0x11c
0x118
0x114
0x110
0x10c
0x108
0x104
0x100

### Code Snippet

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

\[
\begin{align*}
\%eax & \quad 456 \\
\%edx & \quad 0x124 \\
\%ecx & \quad 0x120 \\
\%ebx & \quad 123 \\
\%esi & \quad \\
\%edi & \quad \\
\%esp & \quad \\
\%ebp & \quad 0x104 \\
\end{align*}
\]

\[
\begin{align*}
\text{movl} & \quad 8(\%ebp), \%edx \quad \# \text{ edx } = \text{ xp} \\
\text{movl} & \quad 12(\%ebp), \%ecx \quad \# \text{ ecx } = \text{ yp} \\
\text{movl} & \quad (\%edx), \%ebx \quad \# \text{ ebx } = *\text{xp} \ (t0) \\
\text{movl} & \quad (\%ecx), \%eax \quad \# \text{ eax } = *\text{yp} \ (t1) \\
\text{movl} & \quad \%eax, (\%edx) \quad \# \ *\text{xp} = t1 \\
\text{movl} & \quad \%ebx, (\%ecx) \quad \# \ *\text{yp} = t0
\end{align*}
\]
Complete Memory Addressing Modes

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

- **Special Cases**
  - \( (R_b, R_i) \text{ Mem}[Reg[R_b] + Reg[R_i]] \)
  - \( D(R_b, R_i) \text{ Mem}[Reg[R_b] + Reg[R_i] + D] \)
  - \( (R_b, R_i, S) \text{ Mem}[Reg[R_b] + S*Reg[R_i]] \)
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

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