Machine-Level Programming II: Introduction

Topics
- Addresses and Pointers
- Memory address modes
- Arithmetic operations
- RISC vs. CISC
Addresses and Pointers in C

C programming model is close to machine language

- Machine language manipulates memory addresses
  - Address computation
  - Store addresses in registers or memory
- C employs pointers, which are just addresses of primitive data elements or data structures

Examples of operators * and &

- int a, b; /* declaration of a and b as an integers */
- int *a_ptr; /* a is a pointer to an integer (address of memory)
- a_ptr = a; /* illegal as the types don’t match */
- a_ptr = &a; /* a_ptr holds address of “a” */
- b = *a_ptr; /* dereference (lookup) value at address a_ptr and assign value to b */
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
```
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

Register | Variable
---|---
%ecx | yp
%edx | xp
%eax | t1
%ebx | t0
Understanding Swap

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

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

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</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></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</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

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

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
</tr>
<tr>
<td>0x124</td>
</tr>
<tr>
<td>456</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x11c</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x114</td>
</tr>
<tr>
<td>Rtn adr</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>YP 12</td>
</tr>
<tr>
<td>xp 8</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>%ebp 0</td>
</tr>
<tr>
<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

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</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>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</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

## Memory Layout

<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>0x110</td>
<td>0x108</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
<td>0x124</td>
<td>0x10c</td>
<td>0x104</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
<td></td>
<td></td>
<td>0x100</td>
</tr>
<tr>
<td>0x114</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>-4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Registers

<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>

## Code Snippet

```plaintext
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
```
Indexed Addressing Modes

Most General Form

\[ D(Rb,Ri,S) \quad Mem[Reg[Rb]+S*Reg[Ri]+D] \]

- **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

Special Cases

- \((Rb,Ri)\) \quad Mem[Reg[Rb]+Reg[Ri]]
- \(D(Rb,Ri)\) \quad Mem[Reg[Rb]+Reg[Ri]+D]
- \((Rb,Ri,S)\) \quad Mem[Reg[Rb]+S*Reg[Ri]]
# Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%edx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0xe080</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecx</td>
<td>0x100</td>
</tr>
</tbody>
</table>

Address Computation Instruction

leal \textit{Src},\textit{Dest}

- \textit{Src} is address mode expression
- Set \textit{Dest} to address denoted by expression

Uses

- Computing address without doing memory reference
  - E.g., translation of \texttt{p = \&x[i];}
- Computing arithmetic expressions of the form \texttt{x + k*y}
  - \texttt{k = 1, 2, 4, or 8.}
## Some Arithmetic Operations

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Two Operand Instructions</strong></td>
<td></td>
</tr>
<tr>
<td><code>addl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest - Src</code></td>
</tr>
<tr>
<td><code>imull</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest * Src</code></td>
</tr>
<tr>
<td><code>sall</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code></td>
</tr>
<tr>
<td><code>sar1</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>shrl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest &gt;&gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>xorl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest ^ Src</code></td>
</tr>
<tr>
<td><code>andl</code> <code>Src, Dest</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
</tr>
<tr>
<td><code>orl</code> <code>Src, Dest</code></td>
<td>`Dest = Dest</td>
</tr>
</tbody>
</table>
## Some Arithmetic Operations

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>One Operand Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>incl Dest</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td>decl Dest</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td>negl Dest</td>
<td>Dest = - Dest</td>
</tr>
<tr>
<td>notl Dest</td>
<td>Dest = ~ Dest</td>
</tr>
</tbody>
</table>
Using `leal` for Arithmetic Expressions

```c
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

**arith:**

```c
pushl %ebp
movl %esp,%ebp

movl 8(%ebp),%eax
movl 12(%ebp),%edx
leal (%edx,%eax),%ecx
leal (%edx,%edx,2),%edx
sall $4,%edx
addl 16(%ebp),%ecx
leal 4(%edx,%eax),%eax
imull %ecx,%eax

movl %ebp,%esp
popl %ebp
ret
```

```c
jmp Set Up
```
Understanding arith

```c
int arith
    (int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```
movl 8(%ebp),%eax  # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx  # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx  # edx = 48*y (t4)
addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
```
Understanding arith

```c
int arith
    (int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```assembly
# eax = x
    movl 8(%ebp),%eax
# edx = y
    movl 12(%ebp),%edx
# ecx = x+y  (t1)
    leal (%edx,%eax),%ecx
# edx = 3*y
    leal (%edx,%edx,2),%edx
# edx = 48*y (t4)
    sall $4,%edx
# ecx = z+t1 (t2)
    addl 16(%ebp),%ecx
# eax = 4+t4+x (t5)
    leal 4(%edx,%eax),%eax
# eax = t5*t2 (rval)
    imull %ecx,%eax
```
Another Example

```c
int logical(int x, int y)
{
    int t1 = x ^ y;
    int t2 = t1 >> 17;
    int mask = (1 << 13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

2<sup>13</sup> = 8192, 2<sup>13</sup> − 7 = 8185

```
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

```
eax = x
```

```
eax = x ^ y  \quad (t1)
```

```
eax = t1 >> 17  \quad (t2)
```

```
eax = t2 \& 8185
```

```
set up
```

```
body
```

```
finish
```
ISA Properties

CISC - Complex Instruction Set Computer (e.g. x86)

- Instruction can reference different operand types
  - Immediate, register, memory
- Arithmetic operations can read/write memory
- Memory reference can involve complex computation
  - $R_b + S \times R_i + D$
  - Useful for arithmetic expressions, too
- Instructions can have varying lengths
  - x86 IA32 instructions can range from 1 to 15 bytes

“RISC” - Reduced Instruction Set Computer

- e.g. ARM, PowerPC, Sparc
- Memory operations separate from arithmetic (load/store)
- Fixed length instructions (often 4 bytes each)
- Fewer complex computational instructions (e.g. string compare)
Summary

Today
- C and x86 memory addressing
- Arithmetic instructions

Next Time
- Control instructions (branch, etc.)
Extra slides
Pentium Pro (P6)

History
- Announced in Feb. ‘95
- Basis for Pentium II, Pentium III, and Celeron processors
- Pentium 4 similar idea, but different details

Features
- Dynamically translates instructions to more regular format
  - Very wide, but simple instructions
- Executes operations in parallel
  - Up to 5 at once
- Very deep pipeline
  - 12–18 cycle latency
Pentium Pro Block Diagram

Microprocessor Report
2/16/95
PentiumPro Operation

Translates instructions dynamically into “Uops”
- 118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine
- Uop executed when
  - Operands available
  - Functional unit available
- Execution controlled by “Reservation Stations”
  - Keeps track of data dependencies between uops
  - Allocates resources

Consequences
- Indirect relationship between IA32 code & what actually gets executed
- Tricky to predict / optimize performance at assembly level