x86 basics

ISA context and x86 history

Translation:
Compile C → machine code
Disassemble machine code

x86 Basics:
Registers
Data movement instructions
Memory addressing modes
Arithmetic instructions

Devices (transistors, etc.)
Solid-State Physics

ISA View

ISA View

a brief history of x86

<table>
<thead>
<tr>
<th>ISA</th>
<th>First</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>Intel 8086</td>
<td>1978</td>
</tr>
<tr>
<td>32</td>
<td>Intel 386</td>
<td>1985</td>
</tr>
<tr>
<td>64</td>
<td>AMD Opteron</td>
<td>2003*</td>
</tr>
</tbody>
</table>

First 16-bit processor. Basis for IBM PC & DOS
1MB address space

First 32-bit ISA. Flat addressing, improved OS support

Slow AMD/Intel conversion, slow adoption.
*Not actually x86-64 until few years later. Mainstream only after ~10 years.
**Turning C into Machine 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 machine code in file p

- **text**
  - C program (p1.c p2.c)
  - Compiler (gcc -S)
- **text**
  - Asm program (p1.s p2.s)
  - Assembler (gcc or as)
- **binary**
  - Object program (p1.o p2.o)
  - Linker (gcc or ld)
  - Static libraries (.a)
- **binary**
  - Executable program (p)

---

**Disassembling Object Code (objdump)**

Disassembled by objdump

```
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
```

```
Disassembler
```

```
objdump -d p
```

---

**Disassembling Object Code (gdb)**

```
Object
```

```
Disassembled by GDB
```

```
Within gdb debugger
```

```
gdb p
disassemble sum
(disassemble function)
x/13b sum
(examine the 13 bytes starting at sum)
```

---

**C Code**

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

**Generated IA32 Assembly Code**

Human-readable language close to machine code.

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

```
code.s
code.c
```

```
### IA32: Three Basic Kinds of Instructions

1. **Data movement** between memory and register
   - **Load** data from memory into register
     \[
     \%\text{reg} = \text{Mem}[\text{address}] 
     \]
   - **Store** register data into memory
     \[
     \text{Mem}[\text{address}] = \%\text{reg} 
     \]
   
   *Remember: memory is indexed just like an array[] of bytes!*

2. **Arithmetic** on register or memory data
   \[
   c = a + b; \quad z = x \ll y; \quad i = h \& g; 
   \]

3. **Control flow** to choose what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

### Data movement instructions

- **`movx` Source, Dest**
  - \( x \) is one of \{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”

*Historical terms from the 16-bit days, not the current machine word size*
Data movement instructions

**movl Source, Dest:**

**Operand Types:**

- **Immediate:** Literal integer data
  - Examples: $0x400, -533$
  - Encoded with 1, 2, or 4 bytes
- **Register:** One of 8 integer registers
  - Examples: `%eax, %edx`
  - `%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>var_a = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147,(%eax)</td>
<td>*p_a = -147;</td>
</tr>
</tbody>
</table>

**movl**

| Reg    | Reg | movl %eax,%edx | var_d = var_a; |
| Mem    | Reg | movl %eax,(%edx) | *p_d = var_a; |
| Mem Reg| Reg | movl (%eax),%edx | var_d = *p_a; |

*Cannot do memory-memory transfer with a single instruction. How would you do it?*

---

**Basic Memory Addressing Modes**

**Indirect (R) Mem[Reg[R]]**

- Register R specifies the memory address
  - `movl (%ecx),%eax`

**Displacement D(R) Mem[Reg[R]+D]**

- Register R specifies a memory address
  - (e.g. the start of some memory region)
  - Constant displacement D specifies the offset from that address
  - `movl 8(%ebp),%edx`

---

**Using Basic Addressing Modes**

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

**swap:**

- **Set Up**
  - `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 %edx,%ecx`

- **Body**
  - `movl -4(%ebp),%ebx`
  - `movl %ebp,%esp`
  - `popl %ebp`

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

Understanding Swap

Stack (in memory)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</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>Return addr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebp</td>
</tr>
</tbody>
</table>

Register Value

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

register <-> variable mapping

Understanding Swap

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

Stack (in memory)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</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>Return addr</td>
</tr>
<tr>
<td>0</td>
<td>%ebp</td>
</tr>
<tr>
<td>-4</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

Register Contents

%edx  0xf000
%ecx  0x100
%ebx  0x104
%ebp  0x100

Address Computation Examples

<table>
<thead>
<tr>
<th>Register Contents</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>0xf000</td>
<td>0x103</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x100</td>
<td>0x104</td>
</tr>
<tr>
<td>(%edx, %ecx)</td>
<td></td>
<td>0x105</td>
</tr>
<tr>
<td>(%edx, %ecx, 4)</td>
<td></td>
<td>0x109</td>
</tr>
<tr>
<td>0x8000(, %edx, 2)</td>
<td></td>
<td>0x10c</td>
</tr>
</tbody>
</table>

Complete Memory Addressing Modes

Memory addresses can be computed in several different ways.

Most General Form:

\[ D(Rb,Ri,S) \]  
\[ \text{Mem}[\text{Reg}[Rb] + S \times \text{Reg}[Ri] + D] \]

- **D**: Constant "displacement" value represented in 1, 2, or 4 bytes
- **Rb**: Base register: Any register
- **Ri**: Index register: Any except %esp; %ebp unlikely
- **S**: Scale: 1, 2, 4, or 8 (why these numbers?)

Special Cases: can use any combination of D, Rb, Ri and S

- \((Rb,Ri)\)  
  \[ \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \]
  \[ (S=1,D=0) \]
- \(D(Rb,Ri)\)  
  \[ \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] \times D] \]
  \[ (S=1) \]
- \((Rb,Ri,S)\)  
  \[ \text{Mem}[\text{Reg}[Rb] + S \times \text{Reg}[Ri]] \]
  \[ (D=0) \]
leal Src, Dest  
(load effective address)

Src is address mode expression
Set Dest to address computed by expression
Example: leal (%edx,%ecx,4), %eax

!!!

 DOES NOT ACCESS MEMORY 

Uses
Computing addresses, e.g.: translation of p = &x[i];
Computing arithmetic expressions of the form x + k*i
  k = 1, 2, 4, or 8

Arithmetic Operations

Two-operand instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subl</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imull</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>shll</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andl</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

No distinction between signed and unsigned int (why?)
except arithmetic vs. logical shift right

Arithmetic (IA32)

```
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;
}
```
Understanding arith (IA32)

```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;
}
```

Observations about arith

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Same x86 code by compiling: \((x+y+z) \times (x+4+48*y)\)

Another Example (IA32)

```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;
}
```