15-213
“The course that gives CMU its Zip!”

Machine-Level Programming I: Introduction
Jan. 22, 2008

Topics
- Assembly Programmer’s Execution Model
- Accessing Information
- Registers
- Memory
- Arithmetic operations

Outline

Some computer languages
- Whitespace
- Intercal
- M

Some discussion of x86, x86-64
- Warning: Chapter 3 doesn’t compress well
  - 100 pages of discussion about machine language
  - After 75 pages of data representation in Chapter 2
- Please plan to spend time reading the text!

Whitespace “Explained”

<table>
<thead>
<tr>
<th>Statement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Space][Space][Space] [Tab][LF]</td>
<td>Push 1 onto stack</td>
</tr>
<tr>
<td>[LF][Space][Space][Space] [Space][Tab][Tab][LF]</td>
<td>Set a label at this point</td>
</tr>
<tr>
<td>[Space][LF][Space]</td>
<td>Duplicate the top stack item</td>
</tr>
<tr>
<td>[Tab][LF][Space][Tab]</td>
<td>Output the current value</td>
</tr>
</tbody>
</table>

INTERCAL

Features of INTERCAL
- Designed late one night in 1972 by two Princeton students
- Deliberately obfuscated language

Variables
- 16-bit integers, 1 through .65535
- 32-bit integers, 1 through .65535

Operators
- Binary: “mingle”, “select”
  - Unary: AND, OR, XOR
  - How are those unary???
  - Simple: AND and’s together adjacent bits in a word
  - Simplest way to put .65536 in a 32-bit variable?
  - DO I <= .65536

A Whitespace Program

“Count from 1 to 10” (partial listing)

Features of Whitespace
- Only space, tab, and line-feed encode program statements
- All other characters (A-Z, a-z, 0-9, etc.) encode comments
- Simple stack-based language

Synchronization

Lab 1
- Time roughly 50% done
- Many have started early and made good progress
- Good
- Warning to others...
  - This isn’t the same kind of thing you’ve done before
  - Please don’t leave it to the last minute

Fish-machine log-ins
- Please let us know (staff mailing list) if you can’t log in to any machine
The language “M”

Features of M
- Also designed in the 1970’s
- More widely used than Whitespace, INTERCAL

Variables
- 32-bit integer variables: A, B, C, D, DI, F, S, SI
- One array, M[
- Valid subscripts range from near zero to a large number
- But most subscripts in that range will crash your program!

Statements
- Lots of arithmetic and logical operations
- Input and output use a special statement called OUCH!

A Program in M

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

M
```
   M
```
```
   A = M[S+4]
   A += M[S+8]
```
```
   DONE
```

Had enough of M?
- Too bad! We’ll study it for much of the semester!
- Why???

M is “The Language of the Machines”

IA32 Processors
Totaly Dominate Computer Market

Evolutionary Design
- Starting in 1978 with 8086
- Added more features as time goes on
- Still support old features, although obsolete

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!
### x86 Evolution: Programmer’s View (Abbreviated)

**Name**  
8086  
386  
486  
Pentium 1  
Pentium 2  
Pentium 3  
Pentium 4

**Date**  
1978  
1985  
1989  
1993  
1995  
1999  
2001

**Transistors**  
29K  
275K  
1.9M  
3.1M  
6.5M  
8.2M  
42M

- **8086**: 16-bit processor. Basis for IBM PC & DOS  
- **386**: Extended to 32 bits. Added “flat addressing”.  
- **486**: Pentium, Pentium-MMX, Pentium-Pro, Pentium III, Pentium 4

**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

### Machine Evolution

- **486**: 1989  
- **Pentium**: 1993  
- **Pentium-MMX**: 1997  
- **Pentium-Pro**: 1995  
- **Pentium III**: 1999  
- **Pentium 4**: 2001

- **Pentium 2**: 2002  
- **Itanium 2 Dual-Core**: 2006

- **Itanium**: 2001  
- **Itanium 2**: 2002

**Itanium**
- Extends to IA64, a 64-bit architecture  
- Radically new instruction set designed for high performance  
- Can run existing IA32 programs  
- On-board “x86 engine”  
- Joint project with Hewlett-Packard

**Itanium 2**
- Big performance boost

**Itanium 2 Dual-Core**
- 2006  
- 1.7B

**Itanium** has not taken off in marketplace  
- Lack of backward compatibility

### Intel’s 64-Bit Dilemma

- **Totally different architecture**  
- **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  
- Our Saltwater fish machines

### Our Coverage

- **IA32**
  - The standard x86
  - The emerging standard

- **Presentation**  
- Book has IA32

- **Handout** has x86-64

- **Lecture** will cover both Labs  
  - Lab #2 x86-64  
  - Lab #3 IA32

- **New Species: IA64**  
- Extends to IA64, a 64-bit architecture  
- Radically new instruction set designed for high performance  
- Can run existing IA32 programs  
- On-board “x86 engine”  
- Joint project with Hewlett-Packard

- **Itanium 2**: 2002  
- **Itanium 2 Dual-Core**: 2006

**Itanium** has not taken off in marketplace  
- Lack of backward compatibility

### X86 Evolution: Clones

- **Advanced Micro Devices (AMD)**  
  - Historically  
  - AMD has followed just behind Intel  
  - A little bit slower, a lot cheaper  
  - Recently  
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies  
  - Exploited fact that Intel distracted by IA64  
  - Now are close competitors to Intel

- **Developed x86-64, its own extension to 64 bits**  
  - Started eating into Intel’s high-end server market

### Linux/GCC Evolution

- **None!**
Assembly Programmer’s View

Programmer-Visible State

- **CPU**: Registers, Instruction Code
- **Memory**: Code, Data
- **Object Code**: Segments, Sections
- **Program Data**: Global Variables
- **OS Data**: System Calls
- **Stack**: Local Variables
- **Condition Codes**: Overflow, Carry

CPU

- **Registers**: PC, EIP, EBP, ESP, EAX, EBX, ECX, EDX
- **PC**: Program Counter
- **Register File**: Used for data manipulation
- **Condition Codes**: Store status information from previous instructions

Turning C into Object Code

- **Compiler** (`gcc -S`)
- **Assembler** (`gcc` or `as`)
- **Linker** (`gcc` or `ld`)

C program (`p1.c p2.c -o p1.o`)

- Asm program (`p1.s p2.s`)
- Object program (`p1.o p2.o`)
- Executable program (`p1.o`)

Linking

- Resolves references between files
- Combines with static run-time libraries
- Libraries are dynamically linked

Assembly Characteristics

- **Minimal Data Types**
  - “Integer” data of 1, 2, or 4 bytes
  - Addresses (untyped pointers)
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures

- **Primitive Operations**
  - Arithmetic functions on register or memory data
  - Transfer data between memory and register
  - Load data from memory into register
  - Store data from register to memory
  - Transfer control
    - Unconditional jumps to/from procedures
    - Conditional branches

Machine Instruction Example

- **C Code**
  - `int t = x+y;`
  - `t += y`

- **Object Code**
  - 3-byte instruction
  - Stored at address `0x401046`

- **Assembly**
  - `addl 8(%ebp),%eax`
  - Similar to expression: `x += y`

- **Linker**
  - Resolves references between files
  - Combines with static run-time libraries
  - Linking occurs when program begins execution

Compiling Into Assembly

- **C Code**
  - `int sum(int x, int y) {
    int t = x+y;
    return t;
  }`

- **Generated IA32 Assembly**
  - `pushl %ebp`
  - `movl %esp,%ebp`
  - `movl 12(%ebp),%eax`
  - `addl 8(%ebp),%eax`
  - `movl %ebp,%esp`
  - `popl %ebp`
  - `ret`

Obtain with command `gcc -O -S code.c`

Produces file `code.s`
Disassembling Object Code

```
Disassembled

0x401040 <sum>: push %ebp
0x401041 <sum+1>: mov %esp,%ebp
0x401043 <sum+3>: mov 0xc(%ebp),%eax
0x401046 <sum+6>: add 0x8(%ebp),%eax
0x401049 <sum+9>: mov %ebp,%esp
0x40104b <sum+11>: pop %ebp
0x40104c <sum+12>: ret
0x40104d <sum+13>: lea 0x0(%esi),%esi
```

Disassembler
```
objdump -d /glyph1
```

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

What Can be Disassembled?

```
1-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

Alternate Disassembly

Within gdb Debugger
```
gdb p disassemble sum
```

Disassemble procedure
```
x/13b sum
```
Examine the 13 bytes starting at `sum`

Moving Data: IA32

```
Operand Combinations

Source Dest Src, Dest C Analog

- imm
- Reg
- Mem

movl

C Analog

```
```
Simple Addressing Modes

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

Displacement D(R) Mem[Reg[R]+D]
- Register R specifies 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;
}
```

Body

Set Up

 masked stack [esp, 4 bytes] BP

  esp  ebp

Body

Finish

Understanding Swap

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

Stack

```
  y
  x
  t0
  t1
```

Offset

```
  12
  8
  4
  0
```

Register Variable

```
%eax
t0
%edx
%edi
%esi
%ebp
%esp
```

Address

```
0x100
0x104
0x108
0x110
0x114
0x118
0x120
0x124
```

```
Address
123
124
125
126
127
128
129
130
131
```

```
movl 12(%ebp), %ecx # ecx = yp
movl 8(%ebp), %edx # edx = ap
movl (%ecx), %eax # eax = *yp (t1)
```

```
movl %eax, (%edx) # *xp = eax
movl %ebx, (%ecx) # *yp = ebx
```

```
movl -4(%ebp), %ebx # ebx = *xp
movl %ebp, %esp
```

```
popl %ebp
ret
```
Indexed Addressing Modes

Most General Form

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

Address Computation Instruction

leal Src, Dest

- Src is address mode expression
- Set Dest to address denoted by expression

Uses

- Computing addresses without a memory reference
  - E.g., translation of p = &x[i];
- Computing arithmetic expressions of the form x + k’y
  - k = 1, 2, 4, or 8.

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(%eax,%edi,4)</td>
<td>0xe000 + 4*0x100</td>
<td>0xe400</td>
</tr>
<tr>
<td>(%esi,%edi,4)</td>
<td>0xe000 + 4*0x100</td>
<td>0xe400</td>
</tr>
<tr>
<td>0x100(%esi,%edi,2)</td>
<td>2*0xe000 + 0x100</td>
<td>0xe200</td>
</tr>
</tbody>
</table>

Some Arithmetic Operations

Format  Computation

Two-operand Instructions

addl Src, Dest  Dest = Dest + Src

subl Src, Dest  Dest = Dest - Src

imull Src, Dest  Dest = Dest * Src

addl Src, Dest  Dest = Dest + Src

andi Src, Dest  Dest = Dest & Src

ori Src, Dest  Dest = Dest | Src

xorl Src, Dest  Dest = Dest ^ Src

shrl Src, Dest  Dest = Dest >> SRC

negl Dest  Dest = - Dest

notl Dest  Dest = ~ Dest

Using leal for Arithmetic Expressions

```c
int arith(int x, int y, int z)
{
    int t1 = x + y;
    int t2 = t1 + 1;
    int t3 = 4 * y;
    int t5 = t3 + t4;
    int real = t2 * t5;
    return real;
}
```

Body

Set Up

Dispose of Registers

Finish

```c
// leal (%edx,%esi,4), %edi
```

```c
leal 4(%edx,%esi), %edi
```

```c
leal (%edi,%edi,2), %edi
```

```c
sall $4,%edi
```

```c
addl 16(%ebp),%edi
```

```c
leal 4(%edx,%esi),%esi
```

```c
imull %edi,%esi
```

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

```c
popl %ebp
```

```c
ret
```
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;
}
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;
}
```

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

### Body

```
movl 8(%ebp),%eax      ; eax = x
xorl 12(%ebp),%eax     ; eax = x^y (t1)
sarl $17,%eax          ; eax = t1>>17 (t2)
andl $8185,%eax        ; eax = t2 & 8185  (rval)
```

### Set Up

```
movl $8(%ebp),%eax     ; eax = x
movl $12(%ebp),%eax    ; eax = x^y (t1)
movl $17(%ebp),%eax    ; eax = t1>>17 (t2)
andl $8185,%eax        ; eax = t2 & 8185  (rval)
```

### Finish

```
movl $8(%ebp),%eax     ; eax = x
movl $12(%ebp),%eax    ; eax = x^y (t1)
movl $17(%ebp),%eax    ; eax = t1>>17 (t2)
andl $8185,%eax        ; eax = t2 & 8185  (rval)
```

---

### Data Representations: IA32 + x86-64

**Sizes of C Objects (in Bytes)**

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unaligned</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>2</td>
<td>2</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>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>long double</td>
<td>0</td>
<td>10/12</td>
<td>10</td>
</tr>
<tr>
<td>char*</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

* Or any other pointer

---

### x86-64 General Purpose Registers

```
[Table showing x86-64 registers with different types and sizes, including `rax`, `rdx`, `rcx`, `rbx`, `rsi`, `rdi`, `rsp`, `rbp`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`, `r8d`, `r9d`, `r10d`, `r11d`, `r12d`, `r13d`, `r14d`, `r15d`]
```

* Extend existing registers. Add 8 new ones.
Make `ebp`/`vbp` general purpose
Swap in 32-bit Mode

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

Swap in 64-bit Mode

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

Swap Long Ints in 64-bit Mode

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

Summary

Machine Level Programming
- Assembly code is textual form of binary object code
- Low-level representation of program
- Explicit manipulation of registers
- Simple and explicit instructions
- Minimal concept of data types
- Many C control constructs must be implemented with multiple instructions

Formats
- IA32: Historical x86 format
- x86-64: Big evolutionary step

Credits
Languages
- http://compsoc.dur.ac.uk/whitespace/
- http://catb.org/~esr/intercal/

Screen shots
- Eclipse shot courtesy of Roy Liu, http://hubris.ucsd.edu