Machine-Level Programming I: Introduction

Dr. Steve Goddard
goddard@cse.unl.edu

http://cse.unl.edu/~goddard/Courses/JDEP284

Giving credit where credit is due

- Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
- I have modified them and added new slides.

Topics

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

IA32 Processors

Totally 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
</tr>
<tr>
<td>80286</td>
<td>1982</td>
<td>134K</td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
</tr>
</tbody>
</table>

- 16-bit processor. Basis for IBM PC & DOS
- Limited to 1MB address space. DOS only gives you 640K
- Added elaborate, but not very useful, addressing scheme
- Basis for IBM PC-AT and Windows
- Extended to 32 bits. Added “flat addressing”
- Capable of running Unix
- Linux/gcc uses no instructions introduced in later models

X86 Evolution: Programmer’s View

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>486</td>
<td>1989</td>
<td>1.9M</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>3.1M</td>
</tr>
<tr>
<td>Pentium/MMX</td>
<td>1997</td>
<td>4.5M</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>1995</td>
<td>6.5M</td>
</tr>
</tbody>
</table>

- Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data
- Added conditional move instructions
- Big change in underlying microarchitecture
X86 Evolution: Programmer’s View

Name          Date       Transistors
------------------------
Pentium III    1999       8.2M
- Added “streaming SIMD” instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or floating point data
- Our fish machines

Pentium 4      2001       42M
- Added 8-byte formats and 144 new instructions for streaming SIMD mode

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.
  - Exploited fact that Intel distracted by IA64
  - Now are close competitors to Intel
  - Developing own extension to 64 bits

X86 Evolution: Clones

Transmeta
- Recent start-up
  - Employer of Linus Torvalds
- Radically different approach to implementation
  - Translates x86 code into “Very Long Instruction Word” (VLIW) code
  - High degree of parallelism
  - Shooting for low-power market

New Species: IA64

Itanium        2001       10M
- Extends to IA64, a 64-bit architecture
- Radically new instruction set designed for high performance
- Will be able to run existing IA32 programs
  - On-board “x86 engine”
- Joint project with Hewlett-Packard

Itanium 2      2002       221M
- Big performance boost

Assembly Programmer’s View

Programmer-Visible State
- EIP  Program Counter
  - Address of next instruction
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

Memory
- Addresses
- Data
- Instructions
- Object Code
- Program Data
- OS Data
- Stack

Turning C into Object Code

- Code in files p1.c p2.c
- Compile with command: gcc -O p1.c p2.c -o p
  - Use optimizations (-O)
  - Put resulting binary in file p
  - Compiler (gcc -O)

- 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 (-l)
Compiling Into Assembly

C Code

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

Generated Assembly

```assembly
 .text:
 .globl sum
 sum:
    push %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

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
  - Just contiguously allocated bytes in memory

Primitive 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`

```assembly
  0x401040 <sum>:
  0x55  pushl %ebp
  0x89 e5  movl %esp,%ebp
  0x8b 45 0c  movl 0xc(%ebp),%eax
  0x03 45 08  addl 0x8(%ebp),%eax
  0x89 ec  movl %ebp,%esp
  0x5d  popl %ebp
  0xc3  ret
```

Machine Instruction Example

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

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

Similar to expression

```
  x += y
```

Disassembled Object Code

```assembly
  0x401040:  05 55  push  %ebp
  0x401041:  09 89 e5  mov  %esp,%ebp
  0x401042:  0b 4b 45 0c  mov  0xc(%ebp),%eax
  0x401044:  03 45 08  add  0x8(%ebp),%eax
  0x401046:  5d  pop  %ebp
  0x401047:  c3  ret
```

Alternate Disassembly

Object

```assembly
  0x401040:  05 55  push  %ebp
  0x401041:  09 0f 7e 00  lea  0x0(%esi),%esi
```

```assembly
  0x401044:  03 45 08  add  0x8(%ebp),%eax
```

Within gdb Debugger

```assembly
  0x401046:  5d  pop  %ebp
```

```
gdb p
disassemble sum
```

- Disassemble procedure
- X/13b sum
- Examine the 13 bytes starting at sum
What Can be Disassembled?

```
% objdump -d WINWORD.EXE
WINWORD.EXE:   file format pe-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

Moving Data

Moving Data

- Move 4-byte ("long") word
- Lots of these in typical code

Operand Types

- Immediate: Constant integer data
  - Like C constant, but prefixed with "$"
  - E.g., $0x400, $-533
  - Encoded with 1, 2, or 4 bytes
- Register: One of 8 integer registers
  - But %esp and %ebp reserved for special use
  - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory
  - Various "address modes"

Operand Combinations

```
movl $0x4,%eax
movl $-147,(%eax)
movl %eax,%edx
movl %eax,(%edx)
movl (%eax),%edx
```

Simple Addressing Modes

```
Normal     (R)     Mem[Reg[R]]
- Register R specifies memory address
  `movl (%ecx),%eax`

Displacement  D(R)     Mem[Reg[R]+D]
- Register R specifies start of memory region
  - Constant displacement D specifies offset
  - `movl 8(%ebp),%edx`
```

Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
<td>0x00</td>
<td>%ecx</td>
<td>0x1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x1</td>
<td>0x1</td>
</tr>
<tr>
<td></td>
<td>0xFF</td>
<td>0x13</td>
</tr>
<tr>
<td></td>
<td>263(%ecx)</td>
<td>0x13</td>
</tr>
</tbody>
</table>

Exercise

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x0</td>
<td>%eax</td>
<td>0x104</td>
</tr>
<tr>
<td>0x104</td>
<td>0x1</td>
<td>%ecx</td>
<td>0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x104</td>
<td>0x0</td>
</tr>
<tr>
<td>(%ecx)</td>
<td>0x104</td>
<td>0x2</td>
</tr>
<tr>
<td>4(%eax)</td>
<td>0xC</td>
<td>0x3</td>
</tr>
<tr>
<td>0x3(%ecx)</td>
<td>0x3</td>
<td></td>
</tr>
</tbody>
</table>
Using Simple Addressing Modes

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

Understanding Swap

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

### Stack
- `xp`
- `yp`
- `t0` in `%ebx`
- `t1` in `%eax`
- `%esp`
- `%ebp`

### Address Table

#### Body
- `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`

#### Finish
- `movl -4(%ebp), %ebx` # `ebx = %ebp`
- `movl %ebp, %esp` # `%esp = %ebp`
- `popl %ebp` # `%ebp = %esp`
- `ret` # `%esp = %ebp`

#### Register Variable
- `%ecx` yp
- `%edx` xp
- `%eax` t1
- `%ebx` t0

#### Address
- `0x114` `0x110`
- `0x10c` `0x108`
- `0x104` `0x100`
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
```

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%edx)</td>
<td>0fx000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xe000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xe000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x10e800</td>
</tr>
</tbody>
</table>

Indexed Addressing Modes

Most General Form

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

- \( D \): Constant "displacement" 1, 2, or 4 bytes
- \( Rb \): Base register: Any of 8 integer registers
- \( Ri \): Index register: Any, except for \%esp
- \( S \): Scale: 1, 2, 4, or 8

Special Cases

- \( (Rb,Ri) \equiv \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \)
- \( (Rb,Ri,S) \equiv \text{Mem}[\text{Reg}[Rb]+S\text{Reg}[Ri]] \)

Another Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
<td>0x00</td>
<td>%ecx</td>
<td>0x1</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
<td>%edx</td>
<td>0x3</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operand Value

- 0x11
- 0x100(%ecx,4)
- 0x00
- (%eax, %edx, 4)
- 0x11
- 252(%ecx, %edx)
- 0xFF
Exercise

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
<td>0x00</td>
<td>%ecx</td>
<td>0x104</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
<td>%edx</td>
<td>0x1</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
<td>%ebx</td>
<td>0x8</td>
</tr>
</tbody>
</table>

More on Data Movement

MOVB and MOVW

MOVW moves two bytes, when one of its operands is a register, it must be one of the 8 two-byte registers
e.g. MOVW %ax, %dx

MOVB moves a single byte, when one of its operands is a register, it must be one of the 8 single-byte registers
e.g. MOVB %al, %ah

More on Data Movement

MOVSBL and MOVZBL

MOVSBL sign-extends a single byte, and copies it into a double-word destination
MOVZBL expands a single byte to 32 bits with 24 leading zeros, and copies it into a double-word destination

Example:

- %eax = 0x12345678
- %edx = 0xAAAAABBB

MOV %dh, %al
MOVBSL %dh, %eax
MOVZBL %dh, %eax

Exercise

%eax = 0x12345678
%edx = 0xAAAAABBB

MOV %dh, %al
MOVBSL %dh, %eax
MOVZBL %dh, %eax

Example

Assume register %eax holds value X
%ecx holds value Y

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result in %edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea 8(%eax), %edx</td>
<td>X+8</td>
</tr>
<tr>
<td>lea (%eax,%ecx), %edx</td>
<td>X+Y</td>
</tr>
<tr>
<td>lea 8(%eax,%ecx), %edx</td>
<td>X+Y+8</td>
</tr>
<tr>
<td>lea 8(%eax,%ecx,4), %edx</td>
<td>5X+8</td>
</tr>
<tr>
<td>lea 8(%eax,%ecx,2), %edx</td>
<td>X+2Y+8</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
- **sall** Src, Dest
  - Dest = Dest << Src (also called **shll**)
- **sarl** Src, Dest
  - Dest = Dest >> Src (also called **shrl**)
- **xorl** Src, Dest
  - Dest = Dest ^ Src
- **andl** Src, Dest
  - Dest = Dest & Src
- **orl** Src, Dest
  - Dest = Dest | Src

**One Operand Instructions**

- **incl** Dest
  - Dest = Dest + 1
- **decl** Dest
  - Dest = Dest - 1
- **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 = z + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

#### Set Up

- `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)

#### Body

- `pushl %ebp`
- `movl %esp,%ebp`
- `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)

#### Finish

- `popl %ebp`
- `ret`
Push and Pop
PUSHL takes a single operand: the data source, and store it to the top of stack.
For example,
PUSHL %eax has equivalent behavior as
subl $4, %esp ; stack grows downward
movl %eax, (%esp)
POPL takes the data destination, and pop the top element of stack onto the destination.
POPL %eax has equivalent behavior as
movl (%esp), %eax
addl $4, %esp

CISC Properties
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
- IA32 instructions can range from 1 to 15 bytes

Summary: Abstract Machines
Machine Models
- C
  - char
  - int, float
  - double
  - struct, array
  - pointer
Assembly
- mem
- proc
- stack
- code
- register
Data
- char
- int word
- double
- struct, array
- pointer
Control
- 1) loops
- 2) conditionals
- 3) switch
- 4) Proc. call
- 5) Proc. return

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
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
## Whose Assembler?

### Intel/Microsoft Format

- `lea eax, [edx+edx*2]`
- `sub esp, 8`
- `cmp dword ptr [esp-4], 0`
- `mov eax, dword ptr [eax*4+100h]`

### GAS/Gnu Format

- `lea [ecx, ecx, 2], %eax`
- `subl $8, %esp`
- `cmpl $0, 8(%ebp)`
- `movl $0x100, (%eax, 4), %eax`

### Intel/Microsoft Differs from GAS

- **Operands listed in opposite order**
  - `mov Dest, Src`
  - `movl Src, Dest`

- **Constants not preceded by '$', Denote hex with 'h' at end**
  - `100h $0x100`

- **Operand size indicated by operands rather than operator suffix**
  - `sub` to `subl`

- **Addressing format shows effective address computation**
  - `[eax*4+100h]` to `$0x100(,%eax,4)`