15-213

Machine-Level Programming I: Introduction
Feb. 1, 2000

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

Name **Date** **Transistors**

8086 1978 29K
- 16-bit processor. Basis for IBM PC & DOS
- Limited to 1MB address space. DOS only gives you 640K

80286 1982 134K
- Added elaborate, but not very useful, addressing scheme
- Basis for IBM PC-AT and Windows

386 1985 275K
- Extended to 32 bits. Added "flat addressing"
- Capable of running Unix
- Linux/gcc uses no instructions introduced in later models

486 1989 1.9M

Pentium 1993 3.1M

X86 Evolution: Programmer's View

Name **Date** **Transistors**

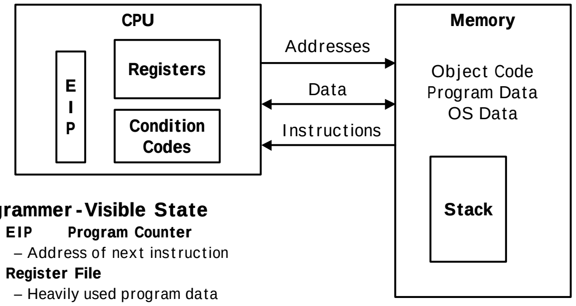
Pentium/MMX 1997 4.5M
- Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data

Pentium II 1997 7M
- Added conditional move instructions
- Big change in underlying microarchitecture

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

Merced 2000? 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"

Assembly Programmer's View



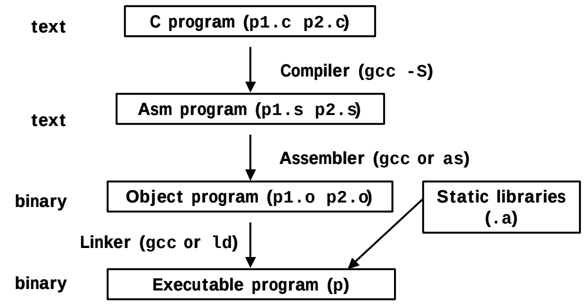
Programmer-Visible State

- **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**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures

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`



Compiling Into Assembly

C Code

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

Generated Assembly

```
_sum:
    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`

Assembly Characteristics

Minimal Data Types

- **"Integer" data of 1, 2, or 4 bytes**
  - Data values
  - Addresses (untyped pointers)
- **Floating point data of 4 or 8 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`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- **Total of 13 bytes**
- **Each instruction 1, 2, or 3 bytes**
- **Starts at address `0x401040`**

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

Machine Instruction Example

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

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

Similar to expression `x += y`

```
0x401046:    03 45 08
```

C Code

- **Add two signed integers**

Assembly

- **Add 2 4-byte integers**
  - "Long" words in GCC parlance
  - Same instruction whether signed or unsigned
- **Operands:**
  - **`x`**: Register **`%eax`**
  - **`y`**: Memory M[**`%ebp+8]`**
  - **`t`**: Register **`%eax`**
    - » Return function value in **`%eax`**

Object Code

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

Disassembling Object Code

Disassembled

```
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
   d:      8d 76 00        lea    0x0(%esi),%esi
```

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

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

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
```

Within gdb Debugger

`gdb p`

`diassemble sum`

- **Disassemble procedure**

`x/13b sum`

- **Examine the 13 bytes starting at `sum`**

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file formatpei-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

`movl` ***Source*,*Dest*: Move 4-byte ("long") word**

- **Accounts for 31% of all instructions in sample**

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"

`%eax`
`%edx`
`%ecx`
`%ebx`
`%esi`
`%edi`
`%esp`
`%ebp`

`movl` Operand Combinations

	Source	Destination		C Analog
movl	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

- **Cannot do memory-memory transfers with single instruction**

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`

Using Simple Addressing Modes

```
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
```

Set Up

Body

Finish

Understanding Swap

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

Stack

Offset	
	• • •
12	yp
8	xp
4	Rtn adr
0	Old %ebp ← %ebp
-4	Old %ebx

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	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
```


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

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***	Arithmetic
`shrl` ***Src,Dest***	***Dest*** = ***Dest*** >> ***Src***	Logical
`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

```
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:
    pushl %ebp                  } Set
    movl %esp,%ebp              } Up

    movl 8(%ebp),%eax           } Body
    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              } Finish
    popl %ebp
    ret
```


Understanding `arith`

```
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;
}
```

Stack

Offset	
	• • •
16	z
12	y
8	x
4	Rtn adr
0	Old %ebp ← %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)
```


Another Example

```
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;
}
```

```
logical:
    pushl %ebp                  } Set
    movl %esp,%ebp              } Up

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

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

$\boxed{2^{13} = 8192,\ 2^{13} - 7 = 8185}$

```
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
```

CISC Properties

Instruction can reference different operand types

- Immediate, register, memory

Arithmetic operations can read/write memory

Memory reference can involve complex computation

- Rb + S*Ri + D
- Useful for arithmetic expressions, too

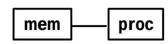
Instructions can have varying lengths

- IA32 instructions can range from 1 to 15 bytes

Summary: Abstract Machines

Machine Models

C



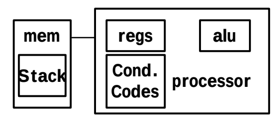
Data

1) `char`
2) `int`, `float`
3) `double`
4) `struct`, array
5) pointer

Control

1) loops
2) conditionals
3) `goto`
4) Proc. call
5) Proc. return

Assembly



1) byte
2) 4-byte long word
3) 8-byte quad word
4) contiguous byte allocation
5) address of initial byte

3) branch/jump
4) `call`
5) `ret`

Pentium Pro (P6)

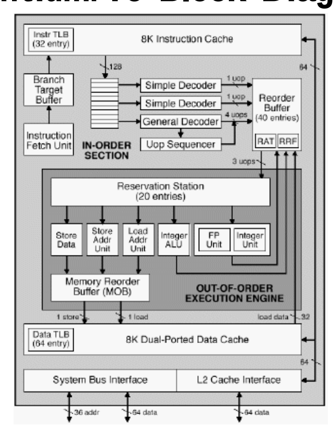
History

- Announced in Feb. '95
- Basis for Pentium II & Pentium III

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 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**
- **Difficult to predict / optimize performance at assembly level**

Whose Assembler?

Intel/Microsoft Format

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

GAS/Gnu Format

```
leal (%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 hexadecimal with 'h' at end**

  100h $0x100
- **Operand size indicated by operands rather than operator suffix**

  sub   subl
- **Addressing format shows effective address computation**

  [eax*4+100h] $0x100(,%eax,4)