

15-213

"The course that gives CMU its Zip!"

Machine-Level Programming I: Introduction Sept. 10, 2002

Topics

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

class05.ppt

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

X86 Evolution: Programmer's View

Name	Date	Transistors
486	1989	1.9M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
		■ Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data
PentiumPro	1995	6.5M
		■ Added conditional move instructions
		■ Big change in underlying microarchitecture

X86 Evolution: Programmer's View

Name	Date	Transistors
Pentium III	1999	8.2M
<ul style="list-style-type: none"> 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
<ul style="list-style-type: none"> Added 8-byte formats and 144 new instructions for streaming SIMD mode 		

- 5 -

15-213, F'02

New Species: IA64

Name	Date	Transistors
Itanium	2001	10M
<ul style="list-style-type: none"> Extends to IA64, a 64-bit architecture Radically new instruction set designed for high performance Will be able to run existing IA32 programs <ul style="list-style-type: none"> On-board "x86 engine" Joint project with Hewlett-Packard 		
Itanium 2	2002	221M
<ul style="list-style-type: none"> Big performance boost 		

- 7 -

15-213, F'02

X86 Evolution: Clones

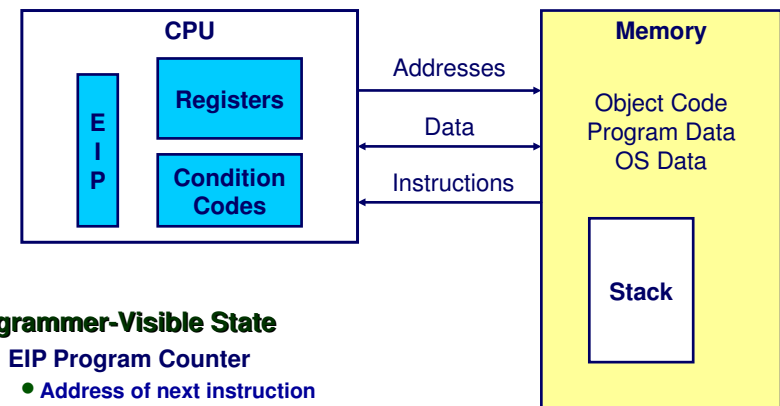
Advanced Micro Devices (AMD)

- 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

- 6 -

15-213, F'02

Assembly Programmer's View



Programmer-Visible State

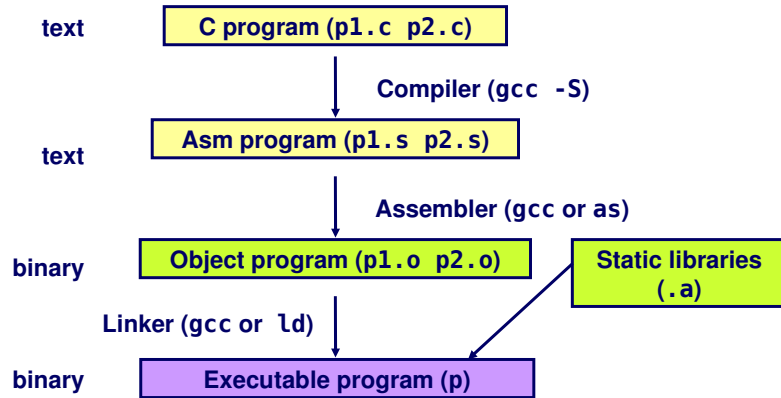
- EIP Program Counter**
 - Address of next instruction
- Register File**
 - Heavily used program data
- Condition Codes**
 - Store status information about most recent arithmetic operation
 - Used for conditional branching
- Memory**
 - Byte addressable array
 - Code, user data, (some) OS data
 - Includes stack used to support procedures

- 8 -

15-213, F'02

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



Assembly Characteristics

Minimal Data Types

- “Integer” data of 1, 2, or 4 bytes
 - Data values
 - 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

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

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;
```

C Code

- Add two signed integers

Assembly

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

Similar to expression
x += y

- 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

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

Object Code

- 3-byte instruction
- Stored at address 0x401046

Moving Data

Moving Data

movl Source, Dest:

- 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”

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

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

movl Operand Combinations

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

- Cannot do memory-memory transfers with single instruction

Simple Addressing Modes

Normal (R) Mem[Reg[R]]

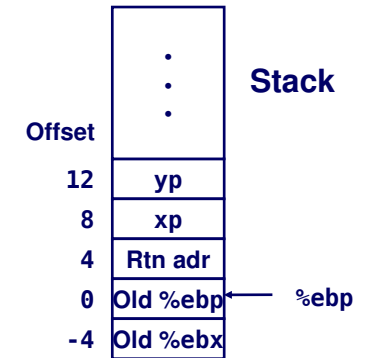
- Register R specifies memory address
- ```
movl (%ecx),%eax
```

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

- Register R specifies start of memory region
  - Constant displacement D specifies offset
- ```
movl 8(%ebp),%edx
```

Understanding Swap

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



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
```

Using Simple Addressing Modes

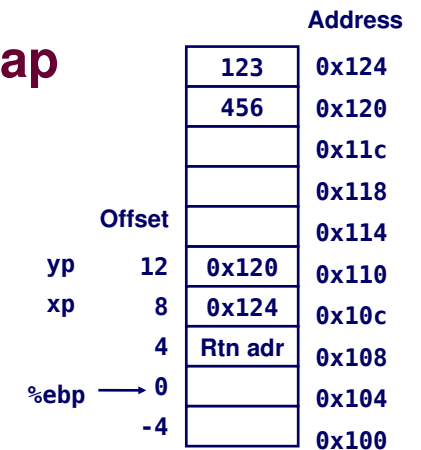
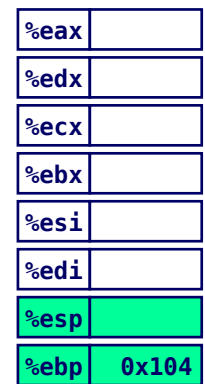
```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

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

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax, (%edx)
    movl %ebx, (%ecx)
    } Body

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

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

		Offset	Address
			123 0x124
			456 0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

```

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	
%esi	
%edi	
%esp	
%ebp	0x104

		Offset	Address
			123 0x124
			456 0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

```

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	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Offset	Address
			123 0x124
			456 0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

```

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

		Offset	Address
			123 0x124
			456 0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp	→ 0		0x104
	-4		0x100

```

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

Offset		Address
		456 0x124
		456 0x120
		0x11c 0x118
		0x114 0x114
yp	12	0x120 0x110
xp	8	0x124 0x10c
	4	Rtn adr 0x108
%ebp	→ 0	0x104
	-4	0x100

```

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]]

Understanding Swap

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

Offset		Address
		456 0x124
		123 0x120
		0x11c 0x118
		0x114 0x114
yp	12	0x120 0x110
xp	8	0x124 0x10c
	4	Rtn adr 0x108
%ebp	→ 0	0x104
	-4	0x100

```

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

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

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 $x + k*y$
 - $k = 1, 2, 4, \text{ or } 8$.

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
    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
```

Set Up

Body

Finish

Some Arithmetic Operations

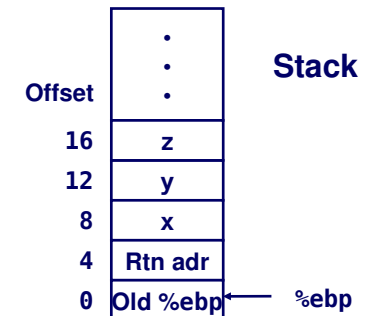
Format Computation

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

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;
}
```



```
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

```
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;
}
```

```
# 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
```

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

CISC Properties

Instruction can reference different operand types

- Immediate, register, memory

Arithmetic operations can read/write memory

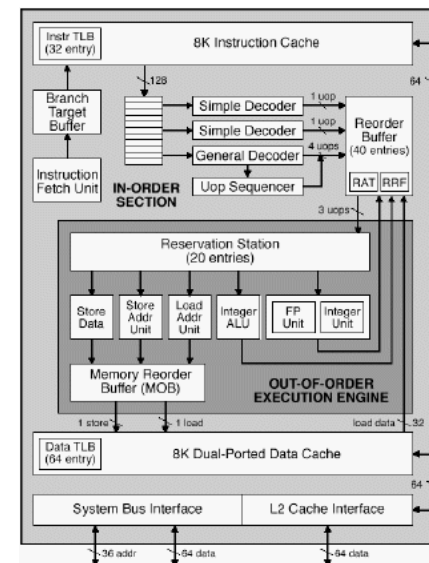
Memory reference can involve complex computation

- $R_b + S^*R_i + D$
- Useful for arithmetic expressions, too

Instructions can have varying lengths

- IA32 instructions can range from 1 to 15 bytes

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
- Tricky to predict / optimize performance at assembly level