

# CS 47

## Machine-Level Programming I: Introduction

### Topics

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

class07.ppt

CS 47 Spring 2008

## 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 C programs
- Hard to match performance of Reduced Instruction Set Computers (RISC)
- But, Intel has done just that!

- 2 -

CS 47 Spring, 2008

## X86 Evolution: Programmer's View

Name	Date	Transistors
<b>8086</b>	<b>1978</b>	<b>29K</b>
<ul style="list-style-type: none"><li>■ 16-bit processor. Basis for IBM PC &amp; DOS</li><li>■ Limited to 1MB address space. DOS only gives you 640K</li></ul>		
<b>80286</b>	<b>1982</b>	<b>134K</b>
<ul style="list-style-type: none"><li>■ Added elaborate, but not very useful, addressing scheme</li><li>■ Basis for IBM PC-AT and Windows</li></ul>		
<b>386</b>	<b>1985</b>	<b>275K</b>
<ul style="list-style-type: none"><li>■ Extended to 32 bits. Added "flat addressing"</li><li>■ Capable of running Unix</li><li>■ Linux/gcc uses no instructions introduced in later models</li></ul>		

- 3 -

CS 47 Spring, 2008

## X86 Evolution: Programmer's View

Name	Date	Transistors
<b>486</b>	<b>1989</b>	<b>1.9M</b>
<b>Pentium</b>	<b>1993</b>	<b>3.1M</b>
<b>Pentium/MMX</b>	<b>1997</b>	<b>4.5M</b>
<ul style="list-style-type: none"><li>■ Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data</li></ul>		
<b>PentiumPro</b>	<b>1995</b>	<b>6.5M</b>
<ul style="list-style-type: none"><li>■ Added conditional move instructions</li><li>■ Big change in underlying microarchitecture</li></ul>		

- 4 -

CS 47 Spring, 2008

## X86 Evolution: Programmer's View

Name	Date	Transistors
<b>Pentium III</b>	<b>1999</b>	<b>8.2M</b>
		<ul style="list-style-type: none"><li>■ Added “streaming SIMD” instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or floating point data</li></ul>
<b>Pentium 4</b>	<b>2001</b>	<b>42M</b>
		<ul style="list-style-type: none"><li>■ Added 8-byte formats and 144 new instructions for streaming SIMD mode</li></ul>

- 5 -

CS 47 Spring, 2008

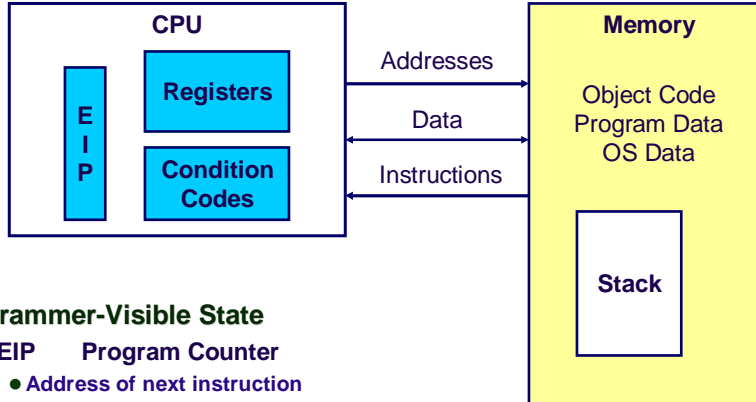
## New Species: IA64

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

- 6 -

CS 47 Spring, 2008

## Assembly Programmer's View



### Programmer-Visible State

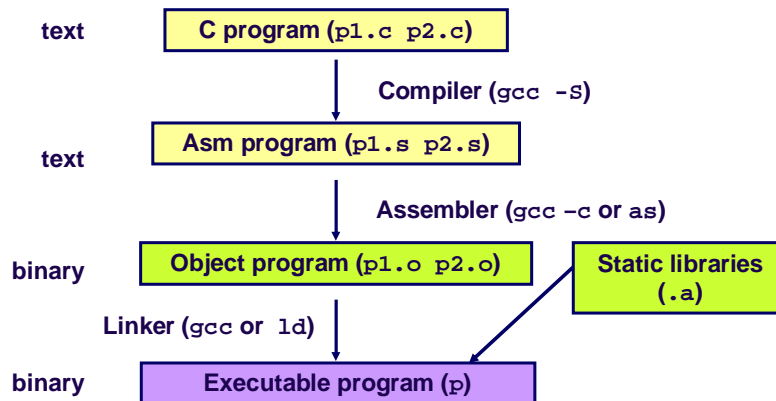
- **EIP** Program Counter
  - Address of next instruction
- **Register File**
  - Heavily used program data
- **Condition Codes**
  - Store status information about most recent arithmetic operation
  - Used for conditional branching
- **Memory**
  - Byte addressable array
  - Code, user data, (some)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

- 7 -

CS 47 Spring, 2008

## 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`



- 8 -

CS 47 Spring, 2008

# 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, 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`

```
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

- 11 -

CS 47 Spring, 2008

# Machine Instruction Example

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

## C Code

- Add two signed integers

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

## Assembly

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

Similar to  
expression  
`y += x`

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

## Object Code

- 3-byte instruction
- Stored at address 0x401046

- 12 -

CS 47 Spring, 2008

# 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

- 13 -

CS 47 Spring, 2008

# 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
```

```
disassemble sum
```

- Disassemble procedure

```
x/13b sum
```

- Examine the 13 bytes starting at `sum`

- 14 -

CS 47 Spring, 2008

# What Can be Disassembled?

```
% 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

# 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

## 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

- 17 -

CS 47 Spring, 2008

## 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

- 18 -

CS 47 Spring, 2008

## 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
```

- 19 -

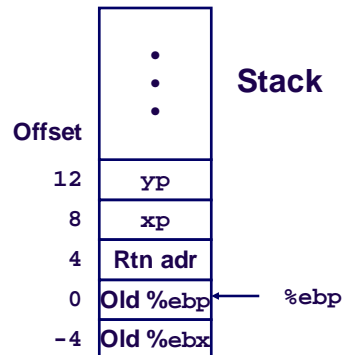
CS 47 Spring, 2008

## 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
```



- 20 -

CS 47 Spring, 2008

# Understanding Swap

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

		Address	
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
	Offset		
	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	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

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

```

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
		456
yp	12	0x120
xp	8	0x124
	4	Rtn adr
%ebp	→ 0	
	-4	

```

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
			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
    
```

# Indexed Addressing Modes

## Most General Form

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

- D: Constant “displacement” or offset
- 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

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

## Address Computation Examples

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x100</code>

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

- 29 -

CS 47 Spring, 2008

## 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.`

- 30 -

CS 47 Spring, 2008

## Some Arithmetic Operations

Format	Computation
--------	-------------

### Two Operand Instructions

<code>addl Src, Dest</code>	$Dest = Dest + Src$
<code>subl Src, Dest</code>	$Dest = Dest - Src$
<code>imull Src, Dest</code>	$Dest = Dest * Src$
<code>sall Src, Dest</code>	$Dest = Dest \ll Src$ Also called <code>shll</code>
<code>sarl Src, Dest</code>	$Dest = Dest \gg Src$ Arithmetic
<code>shrl Src, Dest</code>	$Dest = Dest \gg Src$ Logical
<code>xorl Src, Dest</code>	$Dest = Dest \wedge Src$
<code>andl Src, Dest</code>	$Dest = Dest \& Src$
<code>orl Src, Dest</code>	$Dest = Dest   Src$

## Some Arithmetic Operations

Format	Computation
--------	-------------

### One Operand Instructions

<code>incl Dest</code>	$Dest = Dest + 1$
<code>decl Dest</code>	$Dest = Dest - 1$
<code>negl Dest</code>	$Dest = - Dest$
<code>notl Dest</code>	$Dest = \sim Dest$

# Using lea1 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
```

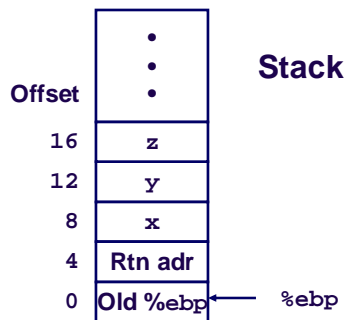
} Set Up

} Body

} Finish

# 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
```

- 35 -

CS 47 Spring, 2008

## 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;
}
```

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

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

logical:

```
pushl %ebp
movl %esp,%ebp
```

} Set Up

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

} Body

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

} Finish

```
eax = x
eax = x^y (t1)
eax = t1>>17 (t2)
eax = t2 & 8185
```

- 36 -

CS 47 Spring, 2008

# 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

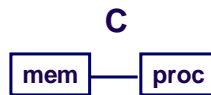
- IA32 instructions can range from 1 to 15 bytes

- 37 -

CS 47 Spring, 2008

# Summary: Abstract Machines

## Machine Models



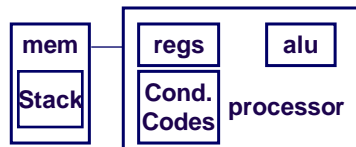
## Data

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

## Control

- 1) loops
- 2) conditionals
- 3) switch
- 4) Proc. call
- 5) Proc. return

## Assembly



- 1) byte
  - 2) 2-byte word
  - 3) 4-byte long word
  - 4) contiguous byte allocation
  - 5) address of initial byte
- 3) branch/jump
  - 4) call
  - 5) ret

- 38 -

CS 47 Spring, 2008

# 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 hex 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)