CS 354 - Machine Organization & Programming
Tuesday, October 31, 2017

Project p3 (6%): DUE at 10 pm TOMORROW, Wednesday, November 1st

Project p4 (6%): DUE at 10 pm on Wednesday, November 15th

Homework hw5 (1.5%): Assigned tomorrow

Last Time

Exam Mechanics
Project p3 and gdb
Assembly Language Intro

Today

Low-level Data
Registers
Instructions - MOV, PUSH, POP
Operand Specifiers
Operands Practice
Operand/Instruction Caveats

Next Time

Midterm Exams Returned
More IA-32 Instructions
Read: B&O 3.5, 3.6

Low-Level View of Data

Instruction Set Architecture ISA

Specifies format and behavior of machine-level programming.

	x86 (AKA IA-32)	x64 (AKA x86-64, NOT IA-64)
address/data	32 bit	64 bti
address space	2^32 = 4GB	2^64 = 16EB max mem today: 2^48

C's View

vars with specific types that can be complex composites built from arrays and structs

Machine's View

Memory is a large byte array
no distinction in memory for data types, pointers, etc.
Asm instructions determine how big? How many bytes of data? What type? How to interpret the bits?

Assembly Data Formats

C	IA-32	Assembly Suffix	Size in bytes	
`char`	byte	b	1	
`short`	word	w	2	NOTE: word isn't 4 bytes as we assumed
`int`	double word	l	4	
`long int`	double word	l	4	
`char*`	double word	l	4	
`float`	single precision	s	4	
`double`	double prec	l	8	Quad word
`long double`	extended prec	t	10, typically 12 bytes	

Registers

What? Registers are: Fastest memory, which stores address and data that can be directly accessed by the ALU

General Registers

Pre-named locations that can store upto 32 bit values

	bit 31	15	8 7	1
`%eax`	accumulator	`%ax`	`%ah`	`%al`
`%ecx`	count	`%cx`	`%ch`	`%cl`
`%edx`	data	`%dx`	`%dh`	`%dl`
`%ebx`	base	`%bx`	`%bh`	`%bl`
`%esi`	Source Index	`%si`		
`%edi`	Destination Index	`%di`		
`%esp`	* Stack Pointer	`%sp`		
`%ebp`	* Stack Base Pointer	`%bp`		

Program Counter `%eip` Extended Instruction Pointer

It stores the address of the next instruction to be executed

Condition Code Registers

Store the status of the last ALU instruction
Used for conditioned changes in the program flow

Instructions - MOV, PUSH, POP

What? Instructions to copy data from one location to another

Why? Enables info to be moved around in registers and memory.

How?

instruction class	opearation	description
MOV S,D movb, movw, movl	D ← S	move s value to d destination s and d are the same size
MOVS S,D movsbw, movsbl, movswl	D ← sign extended S	moves smaller s to d and fills d by copying most significant bit of s
MOVZ S,D movzbw, movzbl, movzwl		Fills d with zeros
pushl S	R[%esp] <-- R[%esp] - 4 M[R[%esp]] <-- S	push s onto stack
popl D	D <-- M[R[%esp]] R[%esp] <-- R[%esp] + 4	pop top of stack to D

Practice with Data Formats

→ What data format suffix should replace the _ given the registers used?

1. `mov_ %eax, %esp`
2. `mov_ (%eax), %dx`
3. `mov_ $0xFF, %bl`
4. `mov_ (%esp, %edx, 4), %dh`
5. `push_ $0xFF`
6. `mov_ %dx, (%eax)`
7. `pop_ %edi`

		Bytes	Bits
b	bytes	1	8
w	word	2	16
l	double word	4	32

Operand Specifiers

What? Operand specifiers are:

- S source specifies value to be used by an instruction
- D destination specifies the location where the result is to be stored

Why?

Enables intructions to access constants (s only), registers and memory locations.

How?

Immediate specifies an operand value that's a constant

	operand value	
$*Imm*	*Imm*	imm in C's format for constants

Register specifies an operand value that's in a register

	operand value
%E_a	$R[E_a]$

Memory specifies an operand value that's

	operand value	addressing mode
Imm	M[*Imm*]	absolute
(E_a)	$M[R[E_a]]$	indirect
Imm(E_b)	M[*Imm*+$R[E_b]$]	base + offset
(E_b,E_i)	$M[R[E_b]+R[E_i]]$	indexed
Imm(E_b,E_i)	M[*Imm*+$R[E_b]$+$R[E_i]$]	index + offset
(,E_i,s)	$M[R[E_i]*s]$	
(E_b,E_i,s)	$M[R[E_b]+R[E_i]*s]$	scaled indexed
Imm(,E_i,s)	M[*Imm*+$R[E_i]$*s]	
Imm(E_b,E_i,s)	M[*Imm*+$R[E_b]$+$R[E_i]$*s]	

( )

s scale factor which is 1, 2, 4, 8

Operands Practice

Given:

Address	Value	Register	Value
0x100	0x FF	%eax	0x 104
0x104	0x AA	%ecx	0x 1
0x108	0x 11	%edx	0x 4
0x10C	0x 22		
0x110	0x 33		

→ What is the value being accessed? (Complete the information below.)

Operand	Value	Type/Mode	Effective Address

1. `(%eax)`
2. `%edx`
3. `0x108`
4. `$0x108`
5. `-4(%eax)`
6. `(%eax,%edx,2)`
7. `0xF8(,%ecx,8)`
8. `259(%ecx,%edx)`
9. `4(%eax,%edx,2)`

Note:

Operand/Instruction Caveats

Missing Combination?

→ Identify each source and destination operand combination.

1. `movl $0xABCD,%ecx` immediate to register
2. `movb $11,(%ebp)` immediate to memory
3. `movb %ah,%dl` register to register
4. `movl %eax,-12(%esp)` register to memory
5. `movb (%ebx,%ecx,2),%al` memory to register

→ What combination is missing?

No memory to memory
instead you must do {memory to register} and then {register to memory}

Instruction Oops!

→ What is wrong with each instruction below?

1. `movl %bl,(%ebp)`
2. `movl %ebx,$0xA1FF`
3. `movw %dx,%eax`
4. `movb $0x11,(%ax)`
5. `movb %si,-12(%esp)`
6. `movw (%eax),(%ebx,%esi)`
7. `movb %sh, %bl`