CS 354 - Machine Organization & Programming
Tuesday March 28 and Thursday March 30, 2023

Midterm Exam - Thurs April 6th, 7:30 - 9:30 pm

- **UW ID and #2 required**
- **closed book, no notes, no electronic devices (e.g., calculators, phones, watches) see "Midterm Exam 2" on course site Assignments for topics**

Homework hw4: DUE on or before Monday, Mar 27

Homework hw5: will be DUE on or before Monday, Apr 10

Project p4A: DUE on or before Friday, Mar 31

Project p4B: DUE on or before Friday, Apr 7

Last Week

Direct Mapped Caches - Restrictive Fully Associative Caches - Unrestrictive Set Associative Caches - Sweet! Replacement Policies Writing to Caches	Writing to Caches (cont) Cache Performance ----- Impact of Stride Memory Mountain C, Assembly, & Machine Code

This Week

Impact of Stride -- L16-8 Memory Mountain - L16-9 C, Assembly, & Machine Code - L16-10 Low-level View of Data Registers Operand Specifiers & Practice L18-7	Instructions - MOV, PUSH, POP Operand/Instruction Caveats Instruction - LEAL Instructions - Arithmetic and Shift Instructions - CMP and TEST, Condition Codes Instructions - SET & Jumps Encoding Targets & Converting Loops
Next Week: Stack Frames and Exam 2 B&O 3.7 Intro - 3.7.5, 3.8 Array Allocation and Access 3.9 Heterogeneous Data Structures	

This page in last lecture notes: **C, Assembly, & Machine Code**

C Function	Assembly (AT&T)	Machine (hex)
`int accum = 0;`		
`int sum(int x, int y)`	`sum:`	
`{`	`pushl %ebp`	`55`
	`movl %esp, %ebp`	`89 e5`
	`movl 12(%ebp), %eax`	`8b 45 0C`
`int t = x + y;`	`addl 8(%ebp), %eax`	`03 45 08`
`accum += t;`	`addl %eax, accum`	`01 05 ?? ?? ?? ??`
`return t;`	`popl %ebp`	`5D`
`}`	`ret`	`C3`

C

-
-
-

→ What aspects of the machine does C hide from us?

Assembly (ASM)

-
-

→ What ISA (Instruction Set Architecture) are we studying?

→ What does assembly remove from C source?

→ Why Learn Assembly?

1.
2.
3.

Machine Code (MC) **is**

-
-

→ How many bytes long is an IA-32 instructions?

Low-Level View of Data

C's View

- vars declared of specific type
- type can be complex, composites, array, struct, union

Machine's View

- mem is an array of bytes indexed by address
- where each element is a byte

❈ *Memory contains bits that do not* distinguish instruction from data or ptr

Distinguish difference through the instruction

→ How does a machine know what it's getting from memory?

1. by how it is accessed

2. by the instruction itself

Assembly Data Formats

C	IA-32	Assembly Suffix	Size in bytes
*`char`	byte	b	1
`short`	word (2 bytes)	w	2
*`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
`long double`	extended prec	t	10 (or sometimes 12

* = memorize

❈ *In IA-32 a word* is a actually 2 bytes (in caching it is 4 bytes)

Registers

What? Registers

- are fastest memory, directly accessed by ALU
- can store 1, 2, or 4 bytes of data, or 4 bytes for addresses

General Registers

pre-named locations that store up to 32 bit values

bit 31 — 16 15 — 8 7 — 0

`%eax`	accumulator	`%ax` high	`%ah`	`%al` low
`%ecx`	count	`%cx`	`%ch`	`%cl`
`%edx`	data	`%dx`	`%dh`	`%dl`
`%ebx`	base	`%bx`	`%bh`	`%bl`
`%esi`	src index	`%si`		
`%edi`	dest index	`%di`		
`%esp`	stack pointer	`%sp`		
`%ebp`	base pointer	`%bp`		

Other Registers

Program Counter `%eip` ext inst ptr

stores addr of next instruction

Condition Code Registers a.k.a. e-flags

1 bit register that stores status of most recently used ALU operant

Operand Specifiers

What? Operand specifiers are

- S source, specify loc of value to be used (read) by inst
- D destination, specify location where result is to be stored

Why?
Enables inst. to specify constants (S only)
registers
mem. loc.

How? IA-32 has 3 kinds of operand specifiers

1.) IMMEDIATE specifies an operand value that's a CONSTANT

specifier	**operand value**	
$*Imm*	*Imm*	Imm is in C' int literal format

$1, $0x1A
$071 (base 8)

2.) REGISTER specifies an operand value that's in a register

specifier	**operand value**	
$\%E_a$	$R[\%E_a]$	R = register, %eax, %cl, %si

arbitrary

3.) MEMORY specifies an operand value that's

	specifier	**operand value**	**effective address**	**addressing mode name**	
	Imm	M[EffAddr]	*Imm*	Absolute	m = memory
( )	$(\%E_a)$	M[EffAddr]	$R[\%E_a]$	Indirect -	
	$Imm(\%E_b)$	M[EffAddr]	$Imm+R[\%E_b]$	Base + offset	
(,)	$(\%E_b,\%E_i)$	M[EffAddr]	$R[\%E_b]+R[\%E_i]$	Indexed	
	$Imm(\%E_b,\%E_i)$	M[EffAddr]	$Imm+R[\%E_b]+R[\%E_i]$	Indexed + offset	
				SCALED INDEX	
*	$Imm(\%E_b,\%E_i,s)$	M[EffAddr]	$Imm+R[\%E_b]+R[\%E_i]*s$	offset + base + index * scale	
(, ,)	$(\%E_b,\%E_i,s)$	M[EffAddr]	$R[\%E_b]+R[\%E_i]*s$	w/o offset	
*** most important mem operand	$Imm(,\%E_i,s)$	M[EffAddr]	$Imm+R[\%E_i]*s$	no base register	
	$(,\%E_i,s)$	M[EffAddr]	$R[\%E_i]*s$	no base register no offset	

Imm is offset
E_b is base reg E_1 index reg must be 32 bit

Operands Practice

Given:

MM

Memory Addr	Value
0x100	0xFF
0x104	0xAA
0x108	0x11
0x10C	0x22
0x110	0x33

CPU

Register	Value
%eax	0x104
%ecx	0x1
%edx	0x4

→ What is the value being accessed? Also identify the type of operand, and for memory types name the addressing mode and determine the effective address.

	Operand	Value	Type:Mode	Effective Address
1.	`(%eax)`	0xAA	Indirect	0x104
2.	`0xF8(,%ecx,8)`	?		
3.	`%edx`	0x4	REGISTER	
4.	`$0x108`	0x108	Immediate	----
5.	`-4(%eax)`			
6.	`4(%eax,%edx,2)`	0x33	SCALED INDEX	%eax + %edx * 2 + 4 = 0x104 + (4*2) + 4 = 0x110
7.	`(%eax,%edx,2)`			
8.	`0x108`			
9.	`259(%ecx,%edx)`			

Instructions - MOV, PUSH, POP

STACK

What? These are instructions to copy data from source (S) to destination (D)

Why? enable info to move around in mem & registers

**ONLY CONCERNED W/ pushing 32 BITS, pushl, popl

How?

instruction class	operation	description	
MOV S, D movb, movw, movl	D <- S	move(copy) S to D	S + D must be same size
MOVS S, D movsbw, movsbl, movswl	D <- S sign.extend	copy S to D	
MOVZ S, D movzbw, movzbl, movzwl	D <- S zero-extend	copy S to D	
pushl S	R[%esp] <- (R[%esp] - 4) M[R[%esp]] <- S	make room on "top" of stack copy S to top of stack	
popl D	D <- M[R[%esp]] R[%esp] <- (R[%esp] + 4)	copy top of stack to D pop(shrink) stack	

Practice with Data Formats

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

```
* 1. mov_   %eax, %esp
* 2. push_  $0xFF
  3. mov_   (%eax), %dx
* 4. mov_   (%esp, %edx, 4), %dh
  5. mov_   0x800AFFE7, %bl
* 6. mov_   %dx, (%eax)
  7. pop_   %edi
```

← assumed to be 32 bits

← 8 bit reg, look at chart for other registers (dx = 16, eax = 32)

✵ *Focus on register type operands* since they can be 1, 2, or 4 bytes

Operand/Instruction Caveats

Missing Combination? S, D

→ Identify each source and destination operand type combinations.

		Size
1. `movl $0xABCD,%ecx`	Imm to Reg	4 bytes
2. `movb $11,(%ebp)`	Imm to Mem	1 byte
3. `movb %ah,%dl`	Reg to Reg	1 byte
4. `movl %eax,-12(%esp)`	Reg to Mem	4 bytes
5. `movb (%ebx,%ecx,2),%al`	Mem to Reg	1 byte

→ What combination is missing? Mem to Imm - can't write to immediate
Mem to Mem - can't be done in 1 instruction,
must instead to Mem to Reg then Reg to Mem

Instruction Oops!

→ What is wrong with each instruction below?

*1. `movl %bl,(%ebp)` instruction and source are not same size

*2. `movl %ebx,$0xA1FF` can't write to immediate, not allowed

3. `movw %dx,%eax`

*4. `movb $0x11,(%ax)` mem must be described with 32 bits

5. `movw (%eax),(%ebx,%esi)`

*6. `movb %sh, %bl` %sh is not a register (** must memorize registers for exam!!!)

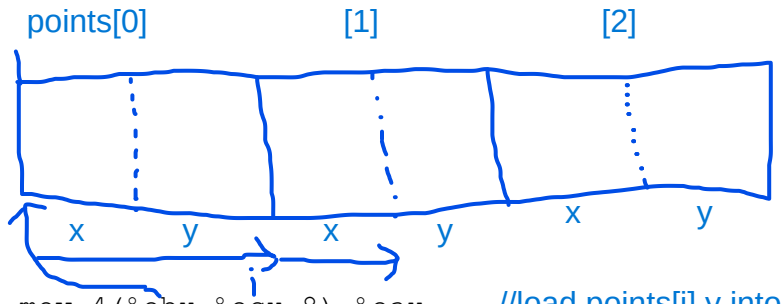
Instruction - LEAL

Load Effective Address l (double-word)

```
leal S,D      D <-- &S
```

LEAL vs. MOV

```
0   struct Point {
4      int x;
       int y;        4 } 8 bytes
    } points[3];     4
```




```
int y = points[i].y;        mov 4(%ebx,%ecx,8),%eax    //load points[i].y into %eax
```

offset to data member y

```
points[1].y;
```

```
int *py = &points[i].y;     leal 4(%ebx,%ecx,8),%eax
```

%ebx + (%ecx * 8) + 4 → D

LEAL Simple Math

```
leal  -3(%ebx), %eax        subl $3, %ebx
                            movl %ebx, %eax
```

Changes ebx

No change to ebx

→ Suppose register %eax (x) holds x and %ecx (y) holds y.
What value in terms of x and y is stored in %ebx for each instruction below?

1. `leal (%eax,%ecx,8),%ebx` — x + (y * 8) => x + 8y
2. `leal 12(%eax,%eax,4),%ebx` — x + (x * 4) + 12 => 5x + 12
3. `leal 11(%ecx),%ebx` — y + 11
4. `leal 9(%eax,%ecx,4),%ebx` — x + (y * 4) + 9 => x + 4y + 9

Instructions - Arithmetic and Shift

Unary Operations

```
INC D       D <-- D + 1
DEC D       D <-- D - 1
NEG D       D <-- -D
NOT D       D <-- ~D
```

2's complement negation D = D * -1
bit-wise complement (flip all bits, don't add 1)

Binary Operations

```
ADD S,D     D <-- D + S
SUB S,D     D <-- D - S
IMUL S,D    D <-- D * S
XOR S,D     D <-- D ^ S
OR S,D      D <-- D | S
AND S,D     D <-- D & S
```

D = D + S
D = D - S sub a,b => b = b - a (memorize order for subtract!!!)
D = D * S
exclusive-OR
bitwise-OR
bitwise-AND

Given: M.M.

0x100	0xFF
0x104	0xAB
0x108	0x10
addr	value

CPU

%eax	0x100
%ecx	0x1
%edx	0x2
reg	values

→ What is the destination and result for each? (do each independently)

1. `incl 4(%eax)` incr(+1) the value at 0x104
2. `addl %ecx,(%eax)` m[0x100] = m[0x100] + 1 => 0x100
3. `addl $32,(%eax,%edx,4)`
4. `subl %edx,0x104`

SAR → 1 0 0 0 1 1 1 1 0 0 0
1 0 0 0 1 1 1 1

Shift Operations

- move bits to left or right by k positions
  0 <= k < 32, k always in reg %cl
- for fast power of 2 multiplication or division

logical shift

```
SHL k,D     D <-- D << K
SHR k,D     D <-- D >> K
```

0 1 1 0 = 6
0 0 1 1 = 3 /2

arithmetic shift

```
SAL k,D     D <-- D << K
SAR k,D     D <-- D >> K
```

sar, 1, %d
1 1 1 1 = -1
1 1 1 1

-2= 1 1 1 0
0 0 0 1
-2= 1 1 1 1

LAST PAGE OF REVIEW FOR MIDTERM 2

Instructions - CMP and TEST, Condition Codes

What? CMP (subtract) TEST

- compare values arithmetically or logical AND

- set condition codes, does not change operands

Why?

to enable relational and logical operations

sub S, D ≅ D <- D - S

How?

```
CMP S2,S1              CC <-- S1 - S2


TEST S2,S1             CC <-- S1 & S2
```

➢ What is done by `testl %eax, %eax`

Condition Codes (CC)

ZF: zero flag	is 1, if result was 0
CF: carry flag	is 1, if result cased unsigned overflow (needs 1 more bit)
SF: sign flag	is 1, if result is negative
OF: overflow flag	is 1, if result caused signed overflow

Instructions - SET

What?

set a byte register to 1 if a condition is true, 0 if false
specific condition is determined from CCs

How?

```
sete D   setz     D <-- ZF                  == equal
setne D  setnz    D <-- ~ZF                 != not equal
sets D            D <-- SF                  < 0  signed (negative)
setns D           D <-- ~SF                 >= 0 not signed (nonnegative)
```

Unsigned Comparisons: t = a - b if a - b < 0 => CF = 1 if a - b > 0 => ZF = 0

below

```
setb D   setnae   D <-- CF                  <  below
setbe D  setna    D <-- CF | ZF             <= below or equal
seta D   setnbe   D <-- ~CF & ~ZF           >  above
setae D  setnb    D <-- ~CF                 >= above or equal
```

above

Signed (2's Complement) Comparisons

```
setl D   setnge   D <-- SF ^ OF             <  less (note l ISN'T size suffix)
setle D  setng    D <-- (SF ^ OF) | ZF      <= less or equal
setg D   setnle   D <-- ~(SF ^ OF) & ~ZF    >  greater
setge D  setnl    D <-- ~(SF ^ OF)          >= greater or equal
```

Demorgan's Law: ~(a & b) => ~a | ~b ~(a | b) => ~a & ~b note ~ bitwise not, ! logical not

Example: a < b (assume int a is in %eax, int b is in %ebx)

S2 S1

1. `cmpl %ebx,%eax` S1 - S2 %eax - %ebx a - b

2. `setl %cl`

3. `movzbl %cl,%ecx`

0000 0001
0000 0000 0001
?

Instructions - Jumps

What? transfer program execution to another location

target: desired location (that you jump to)

Why? enable selection and repetition

How? Unconditional Jump just jump!

indirect jump: target is an address in memory

`jmp *Operand`	%eip <- operand value
jmp *%eax	reg value's target
jmp *(%eax)	reg value is address in memory of target

direct jump: target address is in instruction

`jmp Label` %eip <- Label's address

jmp .L1

.L1: mov

How? Conditional Jumps

- jump if condition is met (based on condition codes of previous instruction)

- can only be direct jump

```
both:      je Label      jne Label     js Label      jns Label
unsigned:  jb Label      jbe Label     ja Label      jae Label
signed:    jl Label      jle Label     jg Label      jge Label
```

endings have same meanings as set

Encoding Targets

What? technique to specify target for direct jump

Absolute Encoding target is a 32-bit address 0x_ _ _ _ _ _ _ _

Problems?

- code is not compact, requires 32 bits for address
- code cannot be moved without changing target

Solution? Relative Encoding

target is specified as distance from jump instruction

IA-32: distance specified in 1, 2, 4 bytes in 2's complement

distance is calculated from NEXT instruction (since next instruction is fetched before determining current instruction is a jump)

→ What is the distance (in hex) encoded in the `jne` instruction?

```
        Assembly Code          Address    Machine Code
        cmpl  %eax, %ecx
        jne .L1                0x_B8      75 ??   04
        movl  $11, %eax        0x_BA
        movl  $22, %edx        0x_BC
.L1:                           0x_BE
```

0xBE - 0xBA = 0x04

→ If the `jb` instruction is 2 bytes in size and is at 0x08011357 and the target is at 0x8011340 then what is the distance (hex) encoded in the `jb` instruction?

~~340 − 357~~

340 − 359 ~~359 − 340~~

-19

~~010011 101100 +1 00001101 (-19)~~

0001 1001

1111 1110 0110

111110 0111

0xE

this example is wrong somehow?!?!?

Converting Loops

→ Identify which C loop statement (for, while, do-while) corresponds to each goto code fragment below.

```
loop1:
    loop_body
    t = loop_condition
    if (t) goto loop1:
```

do-while
do {

} while( );

```
    t = loop_condition
    if (!t) goto done:
loop2:
    loop_body
    t = loop_condition
    if (t) goto loop2
done:
```

while ( ) {

}

```
    loop_init
    t = loop_condition
    if (!t) goto done:
loop3:
    loop_body
    loop_update
    t = loop_condition
    if (t) goto loop3
done:
```

for loop

Most compilers (gcc included) base loop assembly code on do-while form