CS 354 - Machine Organization & Programming
Tuesday Oct 31 and Thursday Nov 2, 2023

Midterm Exam - Thurs Nov 9, 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**

A09 GF18 and p4B_worksheet_completed.pdf

Homework hw4: DUE on or before Monday, Nov 6

Homework hw5: will be DUE on or before Monday, Nov 13

Project p4A: DUE on or before Friday, Nov 3

Project p4B: DUE on or before Friday, Nov 10

Learning Objectives

- learn low-level details of program execution
- identify assembly language data formats
- identify IA-32 registers, by name and usage
- identify size and type of operand by name and syntax
- learn basic assembly language instructions: mov, push, pop, leal, arithmetic
- learn basic assembly language control instructions: cmp, test, set, jmp, br
- interpret and trace sequence of assembly code
- intepret and explain memory addressing modes by name and syntax
- able to encode target for control instructions

This Week

C, Assembly, & Machine Code - L16-10 Low-level View of Data Registers Operand Specifiers & Practice L18-7 Instructions - MOV, PUSH, POP 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	

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

- var are decl. of specific type, char int double
- types can be complex composites

Machine's View

- mem is an array of bytes where each elt is a byte

❋ *Memory contains bits that do not* instructions from data or ptrs

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

1. by how it is accessed : is it instr. fetch vs operand load

2. by instr. itself

Assembly Data Formats

	C	IA-32	Assembly Suffix	Size in bytes
☆	`char`	byte	b	1
	`short`	word	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, usually 12 for align.

❋ *In IA-32 a word* is 2 bytes

Registers

IA-32

What? Registers

- fastest memory, directly access by ALU
- can store 1, 2, or 4 bytes

General Registers

pre-named locations that store 32 bit values

8 bit

“e” extended

new 32 bit registers

	bit 31	16 15	8 7	0
%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	dest. index	%di		
%esp	stack ptr	%sp		
%ebp	base ptr	%bp		

orig. reg.

16 bit

Program Counter

%eip instr ptr to next instr.

Condition Code Registers

1-bit register that stores status of most recent operation

ZF

SF

OF

CF

Operand Specifiers

What? Operand specifiers are

- S source, spec. location of source (read)
- D destination, spec. location of dest (write)

Why?

enable instr. to specific constants, registers, mem locations

How?

dec hex octo
10, 0xAF, 060

1.) IMMED specifies an operand value that's a constant in C's literal format

specifier	**operand value**
$*Imm*	*Imm*

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

specifier	**operand value**
%E$_a$	R[%E$_a$]

3.) Memory specifies an operand value that's in memory at effective addr

specifier	**operand value**	**effective address**	**addressing mode name**
Imm	M[EffAddr]	*Imm*	Absolute (means mem addr)
(%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 base + index
Imm(%E$_b$,%E$_i$)	M[EffAddr]	*Imm*+R[%E$_b$]+R[%E$_i$]	indexed + offset
has everything Imm(%E$_b$,%E$_i$,s)	M[EffAddr]	*Imm*+R[%E$_b$]+R[%E$_i$]*s	scaled index : offset + base + (index * scale)
(%E$_b$,%E$_i$,s)	M[EffAddr]	R[%E$_b$]+R[%E$_i$]*s	no offset
Imm(,%E$_i$,s)	M[EffAddr]	*Imm*+R[%E$_i$]*s	no base
(,%E$_i$,s)	M[EffAddr]	R[%E$_i$]*s	no base, no offset

Scale factor 1, 2, 4, 8

Imm is the offset value

Eb is base register (starting addr)

Operands Practice

godbolt.org

for C → assem. code

Given: MM

Memory Addr	Value
0x100	0x FF
0x104	0x AA
0x108	0x 11
0x10C	0x 22
0x110	0x 33

CPU

Register	Value
%eax	0x 104
%ecx	0x 1
%edx	0x 4

→ 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	Mem: Indir	0x104
	2. `0xF8(,%ecx,8)`		Imm + R[%ecx] * 8	-8 + 8 = 0x0
☆	3. `%edx`	0x4	register	
☆	4. `$0x108`	0x 108	immed	
	5. `-4(%eax)`	0xFF	Imm + base	R[%eax] - 4 = 0x100
☆	6. `4(%eax,%edx,2)`	0x33	Mem: Scaled index	0x104 + (0x4 * 2) + 4 0x 104 + 0x8 + 0x4 0x10C + 0x4 0x 110
	7. `(%eax,%edx,2)`	0xFF	Base + offset * size	0x104 + 0x4 * 2 = 0x110
	8. `0x108`	0x11	Absolute	
	9. `259(%ecx,%edx)`			

Instructions - MOV, PUSH, POP

What? These are instructions to copy data from S to D

Why? To enable info to be moved around in our mem. and registers

How?

instruction class	operation	description
MOV S, D	D ← S	Move (copy) S to D
mov b - byte	mov w - word (2 byte)	mov l - double word (4 byte)
MOVS S, D	D ← sign-extend S	move S to D
movsbw - byte to word	movsbl	movswl
MOVZ S, D	D ← zero-extend S	move S to D
pushl S	move the stack ptr R[%esp] ← (R[%esp] - 4) write the stuff into the ptr M[R[%esp]] ← S	pushes to the stack
popl D	D ← M[R[%esp]] R[%esp] ← R[%esp] +4	pops most reacent push

Practice with Data Formats

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

b, w, or l   S   D

1. `movl   %eax, %esp`   4b register, 4b register
2. `pushl  $0xFF`   1b, push is only l
3. `movw   (%eax), %dx`   mem, 2b register — makes it mem
4. `movb   (%esp, %edx, 4), %dh`   mem, 1b register
5. `movb   0x800AFFE7, %bl`
6. `movw   %dx, (%eax)`
7. `popl   %edi`

✵ *Focus on register type operands*

Operand/Instruction Caveats

Missing Combination?

→ Identify each source and destination operand type combinations.

1. `movl $0xABCD,%ecx` imm, reg
2. `movb $11,(%ebp)` imm, mem
3. `movb %ah,%dl` reg , reg
4. `movl %eax,-12(%esp)` reg , mem
5. `movb (%ebx,%ecx,2),%al` mem, reg

→ What combination is missing?

IA - 32 does not permit mem , mem

Instruction Oops!

→ What is wrong with each instruction below?

1. `mov~~l~~` b `%bl,(%ebp)`
2. `movl %ebx,$0~~xA1~~FF` D can't be imm
3. `movw %dx,%eax` 2b 4b
   movzwl (= sign ext.)
4. `movb $0x11,~~(%ax)~~` must be 32-bit
5. `movw (%eax),(%ebx,%esi)` mem to mem not allowed
6. `movb %s~~h~~, %bl`
   No SH

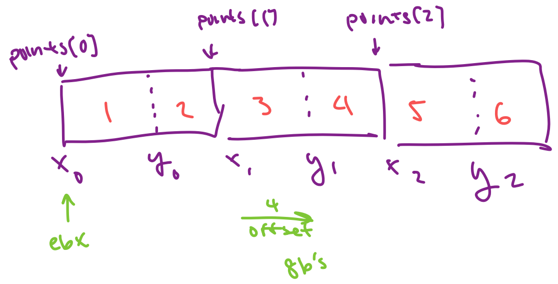
Instruction - LEAL

Load Effective Address

double word
l

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

LEAL vs. MOV



```
struct Point {
   int x;
   int y;
} points[3];
```

creates 1D array of pts

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

= 1

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

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

addr

copies addr into D

LEAL Simple Math

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

almost equivalent

- ebx is not changed in leal
- subl+movl made an arithm. which changes condition codes

→ Suppose register %eax holds x and %ecx 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       D * (-1)
NOT D        D <-- ~D       flip bits
```

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
```

S D
sub (x, y)   y = y - x

Given:

MEM

0x100	0xFF
0x104	0xAB
0x108	0x10

CPU

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

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

1. `incl 4(%eax)`   mem 0x104 : 0xAC
2. `addl %ecx,(%eax)`   (S, D)   mem 0x100 : 0xFF + 0x1 = 0x100
3. `addl $32,(%eax,%edx,4)`   in base 10   mem 0x100 + (0x2 * 4_{10}) = 0x108 : 0x10 + 0x20 = 0x30
4. `subl %edx,0x104`   (S, D)   mem 0x104 : 0xAB - 0x2 = 0xA9   ↑ m(0x104)

D = D - S

Shift Operations

- move bits left or right by K positions, 1 ≤ K < 32
- for fast mult. and div. by powers of 2

```
... 0 0 1 1 0 = 6
      x     2
left shift ... 0 1 1 0 0 = 12

.. 0 0 1 1 0 = 6
      ÷     2
right shift .. 0 0 1 1 = 3
```

logical shift (zero fill)

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

equiv

arithmetic shift (sign-fill)

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

EXAM 2

```
           1111 1111 1111 0000 = -16
SAL 1,D    1111 1111 1111 1000 = -8     SHL
```

add a sign b/c sign matters

Instructions - CMP and TEST, Condition Codes

What?

- compare values arithmetically (subtract (cmp)) or logicaly (AND (test))
- only sets cond code registers, does not change operand

Why?

to enable relational & logical operations

How? sub S, D D ← D - S

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

like subtract, only sets CC

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

like and, only sets C.C.

➢ What is done by `testl %eax, %eax` sets C.C.

Condition Codes (CC)

ZF: zero flag	result is 0
CF: carry flag	result has unsigned overflow
SF: sign flag	" is < 0
OF: overflow flag	result has signed overflow

Instructions - SET

What?

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

How?

there will be cheat sheet on final exam

1 byte register → ah al / ch cl / dh dl / bh bl

```
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

```
setb D   setnae   D <-- CF   unsigned      <  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
```

Signed (2's Complement) Comparisons

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

Comp. 4 bytes   D ← D − S

```
1. cmpl %ebx,%eax
        S    D
   CC ← S1 - S2

2. setl %cl     cl = 0
                S1 - S2 = (-)

3. movzbl %cl,%ecx
   move zero fill cl
     to    ecx
```

```
   1 1
   0 1 1 0            0 1 1 0
 + 0 0 1 0   ←      - 1 1 1 0
 ---------          ---------
 1 0 0 0              0 0 0
```

S2: ebx $\boxed{0110}$ $\boxed{1110}$

S1: eax $\boxed{0110}$ $\boxed{0110}$

ZF = 1
SF = 0

not eq.
ZF = 0
SF = 1
CF = 1
OF = 0

Instructions - Jumps

What? transfer prog execution to another location (instr.)

target: desired location

Why? enables selection & repetition, func. calls

How? Unconditional Jump just jmp

indirect jump:

`jmp *Operand` $\xrightarrow{\text{value}}$ %eip

jmp *%eax   reg value is target

jmp *(%eax)   reg value is mem addr w/ target

direct jump: target is addr of inst

```
jmp Label       jmp .L1
     ⋮
.L1:
```

How? Conditional Jumps

- jump if cond is met (based on CC set previously)
- can only be a dir. jump

both:	je Label	jne Label	$\overset{\text{signed - neg}}{\text{js}}$ Label	$\overset{\text{not signed - not neg}}{\text{jns}}$ Label
unsigned:	$\overset{\text{below}}{\text{jb}}$ Label	$\overset{\text{below, eq.}}{\text{jbe}}$ Label	$\overset{\text{above}}{\text{ja}}$ Label	$\overset{\text{above, eq.}}{\text{jae}}$ Label
signed:	$\overset{\text{less}}{\text{jl}}$ Label	jle Label	$\overset{\text{greater}}{\text{jg}}$ Label	jge Label

Encoding Targets

What? technique used by dir jmp for specific target

Absolute Encoding target is 32 bit addr

Problems?

- code is not compact - target requires 4 bytes
- code cannot be moved w/o changing target

Solution? Relative Encoding

- target is specified as distance from jmp instr. to target

IA-32: dist must be specified in 1, 2, or 4 bytes
2's comp

- dist is calculated from inst. imm. after jump inst.

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

```
      Assembly Code              Address    Machine Code
      cmpl   %eax, %ecx
not eq jne .L1                   0x_B8      75 ??04
      movl   $11, %eax           0x_BA                    eip
      movl   $22, %edx           0x_BC
.L1:                             0x_BE
                      jmp happens after jmp instr.
```

$$\begin{array}{r} BE \\ -BA \\ \hline 04 \end{array}$$

→ 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?

high
357 ← eip 359
340
low

- eip moves to next instr

$$\begin{array}{r} 359 \\ -340 \\ \hline 019 \end{array}$$

```
0011 0101 1001
0011 0100 0000
--------------
0000 0001 1001
     ↓ 2's comp
1111 1110 0111
```

0xE7

Converting Loops

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

DO WHILE

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

do {

loop body

} while (loop cond) ;

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

// does body 0 or more (while)

while (loop cond) {

loop body

}

for

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

Most compilers (gcc included)