CS 354 - Machine Organization & Programming
Tuesday Mar 19 and Thursday Mar 21, 2024

Midterm Exam - Thurs April 4, 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**
- **Exam room information will be sent via email by Friday**

A09 GF18 and p4B_worksheet_completed.pdf

Homework hw4: DUE on or before Monday, 3/18

Homework hw5: will be DUE on or before Monday, April 8

Project p4A: DUE on or before Thursday, Mar 21

Project p4B: DUE on or before Friday, Apr 5

Learning Objectives

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

This Week

Memory Mountain (from L08) 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

- high level language - medium level
- makes us much more productive
- portable - can compile on diff machines

→ What aspects of the machine does C hide from us?
machine instruction, addressing, register used flags

Assembly (ASM)

- human readable
- machine dependent

→ What ISA (Instruction Set Architecture) are we studying? IA-32, x86

→ What does assembly remove from C source?
high level language constructs
- logical control structures
- local variable names and data types
- composites, arrays, structs, union

→ Why Learn Assembly?
1. better understand the stack
2. identify code inefficiencies and vulnerabilities
3. understand compiler optimisations better

Machine Code (MC) **is**

- elementary CPU instructions and data in binary
- the unique encoding that a particular machine understands

→ How many bytes long is an IA-32 instructions? 1 byte to 15 bytes

Low-Level View of Data

C's View

- variables are declared of a specific type
- types can be complex composites built from arrays and structs/union

Machine's View

memory is array of bytes indexed by address

where each element is a byte

❋ *Memory contains bits that do not* distinguish instructions from data from pointer

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

1. by how it's accessed — if instruction is memory pointed to then assumes it is instruction and execute as instruction
   instruction fetch vs operand load
2. by the instruction itself

Assembly Data Formats

C	IA-32	Assembly Suffix	Size in bytes
`char`	byte	b	1
`short`	word	w	2
`int`	double word	l (small L)	4
`long int`	double word	l (small L)	4
`char*`	double word	l (small L)	4
`float`	single precision	s	4
`double`	double prec	l (small L)	8 (quad word)
`long double`	extended prec	t	10 (typically implemented as 12 byte)

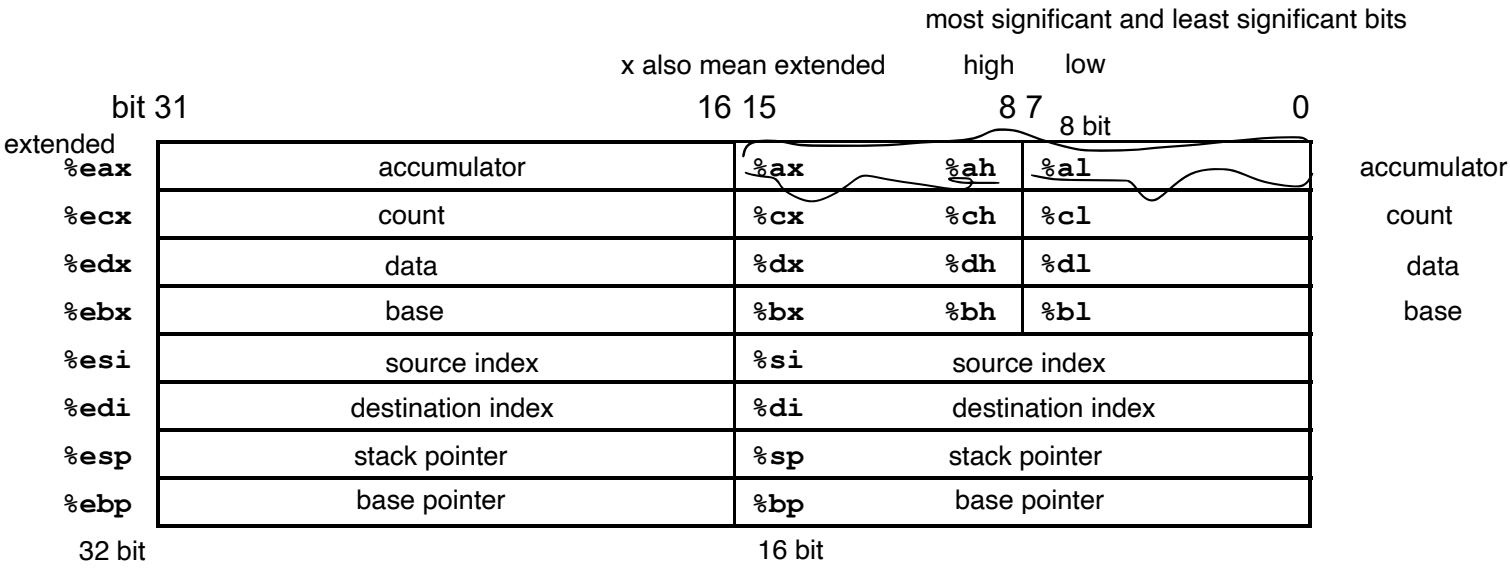
❋ *In IA-32 a word* is actually 2 bytes!

Registers

What? Registers

- fastest memory directly accessed by the ALU
- can store 1, 2 or 4 bytes of data, or 4 bytes for addresses (addressed always 4 bytes)

General Registers pre-named locations that store up to 32 bit values



Program Counter `%eip` extended instructor pointer

holds the next instruction

Condition Code Registers

1 bit registers that store status of most recent operation

ZF - Zero Flag - set if most recent result is 0
SF - Sign Flag - set if most recent result is negative
OF - Overflow Flag - set if most recent result cause unsigned overflow (too large a num to be represented
CF - Carry Flag - set if 2's complement carry out of it

Operand Specifiers

operand is an input / value (eg add 2 numbers)

What? Operand specifiers are

- S Source - can have 2 - specific location of src operand
- D Destination - only can have 1 - specific location of dest operand

Why?

Enables instructions to specify:
- Constants (Immediates), Registers, Main Memory locations

How?

1.) IMMED (immediate) specifies an operand value that's a constant

13 (decimal)
0x13 (hex)
013 (octal)

specifier	**operand value**	
$*Imm*	*Imm*	in C's literal format

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

%ebb, %ax, %cl

specifier	**operand value**
$\%E_a$	R[$\%E_a$]

% says in register, Ea is register name

3.) MEMORY specifies an operand value that's in memory at the effective address

specifier	**operand value**	**effective address**	**addressing mode name**	
Imm	M[EffAddr]	*Imm*	ABSOLUTE	memory addr part of instruction
($\%E_a$)	M[EffAddr]	R[$\%E_a$]	Indirect	effective address in register
Imm($\%E_b$)	M[EffAddr]	*Imm*+R[$\%E_b$]	Base + Offset	address in register + 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	Base + Index + Offset
Imm($\%E_b$,$\%E_i$,s)	M[EffAddr]	*Imm*+R[$\%E_b$]+R[$\%E_i$]*s	Scaled Index	Base + Index*s + Offset
($\%E_b$,$\%E_i$,s)	M[EffAddr]	R[$\%E_b$]+R[$\%E_i$]*s	No Offset	Base + Index*s
Imm(,$\%E_i$,s)	M[EffAddr]	*Imm*+R[$\%E_i$]*s	No Base	Index*s + Offset
(,$\%E_i$,s)	M[EffAddr]	R[$\%E_i$]*s	No Base, No Offset	Index*s

scale factor: 1,2,4,8, 1 is char, 2 shorts, 4 int, 8 double
Imm is offset value
Eb is base register (starting address)

Operands Practice

Given:

MAIN MEMORY

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

CPU

Register	Value
%eax	0x104
%ecx	0x 1
%edx	0x 4

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

	Operand	Value	Type:Mode	Effective Address
1.	`(%eax)`	0xAA	MEM	0x104
2.	`0xF8(,%ecx,8)`	0xFF	MEM: No Base	0xF8 + 0x1 * 8 = 0x100
3.	`%edx`	0x4	REG	—
4.	`$0x108`	0x108	IMM	
5.	`-4(%eax)`	0xFF	MEM: Base + Offset	-4 + 0x104 = 0x100
6.	`4(%eax,%edx,2)`	0x33	MEM: Scaled Index	0x104 + 0x4 * 2 + 4 = 0x110
7.	`(%eax,%edx,2)`	0x22	MEM: No Offset	0x104 + 0x4 * 2 = 0x10C
8.	`0x108`	0x11	absolute	0x108
9.	`259(%ecx,%edx)`	0x11	offset base + index	259 + 0x1 + 0x4 = 0x103 + 1 + 4 = 0x108

Instructions - MOV, PUSH, POP

What? These are instructions to copy data from src (S) to dest (D)

Why? To enable data to move around between register locations

How?

instruction class	operation	description
MOV S, D	D <- S	moves (copy) from S to D

have to be same register type — src and dest must be same size

movb, movw, movl - move 1 byte, move 2 byte, move 4 byte

MOVS S, D — D <- sign-filled S — src is smaller than dest — takes most sig bit and fills rest, eg 1000 copied to 1111 1000

movsbw, movsbl, movswl - 1 to 2, 1 to 4, 2 to 4

MOVZ S, D — D <- zero-filled S — src is smaller than dest — fills with zeros, eg 1000 copied to 0000 1000

movzbw, movzbl, movzwl - 1 to 2, 1 to 4, 2 to 4

pushl S — push to top of stack:
1. grow stack
2. copy value from S

R[%esp] <- (R[%esp] - 4) ===> stack grows downward so minus

M[R[%esp]] <- S

```
subl $4, %esp
movl %eax, (%esp)
```

popl D

D <- M[R[%esp]]

R[%esp] <- (R[%esp] + 4) ===> shift up since value copied out

push to top of stack:
1. copy value to D — `movl (%esp), %eax`
2. shrink stack — `addl $4, %esp`

Practice with Data Formats

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

```
1. movl  %eax, %esp
2. pushl  $0xFF
3. movw  (%eax), %dx
4. movb  (%esp, %edx, 4), %dh
5. movb  0x800AFFE7, %bl
6. movw  %dx, (%eax)
7. popl  %edi
```

register determines how many bytes to be copied

memory always 32 bit register - but dx is 16 bits so 2 bytes

❋ *Focus on register type operands* to determine size (suffix)

Operand/Instruction Caveats

Missing Combination?

→ Identify each source and destination operand type combinations.

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

→ What combination is missing?

mem -> imm imm -> imm reg -> imm imm cant be a dst

mem -> mem not in IA-32

Instruction Oops!

→ What is wrong with each instruction below?

1. `movl %bl,(%ebp)`	%bl too small for double word (l)	fix: change to movb
2. `movl %ebx,$0xA1FF`	reg -> imm: not ok	
3. `movw %dx,%eax`	wrong size 2 bytes -> 4 bytes, must sign/zero fill	fix: change to movzwl
4. `movb $0x11,(%ax)`	must use 32-bit address for mem addresses	fix: (%eax) instead of (%ax)
5. `movw (%eax),(%ebx,%esi)`	mem -> mem: not ok	
6. `movb %sh, %bl`	no %sh register	

Instruction - LEAL

Load Effective Address

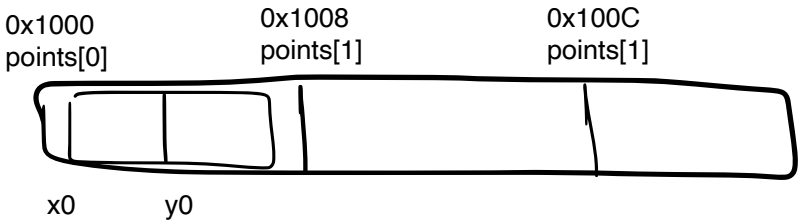
```
leal S,D     D <-- &S
```

move address of source operand into destination

no memory access, no c-flag changes (doesnt affect condition flags)

LEAL vs. MOV

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




```
int y = points[i].y;
```

```
mov 4(%ebx,%ecx,8),%eax
```

0x1000 i sf

offset gets to y

mem -> register

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

movl 4(%ebx, %ecx, 8), %eax

val 4 -> eax

```
int *py = &points[i].y;
```

```
leal 4(%ebx,%ecx,8),%eax
```

copy addr of y to eax

addr of 4 -> eax

LEAL Simple Math

wont change flags

```
leal  -3(%ebx), %eax
```

will change flags

```
subl $3, %ebx
movl %ebx, %eax
```

→ 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 + 8y
2. `leal 12(%eax,%eax,4),%ebx` 5x + 12
3. `leal 11(%ecx),%ebx` y + 11
4. `leal 9(%eax,%ecx,4),%ebx` 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
```

incl %eax (add 1 to whatever in %eax) incb (%edx)

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    bitwise
OR S,D      D <-- D | S    bitwise
AND S,D     D <-- D & S    bitwise
```

Given:

0x100	0xFF	**%eax**	0x100
0x104	0xAB	**%ecx**	0x1
0x108	0x10	**%edx**	0x2

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

base + offset 0x104

1. `incl 4(%eax)` — DEST: 0x104 RESULT: 0xAC

2. `addl %ecx,(%eax)` — DEST: 0x100 RESULT: 0x100

base + scaled index 0x108

3. `addl $32,(%eax,%edx,4)` — DEST: 0x108 RESULT: 0x30

absolute addr

4. `subl %edx,0x104` — DEST: 0x104 RESULT: 0xA9

Shift Operations

- k can be any value from 0 to 31 [0,31]
  if SHLI $33, %eax —> will just look at lower 5 bits, so will only shift left by 1
-

logical shift

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

SHLI $2, %eax <==> IMUL $4, %eax

arithmetic shift

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

arithmetic shift right keeps sign
eg if 1010 shift right 2, will become 1110

Instructions - CMP and TEST, Condition Codes

What?

- compare arithmetically (CMP) or logically (TEST)

- change condition Flags

Why?

enables ability to do comparisons

How?

`CMP S2,S1` `CC <-- S1 - S2` CMP just does subtraction

does not store anywhere but see how it affects flags

`TEST S2,S1` `CC <-- S1 & S2` TEST just does AND

➢ What is done by `testl %eax, %eax`

used to test for zero - if %eax is 0, AND 0 and 0 gives 0 so ZERO flag would be set

Condition Codes (CC)

ZF: zero flag a + b = 0

CF: carry flag unsigned total is too big

0111 + 0010 = 1000
+7 + +2 = 1001 (?) but will set sign flag to 1

SF: sign flag high bit of the result

OF: overflow flag if (a<0 == b < 0) ==> a and b have the same sign
AND (total < 0 != a< 0) ==> overflow condition

total = a + b
if both a and b is 0 or add up to 0, ZF will be set
if a + b is too big to store in DEST register, CF will be set
SF is the result of the highest bit of the register

Instructions - SET

What?

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

How?

stores Zero Flag into register

```
sete D   setz     D <-- ZF                  == equal
setne D  setnz    D <-- ~ZF                 != not equal
```

sete %al

is register 0?

is register not 0?

stores Sign Flag into register

```
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

setBELOW

nae: not above or equal

```
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
```

Signed (2's Complement) Comparisons

setLESS

```
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)

1. `cmpl %ebx,%eax` — CMP does subtraction, so a-b<0 sets SF
2. `setl %cl` — %cl will have 1
3. `movzbl %cl,%ecx` — zero extending out the high 24 bits

Instructions - Jumps

What?

transfer control to somewhere else in code

target: the desired next instruction

Why? enables us to do loops, flow control

How? Unconditional Jump always jump

indirect jump:

target needs to be in a register or memory location

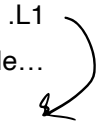
`jmp *Operand` %eip <- operand

jmp *%eax == address of instruction i want executed next is stored in eax

jmp *(%eax) == go to address where eax is pointing, and whatever is found at this address and puts in eip

direct jump:

```
   jmp Label
    jmp .L1
.L0: code…
.L1:
```



How? Conditional Jumps

- jump when some condition is met
- based on condition flags

```
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
```

test %eax, %eax — AND both values, checks if eax is zero, jump if so
je .LABEL

Encoding Targets

What?

jmp 0x2000

Absolute Encoding

target is specified as a 32-bit address
absolute address to go to

Problems?

- code is not compact
- code cannot be moved without chaging the targets

Solution? Relative Encoding

IA-32: relative distance we jump is 1,2, or 4 bytes (SIGNED)

distance is calculated from the address of the next instruction

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

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

→ 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?

0x40-0x59 = -19 (base16)
0001 1001 (flip) -> 1110 0110 (+1) -> 1110 0111 (E7)

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 {
        loop_body()
      } while (loop_conidition);
```

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

```
      while(loop_condition) {
        body()
      }
```

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

```
      for (loop_init, test, loop_update) {
        loop_body();
      }
```

Most compilers (gcc included)

can write any loops as other forms
most compilers generate in the form of do_while