CS 354 - Machine Organization & Programming
Tuesday Oct 29 and Thursday Oct 31, 2024

Midterm Exam - Thurs Nov 7th, 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, 11/4

Homework hw5: will be DUE on or before Monday, ____

Project p4A: DUE on or before Friday, Nov 1

Project p4B: DUE on or before Sunday, Nov 10

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

Stride,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

- **is HLL (high level language) that enable us to be more productive coders**
- **helps us write correct code with syntax and type checking**
- **can be compiled and run on different architectures (portable)**

→ What aspects of the machine does C hide from us?
low-level machine details

Assembly (ASM)

- **is human readable representation of MC**
- **is very machine dependent**

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

→ What does assembly remove from C source? **HLL constructs**

→ Why Learn Assembly?

1. **better understand the stack**
2. **identify code inefficiencies and vulnerabilities**
3. **understand compiler optimization options**

Machine Code (MC) **is**

- **elementary cpu instructions and data in binary (typically generated by assembler)**
- **the unique encodings that a particular machine understands and can execute**

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

Low-Level View of Data

C's View

- **variables are**
- **types can be**

Machine's View

- memory is like
- where each element is

✵ *Memory contains bits that do not*

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

1. By how memory is accessed:

2. By the instruction itself:

Assembly Data Formats

C	IA-32	Assembly Suffix	Size in bytes
`char`	byte		
`short`	word		
`int`	double word		
`long int`	double word		
`char*`	double word		
`float`	single precision		
`double`	double prec		
`long double`	extended prec		

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

Registers

What? Registers

General Registers

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

	bit 31 16	15 8	7 0
`%eax`		`%ax` `%ah`	`%al`
`%ecx`		`%cx` `%ch`	`%cl`
`%edx`		`%dx` `%dh`	`%dl`
`%ebx`		`%bx` `%bh`	`%bl`
`%esi`		`%si`	
`%edi`		`%di`	
`%esp`		`%sp`	
`%ebp`		`%bp`	

Program Counter `%eip`

Condition Code Registers

Operand Specifiers

What? Operand specifiers are

- S
- D

Why?

How?

1.) specifies an operand value that's

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

2.) specifies an operand value that's

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

3.) specifies an operand value that's

specifier	**operand value**	**effective address**	**addressing mode name**
Imm	M[EffAddr]	*Imm*	
$(\%E_a)$	M[EffAddr]	$R[\%E_a]$	
$Imm(\%E_b)$	M[EffAddr]	$Imm+R[\%E_b]$	
$(\%E_b,\%E_i)$	M[EffAddr]	$R[\%E_b]+R[\%E_i]$	
$Imm(\%E_b,\%E_i)$	M[EffAddr]	$Imm+R[\%E_b]+R[\%E_i]$	
$Imm(\%E_b,\%E_i,s)$	M[EffAddr]	$Imm+R[\%E_b]+R[\%E_i]*s$	
$(\%E_b,\%E_i,s)$	M[EffAddr]	$R[\%E_b]+R[\%E_i]*s$	
$Imm(,\%E_i,s)$	M[EffAddr]	$Imm+R[\%E_i]*s$	
$(,\%E_i,s)$	M[EffAddr]	$R[\%E_i]*s$	

Operands Practice

Given:

Memory Addr	Value	Register	Value
0x100	0xFF	%eax	0x104
0x104	0xAA	%ecx	0x1
0x108	0x11	%edx	0x4
0x10C	0x22		
0x110	0x33		

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

Instructions - MOV, PUSH, POP

What? These are instructions to

Why?

How?

instruction class	operation	description
MOV S, D		
MOVS S, D		
MOVZ S, D		
pushl S		
popl D		

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
```

❋ *Focus on register type operands*

Operand/Instruction Caveats

Missing Combination?

→ Identify each source and destination operand type combinations.

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

→ What combination is missing?

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. `movw (%eax),(%ebx,%esi)`
6. `movb %sh, %bl`

Instruction - LEAL

Load Effective Address

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

LEAL vs. MOV

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


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


        points[1].y;




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

LEAL Simple Math

```
leal  -3(%ebx), %eax        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`
2. `leal 12(%eax,%eax,4),%ebx`
3. `leal 11(%ecx),%ebx`
4. `leal 9(%eax,%ecx,4),%ebx`

Instructions - Arithmetic and Shift

Unary Operations

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

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
```

Given:

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

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

1. `incl 4(%eax)`

2. `addl %ecx,(%eax)`

3. `addl $32,(%eax,%edx,4)`

4. `subl %edx,0x104`

Shift Operations

◆

◆

logical shift

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

arithmetic shift

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

Instructions - CMP and TEST, Condition Codes

What?

◆

◆

Why?

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

CF: carry flag

SF: sign flag

OF: overflow flag

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

```
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

```
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`#compare a and b   a - b
2. `setl %cl`#set %cl to 1 if a < b, otherwise 0
3. `movzbl %cl,%ecx`#zero out remaining bytes of %ecx

Instructions - Jumps

What?

transfer execution to another location in the code

target:

Why?

enables selection, repetition, control flow

How? Unconditional Jump

indirect jump:

```
jmp *Operand
```

direct jump:

```
jmp Label
```

How? Conditional Jumps

◆

◆

```
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
```


Encoding Targets

What?

Absolute Encoding

Problems?

- code is not
- code cannot be

Solution?

IA-32:

→ 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
      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?

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:
```

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

```
      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)base loop assembly code on the do-while forms as shown above.*