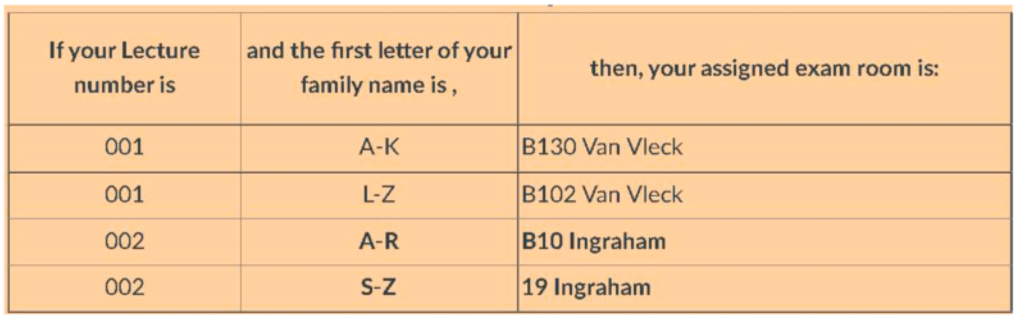
outlineL18-w9TR-student

Thursday, November 3, 2022 4:10 PM

outlineL18-w9TR-student

CS 354 - Machine Organization & Programming
Tuesday Nov 1 and Thursday Nov 3, 2022

Midterm Exam - Thurs Nov 10th, 7:30 - 9:30 pm



If your Lecture number is	and the first letter of your family name is ,	then, your assigned exam room is:
001	A-K	B130 Van Vleck
001	L-Z	B102 Van Vleck
002	**A-R**	**B10 Ingraham**
002	**S-Z**	**19 Ingraham**

- **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, Nov 7

Homework hw5: will be DUE on or before Monday, Nov 14

Project p4A: DUE on or before Friday, Nov 4th

Project p4B: DUE on or before Friday, Nov 11th

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

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	

C, Assembly, & Machine Code

HLL	mnemonic!	In the beginning...
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 that enables more productive coding
- Helps us write code
- Can be compiled and run on different machines

→ What aspects of the machine does C hide from us?
machine instructions
addressing modes
registers, conditional codes

Assembly (ASM)

- Human readable representation of machine code
- Very machine dependent

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

→ What does assembly remove from C source?
- comments, var names · logical control structure → if(){}, switch, loops
no structs, arrays, unions · no data structures

→ Why Learn Assembly?
→ 1. to better understand the stack
2. Identify inefficiencies
3. Understand compiler optimizations

Machine Code (MC) **is**

- elementary CPU instructions → and data in binary.
- the encodings that a particular machine understands

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

Low-Level View of Data

C's View

- Variables declared of a specific type
- types can be complex composites, arrays, structs, unions

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 instructions from data or pointers

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

1. by how it's accessed   INST 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	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)

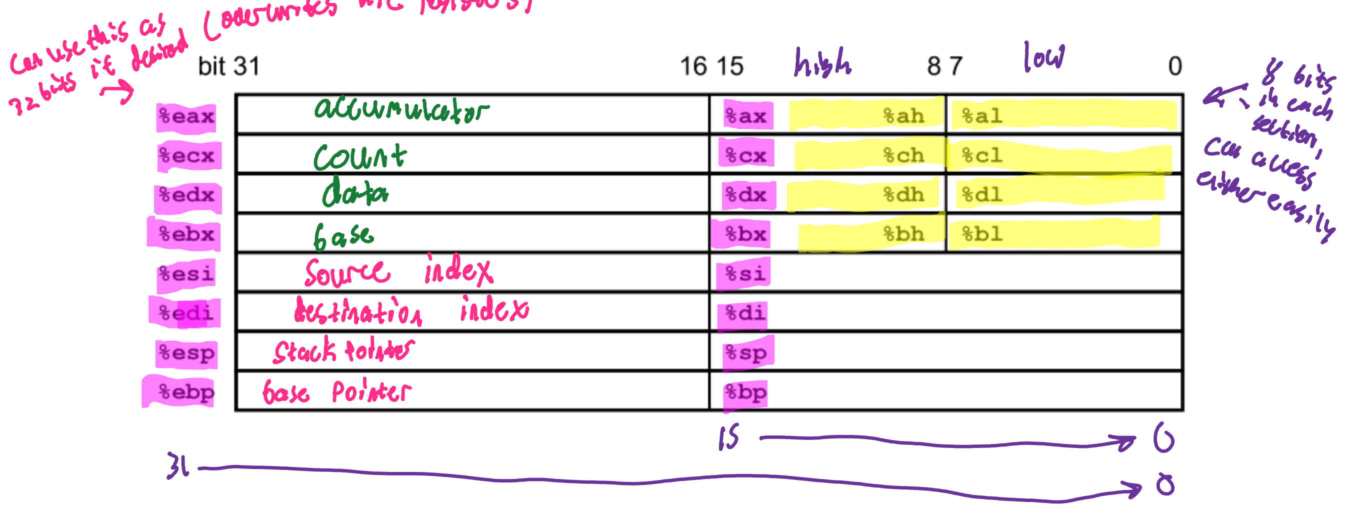
✻ *In IA-32 a word* is actually 2 bytes

Registers

What? Registers
* Fastest memory/storage directly accessed by the ALU

* Can store 1, 2, or 4 bytes of data for addresses

General Registers
Pre-named locations that store up to 32-bit values (overwrites w/L registers)



Program Counter
%eip
extended 32-bit instruction pointer

stores address of next instruction

Condition Code Registers "e-flags"
1-bit registers that store status of most recent ALU operation
↓
recently executed

Can be used for conditional branching

Operand Specifiers

What? Operand specifiers are

- S → source, specifies location to be used by instruction
- D → destination, where result is to be stored

Why? enables instr to specify constants, register & memory location

How? IA-32 has 3 kinds of operand specifiers

1.) IMMEDIATE specifies an operand value that's a constant is value in the instruction, immediate in C's literal format

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

$10 → $0x10

$071

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

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

R - register a - arbitrary

3.) MEMORY specifies an operand value that's

specifier	**operand value**	**effective address**	**addressing mode name**
Imm	M[EffAddr]	*Imm*	ABSOLUTE
(%E$_a$)	M[EffAddr]	R[%E$_a$]	INDIRECT
Imm(%E$_b$)	M[EffAddr]	*Imm*+R[%E$_b$]	BASE + Offset (displacement)
(%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
Imm(%E$_b$,%E$_i$,s)	M[EffAddr]	*Imm*+R[%E$_b$]+R[%E$_i$]*s	scaled index BASE + INDEX·SCALE + IMM
(%E$_b$,%E$_i$,s)	M[EffAddr]	R[%E$_b$]+R[%E$_i$]*s	without offset
Imm(,%E$_i$,s)	M[EffAddr]	*Imm*+R[%E$_i$]*s	without base register
(,%E$_i$,s)	M[EffAddr]	R[%E$_i$]*s	without base + offset

(Imm) Immediate is the offset

E$_b$ is base register, E$_i$ is index register,

s is scale factor → go by 1, 2, 4, 8 bytes

Operands Practice

Given: main memory CPU

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

Register	Value
%eax	0x104
%ecx	0x1
%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)`	0x AA	Indirect	0x104 → we go to address 104
	2. `0xF8(,%ecx,8)`			
*	3. `%edx`	0x4	Register	N/A
*	4. `$0x108`	0x108	IMM	N/A
	5. `-4(%eax)`			
*	6. `4(%eax,%edx,2)`	0x33	Scaled Index	off + b + ri·s 4+ %eax + %edx·s 4+ 0x104 + (0x4·2) 0x104 + 0x8 = 0x10C + 4 = 0x110
	7. `(%eax,%edx,2)`			
	8. `0x108`			
	9. `259(%ecx,%edx)`			

(under `%eax` in item 6: 6)

Notes:

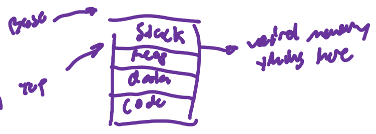
IMM → has $

Memory has parenthesis or a constant without dollar sign

Register has % and no ()s

Instructions - MOV, PUSH, POP

COPY



What? These are instructions to copy data from source to destination

Why? enables info to move from register to/from memory and between registers

How?

instruction class	operation	description
MOV S, D	D ← S	move (copy) S to D
movb, movw, movl		SAME SIZE REGISTERS
MOVS S, D	D ← sign-extended S	move (copy) smaller register to larger
movsbw, movsbl, movswl		
MOVZ S, D	D ← zero-extend S	
movzbw, movzbl, movzbl		
pushl S	R[%esp] ← (R[%esp-4]) M[R[%esp]] ← S	make room at top of stack copy src to top
popl D	D ← M[R[%esp]] R[%esp] ← (R[%esp]+4)	copy from top of stack to D shrink stack by 4

Practice with Data Formats

all push/pop will be 4 bytes

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

```
l  1. mov_  %eax, %esp
l  2. push_  $0xFF
w  3. mov_  (%eax), %dx
b  4. mov_  (%esp, %edx, 4), %dh
b  5. mov_  0x800AFFE7, %bl     ← missing $, moving value
w  6. mov_  %dx, (%eax)
l  7. pop_  %edi
```

✵ *Focus on register type operands* since they can be 1, 2, or 4 bytes (IA-32)

Operand/Instruction Caveats

Missing Combination? S, D

→ Identify each source and destination operand type combinations.

1. movl $0xABCD,%ecx — Imm to Reg → 4 bytes
2. movb $11,(%ebp) — Imm to memory (address) → 1 byte
3. movb %ah,%dl — Register to Register 1 byte
4. movl %eax,-12(%esp) — Register to Mem: 4 byte
5. movb (%ebx,%ecx,2),%al — Mem: Scaled Index to Reg 1 byte

→ What combination is missing?

memory to memory, not possible in IA-32

Instruction Oops!

→ What is wrong with each instruction below?

1. movl %bl,(%ebp) — Instruction + register size don't match
2. movl %ebx,$0xA1FF — Can't copy to immediate
3. movw %dx,%eax — should be movs/movz → sizes don't match
4. movb $0x11,(%ax) — memory address must be 32 bit register
5. movw (%eax),(%ebx,%esi) — no memory to memory in IA-32
6. movb %sh, %bl — Sh not a real register!

Instruction - LEAL

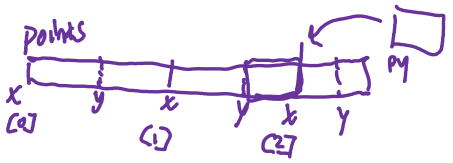
Load Effective Address l (double word - lea L)

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

S must be memory operand

figure out address of S, put in D

LEAL vs. MOV



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

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

scale factor

points   i · 8

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

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

translated with leal

same operands gives us address

compute effective addr

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

Remember scale factor restrictions

Instructions - Arithmetic and Shift

Unary Operations One operand

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

multi by -1

bitwise NOT → flip bits

Binary Operations Two operand

```
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
```

subtract S from D

if one or other but not both → true (exclusive or)

logical OR

logical AND

} bit-wise

Given:

MEM		CPU	
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)` Effective address 0x104 → 0xAC

2. `addl %ecx,(%eax)` R mem add source to destination $0xFF + 0x1 = 0x100$

3. `addl $32,(%eax,%edx,4)` +0x10 $0x100 + 0x2 \cdot 4 = 0x108$ 0x20 0x30

4. `subl %edx,0x104` Subtract source from destination $0xAB - 0x2 = 0xA9$

Shift Operations shift

- move bits from left to right by k positions 0-31
  0 <= k <= 31 k is 1 byte register, l.s. bit of register %ecx, always in %cl
- For fast integer division/multiplication by powers of 2

logical shift (zero fill shift)

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

0110 << 1 → 1100 (12)

0110 >> 1 → 0011 (3)

arithmetic shift

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

-2 1110 << 1 → 1100 -4

1110 >> 1 → 1111 -1 (-2)

Instructions - CMP and TEST, Condition Codes

(CMP) (TEST)

What?

- Compare values arithmetically or logically
- and, sets condition codes (registers)

Why?

to enable relational and logical operations in ASM

How?

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

subtract S2 from S1

like subtract, sets condition codes but doesn't change value

cmpb, cmpw, cmpl

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

like AND, sets C.C. not values

testb, testw, testl

➢ What is done by `testl %eax, %eax`

updates C.C. with true

Condition Codes (CC)

ZF: zero flag if ZF=1, result is 0

CF: carry flag if CF=1, result caused unsigned overflow

SF: sign flag if SF=1, result is negative (2's complement = MSB = 1)

OF: overflow flag if OF=1, result caused overflow

if + plus + is negative → overflow (2's complement overflow)

− plus − is positive

Instructions - SET

What?

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

How?

Set
D if ZF ==1
D if ZF is Ø
D if Sign flag is 1
D if Sign flag is Ø

`sete D`	`D <-- ZF`	Set if equal
`setne D`	`D <-- ~ZF`	Set if not equal
`sets D`	`D <-- SF`	Set if less than 0
`setns D`	`D <-- ~SF`	Set if positive

$\boxed{\text{Unsigned Comparisons:}}$

D if CF is 1

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

$\boxed{\text{Signed (2's Complement) Comparisons}}$

`setl D`	`D <-- SF ^ OF`	set if less than
`setle D`	`D <-- (SF ^ OF) | ZF`	set if less than or equal
`setg D`	`D <-- ~(SF ^ OF) & ~ZF`	set if greater than
`setge D`	`D <-- ~(SF ^ OF)`	set if greater than or equal

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

1. `cmpl %ebx,%eax` — b above %ebx, a above %eax; S2 below %ebx, S1 below %eax; a-b / S1-S2 — compares a-b, sets condition codes
2. `setl %cl` — Set D if previous result is less than
3. `movzbl %cl,%ecx`

move from %cl to %ecx with zero extend
(copy it)

Instructions - Jumps

What? Transfer program execution to another location

target: desired location of next instruction

Why? enables logical control flow (selection + repetition)

How? $\boxed{\textbf{Unconditional Jump}}$ Always jump to the target

indirect jump: Target is in register or memory

```
jmp *Operand          %eip ← %eax
jmp *%eax     register value is target address    %eip ← mem[%eax]
jmp *(%eax)  reg value is memory address with target
```

direct jump: target addr. is in the instruction

```
jmp Label     %eip ← label
jmp .L1

.L1
.cfi
```

$\boxed{\textbf{How? Conditional Jumps}}$

- Jump if condition is met (based on condition codes + instruction)
- Can only be direct 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
```

↑ same meanings as 'set' instructions.

Encoding Targets

What? technique used by direct jump inst. for specific target

Absolute Encoding target specified as specific 32-bit address

Problems?

- code is not compact - target requires 32-bit.
- code cannot be moved without changing target.

Solution? use relative encoding

Target is specified as a distance from jump instruction

(distance can be stored in)

IA-32: 1, 2, 4 bytes

Distance is calculated immediately after jump instruction

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

```
        Assembly Code          Address    Machine Code
        cmpl   %eax, %ecx
    →   jne  .L1        %eip   0x_B8      75 ?? 0x04
        movl   $11, %eax     → 0x_BA
        movl   $22, %edx       0x_BC
   →.L1:                     ↳ 0x_BE   BE
                                     - BA
                                     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?

$$
\begin{array}{r}
0x08011340 \\
-\ 0x08011359 \\
\hline
\end{array}
\approx 20
$$

next instruction : current + size : 0x8011359

$$
\begin{array}{r}
359 \\
-\ 340 \\
\hline
019
\end{array}
\rightarrow \text{make negative } \boxed{-0x19} \leftarrow \text{negative offset}
$$

Copyright © 2016-2022 Jim Skrentny

Converting Loops

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

label

```
loop1:   (Do-while loop)
      loop_body
  --> t = loop_condition
      if (t) goto loop1:
```

do {
body

}while(cond.);

WHILE

```
true  t = loop_condition  cond.
      if (!t) goto done:
                 false
loop2:
      loop_body
      t = loop_condition  cond.
      if (t) goto loop2
done:
```

while(loop){
body
update cond

}

```
   ① loop_init
   ② t = loop_condition
      if (!t) goto done:
loop3:
   ③ loop_body
   ④ loop_update
      t = loop_condition
      if (t) goto loop3
done:
```

for(loop-init; cond; update;){
body

}

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