ECE391 Computer System Engineering Lecture 2

Dr. Zbigniew Kalbarczyk
University of Illinois at Urbana- Champaign

Spring 2017

Lecture Topics

- x86 instructions
- Operate instructions
- Data movement instructions
- Conditional codes
- Control flow instructions
- Assembler conventions
- Code example

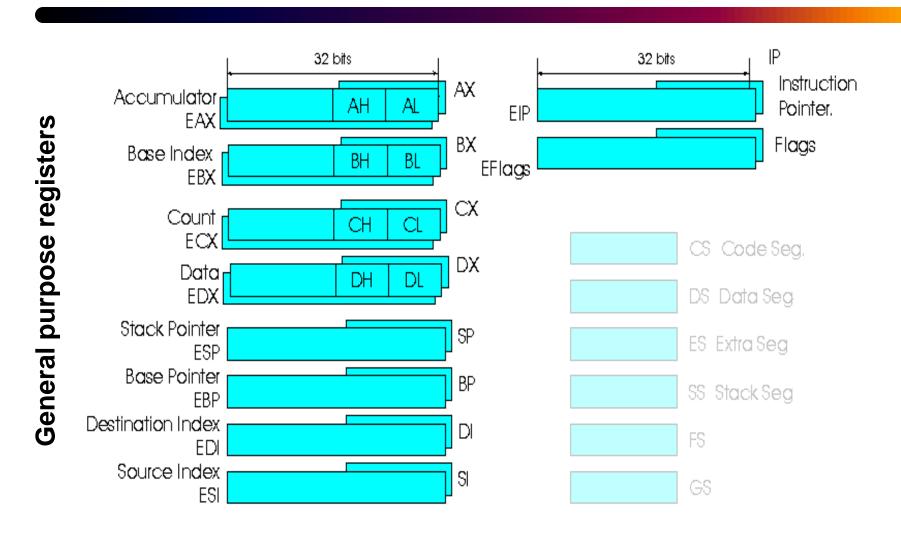
Aministrivia

- MP0
 - In TAs office hours by 01/23
 - you can hand in anytime during office hours

Introduction and Basics

- What is x86? (Intel-32-bit architecture, rather than AMD-64)
 - variable-length instruction encoding (1-16 bytes)
 - small register set: 8 mostly general-purpose
 - 32-bit, byte-addressable address space
 - complex addressing modes
 - many data types supported by hardware
- Most modern ISAs (e.g., MIPS, Sparc, PowerPC) differ on most points (except memory)

Registers



Registers

```
-> extended, i.e., 32-bit
EAX accumulator
                                    instruction pointer
                             EIP
EBX base (of array)
                             EFLAGS flags/condition codes
ECX count (for loops)
EDX data (2<sup>nd</sup> operand)
ESI source index (string copy)
EDI destination index
EBP base pointer (base of stack frame)
ESP stack pointer
```

- Use % as a prefix for registers in assembly
- Other registers: floating-point, MMX, etc. (not discussed in this class)

Data Types

- 8-, 16-, 32-bit unsigned and 2's complement
- IEEE single- and double-precision floating point
- Intel "extended" f.p. (80-bit)
- ASCII strings
- Binary-coded decimal

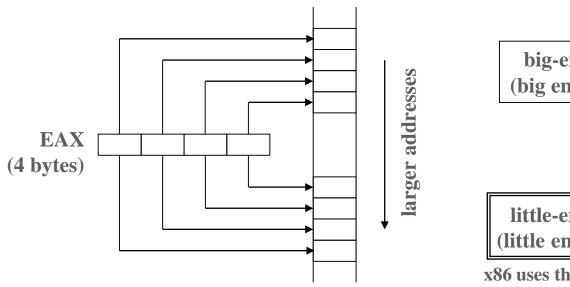
Memory

 Microprocessor addresses a maximum of 2ⁿ different memory locations, where n is a number of bits on the address bus

Memory

- x86 supports byte addressable memory
- byte (8 bits) is a basic memory unit
- e.g., when you specify address 24 in memory, you get the entire eight bits
- when the microprocessors address a 16-bit word of memory, two consecutive bytes are accessed

How are bytes stored to memory?



big-endian (big end first)

little-endian (little end first)

x86 uses this approach

x86 Instructions – Basics

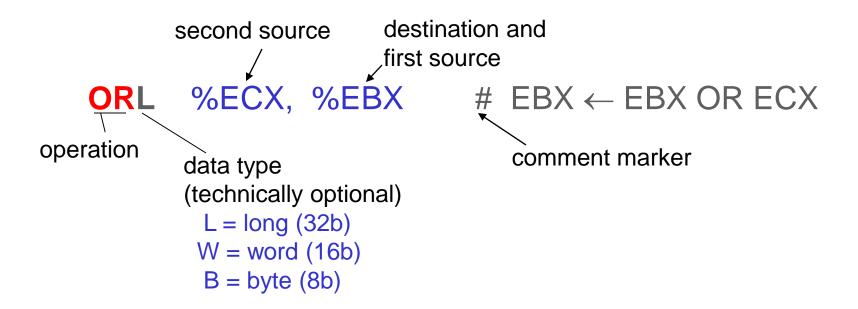
 Operations, data movement, condition codes, control flow, stack ops, data size conversion

Operations

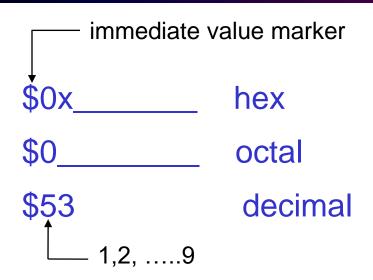
arithmetic	logical	<u>shift</u>
ADD	AND	SHL
SUB	OR	SAR
NEG	NOT	SHR
INC	XOR	ROL
DEC		ROR

 typically 2-operand instructions (destination and one source are the same)

Operations – Example



Immediate Values



- how big can they get?
 - usually up to 32 bits
 - larger constants → longer instructions
 - length of operand must be encoded, too

what does the following instruction do?

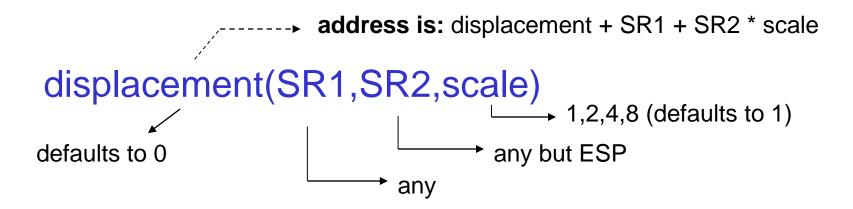
ANDL 0, %EAX

answer is NOT $EAX \leftarrow 0$

instead: $EAX \leftarrow EAX \, AND \, M[0]$ (usually crashes)

Data Movement: Memory Addressing

Memory operand has this general form



Instructions

Examples:

```
MOVW %DX, 0x10(\%EBP) # M[EBP + 0x10] \leftarrow DX MOVB (%EBX,%ESI,4), %CL # CL \leftarrow M[EBX + ESI * 4]
```

Instructions: Examples to Solve

```
EAX \leftarrow M[0x10000 + ECX]
[answer] MOVL 0x10000(%ECX), %EAX
          M[LABEL] \leftarrow DI
[answer] MOVW %DI, LABEL
          ESI \leftarrow LABEL + 4 (two ways!)
[answer] MOVL $LABEL + 4, %ESI
          LEAL LABEL + 4, %ESI
          ESI \leftarrow LABEL + EAX + 4
[answer] LEAL LABEL + 4(%EAX), %ESI
                      expression calculated by assembler;
                      instruction holds one displacement value
```

Condition Codes (in EFLAGS)

Among others (not mentioned in this class)...

SF: sign flag: result is negative when viewed as 2's complement data type

ZF: zero flag: result is exactly zero

CF: carry flag: unsigned carry or borrow occurred (or other, instruction-dependent meaning, e.g., on shifts)

OF: overflow flag: 2's complement overflow (and other instruction-dependent meanings)

PF: parity flag: even parity in result (even # of 1 bits)

What Instructions Set Flags (condition codes)?

- Not all instructions set flags
- Some instructions set some flags!
- Use CMP or TEST to set flags:

```
CMPL %EAX, %EBX # flags ← (EBX – EAX)
TESTL %EAX, %EBX # flags ← (EBX AND EAX)
```

Note that EBX does not change in either case

 What combinations of flags are needed for unsigned/signed relationships comparator?

Control Flow Instructions (1)

 Consider two three-bit values A and B; How to decide if A<B?

	#1	#2	#3	#4	#5	#6
Α	010	010	010	110	110	110
В	-000	<u>-110</u>	-111	-000	-011	<u>-111</u>
C	010	100	011	110	011	111
CF	0	1	1	0	0	1
OF	0	1	0	0	1	0
SF	0	1	0	1	0	1
unsigned <	No	Yes	Yes	No	No	Yes
signed <	No	No	No	Yes	Yes	Yes

Control Flow Instructions (2)

- Note that CF suffices for unsigned <
- What about signed < ?

Answer: OF XOR SF

Control Flow Instructions (3)

Intuition: consider the first bits

```
A: a ... recall OF = a'bc + ab'c'

B: -b ... SF = c

C: c ...

OF \oplus SF = (a'bc + ab'c')c' + (a+b'+c')(a'+b+c)c

= ab'c' + ac + b'c

= ab' + ac + b'c (complementarity (c + c')
```

- Terms correspond to cases in which A < B
 - first term: negative number (a) < non-negative number (b)
 - second term: if subtracting from negative number (a) produces negative answer (c), (b) must be larger than (a)
 - third term: if subtracting a non-negative number (b) produces a negative answer (c), (b) must be larger than (a)

Branch Mnemonics

- Unsigned comparisons: "above" and "below"
- Signed comparisons: "less" and "greater"
- Both: equal/zero

```
unsigned jne jb jbe je jae ja relationship \neq < \leq = \geq > signed jne jl jle je jge jg
```

- in general, can add "n" after "j" to negate sense
- forms shown are those used when disassembling
 - do not expect binary to retain your version
 - e.g., "jnae" becomes "jb"

Other Control Instructions

- Other branches
 - jo jump on overflow (OF)
 - jp jump on parity (PF)
 - js jump on sign (SF)
 - jmp unconditional jump
- Control instructions: subroutine call and return

```
CALL printf # (push EIP), EIP ← printf
```

CALL *%EAX # (push EIP), EIP
$$\leftarrow$$
 EAX

CALL *(%EAX) # (push EIP), EIP
$$\leftarrow$$
 M[EAX]

RET # EIP
$$\leftarrow$$
 M[ESP], ESP \leftarrow ESP + 4

Stack Operations

Push and pop supported directly by x86 ISA

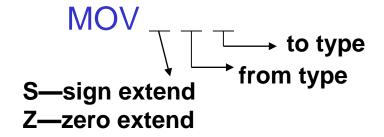
PUSHL %EAX # M[ESP – 4]
$$\leftarrow$$
 EAX, ESP \leftarrow ESP – 4

POPL %EBP # EBP
$$\leftarrow$$
 M[ESP], ESP \leftarrow ESP + 4

PUSHFL # M[ESP – 4]
$$\leftarrow$$
 EFLAGS, ESP \leftarrow ESP – 4

Data Size Conversion

- These instructions extend 8- or 16-bit values to 16- or 32-bit values
- General form



Examples

```
MOVSBL %AH, %ECX # ECX ← sign extend to 32-bit (AH)

MOVZWL 4(%EBP), %EAX # EAX ← zero extend to 32-bit (M[EBP + 4])
```

Assembler Conventions

```
label:
                  requires a colon, and is case-sensitive
                  (unlike almost anything else in assembly)
# comment to end of line
/* C-style comment
    ... (can consist of multiple lines) */
    command separator (NOT a comment as in LC-3)
string "Hello, world!", "me" # NUL-terminated
.byte 100, 0x30, 052 # integer constants of various sizes
.word ...
.long ...
.quad ...
.single ...
                              # floating-point constants
.double ...
If assembly file name ends in .S (case-sensitive!), file is first passed through
```

C's preprocessor (#define and #include)