Lecture 2: x86 instruction set architecture: introduction and instructions.

ECE 391 Computer Systems Engineering

Yih-Chun Hu

University of Illinois

Portions taken from ECE 391 Lecture Notes by Steven Lumetta, Michael Bailey, Zbigniew Kalbarczyk, Wikipedia, the free encyclopedia, David Evans' x86 Assembly Guide, Jonathan Bartlett's Programming from the Ground Up, and the X86 Assembly Wikibook

Administrivia

- MP0
  - **In TAs office hours by 09/01**
  - you can hand in anytime during office hours

At the end of this sequence you should be able to:

- Understand ISAs and their role in computer systems
- Recognize the specifics of x86 ISA instructions and GNU notation
- Demonstrate a working familiarity with x86 assembly by building simple x86 programs
- Follow x86 assembly function calling conventions
- Communicate with devices in x86 assembly

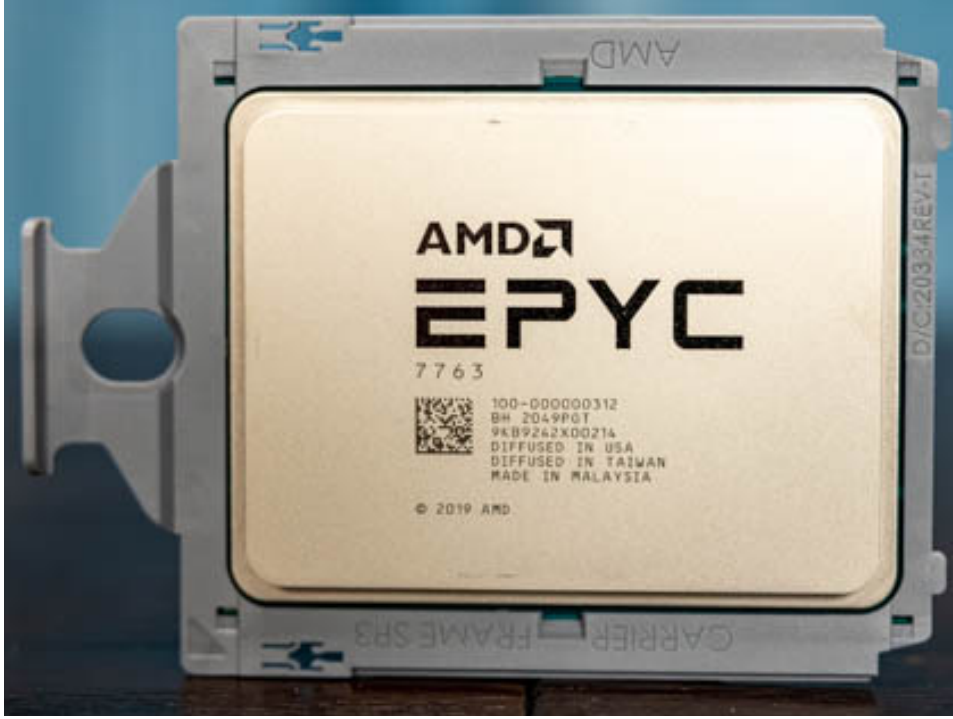
What is an ISA?

RISC v CISC

Instruction Sets

- AMD64
- ARMv[1-8]
- Dec VAX
- Alpha
- IBM System{3,7}
- IBM PowerPC
- Intel i860
- Motorola 68000 family
- MIPS [i-v]
- Sun SPARC

…



Opteron courtesy Konstantin Lanzet, Wikipedia

X86 Characteristics

- variable-length instruction encoding (in bytes)
- small register set: 8 mostly general-purpose
- 32-bit, byte-addressable address space
- complex addressing modes
- many data types supported by hardware

registers

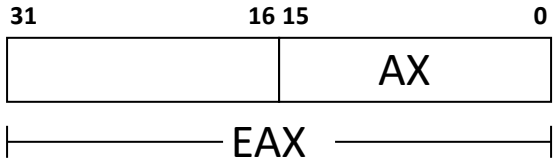
↱ extended, i.e., 32-bit

EAX	accumulator
EBX	base (of array)
ECX	count (for loops)
EDX	data (2nd operand)
ESI	source index (string copy)
EDI	destination index
EBP	base pointer (base of stack frame)
ESP	stack pointer

EIP	instruction pointer
EFLAGS	flags/condition codes

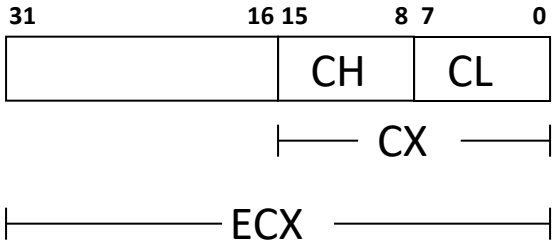
registers

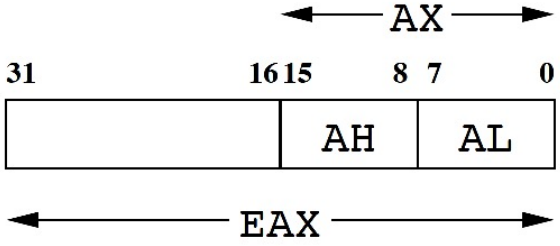
for 16-bit registers, drop "E"



32–bit	16–bit	8–bit high	8–bit low
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		
EDI	DI		
EBP	BP		
ESP	SP		

for 8-bit registers, change "X" to "H" or "L"
(only for A, B, C, and D)





use % as a prefix for registers in assembly

other registers: floating-point, MMX, etc. (not discussed in this class)

data types — many supported!

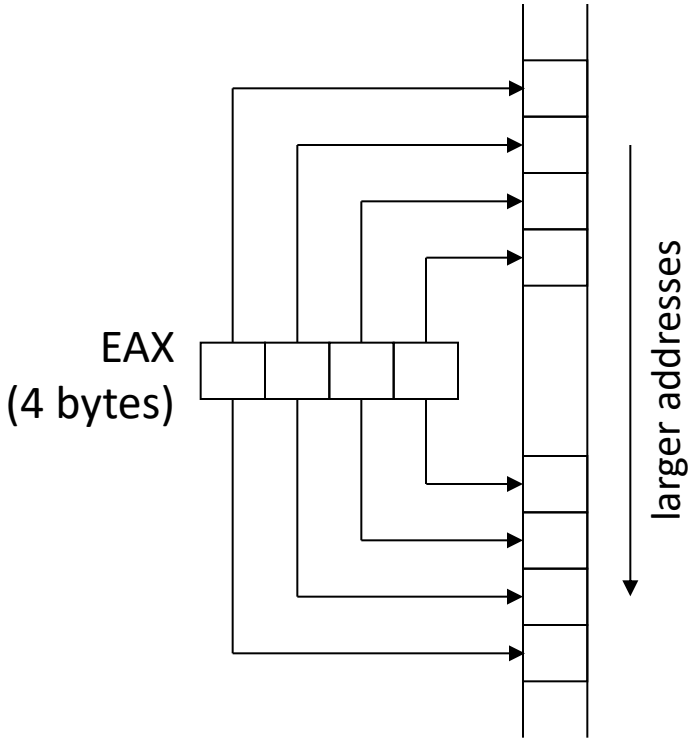
- 8-, 16-, 32-bit unsigned and 2's complement
- IEEE single- and double-precision floating point
- Intel "extended" f.p. (80-bit); pre-IEEE standard
- ASCII strings
- binary-coded decimal
- Etc.

Memory

- Microprocessor addresses a maximum of 2^n 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

Operations

LC-3

ADD LEA

AND NOT

BR RET

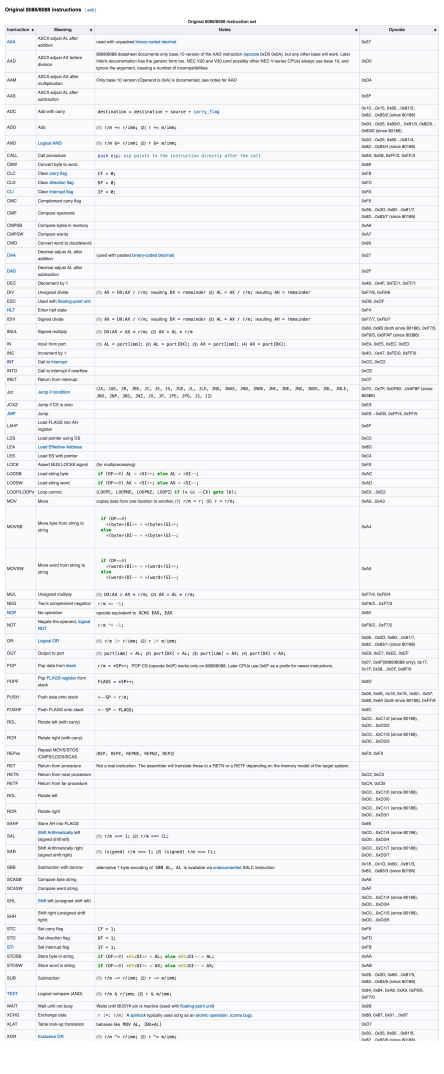
JMP RTI

JSR[R] ST

LD STI

LDI STR

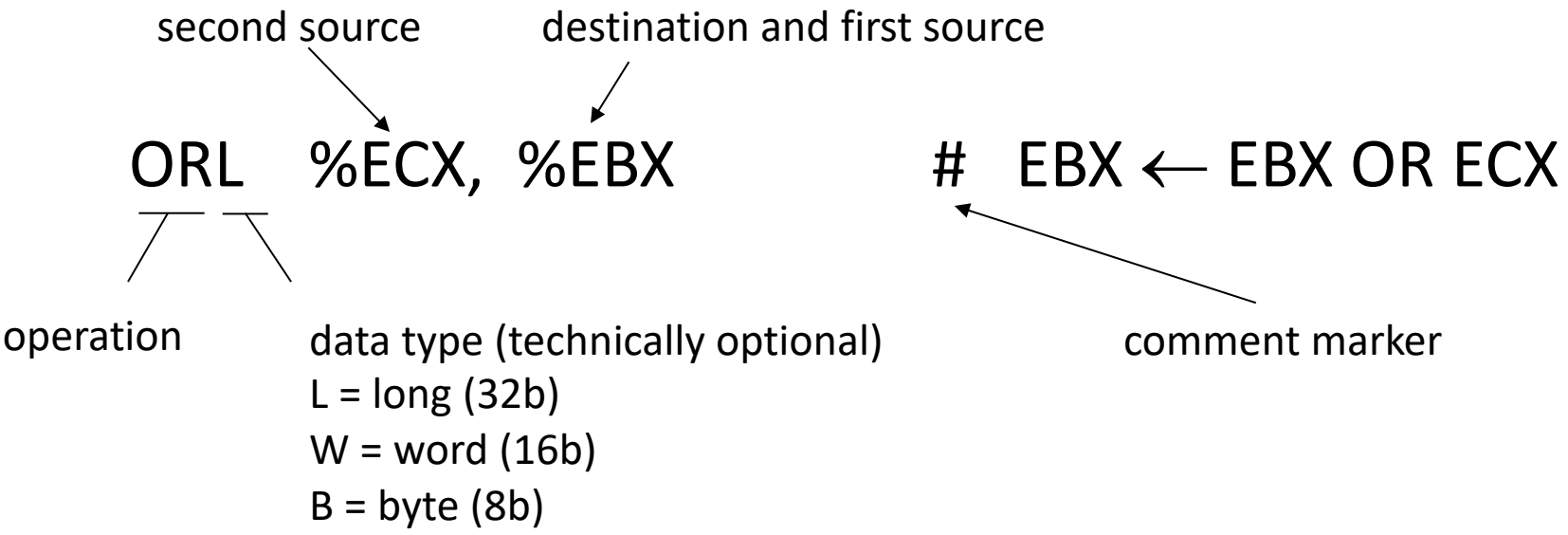
LDR TRAP



Operations—lots of them!

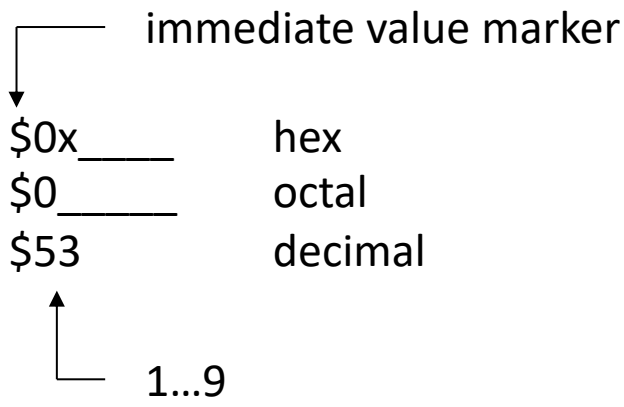
arithmetic	logical	shift
ADD	AND	SHL
SUB	OR	SAR
NEG	NOT	SHR
INC	XOR	ROL
DEC		ROR

Example



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

immediate values



how big can they get?

- usually up to 32 bits
- larger constants ⇒ longer instructions
- length of operand must be encoded, too, of course!

what does the following instruction do?

```
ANDL    0, %EAX
```

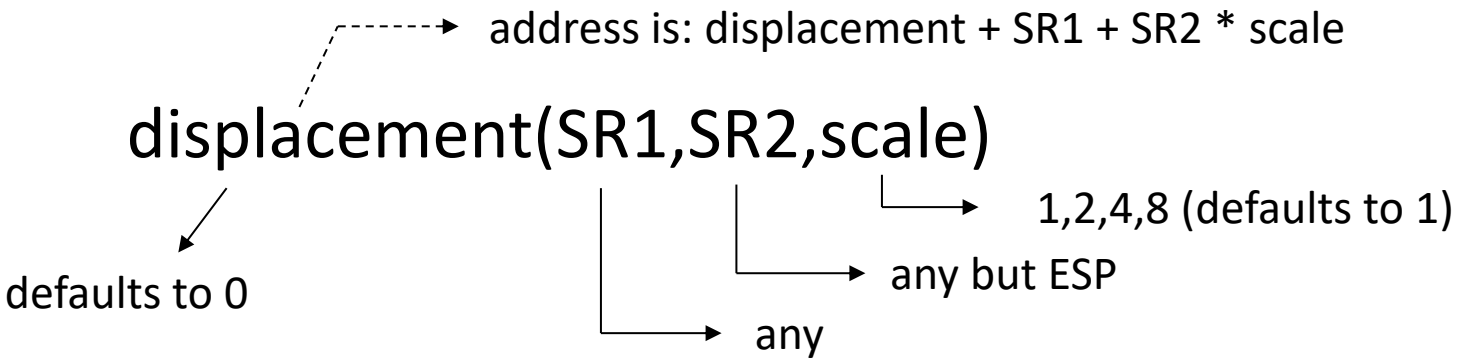
what does the following instruction do?

ANDL 0, %EAX

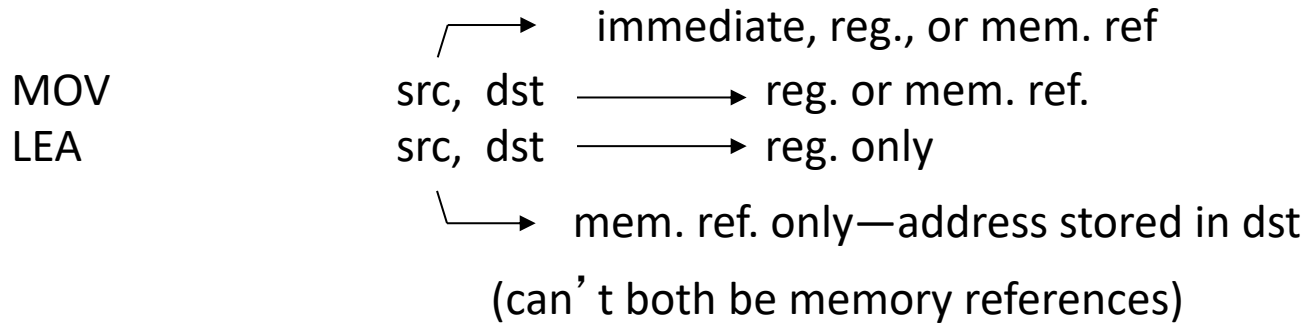
answer is NOT
EAX ← 0

instead:
EAX ← EAX AND M[0]
(usually crashes)

Data Movement: memory addressing



Data movement: instructions



examples

```
MOVW  %DX, 0x10(%EBP)          # M[EBP + 0x10] ← DX

MOVB  (%EBX,%ESI,4), %CL       # CL ← M[EBX + ESI * 4]
```

solve

EAX ← M[0x10000 + ECX]
M[LABEL] ← DI
ESI ← LABEL + 4 (two ways!)

ESI ← LABEL + EAX + 4

solve

EAX ← M[0x10000 + ECX]	MOVL 0x10000(%ECX), %EAX
M[LABEL] ← DI	MOVW %DI, LABEL
ESI ← LABEL + 4 (two ways!)	MOVL $LABEL + 4, %ESI
	LEAL LABEL + 4, %ESI
ESI ← LABEL + EAX + 4	LEAL LABEL + 4(%EAX), %ESI

FLAGS

what instructions set flags (i.e., 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
- Not exclusive as in LC-3

Condition Codes (in EFLAGS)

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

OF — overflow flag: 2's complement overflow

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

(and other instruction-dependent meanings)

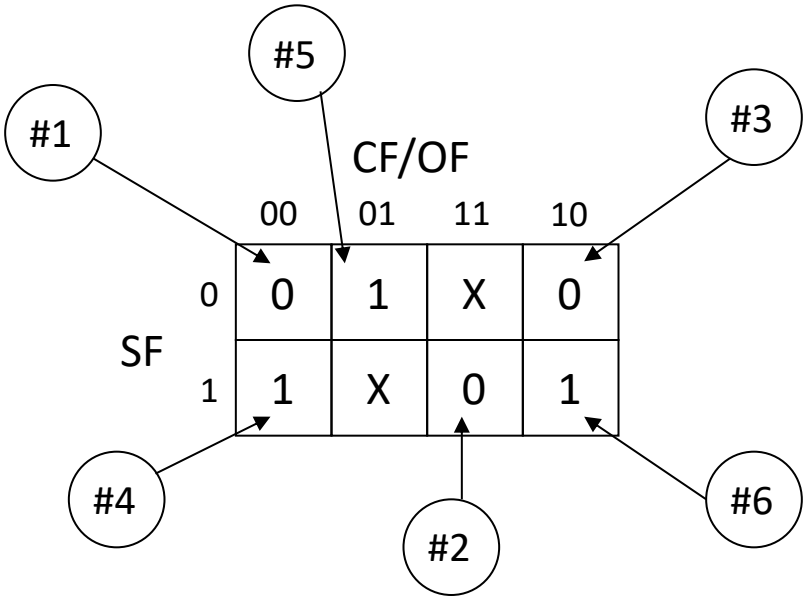
Control Flow Instructions

- branches based on combinations of condition codes
- think about
  - CMPL %ESI, %EDX (sets flags based on (EDX – ESI))
  - what combination of flags needed for unsigned/signed relationship comparisons (EDX < ESI, for example)?

test yourself

	#1	#2	#3	#4	#5	#6
A	010	010	010	110	110	110
B	-000	-110	-111	-000	-011	-111
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 A<B	No	Yes	Yes	No	No	Yes
signed A<B	No	No	No	Yes	Yes	Yes

note that CF suffices for unsigned <
what about signed < ? use a K-map…



answer? OF XOR SF (OF ≠ SF)

Conditional Branches

jo	overflow	OF is set	jb	below	CF is set
jp	parity	PF is set	jbe	below or equal	CF or ZF is set
js	sign	SF is set	jl	less	SF ≠ OF
je	equal	ZF is set	jle	less or equal	(SF ≠ OF) or ZF is set

Jump

- branch mnemonics
  - unsigned comparisons: “above” and “below”
  - signed comparisons: “less” and “greater”
  - both: equal/zero

Unsigned	**`jne`**	**`jb`**	**`jbe`**	**`je`**	**`jae`**	**`ja`**
relationship	≠	<	≤	=	≥	>
Signed	**`jne`**	**`jl`**	**`jle`**	**`je`**	**`jge`**	**`jg`**

- in general, can add “n” after “j” to negate sense
  forms shown are those used when disassembling
  - don’t expect binary to retain your version
  - e.g., “**`jnae`**” becomes “**`jb`**”

control instructions

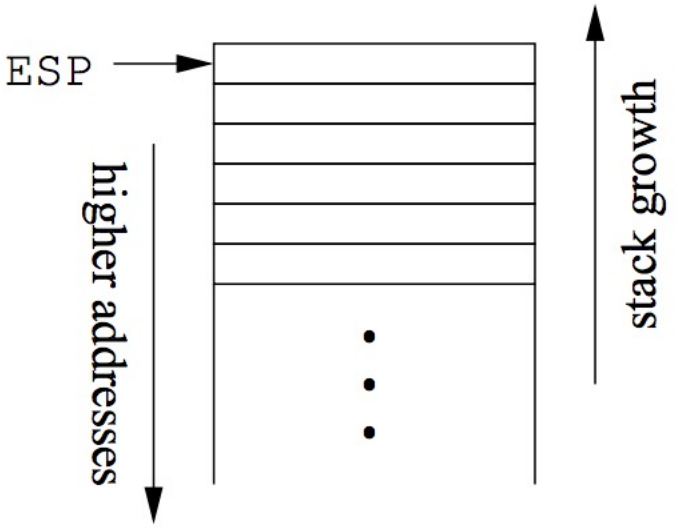
subroutine call and return

```
CALL  printf     # (push EIP), EIP ← printf
CALL  *%EAX      # (push EIP), EIP ← EAX
CALL  *(%EAX)    # (push EIP), EIP ← M[EAX]

RET              # EIP ← M[ESP], ESP ← ESP + 4
```

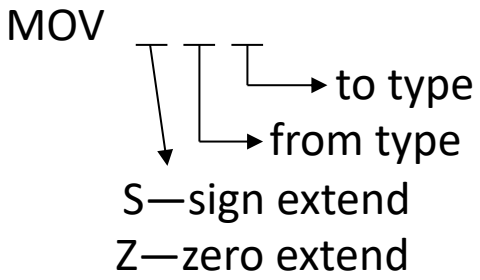
Stack Operations

- push and pop supported directly by x86 ISA
  - PUSHL %EAX # M[ESP – 4] ← EAX, ESP ← ESP – 4
  - POPL %EBP # EBP ← M[ESP], ESP ← ESP + 4
  - PUSHFL # M[ESP – 4] ← EFLAGS, ESP ← ESP– 4
  - POPFL # EFLAGS ← M[ESP], ESP ← 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])
```