

ECE 391 Computer Systems Engineering

Yih-Chun Hu
University of Illinois

Administrivia

- MPO
 - **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

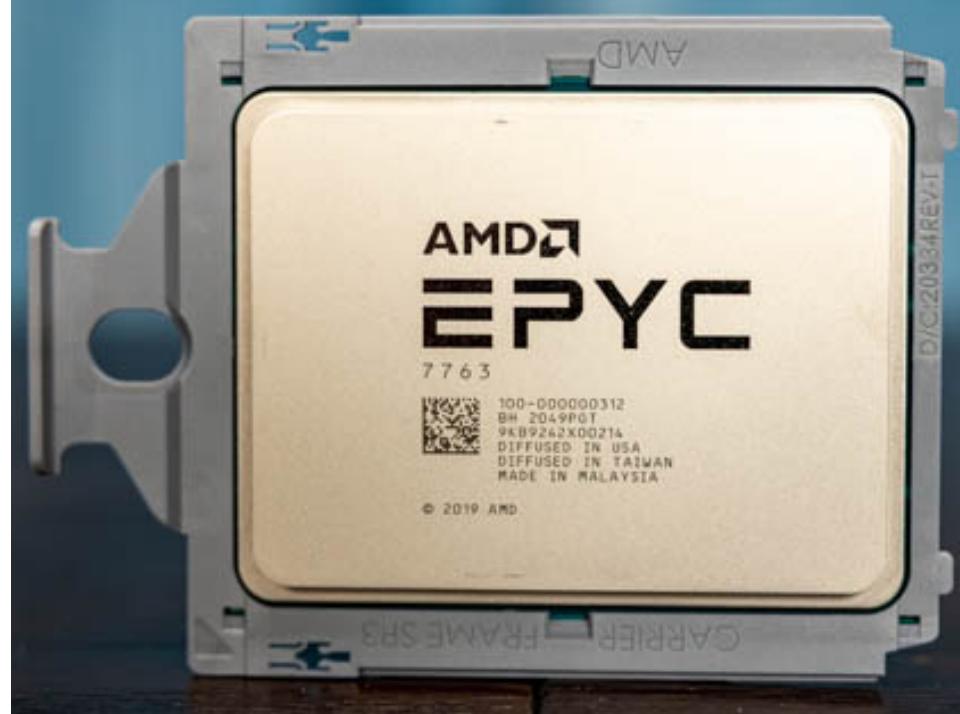
What is an ISA?

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

EIP instruction pointer

EBX base (of array)

EFLAGS flags/condition codes

ECX count (for loops)

EDX data (2nd operand)

ESI source index (string copy)

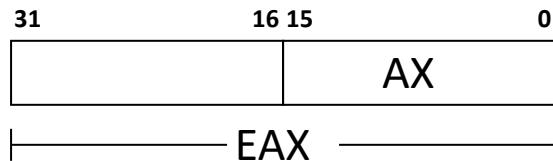
EDI destination index

EBP base pointer (base of stack frame)

ESP stack pointer

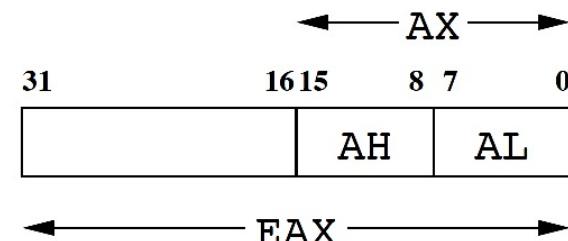
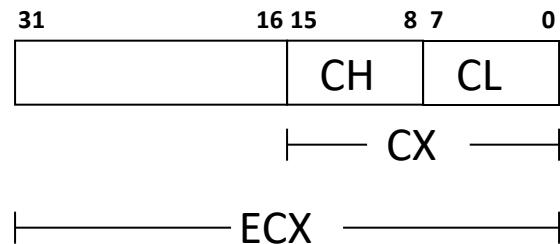
registers

for 16-bit registers, drop “E”



| 32-bit | 16-bit | 8-bit | high | 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)

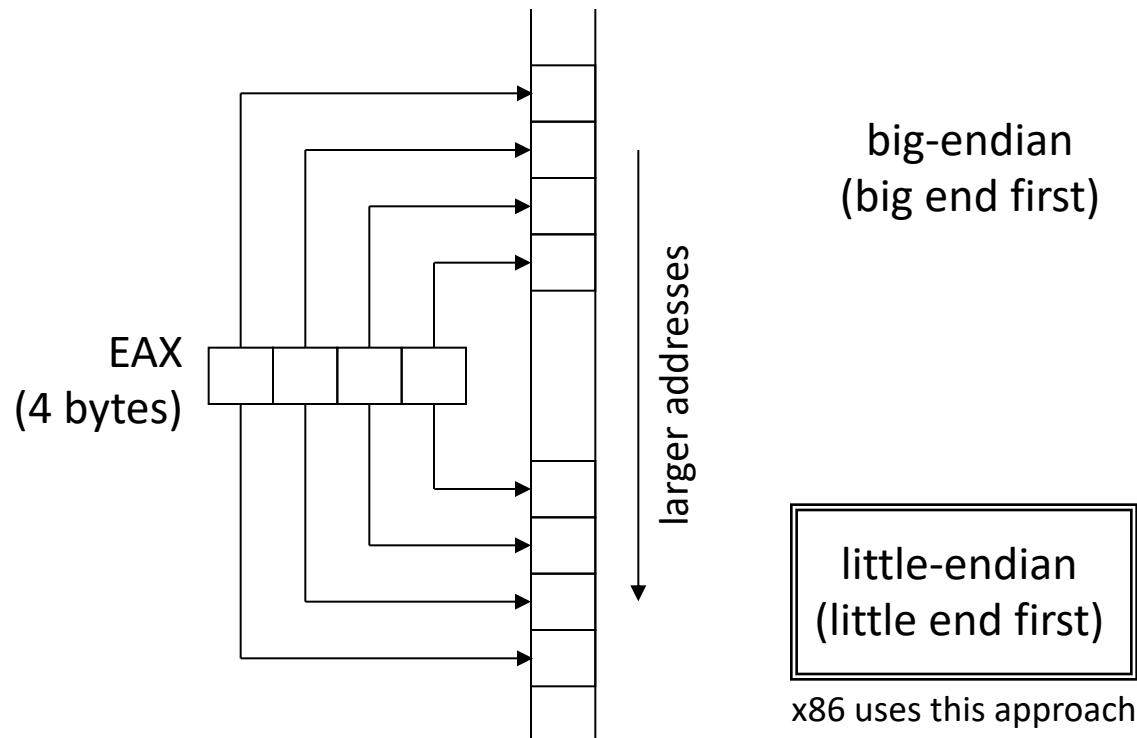
data types — many supported!

- 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?



LC-3

| | |
|--------|------|
| ADD | LEA |
| AND | NOT |
| BR | RET |
| JMP | RTI |
| JSR[R] | ST |
| LD | STI |
| LDI | STR |
| LDR | TRAP |

Operations

| Instruction | Meaning | Notes |
|--|--|---|
| FFREE | Free register | |
| FADD | Integer add | |
| FICOM | Integer compare | |
| FICOMP | Integer compare and pop | |
| FIDIV | Integer divide | |
| FIDIVR | Integer divide reversed | |
| FLD | Load integer | |
| FMUL | Integer multiply | |
| FPINVD | Initialize floating point stack pointer | |
| FINIT | Initialize floating point processor | |
| FIST | Store integer | |
| FISTP | Store integer and pop | |
| FISUB | Integer subtract | |
| FISUBR | Integer subtract reversed | |
| FLD | Floating point load | |
| FLDI | Load 1.0 onto stack | |
| FCUDW | Load control word | |
| FCUDW2 | Load control word | |
| FLDENVW | Load environment state, 16-bit | |
| FLDZB | Load log(0) onto stack | |
| FLDZT | Load log(1) onto stack | |
| FLDUS | Load log(2) onto stack | |
| FLDLN2 | Load ln(2) onto stack | |
| FLDPI | Load π onto stack | |
| FLDZ | Load 0.0 onto stack | |
| FMUL | Multiply | |
| FNEXL | Multiply and pop | |
| FNINIT | Disable interrupts, no wait | |
| FNINIT0 | Disable interrupts, no wait | 8087 only; otherwise FNOP |
| FNINH | Enable Interrupts, no wait | 8087 only; otherwise FNOP |
| FNINIT1 | Initialize floating point processor, no wait | |
| FNOP | No operation | |
| FNSAVE | Save FPU state, no wait, 8-Bit | |
| FNSAVEW | Save FPU state, no wait, 16-bit | |
| FNSTCW | Store control word, no wait | |
| FNSTENV | Store FPU environment, no wait | |
| FNSTENVW | Store FPU environment, no wait, 16-bit | |
| FNSTSW | Store status word, no wait | |
| FRINTM | Partial argument | |
| FRREM | Partial remainder | |
| FTAN | Partial tangent | |
| FRNDINT | Round to integer | |
| FRSTOR | Restore saved state | |
| FRSTORW | Restore saved state | Perhaps not actually available in 8087 |
| FSAVE | Save FPU state | |
| FSAVEW | Save FPU state, 16-bit | |
| FSCALL | Scale factor of 2 | |
| FSINCW | Scale factor of 2 | |
| FST | Floating point store | |
| FCSTCW | Store control word | |
| FCSTENV | Store FPU environment | |
| FCSTENVW | Store FPU environment, 16-bit | |
| FSTP | Store and pop | |
| FSTSW | Store status word | |
| FSUB | Subtract | |
| FSUBP | Subtract and pop | |
| FSUBRP | Reverse subtract and pop | |
| FTST | Test for zero | |
| FWAIT | Wait while FPU is executing | |
| FXAM | Examine condition flags | |
| FXCH | Exchange registers | |
| FXTRACT | Extract exponent and significand | |
| FYL2X | $y \cdot \log_2 x$ | If $y = \log_2$, then the base 2 logarithm is computed |
| FYL2XP1 | $y \cdot \log_2 (x+1)$ | more precise than $\log_2 x$ if x is close to zero |
| Added in specific processors [edit] | | |
| Added with 80387 [edit] | | |
| Instruction | Meaning | Notes |
| FSETM | Set protected mode | 80287 only; otherwise FNOP |
| Added with 80387 [edit] | | |
| Instruction | Meaning | Notes |
| FCOS | Cosine | |
| FLDENV | Load environment state, 32-bit | |
| FSAVED | Save FPU state, 32-bit | |
| FRPREM1 | Partial remainder | Computes IEEE754 remainder |
| FRSTOR | Restore saved state, 32-bit | |
| FSIN | Sine | |
| FSINP | Sine and pop | |
| FTIMOD | Store floating point environment, 32-bit | |
| FUCOM | Unordered compare | |
| FUCOMP | Unordered compare and pop | |
| FUCOMPP | Unordered compare and pop twice | |
| Added with Pentium Pro [edit] | | |
| • FOMOV variants: FOMOVB, FOMOVR, FOMOVNB, FOMOVN, FOMOVN1, FOMOVO | | |
| • FOCMP variants: FOCMI, FOCMP, FOCMI1, FOCMP1 | | |
| Added with SSE [edit] | | |

Operations—lots of them!

arithmetic

ADD

SUB

NEG

INC

DEC

logical

AND

OR

NOT

XOR

shift

SHL

SAR

SHR

ROL

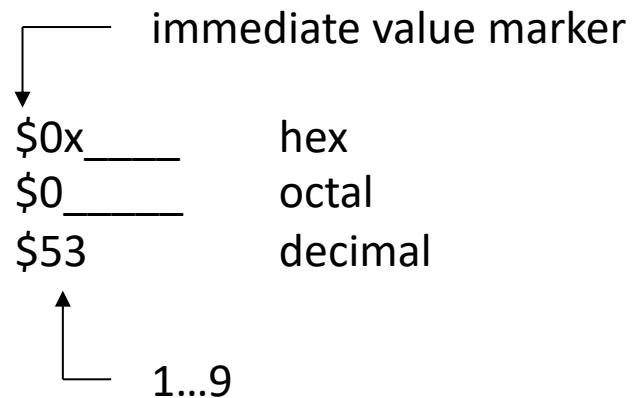
ROR

Example

The diagram illustrates the assembly instruction `ORL %ECX, %EBX` and its assembly equivalent `# EBX ← EBX OR ECX`. The instruction is shown in two columns. The left column contains the instruction `ORL %ECX, %EBX`, where `ORL` is the operation, `%ECX` is the data type (technically optional), and `%EBX` is the second source. The right column contains the instruction `# EBX ← EBX OR ECX`, where `#` is the comment marker and `EBX ← EBX OR ECX` is the destination and first source. Arrows point from the labels to their corresponding parts in the instruction.

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 \Rightarrow longer instructions
- length of operand must be encoded, too, of course!

what does the following
instruction do?

ANDL 0, %EAX

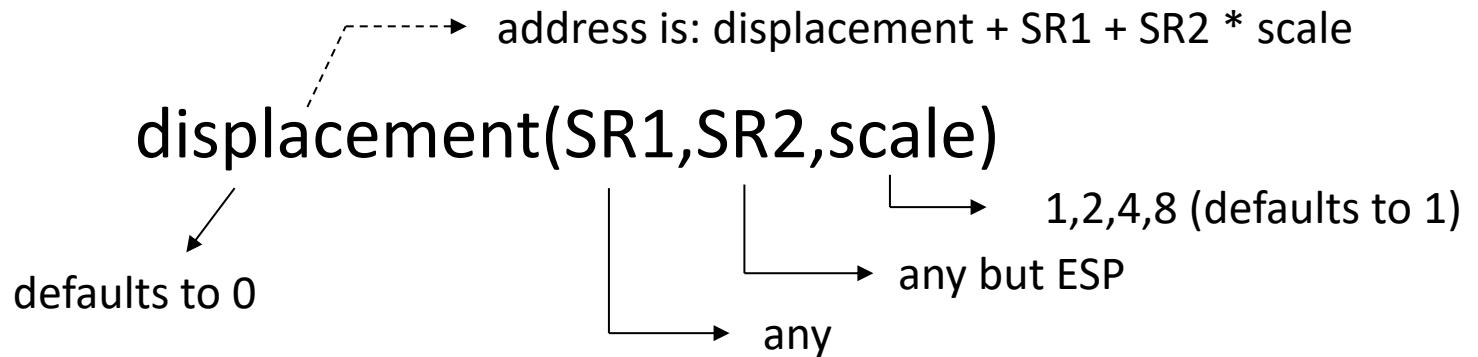
what does the following
instruction do?

ANDL 0, %EAX

answer is NOT
 $EAX \leftarrow 0$

instead:
 $EAX \leftarrow EAX \text{ AND } M[0]$
(usually crashes)

Data Movement: memory addressing



Data movement: instructions

| | | |
|-----|----------|---|
| MOV | src, dst | immediate, reg., or mem. ref |
| LEA | src, dst | reg. or mem. ref. |
| | | reg. only |
| | | mem. ref. only—address stored in dst (can't both be memory references) |

examples

MOVW %DX, 0x10(%EBP) # $M[EBP + 0x10] \leftarrow DX$

MOVB (%EBX,%ESI,4), %CL # $CL \leftarrow M[EBX + ESI * 4]$

solve

$EAX \leftarrow M[0x10000 + ECX]$

$M[LABEL] \leftarrow DI$

$ESI \leftarrow LABEL + 4$ (two ways!)

$ESI \leftarrow LABEL + EAX + 4$

solve

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

ESI \leftarrow LABEL + EAX + 4

MOVL 0x10000(%ECX), %EAX
MOVW %DI, LABEL
MOVL \$LABEL + 4, %ESI
LEAL LABEL + 4, %ESI
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 \leftarrow (EBX – EAX) |
| TESTL %EAX, %EBX | # flags \leftarrow (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)

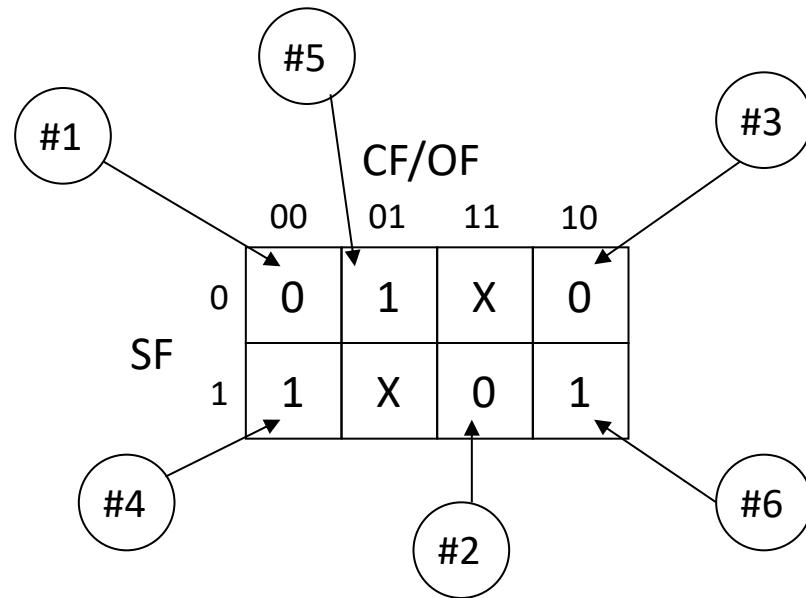
Control Flow Instructions

- 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 \neq 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 \neq OF |
| je | equal | ZF is set | jle | less or equal | (SF \neq 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 \leftarrow printf

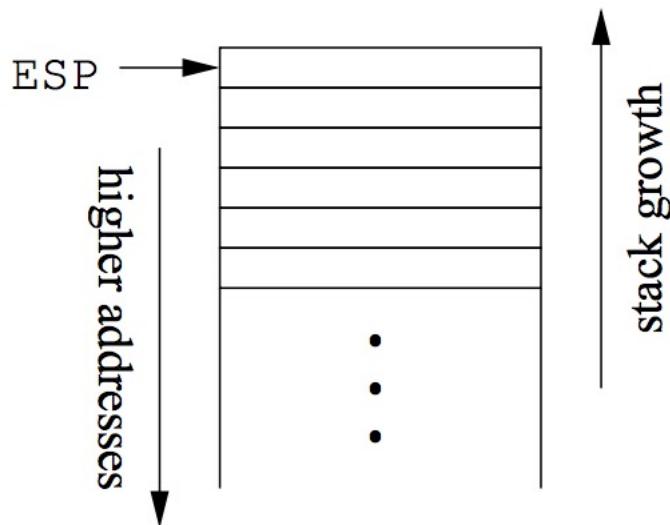
CALL *%EAX # (push EIP), EIP \leftarrow EAX

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

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

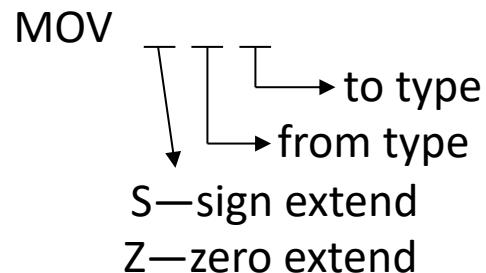
Stack Operations

- 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` $\# M[ESP - 4] \leftarrow EFLAGS, ESP \leftarrow ESP - 4$
 - `POPFL` $\# EFLAGS \leftarrow M[ESP], ESP \leftarrow ESP + 4$



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])