Addressing Modes

- Immediate

  ADD R0, R0, #1 ADDL $1, %EAX

- Register

  ADD R0, R0, R1 ADDL %EBX, %EAX

- Memory

  LD R0, LABEL MOVL LABEL, %EAX

  LEA R0, LABEL LEAL LABEL, %EAX

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)

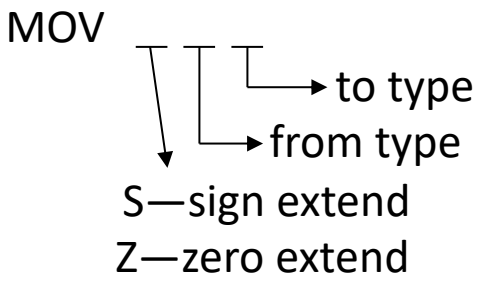
(and other instruction-dependent meanings)

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

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])
```

MOVSX/MOVSXD—Move with Sign-Extension

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F BE /r	MOVSX *r16, r/m8*	RM	Valid	Valid	Move byte to word with sign-extension.
0F BE /r	MOVSX *r32, r/m8*	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX.W + 0F BE /r	MOVSX *r64, r/m8*	RM	Valid	N.E.	Move byte to quadword with sign-extension.
0F BF /r	MOVSX *r32, r/m16*	RM	Valid	Valid	Move word to doubleword, with sign-extension.
REX.W + 0F BF /r	MOVSX *r64, r/m16*	RM	Valid	N.E.	Move word to quadword with sign-extension.
63 /r*	MOVSXD *r16, r/m16*	RM	Valid	N.E.	Move word to word with sign-extension.
63 /r*	MOVSXD *r32, r/m32*	RM	Valid	N.E.	Move doubleword to doubleword with sign-extension.
REX.W + 63 /r	MOVSXD *r64, r/m32*	RM	Valid	N.E.	Move doubleword to quadword with sign-extension.

NOTES:

* The use of MOVSXD without REX.W in 64-bit mode is discouraged. Regular MOV should be used instead of using MOVSXD without REX.W.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8*.
REX + 88 /r	MOV *r/m8****, *r8****	MR	Valid	N.E.	Move *r8* to *r/m8*.
89 /r	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16*.
89 /r	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32*.
REX.W + 89 /r	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64*.
8A /r	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8*.
REX + 8A /r	MOV *r8***,r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8*.
8B /r	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16*.
8B /r	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32*.
REX.W + 8B /r	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64*.
8C /r	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r/m16*.
8C /r	MOV *r16/r32/m16, Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r16/r32/m16*.
REX.W + 8C /r	MOV *r64/m16, Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r64/m16*.
8E /r	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment register.
REX.W + 8E /r	MOV *Sreg,r/m64***	RM	Valid	Valid	Move *lower 16 bits of r/m64* to segment register.
A0	MOV AL,*moffs8**	FD	Valid	Valid	Move byte at (*seg:offset*) to AL.
REX.W + A0	MOV AL,*moffs8**	FD	Valid	N.E.	Move byte at (*offset*) to AL.
A1	MOV AX,*moffs16**	FD	Valid	Valid	Move word at (*seg:offset*) to AX.
A1	MOV EAX,*moffs32**	FD	Valid	Valid	Move doubleword at (*seg:offset*) to EAX.
REX.W + A1	MOV RAX,*moffs64**	FD	Valid	N.E.	Move quadword at (*offset*) to RAX.
A2	MOV *moffs8*,AL	TD	Valid	Valid	Move AL to (*seg:offset*).
REX.W + A2	MOV *moffs8****,AL	TD	Valid	N.E.	Move AL to (*offset*).
A3	MOV *moffs16**,AX	TD	Valid	Valid	Move AX to (*seg:offset*).
A3	MOV *moffs32**,EAX	TD	Valid	Valid	Move EAX to (*seg:offset*).
REX.W + A3	MOV *moffs64**,RAX	TD	Valid	N.E.	Move RAX to (*offset*).
B0+ *rb ib*	MOV *r8, imm8*	OI	Valid	Valid	Move *imm8* to *r8*.
REX + B0+ *rb ib*	MOV *r8****, *imm8*	OI	Valid	N.E.	Move *imm8* to *r8*.
B8+ *rw iw*	MOV *r16, imm16*	OI	Valid	Valid	Move *imm16* to *r16*.

Assembler Conventions

label: requires a colon (but not in its use), 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)

Assembler Conventions

```
.string “Hello, world!”, “me”        # NUL-terminated
.byte 100, 0x30, 052                 # integer constants of various sizes
.word …
.long …
.quad …

.single 1.0, 2.0                     # floating-point constants
.double 2.0, 3.1415
```

if assembly file name ends in .S (case-sensitive!), file is first passed through C’s preprocessor (#define, #include, etc.)

AT&T v Intel Syntax

- AT&T immediate operands are preceded by $; Intel immediate operands are undelimited
  - Intel `push 4' is AT&T `pushl $4’
- AT&T register operands are preceded by %; Intel register operands are undelimited
- AT&T and Intel syntax use the opposite order for source and destination operands.
  - Intel `add eax, 4' is AT&T `addl $4, %eax’
- In AT&T syntax the size of memory operands is determined from the last character of the opcode name. Opcode suffixes of `b', `w’, and `l' specify byte (8-bit), word (16-bit), and long (32-bit) memory references. Intel syntax accomplishes this by prefix memory operands with `byte ptr’, `word ptr', and `dword ptr'.
  - Intel `mov al, byte ptr FOO’ is AT&T `movb FOO, %al’

GNU AS User Guide

Multiplication and Division

```
MULL    %EBX            # unsigned EDX:EAX ← EAX * EBX
IMULL   %EBX            # signed (as above)

# multiple-operand forms are ONLY for signed multiplication/division
IMULL   %ECX,%EBX       # signed EBX ← EBX * ECX (high bits discarded)
IMULL   $20,%EDX,%ECX   # signed ECX ← 20 * EDX (high bits discarded)

DIV     %EBX            # unsigned EAX ← EDX:EAX / EBX
                        # EDX ← remainder
IDIV …                  # (signed version)
```

Data Type Alignment

- memory addresses
  - when loading data from or storing data to memory
  - use address that is multiple of size of data
- examples
  - for bytes, use any address
  - for words (16-bit), use even addresses only (multiple of 2 bytes)
  - for longs (32-bit), use multiple-of-4 addresses only
  - etc.
- rationale: simplifies implementation of processor-memory interface
  - required by many modern ISAs
  - optional on x86 (but very slow if you don't align)
  - x86 has alignment check flag (AC), but usually turned off
- use ".ALIGN 4" (number is an argument) to align x86 assembly
  (for x86 assemblers, you can even do so in the middle of code… [why?])

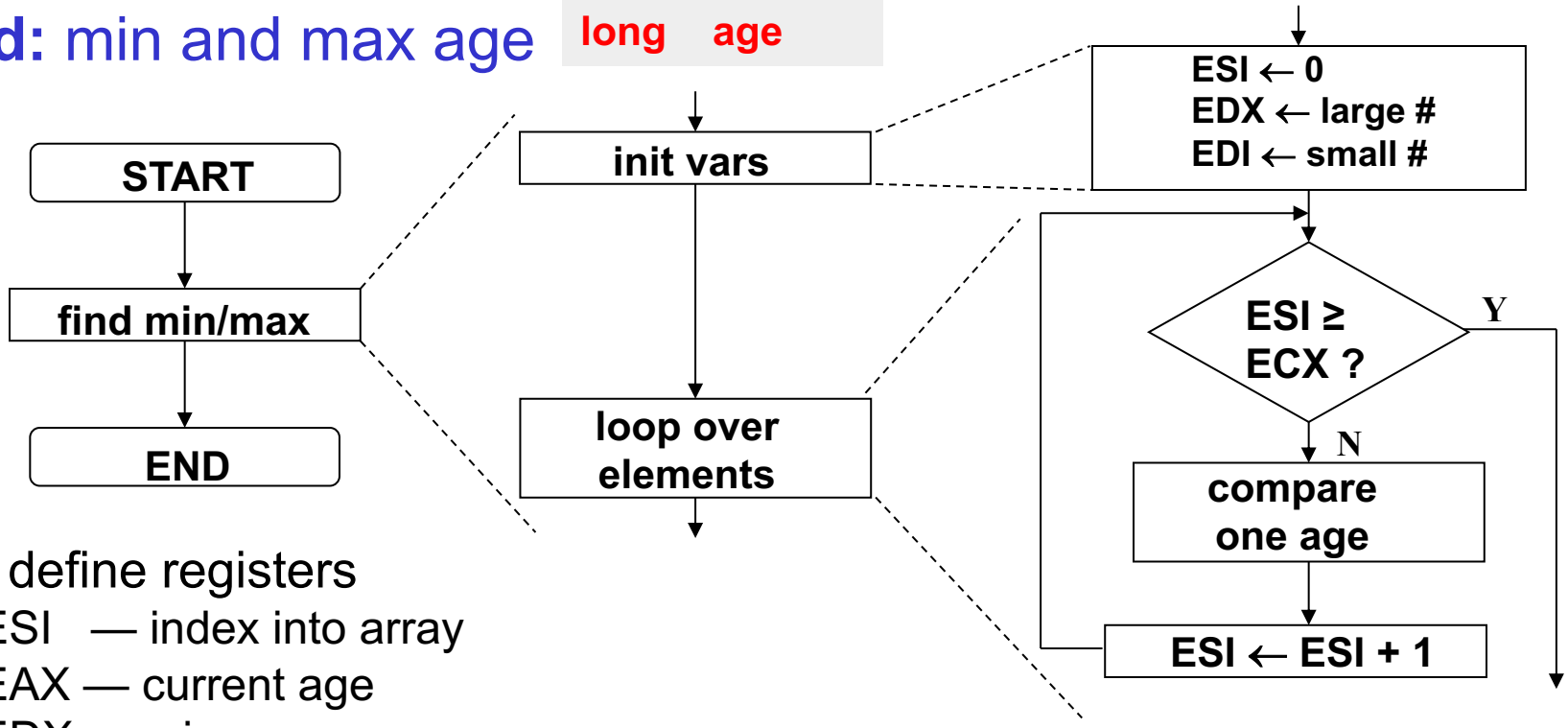
Lecture Topics

- Calling convention and stack frames
- Application to example
- Misc. x86 instructions

Code Example

- **Given:** EBX pointing to an array of structures with ECX elements in the array
- **Find:** min and max age

the structure
char* name
long age



- first, define registers
  - ESI — index into array
  - EAX — current age
  - EDX — min age seen
  - EDI — max age seen
- next, use systematic decomposition…

Code Example - Loop Design Process (1)

- What is the task to be repeated?
  - update the min/max ages
  - based on the age of a single person in the array
- What are the invariants?
  - ESI = array index of person being considered
  - EDX = min age of those earlier in array
  - EDI = max age of those earlier in array
  - ECX = size of array
- What are the stopping conditions?
  - reach the end of the array
  - i.e., ESI ≥ ECX

Code Example - Loop Design Process (2)

- What must be done when a stopping condition is met?
  - nothing! (by definition of this particular problem)
- How do we prepare for the first iteration?
  - set array index to point to first person
  - set min age to something large (or to first person's, but may not exist)
  - set max age to something small
- How do we prepare for subsequent iterations?
  - update min/max age based on current person
  - increment ESI

Code Example - Loop Body