ECE391
Computer System Engineering

Lecture 3

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Spring 2017

Lecture Topics

- Continue x86 instructions
- Conditional codes
- Control flow instructions
- Assembler conventions
- Code example
- Calling convention and stack frames
- Application to example
- Misc. x86 instructions

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)

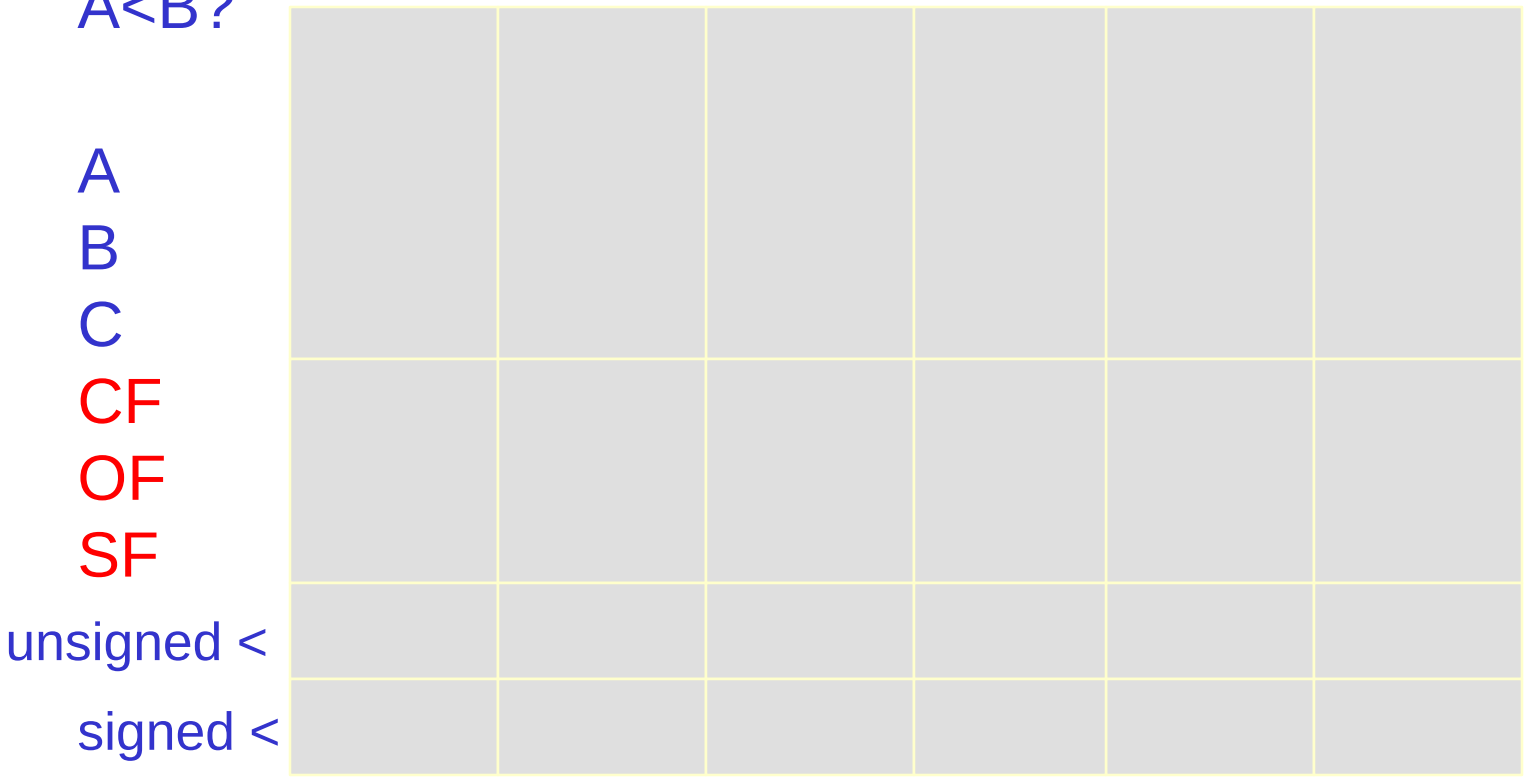
  TESTL %EAX, %EBX # flags ⟵ (EBX *AND* EAX)

- 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?



Control Flow Instructions (2)

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

- Answer: OF *XOR* SF

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
  - 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 ← 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

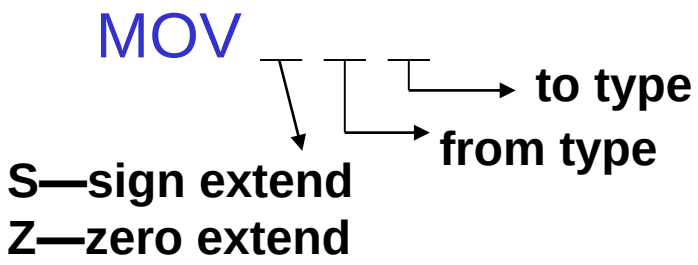
POPL %EBP # EBP ← M[ESP], ESP ← ESP + 4

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])
```

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)

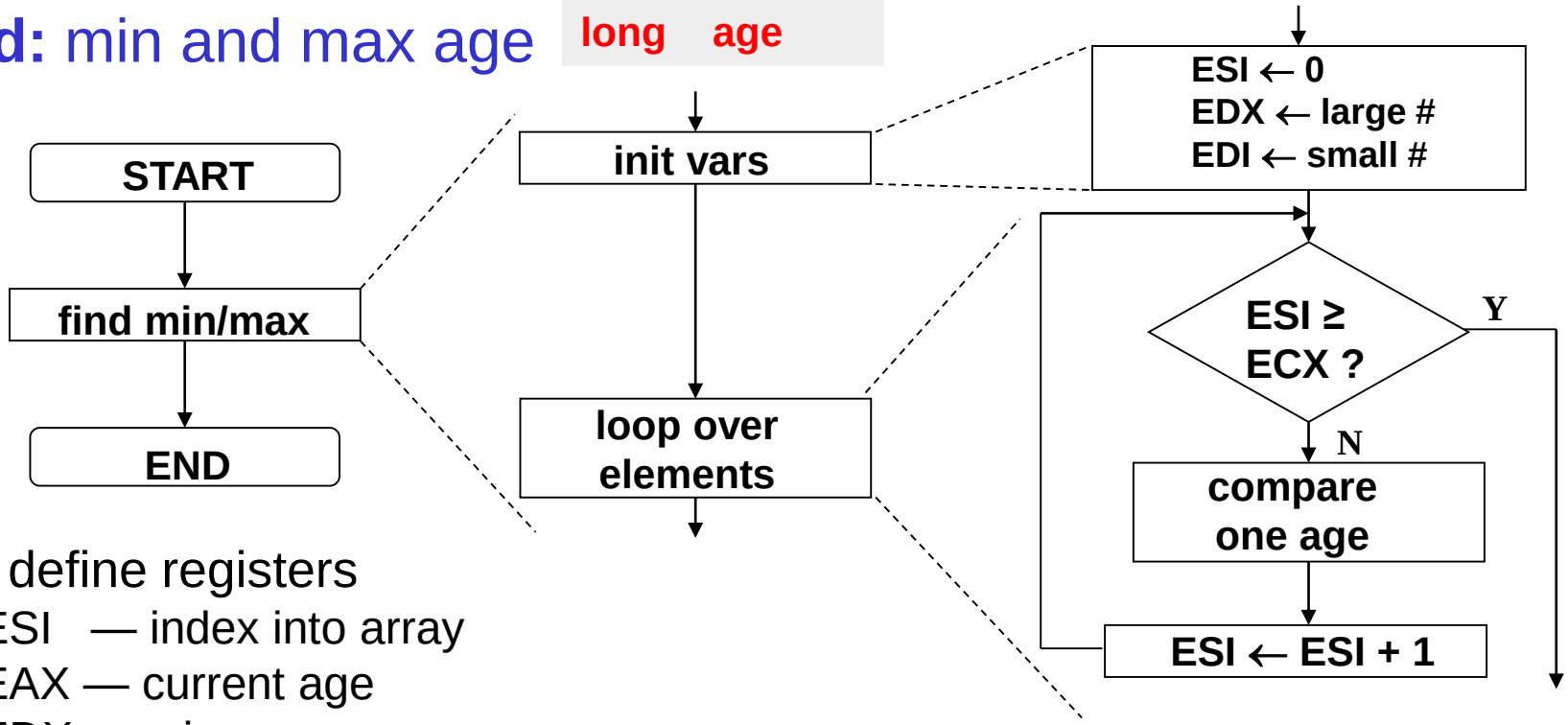
```
.string "Hello, world!", "me"     # NUL-terminated
.byte   100, 0x30, 052            # integer constants of various sizes
.word …
.long …
.quad …
.single …                         # floating-point constants
.double …
```

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

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

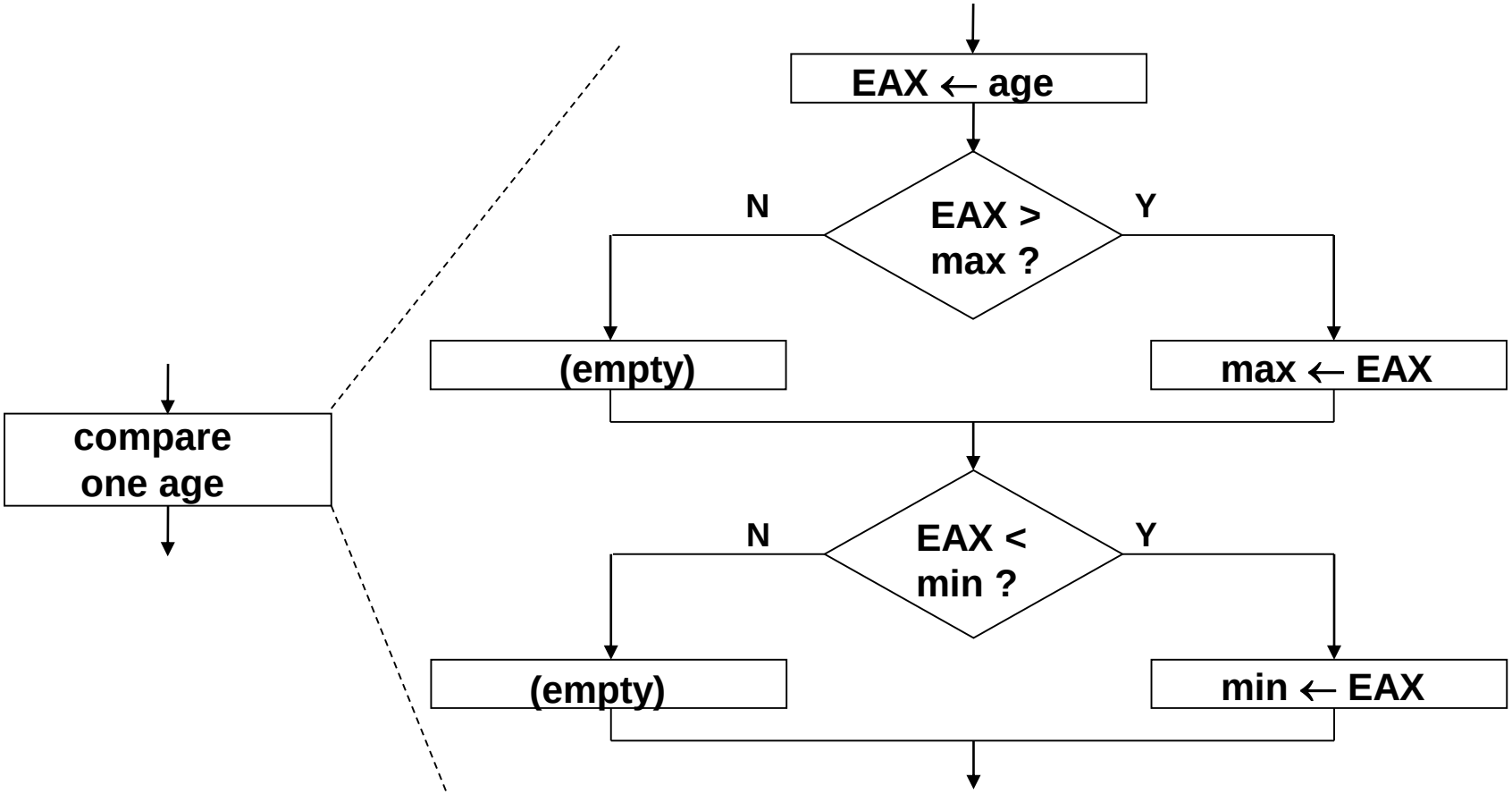
Code Example - Loop Design Process (1)

- 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



Code Example – Assembly (1)

```
        XORL %ESI,%ESI          # init vars
        MOVL TWO_MM,%EDX
        MOVL TWO_MM+4,%EDI
LOOP:
        CMPL %ECX,%ESI          # loop test
        JGE   DONE
        # read the age using one memory reference.
        MOVL 4(%EBX,%ESI,8),%EAX

        CMPL  %EDI,%EAX         # check the max. age
        JLE   NOT_MAX
        MOVL  %EAX,%EDI
```

Code Example – Assembly (2)

```
NOT_MAX:
            CMPL   %EDX,%EAX          # check the min age
            JGE    NOT_MIN
            MOVL   %EAX,%EDX
NOT_MIN:

            INCL %ESI                 # loop update
            JMP  LOOP

DONE:       # more code ...

TWO_MM:     .LONG 0x7FFFFFFF, 0x80000000
```

The Calling Convention (1)

- What is a calling convention?
  - generally: rules for subroutine interface structure
  - specifically
    - how information is passed into subroutine
    - how information is returned to caller
    - who owns registers
  - often specified by vendor so that different compilers' code can work together (it's a CONVENTION)
- Parameters for subroutines
  - pushed onto stack
  - from right to left in C
  - order can be language-dependent

The Calling Convention (2)

- Subroutine return values
  - EAX for up to 32 bits
  - EDX:EAX for up to 64 bits
  - floating-point not discussed
- Register ownership
  - return values can be clobbered by subroutine: EAX and EDX
  - caller-saved: subroutine free to clobber; caller must preserve
    - ECX
    - EFLAGS
  - callee-saved: subroutine must preserve value passed in
    - stack structure: ESP and EBP
    - other registers: EBX, ESI, and EDI

Stack Frames in x86 (1)

- The call sequence
  0. save caller-saved registers (if desired)
     1. **push** arguments onto stack
        2. make the call
     3. **pop** arguments off the stack
  4. restore caller-saved registers

Stack Frames in x86 (2)

- The **callee sequence** (creates the stack frame)
    0. save old base pointer and get new one
        1. save callee-saved registers (always)
            2. make space for local variables
                3. do the function body
            4. tear down stack frame (locals)
        5. restore callee-saved registers
    6. load old base pointer
    7. return

Stack Frames in x86 (3)

- Example of caller code (no caller-saved registers considered)

```
int func (int A, int B, int C);
```

```
func (100, 200, 300);
```

```
PUSHL $300
PUSHL $200
PUSHL $100
CALL func
ADDL $12,%ESP
# result in EAX
```

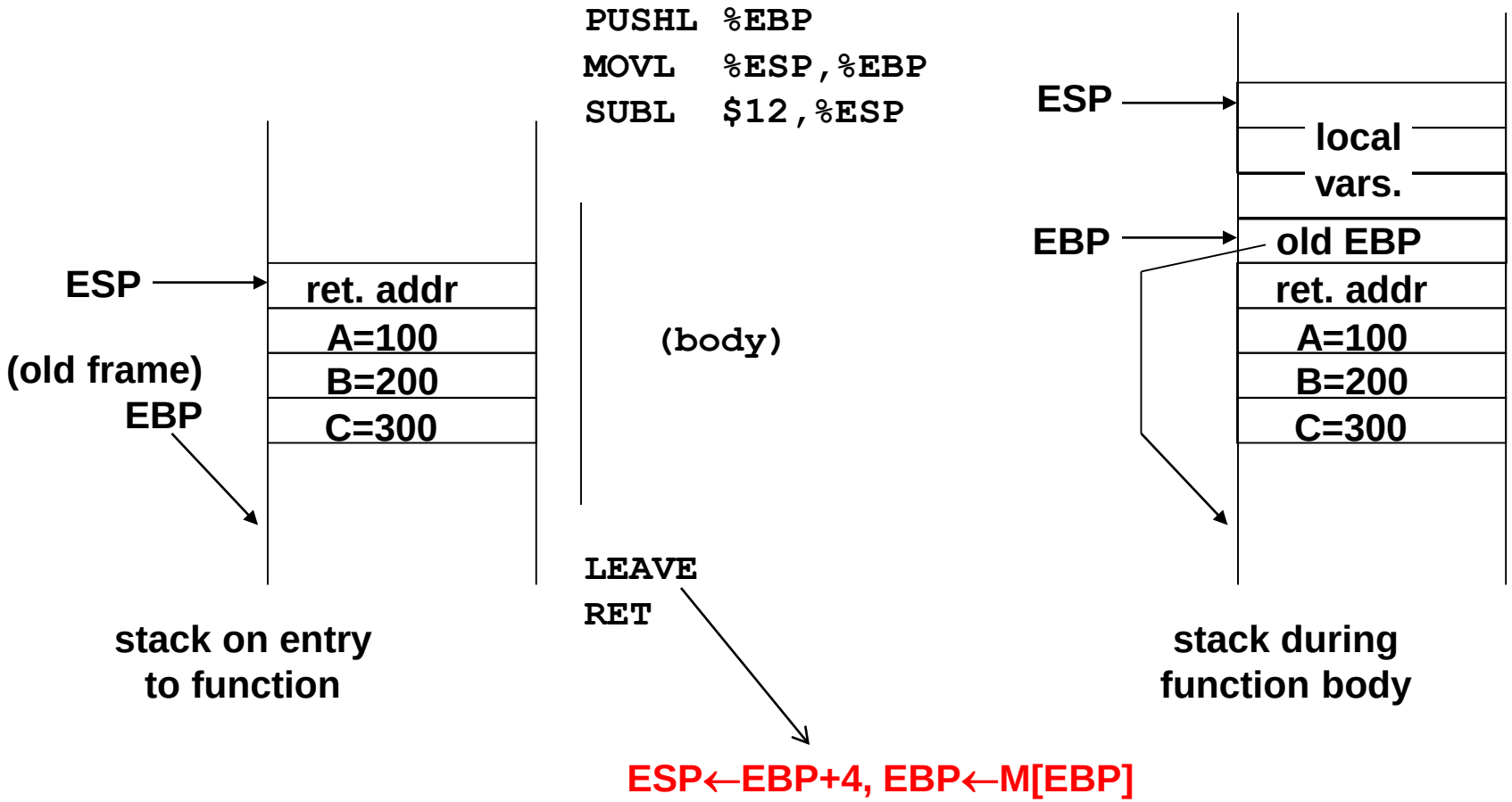
Stack Frames in x86 (4)

- Example of subroutine code and stack frame creation and teardown

```
int func (int A, int B, int C)
{
  /* 12 bytes of local variables */
  …
}

call func (100, 200, 300);
```

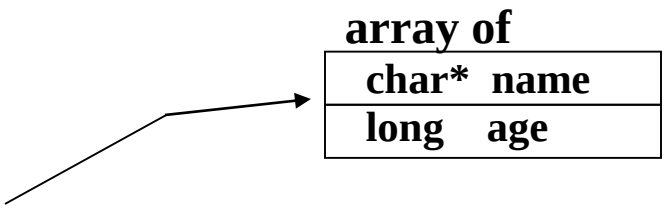
Stack Frames in x86 (4)



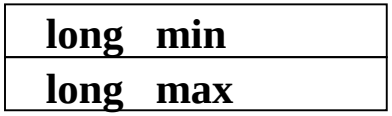
Subroutine Example Code

- Earlier assumptions
  - some values start in registers (array pointer in EBX, length in ECX)
  - could specify output regs (min. age in EDX, max. age in EDI)

As a C function, we could write…



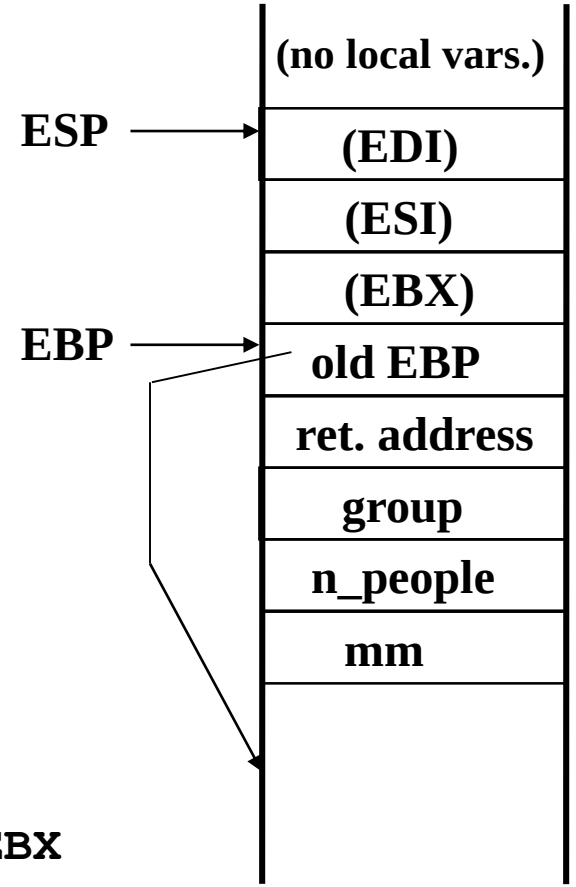

```
void find_min_max (person* group, long n_people,
                   min_max* mm)
```



Subroutine Example Code (cont.)

- step 1: create the stack frame

```
PUSHL %EBP
MOVL  %ESP,%EBP
PUSHL %EBX   # protect callee-saved
             # registers
PUSHL %ESI
PUSHL %EDI
```



step 2: link to our input interface

```
MOVL 8(%EBP),%EBX    # group
MOVL 12(%EBP),%ECX   # n_people
```

step 3: insert our code from before

step 4: link from our output interface

```
MOVL 16(%EBP),%EBX   # load mm into EBX
MOVL %EDX,0(%EBX)    # mm->min
MOVL %EDI,4(%EBX)    # mm->max
```

Subroutine Example Code (cont.)

step 5: tear down stack frame

```
# we have no local variables to remove
# restore callee-saved registers
#(note that order is reversed!)
POPL %EDI
POPL %ESI
POPL %EBX
LEAVE
RET
```

alternate version (used by gcc)

```
LEAL -12(%EBP),%ESP
POPL %EDI
POPL %ESI
POPL %EBX
POPL %EBP
RET
```