Lecture Topics

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

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
```

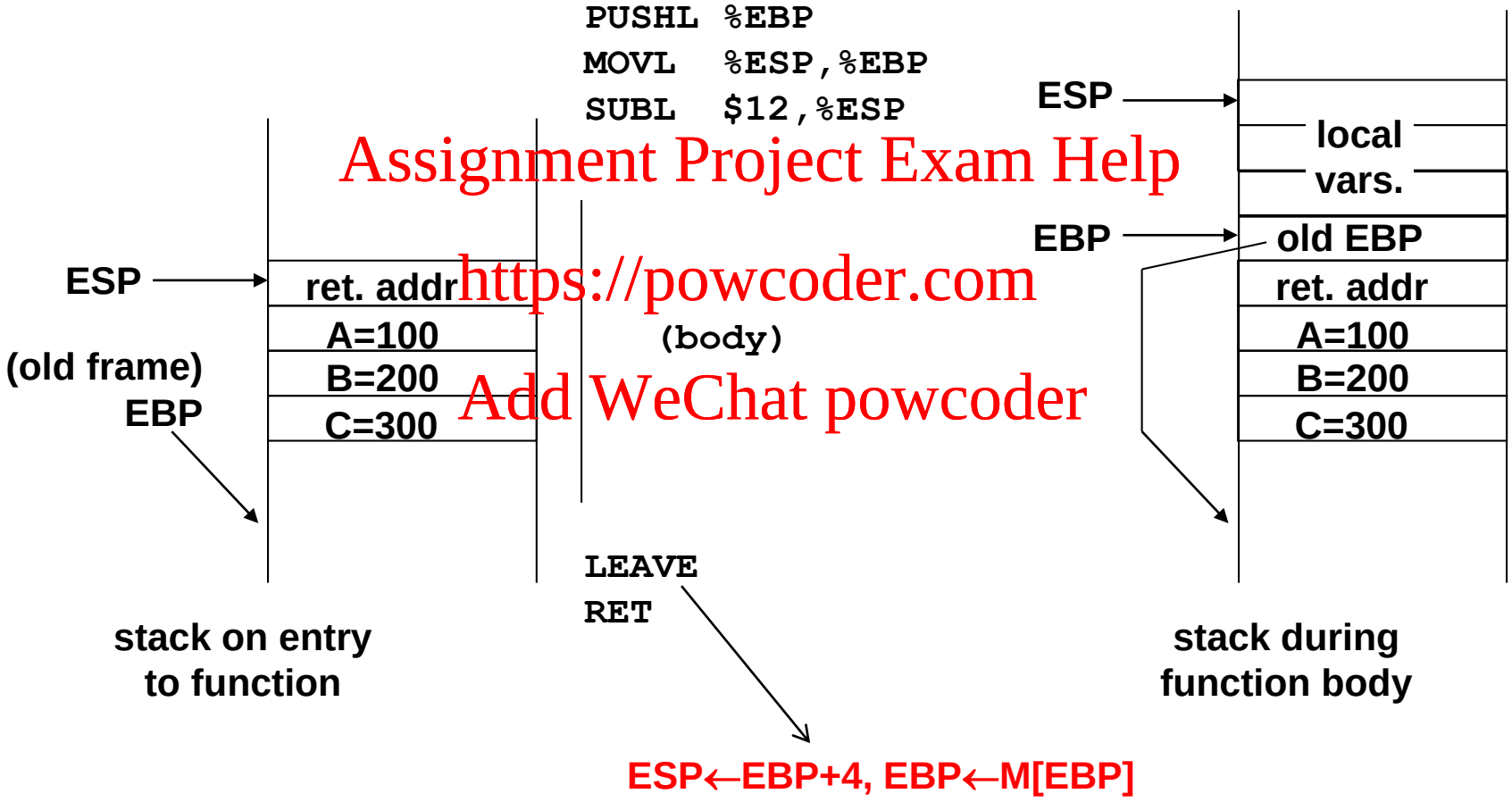
Stack Frames in x86 (4)

- 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)

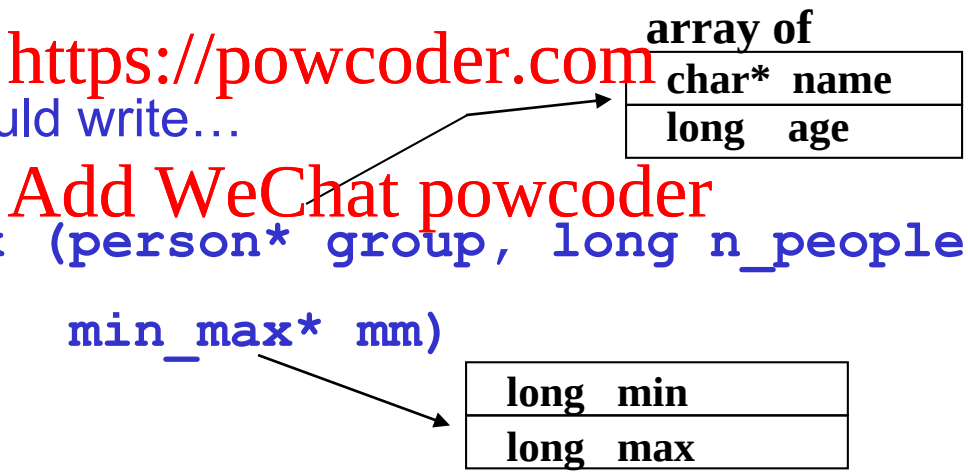


Subroutine Example Code

- 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…

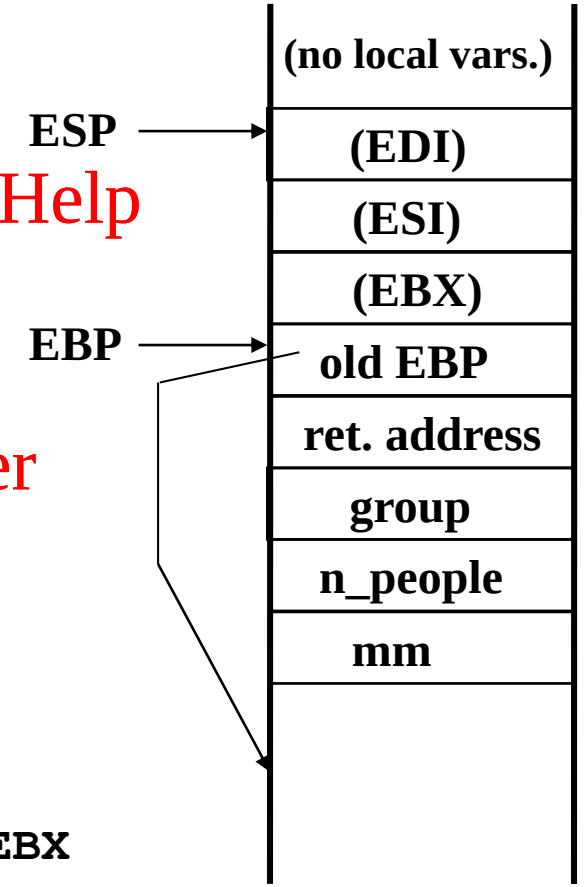
```
void find_min_max (person* group, long n_people,
                   min_max* mm)
```



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

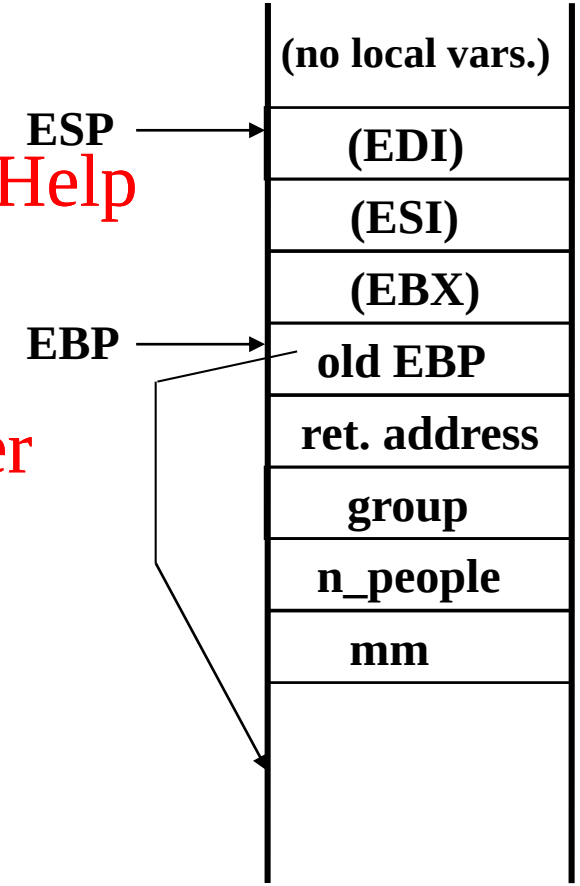
step 4: link from our output interface

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

Multiplication and Division

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



```
# multiple-operand forms are ONLY for signed operations
```



```
IMULL   %ECX,%EBX   # signed EBX ← EBX * ECX (high bits discarded)
IMULL   $20,%EDX,%ECX   # signed ECX ← 20 * EDX (high bits discarded)
```


quotient dividend

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

Data Type Alignment (1)

- 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

Data Type Alignment (2)

- 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

Device I/O

- How does a processor communicate with devices?
- Two possibilities
  - independent I/O — use special instructions and a separate I/O port address space
  - memory-mapped I/O — use loads/stores and dedicate part of the memory address space to I/O
- x86 originally used only independent I/O
  - but when used in PC, needed a good interface to video memory
  - solution? put card on the bus, claim memory addresses!
  - now uses both, although ports are somewhat deprecated