ECE391 Computer System Engineering Lecture 3

Dr. Zbigniew Kalbarczyk
University of Illinois at Urbana- Champaign

Fall 2018

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?

	#1	#2	#3	#4	#5	#6
Α	010	010	010	110	110	110
В	-000	<u>-110</u>	-111	-000	-011	<u>-111</u>
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 <	No	Yes	Yes	No	No	Yes
signed <	No	No	No	Yes	Yes	Yes

Control Flow Instructions (2)

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

		CF/OF						
		00	01	11	10			
SF	0	0	1	X	0			
.	1	1	X	0	1			

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 \neq < \leq = \geq > 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]

Stack

RET # EIP ← M[ESP], ESP ← ESP + 4

© Steven Lumetta, Zbigniew Kalbarczyk

© Steven Lumetta, Zbigniew Kalbarczyk

© Steven Lumetta, Zbigniew Kalbarczyk
```

Stack Operations

Push and pop supported directly by x86 ISA

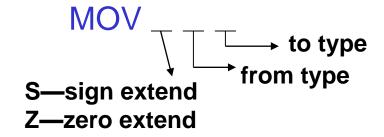
PUSHL %EAX # M[ESP – 4]
$$\leftarrow$$
 EAX, ESP \leftarrow ESP – 4

POPL %EBP # EBP
$$\leftarrow$$
 M[ESP], ESP \leftarrow ESP + 4

PUSHFL # M[ESP – 4]
$$\leftarrow$$
 EFLAGS, ESP \leftarrow ESP – 4

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
```

© Steven Supera 25000 example (24) define and #ice39 stude)

Code Example

Given: EBX pointing to an array of structures with ECX elements in the array the structure char* name Find: min and max age long age $ESI \leftarrow 0$ **EDX** ← large # EDI ← small # init vars **START ESI** ≥ Y find min/max ECX? loop over **END** elements compare one age first, define registers ESI — index into array ESI ← ESI + 1 EAX — current age

• nextenuse systematicade composition ece391

EDX — min age seen

EDI — max age seen

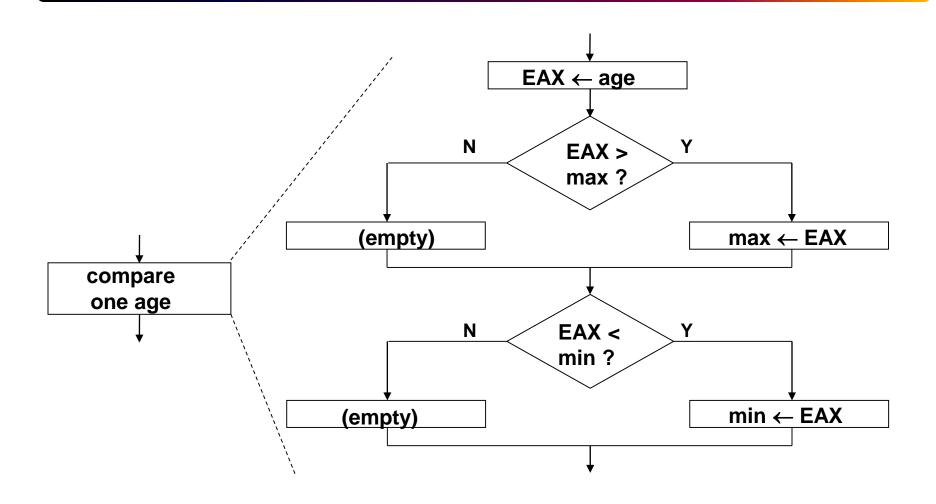
Code Example - Loop Design Process (1)

- 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)

```
# init vars
XORL %ESI, %ESI
                            1E52 K XOD000000
MOVL TWO MM, %EDX
MOVL TWO MM+4, %EDI
CMPL %ECX, %ESI # loop test
JGE
     DONE
# read the age using one memory reference.
MOVL 4(%EBX,%ESI,8),%EAX
     %EDI,%EAX
                      # check the max. age
CMPL
JLE
     NOT MAX
MOVL
     %EAX,%EDI
```

LOOP:

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 0x7FFFFFF, 0x8000000

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);
```

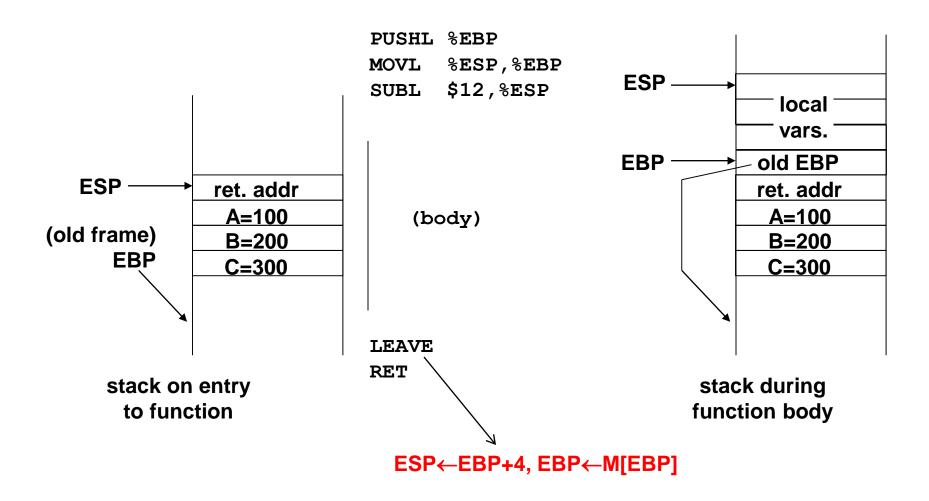
```
PUSHL $300
PUSHL $200
PUSHL $100
CALL func
ADDL $12,%ESP
# result in EAX
```

Stack Frames in x86 (4)

 Example of subroutine code and stack frame creation and teardown

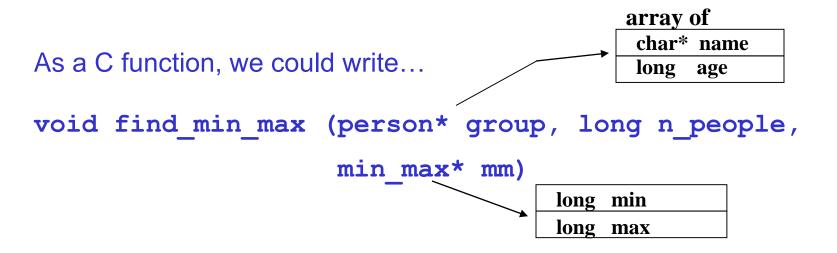
```
int func (int A, int B, int C)
{
  /* 12 bytes of local variables */
  ...
}
call func (100, 200, 300);
```

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)



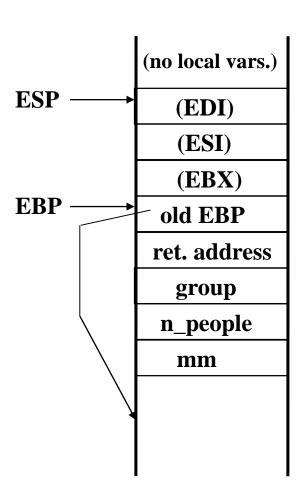
Subroutine Example Code (cont.)

```
step 1: create the stack frame
     PUSHL %EBP
                                                            (no local vars.)
     MOVL %ESP, %EBP
                                                 ESP
                                                               (EDI)
     PUSHL %EBX # protect callee-saved
                                                               (ESI)
                   # registers
                                                               (EBX)
     PUSHL %ESI
                                                 EBP
                                                             old EBP
     PUSHL %EDI
                                                             ret. address
 step 2: link to our input interface
                                                               group
     MOVL 8 (%EBP), %EBX # group
                                                             n_people
     MOVL 12(%EBP),%ECX # n_people
                                                               mm
 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
© Steven Lumetta, Zbigniew Kalbarczyk
```

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
```



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

Device I/O

- I/O instructions have not evolved since 8086
 - 16-bit port space
 - byte addressable
 - little-endian (looks like memory)
 - instructions
 - IN port, dest.reg
 - OUT src.reg, port
 - the register operands are NOT general-purpose registers
 - all data to/from AL, AX, or EAX
 - port is either an 8-bit immediate (not 16!) or DX