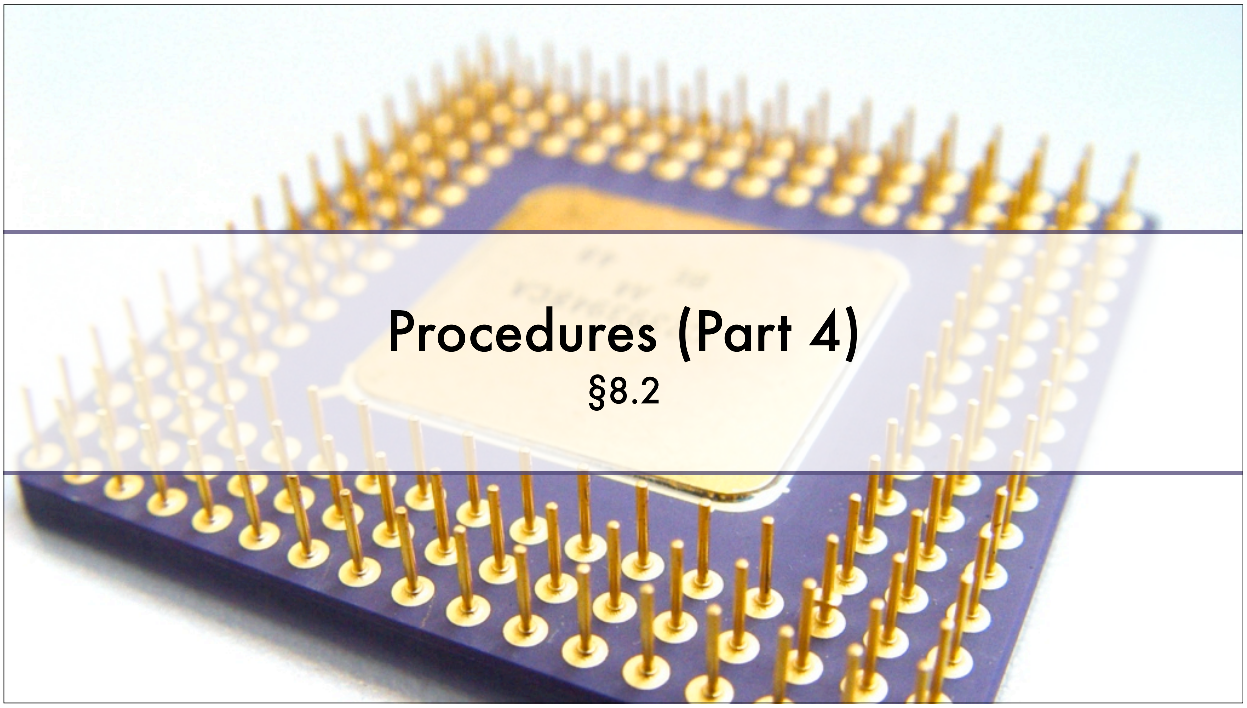

Procedures (Part 4)

§8.2

Activity 12

ESP Alignment & Locals

- ESP should always be aligned on a doubleword boundary, i.e., it must contain a memory address that's divisible by 4
  - Failure to do this may cause page faults, degraded performance
- Round up local variable storage to a multiple of 4 bytes
  - Need a 30-byte local array? Reserve 32 bytes.

```
.code
sample PROC
  push ebp
  mov ebp, esp
  sub esp, 32   ; Reserve 32 bytes, even though we only use 30
  ...
  add esp, 32
  pop ebp
  ret
sample ENDP
```

8-, 16-, 64-bit Arguments

- ESP should always be aligned on a doubleword boundary, i.e., it must contain a memory address that's divisible by 4
  - Failure to do this may cause page faults, degraded performance
- Always push 32-bit values, including stack arguments
  - Expand 8-, 16-bit values to 32 bits

8-, 16-, 64-bit Arguments

- ESP should always be aligned on a doubleword boundary, i.e., it must contain a memory address that's divisible by 4
  - Failure to do this may cause page faults, degraded performance
- Always push 32-bit values, including stack arguments
  - Expand 8-, 16-bit values to 32 bits (MOVZX/MOVSX)
  - Pass multiword arguments in little endian order

```
.data
q QWORD 1234567800ABCDEFh    ; In memory: EF CD AB 00 78 56 34 12
.code                        ;            ^q          ^q+4
push DWORD PTR [q + 4]
push DWORD PTR q             ; Now the 8 bytes on the stack are in the same order as q
call WriteHex64              ;    (little endian order in memory)
```

ENTER and LEAVE

- The ENTER and LEAVE instructions create and terminate stack frames
- Simplify prologue/epilogue code
- `ENTER` *numbytes*`, 0` is equivalent to

```
    push ebp
    mov ebp, esp
    sub esp, numbytes
```

- `LEAVE` is equivalent to

```
    mov esp, ebp
    pop ebp
```

LEA – Load Effective Address

-
  ```
  .data
  array DWORD 10, 20, 30, 40, 50
  ```
- Suppose we read an integer 0–4 into EAX, and we want to display the *memory address* of the element at that index
- We could retrieve that element using the indexed operand `[array + eax*4]`
- The *load effective address* (LEA) instruction determines the address of a memory operand and stores it in a register

-
  ```
  call ReadDec          ; Read integer 0–4 into EAX
  lea eax, [array+eax*4]    ; Store address in EAX
  call WriteHex         ; Display address of that element
  ```

LEA – Load Effective Address

- LEA is useful for creating an indirect operand from an indexed operand

```
; Receives three 32-bit unsigned integers stack parameters
AddThree PROC
   push ebp
   mov ebp, esp
   ; Display each of the three stack parameters
   lea esi, [ebp+8]         ; Point ESI at the first parameter
   mov ecx, 3
top:
   mov eax, [esi]
   call WriteDec
   add esi, SIZEOF DWORD ; Point ESI at the next parameter
   loop top
   ...
```

Topics Covered in Notes:

- ENTER instruction
- LEAVE instruction
- LEA instruction

Call by Value vs. Call by Reference

- *Value parameters* contain a value (e.g., integer)
  - For example, `min(int n, int m)`
  - This is what we've done so far
- *Reference parameters* contain a memory address
  - Passing an array "by value" would mean pushing every value in the array onto the stack – expensive!
  - Instead, pass the *address* (offset) of the array
- In Java, primitives (int, float, etc.) are passed by value;
  objects (arrays, strings, etc.) by reference

Example: Reference Parameters

```
INCLUDE Irvine32.inc

.data
aWord       WORD ?
anotherWord WORD ?

.code
main PROC
  ; Set the value of aWord to 5
  push OFFSET aWord
  call SetToFive

  ; Set the value of anotherWord to 5
  push OFFSET anotherWord
  call SetToFive

  exit
main ENDP
```

```
; Sets a WORD variable to 5. (STDCALL)
; Receives: [ebp+8] Address of variable
; Returns: None
SetToFive PROC
  enter 0, 0
  push edi

  ; Copy the variable’s address into EDI
  mov edi, [ebp+8]
  ; Set the variable’s value to 5
  mov WORD PTR [edi], 5

  pop edi
  leave
  ret 4
SetToFive ENDP

end main
```

Example: Reference Parameters

In C++, reference parameters are denoted by an ampersand (&).

The following compiles to essentially the same code as the previous slide:

```
unsigned short aWord, anotherWord;

void set_to_five(unsigned short &variable) {
  variable = 5;
}

void main() {
  set_to_five(aWord);
  set_to_five(anotherWord);
}
```

Example: Value Parameters

The following code does ***not*** *set the value of* aWord *or* anotherWord. *Why?*

It uses a value parameter. The next slide shows what it essentially compiles to...

```
unsigned short aWord, anotherWord;

void not_set_to_five(unsigned short variable) {
  variable = 5;
}

void main() {
  not_set_to_five(aWord);
  not_set_to_five(anotherWord);
}
```

Example: Value Parameters

```
INCLUDE Irvine32.inc

.data
aWord       WORD ?
anotherWord WORD ?

.code
main PROC
  ; Push the (uninitialized) value of aWord
  push aWord
  call NotSetToFive

  ; Push the value of anotherWord
  push anotherWord
  call NotSetToFive

  exit
main ENDP
```

```
NotSetToFive PROC
  enter 0, 0

  ; Change the value of the parameter
  ; (on the stack) to 5
  mov WORD PTR [ebp+8], 5

  leave
  ret 4
  ; After RET, the parameter is gone
  ; since we destroyed the stack frame
NotSetToFive ENDP

end main
```