Topic 5
Lecture 5
Procedures

CSCI 150

Assembly Language / Machine Architecture

Prof. Dominick Atanasio

Chapter Overview

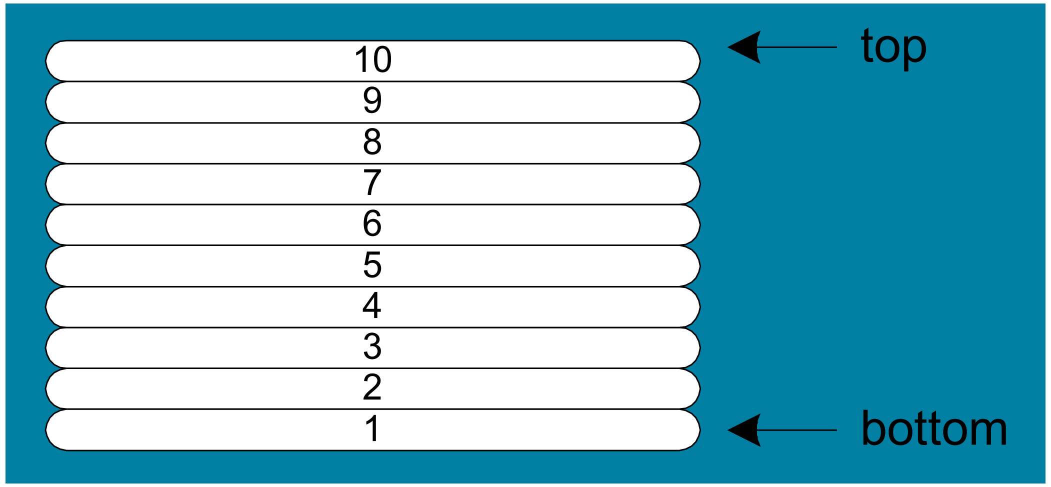
- Stack Operations
- Defining and Using Procedures
- Linking to an External Library

Stack Operations

- Runtime Stack
- PUSH Operation
- POP Operation
- PUSH and POP Instructions
- Using PUSH and POP
- Example: Reversing a String
- Related Instructions

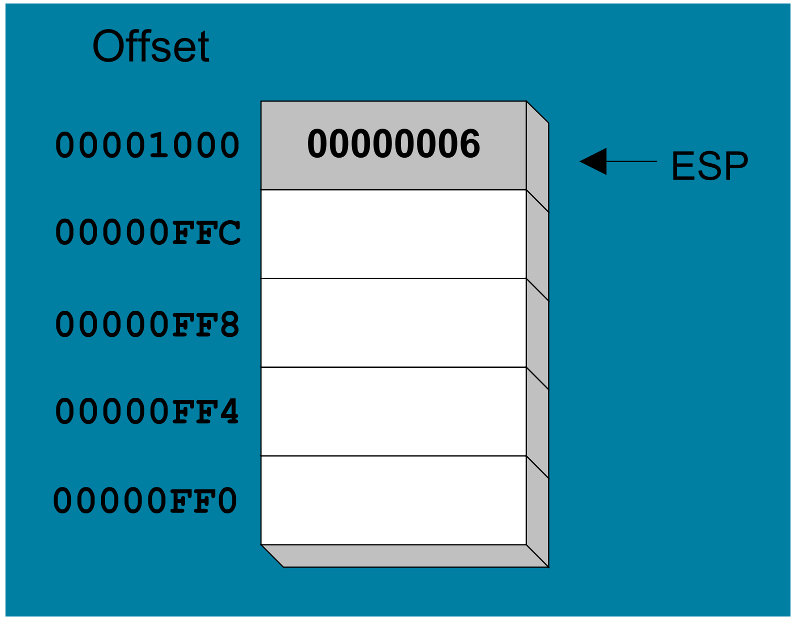
Runtime Stack (1 of 2)

- Imagine a stack of plates . . .
  - plates are only added to the top
  - plates are only removed from the top
  - LIFO structure



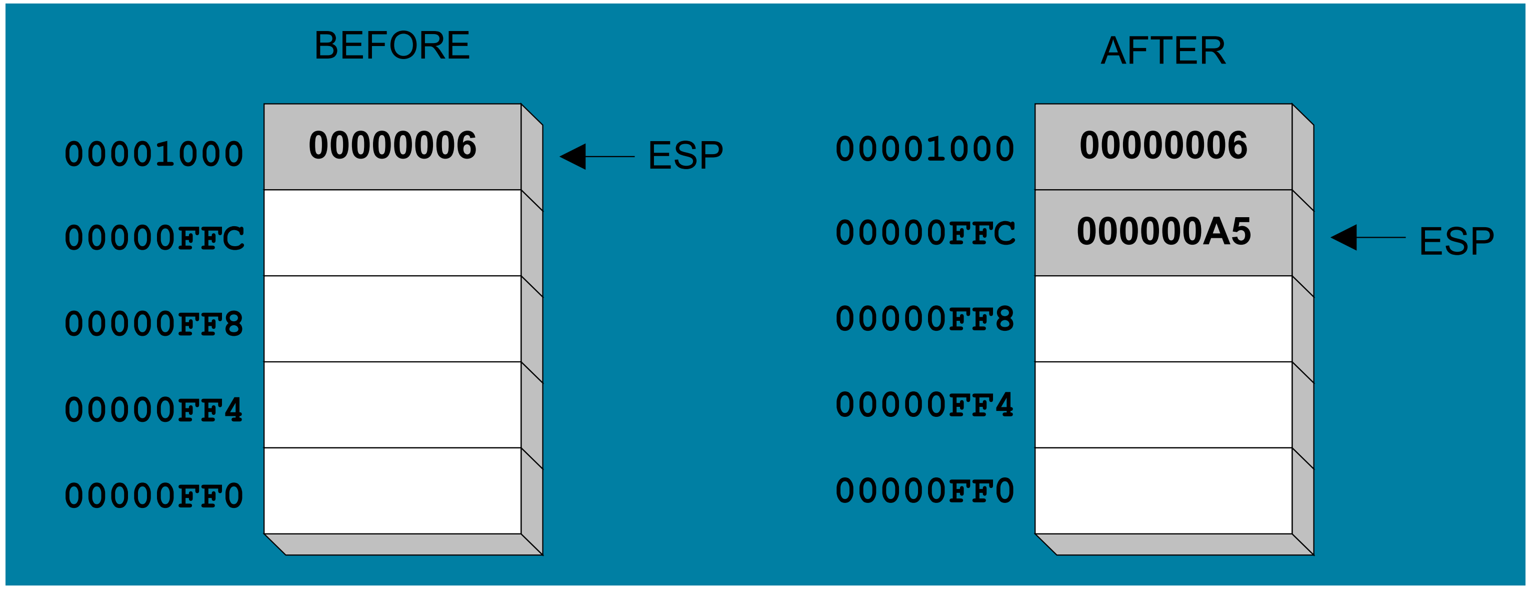
Runtime Stack (2 of 2)

- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) *



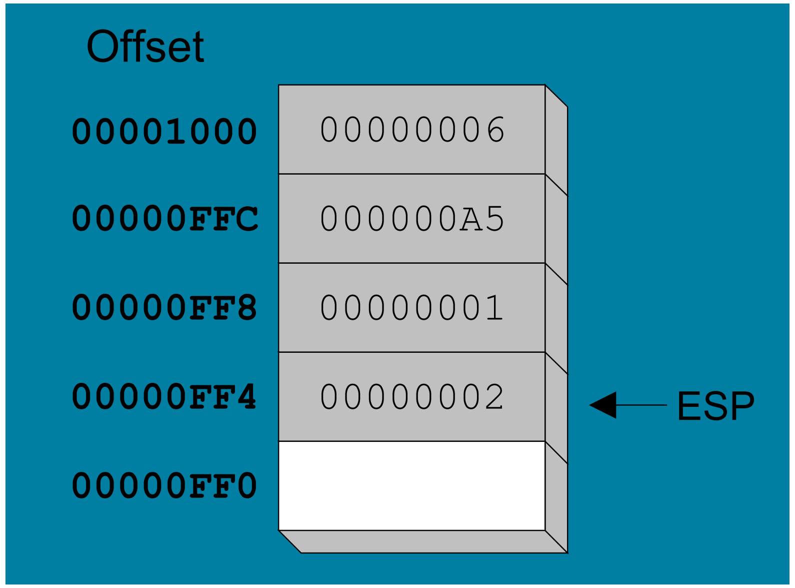
PUSH Operation (1 of 2)

- A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location pointed to by the stack pointer.



PUSH Operation (2 of 2)

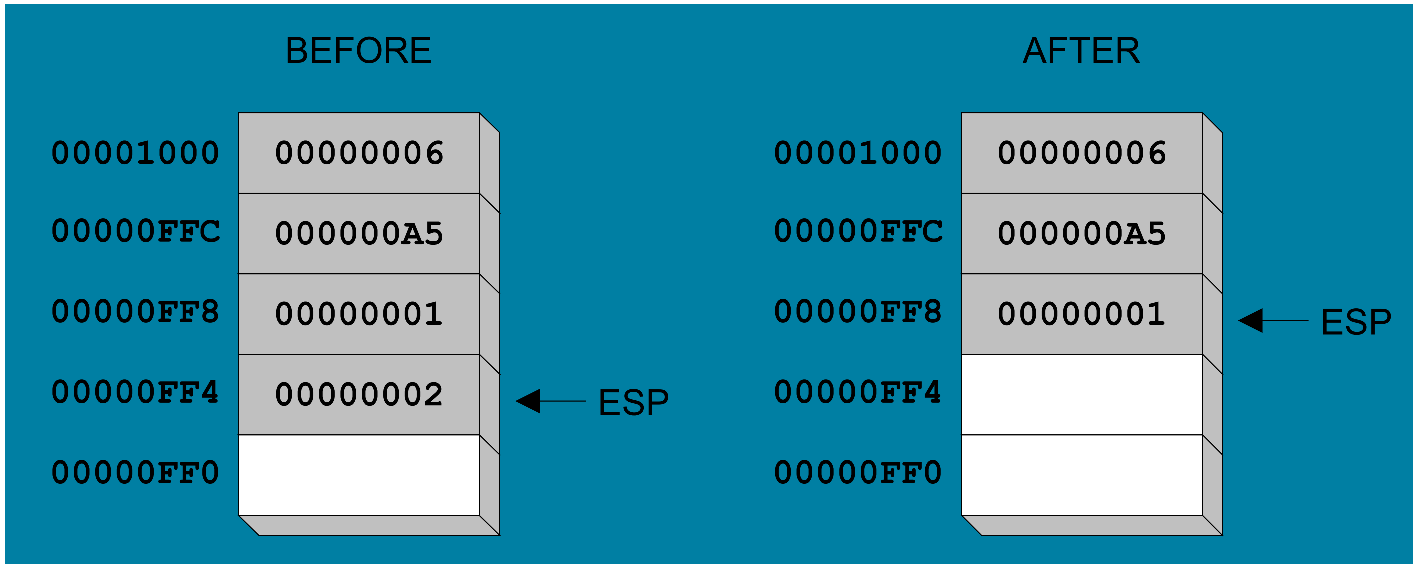
- Same stack after pushing two more integers:



The stack grows downward. The area below ESP is always available (unless the stack has overflowed).

POP Operation

- Copies value at [ESP] into a register or variable.
- Adds *n* to ESP, where *n* is either 2 or 4.
  - value of *n* depends on the attribute of the operand receiving the data



PUSH and POP Instructions

- PUSH syntax:
  - PUSH *r/m16*
  - PUSH *r/m32*
  - PUSH *imm32*
- POP syntax:
  - POP *r/m16*
  - POP *r/m32*

Using PUSH and POP

Save and restore registers when they contain important values. PUSH and POP instructions occur in the opposite order.

```
push eax                                        ; push registers
push ecx
push edx

mov  eax, dwordVal                              ; display some memory
mov  ecx, dwordVal
mov  edx, dwordVal
call dump_mem

pop  edx                                        ; restore registers
pop  ecx
pop  eax
```

Example: Nested Loop

When creating a nested loop, push the outer loop counter before entering the inner loop:

```
        mov ecx, 100                ; set outer loop count
L1:                                 ; begin the outer loop
        push ecx                    ; save outer loop count

        mov ecx, 20                 ; set inner loop count
L2:                                 ; begin the inner loop
        ;
        ;
        loop L 2                    ; repeat the inner loop

        pop ecx                     ; restore outer loop count
        loop L1                     ; repeat the outer loop
```

Example: Reversing a String

- Use a loop with indexed addressing
- Push each character on the stack
- Start at the beginning of the string, pop the stack in reverse order, insert each character back into the string
- Source code
- Q: Why must each character be put in EAX before it is pushed?

Because only word (16-bit) or doubleword (32-bit) values can be pushed on the stack.

Related Instructions

- PUSHFD and POPFD
  - push and pop the EFLAGS register
- PUSHAD pushes the 32-bit general-purpose registers on the stack
  - order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
- POPAD pops the same registers off the stack in reverse order
  - PUSHA and POPA do the same for 16-bit registers

Your Turn . . . (2 of 2)

- Write a program that does the following:
  - Assigns integer values to EAX, EBX, ECX, EDX, ESI, and EDI
  - Uses PUSHAD to push the general-purpose registers on the stack
  - Using a loop, your program should pop each integer from the stack using POP and view it in the debugger

What’s Next

- Stack Operations
- Defining and Using Procedures
- Linking to an External Library

Defining and Using Procedures

- Creating Procedures
- Documenting Procedures
- Example: sum_of Procedure
- CALL and RET Instructions
- Nested Procedure Calls
- Local and Global Labels
- Procedure Parameters
- Flowchart Symbols
- USES Operator

Creating Procedures

- Large problems can be divided into smaller tasks to make them more manageable
- A procedure is the ASM equivalent of a Java method or C++ function
- Following is an assembly language procedure named sample:

```
sample:
    .
    .
    ret
```

Documenting Procedures

Suggested documentation for each procedure:

- A description of all tasks accomplished by the procedure.
- Receives: A list of input parameters; state their usage and requirements.
- Returns: A description of values returned by the procedure.
- Requires: Optional list of requirements called preconditions that must be satisfied before the procedure is called.

If a procedure is called without its preconditions satisfied, it will probably not produce the expected output.

Example: SumOf Procedure

```
;------------------------------------------------------------------------------------------------
sum_of:
;
; Calculates and returns the sum of three 32-bit integers.
; Receives: EAX, EBX, ECX, the three integers. May be signed or unsigned.
; Returns: EAX = sum, and the status flags (Carry, Overflow, etc.) are changed.
; Requires: nothing
;------------------------------------------------------------------------------------------------

   add eax, ebx
   add eax, ecx
   Ret

; End sum_of
;------------------------------------------------------------------------------------------------
```

CALL and RET Instructions

- The CALL instruction calls a procedure
  - pushes offset (EIP) of next instruction on the stack
  - copies the address of the called procedure into EIP
- The RET instruction returns from a procedure
  - pops top of stack into EIP

CALL-RET Example (1 of 2)

0000025 is the offset of the instruction immediately following the CALL instruction

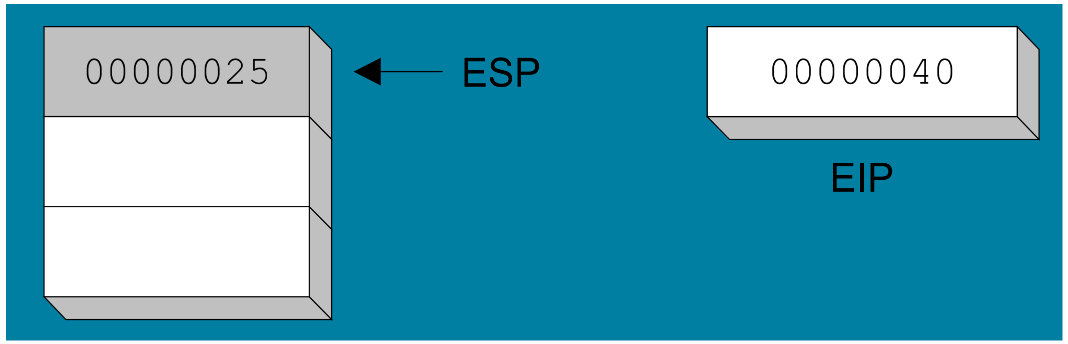
00000040 is the offset of the first instruction inside MySub

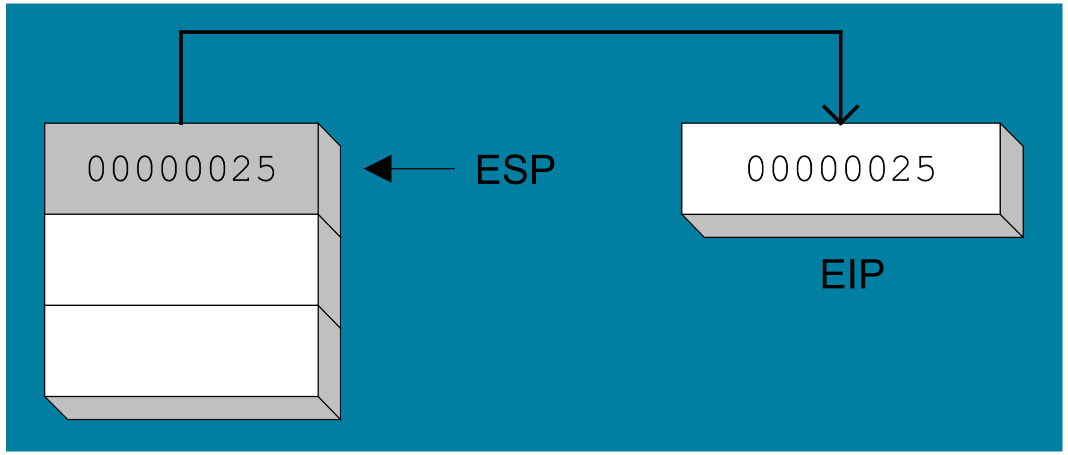
```
main:
        00000020 call my_sub
        00000025 mov eax, ebx
        .
        .

my_sub:
        00000040 mov eax, edx
        .
        .
        ret
```

CALL-RET Example (2 of 2)

The CALL instruction pushes 00000025 onto the stack, and loads 00000040 into EIP

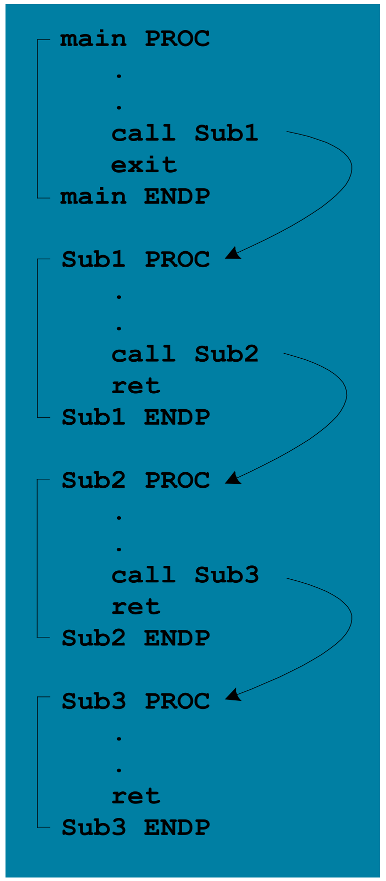
The RET instruction pops 00000025 from the stack into EIP





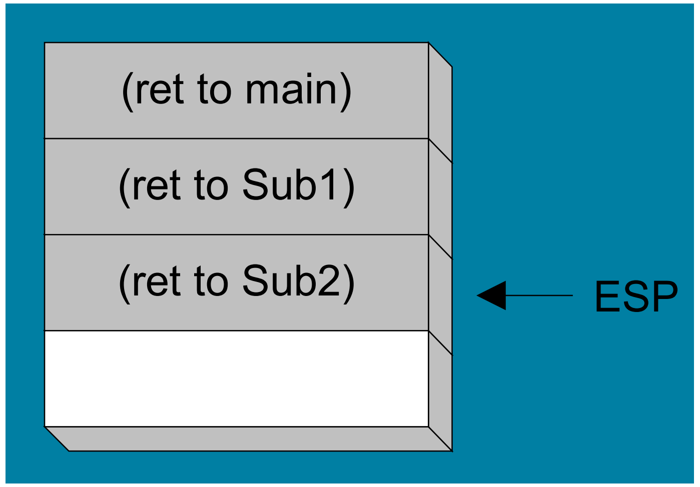
(stack shown before RET executes)

Nested Procedure Calls



```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

Sub1 PROC
    .
    .
    call Sub2
    ret
Sub1 ENDP

Sub2 PROC
    .
    .
    call Sub3
    ret
Sub2 ENDP

Sub3 PROC
    .
    .
    ret
Sub3 ENDP
```

By the time Sub3 is called, the stack contains all three return addresses:



Local and Global Labels

A local label is visible only to statements inside the same procedure. A global label is visible everywhere.

```
main:
        jmp L2                                  ; error
L1:                                             ; global label

        call exit

sub2:
.L2:                                            ; local label
        jmp L1                                  ; ok
        ret

exit:
…
```

Procedure Parameters (1 of 3)

- A good procedure might be usable in many different programs
  - but not if it refers to specific variable names
- Parameters help to make procedures flexible because parameter values can change at runtime

Procedure Parameters (2 of 3)

The ArraySum procedure calculates the sum of an array. It makes two references to specific variable names:

```
array_sum:
        mov esi, array                  ; array address
        mov eax, 0                      ; set the sum to zero
        mov ecx, aLen                   ; set number of elements

.L1:    add eax, [esi]                  ; add each integer to sum
        add esi, 4                      ; point to next integer
        loop .L1                        ; repeat for array size

        mov [sum], eax                  ; store the sum
        ret
```

What if you wanted to calculate the sum of two or three arrays within the same program?

Procedure Parameters (3 of 3)

This version of ArraySum returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:

```
array_sum
;  Receives: ESI which to an array of dwords,
;  ECX = number of array elements.
;  Returns: EAX = sum
;-----------------------------------------------------------------------------------------
        mov eax,0                               ; set the sum to zero

.L1:    add eax, [esi]                          ; add each integer to sum
        add esi,4                               ; point to next integer
        loop .L1                                ; repeat for array size

        ret
```

When not to push a register

- The sum of the three registers is stored in EAX on line (3), but the POP instruction replaces it with the starting value of EAX on line (4):

```
SumOf PROC                          ; sum of three integers
        push eax                    ; 1
        add eax,ebx                 ; 2
        add eax,ecx                 ; 3
        pop eax                     ; 4
        ret
SumOf ENDP
```

Rules of Procedure Calling (1 of 2)

- Caller’s Rules
  - Before calling a subroutine, the caller should save the contents of certain registers that are designated caller-saved.
    - The caller-saved registers are EAX, ECX, EDX.
    - Theses are not preserved by the Callee.
  - If passing parameters to the subroutine using the stack, push them onto the stack before the call.
    - The parameters should be pushed in inverted order (i.e. last parameter first) – since the stack grows down. The first parameter should be at the lowest address.
    - After the subroutine returns, the caller must immediately remove the parameters from stack.
    - The caller restores the contents of caller-saved registers (EAX, ECX, EDX) by popping them off of the stack. The caller can assume that no other registers were modified by the subroutine.
  - To call the subroutine, use the call instruction.
    - This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code.
  - The caller can expect to find the return value of the subroutine in the register EAX.

Rules of Procedure Calling (2 of 2)

- Callee's Rules
  - At the beginning of the subroutine, the function should push the value of EBP onto the stack, and then copy the value of ESP into EBP.
    - Establishes a base pointer for the activation record.
    - EBP must be restored (popped) before the function returns.
  - If local memory storage is needed for the procedure, allocate local variables by making space on the stack.

```
sub esp , 12
```

    - As with parameters, local variables will be located at known offsets from the base pointer
  - The values of any registers that are designated callee-saved that will be used by the function must be saved (pushed onto the stack). They should be popped from the stack in the reverse order.
    - The callee-saved registers are EBX, EDI and ESI.
  - When the function is done, the return value for the function (if any) should be placed in EAX if it is not already there.
  - The function must restore the old values of any callee-saved registers (EBX, EDI and ESI)
  - Local variables nust be deallocated (mov esp , ebp)
  - Immediately before returning, we must restore the caller's base pointer value by popping EBP

What’s Next (1 of 3)

- Stack Operations
- Defining and Using Procedures
- Linking to an External Library

Linking to an External Library

- What is a Link Library?
- How the Linker Works

What is a Link Library?

- A file containing procedures that have been compiled into machine code
  - constructed from one or more object files
- To build a library, . . .
  - start with one or more ASM source files
  - assemble each into an object file
  - link object files with the object file containing _start