Part 1: Stack

- **Stack**
  - The stack is part of memory used for temporary storage of addresses and data.
  - Stack is a Last-In-First-Out (LIFO) data structure

- **Runtime Stack**
  - The runtime stack is a memory array managed directly by the CPU, using the ESP register, known as the extended stack pointer register.
  - The ESP register holds a 32-bit offset into some location on the stack.
  - ESP always points to the last value to be added to, or pushed on, the top of stack.
  - `.STACK` directive specifies a runtime stack
- Two basic stack operations
  - **Push**: inserts a new element on top of the stack
  - **Pop**: deletes top element from the stack

**Push Instruction**
The PUSH Instruction first decrements ESP and then copies either a 16- or 32-bit source operand into the stack.
- A 16-bit operand causes ESP to be decremented by 2.
- A 32-bit operand causes ESP to be decremented by 4.

![Before and After Diagram for Push Instruction](image1)

- There are three instruction formats:
  - PUSH r/m16
  - PUSH r/m32
  - PUSH imm32

**Pop Instruction**
The POP Instruction first copies the contents of the stack element pointed to by ESP into a 16- or 32-bit destination operand and then increments ESP.
- A 16-bit operand causes ESP to be incremented by 2.
- A 32-bit operand causes ESP to be incremented by 4.

![Before and After Diagram for Pop Instruction](image2)

- There are two instruction formats:
  - POP r/m16
  - POP r/m32
Q (True/False): Local variables in procedures are created on the stack.

A True

Q (True/False): The PUSH instruction cannot have an immediate operand.

A False

**Example 1: Nested Loop**

```assembly
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 L2 ; repeat the inner loop
    pop ecx ; restore outer loop count
    loop L1 ; repeat the outer loop
```

**PUSHFD and POPFD Instructions**

The PUSHFD instruction pushes the 32-bit EFLAGS register on the stack, and POPFD pops the stack into EFLAGS:

- pushfd
- popfd

16-bit programs use the PUSHF instruction to push the 16-bit FLAGS register on the stack and POPF to pop the stack into FLAGS.

**PUSHAD, PUSHA, POPAD, and POPA**

- The PUSHAD instruction pushes all of the 32-bit general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI.

- ESP (value before executing PUSHAD)

- The POPAD instruction pops the same registers off the stack in reverse order.

- The PUSHA instruction, pushes the 16-bit general-purpose registers (AX, CX, DX, BX, SP, BP, SI, DI) on the stack in the order listed.

- The POPA instruction pops the same registers in reverse order.
Q Suppose there was no PUSH instruction. Write a sequence of two other instructions that would accomplish the same as PUSH EAX.

A

sub esp,4
mov [esp],eax

Part 2: Procedures

Procedures
- A procedure is the ASM equivalent of a Java or C++ function.
- Called sometimes a function, subprogram, or subroutine
- Receives a list of parameters, also called arguments
- Performs computation and returns results
- Following is an assembly language procedure named sample:

```assembly
sample PROC
  ;
  ret
sample ENDP
```

Q (True/False): It is possible to define a procedure inside an existing procedure.

A False

CALL and RET Instructions

- The CALL instruction calls a procedure
  - pushes offset 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
- The ret instruction returns control to the caller

main PROC
  call MySub ; 00000020
  mov eax,ebx ; 00000025
  ;
main ENDP
MySub PROC
  mov eax,edx ; 00000040
  ;
  ret
MySub ENDP
00000025 is the offset of the instruction immediately following the CALL instruction.
00000040 is the offset of the first instruction inside MySub.

Q What would happen if the RET instruction was omitted from a procedure?

A Execution would continue beyond the end of the procedure, possibly into the beginning of another procedure.

Nested Procedure Calls

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

```
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
```

Parameter Passing

- Two common mechanisms of parameter passing
  - Pass-by-value: parameter value is passed
  - Pass-by-reference: address of parameter is passed

Example:

```
ArraySum PROC
 ; Receives: ESI points to an array of doublewords, ECX = number of array elements.
 ; Returns: EAX = sum
 ;---------------------------------------------
   push esi    ; save esi, it is modified
   push ecx    ; save ecx, it is modified
   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
pop ecx ; restore registers
pop esi ; in reverse order
ret

ArraySum ENDP

- ESI: Reference parameter = array address
- ECX: Value parameter = count of array elements

**USES Operator**

- The USES operator, coupled with the PROC directive, lets you list the names of all registers modified within a procedure.
- USES tells the assembler to do two things:
  - First, generate PUSH instructions that save the registers on the stack at the beginning of the procedure.
  - Second, generate POP instructions that restore the register values at the end of the procedure.
- The USES operator immediately follows PROC, and is itself followed by a list of registers on the same line separated by spaces or tabs (not commas).

**Q** (True/False): The ArraySum procedure receives a pointer to any array of doublewords.

A True

**Q** Which statement(s) in the ArraySum procedure would have to be modified so it could accumulate an array of 16-bit words?

A The following statements would have to be modified:
  - `add eax, [esi]` becomes `add ax, [esi]`
  - `add esi,4` becomes `add esi,2`
Homework:
1. From Book (7th edition)
   Section Review 5.1.3: 1, 2, 3, 4, 5, 6
   Section Review 5.2.7: 1, 2, 3, 4, 5, 6

Quiz Next Week in Chapters 4, 5