Chapter 5 (Part b)

Procedures

Eng. Eman R. Habib

April, 2014
5.4 Stack Operations
- The runtime stack is a memory array managed directly by the CPU, using the ESP register, known as the 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.
- A stack is also called a LIFO structure (Last-In, First-Out) because the last value put into the stack is always the first value taken out.

- Each stack location in this figure contains 32 bits, which is the case when a program is running in 32-bit mode.
- In 16-bit real-address mode, the SP register points to the most recently pushed value and stack entries are typically 16 bits long.

**Push Operation**
A 32-bit push operation:
- Decrements the stack pointer by 4
- Then copies a value into the location in the stack pointed to by the stack pointer.

The following figure shows the effect of pushing 000000A5 on a stack that already contains one value (00000006).

![Stack Operations Diagram](image-url)
The runtime stack grows downward in memory, from higher addresses to lower addresses. Before the push, ESP = 00001000h; after the push, ESP = 0000FFCh.

- A 16-bit operand causes ESP to be decremented by 2.
- A 32-bit operand causes ESP to be decremented by 4.

There are three instruction formats:

- `PUSH reg/mem16`
- `PUSH reg/mem32`
- `PUSH imm32`

❖ **Pop Operation**

A pop operation:
- Removes a value from the stack.
- Increment the stack pointer (by the stack element size) to point to the next-highest location in the stack.

There are two instruction formats:

- `POP reg/mem16`
- `POP reg/mem32`

❖ **PUSHFD Instruction**

Pushes the 32-bit EFLAGS register on the stack.

❖ **POPFD Instruction**

Pops the stack into EFLAGS.
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**
  Pushes all of the 32-bit general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (value before executing PUSHAD), EBP, ESI, and EDI.

- **POPAD**
  Pops the same registers off the stack in reverse order.

Similarly, 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.

- **Example: Reversing a String**
  The *RevStr.asm* program loops through a string and pushes each character on the stack. It then pops the letters from the stack (in reverse order) and stores them back into the same string variable. Because the stack is a LIFO (last-in, first-out) structure, the letters in the string are reversed:

```
TITLE Reversing a String (RevStr.asm)
INCLUDE Irvine32.inc
.data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) - 1
.code
main PROC
    ; Push the name on the stack.
    mov ecx,nameSize
    mov esi,0
L1:     movzx eax,aName[esi]         ; get character
    push eax                       ; push on stack
    inc esi
    loop L1
    ; Pop the name from the stack, in reverse, and store in the aName.
    mov ecx,nameSize
    mov esi,0
L2:     pop eax                      ; get character
    mov aName[esi],al              ; store in string
    inc esi
    loop L2
    ; Display the name.
    mov edx,OFFSET aName
    call WriteString
    call Crlf
    exit
main ENDP
END main
```
Section 5.4 Review

1. Which register (in protected mode) manages the stack? 
   ESP

2. How is the runtime stack different from the stack abstract data type? 
   The runtime stack works at the system level to handle subroutine calls. The stack ADT is a programming construct typically written in a high-level programming language such as C++ or Java. It is used when implementing algorithms that depend on last-in, first-out operations.

3. Why is the stack called a LIFO structure? 
   LIFO stands for "last in, first out". The last value pushed into the stack is the first value popped out from the stack.

4. When a 32-bit value is pushed on the stack, what happens to ESP? 
   Decremented by 4

5. (True/False) Only 32-bit values should be pushed on the stack when using the Irvine32 library.

6. (True/False) Only 16-bit values should be pushed on the stack when using the Irvine16 library.

7. (True/False) Local variables in procedures are created on the stack.

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

9. Which instruction pushes all of the 32-bit general-purpose registers on the stack? 
   PUSHAD

10. Which instruction pushes the 32-bit EFLAGS register on the stack? 
    PUSHFD

11. Which instruction pops the stack into the EFLAGS register? 
    POPFD

12. Challenge: Another assembler (called NASM) permits the PUSH instruction to list multiple specific registers. Why might this approach be better than the PUSHAD instruction in MASM? 
    Here is a NASM example: 
    PUSH EAX EBX ECX
    It is better when we don’t need to save all registers but some of them, and we need the new values of the others.
13. Challenge: Suppose there were no PUSH instruction. Write a sequence of two other instructions that would accomplish the same as PUSH EAX.

```
sub esp,4
mov [esp],eax
```

5.5 Defining and Using Procedures

- Informally, we can define a procedure as a named block of statements that ends in a return statement.
- A procedure is declared using the PROC and ENDP directives. It must be assigned a name (a valid identifier).
- When you create a procedure other than your program’s startup procedure, end it with a RET instruction. RET forces the CPU to return to the location from where the procedure was called.
- Example:

```
sample PROC
   .
   ret
sample ENDP
```

- The startup procedure (main) is a special case because it ends with the exit statement.

❖ CALL and RET Instructions

- The CALL instruction calls a procedure by directing the processor to begin execution at a new memory location.
- The procedure uses a RET (return from procedure) instruction to bring the processor back to the point in the program where the procedure was called.
- The CALL instruction pushes its return address on the stack and copies the called procedure’s address into the instruction pointer.
- When the procedure is ready to return, its RET instruction pops the return address from the stack into the instruction pointer.
- In 32-bit mode, the CPU executes the instruction in memory pointed to by EIP (instruction pointer register). In 16-bit mode, IP points to the instruction.

❖ Example:

```
main PROC
00000020  call MySub
00000025  mov eax,ebx

MySub PROC
00000040  mov eax,edx
   .
   .
   ret
MySub ENDP
```
Executing a CALL Instruction:
- The address following the call (00000025) is pushed on the stack and the address of MySub is loaded into EIP.
- All instructions in MySub execute up to its RET instruction.

Executing the RET Instruction:
- The value in the stack pointed to by ESP is popped into EIP (step 1).
- ESP is incremented so it points to the previous value on the stack (step 2).

 Nested Procedure Calls
A nested procedure call occurs when a called procedure calls another procedure before the first procedure returns. Suppose that main calls a procedure named Sub1. While Sub1 is executing, it calls the Sub2 procedure. While Sub2 is executing, it calls the Sub3 procedure. The process is shown in following figure:
The following diagram shows the stack just before the return from Sub3 is executed:

![Stack Diagram]

- **Passing Register Arguments to Procedures**

  Place all required parameters in a general-purpose register then call the procedure.

  Two common mechanisms of parameter passing:
  - Pass-by-value: parameter value is passed.
  - Pass-by-reference: address of parameter is passed.
**Example: Summing an Integer Array**

```assembly
;--------------------------------------------------------------------------
ArraySum PROC
;
; Calculates the sum of an array of 32-bit integers.
; Receives: ESI = the array offset
; ECX = number of elements in the array
; Returns: EAX = sum of the array elements
;--------------------------------------------------------------------------
push esi ; save ESI, ECX
push ecx
mov eax,0 ; set the sum to zero
L1: add eax,[esi] ; add each integer to sum
    add esi,TYPE DWORD ; point to next integer
    loop L1 ; repeat for array size
pop ecx ; restore ECX, ESI
pop esi
ret ; sum is in EAX
ArraySum ENDP
```

**Calling ArraySum:**
Passing the address of array in ESI (Reference parameter) and the array count in ECX (Value parameter). After the call, we copy the sum in EAX to a variable:

```assembly
.data
array DWORD 10000h,20000h,30000h,40000h,50000h
theSum DWORD ?
.code
main PROC
    mov esi,OFFSET array ; ESI points to array
    mov ecx,LENGTHOF array ; ECX = array count
    call ArraySum ; calculate the sum
    mov theSum,eax ; returned in EAX
```

**Saving and Restoring Registers**
Always save and restore registers that are modified by a procedure so the calling program can be sure that none of its own register values will be overwritten. The exception to this rule pertains to registers used as return values, usually EAX. Do not push and pop them.

**USES Operator**
The USES operator, 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).

❖ Example:

```
ArraySum PROC USES esi ecx
    mov eax,0 ; set the sum to zero
L1:
    add eax,[esi] ; add each integer to sum
    add esi,TYPE DWORD ; point to next integer
    loop L1 ; repeat for array size
ret ; sum is in EAX
ArraySum ENDP
```

❖ Section 5.5 Review

1. **(True/False):** The PROC directive begins a procedure and the ENDP directive ends a procedure.

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

3. **What would happen if the RET instruction was omitted from a procedure?**
   Execution would continue beyond the end of the procedure, possibly into the beginning of another procedure.

4. **How are the words Receives and Returns used in the suggested procedure documentation?**
   - A list of input parameters and their usage, labeled by a word such as Receives.
   - A description of any values returned by the procedure, labeled by a word such as Returns.

5. **(True/False):** The CALL instruction pushes the offset of the CALL instruction on the stack.

6. **(True/False):** The CALL instruction pushes the offset of the instruction following the CALL on the stack.

7. **(True/False):** The RET instruction pops the top of the stack into the instruction pointer.
8. (True/False): Nested procedure calls are not permitted by the Microsoft assembler unless the NESTED operator is used in the procedure definition.

9. (True/False): In protected mode, each procedure call uses a minimum of 4 bytes of stack space.

10. (True/False): The ESI and EDI registers cannot be used when passing parameters to procedures.

11. (True/False): The ArraySum procedure (Section 5.5.3) receives a pointer to any array of doublewords.

12. (True/False): The USES operator lets you name all registers that are modified within a procedure.

13. (True/False): The USES operator only generates PUSH instructions, so you must code POP instructions yourself.

14. (True/False): The register list in the USES directive must use commas to separate the register names.

15. Which statement(s) in the ArraySum procedure (Section 5.5.3) would have to be modified so it could accumulate an array of 16-bit words? Create such a version of ArraySum and test it.
   The following statements would have to be modified:
   add eax,[esi] becomes → add ax,[esi]
   add esi,4 becomes → add esi,2

😊 Best Wishes 😊