CS2422 Assembly Language & System Programming

October 24, 2006

Today’s Topics

• Stack Operations
• Defining and Using Procedures
Study Guide

• Section 5.4: Stack Operations.
• Section 5.5: Defining and Using Procedures.

• Also browse Sections 5.1 to 5.3.

Stack Operations

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

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

* SP in Real-address mode

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)

- This is the 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 stack[ESP] into a register or variable.
- Adds \( n \) to ESP, where \( n \) is either 2 or 4.
  - depends on 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 $r/m32$

$r/m$ meaning register/memory

Using PUSH and POP

Save and restore registers when they contain important values. Note that the PUSH and POP instructions are in the opposite order:

```
push esi         ; push registers
push ecx
push ebx
mov esi,OFFSET dwordVal ; starting OFFSET
mov ecx,LENGTHOF dwordVal ; number of units
mov ebx,TYPE dwordVal ; size of doubleword
call DumpMem     ; display memory
pop ebx          ; opposite order
pop ecx
pop esi
```
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 into the string
- Source code (See pp.157-158, `RevStr.asm`)
- 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
  - PUSHAD and POPA do the same for 16-bit registers
Creating Procedures

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

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

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 (Note: IP=Instruction Pointer)
• The RET instruction returns from a procedure
  – pops top of stack into EIP
A Quick Exercise

• Aren’t CALL and RET two independent instructions? How does RET know where to return to?

CALL-RET Example (1 of 2)

<table>
<thead>
<tr>
<th>main</th>
<th>PROC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00000020 call MySub</td>
</tr>
<tr>
<td></td>
<td>00000025 mov eax,ebx</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>main ENDP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MySub PROC</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000040 mov eax,edx</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>ret</td>
</tr>
<tr>
<td>MySub ENDP</td>
</tr>
</tbody>
</table>
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.

Nested Procedure Calls

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

(ret to main)
(ret to Sub1)
(ret to Sub2)
Local and Global Labels

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

```
main PROC
  jmp L2 ; error!
L1:: ; global label
  exit
main ENDP
sub2 PROC
L2: ; local label
  jmp L1 ; ok
  ret
sub2 ENDP
```

Procedure Parameters (1 of 2)

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

```
ArraySum PROC
  mov esi,0 ; array index
  mov eax,0 ; set the sum to zero
L1: add eax,myArray[esi]; add each integer to sum
  add esi,4 ; point to next integer
  loop L1 ; repeat for array size
  mov theSum,eax ; store the sum
  ret
ArraySum ENDP
```

What if you wanted to calculate the sum of two or three arrays using the same program?
Procedure Parameters (2 of 2)

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

```
ArraySum PROC
    ; Receives: ESI points to an array of doublewords,
    ;         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
ArraySum ENDP
```

USES Operator

- Lists the registers that will be saved

```
ArraySum PROC USES esi ecx
    mov eax,0 ; set the sum to zero
    ...
    pop ecx
    pop esi
    ret
ArraySum ENDP
```
Now, we’ll go through Sections 5.1 to 5.3 very quickly…

Link Library Overview

• A file containing procedures that have been compiled into machine code
  – constructed from one or more OBJ files
• To build a library, . . .
  – start with one or more ASM source files
  – assemble each into an OBJ file
  – create an empty library file (extension .LIB)
  – add the OBJ file(s) to the library file, using the Microsoft LIB utility
Calling a Library Procedure

- Call a library procedure using the CALL instruction. Some procedures require input arguments.
- The INCLUDE directive copies in the procedure prototypes (declarations).
- The following example displays "1234" on the console:

```
INCLUDE Irvine32.inc
.CODE
    mov eax,1234h ; input argument
    call WriteHex ; show hex number
    call Crlf ; end of line
```

Linking to a Library

- Your programs link to Irvine32.lib using the linker command inside a batch file named make32.bat.
- Notice the two LIB files: Irvine32.lib, and kernel32.lib
  - the latter is part of the Microsoft Win32 Software Development Kit
Library Procedures - Overview (1 of 3)

Clrscr - Clears the console and locates the cursor at the upper left corner.

Crlf - Writes an end of line sequence to standard output.

Delay - Pauses the program execution for a specified \( n \) millisecond interval.

DumpMem - Writes a block of memory to standard output in hexadecimal.

DumpRegs - Displays the EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS, and EIP registers in hexadecimal. Also displays the Carry, Sign, Zero, and Overflow flags.

GetCommandtail - Copies the program’s command-line arguments (called the \textit{command tail}) into an array of bytes.

GetMseconds - Returns the number of milliseconds that have elapsed since midnight.

Library Procedures - Overview (2 of 3)

Gotoxy - Locates cursor at row and column on the console.

Random32 - Generates a 32-bit pseudorandom integer in the range 0 to FFFFFFFFH.

Randomize - Seeds the random number generator.

RandomRange - Generates a pseudorandom integer within a specified range.

ReadChar - Reads a single character from standard input.

ReadHex - Reads a 32-bit hexadecimal integer from standard input, terminated by the Enter key.

ReadInt - Reads a 32-bit signed decimal integer from standard input, terminated by the Enter key.

ReadString - Reads a string from standard input, terminated by the Enter key.
Library Procedures - Overview (3 of 3)

SetTextColor - Sets the foreground and background colors of all subsequent text output to the console.

WaitMsg - Displays message, waits for Enter key to be pressed.

WriteBin - Writes an unsigned 32-bit integer to standard output in ASCII binary format.

WriteChar - Writes a single character to standard output.

WriteDec - Writes an unsigned 32-bit integer to standard output in decimal format.

WriteHex - Writes an unsigned 32-bit integer to standard output in hexadecimal format.

WriteInt - Writes a signed 32-bit integer to standard output in decimal format.

WriteString - Writes a null-terminated string to standard output.

Example 1

Clear the screen, delay the program for 500 milliseconds, and dump the registers and flags.

```
.code
  call Clrscr
  mov eax, 500
  call Delay
  call DumpRegs
```

Sample output:

```
EAX=00000613 EBX=00000000 ECX=000000FF EDX=00000000
ESI=00000000 EDI=00000100 EBP=0000091E ESP=000000F6
EIP=00401026 EFL=000000286 CF=0 SF=1 ZF=0 OF=0
```
Example 2

Display a null-terminated string and move the cursor to the beginning of the next screen line.

```
.data
str1 BYTE "Assembly language is easy!",0

.code
  mov edx,OFFSET str1
  call WriteString
  call Crlf
```