LAB - 6

Lab Manual - Subroutines

Important Note: Make stack on your paper and then read, write, push and pop data accordingly. If you are using registers in a function make sure you save and restore their state properly. Do not use any global variables.

Stack Implementation:

Stack operations are facilitated by three registers:

1. The stack segment (SS) register. Stacks are implemented in memory. A system may have a number of stacks that is limited only by the maximum number of segments. A stack may be up to 4 gigabytes long, the maximum length of a segment. One stack is directly addressable at the address located by SS. This is the current stack, often referred to simply as "the" stack. SS is used automatically by the processor for all stack operations.
2. The stack pointer (SP) register points to the top of the push-down stack. It is referenced implicitly by PUSH and POP operations, subroutine calls and returns, and interrupt operations. When an item is pushed onto the stack, the processor decrements SP, then writes the item at the new position. When an item is popped off the stack, the processor copies it from the top position, then increments ESP. In other words, the stack grows down in memory toward lesser addresses.
3. The stack-frame base pointer (BP) register is the best choice of register for accessing data structures, variables and dynamically allocated work space within the stack. BP is often used to access elements on the stack relative to a fixed point on the stack rather than relative to the current top of stack. It typically identifies the base address of the current stack frame established for the current procedure. When BP is used as the base register in an offset calculation, the offset is calculated automatically in the current stack segment (i.e., the segment currently selected by SS). Because SS does not have to be explicitly specified, instruction encoding in such cases is more efficient. BP can also be used to index into segments addressable via other segment registers. [2 and 3]

Following is an example of the working of the stack:

Operation	Explanation

`MOV AX,1234H`
`PUSH AX`
`MOV AX, 5678H`
`POP AX`
	First we load 1234h into AX, and then we push that value to the stack. Now we move 5678h into AX Then we pop the AX register. We retrieve the pushed value from the stack. So AX contains 1234h again.

`MOV AX, 1234H`
`MOV BX, 5678H`
`PUSH AX`
`POP BX`
	The final values will be: AX=1234h BX=1234h. We pushed the AX to the stack and we popped that value in BX.

The CALL and RET Instructions

The call and ret instructions handle subroutine calls and returns. There are five different call instructions and six different forms of the return instruction:

CALL DISP16 ;direct intrasegment, 16 bit relative.
CALL ADRS32 ;direct intersegment, 32 bit segmented address.
CALL MEM16 ;indirect intrasegment, 16 bit memory pointer.
CALL REG16 ;indirect intrasegment, 16 bit register pointer.
CALL MEM32 ;indirect intersegment, 32 bit memory pointer.
RET ;near or far return
RETN ;near return
RETF ;far return
RET DISP ;near or far return and pop
RETN DISP ;near return and pop
RETF DISP ;far return and pop

The call instructions take the same forms as the jmp instructions except there is no short (two byte) intra-segment call.
The far call instruction does the following:

· It pushes the CS register onto the stack.
· It pushes the 16 bit offset of the next instruction following the call onto the stack.
· It copies the 32 bit effective address into the CS:IP registers. Since the call instruction allows the same addressing modes as jmp, call can obtain the target address using a relative, memory, or register addressing mode.
· Execution continues at the first instruction of the subroutine. This first instruction is the opcode at the target address computed in the previous step.

The near call instruction does the following:

· It pushes the 16 bit offset of the next instruction following the call onto the stack.
· It copies the 16 bit effective address into the ip register. Since the call instruction allows the same addressing modes as jmp, call can obtain the target address using a relative, memory, or register addressing mode.

· Execution continues at the first instruction of the subroutine. This first instruction is the opcode at the target address computed in the previous step.

The *call disp16* instruction uses relative addressing. You can compute the effective address of the target by adding this 16 bit displacement with the return address (like the relative jmp instructions, the displacement is the distance from the instruction following the call to the target address).

The *call adrs32* instruction uses the direct addressing mode. A 32 bit segmented address immediately follows the call opcode. This form of the call instruction copies that value directly into the *CS:IP* register pair. In many respects, this is equivalent to the immediate addressing mode since the value this instruction copies into the *CS:IP* register pair immediately follows the instruction.

Call mem16 uses the memory indirect addressing mode. Like the jmp instruction, this form of the call instruction fetches the word at the specified memory location and uses that word's value as the target address.

Call reg16 works just like the memory indirect call above, except it uses the 16 bit value in a register for the target address. This instruction is really the same instruction as the call mem16 instruction. Both forms specify their effective address using a modreg- r/m byte. For the call reg16 form, the mod bits contain 11b so the r/m field specifies a register rather than a memory addressing mode. Of course, this instruction also pushes the 16 bit offset of the next instruction onto the stack as the return address.

The *call mem32* instruction is a far indirect call. The memory address specified by this instruction must be a double word value. This form of the call instruction fetches the 32 bit segmented address at the computed effective address and copies this double word value into the cs:ip register pair. This instruction also copies the 32 bit segmented address of the next instruction onto the stack (it pushes the segment value first and the offset portion second). Like the call mem16 instruction, you can use any valid memory addressing mode with this instruction:

It is relatively easy to synthesize the call instruction using two or three other 80x86 instructions. You could create the equivalent of a near call using a push and a jmp instruction: push <offset of instruction after jmp>

jmp subroutine

A far call would be similar, you'd need to add a push CS instruction before the two instructions above to push a far return address on the stack.

The ***ret (return)*** instruction returns control to the caller of a subroutine. It does so by popping the return address off the stack and transferring control to the instruction at this return address. Intrasegment (near) returns pop a 16 bit return address off the stack into the ip register. An intersegment (far) return pops a 16 bit offset into the ip register and then a 16 bit segment value into the cs register. These instructions are effectively equal to the following:

retn: pop ip
retf: popd cs:ip

Clearly, you must match a near subroutine call with a near return and a far subroutine call with a corresponding far return. If you mix near calls with far returns or vice versa, you will leave the stack in an inconsistent state and you probably will not return to the proper instruction after the call. Of course, another important issue when using the CALL and RET instructions is that you must make sure your subroutine doesn't push something onto the stack and then fail to pop it off before trying to return to the caller. Stack problems are a major cause of errors in assembly language subroutines. [2]

The PUSH and POP Instructions

The 80x86 push and pop instructions manipulate data on the 80x86's hardware stack. Following are the varieties of the push and pop instructions:

PUSH REG16
POP REG16
PUSH SEGREG
POP SEGREG (EXCEPT CS)
PUSH MEMORY
POP MEMORY
PUSHF
POPF

The first two instructions push and pop a 16 bit general purpose register. This is a compact (one byte) version designed specifically for registers.

The third pair of push/pop instructions let you push or pop an 80x86 segment register. Note that the instructions that push FS and GS are longer than those that push CS, DS, ES, and SS. You can only push the CS register (popping the CS register would create some interesting program flow control problems).

The fourth pair of push/pop instructions allow you to push or pop the contents of a memory location. On the 80286 and earlier, this must be a 16 bit value.

The pushf and popf instructions allow you to push/pop the processor status register (the flags).

Following is an algorithmic description of each instruction:

push instructions (16 bits):
SP := SP - 2
[SS:SP] := 16 bit operand (store result at location SS:SP.)

pop instructions (16 bits):
16-bit operand := [SS:SP]
SP := SP + 2

Notice three things about the 80x86 hardware stack. First, it is always in the stack segment (wherever SS points). Second, the stack grows down in memory. That is, as you push values onto the stack the CPU stores them into successively lower memory locations. Finally, the 80x86 hardware stack pointer (SS:SP) always contains the address of the value on the top of the stack (the last value pushed on the stack).

You can use the 80x86 hardware stack for temporarily saving registers and variables, passing parameters to a procedure, allocating storage for local variables, and other uses. The push and pop instructions are extremely valuable for manipulating these items on the stack. *(Extracted from [1], [2] and [4])*

Problems:

Activity 1: Write the sub-routine to calculate factorial. The sub-routine should take as parameter the number to calculate the factorial of and returns factorial through stack. Write a multiplication subroutine instead of using MUL instruction.

Activity 2: Write a program that calculate the following series:

$$ans = \sum_{n=1}^{l} r^n$$

Where r and l are variables to be passed through stack. For the calculation of this series, you are required to make multiple subroutines. Parameter passing from one subroutine to the other should be via stack. The final answer should be returned from stack as well.

For example: If r =2 and l = 8 then final answer is 1FE.

For this question, use MUL instruction.

Activity 3: Consider a return value, 4 parameters, 2 local variables (of subroutine) and 4 register copies are placed on stack.

(i) How can we access all parameters in function/subroutine?
(ii) How can we place return value of function?
(iii) How can we access local stack variables of the function/sub routine?
(iv) How to empty stack before & after leaving subroutine?
(v) How can we pass parameters and retrieve return value in Caller?