Name : Tazmeen Afroz Roll No: 22P-9252 Section : BAI-4A

Manual for Understanding Subroutines, Program Flow, and Stack Management

Overview

In this manual, we will discuss subroutines in assembly language, focusing on the program flow, the use of the CALL and RET instructions, and the role of the system stack. This knowledge is critical for writing reusable and maintainable code in assembly language.

5.1. Program Flow

Understanding program flow in assembly language involves comprehending how the CPU executes instructions sequentially unless altered by jump or call instructions.

Key Concepts

1. **Sequential Execution**: Instructions are executed one after another.
2. **Permanent Diversion**: Instructions like JMP permanently alter the flow.
3. **Temporary Diversion**: Instructions like CALL temporarily alter the flow and are designed to return to the point of diversion.

CALL and RET Instructions

- **CALL**: Temporarily diverts execution to a subroutine. It saves the return address (the address of the next instruction after the CALL) onto the stack.
- **RET**: Returns control to the instruction following the CALL by popping the saved return address off the stack.

Parameters

Parameters are passed to subroutines to make them reusable for different data. Common methods for passing parameters include using registers or the stack.

Example Subroutine: Bubble Sort

Let's examine a bubble sort subroutine that sorts an array of integers.

```
[org 0x100]

jmp start

data:   dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0
swap:   db  0

bubblesort:
    dec  cx
    shl  cx, 1                  ; multiply CX by 2 for word array
mainloop:
    mov  si, 0                  ; array index
    mov  byte [swap], 0         ; reset swap flag
innerloop:
    mov  ax, [bx + si]
    cmp  ax, [bx + si + 2]
    jbe  noswap

    ; Swap elements
    mov  dx, [bx + si + 2]
    mov  [bx + si], dx
    mov  [bx + si + 2], ax
    mov  byte [swap], 1

noswap:
    add  si, 2
    cmp  si, cx
    jne  innerloop

    cmp  byte [swap], 1
    je   mainloop

    ret     ; return to caller

start:
    mov  bx, data
    mov  cx, 10
    call bubblesort     ; sort first array

    ; More code or termination
    mov ax, 0x4c00
    int 0x21            ; terminate program
```

Explanation

1. **Initialization**:
   - jmp start: Jump to the start of the main program.
   - data: Define an array of 10 integers.
2. **Main Program**:
   - mov bx, data: Load the base address of the data array into BX.
   - mov cx, 10: Load the number of elements into CX.
   - call bubblesort: Call the bubblesort subroutine to sort the array.

3. **Bubble Sort Subroutine**:
   - dec cx: Decrement CX to account for zero-based index.
   - shl cx, 1: Multiply CX by 2 because we are working with 16-bit words.
   - mainloop and innerloop: Implement the bubble sort algorithm.
   - mov ax, [bx + si] and cmp ax, [bx + si + 2]: Compare adjacent elements.
   - If elements are out of order, swap them.
   - ret: Return to the instruction following the call in the main program.

Stack Usage

When CALL bubblesort is executed:

1. The CPU pushes the return address onto the stack.
2. Execution jumps to the bubblesort label.
3. RET pops the return address off the stack and resumes execution from that address.

Multiple Arrays Example

```
[org 0x100]

jmp start

data1: dw 60, 55, 45, 50
data2: dw 328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98

swap: db 0

bubblesort:
    dec cx
    shl cx, 1
mainloop:
    mov si, 0
    mov byte [swap], 0
innerloop:
    mov ax, [bx + si]
    cmp ax, [bx + si + 2]
    jbe noswap

    mov dx, [bx + si + 2]
    mov [bx + si], dx
    mov [bx + si + 2], ax
    mov byte [swap], 1

noswap:
    add si, 2
    cmp si, cx
    jne innerloop

    cmp byte [swap], 1
    je mainloop

    ret

start:
    mov bx, data1
    mov cx, 4
    call bubblesort

    mov bx, data2
    mov cx, 10
    call bubblesort

    mov ax, 0x4c00
    int 0x21
```

Explanation

1. **Two Arrays**: Define data1 and data2 arrays.
2. **Main Program**:
   - Sort data1 with 4 elements.
   - Sort data2 with 10 elements.
3. **Subroutine Call**:
   - Each call to bubblesort handles different arrays and sizes.
   - The stack ensures the return address is preserved and execution resumes correctly after sorting each array.

Stack Mechanics in CALL and RET

1. **CALL Instruction**:
   - Pushes the return address onto the stack.
   - Changes the Instruction Pointer (IP) to the subroutine address.
2. **RET Instruction**:
   - Pops the return address off the stack.
   - Restores the IP to resume execution after the CALL.

Stack Pointer (SP) and Stack Segment (SS)

- **SP Register**: Indicates the top of the stack.
- **SS Combination**: Provides the physical address of the stack.

Stack Behavior in 8088 Processor

- **Decrementing Stack**: SP is decremented by 2 for each push operation.
- **Word-Sized Elements**: Only words (not single bytes) can be pushed/popped.

Stack Use in Function Calls

- **CALL and RET**:
  - **CALL**: Saves the instruction pointer (IP) on the stack and jumps to the subroutine.
  - **RET**: Restores the IP from the stack to return to the calling function.
- **RET n**: Increments SP by an additional value after returning.

Example of Stack Operation

- Initial SP: 2000
- Push 017B: SP becomes 1998, 017B is stored at address 1998.
- RET: Restores IP from stack and increments SP back to 2000.

Stack in Nested Subroutine Calls

Nested calls result in multiple push operations, storing return addresses and other data, which are managed by corresponding pop operations to restore the state.

Saving and Storing Registers:

Code

```
[org 0x0100]
jmp start

data: dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0
data2: dw 328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98
dw 888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5
swapflag: db 0

swap:
    push ax                  ; save old value of ax
    mov ax, [bx+si]          ; load first number in ax
    xchg ax, [bx+si+2]       ; exchange with second number
    mov [bx+si], ax          ; store second number in first
    pop ax                   ; restore old value of ax
    ret                      ; go back to where we came from

bubblesort:
    push ax                  ; save old value of ax
    push cx                  ; save old value of cx
    push si                  ; save old value of si
    dec cx                   ; last element not compared
    shl cx, 1                ; turn into byte count
mainloop:
    mov si, 0                ; initialize array index to zero
    mov byte [swapflag], 0   ; reset swap flag to no swaps
innerloop:
    mov ax, [bx+si]          ; load number in ax
    cmp ax, [bx+si+2]        ; compare with next number
    jbe noswap               ; no swap if already in order
    call swap                ; swaps two elements
    mov byte [swapflag], 1   ; flag that a swap has been done
noswap:
    add si, 2                ; advance si to next index
    cmp si, cx               ; are we at last index
    jne innerloop            ; if not compare next two

    cmp byte [swapflag], 1   ; check if a swap has been done
    je mainloop              ; if yes make another pass
    pop si                   ; restore old value of si
    pop cx                   ; restore old value of cx
    pop ax                   ; restore old value of ax
    ret                      ; go back to where we came from

start:
    mov bx, data             ; send start of array in bx
    mov cx, 10               ; send count of elements in cx
    call bubblesort          ; call our subroutine
    mov bx, data2            ; send start of array in bx
    mov cx, 20               ; send count of elements in cx
    call bubblesort          ; call our subroutine again
    mov ax, 0x4c00           ; terminate program
    int 0x21                 ; interrupt to DOS to terminate program
```

Explanation of Changes

The main changes from the previous code involve the addition of PUSH and POP instructions to save and restore register values. This ensures that the registers used within the subroutines do not interfere with the rest of the program.

1. **Saving and Restoring Registers in swap Subroutine:**
    - **Previous Code:** Did not save the state of AX before modifying it.
    - **New Code:**
        - push ax: Saves the current value of AX on the stack.
        - mov ax, [bx+si]: Loads the first number into AX.
        - xchg ax, [bx+si+2]: Exchanges AX with the second number.
        - mov [bx+si], ax: Stores the exchanged value back to memory.
        - pop ax: Restores the previous value of AX from the stack.
    - **Benefit:** Ensures that the original value of AX is preserved and restored after the subroutine finishes execution.
2. **Saving and Restoring Registers in bubblesort Subroutine:**
    - **Previous Code:** Did not save the states of AX, CX, and SI, which could interfere with the main program.
    - **New Code:**
        - push ax: Saves the current value of AX on the stack.
        - push cx: Saves the current value of CX on the stack.
        - push si: Saves the current value of SI on the stack.
        - The main sorting loop and inner sorting loop remain the same.
        - After completing the sort:
            - pop si: Restores the previous value of SI from the stack.
            - pop cx: Restores the previous value of CX from the stack.
            - pop ax: Restores the previous value of AX from the stack.
    - **Benefit:** Ensures that the original values of AX, CX, and SI are preserved and restored after the subroutine finishes execution.
3. **Ensuring Correct Order for PUSH and POP:**
    - The order of PUSH operations is push ax, push cx, push si.
    - The order of POP operations is pop si, pop cx, pop ax.
    - This maintains the Last In First Out (LIFO) order of the stack, ensuring that the registers are correctly restored to their original values.

Bubble sort subroutine taking parameters from stack

```
[org 0x0100]
jmp start

data: dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0
data2: dw 328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98
       dw 888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5
swapflag: db 0
bubblesort:
    push bp                ; save old value of bp
    mov bp, sp             ; make bp our reference point
    push ax                ; save old value of ax
    push bx                ; save old value of bx
    push cx                ; save old value of cx
    push si                ; save old value of si

    mov bx, [bp+6]         ; load start of array in bx
    mov cx, [bp+4]         ; load count of elements in cx
    dec cx                 ; last element not compared
    shl cx, 1              ; turn into byte count

mainloop:
    mov si, 0              ; initialize array index to zero
    mov byte [swapflag], 0 ; reset swap flag to no swaps

innerloop:
    mov ax, [bx+si]        ; load number in ax
    cmp ax, [bx+si+2]      ; compare with next number
    jbe noswap             ; no swap if already in order
    xchg ax, [bx+si+2]     ; exchange ax with second number
    mov [bx+si], ax        ; store second number in first
    mov byte [swapflag], 1 ; flag that a swap has been done

noswap:
    add si, 2              ; advance si to next index
    cmp si, cx             ; are we at last index
    jne innerloop          ; if not compare next

    cmp byte [swapflag], 1 ; check if a swap has been done
    je mainloop            ; if yes make another pass

    pop si                 ; restore old value of si
    pop cx                 ; restore old value of cx
    pop bx                 ; restore old value of bx
    pop ax                 ; restore old value of ax
    pop bp                 ; restore old value of bp
    ret 4                  ; go back and remove two params

start:
    mov ax, data
    push ax                ; place start of array on stack
    mov ax, 10
    push ax                ; place element count on stack
    call bubblesort        ; call our subroutine
    mov ax, data2
    push ax                ; place start of array on stack
    mov ax, 20
    push ax                ; place element count on stack
    call bubblesort        ; call our subroutine again
    mov ax, 0x4c00         ; terminate program
    int 0x21
```

Explanation:

1. **Subroutine bubblesort**:

    - **Entry**: The bp register is pushed to save its previous value, and bp is then set to the current sp value to create a stack frame.
    - **Registers Preservation**: The ax, bx, cx, and si registers are pushed onto the stack to preserve their values.
    - **Parameter Loading**: The start address of the array and the count of elements are loaded from the stack into bx and cx respectively.
    - **Bubble Sort Algorithm**: The bubble sort is implemented with nested loops. The inner loop compares and swaps adjacent elements if they are out of order.
    - **Swap Flag**: A flag is used to track if any swaps were made in an iteration. If no swaps were made, the sorting is complete.
    - **Exit**: The preserved registers are popped back, restoring their values. The bp is restored, and the ret 4 instruction cleans up the stack by removing the parameters.

Bubble sort subroutine using a local variable:

```
[org 0x0100]
jmp start
data: dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0
data2: dw 328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98
      dw 888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5
bubblesort:
    push bp              ; save old value of bp
    mov bp, sp           ; make bp our reference point
    sub sp, 2            ; make two byte space on stack for swap flag
    push ax              ; save old value of ax
    push bx              ; save old value of bx
    push cx              ; save old value of cx
    push si              ; save old value of si
    mov bx, [bp+6]       ; load start of array in bx
    mov cx, [bp+4]       ; load count of elements in cx
    dec cx               ; last element not compared
    shl cx, 1            ; turn into byte count

mainloop:
    mov si, 0            ; initialize array index to zero
    mov word [bp-2], 0   ; reset swap flag to no swaps
innerloop:
    mov ax, [bx+si]      ; load number in ax
    cmp ax, [bx+si+2]    ; compare with next number
    jbe noswap           ; no swap if already in order
    xchg ax, [bx+si+2]   ; exchange ax with second number
    mov [bx+si], ax      ; store second number in first
    mov word [bp-2], 1   ; flag that a swap has been done
noswap:
    add si, 2            ; advance si to next index
    cmp si, cx           ; are we at last index
    jne innerloop        ; if not compare next two

    cmp word [bp-2], 1   ; check if a swap has been done
    je mainloop          ; if yes make another pass

    pop si               ; restore old value of si
    pop cx               ; restore old value of cx
    pop bx               ; restore old value of bx
    pop ax               ; restore old value of ax
    mov sp, bp           ; remove space created on stack
    pop bp               ; restore old value of bp
    ret 4                ; go back and remove two params
start:
    mov ax, data
    push ax              ; place start of array on stack
    mov ax, 10
    push ax              ; place element count on stack
    call bubblesort      ; call our subroutine
    mov ax, data2
    push ax              ; place start of array on stack
    mov ax, 20
    push ax              ; place element count on stack
    call bubblesort      ; call our subroutine again
    mov ax, 0x4c00       ; terminate program
    int 0x21
```

Changes from Previous Code and Explanation

1. **Local Variable Creation:**
   - The line sub sp, 2 was added right after setting bp to sp. This creates a space for a local variable (swap flag) on the stack.
   - The local variable (swap flag) is accessed using bp-2.
2. **Using Local Variable:**
   - In the mainloop, the swap flag is set to 0 with mov word [bp-2], 0.
   - During the innerloop, the swap flag is set to 1 when a swap is performed with mov word [bp-2], 1.
3. **Stack Cleanup:**
   - Before exiting the subroutine, the line mov sp, bp is used to clean up the local variable space on the stack. This restores sp to its original value before the local variable space was allocated.
   - This approach avoids the need to remember the exact amount of space allocated for local variables, making the code cleaner and more maintainable.