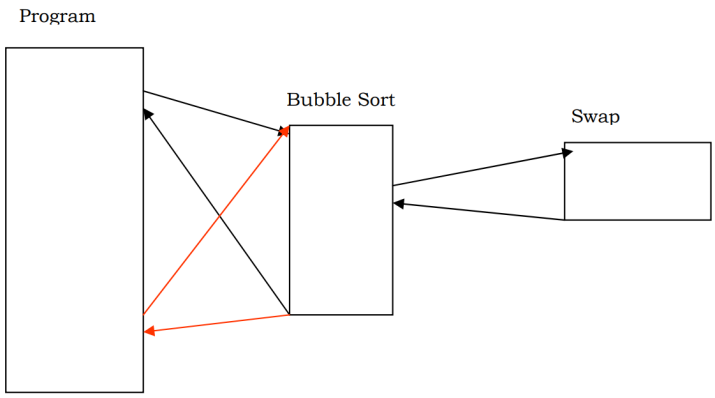
Chapter 5

Subroutines

Definition:

A subroutine is a block of code within an assembly program that performs a specific task. It's like a mini-program within your main program.

Subroutines are essential components of assembly language programming, allowing you to create reusable code. Subroutines promote modularity by breaking down complex tasks into smaller, manageable parts. Each subroutine focuses on a single task.



Structure:

A subroutine has a defined structure:

- **Label:** It starts with a label that serves as the subroutine's name.
- **Code:** The actual code that performs the task.
- **Return:** Typically, a subroutine ends with a return instruction, indicating where the program should continue executing after the subroutine call.

Calling a Subroutine:

- The `CALL` instruction is used to transfer control to a subroutine, making it a callable procedure.
- When you call a subroutine using `CALL`, it pushes the return address onto the stack. The return address is typically the address of the instruction immediately following the `CALL` instruction.
- This is done to ensure that when the subroutine execution is complete (via a `RET` instruction), the program can return to the correct location in the main code.

RET Instruction:

- The `RET` instruction is used to return control from a subroutine back to the calling code.

- When `RET` is executed, it pops the return address from the stack and loads it into the IP register.
- This action effectively resumes execution at the instruction immediately following the original `CALL` instruction.

Stack:

- The stack is a region of memory used to store data temporarily. It operates on a Last In, First Out (LIFO) basis, meaning the last item pushed onto the stack is the first to be popped off.
- In x86 assembly, the stack is typically managed using the SP (Stack Pointer) register.
- The SP points to the top of the stack, and when you push data onto the stack, SP is decremented (from high address to low address) to allocate space for the pushed value. Conversely, when you pop data from the stack, SP is incremented.
- The stack is crucial for maintaining the program's state during subroutine calls, as it stores return addresses and sometimes other data.

```
[org 0x0100]
jmp start

; Define the data arrays
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:
    ; Swap subroutine
    mov ax, [bx+si]      ; Load the first number into ax
    xchg ax, [bx+si+2]   ; Exchange it with the second number
    mov [bx+si], ax      ; Store the second number in the first
    ret                  ; Return to the caller

bubblesort:
    dec cx               ; Last element not compared
    shl cx, 1            ; Convert count to 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 a number into ax
    cmp ax, [bx+si+2]    ; Compare with the next number
```

```
    jbe noswap           ; Jump if already in order

    call swap            ; Call the swap subroutine to swap two elements
    mov byte [swapflag], 1 ; Flag that a swap has been done

noswap:
    add si, 2            ; Advance si to the next index
    cmp si, cx           ; Check if we are at the last index
    jne innerloop        ; If not, compare the next two elements

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

    ret                  ; Return to the caller

start:
    mov bx, data         ; Send the start of the first array in bx
    mov cx, 10           ; Send the count of elements in cx
    call bubblesort      ; Call the bubblesort subroutine

    mov bx, data2        ; Send the start of the second array in bx
    mov cx, 20           ; Send the count of elements in cx
    call bubblesort      ; Call the bubblesort subroutine again

    mov ax, 0x4c00       ; Terminate program
    int 0x21
```

Saving and Restoring Registers:

When using subroutines, it's important to manage and preserve registers to ensure that the main program's state isn't disrupted. Here's how registers can be saved and restored within subroutines:

Need for Preservation: Subroutines often use registers for their operations. To ensure that the calling code's registers are not unintentionally modified, it's essential to save and restore their values. Assembly provides `PUSH` and `POP` instructions for this purpose.

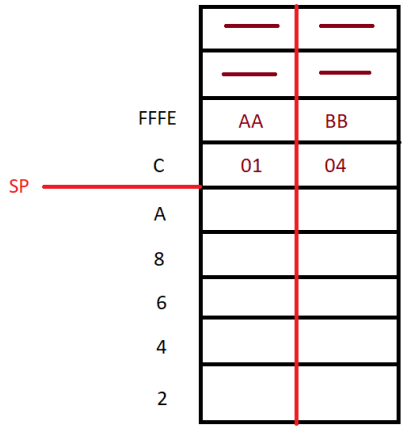
PUSH Instruction:

- `PUSH` decreases the stack pointer (`SP`) by two bytes.
- It transfers a word (usually a register's value) from the source operand to the memory location pointed to by `SP`. This effectively pushes the value onto the stack.
- Typically used to save the values of registers before they are modified within a subroutine.
- For example: `PUSH AX` pushes the value of the AX register onto the stack.
- The operation of PUSH is shown below.
  SP ← SP – 2

[SP] ← AX

POP Instruction:

- `POP` transfers the word from the memory location pointed to by `SP` to the destination operand (usually a register).
- After the transfer, it increments `SP` by two bytes (one word) to point to the new top of the stack.
- Typically used to restore the values of registers to their previous state after a subroutine has finished using them.
- For example: `POP AX` pops the top value from the stack into the AX register.
- The operation of “pop ax” is shown below.
  AX ← [SP]
  SP ← SP + 2



Order Matters: When pushing and popping multiple registers, the order is crucial. Registers must be popped in the reverse order in which they were pushed. This follows the Last In First Out (LIFO) behavior of the stack.

Example: In the provided assembly code example, the `PUSH` and `POP` instructions are used to save and restore the values of registers AX, CX, and SI within the `bubblesort` subroutine.

```
[org 0x0100]
jmp start

; Define the data arrays
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 the first number into ax
    xchg ax, [bx+si+2] ; Exchange it with the second number
    mov [bx+si], ax    ; Store the second number in the first
    pop ax             ; Restore old value of ax
    ret                ; Return to the caller

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          ; Convert count to 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 a number into ax
    cmp ax, [bx+si+2]  ; Compare with the next number

    jbe noswap         ; Jump if already in order

    call swap          ; Call the swap subroutine to swap two elements
    mov byte [swapflag], 1 ; Flag that a swap has been done

noswap:
    add si, 2          ; Advance si to the next index
    cmp si, cx         ; Check if we are at the last index
    jne innerloop      ; If not, compare the next two elements

    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                ; Return to the caller

start:
    mov bx, data       ; Send the start of the first array in bx
    mov cx, 10         ; Send the count of elements in cx
    call bubblesort    ; Call the bubblesort subroutine
```

```
    mov bx, data2      ; Send the start of the second array in bx
    mov cx, 20         ; Send count of elements in cx
    call bubblesort    ; Call the bubblesort subroutine again

    mov ax, 0x4c00     ; Terminate program
    int 0x21
```

Parameter Passing to Subroutines:

In assembly language programming, parameters can be passed to subroutines through the stack. Let's explore how parameter passing through the stack works:

Parameter Limitations with Registers: In assembly language, the number of registers is limited, and passing parameters via registers can be restrictive. Typically, only a limited number of parameters can be passed when registers are used, and this can become a problem when subroutines themselves use registers for their operations.

Stack as an Alternative: The stack provides an alternate way to pass parameters. When data is pushed onto the stack, it remains there, making it accessible across function calls. For example, you can push parameters onto the stack and call a subroutine, which can then access these parameters directly from the stack.

Order of Parameters on the Stack: When parameters are pushed onto the stack before calling a subroutine, they remain on the stack in the order they were pushed. This means that the first parameter pushed onto the stack is at a higher memory address than the second parameter.

Accessing Parameters from the Stack: To access the parameters from the stack, you might initially think of popping them off the stack. However, you must remember that the return address of the subroutine is also on the stack. Attempting to pop parameters directly would disrupt the stack frame.

Using the Base Pointer (BP): The solution to this problem involves using the base pointer (BP) register. In x86 assembly, BP is often associated with the stack segment (SS) and is used as a reference point for accessing parameters on the stack without modifying the stack pointer (SP).

- **Copying SP to BP:** When a subroutine is called, you can copy the current value of SP to BP. This effectively "freezes" the stack at that moment, and BP becomes your reference point for accessing parameters.
- **Accessing Parameters with BP:** With BP set as a reference point, you can access parameters on the stack using offsets relative to BP. For example, if we are passing two parameters and BP points to the return address which is located at [BP + 2], the first parameter might be at [BP + 6] and the second parameter at [BP + 4].
- **Standard Parameter Access in Subroutines:** To access parameters in subroutines, the standard practice is to use the following instructions at the beginning of the subroutine:
    - `push bp`: Save the old value of BP on the stack.
    - `mov bp, sp`: Set BP as a reference point for parameter access.
- **Example:**

```
[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 the second number
    mov [bx+si], ax    ; store the second number in the first
    mov byte [swapflag], 1 ; flag that a swap has been done

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

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

    pop si            ; restore the old value of si
    pop cx            ; restore the old value of cx
    pop bx            ; restore the old value of bx
    pop ax            ; restore the old value of ax
```

```
    pop bp            ; restore the old value of bp
    ret 4             ; go back and remove two params

start:
    mov ax, data
    push ax           ; place the start of the array on the stack
    mov ax, 10
    push ax           ; place the element count on the stack
    call bubblesort   ; call our subroutine

    mov ax, data2
    push ax           ; place the start of the array on the stack
    mov ax, 20
    push ax           ; place the element count on the stack
    call bubblesort   ; call our subroutine again

    mov ax, 0x4c00    ; terminate program
    int 0x21
```

- **Stack Clearing:** After the subroutine finishes execution, it's essential to clear any parameters from the stack. This can be done by either the caller or the callee. In most conventions, stack clearing is done by the callee to ensure proper stack management and prevent stack overflow. This is typically done using `ret n`, where **'n'** is the number of bytes to clear.

Local Variables:

Creating and managing local variables within a subroutine using the stack is a common practice in assembly language programming. These local variables are temporary and only relevant to the execution of the subroutine. Here's how we can create and use local variables within a subroutine:

1. **Creating Local Variables:**
    - To create local variables, you need to allocate space for them on the stack. This is typically done by decrementing the stack pointer (SP) to create a gap for the local variables.
    - The gap size depends on the number and size of local variables you need. You should ensure that the decrement is by an even number (a multiple of 2) since stack operations work with words.
    - The convenient position to create this gap is immediately after saving the value of the base pointer (BP) because it allows you to use the same BP for accessing both parameters and local variables.
    - Here's an example of how to create a gap for a single word-sized local variable:

      ```
      push bp       ; Save old BP
      mov bp, sp    ; Set BP as the new reference point
      sub sp, 2     ; Create a gap of 2 bytes (1 word) for the local variable
      ```

2. **Accessing Local Variables:**

- Once you've created the gap, you can access local variables using negative offsets from the base pointer (BP). Negative offsets move "down" the stack to access the local variables.
- For example, if you have a local variable at the top of the gap (immediately below BP), you can access it using `mov ax, [bp-2]` to load its value into the AX register.

Here's how you can modify the bubble sort subroutine to use a local variable for the swap flag:

```
[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 the stack
  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 the second number
  mov [bx+si], ax    ; Store the second number in the first
  mov word [bp-2], 1 ; Flag that a swap has been done

noswap:
  add si, 2          ; Advance si to the next index
  cmp si, cx         ; Are we at the last index?
```

```
  jne innerloop       ; If not, compare the next two

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

  pop si              ; Restore the old value of si
  pop cx              ; Restore the old value of cx
  pop bx              ; Restore the old value of bx
  pop ax              ; Restore the old value of ax
  mov sp, bp          ; Remove the space created on the stack
  pop bp              ; Restore the old value of bp
  ret 4               ; Go back and remove two parameters

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

In this modified code, the `swapflag` is now a local variable stored in the stack gap created by subtracting 2 from the stack pointer (SP). The bubble sort subroutine accesses this local variable using `[bp-2]`. This way, the swap flag is specific to each invocation of the subroutine and doesn't interfere with other parts of the program.