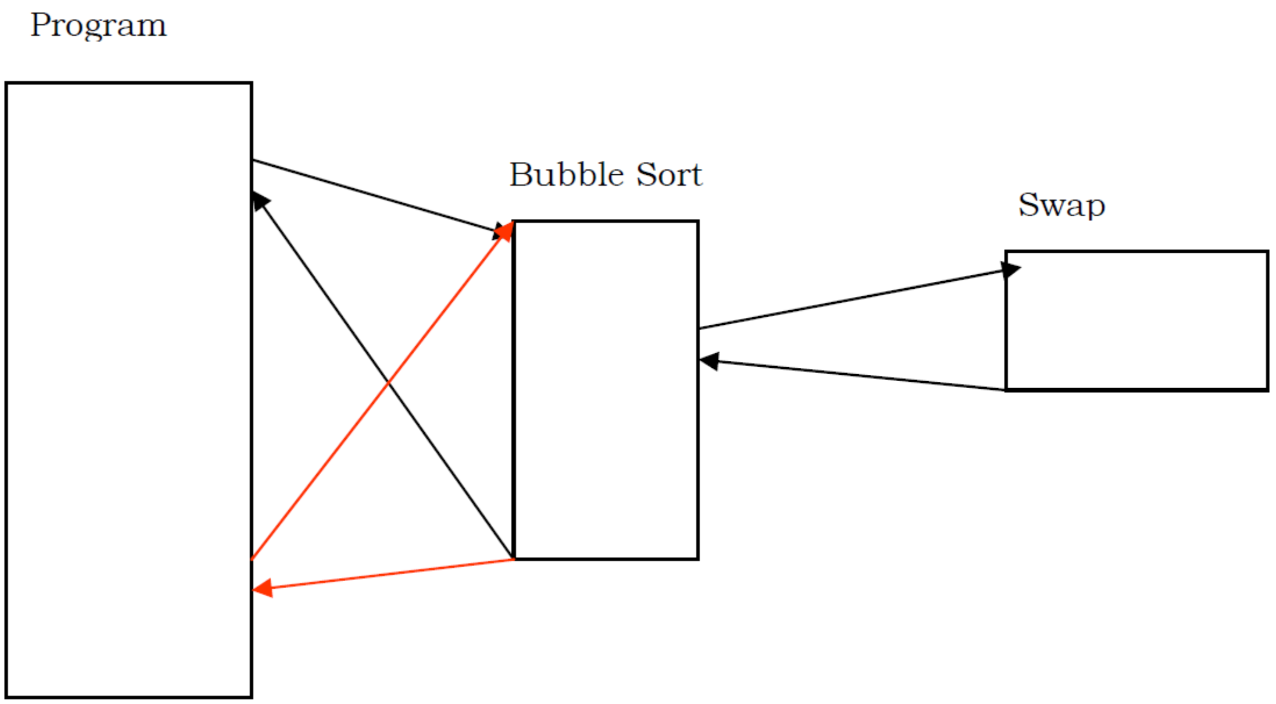
Subroutines

and working with Stack

Outline

- Introduction to subroutines
- Introduction to Stacks
- Saving and restoring registers

Introduction to subroutines



Implementing Subroutine

Use CALL and RET instructions

CALL

- Like JMP, but with additional functionality of saving the return address

RET

- Branch to the return address saved by the last CALL instruction

Example 5.1

Example 5.1

```
; bubble sort algorithm as a subroutine
[org 0x0100]
              jmp start

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

bubblesort:   dec  cx                    ; last element not compared
              shl  cx, 1                 ; turn into byte count

mainloop:     mov  si, 0                 ; initialize array index to zero
              mov  byte [swap], 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

              mov  dx, [bx+si+2]         ; load second element in dx
              mov  [bx+si], dx           ; store first number in second
              mov  [bx+si+2], ax         ; store second number in first
              mov  byte [swap], 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 [swap], 1        ; check if a swap has been done
              je   mainloop              ; if yes make another pass

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

Example 5.1

08-09	The routine has received the count of elements in CX. Since it makes one less comparison than the number of elements it decrements it. Then it multiplies it by two since this a word array and each element takes two bytes. Left shifting has been used to multiply by two.
14	Base+index+offset addressing has been used. BX holds the start of array, SI the offset into it and an offset of 2 when the next element is to be read. BX can be directly changed but then a separate counter would be needed, as SI is directly compared with CX in our case.
32-37	The code starting from the start label is our main program analogous to the main in the C language. BX and CX hold our parameters for the bubblesort subroutine and the CALL is made to invoke the subroutine.

Example 5.2

Example 5.2

```
; bubble sort subroutine called twice
[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
swap:           db   0

bubblesort:     dec  cx                     ; last element not compared
                shl  cx, 1                  ; turn into byte count

mainloop:       mov  si, 0                  ; initialize array index to zero
                mov  byte [swap], 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

                mov  dx, [bx+si+2]          ; load second element in dx
                mov  [bx+si], dx            ; store first number in second
                mov  [bx+si+2], ax          ; store second number in first
                mov  byte [swap], 1         ; flag that a swap has been done

noswap:         add  si, 2                  ; advance si to next index
```

Example 5.2

```
26                  cmp   si, cx              ; are we at last index
27                  jne   innerloop           ; if not compare next two
28
29                  cmp   byte [swap], 1      ; check if a swap has been done
30                  je    mainloop            ; if yes make another pass
31
32                  ret                       ; go back to where we came from
33
34  start:          mov   bx, data            ; send start of array in bx
35                  mov   cx, 10              ; send count of elements in cx
36                  call bubblesort           ; call our subroutine
37
38                  mov bx, data2             ; send start of array in bx
39                  mov cx, 20                ; send count of elements in cx
40                  call bubblesort           ; call our subroutine again
41
42                  mov   ax, 0x4c00          ; terminate program
43                  int   0x21
```

05-07	There are two different data arrays declared. One of 10 elements and the other of 20 elements. The second array is declared on two lines, where the second line is continuation of the first. No additional label is needed since they are situated consecutively in memory.
34-40	The other change is in the main where the bubblesort subroutine is called twice, once on the first array and once on the second.

Nested subroutines

Example 5.3

```
; bubble sort subroutine using swap subroutine
[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:           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
                ret                      ; go back to where we came from

bubblesort:     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
                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
```

Nested subroutines

```
                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
```

11	A new instruction XCHG has been introduced. The instruction swaps its source and its destination operands however at most one of the operands could be in memory, so the other has to be loaded in a register. The instruction has reduced the code size by one instruction.
13	The RET at the end of swap makes it a subroutine.

Template for creating subroutines

```
SubroutineName: <subroutine code here>
                <subroutine code here>
                <subroutine code here>
                ret

Start:          <prepare parameters>
                call SubroutineName
                mov  ax, 0x4C00  ;exit
                int  0x21
```

Issues in last example

a) Unclear where the return address is saved when CALLing a subroutine

b) Subroutine destroys the contents of some registers.
   - In this case CX, SI, AX, DX
   - Caller needs to make sure no important data is kept in these registers before CALL

Stacks

- Data structure that behaves in a first in last out (FILO) manner.
- There is only one way in and out of the container.
- An inserted element sits on top of all other elements. For removal, the one sitting at top is removed first.
- The operation of placing an element on top of the stack is called **pushing** the element
- The operation of removing an element from the top of the stack is called **popping** the element.
- We can read any element from the stack but cannot remove it without removing everything above it.

Stacks in iAPX88

- The address of top of the stack is held in the SP register.
- The physical address of the stack top is obtained by the SS:SP combination.
- The stack segment registers tells where the stack is located and the stack pointer marks the top of stack inside this segment.
- 8088 stack works on word sized elements.
  - Single bytes cannot be pushed or popped from the stack.

Incrementing or Decrementing Stack

- 8088 stack is decrementing (growing backwards).
  - Element push will decrement SP by two.
- Another possibility is an incrementing stack.
- A decrementing stack moves from higher addresses to lower addresses as elements are added in it while an incrementing stack moves from lower addresses to higher addresses as elements are added.
- There is no special reason or argument in favor of one or another, and more or less depends on the choice of the designers.

Decrementing Stack

- Memory is like a shelf numbered as zero at the top and the maximum at the bottom.
- If a decrementing stack starts at shelf 5, the first item is placed in shelf 5, the next item is placed in shelf 4, the next in shelf 3 and so on.
- The push operation copies its operand on the stack, while the pop operation makes a copy from the top of the stack into its operand.
- When an item is pushed on a decrementing stack, the top of the stack is first decremented and the element is then copied into this space.
- With a pop the element at the top of the stack is copied into the pop operand and the top of stack is incremented afterwards.

Stack Use Cases

- The basic use of the stack is to save things and recover from there when needed.
- For example we discussed the shortcoming in our last example that it destroyed the caller’s registers, and the callers are not supposed to remember which registers are destroyed by the thousand routines they use.
- Using the stack, the subroutine can save the caller’s value of the registers on the stack, and recover them from there before returning.
- Meanwhile the subroutine can freely use the registers.
- From the caller’s point of view if the registers contain the same value before and after the call, it doesn’t matter if the subroutine used them meanwhile.

CALL and RET operation

- Similarly during the CALL operation, the current value of the **instruction pointer** is automatically saved on the stack, and the destination of CALL is loaded in the instruction pointer.
- Execution therefore resumes from the destination of CALL.
- When the RET instruction is executed, it recovers the value of the instruction pointer from the stack.
- The next instruction executed is therefore the one following the CALL.
- Observe how playing with the instruction pointer affects the program flow.

Stack Demo

- Stack operations are of word size.
- Stack Pointer (SP) contains the address of **top of stack**
- When loading COM files, DOS initializes SP to 0xFFFE

(Stack Pointer) **SP** →

0147	Free Space
0148	Free Space
…	Free Space
FFEE	Free Space
FFEF	Free Space
FFF0	Free Space
FFF1	Free Space
FFF2	Free Space
FFF3	Free Space
FFF4	Free Space
FFF5	Free Space
FFF6	Free Space
FFF7	Free Space
FFF8	Free Space
FFF9	Free Space
FFFA	Free Space
FFFB	Free Space
FFFC	Free Space
FFFD	Free Space
FFFE	Free Space
FFFF	Free Space

Call/Ret Demo

```
        [org 0x0100]
100     jmp start

103     data: dw 60, 55, 45, ....
117     swap: db 0

        bubblesort:
118        dec   cx
119        shl   cx, 1

...        ;;; sorting code ;;;

146        ret

        start:
147        mov   bx, data
148        mov   cx, 10
14D        call  bubblesort

150        mov   ax, 0x4c00
153        int   0x21
```

	0147	Free Space
	0148	Free Space
	…	Free Space
	FFEE	Free Space
	FFEF	Free Space
	FFF0	Free Space
	FFF1	Free Space
	FFF2	Free Space
	FFF3	Free Space
	FFF4	Free Space
	FFF5	Free Space
	FFF6	Free Space
	FFF7	Free Space
	FFF8	Free Space
	FFF9	Free Space
	FFFA	Free Space
	FFFB	Free Space
	FFFC	Free Space
	FFFD	Free Space
SP →	FFFE	Free Space
	FFFF	Free Space

Call Demo

```
     [org 0x0100]
100  jmp start

103  data: dw 60, 55, 45, ....
117  swap: db 0

     bubblesort:
118     dec  cx
119     shl  cx, 1

...     ;;; sorting code ;;;

146     ret

     start:
147     mov  bx, data
148     mov  cx, 10
14D     call bubblesort

150     mov  ax, 0x4c00
153     int  0x21
```

IP

014D

	0147	Free Space
	0148	Free Space
	…	Free Space
	FFEE	Free Space
	FFEF	Free Space
	FFF0	Free Space
	FFF1	Free Space
	FFF2	Free Space
	FFF3	Free Space
	FFF4	Free Space
	FFF5	Free Space
	FFF6	Free Space
	FFF7	Free Space
	FFF8	Free Space
	FFF9	Free Space
	FFFA	Free Space
	FFFB	Free Space
	FFFC	Free Space
	FFFD	Free Space
SP →	FFFE	Free Space
	FFFF	Free Space

Call Demo

```
    [org 0x0100]
100 jmp start

103 data: dw 60, 55, 45, ....
117 swap: db 0

    bubblesort:
118     dec  cx
119     shl  cx, 1

...     ;;; sorting code ;;;

146     ret

    start:
147     mov  bx, data
148     mov  cx, 10
14D     call bubblesort

150     mov  ax, 0x4c00
153     int  0x21
```

IP

014D

	0147	Free Space
	0148	Free Space
	…	Free Space
	FFEE	Free Space
	FFEF	Free Space
	FFF0	Free Space
	FFF1	Free Space
	FFF2	Free Space
	FFF3	Free Space
	FFF4	Free Space
	FFF5	Free Space
	FFF6	Free Space
	FFF7	Free Space
	FFF8	Free Space
	FFF9	Free Space
	FFFA	Free Space
	FFFB	Free Space
SP →	FFFC	50
	FFFD	01
	FFFE	Free Space
	FFFF	Free Space

Call Demo

```
    [org 0x0100]
100 jmp start

103 data: dw 60, 55, 45, ....
117 swap: db 0

    bubblesort:
118     dec  cx
119     shl  cx, 1

...     ;;; sorting code ;;;

146     ret

    start:
147     mov  bx, data
148     mov  cx, 10
14D     call bubblesort

150     mov  ax, 0x4c00
153     int  0x21
```

IP

0118

SP

0147	Free Space
0148	Free Space
…	Free Space
FFEE	Free Space
FFEF	Free Space
FFF0	Free Space
FFF1	Free Space
FFF2	Free Space
FFF3	Free Space
FFF4	Free Space
FFF5	Free Space
FFF6	Free Space
FFF7	Free Space
FFF8	Free Space
FFF9	Free Space
FFFA	Free Space
FFFB	Free Space
FFFC	50
FFFD	01
FFFE	Free Space
FFFF	Free Space

Ret Demo

```
    [org 0x0100]
100 jmp start

103 data: dw 60, 55, 45, ....
117 swap: db 0

    bubblesort:
118     dec  cx
119     shl  cx, 1

...     ;;; sorting code ;;;

146     ret

    start:
147     mov  bx, data
148     mov  cx, 10
14D     call bubblesort

150     mov  ax, 0x4c00
153     int  0x21
```

IP

0146

SP

0147	**Free Space**
0148	**Free Space**
…	**Free Space**
FFEE	**Free Space**
FFEF	**Free Space**
FFF0	**Free Space**
FFF1	**Free Space**
FFF2	**Free Space**
FFF3	**Free Space**
FFF4	**Free Space**
FFF5	**Free Space**
FFF6	**Free Space**
FFF7	**Free Space**
FFF8	**Free Space**
FFF9	**Free Space**
FFFA	**Free Space**
FFFB	**Free Space**
FFFC	**50**
FFFD	**01**
FFFE	**Free Space**
FFFF	**Free Space**

Ret Demo

```
     [org 0x0100]
100  jmp start

103  data: dw 60, 55, 45, ....
117  swap: db 0

     bubblesort:
118     dec  cx
119     shl  cx, 1

...     ;;; sorting code ;;;

146     ret

     start:
147     mov  bx, data
148     mov  cx, 10
14D     call bubblesort

150     mov  ax, 0x4c00
153     int  0x21
```

IP

0150

SP → FFFC

0147	Free Space
0148	Free Space
…	Free Space
FFEE	Free Space
FFEF	Free Space
FFF0	Free Space
FFF1	Free Space
FFF2	Free Space
FFF3	Free Space
FFF4	Free Space
FFF5	Free Space
FFF6	Free Space
FFF7	Free Space
FFF8	Free Space
FFF9	Free Space
FFFA	Free Space
FFFB	Free Space
FFFC	50
FFFD	01
FFFE	Free Space
FFFF	Free Space

Ret Demo

```
    [org 0x0100]
100 jmp start

103 data: dw 60, 55, 45, ....
117 swap: db 0

    bubblesort:
118    dec  cx
119    shl  cx, 1

...    ;;; sorting code ;;;

146    ret

    start:
147    mov  bx, data
148    mov  cx, 10
14D    call bubblesort

150    mov  ax, 0x4c00
153    int  0x21
```

IP

0150

SP

0147	Free Space
0148	Free Space
...	Free Space
FFEE	Free Space
FFEF	Free Space
FFF0	Free Space
FFF1	Free Space
FFF2	Free Space
FFF3	Free Space
FFF4	Free Space
FFF5	Free Space
FFF6	Free Space
FFF7	Free Space
FFF8	Free Space
FFF9	Free Space
FFFA	Free Space
FFFB	Free Space
FFFC	50 (leftover garbage)
FFFD	01 (leftover garbage)
FFFE	Free Space
FFFF	Free Space

Push Demo

- Suppose AX = 0x219C

PUSH AX

SP →

0147	Free Space
0148	Free Space
…	Free Space
FFEE	Free Space
FFEF	Free Space
FFF0	Free Space
FFF1	Free Space
FFF2	Free Space
FFF3	Free Space
FFF4	Free Space
FFF5	Free Space
FFF6	Free Space
FFF7	Free Space
FFF8	Free Space
FFF9	Free Space
FFFA	Free Space
FFFB	Free Space
FFFC	Free Space
FFFD	Free Space
FFFE	Free Space
FFFF	Free Space

Push Demo

- Suppose AX = 0x219C

PUSH AX

i. SP ← SP – 2

ii. [SP] ← AX

0147	Free Space
0148	Free Space
…	Free Space
FFEE	Free Space
FFEF	Free Space
FFF0	Free Space
FFF1	Free Space
FFF2	Free Space
FFF3	Free Space
FFF4	Free Space
FFF5	Free Space
FFF6	Free Space
FFF7	Free Space
FFF8	Free Space
FFF9	Free Space
FFFA	Free Space
FFFB	Free Space
SP → FFFC	9C
FFFD	21
FFFE	Free Space
FFFF	Free Space

Stack Demo

- Suppose DX = 0x30A2

PUSH DX

0147	Free Space
0148	Free Space
…	Free Space
FFEE	Free Space
FFEF	Free Space
FFF0	Free Space
FFF1	Free Space
FFF2	Free Space
FFF3	Free Space
FFF4	Free Space
FFF5	Free Space
FFF6	Free Space
FFF7	Free Space
FFF8	Free Space
FFF9	Free Space
FFFA	Free Space
FFFB	Free Space
SP → FFFC	9C
FFFD	21
FFFE	Free Space
FFFF	Free Space

Stack Demo

- Suppose DX = 0x30A2

PUSH DX

i. SP ← SP – 2

ii. [SP] ← DX

SP → FFFA

0147	Free Space
0148	Free Space
…	Free Space
FFEE	Free Space
FFEF	Free Space
FFF0	Free Space
FFF1	Free Space
FFF2	Free Space
FFF3	Free Space
FFF4	Free Space
FFF5	Free Space
FFF6	Free Space
FFF7	Free Space
FFF8	Free Space
FFF9	Free Space
FFFA	30
FFFB	A2
FFFC	9C
FFFD	21
FFFE	Free Space
FFFF	Free Space

Stack Demo

POP BX

SP →

0147	Free Space
0148	Free Space
…	Free Space
FFEE	Free Space
FFEF	Free Space
FFF0	Free Space
FFF1	Free Space
FFF2	Free Space
FFF3	Free Space
FFF4	Free Space
FFF5	Free Space
FFF6	Free Space
FFF7	Free Space
FFF8	Free Space
FFF9	Free Space
FFFA	30
FFFB	A2
FFFC	9C
FFFD	21
FFFE	Free Space
FFFF	Free Space

Stack Demo

POP BX

i. BX ← [SP]

ii. SP ← SP + 2

BX gets 0xA230

0147	Free Space
0148	Free Space
…	Free Space
FFEE	Free Space
FFEF	Free Space
FFF0	Free Space
FFF1	Free Space
FFF2	Free Space
FFF3	Free Space
FFF4	Free Space
FFF5	Free Space
FFF6	Free Space
FFF7	Free Space
FFF8	Free Space
FFF9	Free Space
FFFA	30 (leftover garbage)
FFFB	A2 (leftover garbage)
SP → FFFC	9C
FFFD	21
FFFE	Free Space
FFFF	Free Space

Paired push and pop

- Making corresponding (paired) PUSH and POP operations is the responsibility of the programmer.
- If “push ax” is followed by “pop dx”, effectively copying the value of the AX register in the DX register, the processor won’t complain.
  - Whether this sequence is logically correct or not should be ensured by the programmer.
- For example when PUSH and POP are used to save and restore registers from the stack, order must be correct so that the saved value of AX is reloaded in the AX register and not any other register.
- For this the order of POP operations need to be the reverse of the order of PUSH operations.

Saving and restoring registers

- A subroutine will typically destroy many registers, i.e. the register contents before the call would be lost.
- This is problematic for the caller code, which might be expecting those registers to stay the same before and after the call
- For that reason, a subroutine should, at the start, backup all the registers that will be modified by itself, and restore the before ret.
- Stack is a good place to keep such short term data.

Saving and restoring registers

Example 5.4

```
; bubble sort and swap subroutines saving and restoring registers
[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

```

Saving and restoring registers

```
33                  call swap                 ; swaps two elements
34                  mov  byte [swapflag], 1 ; flag that a swap has been done
35
36  noswap:         add  si, 2                ; advance si to next index
37                  cmp  si, cx               ; are we at last index
38                  jne  innerloop            ; if not compare next two
39
40                  cmp  byte [swapflag], 1 ; check if a swap has been done
41                  je   mainloop             ; if yes make another pass
42
43                  pop  si                   ; restore old value of si
44                  pop  cx                   ; restore old value of cx
45                  pop  ax                   ; restore old value of ax
46                  ret                       ; go back to where we came from
47
48  start:          mov  bx, data             ; send start of array in bx
49                  mov  cx, 10               ; send count of elements in cx
50                  call bubblesort           ; call our subroutine
51
52                  mov  bx, data2            ; send start of array in bx
53                  mov  cx, 20               ; send count of elements in cx
54                  call bubblesort           ; call our subroutine again
55
56                  mov  ax, 0x4c00           ; terminate program
57                  int  0x21
```

19-21 When multiple registers are pushed, order is very important. If AX, CX, and SI are pushed in this order, they must be popped in the reverse order of SI, CX, and AX. This is again because the stack behaves in a Last In First Out manner.