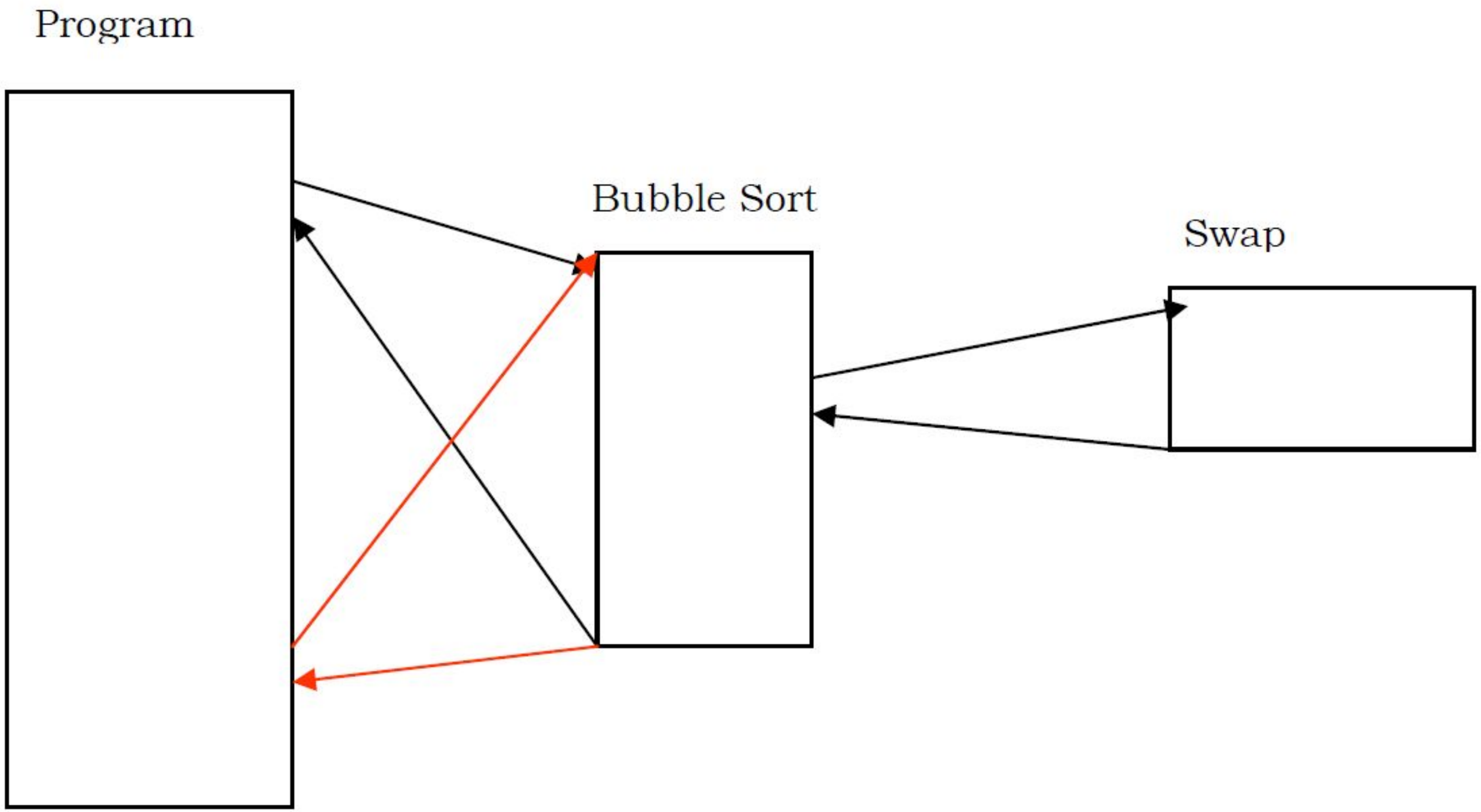
Subroutines

Outline

- Introduction to subroutines
- Introduction to Stacks
- Saving and restoring registers
- Parameter passing through stack
- Local variables

Introduction to subroutines



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

```
; 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
```

```
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.

Stacks

- Stack is a data structure that behaves in a first in last out manner.
- It can contain many elements and there is only one way in and out of the container.
- When an element is inserted it sits on top of all other elements and when an element is removed the one sitting at top of all others is removed first.
- To visualize the structure consider a test tube and put some balls in it.
- The second ball will come above the first and the third will come above the second.
- When a ball is taken out only the one at the top can be removed.
- The operation of placing an element on top of the stack is called pushing the element and the operation of removing an element from the top of the stack is called popping the element.
- The last thing pushed is popped out first; the last in first out behavior.

- We can peek at any ball inside the test tube but we cannot remove it without removing every ball on top of it.
- Similarly we can read any element from the stack but cannot remove it without removing everything above it.
- The stack operations of pushing and popping only work at the top of the stack.
- This top of stack is contained in the SP register.
- The physical address of the stack 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.

- Whenever an element is pushed on the stack SP is decremented by two as the 8088 stack works on word sized elements.
- Single bytes cannot be pushed or popped from the stack.
- Also it is a decrementing stack.
- 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.
- Another processor 8051 by the same manufacturer has an incrementing stack while 8088 has a decrementing one.

- 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 operations of placing items on the stack and removing them from there are called push and pop.
- 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.

- 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.

- 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.
- There is a form of the RET instruction called “RET n” where n is a numeric argument.
- After performing the operation of RET, it further increments the stack pointer by this number, i.e. SP is first incremented by two and then by n.
- Its function will become clear when parameter passing is discussed.

- Now we describe the operation of the stack in CALL and RET with an example.
- The top of stack stored in the stack pointer is initialized at 2000.
- The space above SP is considered empty and free.
- When the stack pointer is decremented by two, we took a word from the empty space and can use it for our purpose.
- The unit of stack operations is a word.
- Some instructions push multiple words; however byte pushes cannot be made.
- Now the value 017B is stored in the word reserved on the stack.
- The RET will copy this value in the instruction pointer and increment the stack pointer by two making it 2000 again, thereby reverting the operation of CALL.
- This is how CALL and RET behave for near calls.
- There is also a far version of these functions when the target routine is in another segment.
- This version of CALL takes a segment offset pair just like the far jump instruction.
- The CALL will push both the segment and the offset on the stack in this case, followed by loading CS and IP with the values given in the instruction.
- The corresponding instruction RETF will pop the offset in the instruction pointer followed by popping the segment in the code segment register.

Apart from CALL and RET, the operations that use the stack are PUSH and POP. Two other operations that will be discussed later are INT and IRET. Regarding the stack, the operation of PUSH is similar to CALL however with a register other than the instruction pointer. For example “push ax” will push the current value of the AX register on the stack. The operation of PUSH is shown below.

```
SP ← SP - 2
[SP] ← AX
```

The operation of POP is the reverse of this. A copy of the element at the top of the stack is made in the operand, and the top of the stack is incremented afterwards. The operation of “pop ax” is shown below.

```
AX ← [SP]
SP ← SP + 2
```

- Making corresponding 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.

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
```

```
                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.

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

```

```
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.

PUSH

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

POP

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

Observe that the operand of PUSH is called a source operand since the data is moving to the stack from the operand, while the operand of POP is called destination since data is moving from the stack to the operand.

CALL

CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. For an intra segment direct CALL, SP is decremented by two and IP is pushed onto the stack. The target procedure's relative displacement from the CALL instruction is then added to the instruction pointer. For an inter segment direct CALL, SP is decremented by two, and CS is pushed onto the stack. CS is replaced by the segment word contained in the instruction. SP again is decremented by two. IP is pushed onto the stack and replaced by the offset word in the instruction.

The out-of-line procedure is the temporary division, the concept of roundabout that we discussed. Near calls are also called intra segment calls, while far calls are called inter-segment calls. There are also versions that are called indirect calls; however they will be discuss later when they are used.

RET

RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two. If RETF (inter segment RET) is used the word at the top of the stack is popped into the IP register and SP is incremented by two. The word at the new top of stack is popped into the CS register, and SP is again incremented by two. If an optional pop value has been specified, RET adds that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the CALL instruction.

Parameter passing through Stack

- Due to the limited number of registers, parameter passing by registers is constrained in two ways.
- The maximum parameters a subroutine can receive are seven when all the general registers are used.
- Also, with the subroutines are themselves limited in their use of registers, and this limited increases when the subroutine has to make a nested call thereby using certain registers as its parameters.
- Due to this, parameter passing by registers is not expandable and generalizable. However this is the fastest mechanism available for passing parameters and is used where speed is important.

- Considering stack as an alternate, we observe that whatever data is placed there, it stays there, and across function calls as well.
- For example the bubble sort subroutine needs an array address and the count of elements.
- If we place both of these on the stack, and call the subroutine afterwards, it will stay there.
- The subroutine is invoked with its return address on top of the stack and its parameters beneath it.

- To access the arguments from the stack, the immediate idea that strikes is to pop them off the stack.
- And this is the only possibility using the given set of information.
- However the first thing popped off the stack would be the return address and not the arguments.
- This is because the arguments were first pushed on the stack and the subroutine was called afterwards.
- The arguments cannot be popped without first popping the return address.
- If a heaving thing falls on someone’s leg, the heavy thing is removed first and the leg is not pulled out to reduce the damage.
- Same is the case with our parameters on which the return address has fallen.

- To handle this using PUSH and POP, we must first pop the return address in a register, then pop the operands, and push the return address back on the stack so that RET will function normally.
- However so much effort doesn’t seem to pay back the price.
- Processor designers should have provided a logical and neat way to perform this operation.
- They did provided a way and infact we will do this without introducing any new instruction.

- Recall that the default segment association of the BP register is the stack segment and the reason for this association had been deferred for now.
- The reason is to peek inside the stack using the BP register and read the parameters without removing them and without touching the stack pointer.
- The stack pointer could not be used for this purpose, as it cannot be used in an effective address.
- It is automatically used as a pointer and cannot be explicitly used.
- Also the stack pointer is a dynamic pointer and sometimes changes without telling us in the background.
- It is just that whenever we touch it, it is where we expect it to be.
- The base pointer is provided as a replacement of the stack pointer so that we can peek inside the stack without modifying the structure of the stack.

- When the bubble sort subroutine is called, the stack pointer is pointing to the return address.
- Two bytes below it is the second parameter and four bytes below is the first parameter.
- The stack pointer is a reference point to these parameters.
- If the value of SP is captured in BP, then the return address is located at [bp+0], the second parameter is at [bp+2], and the first parameter is at [bp+4].
- This is because SP and BP both had the same value and they both defaulted to the same segment, the stack segment.

- This copying of SP into BP is like taking a snapshot or like freezing the stack at that moment.
- Even if more pushes are made on the stack decrementing the stack pointer, our reference point will not change.
- The parameters will still be accessible at the same offsets from the base pointer.
- If however the stack pointer increments beyond the base pointer, the references will become invalid.
- The base pointer will act as the datum point to access our parameters.
- However we have destroyed the original value of BP in the process, and this will cause problems in nested calls where both the outer and the inner subroutines need to access their own parameters.
- The outer subroutine will have its base pointer destroyed after the call and will be unable to access its parameters.

- To solve both of these problems, we reach at the standard way of accessing
- parameters on the stack.
- The first two instructions of any subroutines accessing its parameters from the stack are given below:
  - push bp
  - mov bp, sp
- As a result our datum point has shifted by a word.
- Now the old value of BP will be contained in [bp] and the return address will be at [bp+2].
- The second parameters will be [bp+4] while the first one will be at [bp+6].
- We give an example of bubble sort subroutine using this standard way of argument passing through stack.

Example 5.5

```
; 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 two

```

```
37                 cmp  byte [swapflag], 1 ; check if a swap has been done
38                 je   mainloop           ; if yes make another pass
39
40                 pop  si                 ; restore old value of si
41                 pop  cx                 ; restore old value of cx
42                 pop  bx                 ; restore old value of bx
43                 pop  ax                 ; restore old value of ax
44                 pop  bp                 ; restore old value of bp
45                 ret  4                  ; go back and remove two params
46
47 start:          mov  ax, data
48                 push ax                 ; place start of array on stack
49                 mov  ax, 10
50                 push ax                 ; place element count on stack
51                 call bubblesort         ; call our subroutine
52
53                 mov  ax, data2
54                 push ax                 ; place start of array on stack
55                 mov  ax, 20
56                 push ax                 ; place element count on stack
57                 call bubblesort         ; call our subroutine again
58
59                 mov  ax, 0x4c00         ; terminate program
60                 int  0x21
```

11	The value of the stack pointer is captured in the base pointer. With further pushes SP will change but BP will not and therefore we will read parameters from bp+4 and bp+6.
45	The form of RET that takes an argument is used causing four to be added to SP after the return address has been popped in the instruction pointer. This will effectively discard the parameters that are still there on the stack.
47-50	We push the address of the array we want to sort followed by the count of elements. As immediate cannot be directly pushed in the 8088 architecture, we first load it in the AX register and then push the AX register on the stack.

Stack Clearing by Caller or Callee

- Parameters pushed for a subroutine are a waste after the subroutine has returned. They have to be cleared from the stack.
- Either of the caller and the callee can take the responsibility of clearing them from there.
- If the callee has to clear the stack it cannot do this easily unless RET n exists.
- That is why most general processors have this instruction.
- Stack clearing by the caller needs an extra instruction on behalf of the caller after every call made to the subroutine, unnecessarily increasing instructions in the program.
- If there are thousand calls to a subroutine the code to clear the stack is repeated a thousand times.
- Therefore the prevalent convention in most high level languages is stack clearing by the callee; even though the other convention is still used in some languages.

- If RET n is not available, stack clearing by the callee is a complicated process.
- It will have to save the return address in a register, then remove the parameters, and then place back the return address so that RET will function.
- When this instruction was introduced in processors, only then high level language designers switched to stack clearing by the callee.
- This is also exactly why RET n adds n to SP after performing the operation of RET.
- The other way around would be totally useless for our purpose.
- Consider the stack condition at the time of RET and this will become clear why this will be useless.
- Also observe that RET n has discarded the arguments rather than popping them as they were no longer of any use either of the caller or the callee.

- The strong argument in favour of callee cleared stacks is that the arguments were placed on the stack for the subroutine, the caller did not needed them for itself, so the subroutine is responsible for removing them.
- Removing the arguments is important as if the stack is not cleared or is partially cleared the stack will eventually become full, SP will reach 0, and thereafter wraparound producing unexpected results.
- This is called stack overflow. Therefore clearing anything placed on the stack is very important.

Local variables

- Another important role of the stack is in the creation of local variables that are only needed while the subroutine is in execution and not afterwards.
- They should not take permanent space like global variables.
- Local variables should be created when the subroutine is called and discarded afterwards.
- So that the spaced used by them can be reused for the local variables of another subroutine.
- They only have meaning inside the subroutine and no meaning outside it.

- The most convenient place to store these variables is the stack.
- We need some special manipulation of the stack for this task.
- We need to produce a gap in the stack for our variables.
- This is explained with the help of the swapflag in the bubble sort example.

- The swapflag we have declared as a word occupying space permanently is only needed by the bubble sort subroutine and should be a local variable.
- Actually the variable was introduced with the intent of making it a local variable at this time.
- The stack pointer will be decremented by an extra two bytes thereby producing a gap in which a word can reside.
- This gap will be used for our temporary, local, or automatic variable; however we name it.
- We can decrement it as much as we want producing the desired space, however the decrement must be by an even number, as the unit of stack operation is a word.
- In our case we needed just one word. Also the most convenient position for this gap is immediately after saving the value of SP in BP.
- So that the same base pointer can be used to access the local variables as well; this time using negative offsets.
- The standard way to start a subroutine which needs to access parameters and has local variables is as under.
    - push bp
    - mov bp, sp
    - sub sp, 2

- The gap could have been created with a dummy push, but the subtraction makes it clear that the value pushed is not important and the gap will be used for our local variable.
- Also gap of any size can be created in a single instruction with subtraction.
- The parameters can still be accessed at bp+4 and bp+6 and the swapflag can be accessed at bp-2.
- The subtraction in SP was after taking the snapshot; therefore BP is above the parameters but below the local variables.
- The parameters are therefore accessed using positive offsets from BP and the local variables are accessed using negative offsets.
- We modify the bubble sort subroutine to use a local variable to store the swap flag.
- The swap flag remembered whether a swap has been done in a particular iteration of bubble sort.

Example 5.6

```
; 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
              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
```

```
33  noswap:         add  si, 2                ; advance si to next index
34                  cmp  si, cx               ; are we at last index
35                  jne  innerloop            ; if not compare next two
36
37                  cmp  word [bp-2], 1       ; check if a swap has been done
38                  je   mainloop             ; if yes make another pass
39
40                  pop  si                   ; restore old value of si
41                  pop  cx                   ; restore old value of cx
42                  pop  bx                   ; restore old value of bx
43                  pop  ax                   ; restore old value of ax
44                  mov  sp, bp               ; remove space created on stack
45                  pop  bp                   ; restore old value of bp
46                  ret  4                    ; go back and remove two params
47
48  start:          mov  ax, data
49                  push ax                   ; place start of array on stack
50                  mov  ax, 10
51                  push ax                   ; place element count on stack
52                  call bubblesort           ; call our subroutine
53
54                  mov  ax, data2
55                  push ax                   ; place start of array on stack
56                  mov  ax, 20
57                  push ax                   ; place element count on stack
58                  call bubblesort           ; call our subroutine again
59
60                  mov  ax, 0x4c00           ; terminate program
61                  int  0x21
```

11	A word gap has been created for swap flag. This is equivalent to a dummy push. The registers are pushed above this gap.
23	The swapflag is accessed with [bp-2]. The parameters are accessed in the same manner as the last examples.
44	We are removing the hole that we created. The hole is removed by restoring the value of SP that it had at the time of snapshot or at the value it had before the local variable was created. This can be replaced with “add sp, 2” however the one used in the code is preferred since it does not require to remember how much space for local variables was allocated in the start. After this operation SP points to the old value of BP from where we can proceed as usual.