**Name: WALEED AKRAM**

**Roll no: 20p-0640**

**Section: 3-B**

**Report# 08**

**Computer Organizational & assembly Language**

**Q # 01:**

Program Flow

A very important weapon in our arsenal is the conditional  
jump instruction. During the course of last two chapters we used these tools to write two very useful algorithms of sorting and multiplication. The multiplication algorithm is useful even though there is a MUL instruction in the 8088 instructions set, which can multiply 8bit and 16bit operands. This is because of the extensibility of our algorithm, as it is not limited to 16bits and can do 32bit or 64bit multiplication with minor changes.  
Both of these algorithms will be used a number of times in any program of a reasonable size and complexity. An application does not only need to multiply at a single point in code; it multiplies at a number of places. If multiplication or sorting is needed at 100 places in code, copying it 100 times is a totally infeasible solution. Maintaining such a code is an impossible task.  
The straightforward solution to this problem using the concepts we have acquainted till now is to write the code at one place with a label, and whenever we need to sort we jump to this label. But there is problem with this logic, and the problem is that after sorting is complete how the processor will know where to go back. The immediate answer is to jump back to a label following the jump to bubble sort. But we have jumped to bubble sort from 100 places in code. Which of the 100 positions in code should we jump  
back? Jump back at the first invocation, but jump has a single fixed target.  
How will the second invocation work? The second jump to bubble sort will never have control back at the next line. Instruction are tied to one another forming an execution thread, just like a knitted thread where pieces of cotton of different sizes are twisted together to form a thread. This thread of execution is our program. The jump instruction breaks this thread permanently, making a permanent diversion, like a turn on a highway. The conditional jump selects one of the two possible directions, like right or left turn on a road. So there is no concept of returning.  
However there are roundabouts on roads as well that take us back from where we started after having traveled on the boundary of the round. This is the concept of a temporary diversion. Two or more permanent diversions can take us back from where we started, just like two or more road turns can take us back to the starting point, but they are still permanent diversions in their nature.  
We need some way to implement the concept of temporary diversion in assembly language. We want to create a roundabout of bubble sort, another roundabout of our multiplication algorithm, so that we can enter into the roundabout whenever we need it and return back to wherever we left from after completing the round.

**CALL and RET**In every processor, instructions are available to divert temporarily and to divert permanently. The instructions for permanent diversion in 8088 are the jump instructions, while the instruction for temporary diversion is the CALL instruction. The word call must be familiar to the readers from subroutine called in higher level languages. The CALL instruction allows temporary diversion and therefore reusability of code. Now we can place the code for bubble sort at one place and reuse it again and again. This was not possible with permanent diversion. Actually the 8088 permanent diversion mechanism can be tricked to achieve temporary diversion. However, it is not possible without getting into a lot of trouble. The key idea in doing it this way is to use the jump instruction form that takes a register as argument. Therefore, this is not impossible but this is not the way it is done.  
The natural way to do this is to use the CALL instruction followed by a  
label, just like JMP is followed by a label. Execution will divert to the code  
following the label. Till now the operation has been similar to the JMP  
instruction. When the subroutine completes we need to return. The RET  
instruction is used for this purpose. The word return holds in its meaning that we are to return from where we came and need no explicit destination.  
Therefore RET takes no arguments and transfers control back to the  
instruction following the CALL that took us in this subroutine. The actual  
technical process that informs RET where to return will be discussed later after we have discussed the system stack.  
CALL takes a label as argument and execution starts from that label, until  
the RET instruction is encountered and it takes execution back to the  
instruction following the CALL. Both the instructions are commonly used as a pair, however technically they are independent in their operation. The RET works regardless of the CALL and the CALL works regardless of the RET. If you CALL a subroutine it will not complain if there is no RET present and similarly if you RET without being called it won’t complain. It is a logical pair and is used as a pair in every decent code. However sometimes we play tricks with the processor and we use CALL or RET alone.

**Q # 02:**

First 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

LIST FILE: q02.lst

1

2 [org 0x0100]

3 00000000 E94400 jmp start

4 00000003 3C0037002D00320028- data: dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0

5 0000000C 00230019001E000A00-

6 00000015 0000

7 00000017 00 swap: db 0

8 bubblesort:

9 00000018 49 dec cx ; last element not compared

10 00000019 D1E1 shl cx, 1 ; turn into byte count

11 mainloop:

12 0000001B BE0000 mov si, 0 ; initialize array index to zero

13 0000001E C606[1700]00 mov byte [swap], 0 ; reset swap flag to no swaps

14 innerloop:

15 00000023 8B00 mov ax, [bx+si] ; load number in ax

16 00000025 3B4002 cmp ax, [bx+si+2] ; compare with next number

17 00000028 760D jbe noswap ; no swap if already in order

18 0000002A 8B5002 mov dx, [bx+si+2] ; load second element in dx

19 0000002D 8910 mov [bx+si], dx ; store first number in second

20 0000002F 894002 mov [bx+si+2], ax ; store second number in first

21 00000032 C606[1700]01 mov byte [swap], 1 ; flag that a swap has been done

22 noswap:

23 00000037 81C60200 add si, 2 ; advance si to next index

24 0000003B 39CE cmp si, cx ; are we at last index

25 0000003D 75E4 jne innerloop ; if not compare next two

26 0000003F 803E[1700]01 cmp byte [swap], 1 ; check if a swap has been done

27 00000044 74D5 je mainloop ; if yes make another pass

28 00000046 C3 ret ; go back to where we came from

29 start:

30 00000047 BB[0300] mov bx, data ; send start of array in bx

31 0000004A B90A00 mov cx, 10 ; send count of elements in cx

32 0000004D E8C8FF call bubblesort ; call our subroutine

33 00000050 B8004C mov ax, 0x4c00 ; terminate program

34 00000053 CD21 int 0x21

LST Q2-1

 [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 inde

        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 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

LIST FILE: q02-1.lst

2 [org 0x0100]

3 00000000 E96C00 jmp start

4 00000003 3C0037002D00320028- data: dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0

5 0000000C 00230019001E000A00-

6 00000015 0000

7 00000017 480149018203DB2265- data2: dw 328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98

8 00000020 2029090A006D036301-

9 00000029 6200

10 0000002B 78031502D007FC031E- dw 888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5

11 00000034 00C800F902A7005A00-

12 0000003D 0500

13 0000003F 00 swap: db 0

14 bubblesort:

15 00000040 49 dec cx ; last element not compared

16 00000041 D1E1 shl cx, 1 ; turn into byte count

17 mainloop:

18 00000043 BE0000 mov si, 0 ; initialize array index to zero

19 00000046 C606[3F00]00 mov byte [swap], 0 ; reset swap flag to no swaps

20 innerloop:

21 0000004B 8B00 mov ax, [bx+si] ; load number in ax

22 0000004D 3B4002 cmp ax, [bx+si+2] ; compare with next number

23 00000050 760D jbe noswap ; no swap if already in order

24 00000052 8B5002 mov dx, [bx+si+2] ; load second element in dx

25 00000055 8910 mov [bx+si], dx ; store first number in second

26 00000057 894002 mov [bx+si+2], ax ; store second number in first

27 0000005A C606[3F00]01 mov byte [swap], 1 ; flag that a swap has been done

28 noswap:

29 0000005F 81C60200 add si, 2 ; advance si to next inde

30 00000063 39CE cmp si, cx ; are we at last index

31 00000065 75E4 jne innerloop ; if not compare next two

32 00000067 803E[3F00]01 cmp byte [swap], 1 ; check if a swap has been done

33 0000006C 74D5 je mainloop ; if yes make another pass

34 0000006E C3 ret ; go back to where we came from

35 start:

36 0000006F BB[0300] mov bx, data ; send start of array in bx

37 00000072 B90A00 mov cx, 10 ; send count of elements in cx

38 00000075 E8C8FF call bubblesort ; call our subroutine

39 00000078 BB[1700] mov bx, data2 ; send start of array in bx

40 0000007B B91400 mov cx, 20 ; send count of elements in cx

41 0000007E E8BFFF call bubblesort ; call our subroutine again

42 00000081 B8004C mov ax, 0x4c00 ; terminate program

43 00000084 CD21 int 0x21

**Q # 03:**

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

    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 cxComputer Architecture & Assembly Language Programming Course Code: CS401

        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

LST Q #3

1

2 [org 0x0100]

3 00000000 E96F00 jmp start

4 00000003 3C0037002D00320028- data: dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0

5 0000000C 00230019001E000A00-

6 00000015 0000

7 00000017 480149018203DB2265- data2: dw 328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98

8 00000020 2029090A006D036301-

9 00000029 6200

10 0000002B 78031502D007FC031E- dw 888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5

11 00000034 00C800F902A7005A00-

12 0000003D 0500

13 0000003F 00 swapflag: db 0

14 swap:

15 00000040 8B00 mov ax, [bx+si] ; load first number in ax

16 00000042 874002 xchg ax, [bx+si+2] ; exchange with second number

17 00000045 8900 mov [bx+si], ax ; store second number in first

18 00000047 C3 ret ; go back to where we came from

19 bubblesort:

20 00000048 49 dec cx ; last element not compared

21 00000049 D1E1 shl cx, 1 ; turn into byte count

22 mainloop:

23 0000004B BE0000 mov si, 0 ; initialize array index to zero

24 0000004E C606[3F00]00 mov byte [swapflag], 0 ; reset swap flag to no swaps

25 innerloop:

26 00000053 8B00 mov ax, [bx+si] ; load number in ax

27 00000055 3B4002 cmp ax, [bx+si+2] ; compare with next number

28 00000058 7608 jbe noswap ; no swap if already in order

29 0000005A E8E3FF call swap ; swaps two elements

30 0000005D C606[3F00]01 mov byte [swapflag], 1 ; flag that a swap has been done

31 noswap:

32 00000062 81C60200 add si, 2 ; advance si to next index

33 00000066 39CE cmp si, cx ; are we at last index

34 00000068 75E9 jne innerloop ; if not compare next two

35 0000006A 803E[3F00]01 cmp byte [swapflag], 1 ; check if a swap has been done

36 0000006F 74DA je mainloop ; if yes make another pass

37 00000071 C3 ret ; go back to where we came from

38 start:

39 00000072 BB[0300] mov bx, data ; send start of array in bx

40 00000075 B90A00 mov cx, 10 ; send count of elements in cxComputer Architecture & Assembly Language Programming Course Code: CS401

41 00000078 E8CDFF call bubblesort ; call our subroutine

42 0000007B BB[1700] mov bx, data2 ; send start of array in bx

43 0000007E B91400 mov cx, 20 ; send count of elements in cx

44 00000081 E8C4FF call bubblesort ; call our subroutine again

45 00000084 B8004C mov ax, 0x4c00 ; terminate program

46 00000087 CD21 int 0x21

Q # 04:

**SAVING AND RESTORING REGISTERS**The subroutines we wrote till now have been destroying certain registers  
and our calling code has been carefully written to not use those registers.  
However this cannot be remembered for a good number of subroutines.  
Therefore our subroutines need to implement some mechanism of retaining  
the callers’ value of any registers used.  
The trick is to use the PUSH and POP operations and save the callers’  
value on the stack and recover it from there on return. Our swap subroutine  
destroyed the AX register while the bubble sort subroutine destroyed AX, CX,  
and SI. BX was not modified in the subroutine. It had the same value at  
entry and at exit; it was only used by the subroutine. Our next example  
improves on the previous version by saving and restoring any registers that it  
will modify using the PUSH and POP operations.

**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 discussed 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.

[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 orderComputer Architecture & Assembly Language Programming Course Code: CS401

        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

LIST FILE: q04-1.lst

1 ; bubble sort and swap subroutines saving and restoring registers

2 [org 0x0100]

3 00000000 E97700 jmp start

4 00000003 3C0037002D00320028- data: dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0

5 0000000C 00230019001E000A00-

6 00000015 0000

7 00000017 480149018203DB2265- data2: dw 328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98

8 00000020 2029090A006D036301-

9 00000029 6200

10 0000002B 78031502D007FC031E- dw 888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5

11 00000034 00C800F902A7005A00-

12 0000003D 0500

13 0000003F 00 swapflag: db 0

14 00000040 50 swap: push ax ; save old value of ax

15 00000041 8B00 mov ax, [bx+si] ; load first number in ax

16 00000043 874002 xchg ax, [bx+si+2] ; exchange with second number

17 00000046 8900 mov [bx+si], ax ; store second number in first

18 00000048 58 pop ax ; restore old value of ax

19 00000049 C3 ret ; go back to where we came from

20 0000004A 50 bubblesort: push ax ; save old value of ax

21 0000004B 51 push cx ; save old value of cx

22 0000004C 56 push si ; save old value of si

23 0000004D 49 dec cx ; last element not compared

24 0000004E D1E1 shl cx, 1 ; turn into byte count

25 00000050 BE0000 mainloop: mov si, 0 ; initialize array index to zero

26 00000053 C606[3F00]00 mov byte [swapflag], 0 ; reset swap flag to no swaps

27 00000058 8B00 innerloop: mov ax, [bx+si] ; load number in ax

28 0000005A 3B4002 cmp ax, [bx+si+2] ; compare with next number

29 0000005D 7608 jbe noswap ; no swap if already in orderComputer Architecture & Assembly Language Programming Course Code: CS401

30 0000005F E8DEFF call swap ; swaps two elements

31 00000062 C606[3F00]01 mov byte [swapflag], 1 ; flag that a swap has been done

32 00000067 81C60200 noswap: add si, 2 ; advance si to next index

33 0000006B 39CE cmp si, cx ; are we at last index

34 0000006D 75E9 jne innerloop ; if not compare next two

35 0000006F 803E[3F00]01 cmp byte [swapflag], 1 ; check if a swap has been done

36 00000074 74DA je mainloop ; if yes make another pass

37 00000076 5E pop si ; restore old value of si

38 00000077 59 pop cx ; restore old value of cx

39 00000078 58 pop ax ; restore old value of ax

40 00000079 C3 ret ; go back to where we came from

41 0000007A BB[0300] start: mov bx, data ; send start of array in bx

42 0000007D B90A00 mov cx, 10 ; send count of elements in cx

43 00000080 E8C7FF call bubblesort ; call our subroutine

44 00000083 BB[1700] mov bx, data2 ; send start of array in bx

45 00000086 B91400 mov cx, 20 ; send count of elements in cx

46 00000089 E8BEFF call bubblesort ; call our subroutine again

47 0000008C B8004C mov ax, 0x4c00 ; terminate program

48 0000008F CD21 int 0x21