Name: Waleed Akram

Roll no: 20p-0640

Section: 3-B

Report # 09

Computer Organizational & assembly Language

**Q#5:**

Parameters 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 provide a way and  
in-fact 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.

[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 twoComputer Architecture & Assembly Language Programming Course Code

        cmp byte [swapflag], 1 ; check if a swap has been done

        je mainloop ; if yes make another pass

        pop si ; restore old value of si

        pop cx ; restore old value of cx

        pop bx ; restore old value of bx

        pop ax ; restore old value of ax

        pop bp ; restore old value of bp

        ret 4 ; go back and remove two params

    start:

        mov ax, data

        push ax ; place start of array on stack

        mov ax, 10

        push ax ; place element count on stack

        call bubblesort ; call our subroutine

        mov ax, data2

        push ax ; place start of array on stack

        mov ax, 20

        push ax ; place element count on stack

        call bubblesort ; call our subroutine again

        mov ax, 0x4c00 ; terminate program

        int 0x21

Q#1 LIST File Screen Shoots:

1 [org 0x0100]

2 00000000 E97D00 jmp start

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

4 0000000C 00230019001E000A00-

5 00000015 0000

6 00000017 480149018203DB2265- data2: dw 328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98

7 00000020 2029090A006D036301-

8 00000029 6200

9 0000002B 78031502D007FC031E- dw 888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5

10 00000034 00C800F902A7005A00-

11 0000003D 0500

12 0000003F 00 swapflag: db 0

13 bubblesort:

14 00000040 55 push bp ; save old value of bp

15 00000041 89E5 mov bp, sp ; make bp our reference point

16 00000043 50 push ax ; save old value of ax

17 00000044 53 push bx ; save old value of bx

18 00000045 51 push cx ; save old value of cx

19 00000046 56 push si ; save old value of si

20 00000047 8B5E06 mov bx, [bp+6] ; load start of array in bx

21 0000004A 8B4E04 mov cx, [bp+4] ; load count of elements in cx

22 0000004D 49 dec cx ; last element not compared

23 0000004E D1E1 shl cx, 1 ; turn into byte count

24 mainloop:

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

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

27 innerloop:

28 00000058 8B00 mov ax, [bx+si] ; load number in ax

29 0000005A 3B4002 cmp ax, [bx+si+2] ; compare with next number

30 0000005D 760A jbe noswap ; no swap if already in order

31 0000005F 874002 xchg ax, [bx+si+2] ; exchange ax with second number

32 00000062 8900 mov [bx+si], ax ; store second number in first

33 00000064 C606[3F00]01 mov byte [swapflag], 1 ; flag that a swap has been done

34 noswap:

35 00000069 81C60200 add si, 2 ; advance si to next index

36 0000006D 39CE cmp si, cx ; are we at last index

37 0000006F 75E7 jne innerloop ; if not compare next twoComputer Architecture & Assembly Language Programming Course Code

38 00000071 803E[3F00]01 cmp byte [swapflag], 1 ; check if a swap has been done

39 00000076 74D8 je mainloop ; if yes make another pass

40 00000078 5E pop si ; restore old value of si

41 00000079 59 pop cx ; restore old value of cx

42 0000007A 5B pop bx ; restore old value of bx

43 0000007B 58 pop ax ; restore old value of ax

44 0000007C 5D pop bp ; restore old value of bp

45 0000007D C20400 ret 4 ; go back and remove two params

46 start:

47 00000080 B8[0300] mov ax, data

48 00000083 50 push ax ; place start of array on stack

49 00000084 B80A00 mov ax, 10

50 00000087 50 push ax ; place element count on stack

51 00000088 E8B5FF call bubblesort ; call our subroutine

52 0000008B B8[1700] mov ax, data2

53 0000008E 50 push ax ; place start of array on stack

54 0000008F B81400 mov ax, 20

55 00000092 50 push ax ; place element count on stack

56 00000093 E8AAFF call bubblesort ; call our subroutine again

57 00000096 B8004C mov ax, 0x4c00 ; terminate program

58 00000099 CD21 int 0x21

Q#6:

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  
swap-flag in the bubble sort example.  
The swap-flag 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 swap-flag 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.

[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 ; .............. of bx

        push cx ; .............. of cx

        push si ; -------------- of si

        mov bx, [bp+6] ; load start of array in bx

        mov cx, [bp+4] ; load count of elements in cx

        dec cx ; last element not compared

        shl cx, 1 ; turn into byte count

    mainloop:

        mov si, 0 ; initialize array index to zero

        mov word [bp-2], 0 ; reset swap flag to no swaps

    innerloop:

        mov ax, [bx+si] ; load number in ax

        cmp ax, [bx+si+2] ; compare with next number

        jbe noswap ; no swap if already in order

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

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

        mov word [bp-2], 1 ; flag that a swap has been done

    noswap:

        add si, 2 ; advance si to next index

        cmp si, cx ; are we at last index

        jne innerloop ; if not compare next two

        cmp word [bp-2], 1 ; check if a swap has been done

        je mainloop ; if yes make another pass

        pop si ; restore old value of si

        pop cx ; restore old value of cx

        pop bx ; restore old value of bx

        pop ax ; restore old value of ax

        mov sp, bp ; remove space created on stack

        pop bp ; restore old value of bp

        ret 4 ; go back and remove two params

    start:

        mov ax, data

        push ax ; place start of array on stack

        mov ax, 10

        push ax ; place element count on stack

        call bubblesort ; call our subroutine

        mov ax, data2

        push ax ; place start of array on stack

        mov ax, 20

        push ax ; place element count on stack

        call bubblesort ; call our subroutine again

        mov ax, 0x4c00 ; terminate program

        int 0x21

Q#2 LIST File Screen Shoots:

1 [org 0x0100]

2 00000000 E98200 jmp start

3

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

14 bubblesort:

15 0000003F 55 push bp ; save old value of bp

16 00000040 89E5 mov bp, sp ; make bp our reference point

17 00000042 81EC0200 sub sp, 2 ; make two byte space on stack

18 00000046 50 push ax ; save old value of ax

19 00000047 53 push bx ; .............. of bx

20 00000048 51 push cx ; .............. of cx

21 00000049 56 push si ; -------------- of si

22 0000004A 8B5E06 mov bx, [bp+6] ; load start of array in bx

23 0000004D 8B4E04 mov cx, [bp+4] ; load count of elements in cx

24 00000050 49 dec cx ; last element not compared

25 00000051 D1E1 shl cx, 1 ; turn into byte count

26

27 mainloop:

28 00000053 BE0000 mov si, 0 ; initialize array index to zero

29 00000056 C746FE0000 mov word [bp-2], 0 ; reset swap flag to no swaps

30

31 innerloop:

32 0000005B 8B00 mov ax, [bx+si] ; load number in ax

33 0000005D 3B4002 cmp ax, [bx+si+2] ; compare with next number

34 00000060 760A jbe noswap ; no swap if already in order

35 00000062 874002 xchg ax, [bx+si+2] ; exchange ax with second number

36 00000065 8900 mov [bx+si], ax ; store second number in first

37 00000067 C746FE0100 mov word [bp-2], 1 ; flag that a swap has been done

38

39 noswap:

40 0000006C 81C60200 add si, 2 ; advance si to next index

41 00000070 39CE cmp si, cx ; are we at last index

42 00000072 75E7 jne innerloop ; if not compare next two

43 00000074 817EFE0100 cmp word [bp-2], 1 ; check if a swap has been done

44 00000079 74D8 je mainloop ; if yes make another pass

45 0000007B 5E pop si ; restore old value of si

46 0000007C 59 pop cx ; restore old value of cx

47 0000007D 5B pop bx ; restore old value of bx

48 0000007E 58 pop ax ; restore old value of ax

49 0000007F 89EC mov sp, bp ; remove space created on stack

50 00000081 5D pop bp ; restore old value of bp

51 00000082 C20400 ret 4 ; go back and remove two params

52

53 start:

54 00000085 B8[0300] mov ax, data

55 00000088 50 push ax ; place start of array on stack

56 00000089 B80A00 mov ax, 10

57 0000008C 50 push ax ; place element count on stack

58 0000008D E8AFFF call bubblesort ; call our subroutine

59 00000090 B8[1700] mov ax, data2

60 00000093 50 push ax ; place start of array on stack

61 00000094 B81400 mov ax, 20

62 00000097 50 push ax ; place element count on stack

63 00000098 E8A4FF call bubblesort ; call our subroutine again

64 0000009B B8004C mov ax, 0x4c00 ; terminate program

65 0000009E CD21 int 0x21