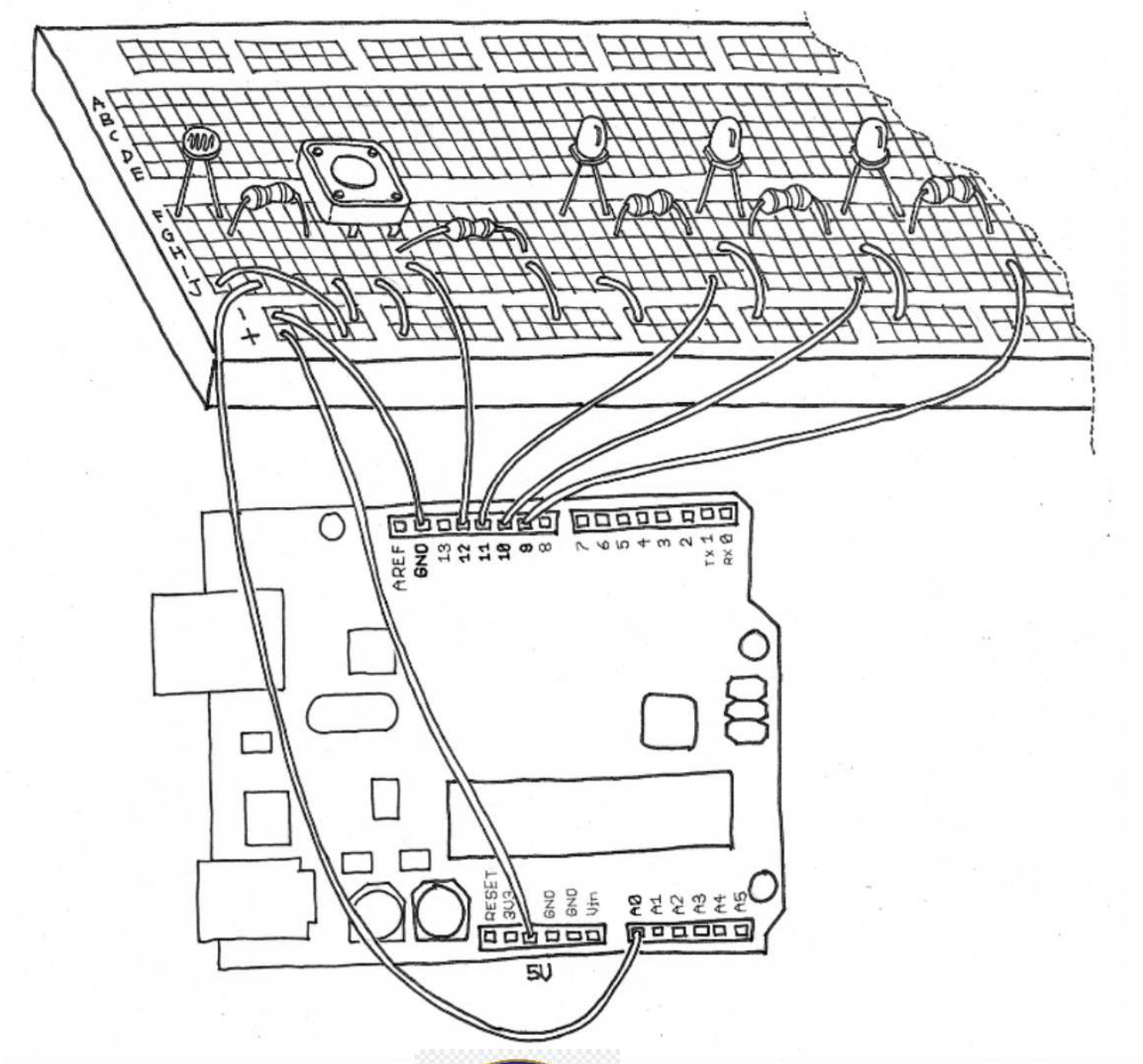
CS/EEE/INSTR F241
Microprocessor Programming and Interfacing

Lab 1- Introduction to DEBUG & DEBUGX



Dr. Vinay Chamola and Anubhav Elhence

Lab 1- Introduction to DEBUG & DEBUGX

What is DebugX?

DEBUG, supplied by MS-DOS, is a program that traces the 8086 instructions. Using DEBUG, you can easily enter 8086 machine code program into memory and save it as an executable MS-DOS file (in .COM/.EXE format). DEBUG can also be used to test and debug 8086 and 8088 programs. The features include examining and changing memory and register contents including the CPU register. The programs may also be single-stepped or run at full speed to a break point.

You will be using DEBUGX which is a program similar to DEBUG but offers full support for 32-bit instructions and addressing. DEBUGX includes the 80x86 instructions through the Pentium instructions.

Installing DosBox and DebugX

To install the software which are prerequisites for x86 programming, follow the link below and these steps:-.

Link:- BITS IoT Lab Download Link

[For Windows]

1. On the above page, click on "*Link to DosBox*" installer, and install DosBox (There should be an icon on the desktop after this).
2. Now, click on the "*Link to MASM*", and extract the MASM611 folder.
3. Copy the MASM611 folder directly in the 'D' drive (Remember the drive, this will be important later)
4. Make sure "debugx" is present in the MASM611→BIN folder.
5. You don't need to install anything from/within MASM611 folder (Everything is already pre-compiled in the folder)

How to Start DebugX

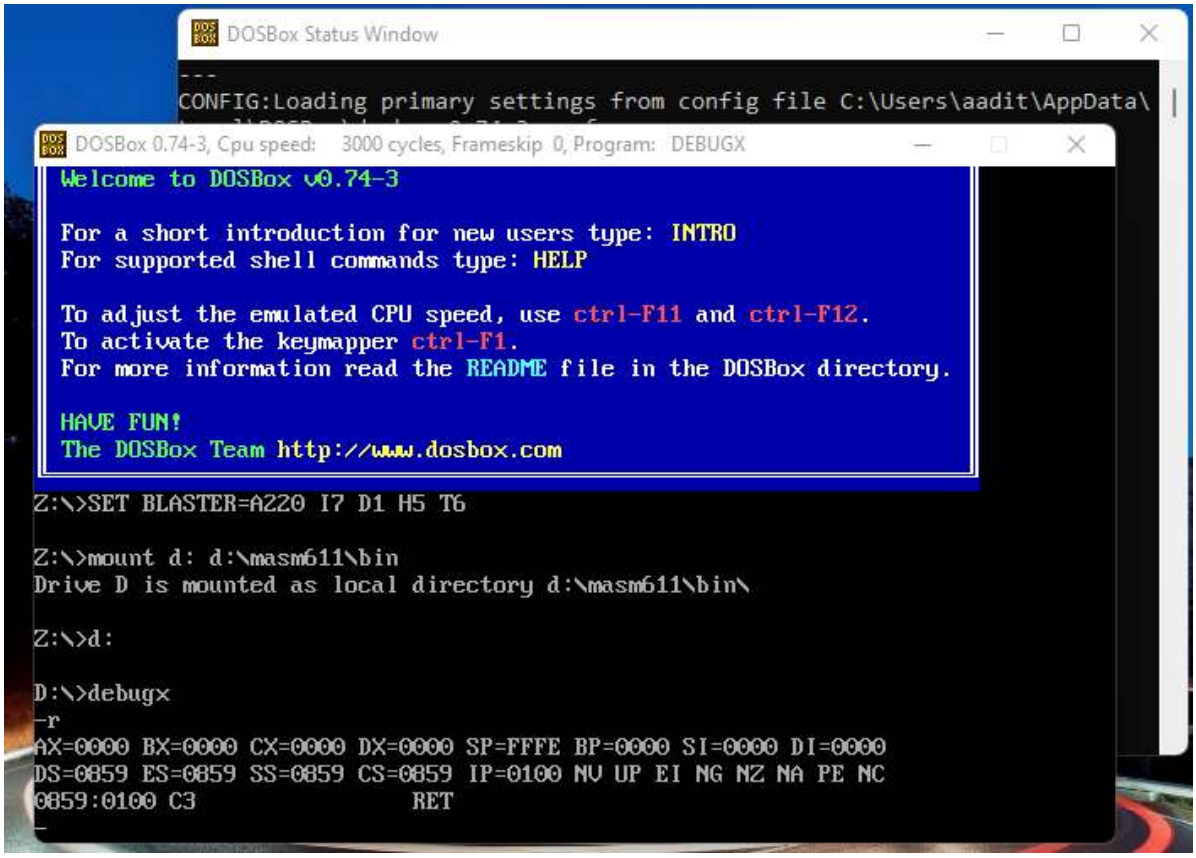
Once you have all the software installed, this is the procedure you'll need to follow to start DebugX. In short, we need to mount the drive where the MASM611 folder is stored (D Drive here). *You will have to do this many times, please note it down*:-

1. Click on the **DosBox icon** on your desktop (This opens two windows, you only need to focus on the one with the command prompt).
2. The command prompt will read 'Z:\>', type in "**mount d: d:\masm611\bin**" (If successful you should get a message 'Drive D is mounted as local directory')
3. Now type "**d:**" to change the command prompt to 'D:\>'
4. Type "**debugx**" to start the program (If succsessful, the cursor moving to the next line.)
5. You can return to DosBox, by typing "q" (Quit command)

In Summary (See figure below):-

- **DOSBox icon**
- **mount d: d:\masm611\bin**
- **d:**
- **debugx**

Great! You are ready to go. To check if everything worked, type r (Register command) in DEBUGX to see all the registers.



DebugX Basic Conventions

- **DebugX is NOT case sensitive.**
- **DEBUGX recognizes ONLY Hexadecimal numbers (without a trailing 'H' by default, so '11' is interpreted as seventeen, not eleven!!)**
- **DEBUGX displays a list of the commands and their parameters by entering a "?" at the command prompt.**
- **Segment and offset are specified as Segment:Offset**
- **Spaces in commands are only used to indicate separate parameters**

List of DebugX Commands

These are some vital commands you will use in DEBUGX, to perform tasks like viewing the registers, to executing and verifying your assembling language programs.

We recommend you to try out these commands, in the following tasks, to get accustomed to using DEBUGX. More details on the commands will be shared in the upcoming lab sessions.

Command Syntax	Description
Register	
RX	Activates 32-bit registers ('386 regs on')
R	Shows the 16-bit registers (Default) or the 32-bit registers, if rx was used before
R <register>	**View a register** and change its value at the prompt
Execution	
A :<offset>	Assemble- Prompts the code segment to **write instructions** (assembled into machine code)
U <offset>	Unassemble- **Displays the Symbolic code** (instructions) written at the offset (from CS)
T	Trace- Execute commands at CS:IP, i.e. **one at a time** (debugging)
G <address of last instruction>	Go- Executes commands **all at once**, until the address specificed (Change IP Value using R first!)
Data	
D :<offset>	Dump- **View the data** at this address (Little Endian)
E :<offset>	Enter- **Edit data** at this address by changing the value at the prompt
Misc.	
?	**View** all debugx commands
Q	**Quit** debugx

Vital DEBUG Commands

A: Assemble

The Assemble command (A) is used to enter assembly mode. In assembly mode, DEBUG prompts for each assembly language statement and converts the statement into machine code that is then stored in memory. The optional start address specifies the address at which to assemble the first instruction. The default start address is 100h. A blank line entered at the prompt causes DEBUG to exit assembly mode.

Syntax: A [address]

D: Dump

The Dump command (D), when used without a parameter, causes DEBUG to display the contents of the 128-byte block of memory starting at CS:IP if a target program is loaded, or starting at CS:100h if no target program is loaded. The optional range parameter can be used to specify a starting address, or a starting and ending address, or a starting address and a length. Subsequent Dump commands without a parameter cause DEBUG to display the contents of the 128-byte block of memory following the block displayed by the previous Dump command.

Syntax: D [range]

R: Register

The Register command (R), when used without a parameter, causes DEBUG to display the contents of the target program's CPU registers. The optional register parameter will cause DEBUG to display the contents of the register and prompt for a new value.

Syntax: R [register]

Syntax: R [register] [value]

T: Trace

The Trace command (T), when used without a parameter, causes DEBUG to execute the instruction at CS:IP. Before the instruction is executed, the contents of the register variables are copied to the actual CPU registers. After the instruction has executed, and updated the actual CPU registers (including IP), the contents of the actual CPU registers are copied back to the register variables and the contents of these variables are displayed. The optional address parameter can be used to specify the starting address. The optional count parameter can be used to specify the number of instructions to execute. To differentiate the address parameter from the count parameter, the address parameter must be preceded with an "=". Note that the first byte at the specified address must be the start of a valid instruction.

Syntax: T [=address] [number]

Tasks to be Completed

1. Using three addressing modes (Immediate, Register, Register-Indirect), write instructions to

- **Move the value 1133H into the register AX.**
- **Swap the lower and higher bytes in AX and move them into BX (If AX is pqrs, BX should be rspq)**
- **Move the value in BX to the memory location at an offset of 20 (from BX)**

Note down the machine code equivalents of the four MOV statements.

(*Hint:* You need to use the following commands- A to write the instructions, and U to view the machine code and unassembled instructions, T to execute and D to view the memory location)

2. Move the first letter of your name (ASCII Character) to the location DS:0120
(*Hint*: Recall the rules for the Immediate addressing mode)

3.Fill 32 (decimal) bytes of the Extra Segment with ASCII characters for the first two letters of your name. (Like "ABABAB...")

(*Hint:* Use the F (Fill) command to fill a memory region with a byte pattern

To fill, for example, the first 8000h bytes of the current data segment with pattern 55:

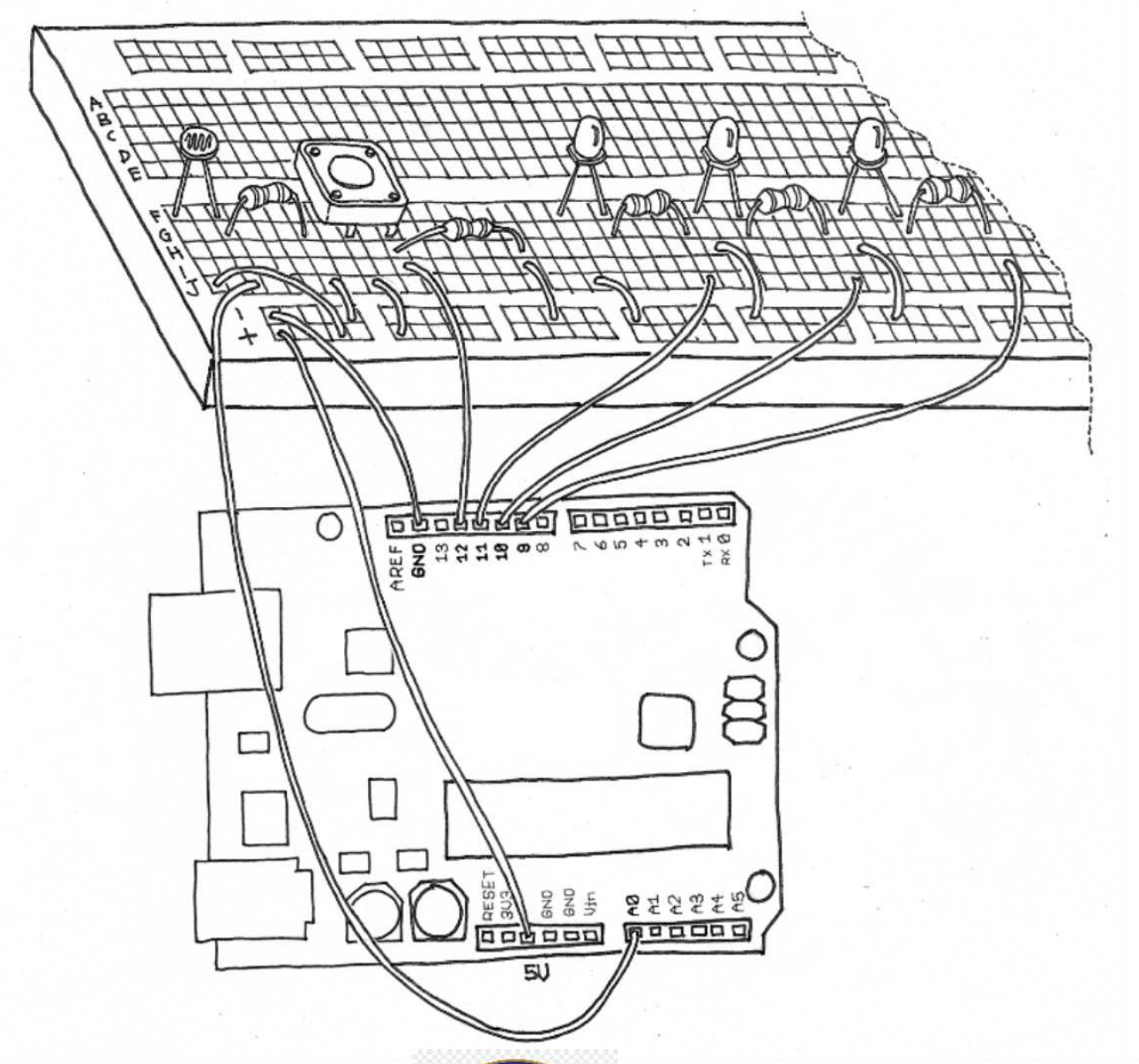
F 0 L 8000 55

[Syntax: F <start-address> L <range> <pattern>])


BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
ज्ञानं परमं बलम्

CS/EEE/INSTR F241
Microprocessor Programming and Interfacing

Lab 2- Debugx Commands in Detail



Dr. Vinay Chamola and Anubhav Elhence

Lab 2 - DEBUGX Commands in Detail

After completing the previous lab session, we hope you are more confident with using DEBUGX. In this lab session, we start with a recap of using DebugX and an overview of the commands. This is followed by a more detailed exploration of the commands. Finally, you will be exposed to a few other commands in the Tasks for this session.

DEBUGX Recap

DEBUGX:-

- **Numbers in Hexadecimal by default**
- **NOT Case Sensitive**
- **Segment:Offset**
- **16-bit registers by default (RX – toggles between 32-bit and 16-bit register mode)**

Starting DEBUGX:-

- **Click on the DOSBox icon**
- **At the command prompt, type '*mount d: d:\masm611\bin*'**
  [Replace 'd' with the drive containing the folder MASM611]
- **Again type '*d:*'**
- **Finally, type '*debugx*'**

Commands Overview:-

- **A: Assembles symbolic instructions into machine code**
- **D: Display the contents of an area of memory in hex format**
- **E: Enter data into memory beginning at specific location**
- **G: Run the executable program in the memory (Go)**
- **P: Proceed, Execute a set of related instructions**
- **Q: Quit the debug session**
- **R: Display the contents of one or more registers in hex format**
- **T: Trace the execution of on instruction**
- **U: Unassemble machine code into symbolic code**
- **?: Debug Help**

Commands in Detail

R, the Register command: examining and altering the content of registers

In DEBUGX, the register command "R" allows you to examine and/or alter the contents of the internal CPU registers.

R <register name>

The "R" command will display the contents of all registers unless the optional <register name> field is entered. In which case, only the content of the selected register will be displayed. The R command also displays the instruction pointed to by the IP. The "R" command without the register name field, responds with three lines of information. The first two lines show you the programmer's model of the 80x86 microprocessor. (The first line displays the contents of the general-purpose, pointer, and index registers. The second line displays the current values of the segment registers, the instruction pointer, and the flag register bits. The third line shows the location, machine code, and Assembly code of the next instruction to be executed.) If the optional <register name> field is specified in the "R" command, DEBUGX will display the contents of the selected register and give you an opportunity to change its value. Just type a <return> if no change is needed, e.g.

- r ECX

ECX 00000000

: FFFF

- r ECX

ECX 0000FFFF

Note that DEBUGX pads input numbers on the left with zeros if fewer than four digits are typed in for 16-bit register mode and if fewer than eight digits in 32 bit addressing mode

A, the Assemble command

The assemble command is used to enter Assembly language instructions into memory.

A <starting address>

The starting address may be given as an offset number, in which case it is assumed to be an offset into the code segment. Otherwise, CS can also be specified explicitly. (eg. “A 100” and “A CS:100” will achieve the same result). When this command is entered, DEBUG/DEBUGX will begin prompting you to enter Assembly language instructions. After an instruction is typed in and followed by <return>, DEBUG/DEBUGX will prompt for the next instruction. This process is repeated until you type a <return> at the address prompt, at which time DEBUG/DEBUGX will return you to the debug command prompt.

Use the “A” command to enter the following instructions starting from offset 0100H.

32-bit format

A 100

xxxx:0100 mov eax,1

xxxx:0106 mov ebx,2

xxxx:010C mov ecx,3

xxxx:0112 add eax, ecx

xxxx:0115 add eax, ebx

xxxx:0118 jmp 100

xxxx:011A <enter>

16-bit format

- A 100

xxxx:0100 mov ax,1

xxxx:0103 mov bx,2

xxxx:0106 mov cx,3

xxxx:0109 add ax, cx

xxxx:010B add ax, bx

xxxx:010D jmp 100

xxxx:010F <enter>

Where xxxx specifies the base value of instruction address in the code segment. Please note that the second instruction starts at xxxx:0106 in 32 bit format and xxxx:0103 in 16-bit format. This implies that first instruction is six bytes long in 32-bit format and three byte long in 16-bit format.

Similarly note the size of all instructions. When DEBUGX is first invoked, what are the values in the general-purpose registers? What is the reason for setting [IP] = 0100 ?

U, the Unassemble command: looking at machine code

The unassemble command displays the machine code in memory along with their equivalent

Assembly language instructions. The command can be given in either format shown below:

U <starting address> <ending address>

32-bit format

U 100 118

xxxx:0100 66B801000000 mov eax,01

xxxx:0106 66BB02000000 mov ebx,02

xxxx:010C 66B903000000 mov ecx,03

xxxx:0112 6603C3 add eax, ebx

xxxx:0115 6603C1 add eax, ecx

xxx:0118 EBE6 jmp 0100

16-bit format

U 100 10D

xxxx:0100 B80100 mov ax,1

xxxx:0103 BB0200 mov bx,2

xxxx:0106 B90300 mov cx,3

xxxx:0109 03C3 add ax,bx

xxxx:010B 03C1 add ax,cx

xxxx:010D EBF1 jmp 100

All the data transfer and arithmetic instructions in 32-bit format have 66 as a prefix why?

E, the Enter Command

The Enter command (E) is used to enter data into memory. The required address parameter specifies the starting location at which to enter the data. If the address parameter does not specify a segment, DEBUG uses DS. If the optional list parameter is not included, DEBUG displays the byte value at the specified address and prompts for a new value. The spacebar or minus key can be used to move to the next or previous byte. Pressing the Enter key without entering a value terminates the command.

E <memory_address>

Examples:

-E 100 01 02 03

-E 100 'ABC'

-E 100 'ABC' 0

T, the Trace command: single-step execution

The trace command allows you to trace through the execution of your programs one or more

instructions at a time to verify the effect of the programs on registers and/or data.

T < =starting address>

T =100

If you do not specify the starting address then you have to set the IP using the R command before using the T command.

Usage

Using r command to set IP to 100 and then execute T 5

Using r command to set IP to 100 and then execute T

Trace through the above sequence of instructions that you have entered and examine the registers and Flag registers

G, the Go command

The go command instructs DEBUG to execute the instructions found between the starting and

stop addresses.

G < = starting address> < stop address>

Execute the following command and explain the result (this is with respect to code you have

written in 32-bit format earlier)

G = 100 118

Caution: the command G= 100 119 will cause the system to hang. Explain why?

Often, it is convenient to pause the program after executing a few instructions, thus effectively creating a breakpoint in the program. For example, the command "g 10c" tells the DEBUG to start executing the next instruction(s) and pause at offset 010CH.

The main difference between the GO and TRACE commands is that the GO command lists the register values after the execution of the last instruction, while the TRACE command does so after each instruction execution.

Try executing the sequence of instructions you entered using several options of the "G" command. Remember to reload 0100 into IP every time you use G without the starting address.

Please refer to the file MASM611\BIN\DEBUG.txt for further details about the commands. It is very well documented.

N, the Name command

The Name command (N) is used to input a filename (actually, a file specification that can include a drive and a path) for a subsequent Load or Write command. Note that this command *does not create a new file*. It is meant for files already present in the BIN folder (files outside of this may result in "Access Denied" errors).

N filename.txt

L, the Load command

The Load command (L) is used to load a file into the specified memory address. *The file to load is specified with a preceding Name command (N)*. The optional address parameter specifies the load address. *The default load address is CS:100* for all files other than EXE files, for which information in the file header determines the load address. After DEBUGX loads the file it *sets BX:CX to the file size in bytes.*

L <memory_address>

Tasks to be Completed

Create a .txt file that contains your First Name in small letters followed by "MUP," separated by a '*.' For example, if your name is 'Apple,' then the content of the file should be

Apple*MUP

Now place this string in the Extra Segment at 0200h, with the "*" replaced by "[". Place an additional "]" at the end of the string too.

ES:0200 *Apple[MUP]*

(*Hint:* ASCII Characters

- **[= 5B**
- **] = 5D**
- *** = 2A**

1. Create a .txt file containing the above string in the BIN folder.

2. load the value 5B into AH and 5D into AL

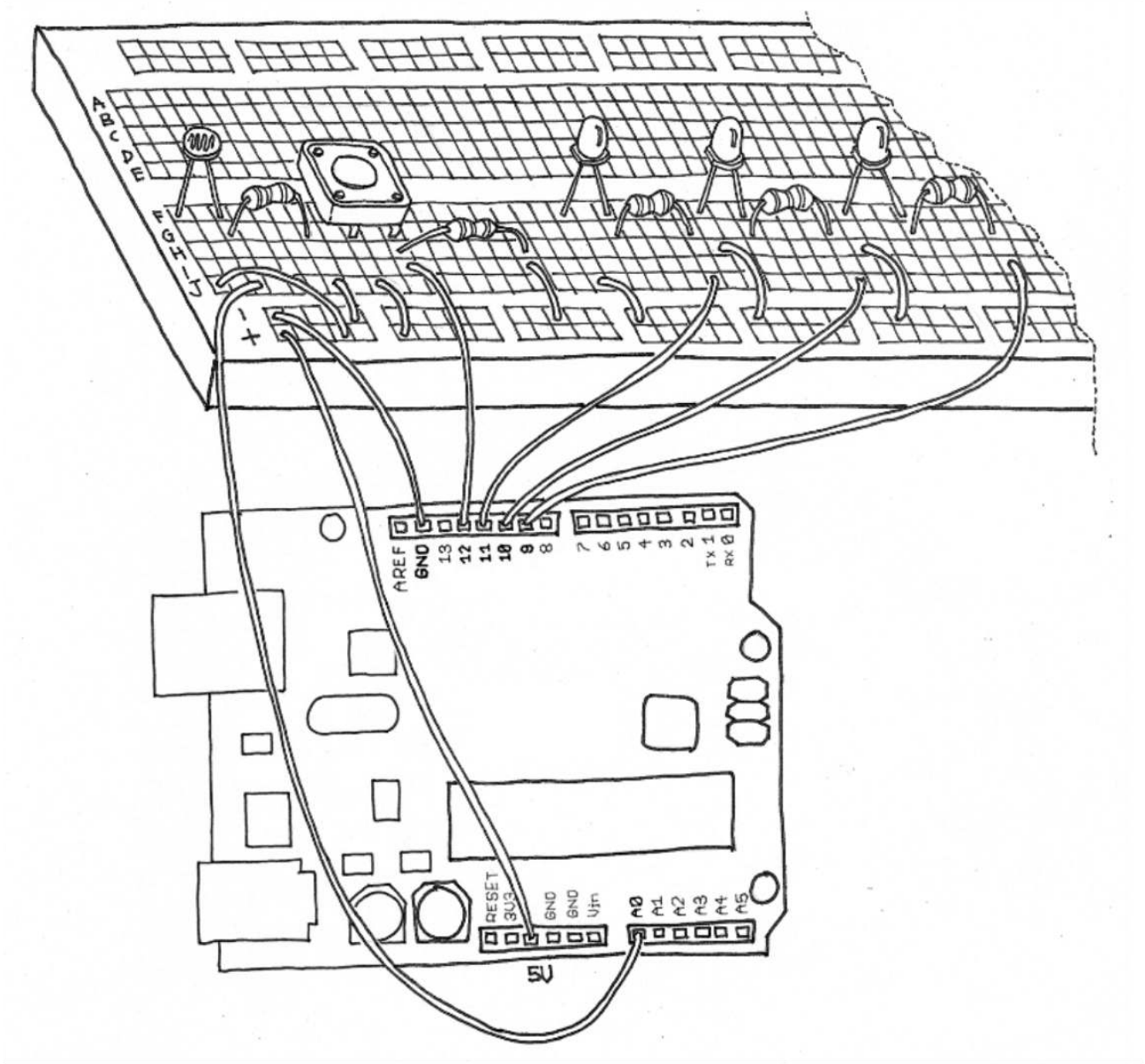
3. Load the file you created in step 1 using the L command into the code segment (assume/use offset to code segment as 0200h)

4. Copy the values AH and AL into the appropriate locations using the MOV instruction with the appropriate offsets.)


BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
ज्ञानं परमं बलम्

CS/EEE/INSTR F241
Microprocessor Programming and Interfacing

Lab 3 - Control Flow in ALP



Dr. Vinay Chamola and Anubhav Elhence

Introduction to Control Flow in ALP

What is CMP instruction?

The CMP instruction in Assembly language is part of the 80x86 instruction set and is used for comparing two values.

The CMP instruction compares two operands and sets the status flags in the flag register based on the result of the comparison. The operands can be immediate values, registers, or memory locations. The flags affected by the CMP instruction are the zero flag (ZF), carry flag (CF), sign flag (SF), overflow flag (OF), and auxiliary carry flag (AF).

Here's an example of how CMP can be used:

```
MOV AX, 5
CMP AX, 10 ; Compare AX (5) to 10
JAE LABEL ; Jump to LABEL if AX >= 10 (CF = 0)
; [.. do something if AX < 10 ..]
LABEL:
```

In this example, the CMP instruction compares the value in AX (5) with the value 10. If AX is greater than or equal to 10, the carry flag (CF) is cleared and the program will jump to the label LABEL. If AX is less than 10, the carry flag is set and the program will continue to the next instruction.

What is LODSB, LODSW, LODSD instruction?

The LODSB, LODSW, and LODSD instructions in Assembly language are part of the 80x86 instruction set and are used for loading data from memory into a register.

- **LODSB loads a byte (8-bit) value from the memory location pointed to by the source index (SI) into the destination register (AL). After the instruction is executed, the source index (SI) is incremented by 1.**
- **LODSW loads a word (16-bit) value from the memory location pointed to by the source index (SI) into the destination register (AX). After the instruction is executed, the source index (SI) is incremented by 2.**
- **LODSD loads a doubleword (32-bit) value from the memory location pointed to by the source index (SI) into the destination register (EAX). After the instruction is executed, the source index (SI) is incremented by 4.**

Here's an example of how LODSB can be used:

```
MOV SI, 1000H ; Load the starting address of the data into SI
MOV AL, 0 ; Clear AL

LOOP:
  LODSB ; Load the byte from memory into AL
  ; [.. do something with the data in AL ..]
  DEC CX ; Decrement the loop counter
  JNZ LOOP ; Repeat the loop if CX is not zero
```

In this example, **LODSB** loads the next byte from memory into AL, and the loop counter **CX** is decremented after each iteration. The loop continues as long as **CX** is not zero. The other instructions **LODSW** and **LODSD** work similarly, but load different sizes of data into the destination register.

What are the different variants of Jump Instruction in 8086 ?

The 80x86 processors have several variants of the Jump instruction. The most common variants are:

1. JMP: Unconditional jump, transfers control to a specified label or memory location.
2. JZ or JE: Jump if zero, transfers control to a specified label or memory location if the zero flag (ZF) is set.
3. JNZ or JNE: Jump if not zero, transfers control to a specified label or memory location if the zero flag (ZF) is not set.
4. JS: Jump if sign, transfers control to a specified label or memory location if the sign flag (SF) is set.
5. JNS: Jump if not sign, transfers control to a specified label or memory location if the sign flag (SF) is not set.
6. JO: Jump if overflow, transfers control to a specified label or memory location if the overflow flag (OF) is set.
7. JNO: Jump if not overflow, transfers control to a specified label or memory location if the overflow flag (OF) is not set.
8. JA or JNBE: Jump if above, transfers control to a specified label or memory location if the carry flag (CF) is not set and the zero flag (ZF) is not set.
9. JAE or JNB: Jump if above or equal, transfers control to a specified label or memory location if the carry flag (CF) is not set.
10. JB or JNAE: Jump if below, transfers control to a specified label or memory location if the carry flag (CF) is set.
11. JBE or JNA: Jump if below or equal, transfers control to a specified label or memory location if the carry flag (CF) is set or the zero flag (ZF) is set.

Here's an example of how JZ can be used:

```
MOV AX, 5
CMP AX, 10 ; Compare AX (5) to 10
JZ LABEL ; Jump to LABEL if AX = 10 (ZF = 1)
; [.. do something if AX ≠ 10 ..]
LABEL:
```

In this example, the CMP instruction compares the value in AX (5) with the value 10. If AX is equal to 10, the zero flag (ZF) is set and the program will jump to the label LABEL. If AX is not equal to 10, the zero flag is cleared and the program will continue to the next instruction.

Lab Task:

Task 1

Write an ALP that will examine the contents of set of 10 bytes starting from location 'array1' for the presence of data '0Ah' and replace it with ASCII character 'E'.

```
1   .model tiny
2   .data
3   array1        db      91h,02h,083h,0Ah,075h,0Ah,047h,012h,076h,61h
```

Task 2

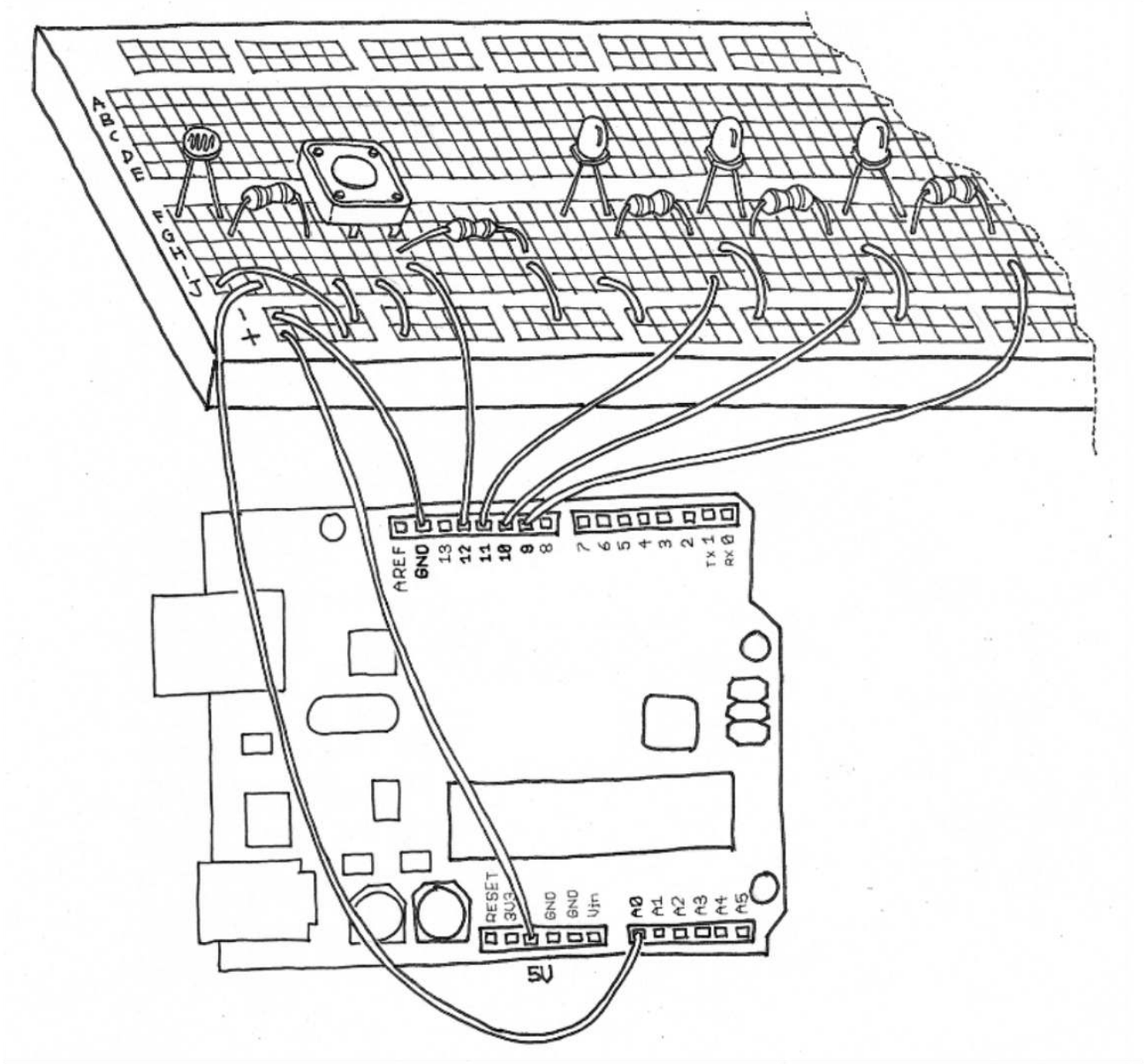
Write an ALP that will count the number of negative numbers in an array of 16-bit signed data stored from location 'array1'. The number of elements in array1 is present in location 'count'. The count of negative numbers must be stored in location 'NEG1'

```
1   .model tiny
2   .data
3   array1        db      ;fill this up yourself
4   count         db      ;since you have filled the above array,
5                         ;therefore you only know the count
6   NEG1          db      ?
```


BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
ज्ञानं परमं बलम्

CS/EEE/INSTR F241
Microprocessor Programming and Interfacing

Lab 4 - String Operations



Dr. Vinay Chamola and Anubhav Elhence

String Operations

What are LODSB, LODSW and LODSD instructions?

LODSB, LODSW, and LODSD are three x86 assembly language instructions used to load a byte (8 bits), a word (16 bits), or a doubleword (32 bits) from memory into the AL, AX, or EAX register, respectively. These instructions are part of the string operations category of instructions and are used to read data from a string of bytes, words, or doublewords in memory.

Here's a brief description of each instruction:

1. LODSB (Load String Byte): This instruction reads a byte from memory pointed to by the DS:(E)SI register pair into the AL register. It then increments or decrements the (E)SI register depending on the direction flag (DF) bit in the flags register. If the DF bit is clear, (E)SI is incremented. If the DF bit is set, (E)SI is decremented. This allows the instruction to read bytes from a string in either direction.
2. LODSW (Load String Word): This instruction reads a 16-bit word from memory pointed to by the DS:(E)SI register pair into the AX register. It then increments or decrements the (E)SI register in the same way as LODSB.
3. LODSD (Load String Doubleword): This instruction reads a 32-bit doubleword from memory pointed to by the DS:(E)SI register pair into the EAX register. It then increments or decrements the (E)SI register in the same way as LODSB.

These instructions are often used in conjunction with other string operations, such as STOSB (store string byte), STOSW (store string word), and STOSD (store string doubleword), to manipulate strings of bytes, words, or doublewords in memory.

What are STOSB, STOSW and STOSD instructions?

STOSB, STOSW, and STOSD are three x86 assembly language instructions used to store a byte (8 bits), a word (16 bits), or a doubleword (32 bits) from a register into memory. These instructions are part of the string operations category of instructions and are used to write data to a string of bytes, words, or doublewords in memory.

Here's a brief description of each instruction:

1. STOSB (Store String Byte): This instruction stores the byte in the AL register into the memory location pointed to by the ES:(E)DI register pair. It then increments or decrements the (E)DI register depending on the direction flag (DF) bit in the flags register. If the DF bit is clear, (E)DI is incremented. If the DF bit is set, (E)DI is decremented. This allows the instruction to store bytes into a string in either direction.
2. STOSW (Store String Word): This instruction stores the 16-bit word in the AX register into the memory location pointed to by the ES:(E)DI register pair. It then increments or decrements the (E)DI register in the same way as STOSB.
3. STOSD (Store String Doubleword): This instruction stores the 32-bit doubleword in the EAX register into the memory location pointed to by the ES:(E)DI register pair. It then increments or decrements the (E)DI register in the same way as STOSB.

These instructions are often used in conjunction with other string operations, such as LODSB (load string byte), LODSW (load string word), and LODSD (load string doubleword), to manipulate strings of bytes, words, or doublewords in memory.

What are SCASB, SCASW and SCASD instructions?

SCASB, SCASW, and SCASD are three x86 assembly language instructions used to compare a byte (8 bits), a word (16 bits), or a doubleword (32 bits) in memory with the AL, AX, or EAX register, respectively. These instructions are part of the string operations category of instructions and are used to search for a byte, word, or doubleword in a string of bytes, words, or doublewords in memory.

Here's a brief description of each instruction:

1. SCASB (Scan String Byte): This instruction compares the byte in the AL register with the byte at the memory location pointed to by the ES:(E)DI register pair. It then increments or decrements the (E)DI register depending on the direction flag (DF) bit in the flags register. If the DF bit is clear, (E)DI is incremented. If the DF bit is set, (E)DI is decremented. This allows the instruction to search for bytes in a string in either direction.
2. SCASW (Scan String Word): This instruction compares the 16-bit word in the AX register with the word at the memory location pointed to by the ES:(E)DI register pair. It then increments or decrements the (E)DI register in the same way as SCASB.
3. SCASD (Scan String Doubleword): This instruction compares the 32-bit doubleword in the EAX register with the doubleword at the memory location pointed to by the ES:(E)DI register pair. It then increments or decrements the (E)DI register in the same way as SCASB.

These instructions are often used in conjunction with other string operations, such as LODSB (load string byte), LODSW (load string word), and LODSD (load string doubleword), to manipulate and search strings of bytes, words, or doublewords in memory. After the comparison is made, the zero flag (ZF) is set if the compared values are equal, and the carry flag (CF) and the sign flag (SF) are set according to the result of the subtraction operation.

Example:

Let's say we have a string of bytes stored in memory, and we want to search for the first occurrence of the byte 0x42 (hexadecimal representation of the decimal number 66) in the string. We can use the SCASB instruction to do this search.

```
.model tiny
.data
    myString db 12h, 34h, 56h, 42h, 78h, 9Ah   ; our string of bytes
    myStringLength db 06h                ; calculate the length of the string
    res dw 00h

.code
.startup
    mov     al, 42h        ; set the byte we want to search for in the AL register
    mov     cx , 06h ; set the loop counter to the length of the string
    lea     di, myString  ; set the destination index to the start of the string

1 reference
searchLoop:
    scasb                  ; compare the byte in AL with the byte at ES:DI, and update DI accordingly
    je      found          ; if the compared bytes are equal, jump to the "found" label
    loop    searchLoop     ; decrement ECX and continue the loop if it's not zero
    jmp     notFound       ; jump to the "notFound" label if the loop completes without finding the byte

1 reference
found:
    sub     di, offset myString ; calculate the index of the found byte in the string
    mov     bx, di
    dec     bx
    lea     si, res
    mov     [si],bx; Do something with the index, for example print it out
;     ; ...

1 reference
notFound:
;     ; Handle the case where the byte was not found in the string
;     ; ...
```

In this example code, we first set the AL register to the byte we want to search for, then we set the loop counter to the length of the string and the destination index to the start of the string.

We then enter a loop where we use the SCASB instruction to compare the byte in AL with the byte at ES:DI, and update DI accordingly. If the compared bytes are equal (i.e., the ZF flag is set), we jump to the "found" label. If the loop completes without finding the byte, we jump to the "notFound" label.

In the "found" label, we calculate the index of the found byte in the string by subtracting the offset of the start of the string from the value of DI. We can then do something with this index, for example print it out. In the "notFound" label, we can handle the case where the byte was not found in the string.

A simpler way to do this is by using the REPNE instruction.

What is REPNE and REPE instruction in 8086?

REPNE (repeat not equal) and REPE (repeat equal) are prefix instructions in the x86 assembly language used to repeat string operations with certain conditions.

The REPNE prefix is used to repeat a string operation as long as the condition for not being equal is met. It can be used with string operations such as SCASB, CMPSB, SCASW, CMPSW, SCASD, and CMPSD. For example, the instruction sequence "REPNE SCASB" can be used to search for a byte in a string until the byte is found or the end of the string is reached.

The REPE prefix is used to repeat a string operation as long as the condition for being equal is met. It can also be used with string operations such as SCASB, CMPSB, SCASW, CMPSW, SCASD, and CMPSD. For example, the instruction sequence "REPE CMPSW" can be used to compare two strings of words until a difference is found or the end of the strings is reached.

Both REPNE and REPE instructions use the CX register as a counter for the number of repetitions, and they decrement CX by one after each repetition. If CX becomes zero, the string operation is terminated.

The above example using REPNE:

BIN > b.asm > end

```
1   .model tiny
2   .data
    2 references
3   array1 db 01h, 02h, 03h, 04h, 05h, 06h, 07h, 08h, 09h, 10h
    5 references
4   res dw 00h
5   .code
6   .startup
7
8       lea si, res
9       lea di, array1
10      mov al, 07h
11      mov cx, 0ah
12      cld
13      REPNE SCASB
14      sub di, offset array1
15      mov bx, di
16      dec bx
17      mov [si],bx
18
19  .exit
    2 references
20  end
21
```

What are CMPSB, CMPSW and CMPSD instructions?

CMPSB, CMPSW, and CMPSD are x86 assembly language instructions used to compare a byte (8 bits), a word (16 bits), or a doubleword (32 bits) in memory at two locations pointed to by the source and destination index registers, SI and DI, respectively. These instructions are part of the string operations category of instructions and are used to compare two strings of bytes, words, or doublewords in memory.

Here's a brief description of each instruction:

1. CMPSB (Compare String Byte): This instruction compares the byte at the memory location pointed to by the DS:SI register pair with the byte at the memory location pointed to by the ES:DI register pair. It then increments or decrements the SI and DI registers depending on the direction flag (DF) bit in the flags register. If the DF bit is clear, both registers are incremented. If the DF bit is set, both registers are decremented. This allows the instruction to compare bytes in two strings in either direction.
2. CMPSW (Compare String Word): This instruction compares the 16-bit word at the memory location pointed to by the DS:SI register pair with the 16-bit word at the memory location pointed to by the ES:DI register pair. It then increments or decrements the SI and DI registers in the same way as CMPSB.
3. CMPSD (Compare String Doubleword): This instruction compares the 32-bit doubleword at the memory location pointed to by the DS:SI register pair with the 32-bit doubleword at the memory location pointed to by the ES:DI register pair. It then increments or decrements the SI and DI registers in the same way as CMPSB.

Take a look at the example, where we try to find out the index where the two string' start to mismatch.

```
.model tiny
.data
1 reference
dat1 db 'anubhavelhence'
2 references
dat2 db 'anubhavElhence'
4 references
res dw 00h
.code
.startup

    lea si, dat1
    lea di, dat2
    mov cx, 0dh
    cld
    REPE CMPSB
    sub di, offset dat2
    mov bx, di
    dec bx
    lea si, res
    mov [si],bx

.exit
2 references
end
```

Lab Task:

Task 1

Write an 8086 program to check whether a given string is palindrome or not. If it is a palindrome, store '01h' in RES or else '00h'.

Input String: "wasitcatisaw"

Output: RES = 01h

Go to below link to download starter code: https://github.com/anubhavelhence/Microprocessor-Programming-and-Interfacing-MuP-Lab-Session/blob/week-4/q1.asm

Task 2

Write an 8086 program to replace a substring S1 of a string S with "*"

Input: S = "BITSIOTLAB", S1 = "IOT"

Output: BITS*LAB

Explanation:

Change the substrings S[4,6] to string "*" modifies the string S to "BITS*LAB"

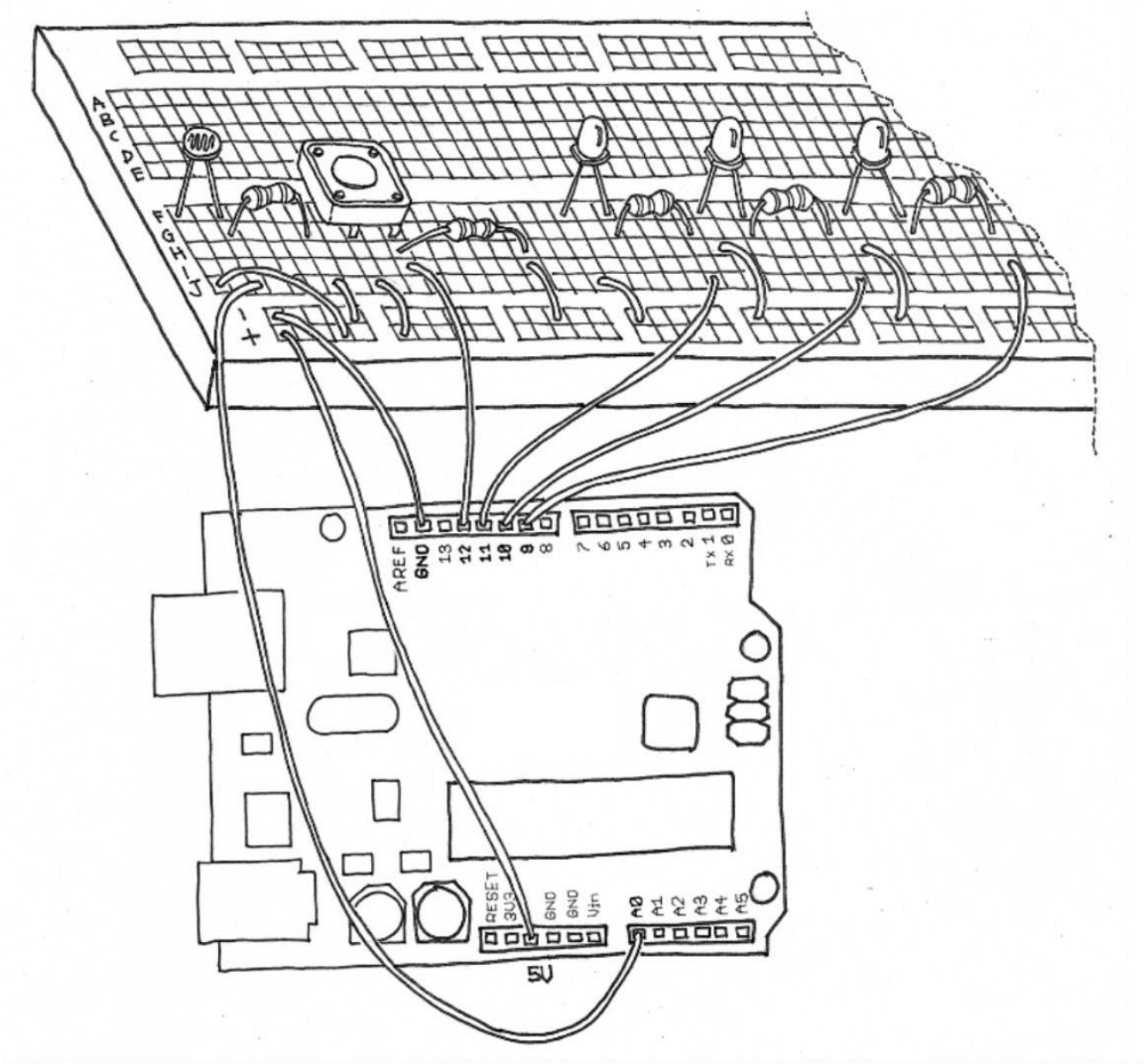
Go to below link to download starter code:

https://github.com/anubhavelhence/Microprocessor-Programming-and-Interfacing-MuP-Lab-Session/blob/week-4/q2.asm


BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
ज्ञानं परमं बलम्

CS/EEE/INSTR F241
Microprocessor Programming and Interfacing

Lab 5 - Dealing with Interrupts



Dr. Vinay Chamola and Anubhav Elhence

Introduction to DOS & BIOS Interrupts

The BIOS is specific to the individual computer system. It contains the default resident hardware-dependent drivers for several devices including the console display, keyboard, and boot disk device. Have you ever wondered how you can type characters and view them even when no OS is loaded, like when the computer is booting?

The BIOS operates at a more primitive level than the DOS kernel. In X86-based systems, there are a total of 256 possible interrupts, out of which some interrupts are reserved for DOS, and BIOS services, which the programmer can access by loading the function number into appropriate registers and then invoking the interrupt.

In providing the DOS functions, the MS-DOS kernel communicates with the BIOS device drivers through the BIOS functions. DOS then provides something like a wrapper over the primitive BIOS function. By providing the wrapper, DOS makes the functions easier to use and more reliable, but in turn, reduces the flexibility and power of the functions. By using the BIOS interrupts directly, the programmer bypasses the DOS kernel.

In the following two experiments, you will be using INT 21 H (DOS Interrupts) – Here we use INT 21H to access the value of the key pressed and display characters.

Keyboard Interrupts

Purpose: Input a character from the keyboard (STDIN) with Echo

```
MOV AH, 01h                    ; AH -01 parameter for INT 21h
INT 21h
```

You have to run the program using debug/debug32 and the ASCII key you press will be stored in AL register.

Purpose: Input a character from the keyboard (STDIN) without Echo

```
MOV AH, 08h                    ; AH -08 parameter for INT 21h
INT 21h
```

You have to run the program using debug/debug32 and the ASCII key you press will be stored in AL register. The difference is the character will not be visible on the screen when you type it

Purpose: Input a string from keyboard (STDIN)

```
.data
max1 db 32                  ; 32 is max no. of chars that a user can type in (max possible – 255)
Act1 db ?                   ; actual count of keys that user types will be stored here after int has
                            ; executed (Note this cannot exceed the value specified in max1 –
                            ; actual keys you enter will 31 as the 32nd will be Enter key)
Inp1 db 32dup(0)            ; Reserve 32 locations for input string

.code
.startup

LEA DX,max1
MOV AH, 0Ah
INT 21h
```

After the interrupt, act 1 will contain the number of characters read, and the characters themselves will start at inp1 The characters will be terminated by a carriage return (ASCII code 0Dh), although this will not be included in the count (Note: this will not be included in the ACT1 but you have to count Enter also when you are specifying it in max1)

Purpose: Output a character to display (STD OUT)

```
MOV  DL, 'A'
MOV  AH, 02h
INT   21h
```

After Interrupt is executed character 'A' will be displayed on the screen.

Purpose: Output a string on display (STDOUT)

```
.data
str1 db 'HELLO$'                    ; all strings must terminate with '$' ASCII value (24h)

.code
.startup

LEA DX, str1
MOV AH, 09h
INT 21h
```

When interrupt is executed the string "HELLO" will be displayed on screen. Remove the '$' sign. What happens

Lab Task 1

Write an ALP that will take in a alphabet entered by the user and display 'The character entered is a' if the character entered is 'a'(in both cases) else display 'not a' if the character entered is any other character but 'a'. The user entered character should not be seen on the screen.

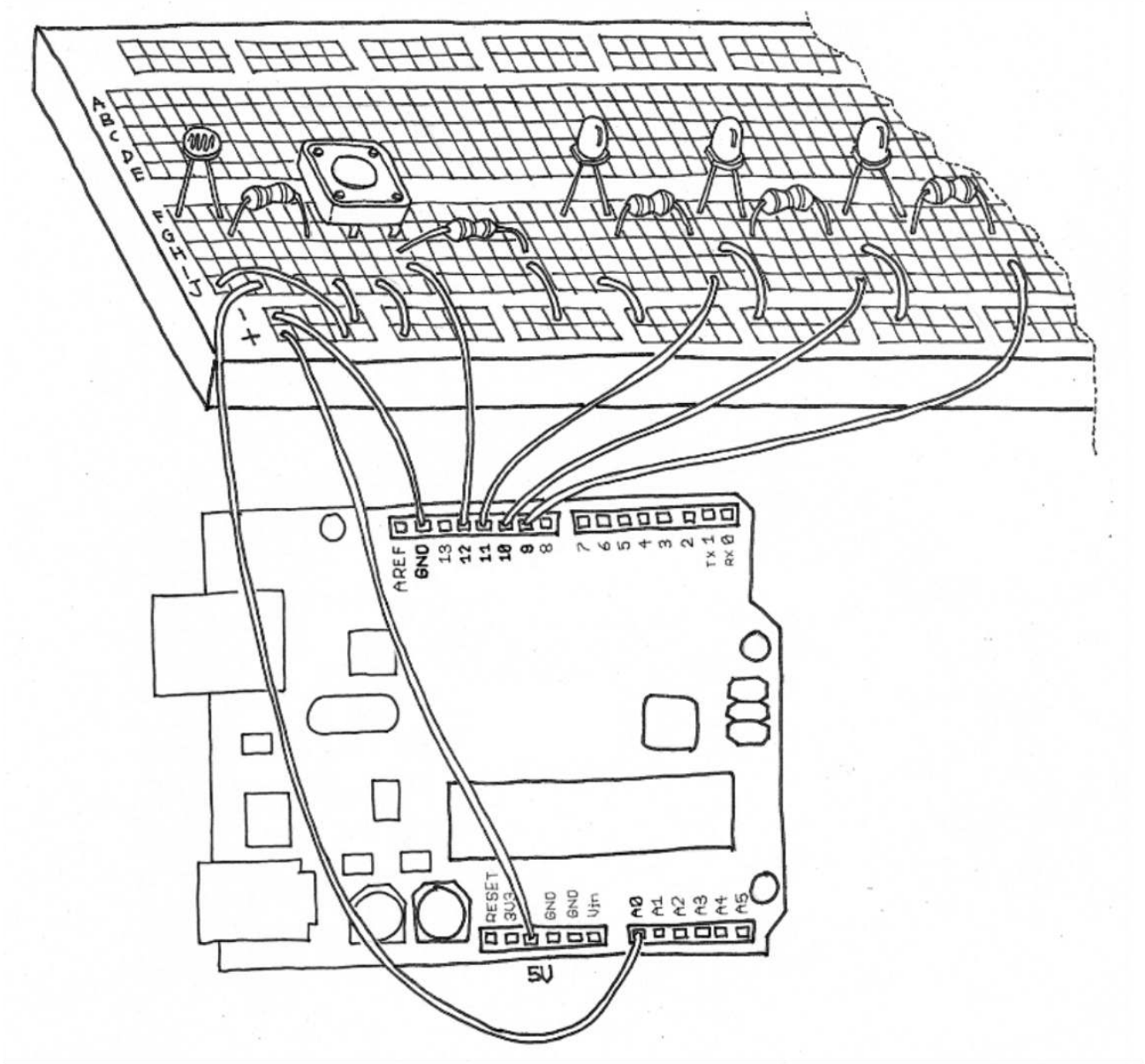
Lab Task 2

Write an ALP to convert an input string from the user (in all lowercase letters only) to uppercase letters and display it to the user.


BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
ज्ञानं परमं बलम्

CS/EEE/INSTR F241
Microprocessor Programming and Interfacing

Lab 6 - File Operations in DOS



Dr. Vinay Chamola and Anubhav Elhence

Introduction to DOS Interrupts for files

All DOS files are sequential files. All sequential files are stored and accessed from the beginning of the file towards the end. File is usually accessed through DOS INT 21H function calls. In this session we will practice, how to create, read and write a file. There are two ways of handling a file. First via the file control block and second via the file handle. We will use file handle, as this is the most common and easier of the two methods.

File handle

DOS File Handle Functions - a group of INT 21h functions that allows DOS to track open file data in its own internal tables. File Handle Functions also permit users to specify file path names. For the purposes of the following discussion, reading means copying all or part of an existing file into memory; writing a file means copying data from memory to a file; rewriting a file means replacing a file's content with other data.

Additional information about the File Handle, File Pointer, and the Error Codes is available at the bottom.

File Interrupts

Function 3Ch: Create a File

Input:

AH = 3Ch

DS:DX = address of filename

(an ASCIIZ string ending with a zero byte)

e.g of ASCIIZ – 'C:\MASM611\BIN\abc.txt',0

CL = attribute

Attribute – Bit map

BITS	7	6	5	4	3	2	1	0
Description	Shareable	-	Archive	Directory	Vol. Label	System	Hidden	Read-Only

Output:

If successful CF =0 , AX = file handle

Error: if CF = 1, AX = error code (3, 4, or 5)

Function 3Dh: Open an existing file

Input:

AH = 3DH

AL = access and sharing modes

0 = open for reading

1 = open for writing

2 = open for read/write

DS:DX = ASCIZ filename

Output:

CF clear if successful, AX = file handle

CF set on error AX = error code (01h,02h,03h,04h,05h,0Ch,56h)

Function 40h: Write to a file

Input:

AH = 40h

BX = file handle

CX = number of bytes to write

DS:DX = data address

Output:

AX = count of bytes written.

If AX < CX, error (disk full).

If CF = 1, AX = error code (5, 6)

Function 3Eh: Close a file

Input:

AH = 3Eh

BX = file handle

Output:

Error if CF = 1, AX = error code (6)

Function 3Fh: Read an existing file

Input:

AH = 3Fh

BX = file handle

CX = number of bytes to read

DS:DX = memory buffer address

Output:

AX = count of bytes actually read.

If AX = 0 or AX < CX, EOF

If CF = 1, AX = error code (5, 6)

Function 41h: Delete a file

Input:

AH = 41H

DS: DX = address of the ASCII-Z string file name

Output:

AX = error code if carry is set

Function 56h: Rename a file

Input:

AH = 56H

DS:DX = ASCIZ filename of existing file

ES:DI = ASCIZ new filename

CL = attribute mask

Output:

CF clear if successful

CF set on error, AX= error code (02h,03h,05h,11h)

Example to Create a file

```
1   .model tiny
2   .data
    3 references
3   fname   db 'test.txt',0
    5 references
4   handle  dw ?
5   .code
6   .startup
7       mov ah, 3ch
8       lea dx, fname
9       mov cl, 1h
10      int 21h
11      mov handle, ax
12  .exit
    2 references
13  end
```

Example to Open, Write and Close a File

```
.model tiny
.data
4 references
fname   db 'sec.txt',0
6 references
handle  dw ?
3 references
msg     db 'MuP docks!'
.code
.startup

    ; Create a file if it
    ; is not existing
    mov ah, 3ch
    lea dx, fname
    mov cl, 1h
    int 21h
    mov handle, ax

    ; open file
    mov ah, 3dh
    mov al, 1h
    lea dx, fname
    int 21h
    mov handle, ax

    ; write msg to file
    mov ah, 40h
    mov bx, handle
    mov cx, 10
    lea dx, msg
    int 21h

    ; close file
    mov ah, 3eh
    int 21h
.exit
```

Example to Open, Write and Close a File

```
.model tiny
.data
4 references
fname   db 'USER.txt', 0
6 references
handle  dw ?
3 references
msg db 20 dup('$')
.code
.startup
    ; open file
    mov ah, 3dh
    mov al, 0h
    lea dx, fname
    int 21h
    mov handle, ax

    ; read content into msg
    mov ah, 3fh
    mov bx, handle
    mov cx, 10
    lea dx, msg
    int 21h

    ; print msg
    lea dx, msg
    mov ah, 09h
    int 21h

    ; close file
    mov ah, 3eh
    int 21h
.exit
2 references
end
```

Lab Task 1:

Read data from console and write to file

1. Print line on screen asking “Enter your name:”
2. Give your name as input, and save it in a local variable
3. Create a new text file
4. Write your name to this file

week6_q1.asm > ...

```
1    .model tiny
2    .data
     0 references
3    str1     db   'Enter your name: $'
4
     0 references
5    max1     db   32
     0 references
6    act1     db   ?
     0 references
7    inp1     db   32 dup('$')
8
     7 references
9    fname    db   'testing.txt',0
     9 references
10   handle   dw   ?
11   .code
12   .startup
13
14       ; WRITE
15       ; YOUR
16       ; CODE
17       ; HERE
18
19   .exit
     3 references
20   end
21
```

Lab task 2

Take substring from 2 files, and write to a third file

1. Part A

1. Create two files, “name.txt” and “id.txt”
2. Write your first name to “name.txt”, and your ID to “id.txt” by taking inputs from the terminal prompt

```
1   .model tiny
2   .data
    0 references
3   fname1  db      'name.txt',0
    1 reference
4   fname2  db      'id.txt',0
    1 reference
5   handle1 dw      ?
    0 references
6   handle2 dw ?
7
    0 references
8   msg1            db 'Anubhav'
    0 references
9   len1            db 06h
    0 references
10  msg2            db '2021PHXP0426P'
    0 references
11  len2            db 0dh
12  .code
13  .startup
14
15          ; WRITE
16          ; YOUR
17          ; CODE
18          ; HERE
19
20
21  .exit
    4 references
22  end
```

2. Part B

1. Create 3rd file “splice.txt”
2. Write a string in a file by concatenating the two strings in the following form :
3. Write the new string in the file called

Example: If name = "Anubhav" & id = "2021PHXP0426P"

then string = "2021PHXP0426PAnubhav"

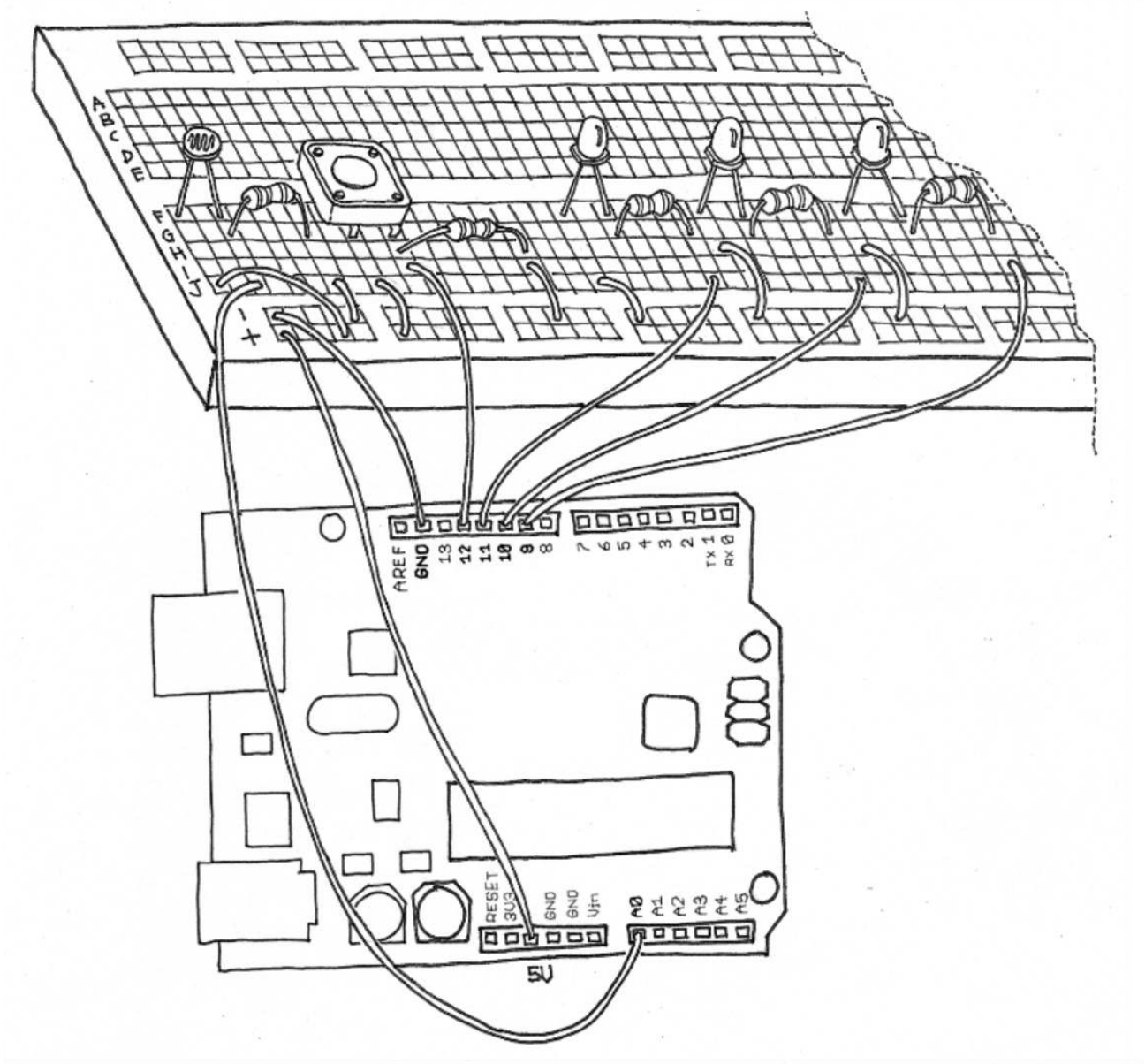
week6_q2_partb.asm > end

```
1    .model tiny
2    .data
     0 references
3    fname1  db 'name.txt',0
     1 reference
4    handle1 dw ?
     1 reference
5    fname2  db  'id.txt',0
     0 references
6    handle2 dw ?
     0 references
7    fname3  db  'splice.txt',0
     0 references
8    handle3 dw  ?
9
     0 references
10   part1       db  8 dup('$')
     0 references
11   part2       db 6 dup('$')
12
13   .code
14   .startup
15
16       ; WRITE
17       ; YOUR
18       ; CODE
19       ; HERE
20
21   .exit
     4 references
22   end
```


BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
ज्ञानं परमं बलम्

CS/EEE/INSTR F241
Microprocessor Programming and Interfacing

Lab 7 - Advanced Operations using Interrupts



Dr. Vinay Chamola and Anubhav Elhence

A complicated Example:

Write an ALP that does the following:

(1) Display the string “Enter 10 character long User Name” and goes to the next line

(2) Takes in the user-entered string of 10 characters and compares with the user name value already stored in memory

(3) If there is no match it should exit saying “wrong Username”

(4) If there is a match it should display the string “Enter 5 character long Password” and goes to the next line

(5) Takes in the password entered by the user and compares it with the password already stored in memory

(6) If there is no match it should exit’

(7) If there is a match it should display “Hello <Username>” where <Username> is replaced by the actual username of the person.

Sample Output:

```
-g 01af
enter 10 character long User Name:
anub@g.com
enter 5 character long password:
******
hello anub@g.com
AX=092A BX=0000 CX=0000 DX=0272 SP=FFFE BP=0000 SI=0230 DI=0235
DS=0863 ES=0863 SS=0863 CS=0863 IP=01AF NV UP EI PL NZ NA PO NC
0863:01AF 8D1E0902          LEA     BX,[0209]                   DS:0209=6E65
```

```
; This Assembly Language Program (ALP) checks the entered username and password, and displays a custom message accordingly.

.model tiny

.data

; Data section contains the messages, the correct username, and password for comparison.

msg1 db "enter 10 character long User Name: $" ; Message 1: Prompt to enter the username

usn1 db "anub@g.com$" ; Correct username for comparison

max1 db 20 ; Maximum length for input

act1 db ? ; Placeholder for action

inp1 db 20 dup("$") ; Buffer to store user's input for username

msg2 db "enter 5 character long password:$" ; Message 2: Prompt to enter the password
```

```
pass1 db "oscar" ; Correct password for comparison

inp2 db 30 dup("$") ; Buffer to store user's input for password

msg3 db "hello $" ; Message 3: Greeting message when both inputs are correct

msg4 db "wrong username$" ; Message 4: Wrong username input

msg5 db "wrong password$" ; Message 5: Wrong password input

nline db 0ah, 0dh, "$" ; New line characters

.code

.startup

; Display message 1 on the screen and go to the next line.

    lea dx, msg1

    mov ah, 09h

    int 21h

; Add a new line after the message.

    lea dx, nline
```

```
    mov ah, 09h
    int 21h
; Take input from the user and store it in inp1.
    lea dx, max1
    mov ah, 0ah
    int 21h
; Compare the entered username with the stored username.
 cld
    lea di, inp1
    lea si, usn1
    mov cx, 11
 repe cmpsb
    jcxz l1
; If the username is incorrect, display the "wrong username" message and exit.
    lea dx, nline
```

```
    mov ah, 09h
    int 21h
    lea dx, msg4
    mov ah, 09h
    int 21h
    mov ah, 4ch
    int 21h
; If the username is correct, display the "enter password" message.
    l1:
    lea dx, nline
    mov ah, 09h
    int 21h
    lea dx, msg2
    mov ah, 09h
    int 21h
```

```
    lea dx, nline

    mov ah, 09h

    int 21h

; Take password input from the user, masking the characters.

    mov cx, 6

    lea di, inp2

    l2:

    mov ah, 08h

    int 21h

    mov [di], al

    mov dl, '*'

    mov ah, 02h

    int 21h

    inc di

    dec cx

    jnz l2
```

```
; Compare the entered password with the stored password.

 cld

    mov cx, 6

    lea di, inp2

    lea si, pass1

 repe cmpsb

    jcxz l3

; If the password is incorrect, display the "wrong password" message and exit.

    lea dx, nline

    mov ah, 09h

    int 21h

    lea dx, msg5

    mov ah, 09h

    int 21h

    mov ah, 4ch
```

```
    int 21h

; If the password is correct, display the greeting message and
the username.

    l3:

    lea dx, nline

    mov ah, 09h

    int 21h

    lea dx, msg3

    mov ah, 09h

    int 21h

    lea dx, usn1

    mov ah,09h

    int 21h

    lea dx, nline

    mov ah, 09h

    int 21h
```

```
; lea bx, msg2

; lea bx, msg2

; lea bx, msg2

; lea bx, msg2

.exit

end
```

Lab Task

Task 1: Take the above problem and modify the ALP such that instead of taking username and password in the .data section, we take it from the user.txt and pswd.txt file.

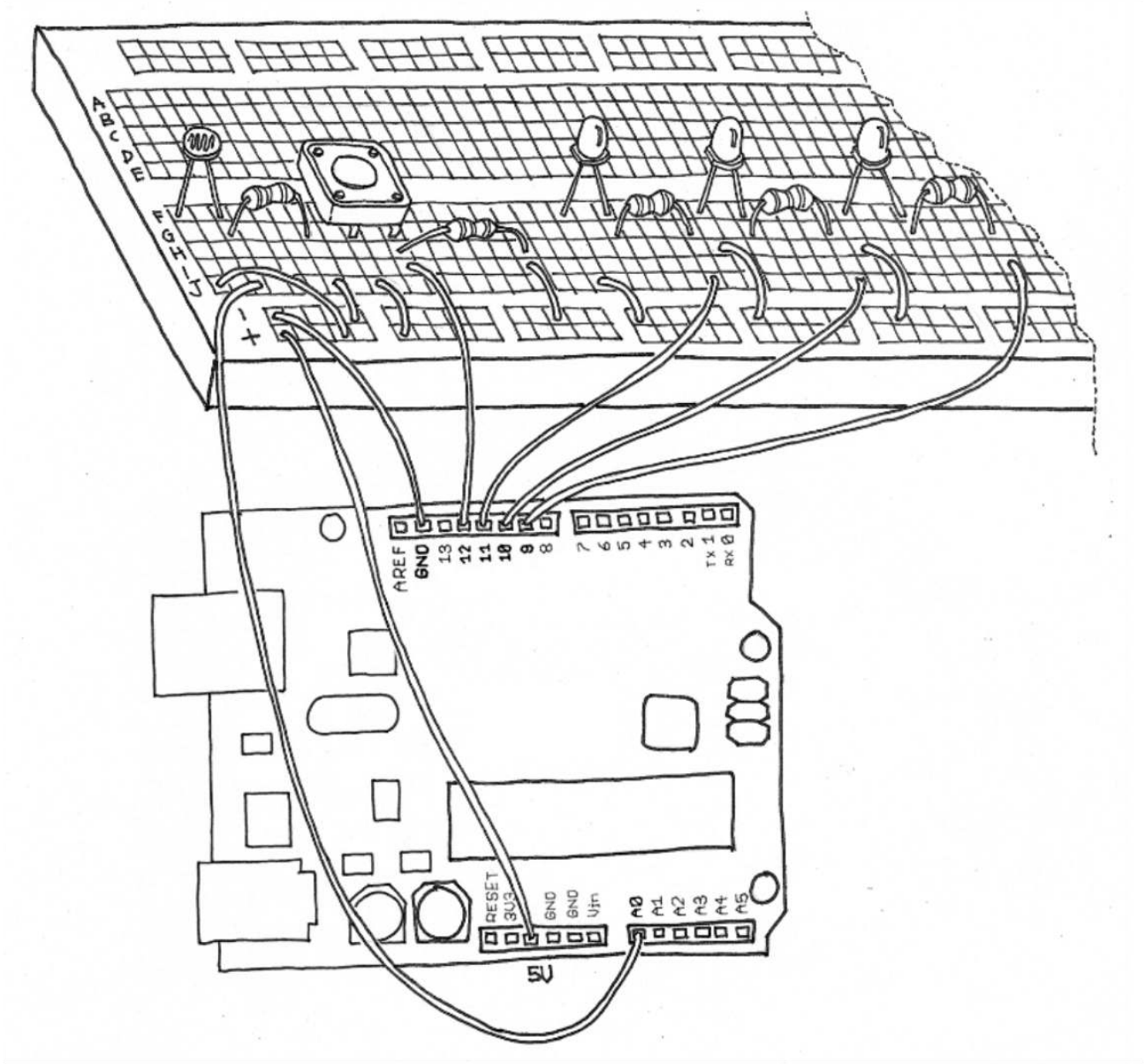
The user.txt file contains 10 character long username with 11th character being '$' and pswd.txt file contains 5 character long password with 6th character being '$'.

Task 2: Now modify the ALP in Task 1, such that the username and password can be of variable length with max number of characters being 20h (i.e. 32 in decimal). The username and password have to be picked from user.txt and pswd.txt file respectively.


BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
ज्ञानं परमं बलम्

CS/EEE/INSTR F241
Microprocessor Programming and Interfacing

Lab 8 - BIOS Interrupts for Display



Dr. Vinay Chamola and Anubhav Elhence

BIOS Interrupts

BIOS interrupt calls are a facility that operating systems and application programs use to invoke the facilities of the Basic Input/Output System. DOS is one of the Operating Systems that BIOS can load. DOS and BIOS interrupts are used to perform some very useful functions, such as displaying data to the monitor, reading data from the keyboard, etc. BIOS function calls allow more control over the video display than do the DOS function calls.

DOS interrupts are executed using the command INT 21H which generates the software interrupt 0x21 (33 in decimal), causing the function pointed to by the 34th vector in the interrupt table to be executed, which is typically an MS-DOS API call.

Similarly, BIOS interrupts are executed using the command INT 10H

There are four main aspects to Video Display

- **Setting an appropriate video mode (or resolution, as you know it)**
- **Reading/Setting the cursor position**
- **Reading/Writing an ASCII character at a given cursor position**
- **Working at the pixel level on the screen (for e.g., drawing a line, square on the screen**

We can choose what action to perform by identifying the interrupt option type, which is the value stored in register AH and providing whatever extra information that the specific option requires. We shall only discuss the important interrupt types in the next section. However, additional interrupts are provided at the end for your benefit.

Purpose: Set Display mode

Input: AH = 0 AL = desired video mode

These video modes are supported:

00h - text mode. 40x25. 16 colours. 8 pages.

03h - text mode. 80x25. 16 colours. 8 pages.

12h - graphical mode. 80x25. 256 colours. 720x400 pixels. 1 page.

Note though 8 pages we always use only the first page

Output

Mode updated

Display cleared

Example: Program Segment to set video mode

```
4    .code
5    .startup
6
7        MOV AH, 00H
8        MOV AL, 12h
9        INT 10H
```

Notice how your display is visible only for a brief second and the program terminates when you do this.

To hold the display we use a Blocking Function.

So, before .exit statement we have to specify a blocking function

e.g. of a blocking function

```
4    .code
5    .startup
6
7        MOV AH, 00H
8        MOV AL, 12h
9        INT 10H
10       mov ah,07h
11       x1: int 21h
12       cmp al,'%'
13       jnz x1
```

System will then retain programmed display mode until the % key is pressed.

Purpose: Set cursor position.

Input:

AH = 02H

DH = row.

DL = column.

BH = page number (0...7). Usually 0

Example: Program Segment to set cursor position

```
4    .code
5    .startup
6
7        MOV AH, 02H
8        MOV DL, 40
9        MOV DH, 12
10       MOV BH, 0
11       INT 10H
```

Purpose: Write character at cursor position

Input:

AH = 09h

AL = character to display.

BH = page number.

BL = attribute.

CX = number of times to write a character.

Output:

Character displayed at current cursor position CX number of times.

Attribute

The attribute byte is used to specify the foreground and background of the character displayed on the screen.

Bits 2-1-0 represent the foreground colour

Bit 3 represents the intensity of foreground colour (0-low , 1-high intensity)

Bits 6-5-4 represent the background colour

Bit 7 is used for blinking text if set to 1

The 3 bit colour code (with their high intensity counterparts (if bit3 is 1) is

```
000 -black (gray)
001 -blue (bright blue)
010 -green (bright green)
011 -cyan (bright cyan)
100 -red (bright red)
101 -magenta (bright magenta)
110 -brown (yellow)
111 -white (bright white)
```

Example Code:

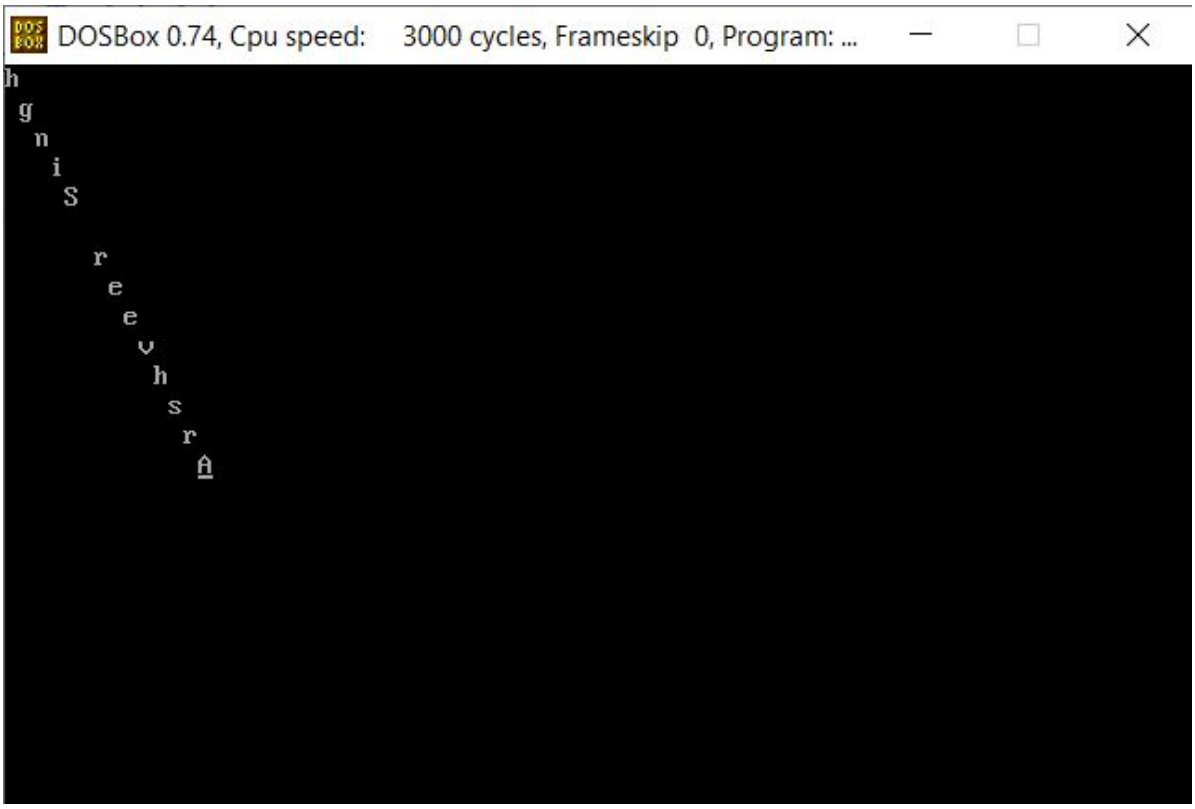
Example 1: Write your name at cursor position (20, 20) in blue blinking text with a black background. Use display mode 03H or Text VGA mode.

```
; This code is written in 16-bit x86 assembly language and uses BIOS interrupt calls for displaying a string
; with custom vertical spacing on the screen. It requires an assembler like NASM for compilation.

.model tiny ; Set memory model to tiny (code and data in one segment)
.386 ; Target 80386 processor

.data ; Data segment
5 references
inp1 db 'MyName' ; Define a byte array 'inp1' containing the input string 'MyName'
2 references
colmstr dw ? ; Define a word 'colmstr' to store the column position of the next character
1 reference
cnt db 06h ; Define a byte 'cnt' containing the length of the input string (6 characters)

.code ; Code segment
.startup ; Executable code starts here
```

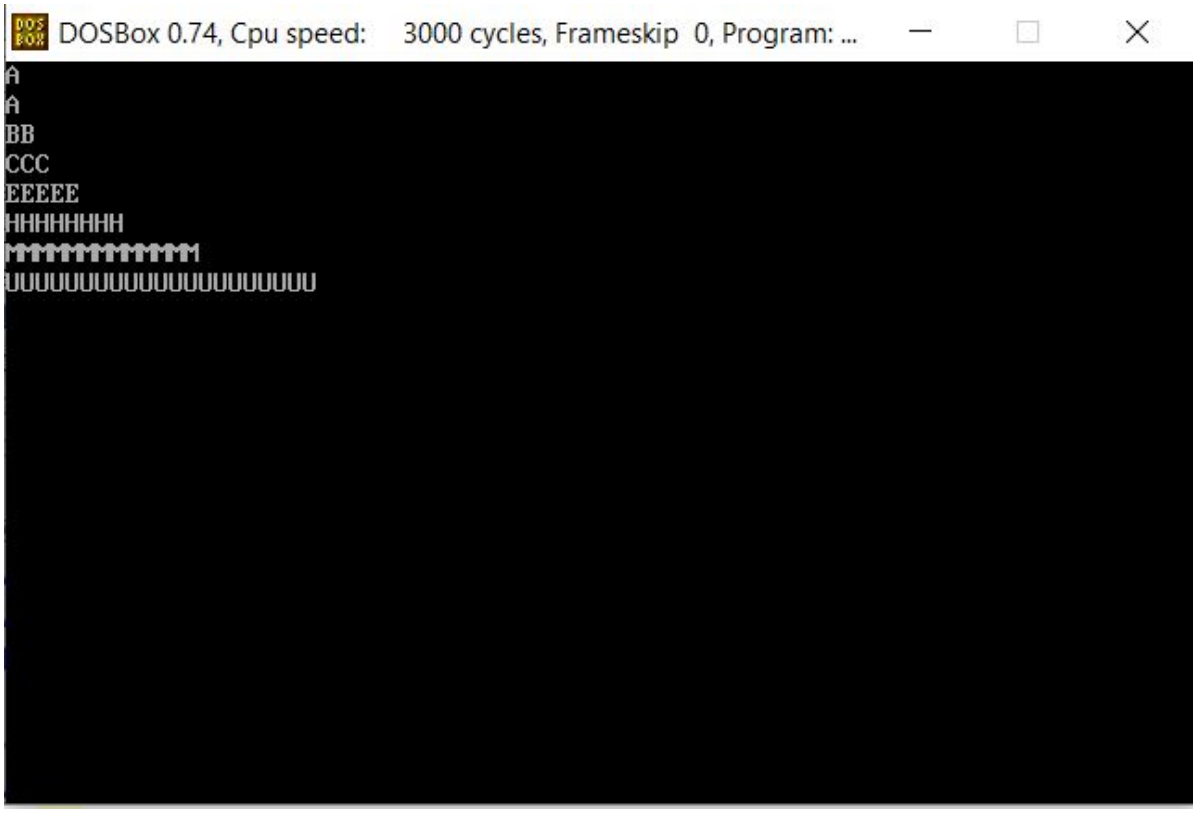
```
14
15      ; SET DISPLAY MODE
16      ; Set video mode to 80x25 text, 16 colors
17      MOV AH, 00H
18      MOV AL, 03H
19      INT 10H
20
21      ; INITIALIZING
22      ; Load the addresses of the input string, length counter, and column position into registers
23      LEA SI, inp1
24      LEA DI, cnt
25      MOV CH, 00h
26      MOV CL, [DI]
27      MOV colmstr, 20 ; Set initial column position to 20
28      LEA DI, colmstr
29
30      ; WRITING CHAR
31      WRITE1:
32      PUSH CX ; Save count value on the stack
33
34      ; SETTING CURSOR POS
35      ; Set the cursor position to row 20 and column specified by colmstr
36      MOV AH, 02H
37      MOV DH, 20
38      MOV DL, [DI]
39      MOV BH, 00
40      INT 10H
41
42      ; Write a single character with custom vertical spacing
43      MOV AH, 09H
44      MOV AL, [SI] ; Load character from input string
45      MOV BH, 00
46      MOV BL, 10001001b ; Set custom vertical spacing
47      MOV CX, 01
48      INT 10H
49      POP CX ; Restore count value from the stack
51        ; CHANGING VERTICES
52        ; Increment the input string pointer, column position, and decrement the length counter
53        INC SI
54        INC WORD PTR[DI]
55        DEC CL
56        JNZ WRITE1 ; Repeat for all characters in the input string
57
58        ; BLOCKING FUNCTION
59        ; Wait for the user to press the '%' key to exit
60        END1:
61        MOV AH, 07H
62        INT 21h
63        CMP AL, "%"
64        JNZ END1
65
66        ; TERMINATE PROGRAM
67        TERM:
68        MOV AH, 4CH ; Exit function
69        INT 21H
70
71  .exit ; Mark the end of the program
      4 references
72  end ; End the program
```

Lab Task

Task 1: Print your full name in reverse diagonal order in white text with a black background. Use display mode 03H or Text VGA mode. For example, if your name is Arshveer Singh it should look like



Task 2: Print the fibonacci sequence vertically where the nth term of the sequence must be repeated n times and must represent the nth term of the alphabet. Print the first 8 terms for convenience. Use the same specifications as given above. Sample output has been provided below:



// ADDITIONAL (Not required to solve tasks)

Purpose: Get Display mode

Input

AH=0Fh

Output

AL=current video mode

AH=number of character columns

BH=page number

Purpose: Get cursor position and size

Input

AH = 3

BH = page number (usually 0)

Output

DH = row.

DL = column.

CH = cursor start line.

CL = cursor bottom line.

Purpose: Set cursor size

Input:

AH = 01h

CH = cursor start line (bits 0-4) and options (bits 5-7).

CL = bottom cursor line (bits 0-4).

Output:

Cursor size changed.

Purpose: Read character at Cursor position

Input:

AH = 08h

Bh = 0 (page no.)

Output:

Error if CF = 1, AX = error code (6)

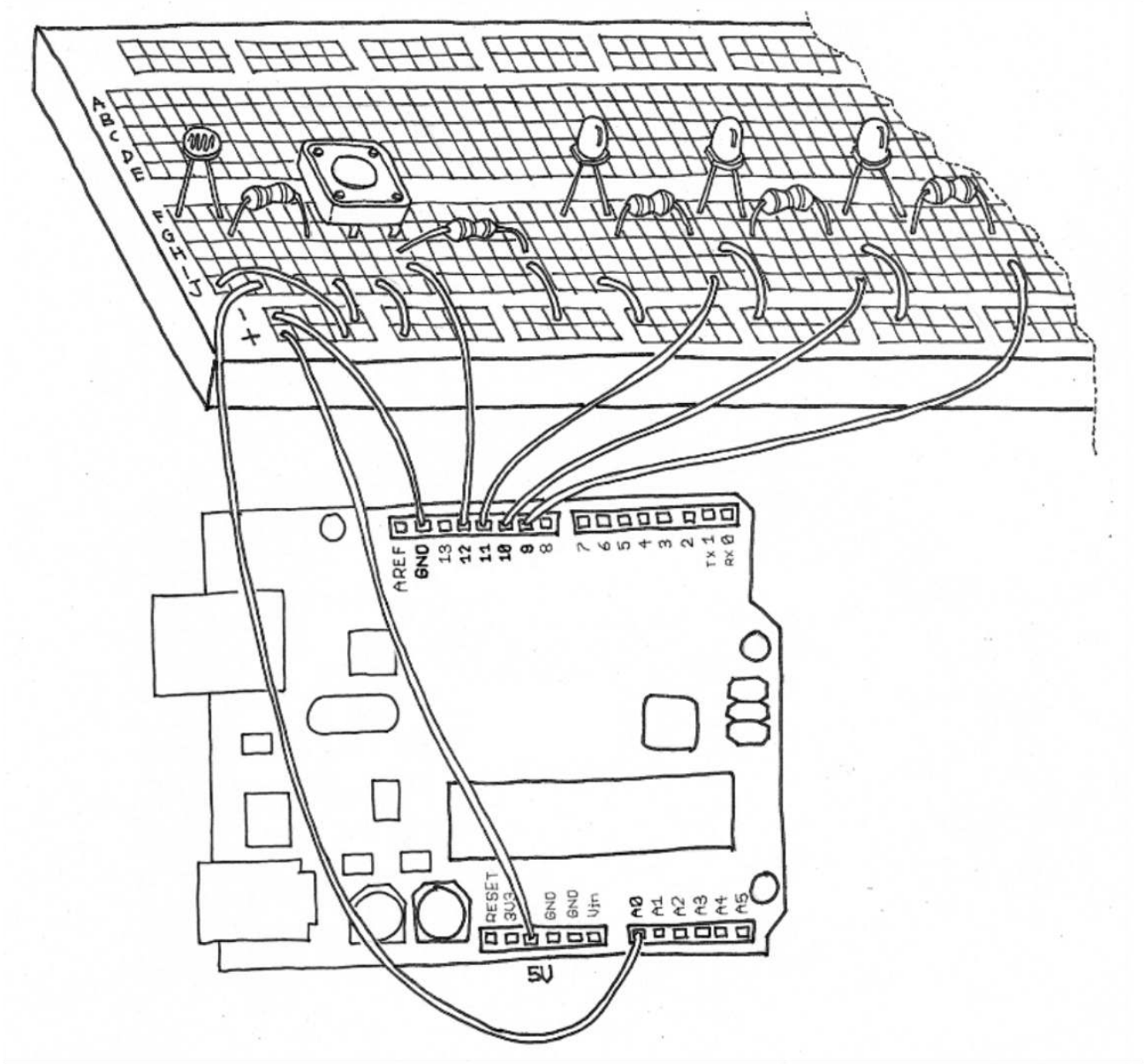
AH = attribute.

AL = character.


BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
ज्ञानं परमं बलम्

CS/EEE/INSTR F241
Microprocessor Programming and Interfacing

Lab 9 - Advanced BIOS Interrupts for Display



Dr. Vinay Chamola and Anubhav Elhence

BIOS Interrupt 10h

There are four main aspects to Video Display

Setting an appropriate video mode (or resolution, as you know it)

Reading/Setting the cursor position

Reading/Writing an ASCII character at a given cursor position

Working at the pixel level on the screen (for e.g., drawing a line, square on the screen

We can choose what action to perform by identifying the interrupt option type, which is the value stored in register AH and providing whatever extra information that the specific option requires. We shall only discuss the important interrupt types in the next section. However, additional interrupts are provided at the end for your benefit.

Filling Up a Pixel

Input:

AH = 0Ch

AL = pixel color //SAME ATTRIBUTE FORMAT AS PREVIOUS LAB

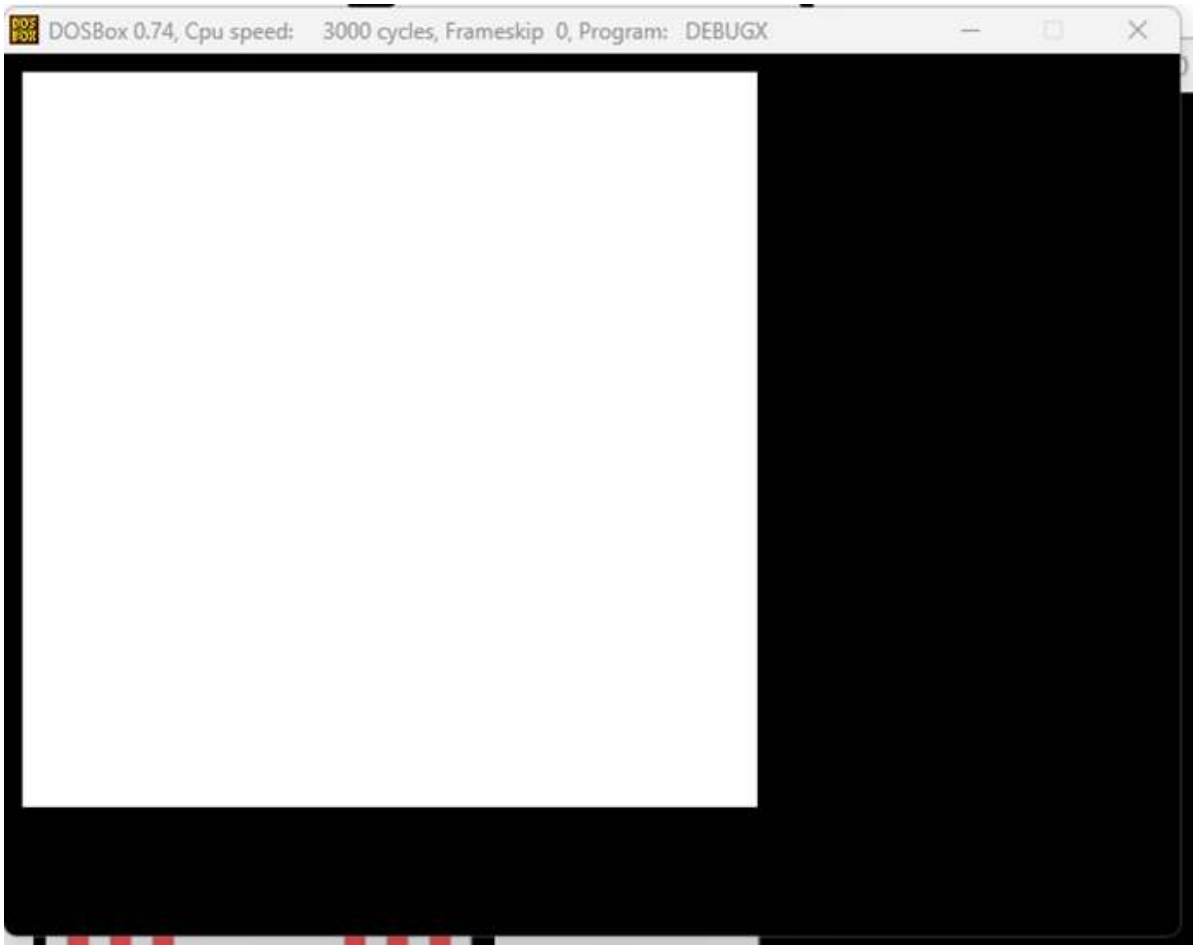
CX = column.

DX = row

```
4    .code
5    .startup
6
7        mov ah, 0
8        Mov al, 12h
9        int 10h ; set graphics video mode.
10
11       mov ah, 0ch
12       mov al, 00001100b
13       mov cx, 10
14       mov dx, 20
15       int 10h ; set pixel.
16
17       mov ah,07h
18       x1: int 21h
19       cmp al,'%'
20       jnz x1
21
```

Follow Along Example:

Write an ALP to print a white square of pixel 400x 400 and starting from pixel (10,10)


DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DEBUGX

- Set display mode to 640x480 with 16 colors:

```
14      ; Set display mode to 640x480 16 colors
15      MOV AH, 00H
16      MOV AL, 12H
17      INT 10H
```

- Set cursor position to (25,25) for graceful exit

```
19      ; Set cursor position to (20, 20)
20      MOV AH, 02H
21      MOV DH, 20
22      MOV DL, 20
23      MOV BH, 00
24      INT 10H
```

- Initialize Parameters for box drawing

```
26      ; Initialize parameters for box drawing
27      MOV rowstr, 10
28      MOV rowend, 410
29      MOV colmstr, 10
30      MOV colmend, 210
31      MOV cnt, 00
```


- Paint the white Box code

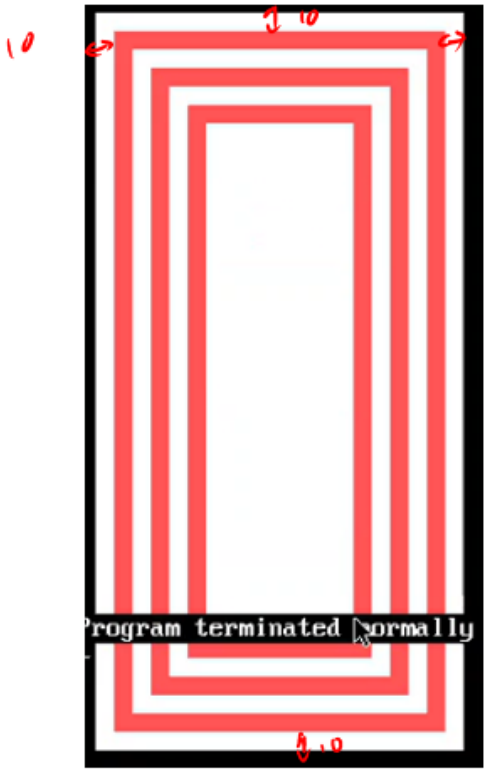
```
; Paint the first box
PAINT1:
MOV SI, rowstr ; Row start
COLM1:
MOV CX, colmend ; Column end
ROW1:
DEC CX
MOV DI, CX
PUSH CX
MOV AH, 0Ch
MOV AL, 1111b ; Color for first box
MOV CX, DI
MOV DX, SI
INT 10h
POP CX
CMP CX, colmstr ; Column start
JNZ ROW1
INC SI
MOV AX, rowend ; Row end
CMP SI, AX
JNZ COLM1
```

- Blocking Function

```
END1:
MOV AH, 07H
INT 21h
CMP AL, "%"
JNZ END1 ; Loop until '%' is received

TERM:
MOV AH, 4CH ; Terminate program
INT 21H
```

Follow along example

- Write an ALP to print the following pattern in 12h mode graphic mode



8

- Set display mode to 640x480 with 16 colors:

```
14      ; Set display mode to 640x480 16 colors
15      MOV AH, 00H
16      MOV AL, 12H
17      INT 10H
```

- Set cursor position to (20,20)

```
19      ; Set cursor position to (20, 20)
20      MOV AH, 02H
21      MOV DH, 20
22      MOV DL, 20
23      MOV BH, 00
24      INT 10H
```

- Initialize Parameters for box drawing

```
26      ; Initialize parameters for box drawing
27      MOV rowstr, 10
28      MOV rowend, 410
29      MOV colmstr, 10
30      MOV colmend, 210
31      MOV cnt, 00
```

- Painting the first box: The code uses a nested loop structure to draw the box by iterating through rows and columns, setting the color, and calling the BIOS interrupt 10h to paint the pixel.

```
33      ; Paint the first box
34      PAINT1:
35      MOV SI, rowstr ; Row start
36      COLM1:
37      MOV CX, colmend ; Column end
38      ROW1:
39      DEC CX
40      MOV DI, CX
41      PUSH CX
42      MOV AH, 0Ch
43      MOV AL, 1111b ; Color for first box
44      MOV CX, DI
45      MOV DX, SI
46      INT 10h
47      POP CX
48      CMP CX, colmstr ; Column start
49      JNZ ROW1
50      INC SI
51      MOV AX, rowend ; Row end
52      CMP SI, AX
53      JNZ COLM1
```

- Change vertices for the next box:
  After drawing the first box, the code adjusts the row and column start and end points, as well as increments the counter variable to keep track of how many boxes have been drawn.

```
; Change vertices for the next box
LEA SI, rowstr
ADD WORD PTR[SI], 10
LEA SI, rowend
SUB WORD PTR[SI], 10
LEA SI, colmstr
ADD WORD PTR[SI], 10
LEA SI, colmend
SUB WORD PTR[SI], 10
LEA SI, cnt
INC BYTE PTR[SI]
```

- Check if the counter has reached 7: If the counter reaches 7 or more, the code jumps to the termination process.

```
MOV AL, 07h
MOV BL, cnt
CMP BL, AL
JGE TERM ; Terminate if cnt >= 7
```

- Paint the second box:
  Similar to the first box, this part paints the second box with a different color using the same nested loop structure

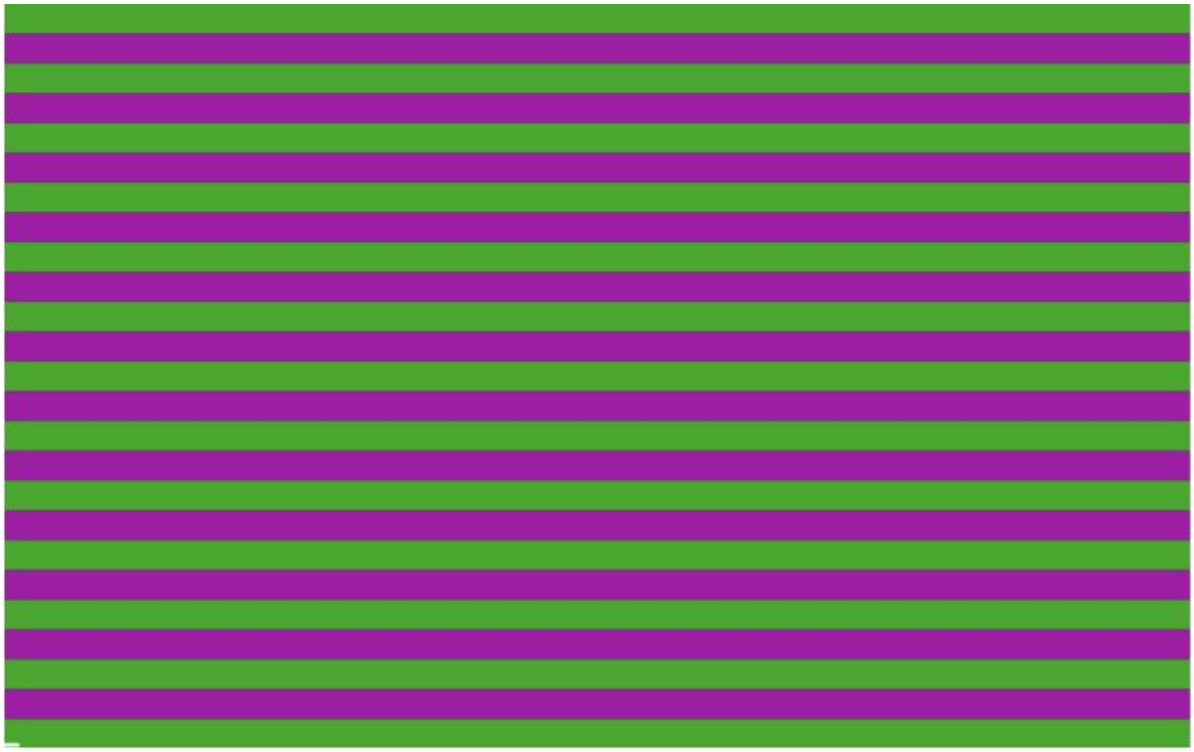
```
; Paint the second box
MOV SI, rowstr ; Row start
COLM2:
MOV CX, colmend ; Column end
ROW2:
DEC CX
MOV DI, CX
PUSH CX
MOV AH, 0Ch
MOV AL, 1100b ; Color for second box
MOV CX, DI
MOV DX, SI
INT 10h
POP CX
CMP CX, colmstr ; Column start
JNZ ROW2
INC SI
MOV AX, rowend ; Row end
CMP SI, AX
JNZ COLM2
```

- Change vertices for the next box: The code again adjusts the row and column start and end points and increments the counter variable.

```
; Change vertices for the next box
LEA SI, rowstr
ADD WORD PTR[SI], 10
LEA SI, rowend
SUB WORD PTR[SI], 10
LEA SI, colmstr
ADD WORD PTR[SI], 10
LEA SI, colmend
SUB WORD PTR[SI], 10
LEA SI, cnt
INC BYTE PTR[SI]
```

Lab Task

Write an ALP that takes in a single user input from the keyboard. The key pressed by the user must not be displayed. The program should compare the user input with the 12th byte in the file lab1.txt and if the user input is equal to the 12th byte (counting from one) in the file then the following pattern must be displayed on the screen.



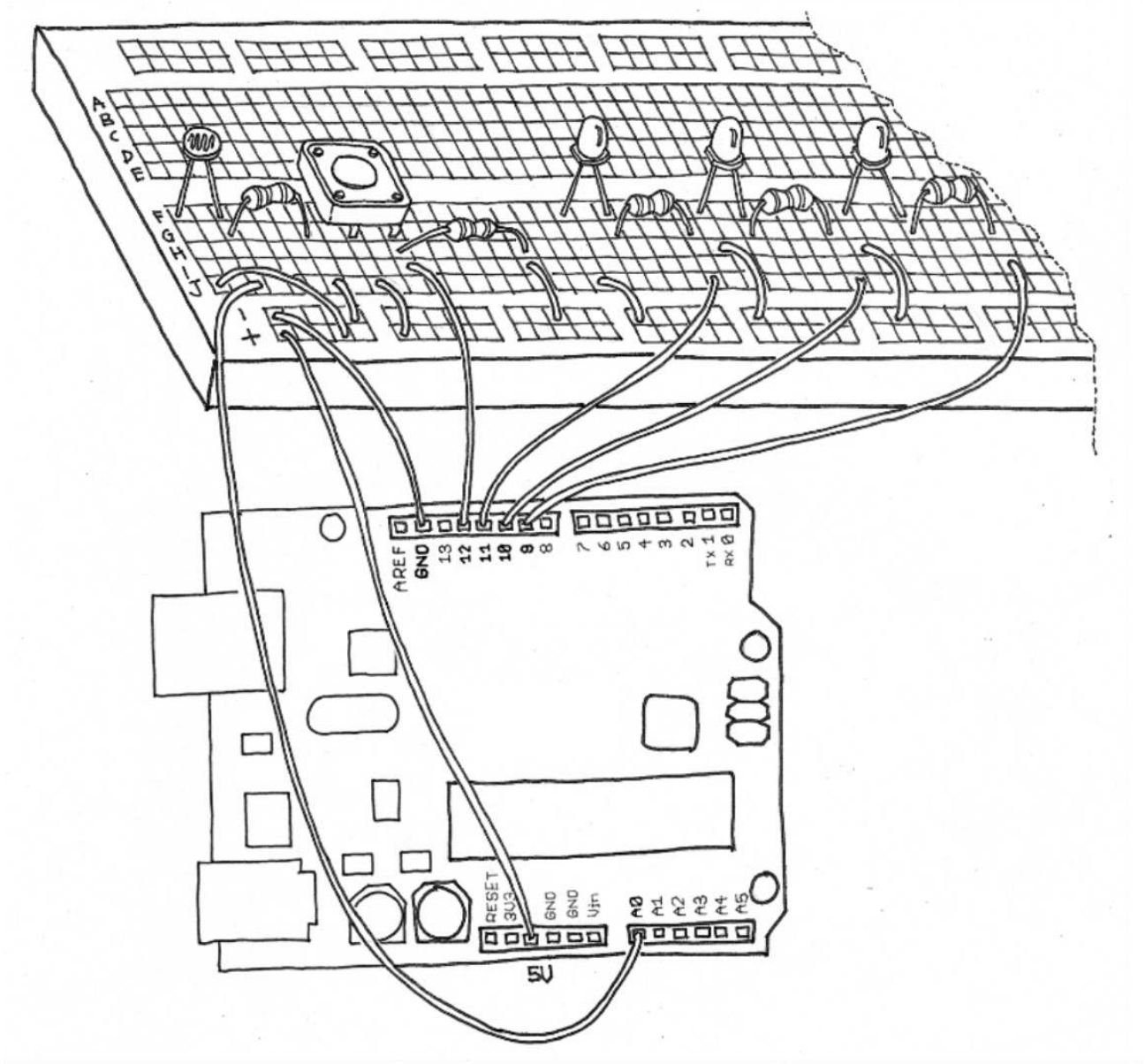
Else the following pattern must be displayed:

The contents of the file can be written by the student to try out both possible displays given.The file must be placed in the BIN folder of MASM.


BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
ज्ञानं परमं बलम्

CS/EEE/INSTR F241
Microprocessor Programming and Interfacing

Lab 9 - Advanced BIOS Interrupts for Display



Dr. Vinay Chamola and Anubhav Elhence

What is a Procedure?

Procedure is a part of code that can be called from your program in order to make some specific task. Procedures make program more structural and easier to understand. Generally procedure returns to the same point from where it was called.

The syntax for procedure declaration:

***name* PROC**

; here goes the code

; of the procedure ...

RET

***name* ENDP**

(*name* - is the procedure name); The same name should be used for both the PROC and ENDP directives! (This is used to check the correct closing of procedures)

PROC and ENDP are compiler directives, so they are not assembled into any real machine code. The compiler just remembers the address of the procedure.

The CALL Instruction

The CALL instruction is used to call a procedure. The RET instruction is used to return to the operating system. The same instruction is used to return from a procedure (actually, the operating system sees your entire program as a special procedure).

For example, in the code below, the program calls the procedure m1, performs MOV BX, 5, and proceeds to the next instruction (MOV AX, 2)

```
CALL m1

MOV AX, 2

RET             ; Return to the OS

m1 PROC     ; Define the procedure 'm1'

MOV BX, 5

RET             ; Return to Caller.

m1 ENDP
```

There are several ways to pass parameters to a procedure. The easiest way to pass parameters is by using registers. Here is another example of a procedure that receives two parameters in AL and BL registers, multiplies these parameters, and returns the result in AX register. Since m2 is called four times, the final result in AX will be 2^4 (1 or 10H)

```
MOV AL, 1

MOV BL, 2

CALL m2

CALL m2

CALL m2

CALL m2

RET             ; Return to the OS

m2 PROC

MUL BL              ; The product of AL, BL is stored in AX

RET             ; Return to the Caller

m2 ENDP
```

The Stack

The Stack is an area of memory for keeping temporary data. The stack is used by the CALL instruction to keep return address for procedure, and the RET instruction gets this value from the stack and returns to that offset.

This also happens when INT instruction calls an interrupt (Recall INT 21h and INT 10h!). It stores the code segment and offset in the stack flag registe. Similar to RET, the IRET instruction is used to return from interrupt call.

The PUSH and POP Instructions

The stack is a LIFO data structure (Last In, First Out) can be accessed to store or retrieve data using these two instructions-

PUSH

PUSH - stores a 16 bit value (from a register or memory location) in the stack.

Syntax:

```
PUSH REG          ; AX, BX, DI, SI etc.

PUSH SREG         ; DS, SS, ES etc.

PUSH memory       ; [BX], [BX+SI] etc.

PUSH immediate ; 5. 3Fh, 10001000b etc.
```

POP

POP - gets 16 bit value from the stack and stores it in a register or a memory location.

Syntax:

```
POP REG         ; AX, BX, DI, SI etc.

POP SREG        ; DS, SS, ES etc.

POP memory      ; [BX], [BX+SI] etc.
```

The following example shows how the stack can be used to swap the values in the registers AX and BX. Notice the order of registers in the pop operation! (What would happen if we perform POP BX first?)

```
MOV AX, 1212h

MOV BX, 3434h

PUSH AX

PUSH BX

POP AX

POP BX
```

Tasks to be Completed

1. ***Reverse a string (your first name) in place using only the stack***

Note: Use dw instead of db in the data section since you'll use PUSH and POP

2. ***Write a procedure to calculate nPr, where the parameter n is stored in BX, the parameter r is stored in DX, and the result is stored in AX.***

Hint:

- **First, write a recursive procedure to calculate the factorial of a number**
- **nPr = n!/(n-r)!**
- **So, 5P2 = 5!/3! = 20 (14h)**

Sample Input:

BX = 05h, DX = 02h

Sample Output:

AX = 14h


BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
ज्ञानं परमं बलम्