Debug

- **A program included with DOS that allows a programmer to monitor the execution of a program for debugging purposes.**
- **Using Debug:**
  - **Enter Debug**

    A:>DEBUG<enter>

    -
  - **Exit Debug**

    -Q<enter>

    A:>

Debug

- **Displaying registers**

-R<enter>
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D00 ES=0D00 SS=0D00 CS=0D00 IP=0100 NV UP DI PL NZ NA PO NC
0D00:0100 B80100 MOV AX,0001

- **Modifying registers**

  -R CX:<enter>
  CX 0000
  :0009<enter>
  -R CX<enter>
  CX 0009
  :<enter>
  -

Debug

- **Assemble command – allows the programmer to enter assembly language instructions into memory.**

-A 100<enter>

0B3C:0100 MOV AX,1<enter>

0B3C:0103 MOV BX,2<enter>

0B3C:0106 ADD AX,BX<enter>

0B3C:0108 INT 3<enter>

0B3C:0109<enter>

-

Debug

- **Unassemble command - allows the programmer to display the machine code in memory along with their assembly language instructions.**

-U 100 L1<enter>

0B3C:0100 B80100 MOV AX,1

-U 100 103

0B3C:0100 B80100 MOV AX,1

0B3C:0103 BB0200 MOV BX,2

-

Debug

- **Go command – allows the programmer to execute instructions found between two given addresses.**

```
-G=100 108<enter>
AX=0004 BX=0003 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B3C ES=0B3C SS=0B3C CS=0B3C IP=0108 NV UP EI PL NZ NA PO NC
0B3C:0108 CC INT 3
```

Debug

- **Trace command - allows the programmer to trace through the execution of a program one or more instructions at a time to verify the effect the program has on registers and/or data.**

```
-T=100 2<enter>
AX=0001 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B3C ES=0B3C SS=0B3C CS=0B3C IP=0103 NV UP EI PL NZ NA PO NC
0B3C:0103 BB0200 MOV BX,0002

AX=0001 BX=0003 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B3C ES=0B3C SS=0B3C CS=0B3C IP=0106 NV UP EI PL NZ NA PO NC
0B3C:0106 01D8 ADD AX,BX
-
```

Debug

- **Dump command (D) - allows the programmer to examine the contents of memory.**
- **Fill command (F) - allows the programmer to fill memory with data.**
- **Enter command (E) - allows the programmer to modify memory content.**

```
-F 100 LF 00<enter>
-D 100 LF
0B3C:0100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
-F 110 11F 20
-D 100 11F
0B3C:0100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0B3C:0110 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
-F 120 LF 20
-E 120 'John Smith'
-D 120 LF
0B3C:0120 4A 6F 68 6E 20 53 6E 69 74 68 20 20 20 20 20 20 John Smith
-
```

Debug

- **Loading programs from a specific file requires two commands, the Name command, N, and the Load command, L.**

```
-N A:\PROG1.EXE
-L
```

- **Loading programs upon entering Debug.**

```
C:\DEBUG A:\PROG1.EXE
```

Debug

- **Links to useful websites:**
  - **DEBUG/ASSEMBLY TUTORIAL by Fran Golden**
    - **http://www.datainstitute.com/debug1.htm**
  - **Rough Guide to Assembly**
    - **http://www.geocities.com/riskyfriends/prog.html**
  - **Paul Hsieh's x86 Assembly Language Page**
    - **http://www.azillionmonkeys.com/qed/asm.html**

Assembly Language Program

- **Series of statements which are either assembly language instructions or directives.**
  - **Instructions are statements like ADD AX,BX which are translated into machine code.**
  - **Directives or pseudo-instructions are statements used by the programmer to direct the assembler on how to proceed in the assembly process.**

Assembly Language Program

- **Statement format:**
  - **[label:] mnemonic [operands][;comments]**
- **Label:**
  - **Cannot exceed 31 characters.**
  - **Consists:**
    - **Alphabetic characters both upper and lower case.**
    - **Digits 0 through 9.**
    - **Special characters (?), (.), (@), (_), and ($).**
  - **The first character cannot be a digit.**
  - **The period can only be used as the first character, but its use is not recommended. Several reserved words begin with it in later versions of MASM.**

Assembly Language Program

- **Label:**
  - Must end with a colon when it refers to an opcode generating instruction.
  - Do not need to end with a colon when it refers to a directive.
- **Mnemonic and operands:**
  - Instructions are translated into machine code.
  - Directives do not generate machine code. They are used by the assembler to organize the program and direct the assembly process.

Assembly Language Program

- **Comments:**
  - **Begin with a “;”.**
  - **Ignored by the assembler.**
  - **Maybe be on a line by itself or at the end of a line:**
    - **;My first comment**
    - **MOV AX,1234H ;Initializing….**
  - **Indispensable to the programmers because they make it easier for someone to read and understand the program.**

Segment Definition

- The CPU has several segment registers:
  - CS (code segment).
  - SS (stack segment).
  - DS (data segment).
  - ES (extra segment).
  - FS, GS (supplemental segments available on 386s, 486s and Pentiums.
- Every instruction and directive must correspond to a segment.
- Normally a program consists of three segments: the stack, the data, and the code segments.

Segment Definition

- **Model definition.**
- **.MODEL SMALL**
  - **Most widely used memory model.**
  - **The code must fit in 64k.**
  - **The data must fit in 64k.**
- **.MODEL MEDIUM**
  - **The code can exceed 64k.**
  - **The data must fit in 64k.**
- **.MODEL COMPACT**
  - **The code must fit in 64k.**
  - **The data can exceed 64k.**
- **MEDIUM and COMPACT are opposites.**

Segment Definition

- **.MODEL LARGE**
  - **Both code and data can exceed 64k.**
  - **No single set of data can exceed 64k.**
- **.MODEL HUGE**
  - **Both code and data can exceed 64k.**
  - **A single set of data can exceed 64k.**
- **.MODEL TINY**
  - **Used with COM files.**
  - **Both code and data must fir in a single 64k segment.**

Segment Definition

- **Segment definition formats:**
  - **Simplified segment definition.**
  - **Full segment definition.**
- **The Simplified segment definition uses the following directives to define the segments:**
  - **.STACK**
  - **.DATA**
  - **.CODE**
  - **These directives mark the beginning of the segments they represent.**

Segment Definition

- **The full segment definition uses the following directives to define the segments:**
  - **Label SEGMENT [options]**

    ;Statements belonging to the segment.

    Label ENDS
  - **The label must follow naming conventions previously discussed.**

Segment Definition

;SIMPLIFIED SEGMENT DEFINITION			;FULL SEGMENT DEFINITION		
.MODEL SMALL					
.STACK 64			STSEG	SEGMENT	
			DB 64 DUP(?)		
			STSEG	ENDS	
.DATA			DTSEG	SEGMENT	
N1	DW	1432H	N1	DW	1432H
N2	DW	4365H	N2	DW	4365H
SUM	DW	0H	SUM	DW	0H
			DTSEG	ENDS	
.CODE			CDSEG SEGMENT		
BEGIN	PROC FAR		BEGIN	PROC FAR	
				ASSUME CS:CDSEG,DS:DTSEG,SS:STSEG	
	MOV AX,@DATA			MOV AX,DTSEG	
	MOV DS,AX			MOV DS,AX	
	MOV AX,N1			MOVAX,N1	
	ADD AX,N2			ADD AX,N2	
	MOV SUM,AX			MOV SUM,AX	
	MOV AH,4CH			MOV AH,4CH	
	INT 21H			INT 21H	
BEGIN	ENDP		BEGIN	ENDP	
	END BEGIN		CDSEG	ENDS	
				END BEGIN	

Program Termination

- **With PC:**
  - **MOV AH,4CH**

    INT 21H
  - **Always return control to the OS.**

Text Editors

- **Use the following text editors to write your programs.**
  - Notepad (Windows).
  - Edit (DOS).
  - Or any other editor capable of generating ASCII files.

DOS and BIOS Interrupts

- **DOS and BIOS interrupts are used to perform some very useful functions, such as displaying data to the monitor, reading data from keyboard, etc.**
- **They are used by identifying the interrupt option type, which is the value stored in register AH and providing, whatever extra information that the specific option requires.**

BIOS Interrupt 10H

- **Option 0H – Sets video mode.**
- **Registers used:**
  - **AH = 0H**
  - **AL = Video Mode.**
    - **3H - CGA Color text of 80X25**
    - **7H - Monochrome text of 80X25**
- **Ex:**
  - **MOV AH,0**
  - **MOV AL,7**
  - **INT 10H**

BIOS Interrupt 10H

- **Option 2H – Sets the cursor to a specific location.**
- **Registers used:**
  - **AH = 2H**
  - **BH = 0H selects Page 0.**
  - **DH = Row position.**
  - **DL = Column position.**
- **Ex:**
  - **MOV AH,2**
  - **MOV BH,0**
  - **MOV DH,12**
  - **MOV DL,39**
  - **INT 10H**

BIOS Interrupt 10H

- **Option 6H – Scroll window up. This interrupt is also used to clear the screen when you set AL = 0.**
- **Registers used:**
  - **AH = 6H**
  - **AL = number of lines to scroll.**
  - **BH = display attribute.**
  - **CH = y coordinate of top left.**
  - **CL = x coordinate of top left.**
  - **DH = y coordinate of lower right.**
  - **DL = x coordinate of lower right.**

BIOS Interrupt 10H

- **Clear Screen Example:**
  - MOV AH,6
  - MOV AL,0
  - MOV BH,7
  - MOV CH,0
  - MOV CL,0
  - MOV DH,24
  - MOV DL,79
  - INT 10H
- **The code above may be shortened by using AX, BX and DX registers to move word size data instead of byte size data.**

DOS Interrupt 21H

- **Option 1 – Inputs a single character from keyboard and echoes it to the monitor.**
- **Registers used:**
  - **AH = 1**
  - **AL = the character inputted from keyboard.**
- **Ex:**
  - **MOV AH,1**
  - **INT 21H**

DOS Interrupt 21H

- **Option 2 – Outputs a single character to the monitor.**
- **Registers used:**
  - **AH = 2**
  - **DL = the character to be displayed.**
- **Ex:**
  - **MOV AH,2**
  - **MOV DL,'A'**
  - **INT 21H**

DOS Interrupt 21H

- **Option 9 – Outputs a string of data, terminated by a $ to the monitor.**
- **Registers used:**
  - **AH = 9**
  - **DX = the offset address of the data to be displayed.**
- **Ex:**
  - **MOV AH,09**
  - **MOV DX,OFFSET MESS1**
  - **INT 21H**

DOS Interrupt 21H

- **Option 4CH – Terminates a process, by returning control to a parent process or to DOS.**
- **Registers used:**
  - **AH = 4CH**
  - **AL = binary return code.**
- **Ex:**
  - **MOV AH,4CH**
  - **INT 21H**

80386

- General purpose processor optimized for multitasking operating systems.
- Supports 32 bits address and data buses.
- Capable of addressing 4 gigabytes of physical memory and 64 terabytes of virtual memory.

Registers

- General purpose registers.
  - There are eight 32 bits registers (EAX, EBX, ECX, EDX, EBP, EDI, ESI, and ESP).
  - They are used to hold operands for logical and arithmetic operations and to hold addresses.
  - Access may be done in 8, 16 or 32 bits.
  - There is no direct access to the upper 16 bits of the 32 bits registers.
  - Some instructions incorporate dedicated registers in their operations which allows for decreased code size, but it also restricts the use of the register set.

Registers

- **Segment registers.**
  - There are six 16 bits registers (CS, DS,ES,FS,GS, and SS).
  - They are used to hold the segment selector.
  - Each segment register is associated with a particular kind of memory access.

Registers

- **Other registers.**
  - EFLAGS controls certain operations and indicates the status of the 80836 (carry, sign, etc).
  - EIP contains the address of the next instruction to be executed.
  - The E prefix in all 32 bits registers names stands for extended.

80386 Architecture

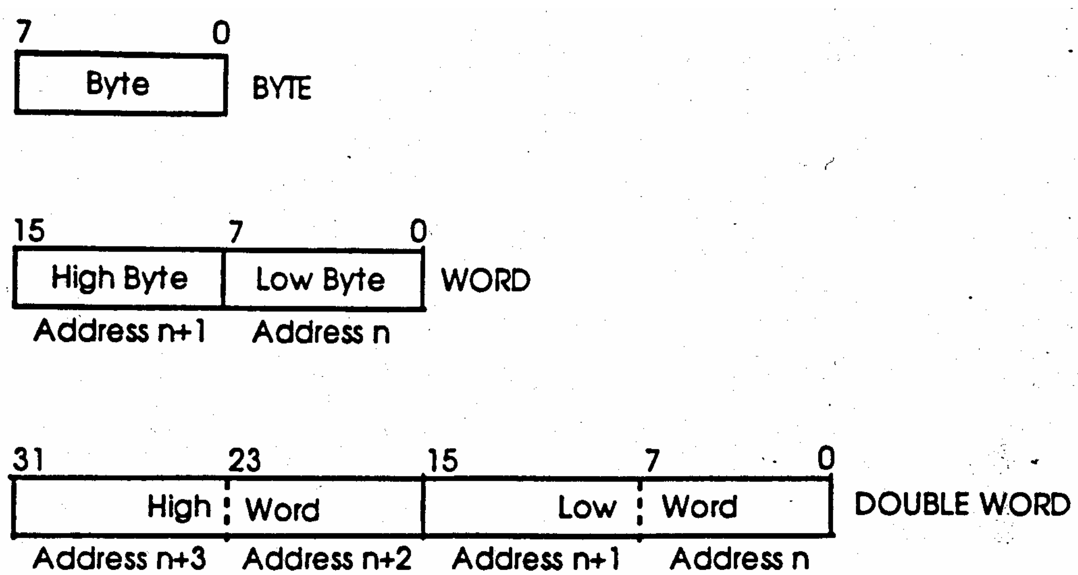
EAX		AX			Accumulator
			AH	AL	
EBX		BX			Base Index
			BH	BL	
ECX		CX			Count
			CH	CL	
EDX		DX			Data
			DH	DL	
EDI		DI			Destination Index
ESI		SI			Source Index
ESP		SP			Stack Pointer
EBP		BP			Base Pointer
EIP		IP			Instruction Pointer
EFLAGS		FLAGS			Flags
		CS			Code
		DS			Data
		ES			Extra
		SS			Stack
		FS			Supplemental
		GS			

Effective, Segment and Physical Addresses

- **Effective address (EA).**
  - Also called offset.
  - Result of an address computation.
- **Segment address (SA).**
  - Also called segment selectors.
  - Addresses stores in segment registers
- **Physical address (PA).**
  - Location in memory.
  - PA = SA * 16 + EA

Memory Organization

- **Sequence of bytes each with a unique physical address.**
- **Data types:**
  - Byte.
  - Word.
  - Double word.

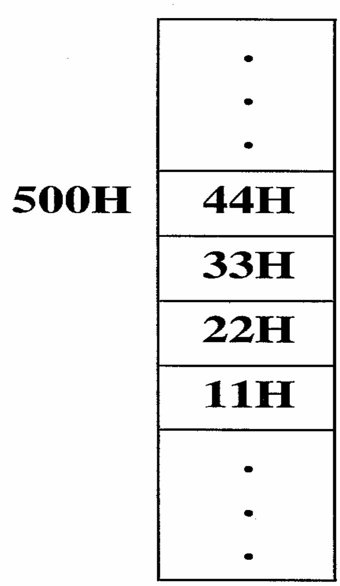


Little Endian Notation

- **The 80386 stores the least significant byte of a word or double word in the memory location with the lower address.**

Assuming EAX = 11223344H

mov ds:[500H], EAX



Constants

- **EQU is used to define constants or to assign names to expressions.**
- **Form:**
  - Name EQU expression.
- **Examples:**
  - PI EQU 3.1415
  - Radius EQU 25
  - Circumference EQU 2*PI*Radius

Variables

- **DB - define byte.**
- **DW - define word.**
- **DD – define double word.**
- **Form:**
  - **Variable Directive oper, . . ,oper**
- **Examples:**
  - **Alpha db ‘ABCDE’**
  - **Alpha2 db ‘A’,’B’,’C’,’D’,’E’**
  - **Alpha3 db 41h,42h,43h,44h,45h**
  - **Word1 dw 3344h**
  - **Double_word1 dd 44332211h**

Addressing Modes

- These are the different ways in which data may be accessed by the microprocessor.
  - Immediate.
  - Register.
  - Memory.
    - Direct.
    - Register indirect.
    - Register relative.
    - Based indexed.
    - Relative based indexed.

Immediate

- Directly accessible to the EU.
- The address is part of the instruction.
- Useful in initializations.
- MOV EAX,1111000B
- MOV CL, 0F1H

Register

- **Directly accessible to the EU.**
- **Most compact and fastest executing instructions.**
- **Operands are encoded in the instruction.**
- **MOV EBX,EDX**
- **MOV AL,CL**

Memory

- **When reading or writing to memory the execution unit passes an offset value, the effective address, to the bus interface unit which then computes the physical address.**

Direct

$$EA = \{[\text{operand}]\}$$
$$PA = \{DS\} \times 16 + \{[\text{operand}]\}$$

- **Simplest memory addressing mode.**
- **Access to simple variables.**
- **MOV EAX,DS:SUM**
- **MOV CL,DS:COUNT+5**
- **MOV DS:[500H],EDX**

Register Indirect

$$\mathrm{EA} = \begin{Bmatrix} (\mathrm{EBX}) \\ (\mathrm{EDI}) \\ (\mathrm{ESI}) \end{Bmatrix}$$

$$\mathrm{PA} = \{\mathrm{DS}\} \times 16 + \begin{Bmatrix} (\mathrm{EBX}) \\ (\mathrm{EDI}) \\ (\mathrm{ESI}) \end{Bmatrix}$$

- MOV EAX, DS:[EBX]
- MOV DS:[EDI],EDX

Register Relative

$$EA = \begin{Bmatrix} (EBX) \\ (EBP) \\ (EDI) \\ (ESI) \end{Bmatrix} + \begin{Bmatrix} \text{8 bit displacement} \\ \text{16 bit displacement} \end{Bmatrix}$$

$$PA = \begin{Bmatrix} DS \\ SS \\ DS \\ DS \end{Bmatrix} \times 16 + \begin{Bmatrix} (EBX) \\ (EBP) \\ (EDI) \\ (ESI) \end{Bmatrix} + \begin{Bmatrix} \text{8 bit displacement} \\ \text{16 bit displacement} \end{Bmatrix}$$

- Access to one dimensional arrays.
- MOV EAX,DS:ARRAY[EBX]
- MOV DS:MESSAGE[EDI], DL

Relative Based Indexed

$$\text{EA} = \begin{Bmatrix} (\text{EBX}) \\ (\text{EBP}) \end{Bmatrix} + \begin{Bmatrix} (\text{EDI}) \\ (\text{ESI}) \end{Bmatrix} + \begin{Bmatrix} \text{8 bit displacement} \\ \text{16 bit displacement} \end{Bmatrix}$$

$$\text{PA} = \begin{Bmatrix} \text{DS} \\ \text{SS} \end{Bmatrix} \times 16 + \begin{Bmatrix} (\text{EBX}) \\ (\text{EBP}) \end{Bmatrix} + \begin{Bmatrix} (\text{EDI}) \\ (\text{ESI}) \end{Bmatrix} + \begin{Bmatrix} \text{8 bit displacement} \\ \text{16 bit displacement} \end{Bmatrix}$$

- Used to access two dimensional arrays or arrays contained in structures.
- MOV DS:ARRAY[EBX][EDI],EAX

Accessing Arrays

- **One dimensional arrays.**
  - MOV DS:ARRAY[ESI*SF],EDX
  - SF = Scaling factor for data size.
- **Two dimensional arrays.**
  - MOV DS:ARRAY[EBX*SF*SR][ESI*SF],EDX
  - SF = Scaling factor for data size.
  - SR = Size of row.

Accessing Arrays

Assume the following array definition:

```
ARRAY       DD      00112233H, 44556677H, 88990011H
```

```
Begin:
        LEA EBX,DS:ARRAY
L1:
        MOV EAX,DS:[EBX]
        INC EBX
        JMP L1
```

```
Begin:
        MOV ESI,O
L1:
        MOV EAX,DS:ARRAY[ESI]
        INC ESI
        JMP L1
```

```
Begin:
        MOV ESI,O
L1:
        MOV EAX,DS:ARRAY[ESI*4]
        INC ESI
        JMP L1
```

Alignment

- It is best to align words with even numbered addresses, and double words to addresses divisible by four, but this is not necessary.
- The alignment allows for more efficient memory access, but it is less flexible.

Immediate - Memory

- When reading or writing to memory using immediate addressing mode, the programmer must specify the data size otherwise the assembler will default to the largest possible data size that processor handles.
- Use the following directives:
  – Byte ptr.
  – Word ptr.
  – Dword ptr.
- MOV DS:BYTE PTR VAR,2H

Unconditional Transfers

- **JMP**
- **CALL**
- **RET**
- **These instructions modify the EIP register to be:**
  - **Displacement following the instruction (label), in the case of JMP and CALL;**
  - **The address stored in the stack by the CALL instruction, in the case of RET.**
- **Ex:**
  - **JMP Again**
  - **CALL Delay**
  - **RET**

Conditional Transfers

- Used with unsigned integers
  - JA/JNBE – Jump if above
  - JAE/JNB – Jump if above or equal
  - JB/JNA – Jump if below
  - JBE/JNA – Jump if below or equal
- Used with signed integers
  - JG/JNLE – Jump if greater
  - JGE/JNL – Jump if greater or equal
  - JL/JNGE – Jump if less
  - JLE/JNG – Jump if less or equal
- Other conditions
  - JE/JZ – Jump if equal
  - JNE/JNZ – Jump if not equal
  - JC – Jump if carry
  - JNC – Jump if not carry
  - JS – Jump if sign
  - JNS – Jump if not sign

Conditional Transfers

  - JO – Jump if overflow
  - JNO – Jump if not overflow
  - JP/JPE – Jump if parity/parity even
  - JNP/JPO – Jump if not parity/parity odd
- These instructions conditionally modify the EIP register to be one of two addresses defined as follows:
  - An address or displacement following the instruction (label);
  - The address of the instruction following the conditional jump.
- Ex:
  -
```
      JE SUM
      SUB EAX,EBX
SUM:
```

Iteration Control

- **LOOP**
- **LOOPE/LOOPZ**
- **LOOPNE/LOOPNZ**
- **The instructions listed above are used to conditionally and unconditionally control the number of iterations a program go through a loop.**
- **Operation of LOOP:**
  - **ECX $\leftarrow$ ECX – 1**
  - **If ECX $\neq$ 0**
    then EIP $\leftarrow$ EIP + displacement
  - **Flags are not affected.**

Iteration Control

- **Ex:**
  - MOV ECX,2
  - Again: NOP
  - LOOP Again
- **What will happen if MOV ECX,2 is replaced by MOV ECX,0 in the code given above.**

Iteration Control

- **Operation of LOOPE/LOOPZ:**
  - **$ECX \leftarrow ECX - 1$**
  - **If $ZF = 1$ and $ECX \neq 0$**
    then $EIP \leftarrow EIP + displacement$
  - **Flags are not affected.**
- **Operation of LOOPNE/LOOPNZ:**
  - **$ECX \leftarrow ECX - 1$**
  - **If $ZF = 0$ and $ECX \neq 0$**
    then $EIP \leftarrow EIP + displacement$
  - **Flags are not affected.**
- **Note that other instructions within the loop have to change the condition of ZF.**

Iteration Control

- **Ex:**
  - MOV ECX,9
  - MOV ESI, -1
  - MOV AL, ‘D’
  - Again: INC ESI
  - CMP AL, LIST[EDI]
  - LOOP NE Again
  - JNZ NOT_FOUND

- **JECXZ/JCXZ – These instructions are conditional jumps if the ECX/CX register are equal to zero. They are used prior to a LOOP instruction to ensure that the iteration count, value in ECX/CX is never zero.**

Interrupts

- **INT**
- **INTO – Interrupt if overflow**
- **IRET**
- **These instructions modify the EIP register to be the address stored at:**
  - **The IDT. The interrupt type or number is used to identify which element of the IDT holds the addresses of the desired interrupt service subroutines;**
  - **The stack. The address stored in the stack by the INT or INTO instruction. This address identifies the return point after the interrupts execution.**

Passing Arguments To Subroutines or Modules

- Via Registers.
  - Number of registers is a major limitation associated with this method.
  - It is important to clearly document registers used.
- Via Memory.
  - Used by DOS and BIOS.
  - Difficult standardization.
  - Defined area of RAM is used to pass arguments.

Passing Arguments To Subroutines or Modules

- **Via Stack.**
  - **Most widely used method of passing parameters.**
  - **Register and memory independent.**
  - **Need to be thoroughly understood due to the fact that the stack is used by both the system and the user, so if the stack gets compromised the program can crash.**

String Instructions

- String instructions were designed to operate on large data structures.
- The SI and DI registers are used as pointers to the data structures being accessed or manipulated.
- The operation of the dedicated registers stated above are used to simplify code and minimize its size.

String Instructions

- The registers(DI,SI) are automatically incremented or decremented depending on the value of the direction flag:
  - DF=0, increment SI, DI.
  - DF=1, decrement SI, DI.

- To set or clear the direction flag one should use the following instructions:
  - CLD to clear the DF.
  - STD to set the DF.

String Instructions

- **The REP/REPZ/REPNZ prefixes are used to repeat the operation it precedes.**
- **String instructions we will discuss:**
  - **LODS**
  - **STOS**
  - **MOVS**
  - **CMPS**
  - **SCAS**

LODS/LODSB/ LODSW/LODSD

- **Loads the AL, AX or EAX registers with the content of the memory byte, word or double word pointed to by SI relative to DS. After the transfer is made, the SI register is automatically updated as follows:**
  - **SI is incremented if DF=0.**
  - **SI is decremented if DF=1.**

LODS/LODSB/ LODSW/LODSD

- **Examples:**
  - **LODSB**

    AL=DS:[SI]; SI=SI ± 1
  - **LODSW**

    AX=DS:[SI]; SI=SI ± 2
  - **LODSD**

    EAX=DS:[SI]; SI=SI ± 4
  - **LODS MEAN**

    AL=DS:[SI]; SI=SI ± 1 (if MEAN is a byte)
  - **LODS LIST**

    AX=DS:[SI]; SI=SI ± 2 (if LIST is a word)
  - **LODS MAX**

    EAX=DS:[SI]; SI=SI ± 4 (if MAX is a double word)

LODS/LODSB/ LODSW/LODSD

Example

Assume:

Location	Content
Register SI	500H
Memory location 500H	'A'
Register AL	'2'

After execution of LODSB

If DF=0 then:

Location	Content
Register SI	501H
Memory location 500H	'A'
Register AL	'A'

Else if DF=1 then:

Location	Content
Register SI	4FFH
Memory location 500H	'A'
Register AL	'A'

STOS/STOSB/ STOSW/STOSD

- **Transfers the contents of the AL, AX or EAX registers to the memory byte, word or double word pointed to by DI relative to ES. After the transfer is made, the DI register is automatically updated as follows:**
  - **DI is incremented if DF=0.**
  - **DI is decremented if DF=1.**

STOS/STOSB/ STOSW/STOSD

- **Examples:**
  - **STOSB**

    ES:[DI]=AL; DI=DI ± 1
  - **STOSW**

    ES:[DI]=AX; DI=DI ± 2
  - **STOSD**

    ES:[DI]=EAX; DI=DI ± 4
  - **STOS MEAN**

    ES:[DI]=AL; DI=DI ± 1 (if MEAN is a byte)
  - **STOS LIST**

    ES:[DI]=AX; DI=DI ± 2 (if LIST is a word)
  - **STOS MAX**

    ES:[DI]=EAX; DI=DI ± 4 (if MAX is a double word)

STOS/STOSB/ STOSW/STOSD

Example

Assume:

Location	Content
Register DI	500H
Memory location 500H	'A'
Register AL	'2'

After execution of STOSB

If DF=0 then:

Location	Content
Register DI	501H
Memory location 500H	'2'
Register AL	'2'

Else if DF=1 then:

Location	Content
Register DI	4FFH
Memory location 500H	'2'
Register AL	'2'

MOVS/MOVSB/ MOVSW/MOVSD

- **Transfers the contents of the the memory byte, word or double word pointed to by SI relative to DS to the memory byte, word or double word pointed to by DI relative to ES. After the transfer is made, the DI register is automatically updated as follows:**
  - **DI is incremented if DF=0.**
  - **DI is decremented if DF=1.**

MOVS/MOVSB/ MOVSW/MOVSD

- **Examples:**
  - **MOVSB**
    ES:[DI]=DS:[SI]; DI=DI ± 1;SI=SI ± 1
  - **MOVSW**
    ES:[DI]= DS:[SI]; DI=DI ± 2; SI=SI ± 2
  - **MOVSD**
    ES:[DI]=DS:[SI]; DI=DI ± 4; SI=SI ± 4
  - **MOVS MEAN**
    ES:[DI]=DS:[SI]; DI=DI ± 1; SI=SI ± 1 (if MEAN is a byte)
  - **MOVS LIST**
    ES:[DI]=DS:[SI]; DI=DI ± 2; SI=SI ± 2 (if LIST is a word)
  - **MOVS MAX**
    ES:[DI]=DS:[SI]; DI=DI ± 4; SI=SI ± 4 (if MAX is a double word)

MOVS/MOVSB/ MOVSW/MOVSD

Example

Assume:

Location	Content
Register SI	500H
Register DI	600H
Memory location 500H	'2'
Memory location 600H	'W'

After execution of MOVSB

If DF=0 then:

Location	Content
Register SI	501H
Register DI	601H
Memory location 500H	'2'
Memory location 600H	'2'

Else if DF=1 then:

Location	Content
Register SI	4FFH
Register DI	5FFH
Memory location 500H	'2'
Memory location 600H	'2'

CMPS/CMPSB/ CMPSW/CMPSD

- **Compares the contents of the the memory byte, word or double word pointed to by SI relative to DS to the memory byte, word or double word pointed to by DI relative to ES and changes the flags accordingly. After the comparison is made, the DI and SI registers are automatically updated as follows:**
  - **DI and SI are incremented if DF=0.**
  - **DI and SI are decremented if DF=1.**

SCAS/SCASB/ SCASW/SCASD

- Compares the contents of the AL, AX or EAX register with the memory byte, word or double word pointed to by DI relative to ES and changes the flags accordingly. After the comparison is made, the DI register is automatically updated as follows:
  - DI is incremented if DF=0.
  - DI is decremented if DF=1.

REP/REPZ/REPNZ

- **These prefixes cause the string instruction that follows them to be repeated the number of times in the count register ECX or until:**
  - **ZF=0 in the case of REPZ (repeat while equal).**
  - **ZF=1 in the case of REPNZ (repeat while not equal).**

REP/REPZ/REPNZ

- **Use REPNE and SCASB to search for the character ‘f’ in the buffer given below.**

- **BUFFER DB ‘EE3751’**

- **MOV AL,’f’**
- **LEA DI,BUFFER**
- **MOV ECX,6**
- **CLD**
- **REPNE SCASB**
- **JE FOUND**

REP/REPZ/REPNZ

- Use REPNE and SCASB to search for the character ‘3’ in the buffer given below.

- BUFFER DB ‘EE3751’

- MOV AL,’f’
- LEA DI,BUFFER
- MOV ECX,6
- CLD
- REPNE SCASB
- JE FOUND

PC Parallel Printer Port

- **Types:**
  - SPP – Standard Printer Port
  - PS/2 – Simple bidirectional
  - EPP – Enhanced Parallel Port
  - ECP – Extended Capabilities Port
- **Addressing:**
  - Base addresses:
    - 278H
    - 378H
    - 3BCH
- **Registers:**
  - Data, 8 bits, base address
  - Status, 5 bits, at base address + 1
  - Control, 6 bits, at base address + 2

PC Parallel Printer Port

Data Register (Base Address)				
Bit	**Pin: DB-25**	**Signal Name**	**Inverted at connector?**	**I/O**
0	2	Data bit 0	No	Output
1	3	Data bit 1	No	Output
2	4	Data bit 2	No	Output
3	5	Data bit 3	No	Output
4	6	Data bit 4	No	Output
5	7	Data bit 5	No	Output
6	8	Data bit 6	No	Output
7	9	Data bit 7	No	Output
Status Register (Base Address + 1)				
Bit	**Pin: DB-25**	**Signal Name**	**Inverted at connector?**	**I/O**
3	15	nError	No	Input
4	13	Select	No	Input
5	12	PaperEnd	No	Input
6	10	nAck	No	Input
7	11	Busy	Yes	Input
Control Register (Base Address + 2)				
Bit	**Pin: DB-25**	**Signal Name**	**Inverted at connector?**	**I/O**
0	1	NStrobe	Yes	Output
1	14	nAutoLF	Yes	Output
2	16	Ninit	No	Output
3	17	nSelectIn	Yes	Output
4		IRQ 1 = enabled		
5		Bidirectional 1 = input		

DB-25 and DB-9 Pin Diagram

The "o" represent holes, the "." represent pins.

DB-25 Connector

Connector 1 (Female)

```
  13 <-------------------- 1
 _____________________________
 \ o o o o o o o o o o o o o /
  \ o o o o o o o o o o o o /
   -------------------------
   25 <----------------- 14
```

Connector 2 (Male)

```
   1 --------------------> 13
 _____________________________
 \ . . . . . . . . . . . . . /
  \ . . . . . . . . . . . . /
   -------------------------
   14 ------------------> 25
```

DB-9 Connector

Connector 3 (Female)

```
   5 4 3 2 1
 _____________
 \ o o o o o /
  \ o o o o /
   ---------
    9 8 7 6
```

Connector 4 (Male)

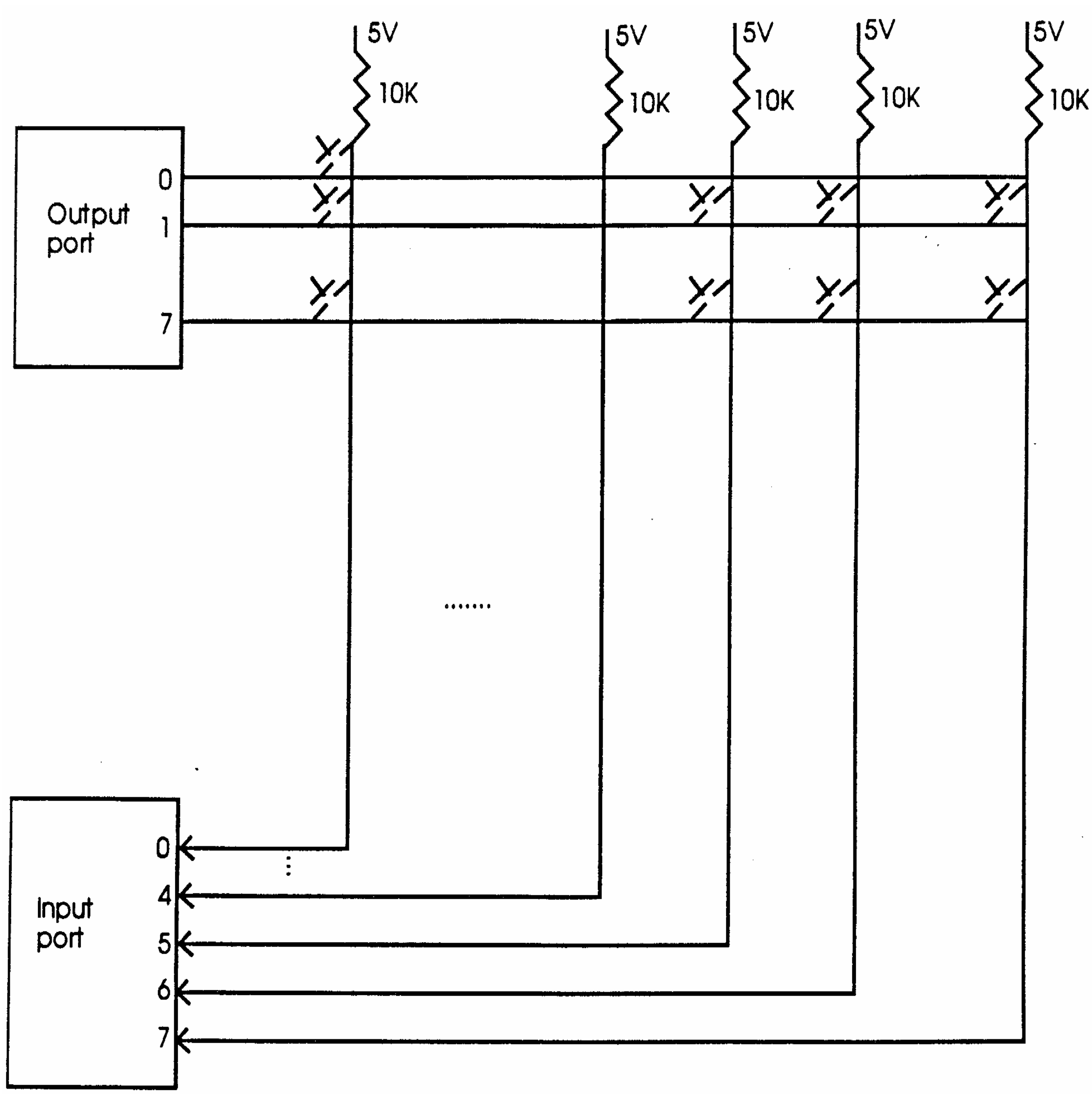
```
   1 2 3 4 5
 _____________
 \ . . . . . /
  \ . . . . /
   ---------
    6 7 8 9
```

Each diagram shown above is the view you see when you look into the end of the cable.

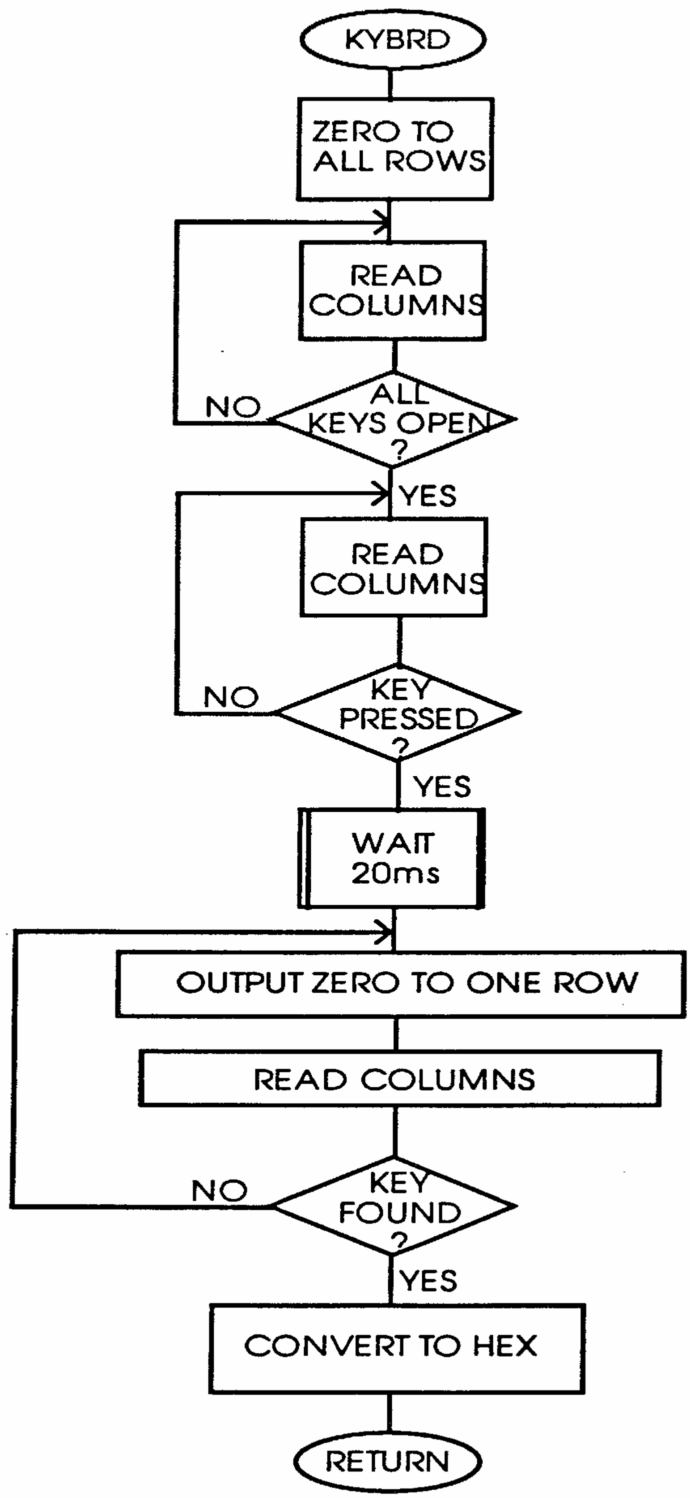
Keyboard Interfacing

- **There are several types of keyboards available for computer usage. Some of the most common types are:**
  - **Mechanical switches**
  - **Membrane switches**
  - **Capacitive switches**
  - **Hall effect key switches**
- **Most keyboards are organized as a matrix of rows and columns. Getting data from the keyboard requires the following steps:**
  - **Detect a key press.**
  - **Debounce the key press.**
  - **Encode the key.**

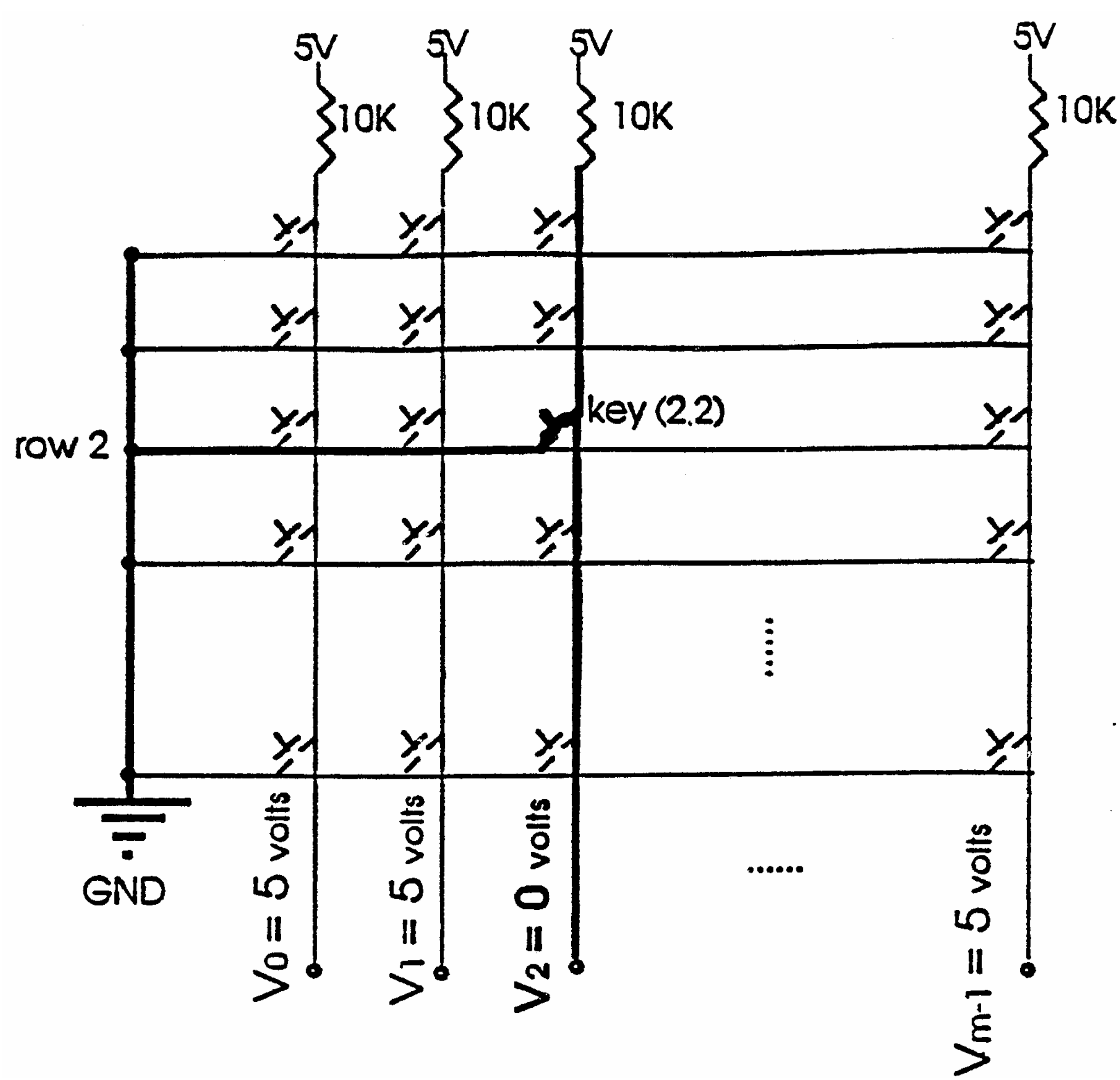
Keyboard Interfacing



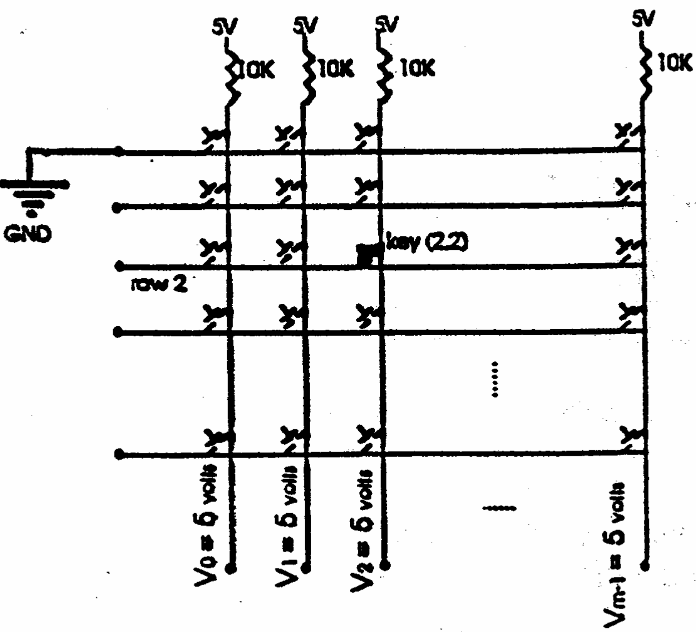
Keyboard Interfacing



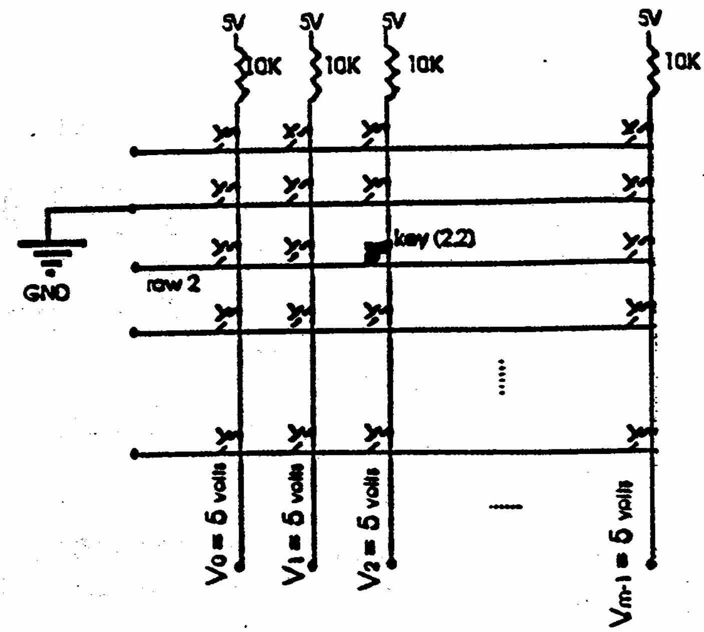
Keyboard Interfacing



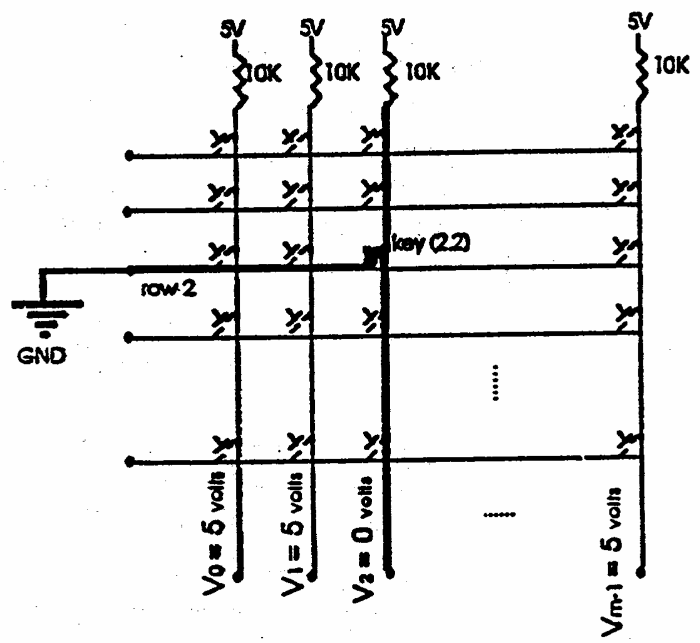
Keyboard Interfacing



a) row not found



b) row not found



Keyboard Interfacing

- **Encoding the key press:**
  - **Find the row and column positions (obtained from the key detection routine).**
  - **Calculate the offset using the following formula:**

    OFFSET = (row * 8) + column

    - **8 is the number of columns in the keyboard matrix.**
  - **Find the proper character using the offset, the base address of the conversion table and XLATB instruction.**

Interrupts

- Interrupts/exceptions are actions prompting the transfer of program execution to some special routine.
- Interrupt/exception Service Routine is the routine executed as a result of an interrupt/exception call.
- Interrupts:
  - Maskable Interrupts (MI):
    - Do not occur unless interrupt flag is set.
    - STI – sets interrupt flag.
    - CLI – clears interrupt flag.
  - Non-Maskable Interrupt (NMI):
    - No mechanism is provided to prevent NMI's.

Interrupts

- **Exceptions:**
  - Some instructions may generate exceptions. Example: DIV may generate the divide by zero exception.
- **Interrupt Descriptor Table (IDT), also known as Interrupt Vector Table, is a data structure used for the purpose of handling interrupts. They associate each interrupt/exception with an address indicating the location of the Interrupt Service Routine which will be used to service the calling interrupt.**