# MICROPROCESSORS AND MICROCONTROLLERS LAB

# DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

ACADEMIC YEAR 2012-2013
IV B.Tech EEE I-SEMESTER



PADMASRI DR B. V.RAJU INSTITUTE OF TECHNOLOGY VISHNUPUR, NARSAPUR, MEDAK (DIST.) – 502 313 Phone No: 08458 – 222031, www.bvrit.ac.in

### **PREFACE**

The significance of the Microprocessors and Microcontrollers Lab is renowned in the various fields of engineering applications. For an Electrical Engineer, it is obligatory to have the practical ideas about the applications of Microprocessors and Microcontrollers. By this perspective we have introduced a Laboratory manual cum Observation for Microprocessors and Microcontrollers Lab.

The manual uses the plan, cogent and simple language to explain the fundamental aspects of Microprocessors and Microcontrollers in practical. The manual prepared very carefully with our level best. It gives all the steps in executing an experiment.

**ACKNOWLEDGEMENT** 

It is one of life's simple pleasures to say thank you for all the help that one has extended

their support. I wish to acknowledge and appreciate Assoc Prof G Venu Madhav, Prof. N.

Bhoopal for their sincere efforts made towards developing the Microprocessors and

Microcontrollers Lab manual. I wish to thank students for their suggestions which are considered

while preparing the lab manual.

I am extremely indebted to Sri. Col Dr. T. S. Surendra, Principal and Professor,

Department of Electrical and Electronics Engineering, BVRIT for his valuable inputs and sincere

support to complete the work.

Specifically, I am grateful to the Management for their constant advocacy and incitement.

Finally, I would again like to thank the entire faculty in the Department and those people

who directly or indirectly helped in successful completion of this work.

(Prof. N. BHOOPAL) HOD - EEE

B.V.R.I.T. Lab Manual ------

3

### GUIDELINES TO WRITE YOUR OBSERVATION BOOK

- 1. Assembly Language Programs (ALP's), Algorithm, Theoretical Result and Practical Result should be on right side.
- 2. Flow Chart should be left side.
- 3. Result should always be in the ending.
- 4. You all are advised to leave sufficient no of pages between ALP's for theoretical or model calculations purpose.

#### **DO'S AND DON'TS IN THE LAB**

#### DO'S:-

- 1. Proper dress has to be maintained while entering in the Lab. (Boys Tuck in and shoes and girls should be neatly dressed)
- 2. Students should carry observation notes and record completed in all aspects.
- 3. ALP and its theoretical result should be there in the observation before coming to the next lab.
- 4. Student should be aware of next ALPs.
- 5. Students should be at their concerned desktop, unnecessary moment is restricted.
- 6. Student should follow the procedure to start executing the ALP they have to get signed by the Lab instructor for theoretical result then with the permission of Lab instructor they need to switch on the desktop and after completing the same they need to switch off and keep the chairs properly.
- 7. After completing the ALP Students should verify the ALP by the Lab Instructor.
- 8. The Practical Result should be noted down into their observations and result must be shown to the Lecturer In-Charge for verification.
- 9. Students must ensure that all switches are in the OFF position, desktop is shut down properly.

#### DON'Ts:-

- 1. Don't come late to the Lab.
- 2. Don't leave the Lab without making proper shut down of desktop and keeping the chairs properly.
- 3. Don't leave the Lab without verification by Lab instructor.
- 4. Don't leave the lab without the permission of the Lecturer In-Charge.

### MICROPROCESSORS AND MICROCONTROLLERS LAB

# **Syllabus**

#### I. Microprocessor 8086:

- 1. Introduction to TASM/MASM
- 2. Arithmetic operation Multi byte addition and subtraction, multiplication and division-signed and unsigned arithmetic operation, ASCII-arithmetic operation.
- 3. Logic operations- Shift and rotate- converting packed BCD to unpacked BCD, BCD to ASCII conversion.
- 4. By using string operation and instruction prefix: Move block, reverse string, sorting, inserting, deleting, length of the string, string comparison.
- 5. DOS/BIOS programming: Reading keyboard (Buffered with and without echo)-Display characters, strings.

# II. Interfacing:

1. 8259 – Interrupt Controller : Generate an Interrupt using 8259 timer.

2. 8279 – Keyboard display : Write a small program to display a string of

Characters.

3. 8255 – PPI : Write ALP to generate sinusoidal wave using

PPI.

4. 8251 – USART : Write a program to establish communication

between two processors.

#### III. Microcontroller 8051

- 1. Reading and writing on a parallel port.
- 2. Timer in different modes.
- 3. Serial communication implementation.

#### **Course Objectives**

- a. Familiarize the architecture of 8086 processor, assembling language programming and interfacing with various modules.
- b. The student can also understand of 8051 Microcontroller concepts, architecture, programming and application of Microcontrollers.
- c. Student able to do any type of VLSI, embedded systems, industrial and real time applications by knowing the concepts of Microprocessor and Microcontrollers.

#### **Course Outcomes**

- Analyze and apply working of 8086.
- Compare the various interface techniques. Analyze and apply the working of 8255, 8279, 8259, 8251, 8257 ICs and design and develop the programs.
- Learning the Communication Standards.

# **INDEX**

- 1. INTRODUCTION TO MASM/TASM
- 2. INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING
  - Levels of programming
  - Program Development Tools
  - Assembler Directives
- 3. MICROPROCESSOR TRAINER KIT
- 4. COMMUNICATION WITH HOST COMPUTER
- 5. PROGRAMMING MODEL OF 8086
- 6. PROGRAMMING IN 8086 MPU
  - Multi byte Addition, subtraction, multiplication, division
  - ASCII arithmetic addition, subtraction, multiplication, division
  - Logical operations-AND, OR, NOT, NAND, XOR
  - Packed BCD to unpacked BCD Conversion
  - BCD to ASCII conversion
  - Moving a block using string instructions
  - String Reversal
  - Comparison of TWO strings
  - DOS/BIOS Programming

# 7. Interfacing

- 8259 Interrupt Controller
- 8279 Keyboard display
- ALP to generate sinusoidal wave using 8255
- 8251 USART

#### 8. Microcontroller 8051

- Reading and writing on a parallel port.
- Timer in Different Modes
- Serial communication implementation

#### 1. Introduction to MASM /TASM

#### **MASM:** (Microsoft assembler)

**To Create Source File:** An editor is a program which allows you to create a file containing the assembly language statements for your program. This file is called a **source file.** 

Command to create a source file

# C:\MASM\BIN> Edit filename. asm

The next step is to process the source file with an assembler. When you run the assembler, it reads the source file of your program. On the first pass through the source program, the assembler determines the displacement of named data items, the offset labels, etc. and puts this information in a symbol table. On the second pass through the source program the assembler produces the binary code for each instruction and inserts the offsets, etc. that it calculated during first pass.

# C:\MASM\BIN > Masm filename. asm X, Y, Z ←

With this command assembler generates three files.

1. The first file (X) called the object file, is given the extension .OBJ

The object file contains the binary codes for the instructions and information about the addresses of the instructions.

- 2. The second file (Y) generated by the assembler is called the assembler list file and is given the extension .LST. The list file contains your assembly language statements, the binary codes for each instruction and the offset for each instruction.
- 3. The third file (Z) generated by this assembler is called the cross-reference file and is given the extension .CRF. The cross-reference file lists all labels and pertinent information required for cross referencing

**NOTE:** The Assembler only finds syntax errors: It will not tell you whether program does what it is supposed to do. To determine whether your program works, you have to run the program and test it.

Next step is to process the object file with linker.

# C:\MASM\BIN>LINK filename . obj

Run File [Filename1.exe] : "filename1.exe"

List file [nul.map] : NUL

Libraries [.lib] : library\_name

Definitions File [ nul.def]

#### Creation of Library: Refer Modular Programming Section

A Linker is a program used to join several object files into one layer object file

**NOTE**: On IBM PC – type Computers, You must run the LINK program on your .OBJ file even if it contains only one assembly module.

The linker produces a link file with the .EXE extension (an execution file)

Next Run C:\MASM\BIN> filename ←

**TASM:** (Turbo Assembler)

**To Create Source File:** An editor is a program which allows you to create a file containing the assembly language statements for your program. This file is called a **source file.** 

Command to create a source file

C:\TASM\BIN> Edit filename. Asm

The next step is to process the source file with an assembler. When you run the assembler, it reads the source file of your program. On the first pass through the source program, the assembler determines the displacement of named data items, the offset labels, etc. and puts this information in a symbol table. On the second pass through the source program the assembler produces the binary code for each instruction and inserts the offsets, etc. that it calculated during first pass.

# C:\TASM\BIN > TASM filename. asm X, Y, Z ←

With this command assembler generates three files.

- 4. The first file (X) called the object file, is given the extension .OBJ The object file contains the binary codes for the instructions and information about the addresses of the instructions.
  - 5. The second file (Y) generated by the assembler is called the assembler list file and is given the extension .LST. The list file contains your assembly language

- statements, the binary codes for each instruction and the offset for each instruction.
- 6. The third file (Z) generated by this assembler is called the cross-reference file and is given the extension .CRF. The cross-reference file lists all labels and pertinent information required for cross referencing

**NOTE:** The Assembler only finds syntax errors: It will not tell you whether program does what it is supposed to do. To determine whether your program works, you have to run the program and test it.

Next step is to process the object file with linker.

C:\TASM\BIN>TLINK filename . obj

A Linker is a program used to join several object files into one layer object file

**NOTE:** On IBM PC – type Computers, You must run the LINK program on your .OBJ file even if it contains only one assembly module.

The linker produces a link file with the .EXE extension (an execution file) Next Run

C:\TASM\BIN> TD filename.exe ←

# **Assembly Language Program Format:**

#### The assembler uses two basic formats for developing S/W

- a) One method uses MODELS and
- b) Other uses Full-Segment Definitions
- \* The models are easier to use for simple tasks.
  - \* The full segment definitions offer better control over the assembly language task and are recommended for complex programs.

#### a) Format using Models:

; ABSTRACT ; 8086 program ; Aim of Program

; REGISTERS ; Registers used in your program ; PORTS ; PORTS used in your program

. MODEL (type of model i.e. size of memory system)

FOR EXAMPLE

. MODEL SMALL

. STACK size of stack; define stack

. DATA ; define data segment

-----

-----Define variables

-----

. CODE ; define code segment

HERE: MOV AX, @DATA ; load ES,DS

MOV ES, AX MOV DS, AX

-----

EXIT 0 ; exit to DOS

**END HERE** 

(or)

We can write Code segment as follows.

. CODE ; Define Code Segment

. STARTUP

-----

. EXIT 0 END

# MEMORY MODELS FOR THE ASSEMBLER

Model Type	Description
TINY	All data and code must fit into one segment. Tiny
	programs are written in .COM format, which means
	that the program must be originated at location 100H
SMALL	This model contains two segments: one data segment
	of 64K bytes and one code segment of 64K bytes.
MEDIUM	This model contains one data segment of 64K bytes
	and any number of code segments for large
	programs.
COMPACT	One code segment contains the program, and any
	number of data segments contains the data.
LARGE	The large model allows any number of code and data
	segments.
HUGE	This model is the same as large, but the data
	segments may contain more than 64K bytes each.
FLAT	Only available to MASM 6.X. The flat model uses
	one segment of 512K bytes to tore all data and code.
	Note that this model is mainly used with Windows
	NT

#### INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING:

#### LEVELS OF PROGRAMMING:

There are three levels of programming

- 1. Machine language
- 2. Assembler language
- 3. High level language

Machine language programs are programs that the computer can understand and execute directly. Assembly language instructions match machine language instructions, but are written using character strings so that they are more easily understood, and High-level language instructions are much closer to the English language and are structured.

Ultimately, an assembly language or high level language program must be converted into machine language by programs called translators. If the program being translated is in assembly language, the translator is referred to as an assembler, and if it is in a high level language the translator is referred to as a compiler or interpreter.

#### **ASSEMBLY LANGUAGE PROGRAM DEVELOPMENT TOOLS:**

**EDITOR:** An editor is a program, which allows you to create a file containing the assembly language statements for your program.

**ASSEMBLER:** An assembler program is used to translate the assembly language Mnemonic instructions to the corresponding binary codes. The second file generated by assembler is called the assembler List file.

**LINKER:** A Linker is a program used to join several object files in to one large object file. The linkers produce link files with the .EXE extension.

**DEBUGGER:** If your program requires no external hardware, then you can use a debugger to run and debug your program. A debugger is a program, which allows you to load your object code program into system memory, execute the program, and troubleshoot or "debug" it.

#### **ASSEMBLER DIRECTIVES:**

An assembler is a program used to convert an assembly language program into the equivalent machine code modules. The assembler decides the address of each label and substitutes the

values for each of the constants and variables. It then forms the machine code for mnemonics and data in assembly language program.

Assembler directives help the assembler to correctly understand assembly language programs to prepare the codes. Commonly used assembler directives are DB, DD, DW, DUP, ASSUME, BYTE, SEGMENT, MACRO, PROC, OFFSET, NEAR, FAR, EQU, STRUC, PTR, END, ENDM, ENDP etc. Some directives generate and store information in the memory, while others do not.

**DB** :- Define byte directive stores bytes of data in memory.

**BYTE PTR** :- This directive indicates the size of data referenced by pointer.

**SEGMENT** :- This directive is to indicate the start of the segment.

**DUP (Duplicate)** :- The DUP directive reserves memory locations given by the

number preceding it, but stores no specific values in any of

these locations.

**ASSUME** :- The ASSUME statement is only used with full segment

definitions. This statement tells the assembler what names have

been chosen for the code, data, extra and stack segments.

**EQU** :- The equate directive equates a numeric ASCII or label to another

label.

**ORG** :- The ORG (origin) statement changes the starting offset address

in a segment.

**PROC and ENDP** :- The PROC and ENDP directives indicate start and end of a

procedure (Sub routine). Both the PROC and ENDP

directives require a label to indicate the name of the procedure.

The PROC directive, must also be followed with the NEAR or

FAR. A NEAR procedure is one that resides in the same code

segment as the program. A FAR procedure may reside at any

location in the memory system.

A macro is a group of instructions that performs one task, just as a procedure. The difference is that a procedure is accessed via a CALL instruction, while a macro is inserted in the program at the point of usage as a new sequence of instructions.

**MACRO** :- The first statement of a macro is the MACRO directive preceded with name of the macro.

**ENDM** :- The last statement of a macro is the ENDM instruction. Never place a label in front of the ENDM statement.

PUBLIC &EXTRN: - The public and extern directives are very important to modular programming. We use PUBLIC to declare that labels of code, data or entire segments are available to other program modules.

We use EXTRN to declare that labels are external to a module.

Without this statement, we could not link modules together to create a program using modular programming techniques.

OFFSET

:- Offset of a label. When the assembler comes across the OFFSET operator along with a label, it first computes the 16 – bit displacement of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement.

**LENGTH** :- Byte length of the label. This directive is used to refer to the length of data array or a string.

# 2. MICROPROCESSOR TRAINER KIT

The microprocessor trainer kit (microprocessor development kit) is an aid to understand the architecture, interfacing and programming of a microprocessor. Here we describe the ESA 86/88 – 2-trainer kit.

ESA 86/88-2 is a powerful, general-purpose microcomputer system, which can be operated either with 8086 CPU or with 8088 CPU. The basic system can be easily expanded through the system BUS connector. The built in Assembler/ Disassembler feature simplifies the programmers task of entering Assembly language programs .The on-board provision for 8087 numeric data processor makes it useful for number crunching applications. On board battery

back up provision is an added feature to take care of frequent power failures while conducting experiments of the trainer using manually assembled code.

It is also provided with peripherals and controllers such as

**8251A:** Programmable communication Interface for serial communication.

**8253-5**: Programmable Interval Timer

8255A: Two Programmable Peripheral Interface Devices provide 48 programmable I/O lines

**8259A**: Programmable Interrupt Controller provides interrupt vectors for 8 sources.

**8288**: Bus Controller for generating control signals

**ESA 86/88-2** is operated from the CRT terminals or a host computer system via the serial monitor and also can be operated from the on board key board.

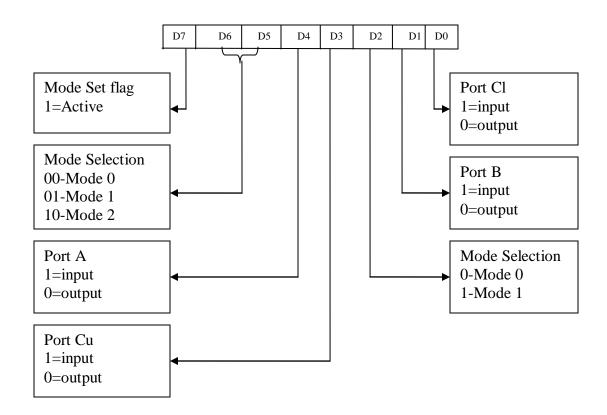
#### 8255 operational modes

8255 ports can be initialized in three different modes.

MODE 0: In this mode, all ports function as simple I/O ports without hand shaking.

MODE 1: This mode is handshake mode where by port A and port B use the bits Port C as handshake signals.

MODE 2: Only port A can be initialized in mode 2. In this mode port A can be used for Bidirectional handshake data transfer. Port B can be initialized in mode 0 or mode1.



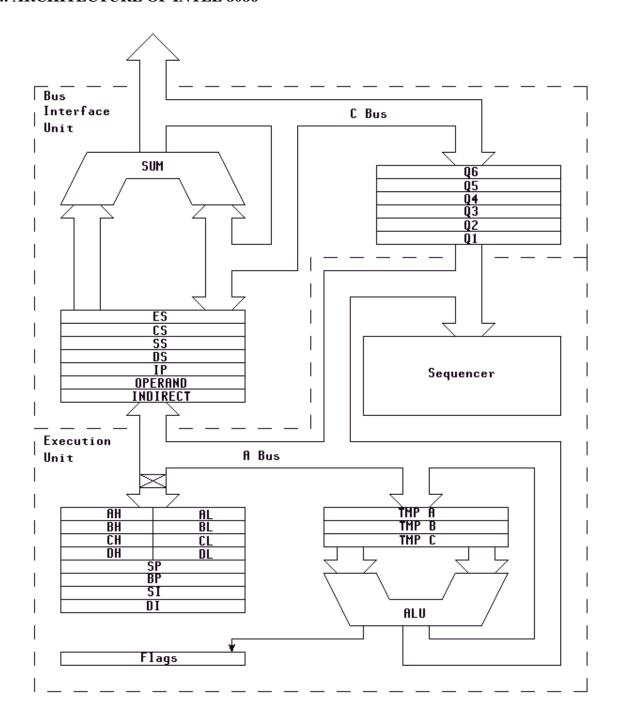
#### COMMUNICATION WITH A HOST COMPUTER SYSTEM

ESA 86/88-2 operating in the serial mode can be connected to either CRT terminal or host computer system. When computer system is the controlling element it must be executing the driver software to communicate with ESA 86/88-2.

XT86 is a package which allows the user to establish a communication link between ESA 86/88-2 system and a computer system. The link is established between asynchronous serial ports of the computer and ESA 86/88-2.A suitable RS232-C cables have to be used for connecting the kit to the computer system.

User can develop assembly language programs on the computer system, cross- assemble them using a suitable cross assembler to generate object code files and then use XT86 to download these object code files into the trainer kit for execution. User can terminate XT86 and return control to DOS by typing Alt + X. XT86 also allows uploading of data from the memory of the kit to the computer. The data so uploaded is same in a disk file.

#### 4. ARCHITECTURE OF INTEL 8086



#### Intel 8086 MPU PROGRAMMING

#### **USING DEBUG TO EXECUTE 80X86 PROGRAMS:**

DEBUG is a utility program that allows a user to load an 80x 86 programs in to memory and execute it step by step. DEBUG displays the contents of all processor registers after each instruction executes, allowing user to determine if the code is performing the desired task. DEBUG only displays the 16-bit portion of the general purpose registers. Code view is capable of displaying the entire 32 bits. DEBUG is a very useful debugging tool. We will use DEBUG to step through number of simple programs, gaining familiarity with DEBUG commands as we do. DEBUG contains commands that can display and modify memory, assemble instructions, disassemble code already placed into memory, trace through single or multiple instructions, load registers with data, and do much more.

DEBUG loads into memory like any other program, in the fist available slot. The memory space used by DEBUG for the user program begins after the end of DEBUG code. If an .EXE or. COM file were specified, DEBUG would load the program according to the accepted conventions.

To execute the program file PROG.EXE use this command:

#### **DEBUG PROG.EXE**

DEBUG uses a minus as its command prompt, so you should see a "-"appear on display.

To get a list of some commands available with DEBUG is:

- T trace (step by step execution)
- U un assemble
- D Dump
- G go (complete execution)
- H Hex

E.g.: to execute the program file PROG.ASM use the following procedure:

TASM PROG.ASM

TLINK PROG.OBJ

**DEBUG PROG.EXE** 

# Turbo Assembler Version 5.3 Copyright (c) 1988, 2000 Inprise Corporation

#### Syntax: TASM [options] source [,object] [,listing] [,xref]

/a, /s Alphabetic or Source-code segment ordering

/c Generate cross-reference in listing

/dSYM[=VAL] Define symbol SYM = 0, or = value VAL /e,/r Emulated or Real floating-point instructions

B.V.R.I.T. Lab Manual -----

/h,/? Display this help screen

/iPATH Search PATH for include files

/jCMD Jam in an assembler directive CMD (eg. /jIDEAL)

/kh# Hash table capacity # symbols

/l,/la Generate listing: l=normal listing, la=expanded listing

/ml, /mx,/mu Case sensitivity on symbols: ml=all, mx=globals, mu=none

/mv# Set maximum valid length for symbols

/m# Allow # multiple passes to resolve forward references

/n Suppress symbol tables in listing

/os,/o,/op,/oi Object code: standard, standard w/overlays, Phar Lap, IBM

/p Check for code segment overrides in protected mode

/q Suppress OBJ records not needed for linking /t Suppress messages if successful assembly

/uxxxx Set version emulation, version xxxx

/w0,/w1,/w2 Set warning level: w0=none, w1=w2=warnings on

/w-xxx,/w+xxx Disable (-) or enable (+) warning xxx /x Include false conditionals in listing /z Display source line with error message

/zi,/zd,/zn Debug info: zi=full, zd=line numbers only, zn=none

# <u>Turbo Link Version 4.01</u> Copyright (c) 1991 Borland International

# Syntax: TLINK objfiles, exefile, mapfile, libfiles, deffile

@xxxx indicates use response file xxxx

/m Map file with publics /x No map file at all

/i Initialize all segments /l Include source line numbers

/L Specify library search paths /s Detailed map of segments

/n No default libraries /d Warn if duplicate symbols in libraries

/c Case significant in symbols /3 Enable 32-bit processing

/o Overlay switch /v Full symbolic debug information

/P[=NNNNN] Pack code segments /A=NNNN Set NewExe segment alignment

/ye Expanded memory swapping /yx Extended memory swapping

/e Ignore Extended Dictionary

/t Create COM file (same as /Tdc)

/C Case sensitive exports and imports

/Txx Specify output file type

/Tdx DOS image (default)

/Twx Windows image

(third letter can be c=COM, e=EXE, d=DLL)

#### **<u>DEBUG- Testing and edition tool help</u>**; MS-DOS based program.

#### MS-DOS prompt/debug [filename .exe/.com/others]

assemble A [address]
compare C range address
dump D [range]

enter E address [list]

fill F range list

go G [=address] [addresses]

hex H value1 value2

input I port

load L [address] [drive] [firstsector] [number]

move M range address name N [pathname] [arglist]

output O port byte

proceed P [=address] [number]

quit Q

register R [register] search S range list

trace T [=address] [value]

unassemble U [range]

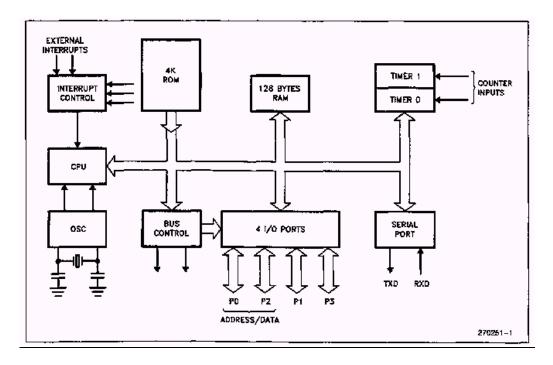
write W [address] [drive] [firstsector] [number]

allocate expanded memory XA [#pages] deallocate expanded memory XD [handle]

map expanded memory pages XM [Lpage] [Ppage] [handle]

display expanded memory status XS

### **ARCHITECTURE OF INTEL 8051**



# 8086 Assembly Language program1: adding two multi byte numbers and store the result as the third number:

**DATA SEGMENT** 

BYTES EQU 08H

NUM1 DB 05H, 5AH, 6CH, 55H, 66H, 77H, 34H, 12H

NUM2 DB 04H, 56H, 04H, 57H, 32H, 12H, 19H, 13H

NUM3 DB 0AH DUP (00)

**DATA ENDS** 

**CODE SEGMENT** 

ASSUME CS: CODE, DS: DATA

START: MOV AX, DATA

MOV DS, AX

MOV CX, BYTES

LEA SI, NUM1

LEA DI, NUM2

LEA BX, NUM3

MOV AX, 00

NEXT: MOV AL, [SI]

ADC AL, [DI]

MOV [BX], AL

**INC SI** 

INC DI

**INC BX** 

DEC CX

JNZ NEXT

INT 3H

**CODE ENDS** 

**END START** 

# 8086 Assembly Language program2: Subtracting two multi byte numbers and store the result as the third number:

**DATA SEGMENT** 

BYTES EQU 08H

NUM2 DB 05H, 5AH, 6CH, 55H, 66H, 77H, 34H, 12H

NUM1 DB 04H, 56H, 04H, 57H, 32H, 12H, 19H, 13H

NUM3 DB 0AH DUP (00)

**DATA ENDS** 

**CODE SEGMENT** 

ASSUME CS: CODE, DS: DATA

START: MOV AX, DATA

MOV DS, AX

MOV CX, BYTES

LEA SI, NUM1

LEA DI, NUM2

LEA BX, NUM3

MOV AX, 00

NEXT: MOV AL, [SI]

SBB AL, [DI]

MOV [BX], AL

**INC SI** 

INC DI

**INC BX** 

DEC CX

JNZ NEXT

INT 3H

**CODE ENDS** 

**END START** 

# 8086 Assembly Language program3: Multiplying two multi byte numbers and store the result as the third number:

**DATA SEGMENT** 

BYTES EQU 08H

NUM1 DB 05H, 5AH, 6CH, 55H, 66H, 77H, 34H, 12H

NUM2 DB 04H, 56H, 04H, 57H, 32H, 12H, 19H, 13H

NUM3 DB 0AH DUP (00)

**DATA ENDS** 

**CODE SEGMENT** 

ASSUME CS: CODE, DS: DATA

START: MOV AX, DATA

MOV DS, AX

MOV CX, BYTES

LEA SI, NUM1

LEA DI, NUM2

LEA BX, NUM3

MOV AX, 00

NEXT: MOV AL, [SI]

MOV DL,[DI]

MUL DL

MOV [BX], AL

MOV [BX+1],AH

**INC SI** 

INC DI

**INC BX** 

**INC BX** 

DEC CX

JNZ NEXT

INT 3H

**CODE ENDS** 

**END START** 

# 8086 Assembly Language program4: Dividing two multi byte numbers and store the result as the third number:

**DATA SEGMENT** 

BYTES EQU 08H

NUM2 DB 05H, 5AH, 6CH, 55H, 66H, 77H, 34H, 12H

NUM1 DB 04H, 56H, 04H, 57H, 32H, 12H, 19H, 13H

NUM3 DB 0AH DUP (00)

**DATA ENDS** 

**CODE SEGMENT** 

ASSUME CS: CODE, DS: DATA

START: MOV AX, DATA

MOV DS, AX

MOV CX, BYTES

LEA SI, NUM1

LEA DI, NUM2

LEA BX, NUM3

NEXT: MOV AX, 00

MOV AL, [SI]

MOV DL,[DI]

MUL DL

MOV [BX], AL

MOV [BX+1],AH

**INC SI** 

INC DI

**INC BX** 

**INC BX** 

DEC CX

JNZ NEXT

INT 3H

**CODE ENDS** 

**END START** 

#### 8086 Assembly Language program5: ASCII Addition:

**CODE SEGMENT** 

ASSUME CS: CODE

START: MOV AL,'5'

MOV BL,'9'

ADD AL, BL

AAA

OR AX, 3030H

INT 3H

**CODE ENDS** 

#### 8086 Assembly Language program6: ASCII Subtraction:

**CODE SEGMENT** 

**ASSUME CS: CODE** 

START: MOV AL,'9'

MOV BL,'5'

SUB AL, BL

**AAS** 

OR AX, 3030H

INT 3H

**CODE ENDS** 

**END START** 

#### 8086 Assembly Language program7: ASCII Multiplication:

**CODE SEGMENT** 

ASSUME CS: CODE

START: MOV AL, 5

MOV BL, 9

MUL BL

**AAM** 

OR AX, 3030H

INT 3H

**CODE ENDS** 

#### 8086 Assembly Language program8: ASCII Division:

**CODE SEGMENT** 

**ASSUME CS: CODE** 

START: MOV AX, 0607H

MOV CH, 09H

AAD

DIV CH

OR AX, 3030H

INT 3H

**CODE ENDS** 

**END START** 

#### <u>Logic Operations 8086 Assembly Language program9: LOGICAL AND:</u>

**CODE SEGMENT** 

ASSUME CS: CODE

START: MOV AL, 85H

MOV BL, 99H

AND AL, BL

INT 3H

CODE ENDS

**END START** 

#### <u>Logic Operations 8086 Assembly Language program10: LOGICAL OR:</u>

**CODE SEGMENT** 

**ASSUME CS: CODE** 

START: MOV AL, 85H

MOV BL, 99H

OR AL, BL

INT 3H

**CODE ENDS** 

#### Logic Operations 8086 Assembly Language program11: LOGICAL XOR:

**CODE SEGMENT** 

**ASSUME CS: CODE** 

START: MOV AL, 85H

MOV BL, 99H

XOR AL, BL

INT 3H

**CODE ENDS** 

**END START** 

#### <u>Logic Operations 8086 Assembly Language program12: NOT OPERATION:</u>

**CODE SEGMENT** 

ASSUME CS: CODE

START: MOV AL, 85H

NOT AL

INT 3H

**CODE ENDS** 

**END START** 

#### <u>Logic Operations 8086 Assembly Language program13: NAND OPERATION:</u>

CODE SEGMENT

**ASSUME CS: CODE** 

START: MOV AL, 85H

MOV BL, 99H

AND AL, BL

NOT AL

INT 3H

**CODE ENDS** 

#### 8086 Assembly Language program 14: Converting Packed BCD to unpacked BCD:

**DATA SEGMENT** 

NUM DB 45H

**DATA ENDS** 

**CODE SEGMENT** 

ASSUME CS: CODE, DS: DATA

START: MOV AX, DATA

MOV DS, AX

MOV AX, NUM

MOV AH, AL

MOV CL, 4

SHR AH, CL

AND AX, 0F0FH

INT 3H

**CODE ENDS** 

# 8086 Assembly Language program 15: Converting BCD to ASCII:

**DATA SEGMENT** 

NUM DB 45H

**DATA ENDS** 

**CODE SEGMENT** 

ASSUME CS: CODE, DS: DATA

START: MOV AX, DATA

MOV DS, AX

MOV AX, NUM

MOV AH, AL

MOV CL, 4

SHR AH, CL

AND AX, 0F0FH

OR AX, 3030H

INT 3H

**CODE ENDS** 

# 8086 Assembly Language program 16: Moving a Block using strings

**DATA SEGMENT** 

SRC DB 'MICROPROCESSOR'

DB 10 DUP (?)

DST DB 20 DUP (0)

**DATA ENDS** 

**CODE SEGMENT** 

ASSUME CS: CODE, DS: DATA, ES: DATA

START: MOV AX, DATA

MOV DS, AX

MOV ES, AX

LEA SI, SRC

LEA DI, DST

MOV CX, 20

CLD

REP MOVSB

INT 3H

**CODE ENDS** 

#### 8086 Assembly Language program 17:String reversal

```
DATA SEGMENT
     ORG 2000H
     SRC DB 'MICROPROCESSOR$'
     COUNT EQU ($-SRC)
     DEST DB?
DATA ENDS
CODE SEGMENT
     ASSUME CS:CODE,DS:DATA
     START: MOV AX, DATA
            MOV DS, AX
            MOV CX, COUNT
            LEA SI,SRC
            LEA DI, DEST
           ADD SI,CX
           DEC CX
     BACK: MOV AL,[SI]
           MOV [DI],AL
           DEC SI
           INC DI
           DEC CX
           JNZ BACK
           INT 3H
CODE ENDS
END START
```

#### 8086 Assembly Language program 18: Comparison of two Strings

```
PRINTSTRING MACRO MSG
     MOV AH, 09H
     LEA DX, MSG
     INT 21H
     ENDM
DATA SEGMENT
     ORG 2000H
     STR1 DB 'MICROPROCESSORS'
     LEN EQU ($-STR1)
     STR2 DB 'MICROPROCSSOR'
     M1 DB "STRINGS R EQUAL$"
     M2 DB "STRINGS R NOT EQUAL$"
DATA ENDS
CODE SEGMENT
     ASSUME CS: CODE, DS: DATA, ES: DATA
         START: MOV AX, DATA
```

MOV DS, AX
MOV ES,AX
LEA SI,STR1
LEA DI, STR2
MOV CX, LEN
CLD
REPE CMPSB
JNE FAIL
PRINTSTRING M1
INT 3H
FAIL: PRINTSTRING M2
INT 3H
CODE ENDS
END START

#### **DOS AND BIOS Programming:**

**DOS** (Disk operating System), **BIOS** (Basic I/O System) are used by assembly language to control the personal computer. The function calls control the personal computer .The function calls control everything from reading and writing disk data to managing the keyboard and displays.

**DOS Function Calls**: - In order to use DOS function calls, always place function number into register AH, and load other information into registers. Following is INT 21H, which is software interrupt to execute a DOS function.

All function calls use INT 21H, and AH contains function call number.

User can access the hardware of PC using DOS subroutine .DOS subroutines are invoked or called via software interrupt INT 21H.

**BIOS Function Calls:** - In addition to DOS Function call INT 21H, some other BIOS function calls are useful in controlling the I/O environment of a computer. BIOS function calls are found stored in the system and video BIOS ROMs. These BIOS ROM functions directly control the I/O devices with or without DOS loaded into a system.

**INT 10H:** - The INT 10H BIOS interrupt is often called the video services interrupt because it directly controls the video display in a system .The INT 10H instruction uses AH to select the video service provided by this interrupt.

**INT 11H:** - This function is used to determine the type of equipment installed in the system.

**INT 12H:** - The memory size is returned by the INT 12 H instruction.

INT 13H: - This call controls the diskettes, and also fixed or hard disk drives attached to the

system.

**DOS FUNCTIONS:** 

**Reading a key with an echo**: To read and echo a character, the AH is loaded with DOS function

number 01H. This is followed by the INT 21H instruction. Upon return from the INT 21H, the

AL register contains the ASCII character typed; the video display also shows the typed character.

Reading a key without an echo: To read a key without an Echo the AH register is loaded with

DOS function number 06H and DL=0FFH to indicate that the function call will read the key

board an echo.

**Read an entire line with an echo:** Sometimes it is advantageous to read an entire line of data

with one function call. Function call number 0AH reads entire line of information up to 255

characters from the keyboard. It continues to acquire keyboard data until it either enter key

(0DH) is typed or the character count expires.

Writing to video display: With almost any program, data must be displayed on the video

display. We use functions 02H or 06H for displaying one character at a time or function 09H for

displaying an entire string of characters. Because function 02H and 06H are identical, we intend

to use function 06H because it is also used to read a key. The character string can be of any

length and may contain control characters such as carriage return (0DH) and line feed (0AH).

**Terminate a process**: To terminate a process the AH register is loaded with function value 4CH.

This function returns control to DOS with the error code saved.

**DOS / BIOS Programming** 

FUNCTION CALLS SHOULD BE AVAILABLE IN AH REGISTER

1. Program to Read a character from keyboard (with Echo)

**Algorithm:** 

Step 1 Load DOS function call for reading a key

Step 2 Execute DOS function call

**Program:** 

.MODEL TINY

; Select tiny model

B.V.R.I.T. Lab Manual ------

36

.CODE ; Start CODE Segment

.STARTUP ; Start Program

MOV AH,01H ; Select function 01H INT 21H ; access DOS to read key

.EXIT ; exit to DOS END ; end of file

After execution ASCII equivalent of typed character will be in AL and typed character can be seen on the screen.

# 2. Reading a Key without Echo

#### **Algorithm:**

Step 1 Load DOS function call for reading a key

Step 2 Execute DOS function call

# **Program:**

.MODEL TINY ; Select tiny model .CODE ; Start CODE Segment

.STARTUP ; Start Program

MOV AH, 07H ; Read without an echo INT 21H ; access DOS to read key

.EXIT ; exit to DOS END ; end of file

After execution AL will have ASCII equivalent of the typed character

#### 3. Program to read a string of 10 characters into a buffer from keyboard

#### Algorithm:

- 1. Define Buffer in data segment
- 2. Initialize length of buffer
- 3. Load DOS function calls for reading into buffer
- 4. Execute the DOS function calls

# **Program:**

.MODEL SMALL ; Select SMALL Model .DATA ; Start DATA segment

BUF DB 257 DUP (?) ; define buf

.CODE ; start code segment .STARTUP ; start program

MOV BUF, 12 ; character count of 10

MOV DX, OFFSET BUF ; address buf MOV AH, 0AH ; read a line

INT 21H ; access DOS to read key

.EXIT ; exit to DOS END ; end of file

#### 4. a) Displaying a Character

; Function Calls for displaying one character at a time are 02H or 06H

; A program that displays a carriage return and a line feed using the DISP

; macro

.MODEL TINY ; Select TINY model
.CODE ; Start code segment
DISP MACRO A ; display A macro
MOV AH, 06H ; DOS function 06H
MOV DL, A ; Place parameter A in DL

INT 21H

**ENDM** 

.STARTUP ; Start Program

MOV AH, 06H MOV DL, 32H INT 21H

DISP 0DH ;Display carriage return DISP 0AH ; Display line feed

MOV AH, 06H MOV DL, 33H INT 21H END ; exit to DOS; end of file

#### 4. (b) Display a character string 5 times using DOS function call

#### **Algorithm:**

- 1. Define a character string in data segment
- 2. Load AH with DOS function call for displaying a string and DX with the offset of the string.
- 3. Execute the DOS function call

#### **Program:**

. MODEL SMALL. DATA; Select SMALL model; start Data Segment

ABC DB 'VNRVJIET\$'

. CODE ; start code segment . STARTUP ; start program

MOV CX, 05H ; load count

X: MOV AH, 09H ; select function 09H MOV DX, OFFSET ABC ; address character string

INT 21H ; access DOS

CALL DIL ; call procedure to display line feed

and carriage return

LOOP X

. EXIT ; exit to DOS

DIL PROC NEAR ; procedure to display line feed &

; carriage return

MOV AH,02H ; Select DOS function to display one

character

MOV DL,0AH ; place parameter for line feed in dl

INT 21H

MOV AH,02H MOV DL,0DH

INT 21H

**RET** 

DIL ENDP ; end of procedure

END ; end of file

# **Interfacing**

#### 1.8259

#### **DESCRIPTION OF THE MODULE:**

- □ The Study module card is connected to the 8086 kit through a 50 pin FRC cable. Before making connections check the polarity of the cable.
- □ It consists of 8259, switches, debounce switches for single stepping, Buffers and Tags for applying interrupts, V<sub>cc</sub> TAGS, LEDS To Display status.
- □ The toggle switch at top is for enabling single stepping. Push to ON for single stepping of every instruction. When single stepping enable will show each data transferred in an instruction on the data bus, including chip select, Read, Write, Interrupt and Interrupt Acknowledge Signals.
- □ 8259 is connected to IR4

# **DEMONSTRATION:**

This is a demonstration in which 8259 will be used in the stand alone mode. Program 1 will be used to illustrate this concept. In the main loop of program, it will be displaying (HELP), and when L to H pulse is applied to IRO line using patch cord in the ISR, it displays "IRO"

#### Step 1

Connect the 8259 PIC study module through 50 pin FRC cable.

#### Step 2

Enter the program as given

#### Step 3

Enable Single Stepping by Switch of module & kit.

#### Step 4

Execute the Program and observe the results on the LEDs

#### Step 5

'0' implies LED is OFF and '1' implies LED is ON

	DATA	$\overline{RD}$	$\overline{WR}$	$\overline{\mathit{INTA}}$	$\overline{A0}$	$\overline{CS}$	Comments
	BUS					M	
START	17	0	1		0	1	ICW1

STEP 1	00	0	1	 1	1	ICW2
STEP 2	01	0	1	 1	1	ICW4
STEP 3	FE	0	1	 1	1	Enable IR0
STEP 4	FF	1	0	 1	-	ICW1 for Slave

At this stage "WAITING FOR INT" will display on the LCD of VMC  $-\,8609$ . Give the interrupt to the study card by connecting IRO (Lower) to  $V_{CC}$ . Just after giving the interrupt "IRO" will display on LCD.

Addresses FOR 8086 CPU

IRR (Interrupt Request Register) - 60 H IMR (Interrupt Mask Register) - 66 H

Note: While using 8259 Study Module with 8086 kit switch of all position of SW1 Dip Switch Position.

#### PROGRAM:

This program is to demonstrate the use of 8259 PIC. Here only Master 8259 is used, during the main program, "HELP" is displayed while in the interrupt service loop, "IRO" is displayed.

0400	B8 00 00	MOV AX, 0000	; Data segment is initialize to zero
0403	8E D8	MOV DS, AX	
0405	B8 00 20	MOV AX, 2000	; interrupt location is defined
0408	89 06 00 00	MOV [0000], AX	_
040C	B8 00 00	MOV AX, 0000	
040F	89 06 02 00	MOV [0002], AX	
0413	B0 17	MOV AL, 17	; ICW1 Command
0415	E6 60	OUT 60, AL	
0417	B0 00	MOV AL, 00	;ICW2 Command
0419	E6 66	OUT 66, AL	
041B	B0 01	MOV AL, 01	; ICW4 Command
041D	E6 66	OUT 66, AL	
041F	BO FE	MOV AL, FE	; unmask IRQ0
0421	E6 66	OUT 66, AL	
0423	9A 7C F0 00 F0	CALL F0000 : F07C	; clear display
0428	B3 80	MOV BL, 80	; input parameter of subprogram is stored in BL, clear 1 <sup>st</sup> line.
042A	9A 78 F0 00 F0	CALL F000: F078	
042F	B0 80	MOV AL,80	; write all the commands in AL into
			LCD modulator
0431	9A 44 F0 00 F0	CALL F000: F044	
0436	0E	PUSH CS	
0437	1F	POP DS	

043F 0440 0445 0447	BE 00 06 B9 0F 00 FC L1: AC L0: 9A 48 F0 00 F0 E2 F7 FB E9 FD FF	MOV SI, 600 MOV CX,000F CLD DSB CALL F000: F048 LOOP 043E STI JMP 0448	; starting address of table is stored into SI; store table checking length in CX; clear direction flag; input AL data into LCD modulator; set interrupt flag
Interru	apt Sub – routine at 00	00: 2000	
2000 2005 2007	9A 7C F0 00 F0 B3 80 9A 78 F0 00 F0	CALL F000: F07C MOV BL, 80 CALL F000: F078	; delete the first line
	B0 86	MOV AL, 86	; write all the commands in Al into LCD modulator
2013	9A F0 00 F0 0E 1F	CALL F000: F044 PUSH CS POP DS	
2015 2018	BE 21 06 B9 0D 00	MOV SI, 621 MOV CX, 0D	; address of table is stored in Si ; table length stored in CX
201C	FC L3: AC L0: 9A 48 F0 00 F0	CLD DSB CALL F000 : F048	; input AL data into LCD modulator
2022 2024	E2 F7 CF	LOOP 201B IRET	; return to the execution program
0600	57 41 49 54 49 4E 47 46 4F 52 20 49 4E 54	-	WAITING FOR IRQ0 INTERRUPT
0621	49 52 30 20 49 4E 54 45 52 52 55 50 54	Į.	

------ Microprocessors Lab

#### 2. 8279 Keyboard Display

Interface keyboard and display controller 8279 with 8086 at addresses 0080H. Write an ALP to set up 8279 in scanned keyboard mode with encoded scan, N-key rollover mode. character display in right entry display format. Then clear the display RAM with zeroes. Read the FIFO for key closure. If any key is closed, store its code to register CL. Then write the byte 55 to all the displays and return to DOS. The lock input to 8279 is 2MHz, operate it at 100 kHz.

Tools Required: TASM, 8086 Kit, 8279 interfacing card.

# **Theory:**

The 8279 is interfaced with lower byte of the data bus, i.e. D0-D7. Hence the A0 input of 8279 is connected with address line A1. The data register of 8279 is to be addressed as 0080H, i.e. A0=0. For addressing the command or status word A0 input of 8279 should be 1 (the address line A1 of 8086 should be 1), i.e. the address of the command word should be 0082H.

# **Procedure:**

Step1: Set 8279 command words according to program i.e. Keyboard/Display Mode Set CW, Program clock selection, Clear Display RAM, Read FIFO, Write Display RAM commands.

Step2: Read FIFO command for checking display RAM.

Step3: Wait for clearing of Display RAM by reading FIFO Du bit of the status word i.e. if Du bit is not set wait, else proceed.

Step4: Read FIFO command for checking key closure, also read FIFO status.

Step5: Mask all bits except the number of characters bits. If any key is pressed, take required action; otherwise proceed to write display RAM by using write display command.

Step 6: Write the byte 55H to all display RAM locations.

Step 7: Call routine to read the key code of the pressed key is assumed available.

This Program displays the code of the key, which is pressed on the keyboard pad. The code is displayed in the data field and remains unchanged till the next key is pressed.

# **Description of the Program:**

The port of 8255 i.e. P1 is initialized to make port A as input port and port C as output port. The three Rows of the key are scanned one by one and process is repeated till the key is pressed, in the routine code and F code (final code). The information of code is then displayed and the monitor jumps back again to see if any other key is pressed.

Addresses	Opcodes	Label	Mnemonics	Comments
0400	BA FF FF	KBD	MOV DX,FFFF	Initialize the port – B and C as an output
0403	B0 90		MOV AL,90	1
0405	EE		OUT DX,AL	
0406	B7 00	INIT	MOV BH,00	Initialize the final key code in Reg. BH
0408	B3 01		MOB BL,01	Put the walking one pattern in register C with one LSB position
040A	88 D8	SCAN	MOV AL, BL	Move the pattern in AL on port C
040C	BA FD FF		MOV DX, FFFD	
040F	EE		OUT DX, AL	
0410	BA F9 FF		MOV DX, FFF9	
0413	EC		IN AL,DX	Input Port – A
0414	E8 27 00		CALL CODE	Classify the 8 word into 8 bits
0417	3C 08		CMP AL,08	Any Ke closure
0419	78 10		JS DISP	Yeas – go to display it
041B	80 C7 08		ADD BH,08	Increment the PC code in the partial result.
041E	80 FF 18		CMP BH,18	Has PC code become 18
0421	79 E3		JNS INIT	Yes – go start scanying from Row 0
0423	88 D8		MOV AL,BL	No
0425	D0 D0		RCL AL,01	Move the walking one to scan the next line
0427	88 C3		MOV BL,AL	
0429	EB DF		JMP SCAN	Continue scanning
042B	08 F8	DISP	OR AL,BH	OR the PA code with PC code
042D	B4 00		MOV AH, 00	Display the code in data field
042F	50		PUSH AX	-
0430	B0 00		MOV AL, 00	

0432	50		PUSH AX	
0433	B0 01		MOV AL, 01	
0435	50		PUSH AX	
0436	50		PUSH AX	
0437	9A E0 0B 00 FF		CALL DB	
			"OUTWARDS"	
043C	EB C8		JMP INIT	Go to scan the keyboard again
043E	08 C0	CODE	OR AL, AL	Checking for valid key press
0440	75 03		JNZ CODE2	If yes go to code2 else
0442	B0 08		MOV AL, 08	
0444			RET	
	<i>C3</i>			
0445	B5 00	CODE 2	MOV CH, 00	
0447	D0 C8	CODE 5	ROR Al,01	Let LSB in AL go to carry
0449	72 04		JC CODE 10	Go to return if this bit was one
044B	FE C5		INC CH	Increment counter
044D	EB F8		JMP CODE5	Check the next bit
044F	88 E8	CODE10	MOV AL, CH	
0451	C3		RET	

# 3. Write an ALP to generate Sinusoidal Wave Using 8255

ASSUME CS:CODE,DS:DATA

SINE DB 0,11,22,33,43,54,63,72,81,90,97,104,109,115,119,122

DB 125,,126,127,126,122,119,115,109,104,97,90,81,72,63,54,43,33,22,11

PA EQU 44A0H

CR EQU 44A3H

**DATA ENDS** 

**CODE SEGMENT** 

START: MOV AX, DATA

MOV DS, AX

MOV DX, CR

MOV AL, 80H

OUT DX, AL

REPEAT: MOV DX, PA

LEA SI, SINE

MOV CX, 36

**NEXT: MOV AL,[SI]** 

ADD AL, 128

OUT DX, AL

**INC SI** 

LOOP NEXT

MOV CX, 36

LEA SI, SINE

NEXT1: MOV AL, 128

MOV AH,[SI]

SUB AL, AH

OUT DX, AL

**INC SI** 

LOOP NEXT1

JMP REPEAT

MOV AH, 4CH

INT 21H

**CODE ENDS** 

**END START** 

# 4. <u>8251 (USART)</u>

# **DEMONSTRATION:**

In this experiment we will be using 8253 in Mode3, using counter 0 and load the count with 16 bit count. The 8251 is also initialized by specifying both command as well as the mode word. In the Experiment whatever data is transmitted from the CPU (with the help of RS -232) will be received by the 8251 and then will be transmitted back to the CPU and displayed on the screen. The program can be run either in free run mode or single stepping mode.

# Step 1:

Connect the 8253 / 8251 study module card to the 8086 kit via a 50 pin FRC. Check the polarity of the cable for proper data transmission.

#### **Step 2:**

Connect the 8253 kit to the computer input / output port with a RS232 cable.

#### **Step 3:**

Connect the CLK 0 tag to the 8086 CLK

#### **Step 4:**

Connect Gate 0 tag to  $+5V\ V_{CC}$ 

#### **Step 5:**

Connect out O tag to R x C & T x C tags

#### **Step 6:**

Enter the Program given below

# **Step 7:**

Enter the Program by pressing Reset, Exmem, Next Keys.

#### **Step 8:**

Execute the Program using <Reset>, <Go>, <.> Keys

# **PRORGAM:**

	MOV DX, FFD6	; Load Mode Control Word and Send it
	MOV AL, CEH	
	OUT DX, AL	
	MOV CX, 2	
X:	LOOP X	; Delay
	MOV AL, 36	; Load Command Word and Send it
	OUT DX, AL	
TEST1:	IN AL, DX	; Read Status
	AND AL, 81H	; and Check Status of Data Set Ready and
		; Transmit Ready
	CMP AL, 81H	; Is it Ready?
	JNE TEST 1	; Continue to poll if not Ready
Y:	MOV DX, FFD0	; Otherwise Point it Data Address
	MOV AL, Data to send	; Load data to send
	OUT DX, AL	; send it
	MOV AL, 00	
	OUT DX, AL	
	JMP Y	

# **8051**

- 1. Reading and writing on a parallel port.
  - 1. Writing to a port pin

SETB P3.5; set pin 5 of port 3

MOV P1, #4AH; sending data 4AH to port 1 - the binary pattern on the port will be 0100 1010

MOV P2, A; send whatever data is in the accumulator to port 2

# 2. Reading a port pin

SETB P1.0; initialize pin 0 of port 1 as an input pin

MOV P2, #FFH; set all pins of port 2 as inputs

MOV C, P1.0; move value on pin 0 of port 1 to the carry

MOV R3, P2; move data on port 2 into R3

# 2. Timer in Different Modes

The basic 8051 has two on-chip timers that can be used for timing durations or for counting external events Interval timing allows the programmer to perform operations at specific instants in time. For example, in our LED flashing program the LED was turned on for a specific length of time and then turned off for a specific length of time. We achieved this through the use of time delays. Since the microcontroller operates at a specific frequency, we could work out exactly how many iterations of the time delay was needed to give us the desired delay. However, this is cumbersome and prone to error. And there is another disadvantage; the CPU is occupied, stepping through the loops. If we use the on-chip timers, the CPU could be off doing something more useful while the timers take on the menial task of keeping track of time

#### The Timers' SFRs

The 8051 has two 16-bit timers. The high byte for timer 1 (TH1) is at address 8DH while the low byte (TL1) is at 8BH The high byte for timer 0 (TH0) is at 8CH while the low byte (TL0) is at 8AH.Both timers can be used in a number of different modes. The programmer sets the timers to a specific mode by loading the appropriate 8-bit number into the Timer Mode Register (TMOD) which is at address 89H.

# **Timer Mode Register**

TM	TMOD				
Bit	Name	Timer	Description		
7	Gate	1	Gate bit; when set, timer only runs while INT-bar is high. This bit is used in conjunction with interrupts and will be dealt with later.		
6	C/T- bar	1	Counter/timer select bit; when set timer is an event <b>c</b> ounter, when cleared timer is an interval <b>t</b> imer.		
5	M1	1	Mode bit 1		
4	<b>M</b> 0	1	Mode bit 0		
3	Gate	0	Gate bit; when set, timer only runs while INT-bar is high.		
2	C/T- bar	0	Counter/timer select bit; when set timer is an event <b>c</b> ounter, when cleared timer is an interval <b>t</b> imer.		
1	M1	0	Mode bit 1		
0	<b>M</b> 0	0	Mode bit 0		

The functions of the 8-bits of TMOD are described in the above table. The top four bits are for timer 1 and the bottom four bits have the exact same function but for timer 0. The *Gate* bits are used in conjunction with interrupts and will be dealt with at a later stage. For the moment we can take it that bits 7 and 3 are always cleared. As mentioned above, the timers can be used for counting external events or for timing intervals. If you wish the timer to be an event counter you set the corresponding C/T-bar bit. Similarly, if you wish it to be an interval timer you reset the corresponding C/T-bar bit. There are two mode bits (M1 and M0) for each timer. The table below describes their function

M1	<b>M</b> 0	Mode	Description
0	0	0	13-bit timer mode (this mode exists simply to keep the 8051 backwards compatible with its predecessor, the 8048, which had a 13-bit timer) - we will not be using mode 0.
0	1	1	16-bit timer mode
1	0	2	8-bit auto-reload mode
1	1	3	Split timer mode - this mode will be dealt with at a later stage

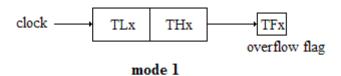
There are four timer modes, set by the bits M1 and M0. Mode 0 is not commonly used.

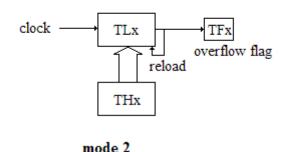
#### Mode 1 - 16-bit mode

The high byte (THx) is cascaded with the low byte (TLx) to produce a 16-bit timer. This timer counts from 0000H to FFFFH - it has  $2^{16}$  (65,536) states. An overflow occurs during the FFFFH to 0000H transition, setting the overflow flag (to be dealt with shortly).

#### Mode 2- 8-bit auto-reload mode

The timer low byte (TLx) operates as an 8-bit timer (counting to FFH) while the high-byte holds a reload value. When the timer overflows from FFH, rather than starting again from 00H, the value in THx is loaded into TLx and the count continues from there.





A diagrammatic representation of mode 1 and mode 2.

# 3. Serial communication implementation

All communication we are dealing with can be serial or parallel.

In Parallel communication, data being transferred between one location and another (R0 to the accumulator, for example) travel along the 8-bit data bus. Because of this data bus, data bytes can be moved about the microcontroller at high speed.

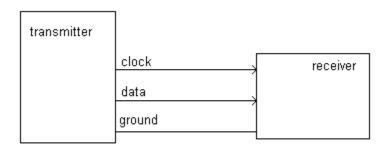
However, parallel communication has the disadvantage of requiring at least eight separate lines (in an 8-bit system) and in most cases extra lines to synchronize the data transfer (in the case of the microcontroller, the control bus).

Serial communication has the advantage of requiring only one line for the data, a second line for ground and possibly a third line for the clock. Therefore, because serial communication requires less physical wires, it is more suitable for transmitting data over longer distances.

The obvious disadvantage of serial communication, compared with parallel, is the reduction in the data transfer rate. If we imagine a system where it takes 1 us for data to settle on the data bus, we could say it takes 1 us to transfer a data byte using parallel communication. If we imagine the same timeframe for data bits settling on the serial line, it would take 8us to transfer a data byte using serial communication (1us for each bit).

# **Synchronous Serial Communication**

Synchronous serial communication requires an extra line for the clock signal. For serial communication, the 8-bit parallel data byte must be shifted down the serial line (in transmission). Therefore, one bit is followed by another. Some kind of system must be used to determine how long each bit is on the line. For example, the serial system designer may decide each bit will be on the line for 1us and, as explained above, transmission of the full eight bits would take 8us. With synchronous communication, the clock signal is transmitted on a separate line, as shown in the diagram below



In this way, the receiver is synchronized with the transmitter. As we shall see, the 8051 serial port in mode 0 is an example of synchronous serial communication.

#### **Asynchronous Serial Communication**

A good example of asynchronous serial communication is the interface between a keyboard and a computer. In this case, the keyboard is the transmitter and the computer is the receiver. With asynchronous communication, a clock signal is not sent with the data. There are a number of reasons why this form of communication might be desirable over synchronous communication. One advantage is the fact that the physical line for the clock is not needed. Also, asynchronous communication is better over long distances. If we try to synchronize a remote receiver by sending the clock signal, due to propagation delays and interference, the validity of the clock is lost.

Another reason for not transmitting the clock arises when the data rate is erratic. For example, data rate from a keyboard to a computer is dependent upon the typist. The user may type at a rate of sixty words per minute, but at other times he/she may type a lot less. And for long periods there may be no data sent at all. Because of this erratic data rate an asynchronous communication system is suitable.

#### **Serial Communication Protocol**

In any communication system, the receiver must know what kind of data to expect and at what rate the data will arrive. In both synchronous and asynchronous serial communication, the receiver needs to know with which bit the transmitter begins. In most systems the LSB is the first bit transmitted. For an asynchronous system, the number of bits transmitted per second must be known by the receiver. Since the clock signal is not transmitted, the receiver needs to know what clock frequency the transmitter is using so that it can use the same. The receiver also needs to know how many bits per word the transmitter is using (in most cases we deal with 8-bit words, but we will see cases where nine bits are transmitted per word). And the receiver needs to know where the data begins and where the data stops. All these parameters make up the protocol. If the receiver uses the same protocol as the transmitter is should receive the data correctly (although errors can occur and we will look at how we catch these errors at a later date). If the receiver uses a protocol other than the one used by the transmitter, then the two devices are effectively speaking two different languages and the data received will be garbage.

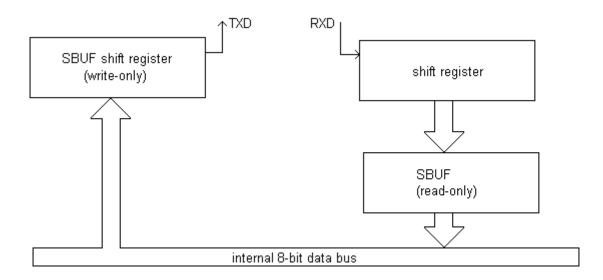
# **Start Bits and Stop Bits**

In asynchronous communication, at least two extra bits are transmitted with the data word; a start bit and a stop bit. Therefore, if the transmitter is using an 8-bit system, the actual number of bits transmitted per word is ten. In most protocols the start bit is a logic 0 while the stop bit is logic 1. Therefore, when no data is being sent the data line is continuously HIGH. The receiver waits for a 1 to 0 transition. In other words, it awaits a transition from the stop bit (no data) to the start bit (logic 0). Once this transition occurs the receiver knows a data byte will follow. Since it knows the data rate (because it is defined in the protocol) it uses the same clock as frequency as that used by the transmitter and reads the correct number of bits and stores them in a register. For example, if the protocol determines the word size as eight bits, once the receiver sees a start bit it

reads the next eight bits and places them in a buffer. Once the data word has been read the receiver checks to see if the next bit is a stop bit, signifying the end of the data. If the next bit is not logic 1 then something went wrong with the transmission and the receiver dumps the data. If the stop bit was received the receiver waits for the next data word, ie; it waits for a 1 to 0 transition

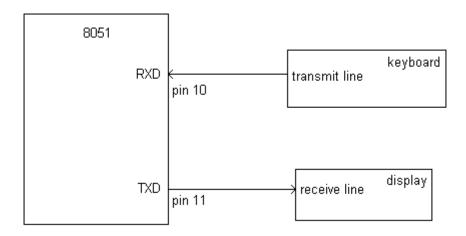
#### The 8051 Serial Port

The 8051 includes an on-chip serial port that can be programmed to operate in one of four different modes and at a range of frequencies. In serial communication the data is rate is known as the baud rate, which simply means the number of bits transmitted per second. In the serial port modes that allow variable baud rates, this baud rate is set by timer 1.



The 8051 serial port is full duplex. In other words, it can transmit and receive data at the same time. The block diagram above shows how this is achieved. If you look at the memory map you will notice at location 99H the serial buffer special function register (SBUF). Unlike any other register in the 8051, SBUF is in fact two distinct registers – the write-only register and the read-only register. Transmitted data is sent out from the write-only register while received data is stored in the read-only register. There are two separate data lines, one for transmission (TXD) and one for reception (RXD). Therefore, the serial port can be transmitting data down the TXD line while it is at the same time receiving data on the RXD line.

The TXD line is pin 11 of the microcontroller (P3.1) while the RXD line is on pin 10 (P3.0). Therefore, external access to the serial port is achieved by connecting to these pins. For example, if you wanted to connect a keyboard to the serial port you would connect the transmit line of the keyboard to pin 10 of the 8051. If you wanted to connect a display to the serial port you would connect the receive line of the display to pin 11 of the 8051. This is detailed in the diagram below.



# **Transmitting and Receiving Data**

Essentially, the job of the serial port is to change parallel data into serial data for transmission and to change received serial data into parallel data for use within the microcontroller.

- Serial transmission is changing parallel data to serial data.
- Serial reception is changing serial data into parallel data.
- Both are achieved through the use of shift registers.

As discussed earlier, synchronous communication requires the clock signal to be sent along with the data while asynchronous communication requires the use of stop bits and start bits. However, the programmer wishing to use the 8051 need not worry about such things. To transmit data along the serial line you simply write to the serial buffer and to access data received on the serial port you simply read data from the serial buffer.

#### For example:

- MOV SBUF, #45H this sends the byte 45H down the serial line
- MOV A, SBUF this takes whatever data was received by the serial port and puts it in the accumulator.

# How do we know when the complete data byte has been sent?

As mentioned earlier, it takes a certain length of time for a data byte to be transmitted down the serial line (determined by the baud rate). If we send data to SBUF and then immediately send more data to SBUF, as shown below, the initial character will be overwritten before it was completely shifted down the line.

- MOV SBUF, #23H
- MOV SBUF, #56H

Therefore, we must wait for the entire byte to be sent before we send another. The serial port control register (SCON) contains a bit which alerts us to the fact that a byte has been transmitted; ie; the transmit interrupt flag (TI) is set by hardware once an entire byte has been transmitted down the line. Since SCON is bit-addressable we can test this bit and wait until it is set, as shown below:

MOV SBUF, #23H; send the first byte down the serial line JNB TI, \$; wait for the entire byte to be sent CLR TI; the transmit interrupt flag is set by hardware but must be cleared by software MOV SBUF, #56H; send the second byte down the serial line

#### How do we know when data has been received?

Similarly, we need to know when an entire byte has been received by the serial port. Another bit in SCON, the receive interrupt flag (RI) is set by hardware when an entire byte is received by the serial port. The code below shows how you would program the controller to wait for data to be received and to then move that data into the accumulator.

JNB RI, \$; wait for an entire byte to be received CLR RI; the receive interrupt flag is set by hardware but must be cleared by software MOV A, SBUF; move the data stored in the read-only buffer to the accumulator