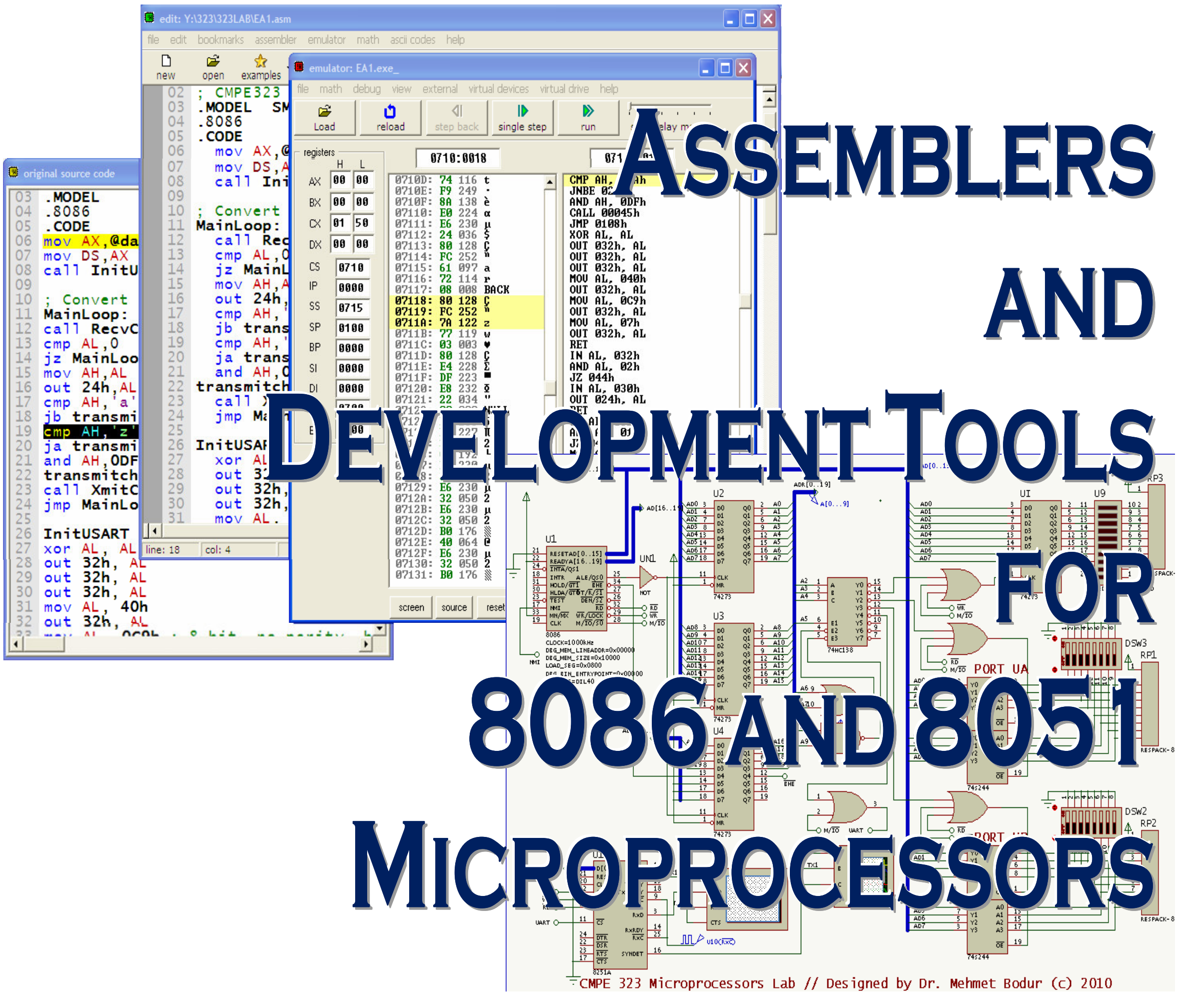
Eastern Mediterranean University
Computer Engineering Department

Assemblers and Development Tools for 8086 and 8051 Microprocessors



CMPE323 Microprocessors Lab Manual

Dr. Mehmet Bodur

Foreword

The objective of this book is to supply sufficient guidance to exploit the tools for developing microprocessor based design and application projects up to physical level of the implementation. The contents of is book is a collection of the hands-on experiments to practice several hardware/interfacing/software issues for an introductory level microprocessor course in a Computer Engineering program.

You may find considerable amount of practical information to guide a student in using the modern microprocessor development tools along with the classical assembly programming environments. The material is displayed in ten experimental chapters, where the first five experiments are mainly on the development and demonstration of software in 8086 assembly language, next three are on the 8051 hardware for microprocessor interface units including ports, memory, analog to digital converters and serial communication ports. Furthermore it contains two 8051 system examples with development details in higher level languages Keil-C51 C compiler. These two design examples are expected to serve for term assignments to an introductory level microprocessor course such as CMPE 323 in Computer Engineering Program of the Computer Engineering Department at Eastern Mediterranean University, where the experiments are currently carried as lab activities of CMPE 323 course.

The author of this book is aware of lots of books concentrating on both application design and practical issues on using microprocessors. In the perspective of the author, the shift of the microprocessor based applications from the assembly to the higher level languages is inevitable while the interfacing units, memory size, and processing power of the processors are developed in Moore’s law, almost doubling at every two or three years.

Finally it is the authors pleasure to acknowledge his colleagues Dr. Mohammed Salamah and Prof. Dr. Hasan Komurcugil who contributed to the previously given microprocessor courses, CMPE222, CMPE 326 and CMPE328. The finalized experiments are a product of an evolution starting from the mentioned courses.

This kind of books to guide the practical applications on diverged microprocessor development tools are not expected to be error-free, although the author spent considerable effort for the correction of the errors during the practical laboratory exercise of the students who followed the included experimental procedures. The author welcomes your comments, suggestions, and corrections for the corrected editions of these laboratory notes.

Welcome to work with the microprocessors, their languages, and their development tools.

Dr. Mehmet Bodur

Contents

FOREWORD .... **III**

CONTENTS .... **V**

1. TASM, EDIT, DEBUG AND EMU8086 ASSEMBLER TOOLS **1**

1.1 OBJECTIVE 1
1.2 INTRODUCTION 1
1.2.1. Editing the source file *1*
1.2.2. Assembling to an object file *1*
1.2.3. Linking to an Executable or Command File *2*
1.2.4. Tracing and Debugging of an EXE file *3*
1.2.5. Emu86 IDE *4*
1.2.6. EMU8086 Source Editor *4*
1.2.7. EMU8086 / MASM / TASM compatibility *5*
1.3 EXPERIMENTAL PART 7
1.3.1. Writing a Source File *7*
1.3.2. Assembling with TASM *8*
1.3.3. Assembling with Emu8086 .... *9*

2. DATA TYPES, AND EFFECT OF ALU INSTRUCTIONS ON FLAGS **11**

2.1 OBJECTIVE 11
2.2 PRELIMINARY STUDY 11
2.3 EXPERIMENTAL PART 11
2.3.1. Data types and Data directives *11*
2.3.2. ALU Operations and Flags *13*

3. SIMPLE VIRTUAL 8086 DEVELOPMENT BOARD **15**

3.1 OBJECTIVE 15
3.2 INTRODUCTION 15
3.2.1. 8086 and main memory *15*
3.2.2. 8086 Processor Bus *15*
3.2.3. Address Latching *16*
3.2.4. System Configuration *16*
3.2.5. IO Address decoding *16*
3.2.6. Simple Output Port UL *18*
3.2.7. Simple Input Ports UA and UB *18*
3.2.8. Serial Communication Device *19*
3.3 EXPERIMENTAL PART 21
3.3.1. Execution of a code on a virtual 8086 system *21*
3.3.2. Adding Port UA and Port UB *22*
3.3.3. USART and Capitalization *23*

4. BIOS AND DOS SERVICES **29**

4.1 OBJECTIVE 29
4.2 PRELIMINARY STUDY 29
4.3 EXPERIMENTAL PART 29
4.3.1. DOS services for String Display and Input *29*
4.3.2. Subroutines and Include files *31*

5. USING SIGNED NUMBERS AND LOOK-UP TABLES 35

5.1 OBJECTIVE 35
5.2 PRELIMINARY STUDY 35
5.3 EXPERIMENTAL PART 35
5.3.1. Macro Library for BIOS and DOS Services 35
5.3.2. Average by Signed Arithmetic Operations 38
5.3.3. Look-Up Table for the Square Root of an Integer. 39
5.3.4. Simple Look-Up Table for Fibonacci Numbers. 40

6. I/O AND EXTERNAL MEMORY INTERFACE FOR 8051 45

6.1 OBJECTIVE 45
6.2 INTRODUCTION 45
6.2.1. Typical features 45
6.2.2. Registers 45
6.2.3. Instruction Set 47
6.2.4. The 8051 Ports 48
6.2.5. Command line Assembler for 8051 48
6.2.6. IDE Tool for Coding of 8051 49
6.2.7. Simulation in ISIS 50
6.3 EXPERIMENTAL PART 50
6.3.1. Installation of A51 to your work folder 50
6.3.2. Simulation of a Microcontroller Circuit. 52

7. 8051 MEMORY DECODERS AND MEMORY INTERFACE 55

7.1 OBJECTIVE 55
7.2 8051 MEMORY INTERFACING 55
7.3 EXPERIMENTAL PART 55
7.3.1. Installation of KC51 and preparation of -.HEX files 55
7.3.2. Simulation of 8051 with External Memory. 57

8. 8051 MEMORY MAPPED I/O AND 8255A INTERFACING 61

8.1 OBJECTIVE 61
8.2 8051 EXTERNAL IO INTERFACING 61
8.3 EXPERIMENTAL PART 61
8.3.1. Memory Mapped I/O interfacing. 61
8.3.2. Interfacing 8255 to 8051 Microcontroller. 64
8.3.3. Interfacing 8086 to a stepper Motor. 66

9. DESIGN AND CODING OF AN INTELLIGENT RESTAURANT SERVICE TERMINAL 69

9.1 OBJECTIVE 69
9.2 INTRODUCTION 69
9.2.1. Installing KC51 on your drive 69
9.2.2. Starting a 8051 or 8052 project in KC51 69
9.2.3. LCD display 70
9.2.4. Serial Port 72
9.2.5. ADC interfacing 73
9.2.6. Switches and Operation of the System. 74
9.3 ABOUT KEIL C51 COMPILER 75
9.4 DESIGN REQUIREMENTS 75
9.5 REPORTING 77

10. DESIGN AND CODING OF AN INTELLIGENT HUMAN WEIGHT SCALE 79

10.1 OBJECTIVE 79
10.2 INTRODUCTION 79
10.2.1. Installing KC51 on your drive 79
10.2.2. Starting a project in KC51 for 8051 or 8052 projects. 79
10.2.3. LCD display 79
10.2.4. Serial Port 79
10.2.5. ADC interfacing 79

10.2.6. Switches and Operation of the System *79*
10.3 About Keil C51 compiler 79
10.4 Design Requirements 80
10.5 Reporting 81

11. APPENDIX 83

Complete 8086 instruction set 83
Mnemonics *83*
Operand types: *83*
Notes: *83*
Instructions in alphabetical order: *84*
Summary Sheet for Assembly Programming 96

1. TASM, EDIT, DEBUG and Emu8086 Assembler Tools

1.1 Objective

TASM is one of the well known **8086 Assembler** programs. This experiment will introduce you **TASM**, its input, and output file types.
Our objective covers hands-in experience to use

“**Notepad**” to create an assembler source file,
“**TASM**” to assemble the a source file into an object code
“**TLink**” to link an object code into an executable file.
“**TD**” and “**Emu8086**” debuggers to trace an executable file.

1.2 Introduction

Assembly language is the lowest level of symbolic programming for a computer system. It has several advantages and disadvantages over the higher level programming languages. Assembly language requires an understanding of the machine architecture, and provides huge flexibility in developing hardware/software interface programs such as interrupt service routines, and device drivers. **8086 Turbo Assembler** is one of the well known assembler programs used for PC-XT and AT family computers.

1.2.1. Editing the source file

The source for an assembly program is written into a text file with the extension -.ASM, in ASCII coding. Any ASCII text editor program can be used to write an assembly source file. We recommend to use **NOTEPAD** as a general purpose text editor, or the source editor of the **Emu86**, which is especially tailored to write **8086 Flat ASM** sources for your experiments.

1.2.2. Assembling to an object file

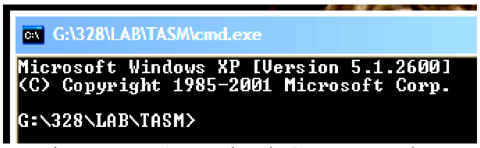


Figure 1. A typical Command Window in the Windows Environment.

Once the source file is ready for assembling, you will need **TASM** program to be executed on the source file. **TASM** is a quite old program, written for **DOS** environment. Indeed, in most embedded system application DOS operating system is preferred over Windows because Windows is unnecessary, too bulky and too expensive for most embedded applications. In the **Windows** operating system, you can invoke a **DOS** command window by running the “**CMD.EXE**” executable. Figure 1 shows a Command Window, with its typical cursor. You may change the font and the colors of the Command window by the defaults and properties dialog which is opened with a left-click on the windows title. Colors such as screen text black on white, popup text blue on gray, and fonts Lucida-Console 18 point will make your command window much more readable. Whenever you want, you can use **CLS** command of DOS to clear the screen and the screen buffer.

The Turbo Assembler program (**TASM.EXE**) can be started in the command window by writing TASM <source-file-name> , and transmitting it to DOS using the"ENTER" key. The full syntax of TASM command is:

```
>TASM [options] source [,object] [,listing] [,xref]
```

TASM command line options are shown in Table 1.

Table 1. Possible Switches of the Turbo Assembler Program.

/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
/h,/?	Display this help screen
/iPATH	Search PATH for include files
/jCMD	Jam in an assembler directive CMD (eg. /jIDEAL)
/kh#,/ks#	Hash table capacity #, String space capacity #
/l,/la *	Generate listing: l=normal listing, la=expanded listing
/ml,/mx,/mu	Case sensitivity on symbols: ml=all, mx=globals, mu=none
/n	Suppress symbol tables in listing
/p	Check for code segment overrides in protected mode
/t	Suppress messages if successful assembly
/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	Debug info: zi=full, zd=line numbers only

In DOS and Assembly programming, the names are not case-dependent, which means writing **TASM FIRST**, **Tasm first**, **tasm FIRST** or **tasm firST** does not make any difference.

Assume that you have written the following simple assembly program into a text file with the name **first.asm**. To assemble it into **first.obj** file, you shall simply write the command

```
>tasm first
```

1.2.3. Linking to an Executable or Command File

The object files contains the program code but some of the labels are still in symbolic form. A linker converts them into the executable file replacing all symbols with their corresponding values. The use of library procedures, and splitting the large programs into modules are possible since a linker can calculate a label referred from a different object file. The file first.obj is converted to an executable by the DOS command

```
>tlink first
```

Figure 2 shows typical command window message after tasm and tlink is executed.

```
G:\328\LNotes\First\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

G:\328\LNotes\First>tasm first
Turbo Assembler  Version 1.0  Copyright (c) 1988 by Borland International

Assembling file:   FIRST.ASM
Error messages:    None
Warning messages:  None
Remaining memory:  455k


G:\328\LNotes\First>tlink first
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International

G:\328\LNotes\First>
```

Figure 2 Command Window after tasm and tlink are executed.

After running Tlink, you shall find the executable file **first.exe** in your working folder. First.exe terminates with a return to DOS interrupt, without giving any message. An assembly debugging tool can trace what happens during the execution of the first.exe file.

1.2.4. Tracing and Debugging of an EXE file

Turbo Debugger, **td.exe**, is an 8086 debugging tool which gives a convenient view of the CPU status, and the memory segments. The command line syntax of TD has options, program-file-name, and arguments

`>TD [options] [program [arguments]]   -x-  = turn option x off`

The options of td.exe is shown in Table 2.

Table 2. Command Line Options for Turbo Debugger TD.EXE

-c<file>	Use configuration file <file>
-do,-dp,-ds	Screen updating: do=Other display, dp=Page flip, ds=Screen swap
-h,-?	Display this help screen
-i	Allow process id switching
-k	Allow keystroke recording
-l	Assembler startup
-m<#>	Set heap size to # kbytes
-p	Use mouse
-r	Use remote debugging
-rn<L;R>	Debug on a network with local machine L and remote machine R
-rp<#>	Set COM # port for remote link
-rs<#>	Remote link speed: 1=slowest, 2=slow, 3=medium, 4=fast
-sc	No case checking on symbols
-sd<dir>	Source file directory <dir>
-sm<#>	Set spare symbol memory to # Kbytes (max 256Kb)
-sn	Don't load symbols
-vg	Complete graphics screen save
-vn	43/50 line display not allowed
-vp	Enable EGA/VGA palette save
-w	Debug remote Windows program (must use -r as well)
-y<#>	Set overlay area size in Kb
-ye<#>	Set EMS overlay area size to # 16Kb pages

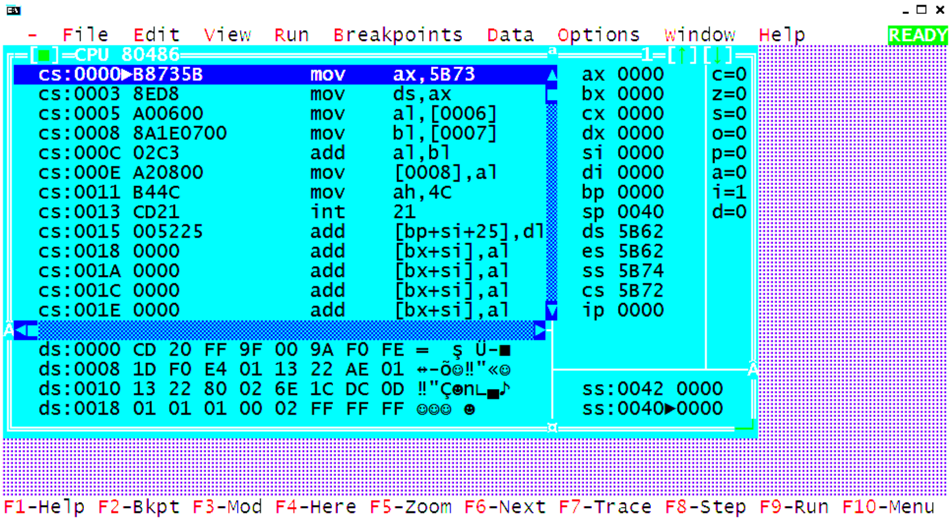


Figure 3. The turbo debugger started with first.exe file.

Entering the command

```
>td first
```

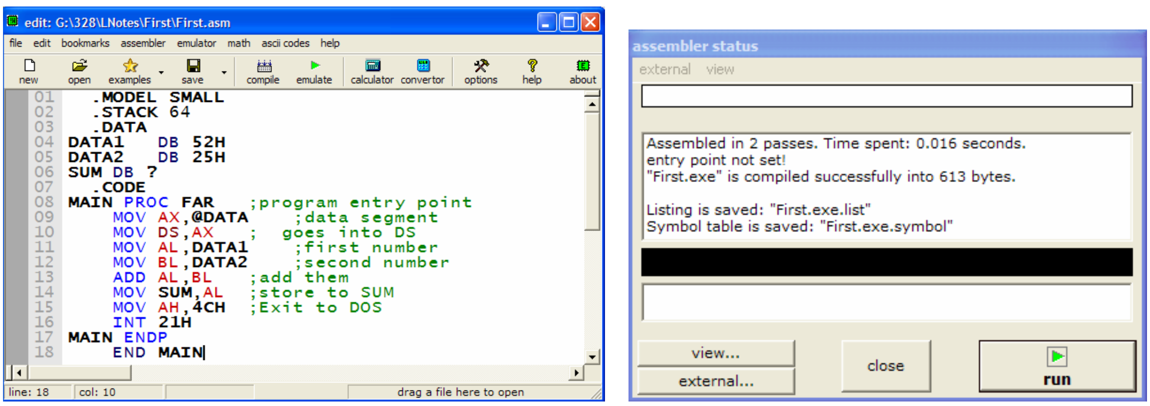
into the command window will start the debugger to load the executable **first.exe** to its memory space. The screenshot of **TD** is shown in Figure 3. In Turbo debugger, you can execute the instructions step by step and trace the execution of the code. Any message written to the screen will invoke the screen display mode to let you observe the message.

1.2.5. Emu86 IDE

An Integrated Development Environment (**IDE**) provides a convenient environment to write a source file, assemble and link it to a **-.COM** or **-.EXE** file, and trace it in both source file, and machine code. Emu86 is an educational IDE for assembly program development. You can download the latest student version of EMU86 from the web page **www.emu8086.com**. It is a Windows program, and will run by dragging an **-.ASM**, **-.OBJ**, **-.LST**, **-.EXE**, or **-.COM** file into the **emu86** shortcut icon. By this action, asm or lst files will start the 8086 assembler source editor, while obj and exe files starts the disassembler and debugger units.

1.2.6. EMU8086 Source Editor

The source editor of EMU86 is a special purpose editor which identifies the 8086 mnemonics, hexadecimal numbers and labels by different colors as seen in Figure 4.



a) b)

Figure 4. a) EMU8086 Source Editor, and b) assembler status report windows.

The compile button on the taskbar starts assembling and linking of the source file. A report window is opened after the assembling process is completed. Figure 5 shows the emulator of 8086 which gets opened by clicking on emulate button.

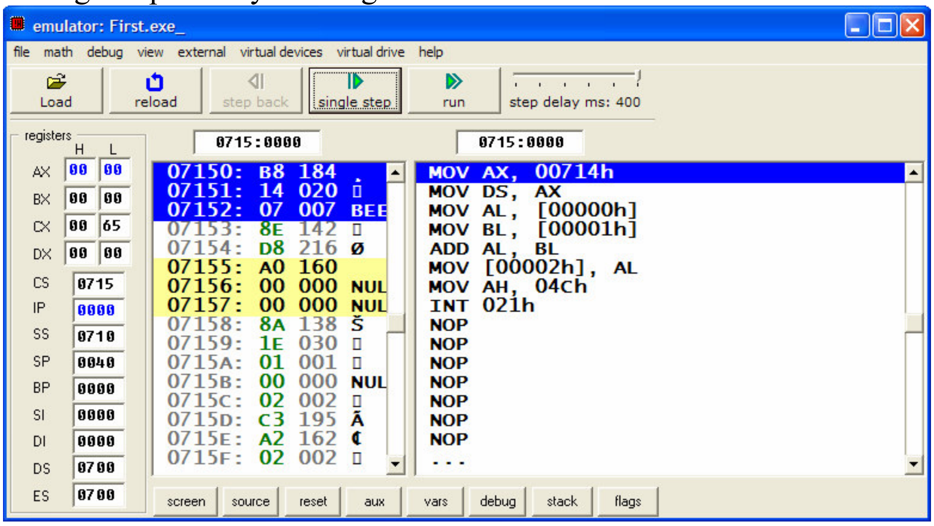


Figure 5. first.exe in the emulator window of EMU8086 debugging environment

Emul8086 environment contains templates to generate command and executable files. Another benefit of Emul8086 is its emulation of a complete system, including the floppy disk, memory, CPU, and I/O ports, which raises opportunity to write custom bios and boot programs together with all other coding of a system. More over, its help is quite useful even for a beginner of **asm** programming.

1.2.7. EMU8086 / MASM / TASM compatibility

Syntax of emu8086 is fully compatible with all major assemblers including *MASM* and *TASM*; though some directives are unique to this assembler.

1) If required to compile using any other assembler you may need to comment out these directives, and any other directives that start with a '#' sign:

```
#make_bin#
#make_boot#
#cs=...#
```

etc...

2) Emu8086 ignores the ASSUME directive. manual attachment of CS:, DS:, ES: or SS: segment prefixes is preferred, and required by emu8086 when data is in segment other then DS. for example:

```
mov ah, [bx]        ; read byte from DS:BX
mov ah, es:[bx]     ; read byte from ES:BX
```

3) emu8086 does not require to define segment when you compile segmentless COM file, however MASM and TASM may require this, for example:

```
name test
CSEG    SEGMENT     ; code segment starts here.
ORG 100h
start:  MOV AL, 5   ; some sample code...
        MOV BL, 2
        XOR AL, BL
        XOR BL, AL
        XOR AL, BL

        RET
CSEG    ENDS        ; code segment ends here.
END     start       ; stop compiler, and set entry point.
```

4) entry point for COM file should always be at 0100h, however in MASM and TASM you may need to manually set an entry point using END directive even if there is no way to set it to some other location. emu8086 works just fine, with or without it; however error message is generated if entry point is set but it is not 100h (the starting offset for com executable). the entry point of com files is always the first byte.

5) if you compile this code with Microsoft Assembler or with Borland Turbo Assembler, you should get *test.com* file (11 bytes). Right click it and select send to and emu8086. You can see that the disassembled code doesn't contain any directives and it is identical to code that emu8086 produces even without all those tricky directives.

6) emu8086 has almost 100% compatibility with other similar 16 bit assemblers. the code that is assembled by emu8086 can easily be assembled with other assemblers such as TASM or MASM, however not every code that assembles by TASM or MASM can be assembled by emu8086.

7) a template used by emu8086 to create **EXE** files is fully compatible with *MASM* and *TASM*.

8) The majority of **EXE** files produced by *MASM* are identical to those produced by *emu8086*. However, it may not be exactly the same as TASM's executables because *TASM* does not calculate the checksum, and has slightly different EXE file structure, but in general it produces quite the same machine code. There are several ways to encode the same machine instructions for the 8086 CPU, so generated machine code may vary when compiled on different compilers.

9) Emu8086 integrated assembler supports shorter versions of **byte ptr** and **word ptr**, these are: **b.** and **w.** For *MASM* and *TASM* you have to replace **w.** and **w.** with **byte ptr** and **word ptr** accordingly.

for example:

```
lea bx, var1
mov word ptr [bx], 1234h ; works everywhere.
mov  w.[bx],  1234h           ;  same  instruction  /  shorter  emu8086
syntax.
hlt

var1  db  0
var2  db  0
```

10) LABEL directive may not be supported by all assemblers, for example:

```
TEST1 LABEL BYTE
; ...
LEA DX,TEST1
```

the above code should be replaced with this alternative construction:

```
TEST1:
    ; ...
MOV DX, TEST1
```

the offset of TEST1 is loaded into DX register. this solutions works for the majority of leading assemblers.

1.3 Experimental Part

In this experiment you will use TASM, TLINK, and EMU8086 to generate an executable from an assembly source, and to trace the step-by-step execution of the executable in TD debugger and in EMU8086 emulator

1.3.1. Writing a Source File

Objective: to practice writing and editing an ASCII assembly source file using notepad.
Procedure: Generate a folder asm. Copy the files tasm.exe, tlink.exe, td.exe into asm folder. Generate a working folder with name **exp1**, and start a text file in your working folder In the explorer while folder is open

- click on right button of mouse, and
- select new, select text document. "New Text Document.txt" will be generated.
- Rename it "**exp1.asm**"

Now, you have an empty text file, with the name **exp1.asm**. Use **windows-start > all-programs > accessories > notepad** to open the Notepad text editor. Drag the file **exp1.asm** to the title-bar of the Notepad. The title will change to **exp1.asm – Notepad**. It means that you successfully opened the file **exp1.asm** for editing in notepad. Write the following source program into the edit window.

------file: exp1.asm-----

```
; STUDENT NAME and SURNAME:
; STUDENT NUMBER:

TITLE PROG2-2 (EXE) PURPOSE :ADD 4 WORDS OF DATA
PAGE  60,132
      .MODEL SMALL
      .STACK 64
;----------------------------------------------------------
      .DATA
DATA_IN     DW 234DH,1DE6H,3BC7H,566AH
      ORG 10H
SUM   DW ?
;----------------------------------------------------------
      .CODE
MAIN  PROC FAR    ;THIS IS THE PROGRAM ENTRY POINT
      MOV   AX,@DATA          ;load the data segment adress
      MOV   DS,AX             ;assign value to DS
      MOV   CX,04             ;set up loop counter CX=4
      MOV   DI,OFFSET DATA_IN ;set up data pointer DI
      MOV   SI,OFFSET SUM
      MOV   BX,00             ;initialize BX
ADD_LP:
      ADD   BX,[DI]           ;add contents pointed at by [DI] to BX
      INC   DI                ;increment DI twice
      INC   DI                ;to point to next word
      DEC   CX                ;decrement loop counter
      JNZ   ADD_LP            ;jump if loop counter not zero
      MOV   SI,OFFSET SUM    ; SI points SUM
      MOV   [SI],BX           ;store BX to SUM in data segment
      MOV   AH,4CH            ;set up return
      INT   21H               ;return to DOS
MAIN  ENDP
      END MAIN                ;this is the program exit point
```

------end of file ------

Use tabs to start the mnemonics at the same column.

Reporting:

Start a text file (you may use *notepad*) with name **exp1.txt**. Fill in the following title to your text file.

```
CMPE 323 Experiment-1 Report. <your name surname, student number>
PART1 Assembly source file
```

Copy-and-paste your exp1.asm into your report file.

```
; STUDENT NAME and SURNAME: ALI VELI
; STUDENT NUMBER:   012345

TITLE  PROG2-2 (EXE) PURPOSE :ADD 4 WORDS OF DATA
PAGE   60,132
       .MODEL SMALL
...
...
```

Keep your report file in a safe place until you complete the experiment and e-mail it to the specified address.

1.3.2. Assembling with TASM

Objective: Assembling the source file with TASM, and tracing it in TD.
Procedure: You have already written the source file **exp1.exe** .

- Organize a folder structure such as
  ASM folder contains
  files **TASM.EXE**, **TLINK.EXE**, and **TD.EXE**.
  folder **exp1**, which contains **exp1.asm** and **exp1.bat.**
-Edit **exp1.asm** to contain the complete source text by copy and paste. Fill your student name and number to the first two lines.
-Edit **exp1.bat** to have the following text lines in it.

```
..\tasm -l exp1
pause
..\tlink exp1
pause
..\td  exp1
pause
```

-Click on **exp1**.**bat** to execute assembler. You will observe a DOS window opened, and tasm executed on **exp1**.asm, with the list option active. DOS window will pause and will allow you to read the messages generated by TASM. You will observe **exp1.obj**, **exp1.lst**, and **exp1.map** files generated in folder **exp1**.
-If you press on space-bar, bat file will continue to execution, and it will execute the linker tlink on **exp1**.**obj**. Tlink will generate **exp1.exe** file into the **exp1** folder. Batch file will pause until you press the space-bar.
-Press the space-bar again to execute turbo debugger on **exp1.exe** file. In the debugger, you can trace the execution by executing each line of the assembly program stepwise.

Reporting:
In **td** read the hexadecimal contents of the program code **exp1.exe** (28 bytes), and the contents of the memory location cs:0009. Start `PART2` in your report file, and fill in (as text, i.e., `A3 02` etc)

```
PART2
B8 68 5B 8E D8 ...
cs:0009 contains ....
```

Then open exp1.lst, which is generated by turbo assembler in a text editor (notepad). Copy-and-paste the first page of the listing into your report file

```
exp1.lst contains ------------------------
Turbo Assembler  Version 1.0       01/13/11 11:32:32        Page 1
EXP1.ASM

     1                              ; STUDENT NAME and SURNAME:
     2                              ; STUDENT NUMBER:
     3
     4 0000                                  .MODEL SMALL
```

```
 0000                                 .STACK 64
                                ;--------------------------------------------------------
 0000                                 .DATA
 0000  234D 1DE6 3BC7 566A      DATA_IN DW 234DH,1DE6H,3BC7H,566AH
                                      ORG 10H
 0010  ????                     SUM     DW ?
                                ;--------------------------------------------------------
 0012                                 .CODE
 0000                           MAIN    PROC FAR        ;THIS IS THE PROGRAM ENTRY POINT
 0000  B8 0000s                         MOV     AX,@DATA                ;load the data segment address
     …
     …
```

Save your report file in a safe place until you complete the experiment and e-mail it to the specified address.

1.3.3. Assembling with Emu8086

Objective: Assembling a source file with Emu8086 assembler/emulator

Procedure:

- -Start Emu8086, and close the welcome window. Use “**open**” in taskbar to start the file browser. Select the folder **exp1**, and open **exp1.asm**.
- -Emu8086 cannot use **title**, **page**, and **org** directives. Put a semicolon to make them a comment line. Then, use emulate in taskbar to assemble, and start the emulator window with the **exp1.exe**.
- -Use the taskbar-button “**single step**” to execute each line of the assembly source.

Reporting

In `PART3` of your report answer the following questions in full sentences.

a) How many times the loop passes through the **add** instruction?

b) What is the effective address of the **add** instruction in the code segment?

After completing the experiment, write an e-mail that contains

```
Please find the attached report file of experiment 1.
Regards.
012345 Ali Veli
```

attach the report file to the e-mail and send it

- from your student-e-mail account
- to the e-mail address **cmpe323lab@gmail.com**
- with the subject: **”exp1”.**

Late and early deliveries will have 20% discount in grading. No excuse acceptable.

2.
Data Types, and Effect of ALU instructions on Flags

2.1 Objective

The aim of this experiment consists of
i- Experimenting with data types, and assembler directives.
ii- Observing the effect of ALU instructions on flags.
iii- Exercising some DOS interrupt services.

2.2 Preliminary Study

Before attending the lab, study from Mazidi&Mazidi textbook
- Section 1.4 and 2.5 to understand the data types and directives.
- Section 1.3, 1.4, and 1.5 to understand the MOV and ADD instructions, and the flags.

2.3 Experimental Part

2.3.1. Data types and Data directives

Objective: to observe the coding of several data types in various formats.
Procedure-1:

- Organize a folder structure such as
  ASM folder contains
  files **TASM.EXE**, **TLINK.EXE**, and **TD.EXE**.
  folder **exp2**, which contains **exp2p1.asm** and **exp2p1.bat.**

-Edit **exp2p1.asm** to contain the following source text by copy and paste.
Fill your student name and number to the first two data items.
---file exp2p1.asm------

```
     .model small
     .stack 64
     .data
data1 db  'Name-Surname'
data2 db  'Number'
data3 db  45, 4Ch
data4 dw  0123, 0123h
data5 dd  3, 2 dup(5)
data8 db  'Hello world! $'
     .code
     mov ax,@data
     mov ds,ax
     mov dx,offset data8
     mov ah,9
     int 21h  ; displays message
     mov ah,4ch
     int 21h  ; return to dos
     end
```

------end of file----------

In this program, `data8` is a DOS screen message, and all DOS screen messages shall terminate with a “$” character. `data8` contains the ASCII message string to be printed on the screen. `mov dx,offset data8` loads the offset of **data8** in

ds into **dx**. `mov ah,09h` determines **"print the pointed string to the screen"** service among many other DOS **int 21h** services. Similarly, **ah=4ch** selects **"exit to DOS"** service among many int 21h DOS services.

- **exp2p1.bat** should have the following text lines in it.

```
..\tasm -l exp2p1
pause
..\tlink exp2p1
pause
exp2p1
pause
```

- Execute the batch file, and press space bar to proceed with **tlink** and **exp2p1**. You will observe the message "Hello world" written on the dos command window before pressing the space bar for the third pause.
- Open the **exp2p1.lst** file in notepad to observe how the data directives place the data items into the reserved memory locations in the data segment (First start notepad, then open the file from browser, or drag the file into notepad window). You will observe the followings in the list file.

Observations-1:

1- The quoted strings are converted to ASCII coding. Check the coded characters against the following printable ASCII character table.

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
2-		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3-	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4-	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5-	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6-	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7-	p	q	r	s	t	u	v	w	x	y	z	{	|	}	→	←

2- **db** directive codes the numbers in single bytes, in the listed order.

3- **dw** directive codes the numbers in two-byte groups, in little endian convention.

4- **dd** codes the numbers in four-byte groups, in little endian convention.

5- **dup()** codes repeated number of data into data area. In the list file data is shown by dup() function. However, sufficient number of bytes are allocated for the duplicate data.

Reporting:

1- Start a text file with the name **exp2.txt**.

2- Write the Report Title in the following format

```
CMPE328 Experiment 2,  Report file by <name surname studentnr>
Part 1
```

2- Copy the data definition lines (data1 ... data8) from **lst** file to **exp2.txt**.

3- Save the text file to report the coming report item.

Procedure-2:

1- Open **exp2p1.exe** in **td** (i.e., first start td.exe, then open the file exp2p1.exe in td).

2- Right click on **ds**, and change its contents to the immediate value of the first instruction in the code segment (i.e, for **mov ax,5B68** make ds=5B68h.)

3- Click on **view > dump** to open the data segment window.

4- Right click on command window title-bar. From the pop-up menu click **edit-mark**.

5- Drag the mouse while left-clicked on data-segment dump window, to mark the ds- dump from your name to hello world message (including both lines as well).

6- While the marked area stays on the dump window, right-click on command window title-bar, and click **edit-copy** in the pop-up window. Then open **exp2.txt** in notepad, and use paste to transfer the copied text into **exp2.txt**. Your text will be similar to the following, however it will be different in some fields and addresses.

Typical exp2.txt file after Procedure-2, step-6

```
CMPE328 Experiment 2,  Report file by <name surname studentnr>
Part 1
      4 0000  4E 61 6D 65 2D 53 75 + data1 db  'Name-Surname'
      5       72 6E 61 6D 65
      6 000C  4E 75 6D 62 65 72      data2 db  'Number'
      7 0012  2D 4C                  data3 db  45, 4Ch
      8 0014  007B 0123              data4 dw  0123, 0123h
      9 0018  00000003 02*         + data5 dd  3, 2 dup(5)
     10       (00000005)
     11 004A  48 65 6C 6C 6F 20 77 + data8 db  'Hello world! $'
     12       6F 72 6C 64 21 20 24

ds:0000 4E 61 6D 65 2D 53 75 72 Name-Sur
ds:0008 6E 61 6D 65 4E 75 6D 62 nameNumb
ds:0010 65 72 2D 4C 7B 00 23 01 er-L{ #?
ds:0018 03 00 00 00 05 00 00 00 ?   ?
ds:0020 05 00 00 00 03 00 00 00 ?   ?
ds:0028 00 00 00 00 14 31 82 00     ¶1é
ds:0030 00 00 00 00 00 00 21 00       !
ds:0038 00 00 00 00 00 00 00 00
ds:0040 00 00 00 00 00 00 00 00
ds:0048 00 00 48 65 6C 6C 6F 20   Hello
ds:0050 77 6F 72 6C 64 21 20 24 world! $
```

Save **exp2.txt**, and observe the following items on the edit window.

Observations-2:

1- `data3 db 45, 4Ch` is expressed in lst file memory listing by `2D 4C` (45=2Dh).

2- `data4 dw 0123, 0123h` is converted to `007B 0123` in the lst file, but it is written in little endian convention into the memory area as `7B 00 23 01` (shown in circles).

3- `data5 dd 3, 2 dup(5)` is expressed in lst file by `00000003 02*(00000005)`, but it is filled into memory as `03 00 00 00 05 00 00 00 05 00 00 00` (in little-endian double-words, and 5 repeated twice.)

2.3.2. ALU Operations and Flags

Objective is to observe the changes of flags with the **add**, **sub**, **cmp**, **inc**, **dec**, **and, or**, **neg, mov** instructions.

Procedure:

- In this experiment you will use **Emu8086** emulator.
- Take your list of instructions from your assistant. The list will contain **add**, **sub**, **cmp**, **inc**, **dec**, **and, or**, **neg,** and **mov** instructions with immediate and register addressing modes.
- Start Emu8086 emulator. Close the welcome window. Open the file **exp2p1.asm.** Use Save-as to save it with the name **exp2p2.asm**.
- Emu8086 does not allow some data directives. Place a semicolon before **data6** and **data7** to get rid of **dq** and **dt** directives.
- Insert the code you've taken from your assistant after the **mov ds,ax** line.
- Emulate the assembler code by clicking on Emulate toolbar-button.
- In the emulator window, click on flags-button to open the flags-window.

Reporting: Use single-step button to execute each instruction. For each executed instruction in your list, fill in the flag status into the report file **exp2.txt**, i.e.,

AX and Flags you read after the instruction is executed.

```
Part 2
                       AX    CZSOPA
    mov ax,08803h  8803  000000
    add ax,07654h  FE57  001000
    sub ax,0F803h  0654  000000
    or  ax,0F000h  F654  001000
    and ax,0000Fh  0004  000000
    mov ax,0FFFFh  FFFF  000000
    inc ax         0000  010011
    dec ax         FFFF  001011
    add ax,1       0000  110011
    sub ax,1       FFFF  101011
    sub ax,08000h  7FFF  000010
    cmp ax,07000h  7FFF  000010
    cmp ax,09000h  7FFF  101110
    ...            ...
```

You shall observe

1- **mov** instructions never change any flags,

2- **inc**, and **dec** never change carry flag,

3- an immediate **sub** can do same job with **inc**, but it effects carry, and its code takes 2-bytes longer than **dec**.

4- The flags changed by each instruction is given in the 80386 instruction sheet.
add, sub, neg, cmp determine flags **CZSOPA** ;
inc, dec determine flags **ZSOPA** ;
and, or determine flags **CZSOP** ;
mov does not change any flag (it is not an ALU operation)

The flags affected by each instruction is listed in 80x86-instruction-set table.

After you complete the procedures, please save and close **exp2.txt** file, and e-mail it using your student e-mail account to cmpe323lab@gmail.com with the subject line **"exp2"** within the same day before the midnight.

Late and early deliveries will have 20% discount in grading. No excuse acceptable.

Free time practice:

Modify the program **exp2p1.asm** to replace `mov dx,offset data8` with the instruction `mov dx,offset data1`.

What do you expect to be printed on the display?

What does it display when you run the assembled exe file?

What shall you do to display only your name-surname?

3.

Simple Virtual 8086 Development Board

3.1 Objective

This experiment includes introduction to design of a virtual simple educational 8086 development board (VSED board) with simple digital i/o ports, and a UART-terminal connection. Our experimental part aims to give concepts of input and output ports with a hands on practice for verification of an executable code on a virtual simple educational 8086 system.

3.2 Introduction

3.2.1. 8086 and main memory

Virtual Simulation Model (VSM) samples in ISIS provide 8086 simulation that loads exe files to its internal memory. The executable files may be produced using any 8086 compiler including C or 8086 Assembler tools.

3.2.2. 8086 Processor Bus

ISIS provides a virtual simulation model (VSM) of 8086 including the 8086 processor bus. The simulation model provided by ISIS contains configurable internal memory which simplifies simulation of 8086 systems.

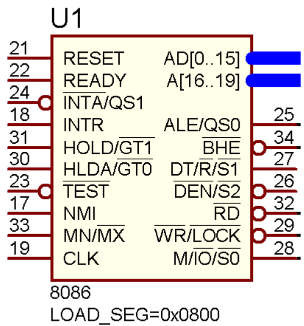


Figure 1. 8086 processor of Prosis 7.7. It contains internal memory which is configured by properties.

Bus is suitable for memory and IO interfacing. In this experiment, we plan to use it for IO interfacing.

3.2.3. Address Latching

8086 has AD0-AD15 multiplexed address lines which transfers both data and address signals. Address is valid while ALE is high, and data is valid while ALE is low and either ~RD or ~WR line is low. 74237 octal latches are suitable for address latching purpose.

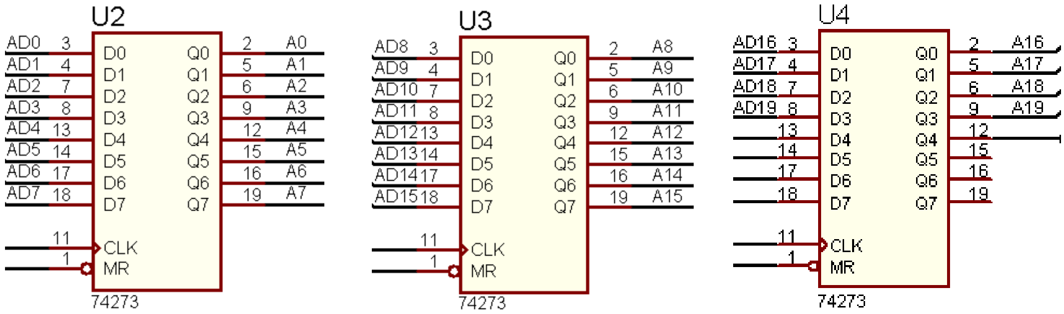


Figure 2. Address Latching Circuit for 8086 system.

CLK lines of U2, U3 and U4 are connected to ~ALE, which is obtained by inverting the ALE output (pin25) of the 8086 processor. MR is clear input of 74273 (memory reset) and all MR inputs are connected to high (Vss). The latch outputs A0 … A19 are the buffered address bus of the system. AD0 … AD15 are the unbuffered data lines of the 8086 system, and directly connected to the IO ports.

3.2.4. System Configuration

SED system has internal 64 k byte memory integrated into the 8086 device, starting from address 0x00800. The executable file shall be compiled in small model, and include its stack, data and code within the 64k memory range. The data, control and buffered address bus of 8086 is utilized to access to an 8-bit output port, two 8-bit input ports, and a universal serial asynchronous receiver transmitter (USART) unit.

3.2.5. IO Address decoding

A 74HC138 provides address decoding for the chip select signals of these IO devices.

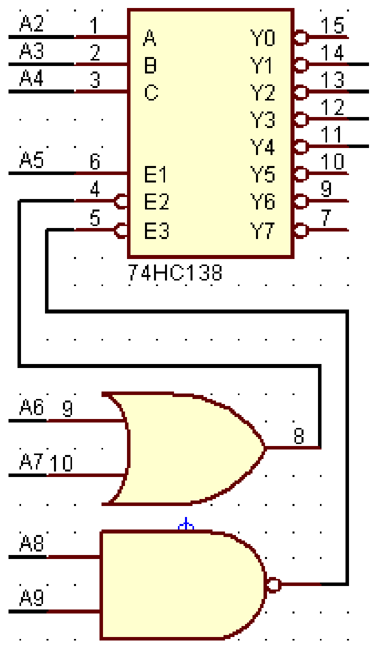


Figure 3. The IO address decoder of Small Educational Development System

The ~E3 input of 74138 (3 to 8 line decoder) gets enabled only during IO-read an IO-write bus cycles of the 8086 processor. The buffered address lines A6, A5, A4, A3, and A2 are used for enable and select inputs of the decoder. Consequently the decoding map of the decoder is obtained in Table 1.

Table 1. Address decoding map for 74138 decoder.

A9	A8	A7	A6	A5	A4	A3	A2		
E3		E2		E1	C	B	A	~Y0 … ~Y7	Enabled output
X	X	X	X	0	X	X	X	H H H H H H H H	none
X	X	X	1	X	X	X	X	H H H H H H H H	none
X	X	1	X	X	X	X	X	H H H H H H H H	none
0	X	X	X	X	X	X	X	H H H H H H H H	none
X	0	X	X	X	X	X	X	H H H H H H H H	none
1	1	0	0	1	0	0	0	L H H H H H H H	~Y0 – not connected
1	1	0	0	1	0	0	1	H L H H H H H H	~Y1 – output port UL
1	1	0	0	1	0	1	0	H H L H H H H H	~Y2 – input port – UA
1	1	0	0	1	0	1	1	H H H L H H H H	~Y3 – input port – UB
1	1	1	0	1	1	0	0	H H H H L H H H	~Y4 – USART
1	1	1	0	1	1	0	1	H H H H H L H H	~Y5 – not connected
1	1	1	0	1	1	1	0	H H H H H H L H	~Y6 – not connected
1	1	1	0	1	1	1	1	H H H H H H H L	~Y7 – not connected
1	1	X	1	X	X	X	X	H H H H H H H H	none

Thus, the 8-bit address map of Enable signals are given in Table 2.

Table 2. IO Port Addresses

A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	hex	port
1	1	0	0	1	0	0	1	X	X	324h – 327h	UL
1	1	0	0	1	0	1	0	X	X	328h – 32Bh	UA
1	1	0	0	1	0	1	1	X	X	32Ch – 32Fh	UB
1	1	0	0	1	1	0	0	X	X	330h – 333h	USART

For each IO device the first address of the address ranges are used to address the device conveniently. Simply, 324h is the address of UL, 328h and 32C are the addresses for UA and UB. We will consider the USART address later since it has two internal registers namely control and data.

3.2.6. Simple Output Port UL

The output port UL is constructed using 74273 octal D-flip-flops with common clear (~MR) and common clock (CLK) inputs. ~MR is permanently disabled by connecting it to high. The active low enable output ~Y1 of the address decoder and the active low write output of 8086 are connected to the CLK input of the port through a NOR gate to enable the clock (with a high) when both ~WR and ~Y1 are low.
In the program we use the instructions

```
mov DX,324h
out DX,AL
```

to output the contents of AL to output port UL.

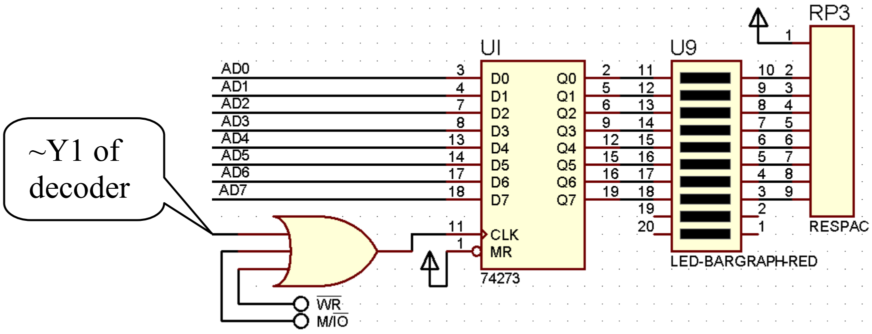


Figure 4. Simple isolated output port at address 24h installed with LED displays.

The outputs of the 74273 D-flip-flops are connected to digital LED array to display the output status in a convenient form. Note that the LED indicators glow while the latch outputs are high. For example, with the instructions

```
mov DX,324h
mov AL, 03h
out DX, AL
```

After the execution of the code LEDs of Q0 and Q1 shall remain dark, and Q3, Q4, Q5, Q6, and Q7 shall start to glow.

3.2.7. Simple Input Ports UA and UB

Input Ports UA and UB are designed to read the 8-bit dip-switch status into register AL. The instructions

```
mov DX,328h
in AL,DX
```

and

```
mov DX,32Ch
in AL,DX
```

read the status of the switches SW1 and SW2 into AL.

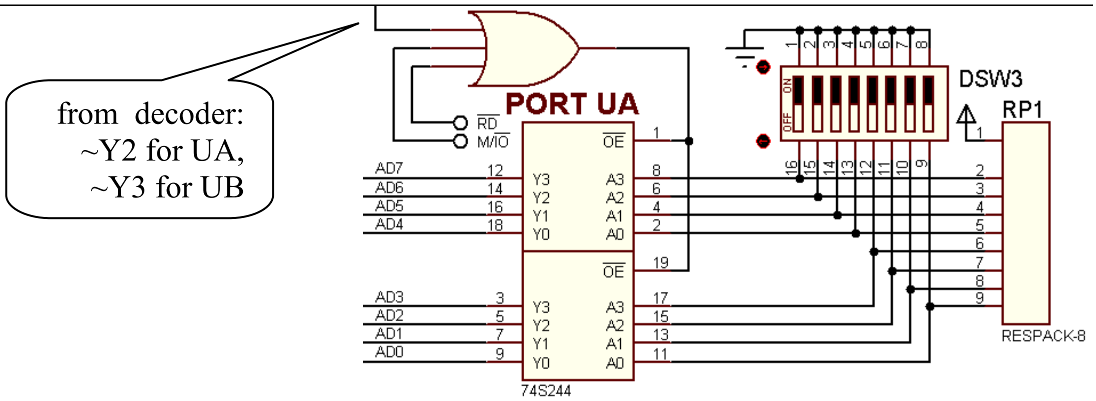
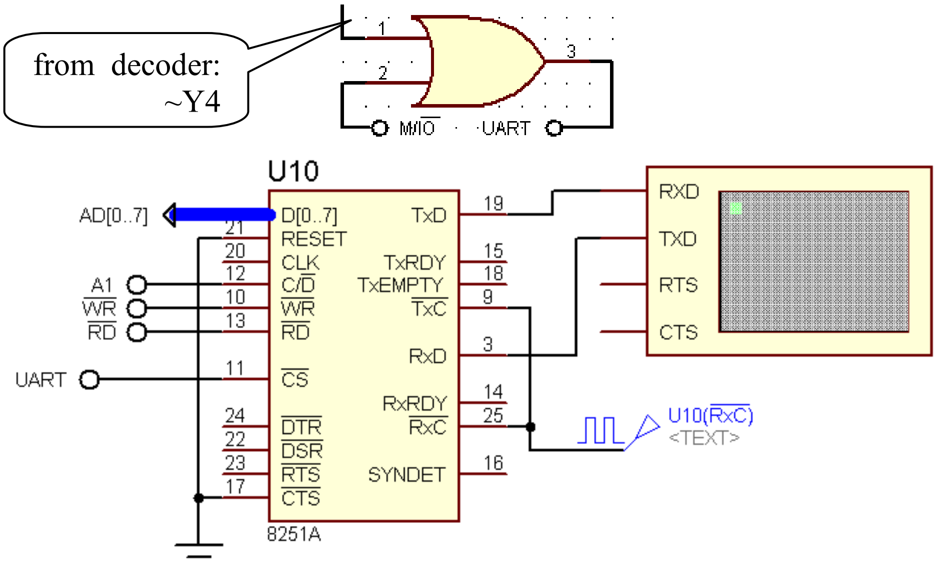


Figure 5. Simple isolated input port at address 328h and 32Ch installed with switch array.

For example, if the switch positions of SW1 were set to On, On, On, Off, On, On, Off, On (in the order from 1 to 8) and the instruction `in AL,28h` was executed the corresponding bit of AL for On position contains 0, and for Off position it will be 1, resulting in AL=12h.

3.2.8. Serial Communication Device

The USART 8251A is enabled by ~Y4 of the address decoder, and additionally it has a Control/~Data select line which is connected to A1. Moreover, the ~RD and ~WR lines provide reading and writing to control and data registers



Consequently it has the following address mapping

A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	In/Out	hex address	addressed port
1	1	0	0	1	1	0	0	0	X	Out	330h – 331h	USART data out
1	1	0	0	1	1	0	0	0	X	In	330h – 331h	USART data in
1	1	0	0	1	1	0	0	1	X	Out	332h – 333h	USART control
1	1	0	0	1	1	0	0	1	X	In	332h – 333h	USART status

USART has configuration registers which needs initialization. The Reset sequence of the USART provides safe reset of the device under the control of program.

```
xor AL, AL
```

```
      mov DX, 332h
      out DX, AL
      out DX, AL
      out DX, AL
      mov AL, 40h
      out DX, AL
; After reset sequence, USART expects the mode control,
; 8251 Mode=sdppbbmm,
;   async mode << sd=00,
;   no parity << pp=00;
;   data-bits: 5<<bb=00; 6<<bb=01; 7<<bb=10; 8<<bb=11;
;   baud rate factor: x1<<mm=01; x16<<mm=10; x64<<mm=11;
      mov AL, 0Dh    ; mode8251 8-bit, no parity, baud=clock x1
      out DX, AL
; Next, USART waits command control
; 8251 Command = hmrtRdT
;   search SYN char: disable<<h=0 (async mode); enable<<h=1
;   internal reset:  reset (expects mode) << m=1; command << m=0;
;   request to send: forces RTS low << r=1;
;   error reset : resets all error flags << r=1;
;   send break:  forces TxD low << t=1;
;   receive enable: enable << R=1;
;   data terminal ready: forces DTR low << d=1
;   transmit enable: enable << T=1;
      mov   AL, 37h    ; comd8251 both RC & TX, reset errors, RTS, DTR
active
      out DX, AL
```

After this initialization code, USART is ready to transmit characters by putting them into data-out register. It is possible to poll the status register to check the data-out and data-in registers are full or empty. User may get the received character from data-in register when bit-1 of status register is high, and may write the character to be transmitted into the data-out if bit-0 of the status register is high.

```
; This code reads received character into AL.
; If no character received then AL returns zero.
      mov DX,332h   ; status/control address
      in AL, DX     ; read status register
      test AL,01h   ; zero flag is set if AL .AND. 01h is nonzero
      jz NotReceived
      mov DX,330h   ; data-in/data-out address
      in AL, DX     ; read received bits from data-in into AL.
      shr AL,1      ; Purge out the start bit, remaining bits are data.
NotReceived:
; Any code that process the received character shall be placed here.
```

Data transmission through USART is obtained by writing character into data-out register after USART unit is ready for transmission of a character

```
; this code transmits the contents of AH register to USART.
WaitReady:
      mov DX,332h   ; status/control address
      in AL, DX     ; read status register
      test AL,02h    ; zero flag is set if AL .AND. 02h is nonzero
      jz WaitReady   ; Wait until flag is set
      mov AL,AH
      mov DX,330h   ; data-in/data-out address
      out DX,AL     ; received character transferred into AL.
```

In most applications serial io is managed through an input and an output buffer. USART generates an interrupt request whenever a character is received or transmission of data-out buffer is over. The related interrupt service routine transfers the received character from the data-in register to the input buffer, and it transfers any characters from the output buffer to the data-out register.

3.3 Experimental Part

In this experiment you will write and assemble short programs using 8086 instructions **`in`**, **`out`**, **`mov`**, **`add`**, **`jmp`**, **`test`**, **`jz`**, **`jnz`** instructions, and you will use **EMU8086** assembler/emulator to obtain its executable code. Next, you will verify the executable code by PROSIS simulation of a virtual simple 8086 educational development system.

At the first part of the experiment we will write a code to display either **`num1`** or **`num2`** on the LED array depending on the bit-0 switch status of port UA. At the second part, we will display the sum of the two numbers switch status

3.3.1. Execution of a code on a virtual 8086 system

Procedure:

-Start Emu8086, and close the welcome window. Write the following program into the new-source window of the Emu8086 editor.

```
; Your Student Number, Name, Surname . . . . . .
; CMPE323 Lab-1 Simple I/O port with 8-bit addressing
.MODEL    SMALL
.8086
.CODE
  mov ax,@DATA
  mov DS,ax
W1:
  mov dx,328h
  in  al,dx
  test al,01h
  mov al,num1
  jz W2
  mov al,num2
W2:
  mov dx,324h
  out dx,al
  jmp W1
.stack
.data
num1 db 20
num2 db 30
END
```

-Save the file to your work-folder with the file name **exp3A.asm**

-Use the taskbar-button "**compile**" to assemble your source to **exp3A.exe** into your working folder.

-Start ISIS and load the design file **VSED_WA.dsn** (drag and drop it into ISIS window).

- R-click (right click) on 8086 processor on the system diagram. 8086 will be selected and turned to red, and a pop-up menu will appear. L-click (left-click) mouse on **Edit Properties** to open **Edit Component** window. Change the program file browsing **exp3A.exe**. R-click mouse on OK to close **Edit Component** window. R-click mouse on any empty part of the diagram window to unselect the processor.

-From ISIS simulation bar  L-click on **step** button (2nd button) to start debugging. From the ISIS menu-bar L-click on **debug >> 8086 >> registers** to open register window. On the register window **R-click >> set font >> Lucida Console / Bold / 12** to make the font readable. L-clicking on step button will execute each instruction and update the registers accordingly. Trace the program while PORT UA A0 switch is at on position and at off position. On your report sheet write the instruction pointer contents and the instructions for each step of execution until IP becomes 0005 for the second time.

Reporting:

1- Start a text file with the name **exp3.txt**.

2- Write the Report Title in the following format

```
CMPE328 Experiment 3,  Report file by <name surname studentnr>
Part 1
```

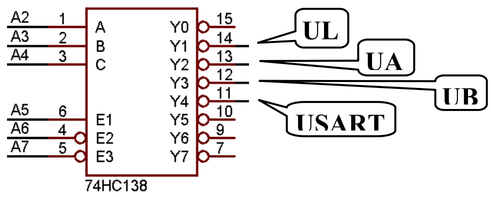
3- Open the list file **exp3A.exe.list** and use copy-and-paste to copy it into your report file.

4- Save **exp3.txt** to report the coming report item.

3.3.2. Adding Port UA and Port UB

This experiment uses a different board, **VSED_BA.dsn**, with an 8-bit IO address decoder for port addresses

It may be obtained from the 16-bit IO addressed **VSED_WA.dsn** circuit by removing the AND and OR gates which are connected to ~E2 and ~E3 of 74HC138, and connecting A6 and A6 to ~E2 and ~E3 lines so that decoder is enabled when (A7A6A5A4) is (001x).



A7	A6	A5	A4	A3	A2	A1	A0	hex	port
0	0	1	0	0	1	X	X	24h – 27h	UL
0	0	1	0	1	0	X	X	28h – 2Bh	UA
0	0	1	0	1	1	X	X	2Ch – 2Fh	UB
0	0	1	1	0	0	X	X	30h – 33h	USART

Procedure:

-Start Emu8086, and close the welcome window. Write the following program into the new-source window of the Emu8086 editor.

```
; Your Student Number, Name, Surname . . . . . .
; CMPE323 Lab-1B Simple I/O port with 8-bit addressing
.MODEL    SMALL
.8086
.CODE
  mov ax,@data
  mov ds,ax
W1:
  in  al,28h ; first number from UA
  mov ah,al
  in  al,2Ch ; second number from UB
  add al,ah
  out 24h,al
  jmp W1
.stack
.data
```

```
END
```

-Save the file to your work-folder with the file name **exp1B.asm**

-Use the taskbar-button "**compile**" to assemble your source to **exp1B.exe** into your working folder.

-Start ISIS and load the design file (simply drag and drop it into ISIS window.

- R-click (right click) on 8086 processor on the system diagram. 8086 will be selected and turned to red, and a pop-up menu will appear. L-click (left-click) on **Edit Properties** to open **Edit Component** window. Change the program file browsing **exp1B.exe**. R-click on OK to close **Edit Component** window. R-click on any empty part of the diagram window to de-select the processor.

-From ISIS simulation bar  L-click on **step** button (2nd button) to start debugging. From the ISIS menu-bar L-click on **debug** >> **8086** >> **registers** to open register window. If the font is too small to read then R-click on the register window, select **set font >> Lucida Console / Bold / 12** to make the font readable. L-clicking on **step** button will execute each instruction and update the registers accordingly. Trace the program to add the last two digit of your student number to the third&fourth digits in hexadecimal format. For example if your student number is 123456, then you shall write 34h to port UA, and 56h to port UB. Read the result from the LEDs of port UL.

Reporting

Write your observations into `PART2` of your report file in full sentences. (i.e., "`I set port UA to 34h by making (AD7..AD4)=0011, (AD3..AD2)=0100, and port UB to 56h by making (AD7..AD4)=0101, (AD3..AD2)=0110. Then, I read from port UL Q0=0, Q1=1, Q2=0, Q3=1,Q4=0,Q5=0,Q6=0, Q7=1, which makes in binary 10001010 = 8Ah.`")

3.3.3. USART and Capitalization

Procedure:

-Start Emu8086, and close the welcome window. Write the following program into the new-source window of the Emu8086 editor.

```
; Your Student Number, Name, Surname . . . . . .
; CMPE323 Lab-1C Serial Communication
.MODEL      SMALL
.8086
.CODE
  mov AX,@data
  mov DS,AX
  call InitUSART
  ; Convert all characters to Upper Case
MainLoop:
  mov BX,offset inbfr
  mov CX,0
Recv:
  call RecvChar ; character is in AL
  cmp AL,0
  jz Recv       ; no character
  mov [BX],AL   ; put chr into buffer
  inc BX        ; point empty byte in buffer
  inc CX        ; keep number of received chars
  mov DX,324h   ; LED-display
  out DX,AL
  cmp AL,0Dh    ; is the character line feed
  jnz Recv      ; if not line feed receive next char.
```

```
; transmit the buffer after making upper case
  mov BX,offset inbfr
Txmt:
  mov AH,[BX]    ; character from the buffer
  inc BX         ; point next char.
  cmp AH,'a'     ; is it lower case alphabetic
  jb transmitchar
  cmp AH,'z'
  ja transmitchar
  and AH,0DFh    ; now the character is uppercase
transmitchar:
  call XmitChar ; Transmit the processed character.
  mov  AX,200
delay:
  dec  AX
  jnz delay
  loop Txmt
  jmp MainLoop

InitUSART proc
  xor AL, AL
  mov DX, 332h
  out DX, AL
  out DX, AL
  out DX, AL
  mov AL, 40h
  out DX, AL
  mov AL, 04Dh  ; 8-bit, no parity, baud=clock x1
  out DX, AL
  mov AL, 05h   ;  start both receive and transmit
  out DX, AL
  ret
  endp

RecvChar proc
; reads received character into AL.
; If no character received then AL returns zero.
  push DX
  mov DX,332h    ; status/control address
  in  AL,DX      ; read status register
  and AL,02h     ; zero flag is set if AL .AND. 01h is nonzero
  jz NotReceived
  mov DX,330h    ; data-in/data-out address
  in  AL,DX      ; received character transferred from data-in into AL.
  shr AL,1
NotReceived:
  pop DX
  ret
  endp

XmitChar proc
; transmits the contents of AH register to USART.
  push DX
  mov DX,332h    ; status/control address
  in  AL,DX      ; read status register
  and AL,01h    ; zero flag is set if AL .AND. 02h is nonzero
  jz XmitChar    ; Wait until flag is set
  mov AL,AH
  mov DX,330h    ; data-in/data-out address
```

```
  out DX,AL     ; received character transferred into AL.
  pop DX
  ret
  endp
.data
bptr  dw 0102h
inbfr db 0 dup(32)
.stack 32

 END
```

-Save the file to your work-folder with the file name **exp1C.asm**

-Use the taskbar-button "**compile**" to assemble your source to **exp1C.exe** into your working folder.

-Start ISIS and load the design file **VSED_WA.dsn** (drag and drop it into ISIS window).

- Rclick (right click) on 8086 processor on the system diagram. 8086 will be selected and turned to red, and a pop-up menu will appear. Lclick (left-click) on **Edit Properties** to open **Edit Component** window. Change the program file browsing **exp1B.exe**. Rclick on OK to close **Edit Component** window. Right-click on any empty part of the diagram window to de-select the processor.

-From ISIS simulation bar  Lclick on **run** button (1nd button) to start execution.

-If the terminal page does not appear on the screen then Lclick on ISIS-menu-bar-**debug** >> **virtual terminal** to open terminal monitor window. Right-Click into the terminal window and check "Echo typed characters". -If the font is too small to read then right-click on the terminal window, select **set font >> Lucida Console / Bold / 12** to make the font readable.

- Click on terminal window, and then use keyboard to write Hello, and end the line with return (enter-key). You shall see

```
Hello
HELLO
```

on the monitor. The first character of each character pair is what you entered from keyboard echoed on the monitor, and the second character is the character sent from 8086 code.

- If you have the oscilloscope settings **horizontal** (sweep-time) at 1ms/div, **Channel-A** and **Channel-B** at DC 2V/div, **trigger** at DC with source A, at level 20, negative edge, and Auto-mode then you may observe the received and transmitted waveform of serial signal on the scope window.

Write your name in lower-case characters, set the trigger of scope to one-shot, and then send the return character to catch the transmitted string from USART to terminal.

Reporting:

Use Oscilloscope to measure the total time period to transmit your name, and write it in full sentence into PART3 of your report (i.e., `I entered my name "Ali Veli" and set the oscilloscope to one-shot trigger mode. After I sent a return character I used cursor to measure total transmission time T=34.25ms at time-base setting 5ms/div`).

After you complete the procedures, please save and close **exp3.txt** file, and e-mail it using your student e-mail account to **cmpe323lab@gmail.com** with the subject line **"exp3"** within the same day before the midnight.

4.
BIOS and DOS Services

4.1 Objective

The aim of this experiment consists of
i- Exercising keyboard and screen related BIOS and DOS interrupt services.
ii- Coding with macros and procedures
iii- Using include files.

4.2 Preliminary Study

Before attending the lab, study from Mazidi&Mazidi textbook

- Section 2.3 and 2.4 to understand Control Transfer Instructions.
- Section 3.4 to understand BCD, packed-BCD, ASCII-decimal, representation of numbers.
- Section 4.1 BIOS interrupt service to clear the screen.
- Section 4.2 DOS interrupt services to display a single character, to display a string, to input a single character, and to display a string.
- Section 4.3 DOS Keyboard interrupt service to test the keyboard buffer, and return the pressed key.
- Section 5.1 MACRO definitions, and include files

4.3 Experimental Part

4.3.1. DOS services for String Display and Input

Objective: to observe the coding of several data types in various formats.
Procedure-1:

- Organize a folder **exp4** under your **asm** folder.

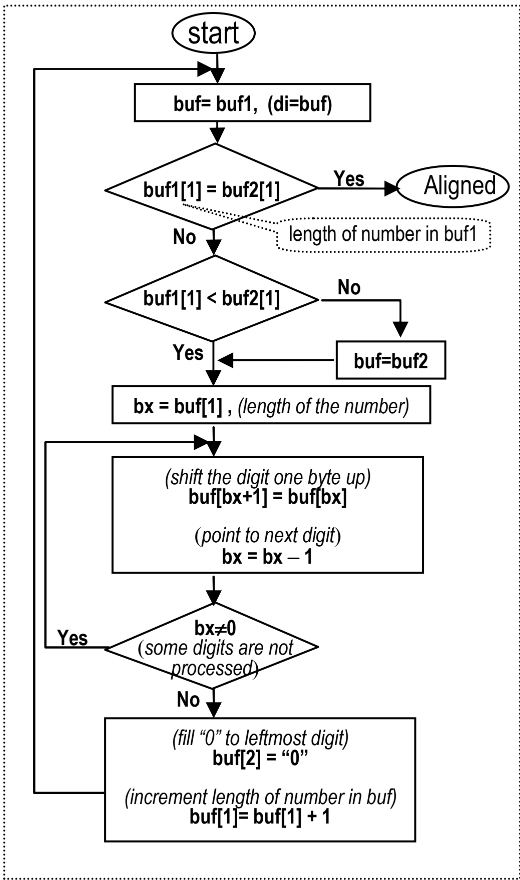
- In exp4 folder, create and edit **exp4p1.asm** to contain the following source text (please use copy and paste, but correct all mistakes in the code. Do not forget to fill in your student number to the first line of the source code).

---file exp4p1.asm------

```
; exp4p1 student nr:
.model small
.stack 64
.data
msg1 db 13,10,"I will add two numbers."
msg2 db 13,10,"  Give me one number:$"
msg3 db 13,10,"  Give me second one:$"
msg4 db 13,10,"         The sum is "
sum  db "        $"
buf1  db 10,0,"          "
buf2  db 10,0,"          "
.code
start:
  mov ax,@data
  mov ds,ax
;display msg1 and msg2
  mov ah, 09h
  mov dx, offset msg1
  int 21h
;input the first number
  mov ah, 0Ah
```

```
      mov dx, offset buf1
      int 21h
    ;display msg3
      mov ah, 09h
      mov dx, offset msg3
      int 21h
    ;input the second number
      mov ah, 0Ah
      mov dx, offset buf2
      int 21h
    ; align the numbers
      mov di, offset buf1+1
      mov si, offset buf2+1
      cmplengths:
       mov al, [di]
       cmp al, [si]
       je aligned
       jb shiftbuf1
         ;swap buffers
         mov ax,di
         mov di, si
         mov si,ax
       shiftbuf1:
       xor bh,bh
       mov bl,[di]
       shiftloop:
         mov al,[bx][di]
         mov [bx][di]+1,al
         dec bl
         js  endloop
         jnz shiftloop
       endloop:
       mov [di]+1,'0'
       inc [di]
       jmp cmplengths
    aligned:
       mov bx,offset sum
       xor ch,ch
       mov cl,[di]
       add di,cx
       add si,cx
       add bx,cx
       clc
       addloop:
         mov al,[di]
         adc al,[si]
         aaa
         pushf      ; save flags
         or  al,30h ; make it ASCII
         mov [bx],al
         dec si
         dec di
         dec bx
         popf       ; restore flags
         loop addloop
       mov ah,09h
       mov dx,offset msg4
       int 21h

       mov ah,4ch
       int 21h
       end
    ------end of file----------
```



In this program **buf1** and **buf2** are input string buffers. An input-string buffer consists of three fields.

The **first byte** of the buffer is single byte **buffer-size field**.

The **second byte** is single-byte **input-string-length field**.

The remaining bytes are reserved for the **ASCII-coded-input-string**.

- You will use EMU8086 in tracing the assembly code. Open **exp4p1.asm** in EMU8086.

- Ask to your Lab-assistant the first and second numbers to be used in tracing the code. Start the emulation, and go in single steps until you will get the message "waiting for input" on the emulator window.
- Switch to the screen by clicking the screen-button on the emulator window. Then write the first number, and press enter-key to complete the string-input service. In the emulator window "waiting …" message will disappear.

- Continue to single step emulation and enter the second number.

- Now, open variables window (by clicking the var button).

- In the variables window, click on buf1, and make its size qword. Then make both buf2 and sum qword as well.

- Write the **qword values** of **buf1** and **buf2** into the report file **exp4.txt**, as shown below:

```
CMPE328 Experiment 4 Report file by <Name-Surname> <number>
 Part-1
  buf1:  0A……………… h
  buf2:  0A……………… h
```

- Continue to tracing until it reaches to JB instruction. Does it execute "`mov ax,di`", or "`xor bh,bh`" after the `jb` instruction. Write this first instruction that is executed after `jb` to the **exp3.txt** file (either mov, or xor).

```
  after jb   ………  is executed.
```

- Continue to tracing until it reaches to "`xor ch,ch`" instruction. Open the variables window, and write the new **qword values** of **buf1** and **buf2** to the **exp3.txt** file.

```
  after aligned:
  buf1:  0A……………… h
  buf2:  0A……………… h
```

- Run the code to the end (use run button). Then, in the variable window find the qword value of sum, and write it into **exp4.txt.**

```
  sum:     ………………… h
```

4.3.2. Subroutines and Include files.

Objectives:

-to observe usage of macros in improving the readability of the assembly sources.
-to make and use an include file for the subroutines.

Procedure-1:

-The following assembly code finds the maximum and the minimum of an array of two digit decimal numbers (i.e., numbers between 0 and 99). Write it into **exp4p2.asm** in the **exp4** folder. Don't forget to fill your name and number into the first line of the file.

```
; exp4p2.asm  student name and number :
.MODEL SMALL
.STACK 100h
.DATA
MESSAGE1 DB 13,10,'   The smallest is: '
SMALLEST DB '     '
MESSAGE2 DB 13,10,"   The biggest is:  "
BIGGEST  DB '     $'
MESSAGE3 DB    ?
```

```
NUMCOUNT EQU 6
NUMBERS  DB 51,98,2,18,11,40
ROW      EQU 08
COLUMN   EQU 05
.CODE
MAIN PROC FAR
      MOV AX,@DATA
      MOV DS,AX
      MOV SI,OFFSET MESSAGE3
      CALL CLEAR
       MOV DL,COLUMN
       MOV DH,ROW
      CALL CURSOR

      MOV CX, NUMCOUNT-1

      MOV DI, OFFSET NUMBERS
      MOV SI, DI ; [SI] is smallest
      MOV BX, DI ; [BX] is biggest
BACK: INC DI
; is [DI]<[SI]
      MOV AL,[DI]
      CMP AL,[SI]
      JAE BIG     ; skip if big
      MOV SI, DI  ; update if small
      JMP SML
; is [DI]>[BX]
BIG:  MOV AL,[DI]
      CMP AL,[BX]
      JB  SML
      MOV BX, DI
SML:  LOOP BACK
      mov AL,[SI]
      mov AH,0
      call HEX2ASCII
      xchg AH,AL ; ascii strings big-endian
      mov WORD PTR SMALLEST,ax
      mov AL,[BX]
      mov AH,0
      call HEX2ASCII
      xchg AH,AL ; ascii strings big-endian
      mov WORD PTR BIGGEST,ax

      mov DX, OFFSET MESSAGE1
      CALL SCREEN
      MOV AH,4CH
      INT 21H
MAIN ENDP
;-------------------------------------------------
HEX2ASCII PROC
; converts ah=0, al=binary_number to ax=ascii number
AGAIN:
      CMP AL,10
      JB  CONVERTED
      sub al,10
      inc AH
      jmp AGAIN
CONVERTED:
      or  ax,3030h
      ret
HEX2ASCII endp
;-------------------------------------------------
CLEAR PROC
; clears 25rows,80cols screen
      MOV      AX,0600H          ;scroll the entire page
      MOV      BH,0F0h           ;normal attribute
      MOV      CX,0000           ;row and column of top left
      MOV      DX,184FH          ;row and column of bottom right
      INT      10H                   ;invoke the video BIOS service
      RET
CLEAR   ENDP
;-------------------------------------------------
CURSOR  PROC                         ;SET CURSOR POSITION
; sets cursor to DH=row,DL=col.
     MOV       AH,02
     MOV       BH,00
     INT       10H
```

```
         RET
CURSOR  ENDP
;------------------------------------------------
SCREEN  PROC
; displays a $-terminated string pointed by DH.
         MOV      AH,09
         INT      21H
         RET
SCREEN  ENDP
END MAIN
```

- You will use Emu8086 to trace this assembly code. Open **exp4p2.asm** in the Emu8086, and replace the data entries **NUMCOUNT** and **NUMBERS** with the data supplied to you by your lab instructor.

- Click the emulate button to start emulation. Then click the aux button and select listing to open the list file. debug button in the emulator windows to open the debug listing. Use Ctrl-A, and then Ctrl-C to copy the debug listing into clipboard. Then paste them to the end of the reporting file **exp4.txt**. The added text will look like the following text.

```
EMU8086 GENERATED LISTING. MACHINE CODE <- SOURCE.

exp4p2.exe_ -- emu8086 assembler version: 4.05

[ 3/23/2008  --  23:18:53 ]

===============================================================
[LINE]     LOC: MACHINE CODE                          SOURCE
===============================================================

[   1]        :                                       .MODEL SMALL
[   2]        :                                       .STACK 100h
[   3]        :                                       .DATA
[   4]    0100: 0D 0A 20 20 20 54 68 65 20 73 6D 61   MESSAGE1 DB 13,10,'   The smallest is: '
                6C 6C 65 73 74 20 69 73 3A 20
 . . .  . . .
```

- Now, you shall build an include file with the name **"exp4p2b.asm"**.
  First save the file exp4p2.asm twice with the new names exp4p2a.asm and exp4p2b.asm.
  In **exp4p2a.asm**, delete the procedures **HEX2ASCII, CLEAR, CURSOR, SCREEN** and insert a line after **MAIN ENDP** that contains `include exp4p2b.asm` , i.e.,

```
         ...  ...
        MOV AH,4CH
        INT 21H
        MAIN ENDP
include myproc.asm
        END MAIN
```

In **exp4p2b.asm** leave only the procedures **HEX2ASCII, CLEAR, CURSOR, SCREEN**, so that it will look like

```
;------------------------------------------------
HEX2ASCII PROC
; converts ah=0, al=binary_number to ax=ascii number
AGAIN:
        CMP AL,10
. . .
. . .
        RET
CURSOR  ENDP
;------------------------------------------------
SCREEN  PROC
; displays a $-terminated string pointed by DH.
        MOV      AH,09
        INT      21H
        RET
SCREEN  ENDP
```

- Now open **exp4p2a.asm** in Emu8086, emulate and run. You will observe that it runs the same as the single-file source code. In the listing of **exp4p2a.asm**, the included code will appear missing. Copy all listing to **exp4.txt**.

Reporting:

After you complete the procedures, please save and close **exp4.txt** file, and e-mail it using your student e-mail account to cmpe323lab@gmail.com with the subject line **"exp4"** within the same day before the midnight.

Late and early deliveries will have 20% discount in grading. No excuse acceptable.

Free time practice-1:

In your free time, convert the code exp4p2.asm to two files: File exp4p2c.asm that contains source code invoking macros, and file exp4p2c.mac that contains macro definitions. Instead of converting the procedures into parameterless macros, try to include necessary calling parameters as well into the definition of macro i.e.,

Table-1 Converting subroutines to macros with parameters.

CURSOR PROC ;SET CURSOR POSITION
MOV AH,02H
MOV BH,00
INT 10H
RET
CURSOR ENDP
	⇨	CURSOR MACRO ROW, COL
		;SET CURSOR POSITION
		MOV DH, ROW
		MOV DL, COL
		MOV AH,02H
		MOV BH,00
		INT 10H
		CURSOR ENDM

SCREEN PROC MOV AH,09 INT 21H RET SCREEN ENDP	⇨	SCREEN MACRO STROFFSET MOV DX,offset STROFFSET MOV AH,09 INT 21H SCREEN ENDM

Then, you need also modifications in **exp4p2c.asm** for invoking the macros

Table-2 Invoking macros with parameters instead of parameters passed in register.

MOV DL,COLUMN
MOV DH,ROW
CALL CURSOR
	⇨	CURSOR ROW,COLUMN

mov DX, OFFSET MESSAGE1 CALL SCREEN	⇨	SCREEN MESSAGE1

5.
Using Signed Numbers and Look-up Tables

5.1 Objective

The aim of this experiment is
i- Coding with macro and procedure libraries
ii- Using signed numbers in calculations.
iii- Using Look-Up Tables.

5.2 Preliminary Study

Before attending the lab, study from Mazidi&Mazidi textbook
- Section 2.3 and 2.4 to understand Control Transfer Instructions.
- Section 4.1 BIOS interrupt service to clear the screen.
- Section 4.2 DOS interrupt services to display a single character, to display a string, to input a single character, and to display a string.
- Section 4.3 DOS Keyboard interrupt service to test the keyboard buffer, and return the pressed key.
- Section 5.1 MACRO definitions, and include files
- Section 6.1 For signed integer arithmetic operations

5.3 Experimental Part

5.3.1. Macro Library for BIOS and DOS Services

Objective: to use a macro library for BIOS and DOS service.
Procedure-1:
- Organize a folder **exp5** under your **asm** folder.
- In exp5 folder, create and edit **exp5.inc** to contain the following source text (please use copy and paste, but correct all mistakes in the code. Do not forget to fill in your student numbers to the first line of the source code).

-----file exp5.inc------

```
; MACRO Library exp5
; student nr1:
; student nr2:

; ASCII code for carriage return
CR equ 0Dh
; ASCII code for line feed
LF equ 0Ah

al2asc macro buffer
; al to ascii-decimal conversion
   xor ah,ah
   mov cx,100*256+10
   div ch
   mov buffer,al
   or  buffer,30h
   mov al,ah
   xor ah,ah
   div cl
```

```
   mov buffer+1,al
   or  buffer+1,30h
   mov buffer+2,ah
   or  buffer+2,30h
   mov buffer+3,'$'
al2asc endm


asc2al macro buf
;converts ascii str to number in al
   local hexnumber,numer1,numer2,negative,completed
   mov bl,byte ptr buf+1 ; size of the string
   mov bh,0
   mov al,[bx + offset buf+1]
   or  al,20h ; lowercased
   cmp al,'h'
   je  hexnumber
;number is decimal
   and al,0Fh
   mov cl,al
   dec bx
   je  completed
   mov al,[bx + offset buf+1]
   cmp al,'-'
   je  negative
   and al,0Fh
   mov ch,10
   mul ch
   add cl,al
   dec bx
   je  completed
   mov al,[bx + offset buf+1]
   cmp al,'-'
   je  negative
   and al,0Fh
   mov ch,100
   mul ch
   add cl,al
   dec bx
   je  completed
   mov al,[bx + offset buf+1]
   cmp al,'-'
   je  negative
   jmp completed
hexnumber:
   dec bx
   je  completed
   mov al,[bx + offset buf+1]
   cmp al,'9'
   jna numer1
   add al,9 ; letter correction
numer1:
   and al,0Fh
   mov cl,al
   dec bx
   je  completed
   mov al,[bx + offset buf+1]
   cmp al,'-'
   je  negative
   cmp al,'9'
   jna numer2
   add al,9 ; letter correction
numer2:
   and al,0Fh
   mov ch,16
   mul ch
   add cl,al
   dec bx
   je  completed
   mov al,[bx + offset buf+1]
   cmp al,'-'
   je  negative
   jmp completed
negative:
   neg cl
completed:
   mov al,cl
asc2al endm


dispclr macro
   mov    ax,0600h
   mov    bh,0F0h
   mov    cx,0000
   mov    dx,184Fh
   int    10h
dispclr endm
```

```
dispstr macro string
  mov ah, 09h
  mov dx, offset string
  int 21h
dispstr endm

imultx macro prod,op1,op2
  mov ax,op1
  cwd
  mov cx,op2
  imul cx
  mov prod,ax
imultx endm

idivx macro quot,num,denom
; remainder returns in dx
  mov ax,num
  cwd
  mov cx,denom
  idiv cx
  mov quot,ax
idivx endm

getstr macro buffer
  mov ah, 0Ah
  mov dx, offset buffer
  int 21h
getstr endm

keybch macro
   mov ah, 01h
   int 16h
keybch endm

setcurs macro row, col
   mov    ah,02
   mov    bh,00
   mov    dl,col
   mov    DH,row
   int    10H
setcurs endm

exitdos macro
   mov ah,4ch
   int 21h
exitdos endm
```

------end of file---------

- In **exp5** folder, create and edit **exp5p1.asm** to contain the following source text

```
; Source exp5p1
; student nr1:
; student nr2:

  include exp5.inc
  .model small
  .stack 100h
  .data
rowno   equ 08
colno   equ 05
Message1 db 'What is your last name? ','$'
Buffer1  db 24,?,24 DUP (0)
Message2 db CR, LF,'Letter-count of your last name is: '
Message3 db '     $'

  .code
  mov ax,@data
  mov ds,ax
  dispclr

  setcurs rowno,colno
  dispstr Message1
  getstr buffer1
  ; Mem[buffer1+1] contains the stringlength
  mov  al,Buffer1+1
  al2asc Message3
  dispstr Message2
waitkey:
  keybch
  jz waitkey
  exitdos
  end
```

- You will use EMU8086 in tracing the assembly code. Open exp5p1.asm in EMU8086.

- Click on Emulate to start the emulator.
- In the emulator window, click on menu-bar item view -> listing . You will get the list file opened.

  Reporting: Start a text file with the name **exp5.txt**. Write the Report Title in the following format

```
CMPE328 Experiment 5,  Report file by <name surname studentnr>
Part 1
```

- Copy the listing lines corresponding to the code segment (starting from .code) into your report file **exp5.txt**, as shown below:

```
CMPE328 Experiment 5 Report file
 by <Name-Surname> <number>  and  <Name-Surname> <number>
 Part-1
 [  16]        :                              .code
 [  17]     0160: B8 10 00                    mov ax,@data
 [  18]     0163: 8E D8                       mov ds,ax
 [  19]     0165: B8 00 06 B7 F0 B9 00 00 BA 4F 18 CD   dispclr
                  10
 [  20]        :
          . . .
          . . .
```

- Inspect carefully the first and the second occurance of invoking **dispstr** macro. Are there any difference? Why are they different?

  Reporting: Write your answer to report file

```
Dispstr macros are different because . . . . . .
```

- Close the listing, and trace the execution using single-step. When the emulator warns you to enter the string, write your **surname** on the DOS window.
- Open “vars” window (click on vars button), and click on “buffer1”. Then fill in to “elements” box 20.

  Reporting: Write the array of bytes in the buffer1 to your report file **exp5.txt** including the first zero byte.

```
BUFFER1: 18 05 62 . . .  75 72 0D 00
```

- Can you understand the length of the string from the second byte in buffer1? Is it consistent with the remaining bytes?
- Close the emulator window. On the edit window, click on “compile”. A “file-save browser” will get opened to save the exe file. Save the **exp5p1.exe** file into your **exp5** folder. Then, execute the **exp5p1.exe** to observe how it works.
- **Reporting:** Save the report file, and start to the second part of the experiment.

5.3.2. Average by Signed Arithmetic Operations .

Objectives:

-to demonstrate signed arithmetic operations on a code finding the average of signed numbers.

Procedure:

-The following assembly code finds the average of an array of bytes. Write it into **exp5p2.asm** in the **exp5** folder. Don’t forget to fill your name and number into the file.

```
; exp5p2.asm
; Student name and number 1:
; Student name and number 2:
   include exp5.inc
   .model small
   .stack 100h
   .data
snum  dw  4
sdata db -3, -12, 5, 2
aver  dw  ?
remn  dw  ?
MessageA db "Average is $"
MessageR db "Remainder is $"
NextLine db 13,10,"$"
dstr  db 10 dup(20h),'$'
   .code
   mov ax,@data
   mov ds,ax
```

```
        mov cx, snum
        mov bx, offset sdata
        mov dx,0
    addloop:
         mov ax,[bx]
         cbw
         add dx,ax
         inc bx
         loop addloop
        mov ax,dx
        cwd
        mov cx,snum
        idiv  cx
        mov aver,ax
        mov remn,dx
        mov ax,aver
        cmp ax,0
        jge positive
        mov dstr,'-'
        neg ax
    positive:
        al2asc dstr+1
        dispstr NextLine
        dispstr MessageA
        dispstr dstr
        mov ax,remn
        al2asc dstr
        dispstr NextLine
        dispstr MessageR
        dispstr dstr
    waitch:
        keybch
        jz waitch
        exitdos
        end
```

- You will use Emu8086 to trace this assembly code. Open **exp5p2.asm** in the Emu8086.
- Click the emulate button to start emulation. Observe carefully how the addition and division operations are performed, how the result is converted to ascii, and how it is written to display.
-Compile the executable file of the exp5p2.asm file. Execute and observe its operation.
  Reporting: In PART2 of your report file fill in the screen output to your report after the program stops.

5.3.3. Look-Up Table for the Square Root of an Integer.

Objectives:

-to demonstrate the input value search, and the output access for a Look Up table.

Procedure:

-The following assembly code finds the average of an array of bytes. Write it into **exp5p3.asm** in the **exp5** folder. Don’t forget to fill your name and number into the file.

```
; exp5p3.asm
; Student name and number 1:
; Student name and number 2:
    include exp5.inc
   .model small
   .data
Msg1    db 'I''ll find the square root using '
        db 'a look-up table.',13,10
        db ' Give me a number in the range [0, 255]: $'
Msg2    db 13,10,' Square-root is $'
lutcnt  dw 15
lutin   db  0, 1, 4, 9, 16, 25, 36, 49, 64
        db  81, 100, 121, 144,169,196, 225
lutout  db  0, 1, 2, 3,  4,  5,  6,  7,  8,
        db  9, 10,  11,  12, 13, 14, 15
buf     db 10h,?,10h dup(' ');
output  db 5 dup(' '), '$'
     .code
     mov ax, @data
     mov ds,ax
     dispstr Msg1
     getstr buf
     asc2al buf
     ; find index
     mov cx,lutcnt
lutlp:
     mov  bx,cx
```

```
          cmp  al,[bx + offset lutin]
          jae   lutexit
          loop lutlp
     lutexit:
          ; read output
          mov al,[bx + offset lutout]
          al2asc output
          dispstr Msg2
          dispstr output
     waitch:
          keybch
          jz waitch
          exitdos
          end
```

- You will use Emu8086 to trace this assembly code. Open **exp5p3.asm** in the Emu8086.
- Click the emulate button to start emulation.
- During the single-step emulation
    - Enter string "200" when the emulator asks an input value.
    - Observe carefully how the ascii input string is converted to 8-bit value by asc2al macro.
    - Observe carefully how the input array is searched from the last down to the first until an entry is found smaller than the input value.
    - Observe carefully how the output value is accessed once the index corresponding to the input value is obtained.
- Generate the executable file (use compile), and run it to see the operation of the program. Use input values 1, 5, 42, 64, 4Dh and 182 to see how it works. **Reporting:** In **PART3** of your report write what happens for each input.
- Hide the lines containing **keybch** and **jz waitch**. behind semicolons. Then generate its executable and observe the difference in operation.

5.3.4. Simple Look-Up Table for Fibonacci Numbers.

Objectives:

-to demonstrate the input value search, and the output access for a Look Up table.

Fibonacci Numbers:

According to Wikipedia pages, the Fibonacci numbers first appeared, under the name mātrāmeru (mountain of cadence), in the work of the Sanskrit grammarian Pingala (Chandah-shāstra, the Art of Prosody, 450 or 200 BC). Prosody was important in ancient Indian ritual because of an emphasis on the purity of utterance.

In the West, the sequence was first studied by Leonardo of Pisa, known as Fibonacci, in his Liber Abaci (1202). He considers the growth of an idealised (biologically unrealistic) rabbit population, assuming that:

in the first month there is just one newly-born pair,
new-born pairs become fertile from after their second month
each month every fertile pair begets a new pair, and
the rabbits never die

Let the population at month n be F(n). At this time, only rabbits who were alive at month n−2 are fertile and produce offspring, so F(n−2) pairs are added to the current population of F(n−1). Thus the total is **F(n) = F(n−1) + F(n−2)**.

Procedure:

-The following assembly code finds the i-th Fibonacci number. Write it into **exp5p4.asm** in the **exp5** folder. Fill your name and number into the file.

```
; exp5p4.asm
; Student name and number 1:
; Student name and number 2:
    include exp5.inc
    .model small
    .data
luacnt      dw 12
lua  db 1,1,2,3,5,8,13,21,34,55,89,144,233
fibnr  db '    $'
buf  db 20,?, 20 dup(' ')
msga db 'I have a look-up table to get'
```

```
       db ' the n-th Fibbonachi number.$'
msgb db cr,lf,'Give me a number in the range [0,12] : $'
msgc db cr,lf,'Your Fibonachi number is : $'
    .code
    mov ax,@data
    mov ds,ax
    dispstr msga
again:
    dispstr msgb
    getstr buf
    mov al,byte ptr buf+1
    cmp al,0
    jz  emptystr
     asc2al buf
    xor ah,ah  ; zero extend to ax
    mov bx,ax
    mov al, [bx + offset lua]
    al2asc fibnr
    dispstr msgc
    dispstr fibnr
     jmp again
emptystr:
    exitdos
    end
```

- Use Emu8086 to trace this assembly code. Open **exp5p4.asm** in the Emu8086 and start single-step emulation.
- Generate the executable file (use compile), and run it to see the operation of the program. Use input values “3”, “6”, “Ah”, “12” to see how it works.

Reporting:

After you complete the procedures, please save and close **exp5.txt** file, and e-mail it using your student e-mail account to **cmpe323lab@gmail.com** with the subject line **“exp5”** within the same day before the midnight.

Late and early deliveries will have 20% discount in grading. No excuse acceptable.

Free time practice-1:

In your free time, write assembly code of a program to return 255 sin(180 i /32) from a simple look-up table of 32 elements. (i.e., using a look-up table like this one)

```
lutcnt db 32
lutout db 0, 25, 50, 74, … , 0
```

Your program shall

write an explanation that it will return 255 sin(180 i/32), and that the user shall enter the number i.

If the entered number i is out of limits, program shall write wrong number.

Else, it will read the table, and print the result to the display with a reasonable message.

After printing the result it shall give a message and wait the next i in a loop until an empty string is entered in.

6.
I/O and External Memory Interface for 8051

6.1 Objective

The aim of this experiment is
i- An introduction to microcontroller architecture and instruction set of 8051.
ii- An introduction to the hardware-software simulation of 8051 in Prosys.
iii- An introduction of LED indicator output and switch input circuits.

6.2 Introduction

A microprocessor on a single integrated circuit intended to operate as an embedded system. As well as a CPU, a microcontroller typically includes small amounts of RAM and PROM and timers and I/O ports.

Intel introduced the first 8-bit microcontroller family MCS-48 in 1976. After four years development, Intel upgraded the MCS-48 family to 8051, an 8-bit microcontroller with on-board EPROM memory in 1980. Intel's 8051 is used in almost all embedded control areas including the car engine control.

6.2.1. Typical features

A typical 8051 family member, 80C51 has the following features:

- 4K Bytes of In-System Reprogrammable Flash Memory;
- Fully Static Operation: 0 Hz to 16MHz;
- 128 × 8-bit Internal RAM ;
- 32 Programmable I/O Lines;
- Two 16-bit Timer/Counters;
- Six Interrupt Sources;
- Programmable Serial Channel

The 8051 microcontroller is available in 40 pin DIP package with the pin layout given in Fig.1. This section will provide short information on the register-memory architecture, and the instruction set of 8051 microcontroller.

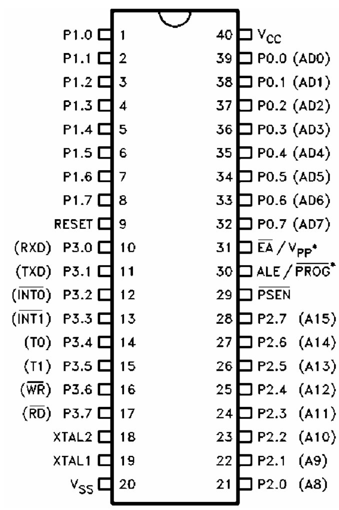


Fig. 1. Pin Layout of 40-pin DIP 8051 package

6.2.2. Registers

The 8051 microcontroller has two accumulator registers A and B, and eight general-purpose-data registers numbered from R0 to R7. The following is a list of predefined assembler labels corresponding to special function registers associated with direct memory access. Although they can be used with any immediate data evaluation. Associated label values are given in hexadecimal notation.

Table 1.1 Special Function Register definitions of 8051 microcontroller

SFR definitions (Alphabetic Order)

Label	Value	Description
A	E0	Accumulator
ACC	E0	Accumulator
B	F0	B register
DPL	82	Data Pointer Low byte
DPH	83	Data Pointer High byte
IE	A8	
IP	B8	
P0	80	Port 0
P1	90	Port 1
P2	A0	Port 2
P3	B0	Port 3
PCON	87	
PSW	D0	Program Status Word
RCAP2L	CA	
RCAP2H	CB	
SCON	98	
SBUF	99	
SP	81	Stack Pointer
T2CON	C8	
TCON	88	
TH0	8C	Timer/Counter 0 High byte
TL0	8A	Timer/Counter 0 Low byte
TH1	8D	Timer/Counter 1 High byte
TL1	8B	Timer/Counter 1 Low byte
TH2	CD	Timer/Counter 2 High byte
TL2	CC	Timer/Counter 2 Low byte
TMOD	89	

SFR definitions (Direct Mem. Addr. Order)

Label	Value	Description
P0	80	Port 0
SP	81	Stack Pointer
DPL	82	Data Pointer Low byte
DPH	83	Data Pointer High byte
PCON	87	
TCON	88	
TMOD	89	
TL0	8A	Timer/Counter 0 Low byte
TL1	8B	Timer/Counter 1 Low byte
TH0	8C	Timer/Counter 0 High byte
TH1	8D	Timer/Counter 1 High byte
P1	90	Port 1
SCON	98	
SBUF	99	
P2	A0	Port 2
IE	A8	
P3	B0	Port 3
IP	B8	
T2CON	C8	
RCAP2L	CA	
RCAP2H	CB	
TL2	CC	Timer/Counter 2 Low byte
TH2	CD	Timer/Counter 2 High byte
PSW	D0	Program Status Word
A	E0	Accumulator
ACC	E0	Accumulator
B	F0	B register

The predefined labels for bit addressable memory locations are limited by 8051 architecture. In Table 1.2, `.x` represents a value in the range of 0 to 7. For example `P0.x` is short hand to represent `P0.0, P0.1, P0.2, P0.3, P0.4, P0.5, P0.6` and `P0.7`. With `P0.0` = `80h`, `P0.1` equal to `81h`, etc. Associated label values are given in hexadecimal notation.

Table 1.2 Predefined Bit Labels

Label	Value	Description
ACC.x	E0 - E7	Accumulator (bits 0 through 7)
B.x	F0 - F7	B register (bits 0 through 7)
P0.x	80 - 87	Port 0 (bits 0 through 7)
P1.x	90 - 97	Port 1 (bits 0 through 7)
P2.x	A0 - A7	Port 2 (bits 0 through 7)
P3.x	B0 - B7	Port 3 (bits 0 through 7)
PSW.x	D0 - D7	Program Status Word (bits 0 through 7)
SCON.x	98 - 9F	Serial Control register (bits 0 through 7)
IE.x	A8 - AF	
IP.x	B8 - BF	
TCON.x	88 - 8F	Timer Control register (bits 0 through 7)
T2CON.x	C8 - CF	Timer 2 Control register (bits 0 through 7)
IT0	88	
IE0	89	
IT1	8A	
IE1	8B	
TR0	8C	
TF0	8D	
TR1	8E	
TF1	8F	
RI	98	Receive Interrupt flag
TI	99	Transmit Interrupt flag
RB8	9A	
TB8	9B	

REN	9C	
SM2	9D	
SM1	9E	
SM0	9F	
EX0	A8	
ET0	A9	
EX1	AA	
ET1	AB	
ES	AC	
ET2	AD	
EA	AF	
PX0	B8	
PT0	B9	
PX1	BA	
PT1	BB	
PS	BC	
PT2	BD	
P	**D0**	**Parity flag**
OV	**D2**	**Overflow flag**
RS0	D3	Register Select (bit 0)
RS1	D4	Register Select (bit 1)
F0	D5	
AC	D6	Auxiliary Carry flag
CY	**D7**	**Carry flag**

6.2.3. Instruction Set

The 8051 instruction set contains data-transfer, ALU, bit-manipulation, and program branching instructions. The complete instruction set is given in the following table.

Instruction Set of 8051.

Key: direct : direct memory address **Ri :** registers i=0,..,7

	Arithmetic Operations		
Mnemonic	**Description**	**Size**	**Cyc**
ADD A,Rn	Add register to Accumulator (ACC).	1	1
ADD A,direct	Add direct byte to ACC.	2	1
ADD A,@Ri	Add indirect RAM to ACC.	1	1
ADD A,#data	Add immediate data to ACC.	2	1
ADDC A,Rn	Add register to ACC with carry.	1	1
ADDC A,direct	Add direct byte to ACC with carry.	2	1
ADDC A,@Ri	Add indirect RAM to ACC with carry.	1	1
ADDC A,#data	Add immediate data to ACC with carry.	2	1
SUBB A,Rn	Subtract register from ACC with borrow.	1	1
SUBB A,direct	Subtract direct byte from ACC with borrow	2	1
SUBB A,@Ri	Subtract indirect RAM from ACC with borrow.	1	1
SUBB A,#data	Subtract imm. data from ACC with borrow.	2	1
INC A	Increment ACC.	1	1
INC Rn	Increment register.	1	1
INC direct	Increment direct byte.	2	1
INC @Ri	Increment indirect RAM.	1	1
DEC A	Decrement ACC.	1	1
DEC Rn	Decrement register.	1	1
DEC direct	Decrement direct byte.	2	1
DEC @Ri	Decrement indirect RAM.	1	1
INC DPTR	Increment data pointer.	1	2
MUL AB	result is 16-bit B:A ← A x B ;	1	4
DIV AB	A ← A / B (int.result); , B <- A%B (remainder)	1	4
DA A	Decimal adjust ACC.	1	1
	Logical Operations		
Mnemonic	**Description**	**Size**	**Cyc**
ANL A,Rn	AND Register to ACC.	1	1
ANL A,direct	AND direct byte to ACC.	2	1
ANL A,@Ri	AND indirect RAM to ACC.	1	1
ANL A,#data	AND immediate data to ACC.	2	1
ANL direct,A	AND ACC to direct byte.	2	1
ANL direct,#data	AND immediate data to direct byte.	3	2
ORL A,Rn	OR Register to ACC.	1	1
ORL A,direct	OR direct byte to ACC.	2	1
ORL A,@Ri	OR indirect RAM to ACC.	1	1
ORL A,#data	OR immediate data to ACC.	2	1
ORL direct,A	OR ACC to direct byte.	2	1
ORL direct,#data	OR immediate data to direct byte.	3	2
XRL A,Rn	Exclusive OR Register to ACC.	1	1
XRL A,direct	Exclusive OR direct byte to ACC.	2	1
XRL A,@Ri	Exclusive OR indirect RAM to ACC.	1	1
XRL A,#data	Exclusive OR immediate data to ACC.	2	1
XRL direct,A	Exclusive OR ACC to direct byte.	2	1
XRL direct,#data	XOR immediate data to direct byte.	3	2
CLR A	Clear ACC (set all bits to zero).	1	1
CPL A	Compliment ACC.	1	1
RL A	Rotate ACC left.	1	1
RLC A	Rotate ACC left through carry.	1	1
RR A	Rotate ACC right.	1	1
RRC A	Rotate ACC right through carry.	1	1
SWAP A	Swap nibbles within ACC.	1	1
	Other Instructions		
Mnemonic	**Description**	**Size**	**Cyc**
XCH A,Rn	Exchange register with ACC.	1	1
XCH A,direct	Exchange direct byte with ACC.	2	1
XCH A,@Ri	Exchange indirect RAM with ACC.	1	1
XCHD A,@Ri	Exchange low nibble of indirect RAM with lower nibble of ACC.	1	1
NOP	No operation.	1	1

	Data Transfer		
Mnemonic	**Description**	**Size**	**Cyc**
MOV A,Rn	Move register to ACC.	1	1
MOV A,direct	Move direct byte to ACC.	2	1
MOV A,@Ri	Move indirect RAM to ACC.	1	1
MOV A,#data	Move immediate data to ACC.	2	1
MOV Rn,A	Move ACC to register.	1	1
MOV Rn,direct	Move direct byte to register.	2	2
MOV Rn,#data	Move immediate data to register.	2	1
MOV direct,A	Move ACC to direct byte.	2	1
MOV direct,Rn	Move register to direct byte.	2	2
MOV direct,direct	Move direct byte to direct byte.	3	2
MOV direct,@Ri	Move indirect RAM to direct byte.	2	2
MOV direct,#data	Move immediate data to direct byte.	3	2
MOV @Ri,A	Move ACC to indirect RAM.	1	1
MOV @Ri,direct	Move direct byte to indirect RAM.	2	2
MOV @Ri,#data	Move immediate data to indirect RAM.	2	1
MOV DPTR,#data16	Move immediate 16 bit data to data pointer register.	3	2
MOVC A,@A+DPTR	Move code byte rel. to DPTR to ACC (16 bit address).	1	2
MOVC A,@A+PC	Move code byte rel. to PC to ACC (16 bit address).	1	2
MOVX A,@Ri	Move external RAM to ACC (8 bit address).	1	2
MOVX A,@DPTR	Move external RAM to ACC (16 bit address).	1	2
MOVX @Ri,A	Move ACC to external RAM (8 bit address).	1	2
MOVX @DPTR,A	Move ACC to external RAM (16 bit address).	1	2
PUSH direct	Push direct byte onto stack.	2	2
POP direct	Pop direct byte from stack.	2	2
	Boolean Variable Manipulation		
Mnemonic	**Description**	**Size**	**Cyc**
CLR C	Clear carry flag.	1	1
CLR bit	Clear direct bit.	2	1
SETB C	Set carry flag.	1	1
SETB bit	Set direct bit.	2	1
CPL C	Compliment carry flag.	1	1
CPL bit	Compliment direct bit.	2	1
ANL C,bit	AND direct bit to carry flag.	2	2
ANL C,/bit	AND compliment of direct bit to carry.	2	2
ORL C,bit	OR direct bit to carry flag.	2	2
ORL C,/bit	OR compliment of direct bit to carry.	2	2
MOV C,bit	Move direct bit to carry flag.	2	1
MOV bit,C	Move carry to direct bit.	2	2
	Program Branching		
Mnemonic	**Description**	**Size**	**Cyc**
ACALL addr11	Absolute subroutine call.	2	2
LCALL addr16	Long subroutine call.	3	2
RET	Return from subroutine.	1	2
RETI	Return from interrupt.	1	2
AJMP addr11	Absolute jump.	2	2
LJMP addr16	Long jump.	3	2
SJMP rel	Short jump (relative address).	2	2
JMP @A+DPTR	Jump indirect relative to the DPTR.	1	2
JC rel	Jump if carry is set.	2	2
JNC rel	Jump if carry is not set.	2	2
JB bit,rel	Jump if direct bit is set.	3	2
JNB bit,rel	Jump if direct bit is not set.	3	2
JBC bit,rel	Jump if direct bit is set & clear bit.	3	2
JZ rel	Jump relative if ACC is zero.	2	2
JNZ rel	Jump relative if ACC is not zero.	2	2
CJNE A,direct,rel	Comp. direct byte to ACC and jump if not equal.	3	2
CJNE A,#data,rel	Comp. imm. byte to ACC and jump if not equal.	3	2
CJNE Rn,#data,rel	Comp. imm. byte to reg. and jump if not equal.	3	2
CJNE @Ri,#data,rel	Comp. imm. byte to ind. and jump if not equal.	3	2
DJNZ Rn,rel	Decrement register and jump if not zero.	2	2
DJNZ direct,rel	Decrement direct byte and jump if not zero.	3	2

8051 can access the program code ROM or Flash memory by **`MOVC`** instructions. External RAM by **`MOVX`** instructions, and the internal RAM memory (locations 0 … 128 for MCS51, 0…256 for MCS52) by **`MOV`** instructions.

6.2.4. The 8051 Ports

The 8051 microcontroller provides three ports for the users, denoted by symbols **`P0`**, **`P1`**, **`P2`** and **`P3`**. 8051 i/o ports are memory mapped registers with input/output connection to the external circuits. The addresses of these ports are available in Table 1.1.

The ports are bit addressable as seen in Table 1.2. Ports **`P1`**, **`P2`** and **`P3`** have weak internal pull-up resistors, while the pins of **`P0`** has no internal pull-ups, because it is also used as **`AD0-AD7`** lines for external memory access. Therefore external pull ups are necessary to interface a switch to a **`P0`** pin, similar to resistors R00 and R01 in Fig. 2.

An i/o pin of the ports is suitable for input only when it is set to high. For example: **`CLR P1.3`** makes **`P1.3`** pin 0V, and it is not suitable for input, since **`P1.3`** will sink external current strongly to the ground. **`SET P1.3`** makes **`P1.3`** pin 5V with a weak current source. The external circuit can easily drive **`P1.3`** below the logic-threshold voltage, and make it read 0. A reset (RST high) starts the ports with **`P0`**=**`P1`**=**`P2`**=**`P3`**=**`0x0FF`**, suitable for input.

An output pin can drive a LED indicator in the common-cathode mode. In Fig.2, the component pair {R30, DB1} connected to **`P3.0`** pin is a typical LED indicator. DB1 gets lighted when the output pin **`P3.0`** delivers low (=0V, or logic-0), and DB1 stays dark while **`P3.0`** stays at high (=5V, or logic "1").

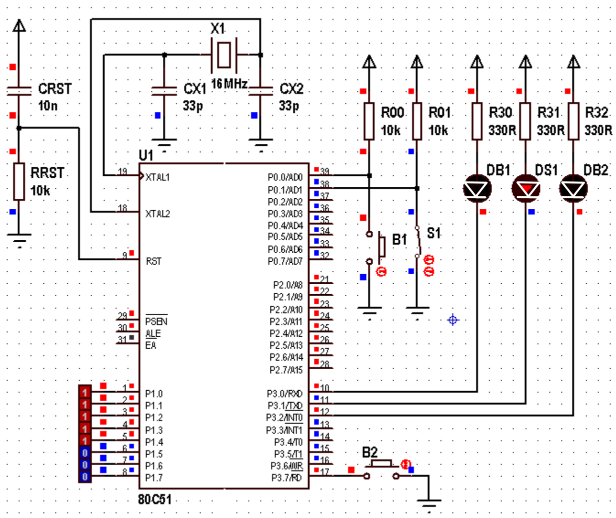


Fig. 2 . Switch and LED interfacing configurations.

In Fig.2, S1-RD1 forms a pull-up biased switch circuit. It gives high to the input **`P0.1`** while switch is open (open-circuit = off), and makes **`P0.1`** low while switch is closed (closed circuit = on). In summary, **`P0.1`** reads 0 if switch is turned on, and it reads 1 otherwise.

6.2.5. Command line Assembler for 8051

Keil products supplies professional integrated development tools for 8051 family devices. The currently available Keil student version can code up to 2-kBytes of hexadecimal coding for any 8051 device. Keil-C (C51) and assembler (A51) are usually called by its development environment UV3. However, we will use them calling in DOS-Command environment through a batch file. Keil C is an almost-ANSI C compatible C-compiler for writing programs in tiny-os operating system. Compiler C51 and assembler A51 produce an object file, which needs linking into an absolute code using BL51. Absolute code is further converted to INTEL HEX format by the code converter Oh51. The following listing is the **compile.bat** batch file .

```
echo off
PATH=.\8051\C51\BIN
SET TMP=.\8051\TMP
del exp6.hex
```

```
a51 exp6.a51 debug object(p.obj)
bl51 p.obj
oh51 p hexfile(exp6.hex)

pause
del p.*
del exp6.lst
```

The environment settings of the batch file is valid only if the folder 8052 is under the work folder of the experiment. It works on desktop folder, or on the root folder of a flash disk.

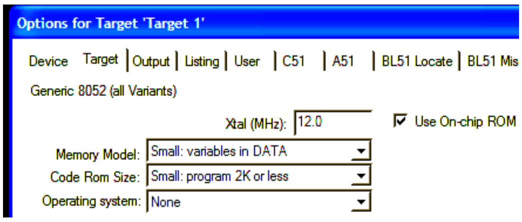
6.2.6. IDE Tool for Coding of 8051

Keil products supplies professional integrated development tools for 8051 family devices. The currently available Keil IDE mvision-3 (uv3), and a limited capacity trial version can code up to 2-kBytes of hexadecimal coding for any 8051 device. UV3 is Keil-C (C51) and assembler (A51) compatible. Keil C is an almost-ANSI C compatible C compiler environment for writing programs in Tiny-OS operating system. Keil IDE produces the hex file to transfer the program code into the target 8051 device. The free trial version of Keil-IDE does not require any registration into Windows operating system. Its initialization parameters are stored in tools.ini file, and can be edited by a text editor. The software pack can be easily installed by copying the **KC51** folder at the root of any drive, and correcting the drive name in the **tools.ini** file.

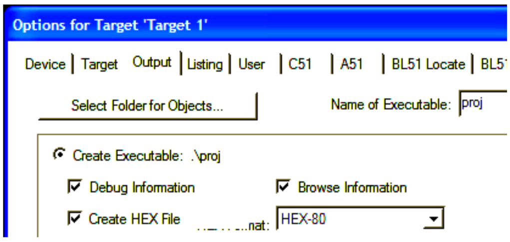
UV3 environment does not need installation other than modification of the C51 path in tools.ini file. A copy of KC51 is available on the C-drive, and you may use it also on your flash-disk drive (about 50Mbytes).

Installation and starting a C or Assembly project with Keil-C51 are quite simple. If KC51 is not yet installed on your computer follow the steps to install it on your hard disk (C:) or your floppy disk (E:).

- **Installing KC51:** Download the rarred KC51 IDE folder from the coarse web side, open the rar-archive, and copy the folder **KC51** to the root of your drive (C:) or (E:), so that **E:\KC51** folder contains folders **C51**, **UV3** and the file **TOOLS.INI**. Then edit path statement of tools.ini to **E:\KC51\C51**. Your KC51 is ready for execution.
- **Making a Work Folder:** Start a working folder similar to **E:\323\012345\ExpXX** . Copy all necessary C (-.C , -.H . and -.C51 files) and Assembler (-.ASM and -.A51 files) source files together with Proteus Circuit Simulation files (-.DSN) into your work folder.
- **Opening an existing Project:** If a KC51project definition file (-.UV2) is available in the work folder use (*Project* → *Open Project*) to start the project with its settings.
- **Starting a New Project**: Start **KC51\UV3\UV3.exe** file. Close the initially opened project file using menu (*Project* → *Close Project*) . Start a new project by (*Project* → *New uVision Project*) browsing your work folder, and entering project name, let's say "**proj**". From the popped CPU-dialog-box, select "*Generic – 8052 (all variants)*". Click "*No*" if it asks to "*copy 8051 startup code to project folder …*" . Click on Target 1 to select it, and with right-click open the "*Options for Target-1*" dialog window. Check that Device is Generic 8052 and Linker is BL51. Set Target Xtal(MHz) as required for the application, Memory Model Small, Code Rom Size Small, Operating System None, and put check for Use On-chip ROM (0x0-0x1FFF).



Set *Output* to create both *executable* and *hex* file with debug information. You may change the name of the executable and Hex-file by entering it into *Name of Executable* box



- Generating a list file: List file contains debug messages and symbol tables. You can generate **-.lst** file by putting a check into the *Assembler Listing* box in *Listing* window of Options dialog.

After setting all of the above options click **OK** to close the *Options* dialog.

- **Adding Assembly files to the project:** Open the dependents list of Target 1 by clicking on plus sign next to it. Right-click on *"Source Group 1"* to get the quick menu for "*adding source files to Group 1*". Click it to start the file browser to add your source file. First set the folder to your work folder that contains your **-.asm** file. Then set "*Files of type*" field to "*asm source file*". Your **-.a51** file will appear in the browser window. Select the file and click on "*add*".
- **Adding C files to the project:** Apply the same procedure, but set "*Files of type*" field to "*C source file*". Your -.C file will appear in the browser window. Select the file and click on "*add*".
- **Building the project:** On the toolbar use the icons (build and rebuild) to build the project and generate the executable and **-.hex** file.

6.2.7. Simulation in ISIS

Simulation is the best methodology to verify operation of the circuit and the program code in a time-efficient manner. It is always a good idea to simulate the circuits and codes using convenient simulation software instead of rushing to build the circuit and code the chip for a real-life test.

ISIS is able to simulate many microcontrollers with their peripheral circuits. The circuit diagrams are composed of components, and connections between the component terminals. A component that needs a program code is linked to the program code file writing the code folder and file name (.hex file name) into its configuration window. ISIS can simulate this graphical circuit representation and update the appearance of the display elements in regular periods of about 50ms.

6.3 Experimental Part

6.3.1. Installation of A51 to your work folder

Objective: preparation of a work folder for A51 IDE.

Procedure-1:

1- Download the **exp6.rar** file which contains all necessary files and folders to a convenient place i.e. onto the desktop. Extract and open the work folder **Exp6**.

2- Open the source file Exp6.a51. The file shall contain the following lines

```
; Exp6.a51 test file
; (c) 2008, Dr. Mehmet Bodur
xtal equ 16    ; Crystal frequency in MHz

; power-on reset starts execution from address 0
  org 0

  mov P0,#00000011b ; make P0.1 and P0.0 suitable for input
  mov P3,#10000000b  ; prepare P3.7 for input

back:
; copy port0 switch B1,S1  states to acc
  mov a,P0
  anl a,#00000011b  ; P0.1 and P0.0 are selected
  orl a,#10000000b  ; prepare P3.7 for input

; copy bit  P3.7 to bit P2.2
  mov C, p3.7   ; copy P3.7 to Carry Flag
  mov acc.2, C  ; copy Carry to acc.2
  mov P3,a      ; apply result to P3

; increment P1
  inc P1

; delay for 25ms delay
  mov A,#250
  acall dly100u
 sjmp back

dly100u:
; delay loop takes A*100u
  mov r1,A
dlylp1: mov  r0,#(xtal*62/10)
dlylp2: djnz r0,dlylp2
  djnz r1,dlylp1
  ret
 end
```

3- Double-click on compile.bat to start assembling of the source file **exp6.a51** . Batch file will stop on pause waiting a key press. Before you press any key check your work folder and find the generated **exp6.lst** file.

Reporting:

Open **exp6.lst** file in a text editor, and copy the first page (up to symbol table) to your reporting file. After you close the text editor activate batch file window and press the space-bar to end the batch session.

4- In your work folder you will find the file "**exp6.hex**" which is generated by the batch operation as a product of assembly, link, and conversion processes.

Reporting:

- Open the **exp6.hex** file in a text editor, and copy the contents to your report file. The hex file contains the machine code to be coded into the micro-controllers program memory. This file will be used in the next section of the experiment.

- Save your reporting file for other report deliverables.

6.3.2. Simulation of a Microcontroller Circuit

ISIS release 6.9 of Labcenter Electronics can successfully simulate the digital-analog hybrid circuits including the PIC16, PIC18, 68HC11 and MCS51 family micro controllers.

Objective:

Our objective is getting familiar with the ISIS simulation environment.

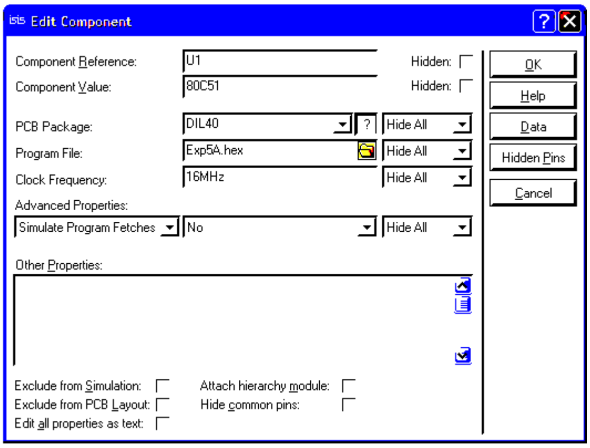


Fig. 4. Edit component window of 8051

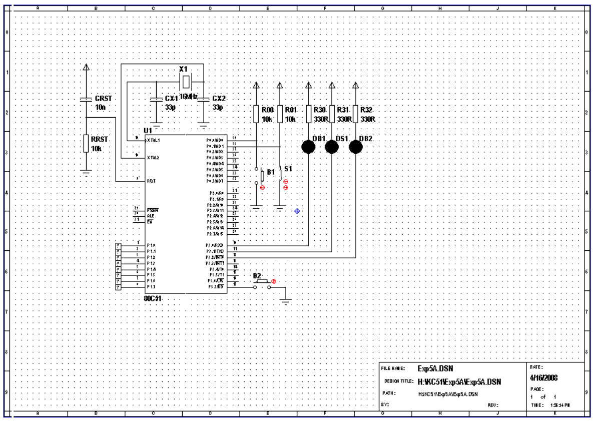


Fig. 3. Design window of Exp5A.DSN

Procedure

1- Start Proteus Professional→ISIS 6 Professional in windows.

2- Use File→Load design to open the file-browser, navigate to **Exp6A** folder, and load **Exp6A.DSN** file to ISIS. You will get the design window seen in Fig.3.

3- Right click once on the 8051 processor. The processor will turn to red, indicating that it is selected. Left click once on 8051 to open the "edit component" window of 8051 seen in Fig.4. The Program File shall contain the file name Exp5A.hex, which is generated in Section 3.1. You can link a file using the file browser icon, or directly by editing the file name. Do not forget to OK the new file name.

4- Close the edit-window, and right-click on the empty part of the design window to deselect components. All red components will take their original colors.

5- Two kind of switches are shown in Fig.5 . These switches are active circuit elements changing state by clicking on their control buttons.

6- Click on  button to start the component insertion mode. This mode supports interaction to the active components (switches, buttons, and logic-states) using the mouse.

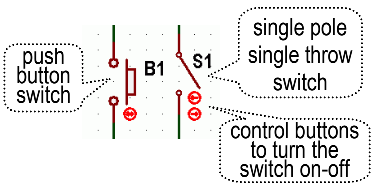
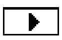


Fig.5. Circuit symbos of Pushbutton and SPST switches

7- Click on start button  to start simulation. Turn the toggle and button switches on and off, and observe the logic status at the port inputs P0.0, and P0.1.

Reporting:

Write your observations into the report file Exp6.txt as seen below filling the question marks with your observations.

```
3.2 Simulation section:
B1= Pressed,  P0.0 = "low/high ?"; P3.0 = "low/high ?"
B1= Released, P0.0 = "low/high ?"; P3.0 = "low/high ?"
S1= On,  P0.1 = "low/high ?"; P3.1 = "low/high ?"
S1= Off, P0.1 = "low/high ?"; P3.1 = "low/high ?"
B2= Pressed,  P3.7 = "low/high ?"; P3.2 = "low/high ?"
B2= Released, P3.7 = "low/high ?"; P3.2 = "low/high ?"
```

9- Click on stop button ■ to stop simulation. Right-click on DB1, and make its full drive current 20 mA (nominal current of the old low-efficiency LED). Then start the simulation, push on B1. LED DB1 will glow. Then push on B2 to glow DB2. Report any difference between the LED illumination levels in your reporting file.

Explanations:

The code **Exp6A.a51** executed in 8051 makes pins P0.0, P0.1 and P3.7 input pin.

```
mov P0,#00000011b ; make P0.1 and P0.0 suitable for input
mov P3,#10000000b  ; prepare P3.7 for input
```

All other bits initially start giving low output (near 0V). Then, a loop starts with the label "**back**",

```
back:
```

In the loop, P0 is copied to accumulator. An and-mask keeps bit-0 and bit-1, and clears all other bits. Then, an or-mask sets bit-7.

```
; copy port0 switch B1,S1  states to acc
  mov a,P0
  anl a,#00000011b  ; P0.1 and P0.0 are selected
  orl a,#10000000b  ; prepare P3.7 for input
```

Next, bit-7 (button B2 status) is copied to bit-2 of the acc register. Acc is copied to P3 to display the new status on LED indicators.

```
; copy bit  P3.7 to bit P2.2
  mov C, p3.7   ; copy P3.7 to Carry Flag
  mov acc.2, C  ; copy Carry to acc.2
  mov P3,a      ; apply result to P3
```

There after, port P1 is incremented by one,

```
; increment P1
  inc P1
```

Finally, a delay of approximately 25 ms is called to slow down the counting on P1,

```
; delay for 25ms delay
  mov A,#250
  acall dly100u
```

And the code in the back loop is repeated forever.

```
  sjmp back
```

The delay is obtained by looping idle a preset amount of cycles depending on crystal frequency.

```
dly100u:
; delay loop takes A*100u
```

```
    mov r1,A
  dlylp1: mov  r0,#(xtal*62/10)
  dlylp2: djnz r0,dlylp2
    djnz r1,dlylp1
    ret
   end
```

Reporting:

After you complete the procedures, please save and close **exp6.txt** file, and e-mail it using your student e-mail account to **cmpe323lab@gmail.com** with the subject line **"exp6"** within the same day before the midnight.

Late and early deliveries will have 20% discount in grading. No excuse acceptable.

Free time practice:

Write a 8051 assembler source (file name **Exp6P.a51**) for the circuit of Exp6A, that
- initially turn off all three LED, and make P0.0, P0.l, and P3.7 input pins. Clear R3 and R4.
- in the mainloop

  call dly100u with acc=100 (for 10ms delay)
  increment R3,
  if R3 exceeds 10, reset R3=0, and increment R4.
  turn off all LEDs
  if R4=1, turn on the LED connected to P3.2 .
  if R4=2, turn on the LED connected to P3.1 .
  if R4=3, turn on the LED connected to P3.0 .
  if R4=4, turn on all of LEDs, connected to P3.0, P3.1, and P3.2,
  if R4=5, make R3=0; R4=0.
  continue looping forever.

Assemble your source, and execute your code in ISIS. You shall edit compile.bat file with a text editor to change `exp6.a51` and `exp6.lst` to `exp6P.a51` and `exp6P.lst`.
After these changes **compile.bat** will generate **exp6.hex** file by assembling the source file **exp6P.a51**.

Start execution of the code in ISIS and observe the LEDs.

Does it light the LEDs in a sequence at every 1 second?

7. 8051 Memory Decoders and Memory Interface

7.1 Objective

The aim of this experiment is to observe the operation of a memory address decoder on a 8051 external memory circuit on the ISIS external memory interfacing simulation.

7.2 8051 Memory Interfacing

The 8051 microcontroller instruction set includes an external memory dedicated data transfer instruction: **MOVX**, and the processor supports up to 64 kbytes external memory addressing through the ports **P0**, **P2** and **P3**. Accessing external memory occupies **P0** to carry **AD[0..7]** address-data lines, **P2** to carry **A[8..15]** high address byte, and the pins **P3.6** and **P3.7** to carry **~RD** and **~WR** control signals. The address latch enable **~ALE** pin supplies a negative-edge to trigger the D-FF register while 8051 delivers the lower address byte **A[0..7]** through **AD[0..7]** lines, similar to the 8088 local bus. Total 16 address lines provide 64kbytes address space for external memory. This address space is usable for external code or data memory, and also for memory mapped i/o devices.

ISIS6.9 provides simulation of external memory addressing of the 8051 microcontroller, which serves in this experiment for observing the operation of a **74LS138** decoder, **6116** RAM devices, and **2764** EPROM devices. The simulation power of ISIS is restricted to only 8051bus devices with a limited memory options.

ISIS simulates a **2764** EPROM chip with its programmed contents by linking the contents filename (.hex format) to its properties. In this experiment, we will have two program projects: **Exp7Bus.Uv2** to generate the program code file **Exp7Bus.hex** that runs in 8051 processor, and **Exp7_2764.Uv2** to generate the data code file **2764.hex** for the **2764** EPROM chip.

7.3 Experimental Part

7.3.1. Installation of KC51 and preparation of -.HEX files

Objective: preparation of a workfolder for KC51 IDE and generation of -.hex files for the simulation. If KC51 is already installed on the computer skip steps 1 to 3 of Procedure-1.

Procedure-1:

1- Download the rarred KC51 IDE folder from the coarse web side, open the rar-archive, and copy the folder **KC51** to the desktop or to a flash-disk.

2- In the explorer, open M51 folder under the KC51 folder. Copy the folder address "…\KC51\C51" to the clipboard.

3- Open **"tools.ini"** in notepad. Paste the folder address into PATH= "…\" at the [C51] section of the ini file.

- If you plan to work on flashdisk (let's say drive **E:**) then copy **KC51** folder to the root folder so that **E:\KC51** folder contains folders **C51**, **UV3** and the file **TOOLS.INI**. Then edit path statement of tools.ini to **E:\KC51\C51**.

4- On the root folder create folder **x:\323\012345\exp7**, where **012345** stands for your student number. In the folder …**\exp7** start a txt file with the name "**Exp7.txt**" for reporting. Write your student name and number on the first line of the file similar to.
 CMPE 328 Exp7 Report file by <your-name, surname, student number>

5- Start UV3.exe (Keil-IDE) by clicking on the shortcut. Close the projects (menu→project→ close project) if any project is open.

6- Open the project file **"Exp7_2764.Uv2"** in the **"KC51/Exp6A"** folder. In the Project-Workspace window, click on the target, and the source-group-1 folders to turn on the project source file list. There must be "**2764.a51**" in your projects sources. If the file is not yet open, open it by clicking on this item.

7- The file shall contain the following lines

```
; 2764 EPROM contents source file.
; 2008 (c) Mehmet Bodur
  org 0
  db 0xE0,0xE1,0xE2,0xE3,'Hello World. '
  end
```

8- Build the project by clicking to Build-Target button (). You shall see the following messages in the "Build" message window if the installation is successful.

```
Build target 'Target 1'
assembling 2764.a51...
linking...
Program Size: data=8.0 xdata=0 code=17
creating hex file from "2764"...
"2764" - 0 Error(s), 0 Warning(s).
```

9- Open the project folder "**Exp7A**" in the explorer. From the date and time marks of the files, you will see the following files created recently.

Reporting:

- Open the **-.lst** file in a text editor, and copy the first page (up to the **"end"** in the source code) to your reporting file.
- Check whether the **-.hex** file in a text editor is generated. This file will be used for the contents of the external EPROM chip.
- Save your reporting file for other report deliverables.

10- Open the project file **"Exp7Bus.Uv2"** in Keil-IDE. You will find the following source file in the project with the filename **"extmemread.a51"** .

```
; Exp.7 8051 External Memory
; ( c ) 2008 Mehmet Bodur
;
  org 0
  mov p0,#0
start:
  mov dptr,#0001h
  mov a,#0x23
  movx @dptr,a
  mov p1,a

  mov dptr,#2001h
  mov a,#0x45
  movx @dptr,a
  mov p1,a
```

```
mov dptr,#0001h
movx a,@dptr
mov p1,a

mov  dptr,#2001h
movx a,@dptr
mov p1,a

sjmp start
end
```

This program code writes two bytes to external memory locations, first 0x23 to 0x0001, then 0x45 to 0x2000. Next, it reads these two data bytes from the same locations: 0x0001 and 0x2001. This program code displays on port-1 data bytes after a read or write operation.

10- Build the project by clicking to Build-Target button (). You shall see the following messages in the "Build" message window if the installation is successful.

```
Build target 'Target 1'
assembling extmemread.a51...
linking...
Program Size: data=8.0 xdata=0 code=33
creating hex file from "Exp7Bus"...
"Exp7Bus" - 0 Error(s), 0 Warning(s).
```

11- In the project folder "**Exp7A**" check the -.hex and -.lst files to be sure that they are generated. Copy the first page (upto the end line of assembly) into your reporting file.

7.3.2. Simulation of 8051 with External Memory

Labcenter Electronics Portable Proteus 7.6 ISIS will simulate the extended memory of an 8051 micro controller.

Objective:

Our objective is getting familiar to the ISIS simulation environment.

Procedure

1- Start Proteus 7 Portable→ISIS ... in windows.

2- Use File→Load design to open the file-browser, navigate to **Exp7A** folder, and load **Exp7Bus.DSN** file to ISIS. You will get the design window seen in Fig.3.

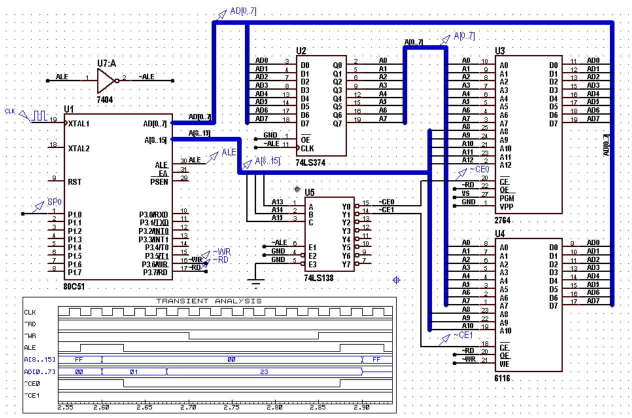


Fig. 2. Design window of Exp5A.DSN

3- Right click, and then left click once on the 8051 processor. The Program File in the "edit component" window of 8051 shall contain the file name **Exp7Bus.hex**, which is generated in Section 3.1. Check that its clock frequency is 40. This frequency is selected because the animation display rate of ISIS is frames per second, and it executes in 50ms steps at every click on the button. Close the edit-window.

4- Apply the same procedure described in (3) on 2764 EPROM chip to link **“`2764.hex`”** to this EPROM device. After this process close the edit-window.

6- Click on button to start the component insertion mode. Click on step button to start simulation. Start of simulation will enable the memory windows in the debug menu. Open the memory windows, and observe the initial contents of U3 (=2764) and U4 (=6116) memory chips.

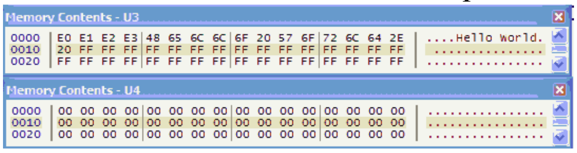


Fig. 3. The initial contents of the memory chips.

Reporting:

Write the first 8 bytes of each memory contents to your reporting file.

```
3.2 Simulation section:
initial contents of U3: E0 E1 E2 E3 48 65 6C 6C
initial contents of U4: 00 00 00 00 00 00 00 00
```

7- Click on execute button to execute the code for a couple of seconds. Then pause the simulation by clicking on button. Observe the contents of U3, 2764 and U4, 6116 memory chips.

Reporting:

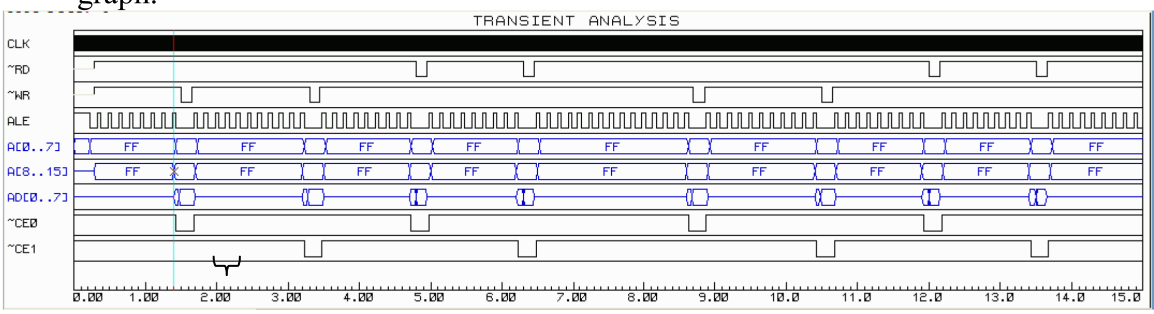
Write the first 8 bytes of each memory contents to your reporting file.

```
after 10s contents of U3: E0 … … …
after 10s contents of U4: 00 … … …
```

Explanations:

You shall expect that EPROM is non-volatile, and it is a read-only memory. Therefore the written bytes shall not change the contents of the EPROM memory. On contrary, 6116 RAM will change the contents of the locations whenever a data is written on its locations.

8- Click on the graph title “Transient Analysis”. A graph window will get opened. The control buttons are for **EditWindow, AddTrace, Execute, NavigateLeft, NavigateRight, ZoomIn, ZoomOut, ZoomAll, ZoomManuel, and ViewLogFile**. Clicking on **Execute**, and then **ZoomAll** will display the following graph.



zoom in range

Fig.4. ZoomAll view of the memory write and read cycles.

Explanations:

CLK frequency is too high to display the clock pulses individually. **`~RD, ~WR, ALE, A[8..15]`** are microcontroller outputs. **`AD[0..7]`** is multiplexed address-data lines. A 74374 positive edge-triggered D-Latch stores address value **`A[0..7]`** given from **`AD[0..7]`** lines at the positive . **`~CE0`** and **`~CE1`** are address decoder outputs.

9- Use ManualZoom to zoom in to the first write cycle of the graph, as seen below.

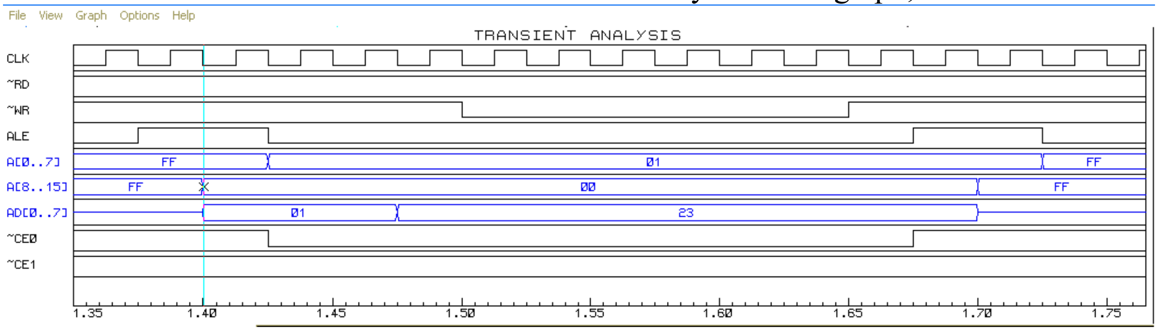


Fig.4. Zoom in [1.35 , 1.75] seconds view showing the write 0x23 to the external memory location 0x0001.

Reporting:

Attach the blue start line to the start of the **`AD[0..7]`** valid period by left-clicking at that point while you press the control-key. Now, measure the duration of the **`AD[0..7]=0x01`** and **`=0x23`**. Write the durations both in total number of clock cycles and time in seconds.

`Duration of AD[0..7]=0x01 is … cc , = … … seconds`
`Duration of AD[0..7]=0x23 is … cc , = … … seconds`

Explanations:

One external memory write bus cycle starts from valid address on **`AD[0..7]`** , and ends when **`AD[0..7]`** becomes floating.

10- Use manual zoom to display the first read bus cycle on the graph. This is a read from 2764 EPROM device. An Intel 8051 external memory read bus cycle takes exactly 12 clock cycles.

Reporting:

Explain in your report how you conclude that the memory cycle is a read cycle from the EPROM (Use the status of **`~WR`**, **`~RD`**, and **`~CE#`** lines). Explain what is value of the data byte sent from the EPROM to the processor.

11- Use manual zoom to display the second read bus cycle. This is a read from 6116 RAM device. An Intel 8051 external memory read bus cycle takes exactly 12 clock cycles.

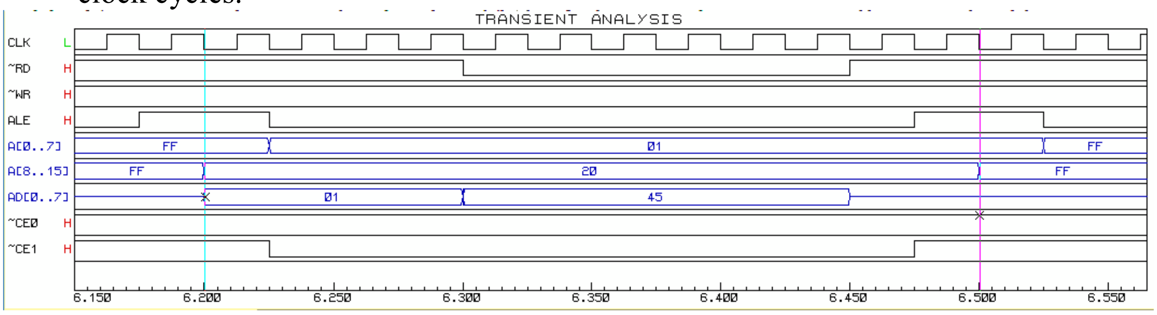


Fig.4. Zoom in [6.15 , 6.5] seconds view showing the read from the external memory location 0x2001.

Reporting:

Explain in your report how you conclude that the memory cycle is a read cycle from the RAM (Use the status of **`~WR`**, **`~RD`**, and **`~CE#`** lines). Explain what is value of the data byte sent from the RAM to the processor. Is the data byte value the same with the written data value?

Reporting:

After you complete the procedures, please save and close **exp7.txt** file, and e-mail it using your student e-mail account to **cmpe323lab@gmail.com** with the subject line **“exp7”** within the same day before the midnight.

Late and early deliveries will have 20% discount in grading. No excuse acceptable.

8.
8051 Memory Mapped I/O
and
8255A Interfacing

8.1 Objective

The aim of the first part of this experiment is to observe
a- an I/O address decoder for memory mapped i/o system of an 8051 processor.
b- a simple output port implemented with a 74 LS374 latch,
c- a simple input port implemented with a 74S244 three-state buffer.
d- interfacing button switches to an input port
e- interfacing a 7-segment LED display to an output port
The aim of the second part consists of
a- interfacing an 8255 to a 8051 processor,
b- interfacing a 6-digit multiplexed 7-segment display to an 8255.
The aim of the third part is to demonstrate how the rotation of a stepper motor is controlled with 80x86 code.

8.2 8051 External IO Interfacing

The **MOVX** instruction of 8051 microcontroller offers a method to interface memory mapped io devices **using** the ports **P0**, **P2** and **P3** for external memory addressing. **P0** carries **AD[0..7]** address-data lines, **P2** carries **A[8..15]** high address byte, and the pins **P3.6** and **P3.7** provide **~RD** and **~WR** control signals. In contrast to external memory interfacing, we do not need to latch A[0..7] since A[8..15] is sufficient to address up to 256 io devices.

In this experiment we will construct simple input and output ports using AD[0..7] lines for only data transfer, and A[8..15] lines only for addressing the io devices. The address will be decoded by an address decoder made of 74LS138 and 74LS139 decoders.

8.3 Experimental Part

8.3.1. Memory Mapped I/O interfacing

Objective:
To prepare a workfolder for KC51 IDE and generation of -.hex files for the simulation.
To observe the simulated circuit while it executes the assembled program code on 8051 with a memory mapped output and input interfacing to drive a 7-segment LED and to read four switches.

Procedure-1.a : Preparation of the -.hex file

1- If C:\KC51\ folder is not available download KC51 from the course web page and copy it on hard disk or your flash disk (let's say E:). Correct the PATH statement on the file **E:\KC51\TOOLS.INI** to **PATH="E:\KC51\C51"**. Download and extract **EXP8A.rar** into folder **E:\323\012345\EXP8A**.

2- Start a **-.txt** file with the name "**E:\323\012345\Exp8.txt**" for reporting. Write your student name and number on the first line of the file similar to.

CMPE328 Exp8 Report file by <your-name, surname, student number>

3- Find and start "**…/KC51/UV3/UV3.exe**". Close the projects (menu→project→ close project) if any project is open. Open the project file **"E:\323\012345\Exp8A.Uv2"** . In the Project-Workspace window, click on the target, and the source-group-1 folders to turn on the project source file list. There must be **"Exp8A1.a51"** in your projects sources. If the file is not yet open, open it by clicking on this item.

4- The file shall start with the following lines. Fill in your name and number.

```
; Exp8A1.a51
; Student Name:
; Student Number:
;
; ( c ) 2008 Mehmet Bodur
;$ge

; Display value in RAM memory
; Old keys to detect negative edge.
; Hide/Display flag
Disp     equ R0
```

5- Build the project by clicking to Build-Target button ( ). You shall see the following messages in the "Build" message window if the installation is successful.

```
Build target 'Target 1'
assembling Exp8A1.a51...
linking...
Program Size: data=8.0 xdata=0 code=107
creating hex file from "Exp8A1"...
"Exp8A1" - 0 Error(s), 0 Warning(s).
```

This project contains macros. In the target options, it needs the extended linker and Ax51 instead of A51 assembler; and the output shall be set to create hex file. The list file expands the macros only if listing is set to all-expansions. The macros in this experiment can be handled both by standard and MPL macro processor.

6- Open the project folder "Exp8" in the explorer. From the date and time marks of the files, you will see the most recently created **-.hex** and **-.lst** files.

Reporting:

- Open the **-.lst** file in a text editor, and copy the first 35 lines (including "main:") to your reporting file.

```
A51 MACRO ASSEMBLER  EXP8A1                              05/07/2008 18:32:16 PAGE     1


MACRO ASSEMBLER A51 V8.01
OBJECT MODULE PLACED IN Exp8A1.OBJ
ASSEMBLER INVOKED BY: H:\KC51\C51\BIN\A51.EXE Exp8A1.a51 SET(SMALL) DEBUG EP

LOC  OBJ            LINE     SOURCE

                       1     ; Exp8A1.a51
                       2     ; Student Name:
                       3     ; Student Number:
                       4     ;
                       5     ; ( c ) 2008 Mehmet Bodur
                       6     ;$ge
                       7
                       8     ; Display value in RAM memory
                       9     ; Old keys to detect negative edge.
                      10     ; Hide/Display flag
  REG                 11     Disp    equ R0
  REG                 12     Keys    equ R1
  REG                 13     OldKeys equ R2
  REG                 14     Hide    equ R3
  REG                 15     Tmr     equ R4
                      16
                      17     ; simple output port
  0080                18     PA    equ 80h
                      19     ; simple input port
  0081                20     PB    equ 81h
                      21     ; port-1 for debug
```

```
                         22     ;P1    equ 90h
                         23     ; reset vector
0000                     24        org 0
0000  010C               25        ajmp main
                         26
0002                     27     lutcode:
                         28     ;      gfedcba     gfedcba
0002  3F06               29      db  00111111b, 00000110b
0004  5B4F               30      db  01011011b, 01001111b
0006  666D               31      db  01100110b, 01101101b
0008  7D07               32      db  01111101b, 00000111b
000A  7F6F               33      db  01111111b, 01101111b
                         34
000C                     35     main:
```

- Save your reporting file for other report deliverables.

Procedure-1.b : Execution of the -.hex file on 8051 simulated in ISIS

1- Start **`Portable Proteus 7.6 → ISIS`** in windows.

2- Use **`File→Load design`** to open the file-browser, navigate to **`Exp8`** folder, and load **`Exp8A1.DSN`** file to ISIS. You will get the design window seen in Fig.1.

3- Right click, and then left click once on the 8051 processor. The Program File in the "edit component" window of 8051 shall contain the file name **Exp8A1.hex**, which is generated in Section 3.1. Close the edit-window.

6- Click on button to start the component insertion mode. Click on start button to start simulation.

While the simulation works, a number will appear on the 7-seg-LED display.
Click on UP and DN push-button switches to change the number as you wish.
Click on Hide to make the 7-seg-LED off.
Click on Disp to make the number reappear.
You may observe the bus timing for input and output port using the digital analyzer.

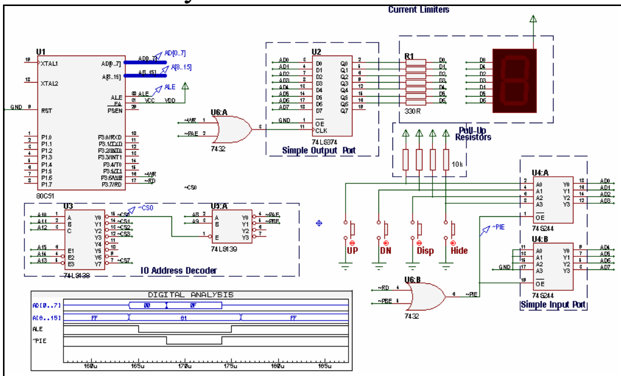


Fig. 1. Experimental Circuit Exp7A0.DSN in ISIS.

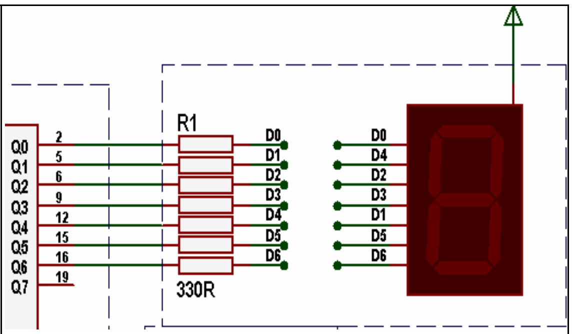


Fig.2 Modified display connections.

```
  lutcode:
;     egfcbda     egfcbda
db  01011111b, 00001100b
db  01100111b, 00101111b
db  00111100b, 00111011b
db  01111011b, 00001101b
db  01111111b, 00111111b
```

Fig.3 Modified lookup table

7- Get from your lab assistant a new combination of connections between port pins and display pins (i.e., Q0 → a, Q1→ d, Q2→b, Q3→c, Q4→f, Q5→g, Q6→e). You shall modify the connections between the 74LS374 and the display accordingly as you see in Fig.2 . Then modify the display-code look-up table in the assembly source for the correct display of the numbers on the display as shown in Fig.3 .

Reporting:

Write the combination given to you by your assistant in a table form like

```
Q: 7 6 5 4 3 2 1 0
D: - e g f c b d a
```

Thereafter copy the first 35 lines of -.lst file obtained with your modified code i.e.

```
AX51 MACRO ASSEMBLER  EXP7B                    05/07/08 18:34:01 PAGE     1
```

```
MACRO ASSEMBLER AX51 V3.03c
OBJECT MODULE PLACED IN Exp7B.OBJ
ASSEMBLER INVOKED BY: H:\KC51\C51\BIN\AX51.EXE Exp7B.a51 SET(SMALL) DEBUG EP

LOC    OBJ             LINE     SOURCE

                          1     ; ( c ) 2008 Mehmet Bodur
                          2     ; Macro Definitions for 8088 style io
                          3     $ge
                          4
                          5     in macro al,p8
                          6       mov DPH,#p8
                          7       movx al,@DPTR
                          8       endm
                          9
                         10     out macro p8,al
                         11       mov A,al
                         12       mov DPH,p8
                         13       movx @DPTR,A
                         14       endm
                         15
 0083                    16     ComR equ 83h
 0080                    17     PA   equ 80h
 0081                    18     PB   equ 81h
                         19     ; start PPI in all output mode.
                         20
000000                   21       org 0
000000 0100       F      22       ajmp main
                         23
000002                   24     lutseg:
                         25     ;       gfedcba    gfedcba
000002 3F06              26       db  00111111b, 00000110b
000004 5B4F              27       db  01011011b, 01001111b
000006 666D              28       db  01100110b, 01101101b
000008 7D07              29       db  01111101b, 00000111b
00000A 7F6F              30       db  01111111b, 01101111b
                         31
00000C 33323824          32     msg:    db '328$'
 0002                    33     msglen  equ 2
                         34
000010                   35     main:
```

Show that the simulation works properly for all numbers to your assistant to get performance points of this experiment.

8.3.2. Interfacing 8255 to 8051 Microcontroller.

Objective:

To observe the slow-motion simulation of the multiplexed 3-digit common-anode 7-segment LED display, and to observe the simulation of a 6 digit common cathode 7-segment LED display at full speed.

Procedure-2.a : Preparation of the -.hex file

1- Start **UV3.exe** . Close the projects (menu→project→ close project) if any project is open. Open the project file **"Exp8B.Uv2"** in the **"KC51/Exp8B"** folder. In the Project-Workspace window, click on the target, and the source-group-1 folders to turn on the project source file list. There must be "**Exp8B.a51**" in your projects sources. If the file is not yet open, open it by clicking on this item.

3- The file shall start with the following lines

```
; Student Name:
; Student Number:
; File: Exp8B.a51
; ( c ) 2008 Mehmet Bodur
; Macro Definitions for 8088 style io
$ge

in macro al,p8
  mov DPH,#p8
  movx al,@DPTR
  endm
```

8- Build the project by clicking to Build-Target button ( ). You shall see the following messages in the "Build" message window if the installation is successful.

```
Build target 'Target 1'
assembling Exp8B.a51...
```

```
linking...
Program Size: data=8.0 xdata=0 const=0 code=87
creating hex file from "Exp7B"...
"Exp7B" - 0 Error(s), 0 Warning(s).
```

9- Open the project folder "**Exp8B**" in the explorer. From the date and time marks of the files, you will see the **-.hex** and **-.lst** files created recently.

Reporting:

- Open the **-.lst** file in a text editor, and copy the first 11 lines (up to the line to your reporting file.

```
AX51 MACRO ASSEMBLER  EXP7B                                    05/07/08 22:01:16 PAGE     1

MACRO ASSEMBLER AX51 V3.03c
OBJECT MODULE PLACED IN Exp7B.OBJ
ASSEMBLER INVOKED BY: C:\_AB\SW\KC51\C51\BIN\AX51.EXE Exp7B.a51 SET(SMALL) DEBUG EP

LOC    OBJ              LINE     SOURCE

                           1     ; Student Name:
                           2     ; Student Number:
                           3     ; File: Exp7B.a51
                           4     ; ( c ) 2008 Mehmet Bodur
                           5     ; Macro Definitions for 8088 style io
                           6     $ge
                           7
                           8     in macro al,p8
                           9       mov DPH,#p8
                          10       movx al,@DPTR
                          11       endm
```

- Save your reporting file for other report deliverables.

Procedure-2.b : Execution of the -.hex file on 8051 simulated in ISIS

1- Start **PortableProteus →ISIS 7 Professional** in windows.

2- Use **File→Load design** to open the file-browser, navigate to **Exp8B** folder, and load **Exp8B.DSN** file to ISIS. You will get the design window seen in Fig.4.

3- Right click, and then left click once on the 8051 processor. The Program File in the "edit component" window of 8051 shall contain the file name **Exp8B.hex**, which is generated in Section 3.2.a. Also check that the Clock Frequency box contains 120k instead of 12M. With this settings, simulation will work 100 times slower than its full speed. Close "edit component" window.

4- Click on  button to start the component insertion mode. Click on start button  to start simulation. While the simulation works, numbers 8, 2 and 3 will appear on the 7-seg-LED displays.

You may observe the bus timing for input and output port using the digital analyzer.

Reporting:

- Look at the program code and explain in two paragraphs what shall you change in hardware and software if you need 8 digits instead of only 3 digits.

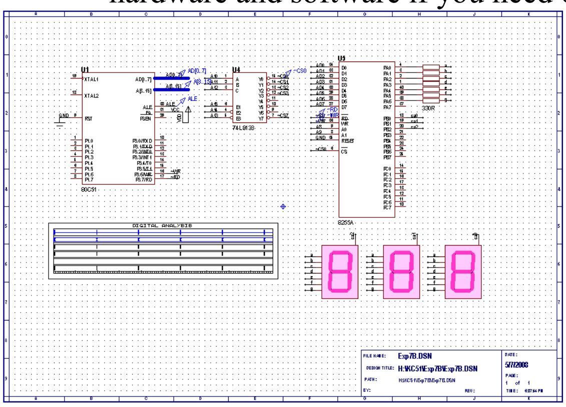


Fig. 4. Experimental Circuit Exp7B.DSN in ISIS.

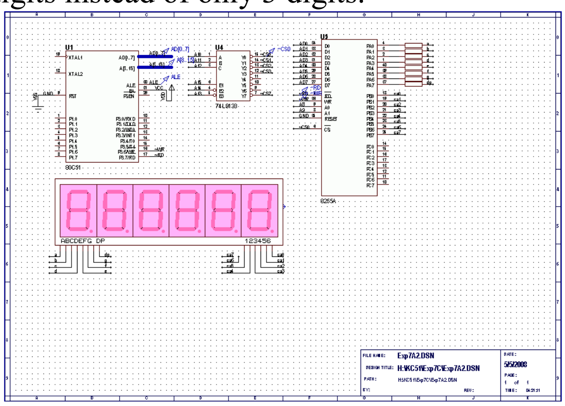


Fig. 5. Experimental Circuit Exp7C.DSN in ISIS.

Procedure-2.c : Common Cathode Displays running at full speed.

Explanation: In the first two experiments you worked with common anode displays. Now, you will use a common cathode 7-segment LED array in this simulation.

1- Start "**UV3.exe**". Close all projects (menu→project→close project). Open the project file **"Exp8C.Uv2"** in the **"Exp8C"** folder. In the Project-Workspace window, click on the target, and the source-group-1 folders to turn on the project source file list. There must be "**Exp8C.a51**" in your projects sources. If the file is not yet open, open it by clicking on this item. Build the project to generate the **-.lst** and **-.hex** files.

Explanation: This code is almost the same with the 3-digit display code you assembled in Procedure 2.b. The only difference is, the digit select changed to active low (i.e., $0FE selects digit-0), and the complement instruction **cpl a** is canceled because common-anode segments need active-high excitation.

3- Start **Proteus7.6Portable→ISIS** in windows and use **File→Load design** to open the file-browser, navigate to **Exp8C** folder, and load **Exp8C.DSN** file to ISIS. You will get the design window seen in Fig.5.

3- Right click, and then left click once on the 8051 processor to open "edit component" window. The Program File of 8051 shall contain the file name **Exp8C.hex**. Also check that the Clock Frequency box contains 12M (it is 12 Mega Hertz, do not confuse with 12m = 12 milliHertz). With this settings, simulation will work at its full speed. Close "edit component" window.

4- Click on  button to start the component insertion mode. Click on start button  to start simulation. While the simulation works, numbers 054321 will appear on the 7-seg-LED displays.

5- Stop the simulation, and set the clock frequency of 8051 to 120k. Then start the the simulation. Write your observation (how the numbers shift) into the report file.

```
Section 2.c
At 12M  clock frequency: … …
At 120k clock frequency: … …
```

6- Before you complete your lab, modify the code to write your student number on the display (at 12M clock frequency) to get the performance grade for this part of the experiment.

8.3.3. Interfacing 8086 to a stepper Motor.

Objective:

The aim of this part is to demonstrate the operation of a stepper motor control by 8086 assembly code.

Procedure-3:

1- Create a subfolder "`E:\323\012345\Exp8D\`" in the KC51 folder. Create a text file in Exp8D folder with the name "`Exp8D2.ASM`". Write the following program into the **Exp8D2.asm** file:

```
; Your Student Number, Name, Surname . . . . . .
; CMPE323 Lab Stepper Motor and UART
; Stepper Motor control.
;
; in the mainloop
;    read a character from UART into rchr
;    if rchr="1" step forward
;    else if rchr="2" step  backward
```

```
;    else do nothing
;    looping in mainloop

 .MODELSMALL
 .8086
 .CODE
  jmp Main

; Data in the code segment
rchr db 0
step db 0
smtb db 3, 6, 12, 9  ; double coil drive

; Code starts here
Main:
  mov AX,CS
  mov DS,AX
  call InitUSART
MainLoop:
  call RecvChar
  ; reads received character into AL.
  ; If no character received then AL returns zero.
  cmp al,0
  jz Mainloop
  mov rchr,al
  cmp rchr,'1'
  jnz skipforward
; forward step
  inc step
  mov bx,0003h
  and bl,step
  mov al,[bx]+offset smtb
  mov dx,324h
  out dx,al
skipforward:
  cmp rchr,'2'
  jnz MainLoop
; backward step
  dec step
  mov bx,0003h
  and bl,step
  mov al,[bx]+offset smtb
  mov dx,324h
  out dx,al
  jmp MainLoop

InitUSART proc
  xor AL, AL
  mov DX, 332h
  out DX, AL
  out DX, AL
  out DX, AL
  mov AL, 40h
  out DX, AL
  mov AL, 04Dh ; 8-bit, no parity, baud=clock x1
  out DX, AL
  mov AL, 05h   ;  start both receive and transmit
  out DX, AL
  ret
```

```
  endp

RecvChar proc
; reads received character into AL.
; If no character received then AL returns zero.
  push DX
  mov DX,332h   ; status/control address
  in  AL,DX     ; read status register
  and AL,02h    ; zero flag is set if AL .AND. 01h is nonzero
  jz NotReceived
  mov DX,330h   ; data-in/data-out address
  in  AL,DX      ; received character transferred from data-in into
  AL.
  shr AL,1
NotReceived:
  pop DX
  ret
  endp

.data
.stack 32

 END
```

2- Use **EMU8086** to assemble the source file to an exe file "**EXP8D2.exe**". Start Proteus-Professional 7.6 ISIS and open **VSED_WA_SMOTOR.DSN** in ISIS. Link the 8086 processors program file to **EXP8D2.EXE** file. Observe how the motor turns when pressing to key "1" and key "2". Write your observation into your report file.

Your report shall contain

EXP8D2:
-With SMTB 3, 6, 12, 9
on key "1" rotor rotates .... (ccw or cw?)
on key "2" rotor rotates .... (ccw or cw?)
when PA is 00000011 the rotor alignes to .... degrees position.
when PA is 00000110 the rotor alignes to .... degrees position.
when PA is 00001100 the rotor alignes to .... degrees position.
when PA is 00001001 the rotor alignes to .... degrees position.

3- Modify the step motor look-up table SMTB to contain 1, 2, 4, 8 instead of 3, 6, 12, 9. Assemble it to **EXP8D2.EXE** and simulate in ISIS with the same circuit. Observe how the motor turns when pressing to key "1" and key "2". Write your observation into your report file.

-With SMTB 1, 2, 4, 8
on key "1" rotor rotates .... (ccw or cw?)
on key "2" rotor rotates .... (ccw or cw?)
when PA is 00000001 the rotor alignes to .... degrees position.
when PA is 00000010 the rotor alignes to .... degrees position.
when PA is 00000100 the rotor alignes to .... degrees position.
when PA is 00001000 the rotor alignes to .... degrees position.

Reporting:

After you complete the procedures, please save and close **exp8.txt** file, and e-mail it using your student e-mail account to **cmpe323lab@gmail.com** with the subject line **"exp8"** within the same day before the midnight.

Late and early deliveries will have 20% discount in grading. No excuse acceptable.

9. Design and Coding of an Intelligent Restaurant Service Terminal

9.1 Objective

The aim of this project is to use an A/D converter, four switches, an LCD and the serial output port of an 8051 to construct an intelligent terminal for the restaurant service stations.

9.2 Introduction

The file **proj09.zip** contains the C code, two header files, and the circuit design file of a 8051 system. The presented system reads an analog voltage and states of four switches, displays these readings on LCD screen, and transmits the digital value through the serial port with 4800 baud, 8-bit, no parity, one stop bit settings. The code is written with student version Keil C compiler. The ISIS circuit schematics design file may be executed using ISIS of the Portable PROSIS 7.6.

9.2.1. Installing KC51 on your drive

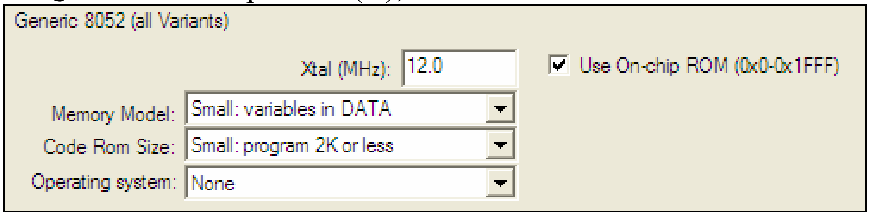
KC51 does not support folder names longer than 32 characters. Therefore you shall copy the **`proj09`** folder to the root of a flash disk (E:) or to your hard disk (C:) drive. For a trouble free operation we recommend to work in folder **`C:\323\012345\proj09\`**, where **`012345`** stands for your student number. Copy **`KC51`** folder to **`C:\KC51\`** so that the folder **`C:\KC51\`** contains folders **`C51`**, **`UV3`** and the **`TOOLS.INI`** file. Edit the **`path`** line of the TOOLS.INI file to change it to **`PATH="C:\KC51\C51\"`** so that KC51 programs can be called while your source file is in folder **`C:\323\012345\proj09\`**. If you copy KC51 folder to another place do not forget to update the path statement accordingly. For example, if **`KC51`** is directly on the root of your flash disk **`E:`**, you shall make the path statement **`PATH="E:\KC51\C51\"`**.

9.2.2. Starting a 8051 or 8052 project in KC51

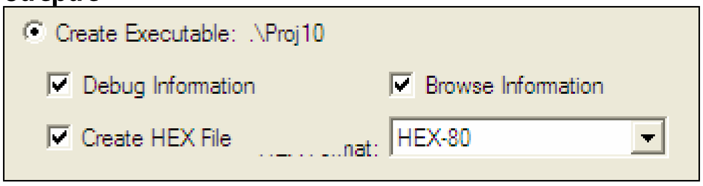
1. Extract **`proj09`** folder to "**`C:\323\012345\`**". If KC51 is not yet installed in your drive the copy **`KC51`** folder to file to **`C:\323\`**, and update PATH statement in the TOOLS.INI file according to installation directives stated in previous subsection.
2 Start **`C:\323\KC51\UV3\Uv3.exe`** and start a "**`New uVision project`**" from project menu. Use "**`Generic`**" and "**`8052 all variants`**", and click "**`No`**" for question "*Copy standard 8052 startup code?*" .

3. With a right-click on **`Target 1`** enter options

`target`: use on chip ROM (X);



`Output`



5 In **`C:\323\012345\proj08`** the template source **`prog08.C`** is available for your use. Add **`prog08.C`** to your project using "**`Add files`**" to "**`Source Group 1`**". Compile it to obtain its hex file.

6 Start ISIS, and open the design file **`C:\KC51_proj08\proj08\Proj.DSN`**, which uses a 8051 (it is also compatible to generic 8052). Link the hex file **"`Proj.hex`"** to the properties of **`8051`**, entry: "program file". Then start simulation in ISIS. It displays the ADC reading and switch readings on LCD display. It also prints the ADC reading to the terminal window when you push SW1.

7 Write your program into the template **`prog08.C`** to satisfy the project requirements. Debug, compile, and simulate in ISIS until you obtain stable operation of the system.

The electronic circuit of this project is available in **`Proj.dsn`** file and it is shown in Fig. 1.

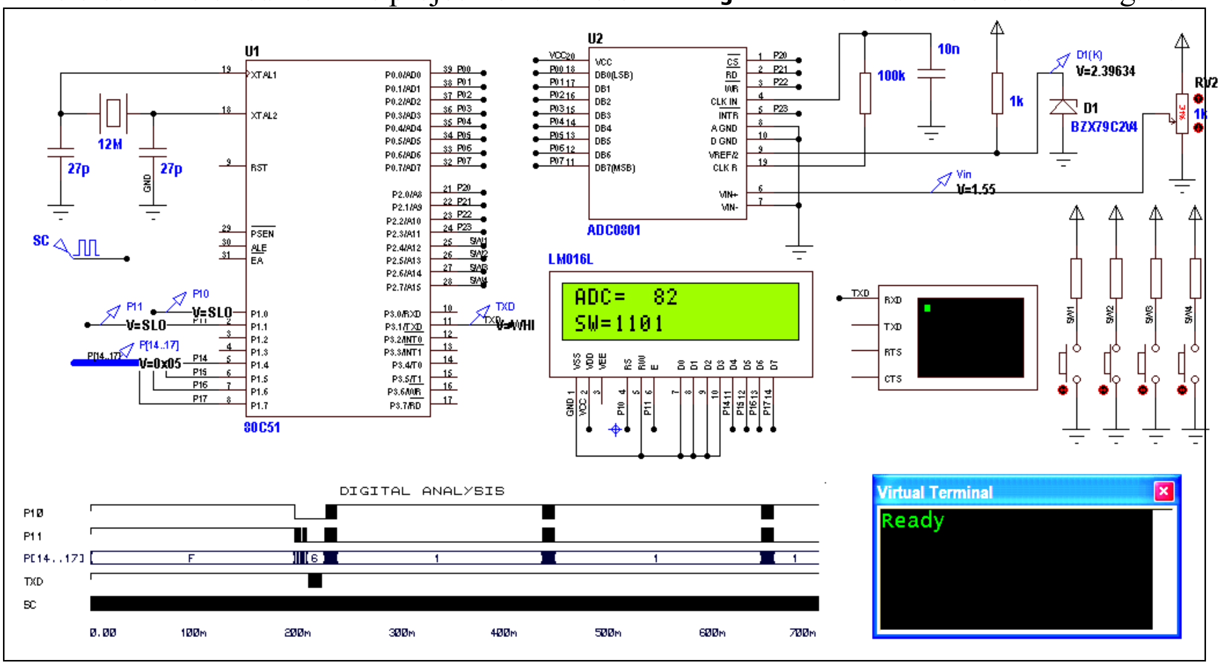


Fig.1 Sample Design Template Circuit

9.2.3. LCD display

The sample code is written for **`LM016L`** (2-line by 16 column) LCD display in 4-bit data transmission mode.

The following bit definitions assign symbols to the port pins for LCD.

```
#include <REG51.H>
#include <stdio.H>
//  Special Function Bits declared for LCD
sbit RS = P1^0; // Control signal RESET of the LCD connected to pin P2.0
sbit EN = P1^1; // Enable (EN) LCD control signal connected to pin P2.2
sbit RW = P1^2; // Write (RW) Signal pin connected to pin P2.1
bit  RSF,RSC ;// RS Flag,
```

where, **RSF** stores the state of control mode (1) or text mode (0). **RS**, **EN** and **RW** declares the symbols corresponding to RS, EN and RW pins of the LCD unit.

The following three subroutines support printing strings on LCD.

The delay(int) procedure

```
void delay(int dd)  { // Delay function.
int j=dd; while(j--);}
```

provides necessary delays for LCD and mainloop. The delay time is proportional to **dd**, and it gives **1 ms** delay for **dd=100**.

The **LCDChar(char)** procedure sends one character to LCD display by making enable signal **EN**=high, and **EN**=low while the higher- and lower-nibbles of the character is applied to the data lines. It also calls sufficient delay (1ms) after sending each control character.

```
void LCDChar(char ch ){
 char Ct=ch;
  P1= Ct&0xF0; if(RSF&&RSC){RS=1;}
  EN=0; delay(10);
  EN=1; delay(10);
  EN=0; delay(5);
  Ct= ch << 4 ;
  P1= Ct&0xF0; if(RSF&&RSC){RS=1;}
  EN=0;  delay(10);
  EN=1;  delay(10);
  EN=0;  delay(5);
  if(!RSF) delay(120);  //1.2ms
}
```

The procedure **PrintLCD(*char)** sends the control and text characters to LCD. As a feature of this procedure, printing a "**\x0FF**" toggles the text mode to control mode by sending the characters with **RS** line high. The printed string must end with a null character as usual in C language.

```
void PrintLCD(char *ch){
  char Ct, n=0 ;
  EN  =0 ; RSF=1;
  Ct=ch[n];
  while(Ct){  RSC=1;
    if(Ct&0x80) {RSC= 0;} // Ct>0x7F -> RSC=0
    if(~Ct==0) {RSF= 0;}  // Ct=0xFF -> RSF=0
    else{ LCDChar(Ct);}
    n++;Ct=ch[n]; }
  }
```

The control characters valid for LM016L-LCD unit is given in the following Table.

Table of command codes for LCD displays

Hex	Action	Hex	Action
01	Clear display screen	**02**	Return home
04	Decrement cursor (shift cursor left)	**05**	Shift display row right
06	Increment cursor (shift cursor right)	**07**	Shift display row left
0C	Display on, Cursor hidden	**0F**	Display on, cursor blinking
10	Shift cursor position to left	**14**	Shift cursor position to right
18	Shift the entire display left	**1C**	Shift the entire display right
28	4-bit data, 2 lines, 5x7 matrix	**38**	8-bit data, 2 lines, 5x7 matrix
	Cursor Placement Commands – row-1		Cursor Placement Commands – row-2
80	Move cursor to 1st column of 1st row	**C0**	Move cursor to 1st column of 2st row
81	Move cursor to 2nd column of 1st row	**C1**	Move cursor to 2nd column of 2st row
…		…	
8F	Move cursor to 16th column of 1st row	**C1**	Move cursor to 16th column of 2st row

The placement of the cursor is achieved with the control codes { **80h, … ,8Fh** } for the first line, and with the control codes { **C0h, … ,CFh** } for the second line. For example, to start the text "**Hello**" from the second line, third column you shall use **PrintLCD("\x0C2Hello"),** where **\x0C2** sets the cursor to second line third column, and the text **Hello** is written to the display. The cursor placement characters are over **0x7F**, and **PrintLCD()** process them as commands without needing a command mode character **\x0FF**.

In the **Init()** procedure, **PrintLCD** sends a collection of commands (**\xff**) to LCD to initialize it to 4-bit mode (**\x02\x28**), clear the display (**\x01**), hide the cursor (**\x0c**), and with each written character shift the cursor to right (**\x06**).

```
void INIT(void){
 // Initialization of the LCD by giving proper commands
 // comm-mode,ret-home,4-bit,clr, hide-cursor, shift-cursor-right
    PrintLCD("\xff\x02\x28\x01\x0c\x06\0"); // Initialize 4-bit LCD.
    ...
```

9.2.4. Serial Port

The 8051 has an on-chip UART to implement serial communication with RS-232 communication protocol. RS232 communication may be useful for user interface as well as in code development

a) to debug embedded applications, using a desktop PC;
b) to load code into flash memory for 'in circuit programming'.
c) to transfer data from embedded data acquisition systems to a PC, or to other embedded processors.

In our project, UART is used to transfer data to a PC at 4800 baud.

8051 UART can work in one of four modes, three of them being asynchronous and one of them synchronous. For the simplicity of the project, we will give the receipt of how to work in mode-1 at 4800 and 9600 baud rates.

In **mode-1**, the baud rate is determined by the overflow rate of Timer 1 or Timer 2. If we use Timer 1, the baud rate is determined by the overflow rate and the value of **SMOD** as follows:

$$\mathbf{BaudRate} = \frac{(\mathtt{SMOD}+1)\cdot \mathtt{Fosc}}{32 \cdot \mathtt{CPI} \cdot (256 - \mathtt{TH1})}$$

where **SMOD** is the '**double baud rate**' bit in the **PCON** register;
Fosc is the oscillator (or resonator) frequency (in Hz);
CPI is the number of machine cycles per instruction (e.g. 12 or 6)
TH1 is the reload value for Timer 1.

With **SCON=0x50**, Using **TH1= FAh** (=250 = – 5), and oscillator frequency **11.06 MHz**, the baud rate becomes **4800**. **TH1**=FDh (= – 3) sets the baud rate to 9800. Thus, the initialization procedure INIT() contains

```
 void INIT(void){
   . . .
// Serial port initialization
   TMOD=0X20; TH1=0x0FA;    // select baud rate 4800
   SCON=0x50;   // set mode-1
   TR1=1;       // start timer.
   TI=0;}
```

which sets Timer-1 to automatic load mode, and serial port to 4800 baud receive/transmit mode so that writing a character to **SBUF** transmits the character. Further, the **char putchar(char)** procedure in **stdio.h** is canceled, and then putchar is declared in the program code as

```
char putchar(char ch){  // For serial output
   SBUF=ch; while(!TI); TI=0; return 0;}
```

so that the `int printf(*char, … )` prints directly to the serial port by calling `putchar`.

9.2.5. ADC interfacing

ADC0801 is a single channel successive approximation register (SAR) AD converter compatible to micro processor system bus interfacing.

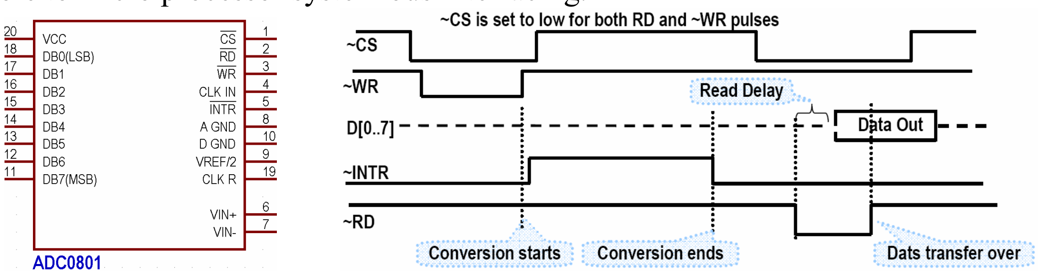


Fig.3. ADC0801 pin layout and control timing

The Pins **DB[1..7]** are connected to system data bus, the control pins **~CS,~RD, ~WR,** are used for chip-select, conversion data read, and ADC start purposes as described in timing chart given in Fig.3. The following port-bits and variables are declared to implement this timing.

```
sbit ADCS   = P2^0; // ADC chip select
sbit ADRD   = P2^1; // ADC read enable
sbit ADWR   = P2^2; // ADC write enable
sbit ADINTR = P2^3; // ADC conversion over
unsigned char ADCVAL;
```

The ADCRead ADC0801 conversion cycle starts by making the port **P0** an input-port. Then, the conversion starts after making **~CS** low, and **~WR** low. **delay(2)** is placed there to observe the port easily on the digital analysis window. The code stays in a loop while **~INTR** is high, which means conversion is not completed.

```
void ADCRead(void){ // Analog Digital Converter
  // Reads ADC into ADCVAL
  //Make the ADC port Input port
  P0=0x0FF;
  // start conversion
  ADCS =0;  ADWR =0;  ADWR =1;
  // wait till conversion is over
  do{}while(ADINTR);
  // read data of ADC into ADCVAL
  ADRD =0; ADCVAL =P0; P2=0x0FF; }
```

Then, the reading of the conversion is written to the global variable **ADCVAL**. Parameter transfer in global variables is frequently seen in microcontroller programming because it is code-efficient.

9.2.6. Switches and Operation of the System

The lower four pins of **P2** port are used for ADC interfacing. ADC read procedure makes **P2** an input port after it completes ADC read operation. The higher 4 pins of **P2** are interfaced to four pushbutton switches, **SW1, SW2, SW3,** and **SW4**. The detection of the press and release instants are obtained by reading the switch states into **SW**, and keeping the old switch states in **SWP**. Both **SW** and **SWP** are 8-bit unsigned global integers.

```
unsigned char  SW, SWP;
```

For the consistency of operation in the mainloop **P2** is read into **SW** only once at the beginning of the mainloop. For coding simplicity, the lower 4-bits are purged out by the shift operation

```
SWP=SW; SW=P2>>4;  // past and present value of switches
```

The switch readings are converted to binary sequence of “0” and “1” characters by

```
j[0]=(SW>>3&1)|'0'; j[1]=(SW>>2&1)|'0';
j[2]=(SW>>1&1)|'0'; j[3]=SW&1|'0'; j[4]=0;
```

You can test the switch status by an if statement

```
if(SW&0x01) { … ;}       // while SW1 released
```

to execute a block of code on switch is open, and

```
if(SW&0x02^0x02) { … ;} // while SW2 pressed
```

to execute the code on switch is closed.

If you need to execute a code only once when a switch is pressed or released. Then, before reading the states of switches into **SW** you shall store the past value of **SW** in **SWP**.

```
if( ( SW&(SW^SWP)&0x04 ) { // once only when SW3 released
```

to execute only once when switch is released (opened).

```
if( ~SW&(SW^SWP)&0x08 ) {  // once only when SW4 pressed
```

and the test for both pressing and releasing is

```
if( (SW^SWP)&0x01 ) {  // once whenever SW1 released or pressed
```

In these three cases, **SW=0x0F** must be initialized (all buttons are released) before the mainloop.

The template code given for this project does the followings in its mainloop

```
void main (void) {
  char num[16]; int i; char j[5];
  delay(20000);  // we need 200ms delay for LCD
  INIT(); // LCD initialized
  printf("Ready\r"); // This goes to UART
  while(1) {
```

```
    ADCRead(); i= (unsigned) ADCVAL;
    SWP = SW; SW=P2>>4;  // past and present value of switches
    if( ((SW^SWP)&~SW)&0x01 ) // only once when sw1 is pressed
          printf("  %u \r" , i);
    j[0]=(SW>>3&1)|'0'; j[1]=(SW>>2&1)|'0';
    j[2]=(SW>>1&1)|'0'; j[3]=SW&1|'0'; j[4]=0;
    sprintf(num,"\x080ADC=%4u\x0C0SW=%s", i, j); PrintLCD(num);
    delay(20000);  // 200ms delay
    }
}
```

1. It waits 200 ms before initializing LCD unit.
2. It initialize serial port for 4800 baudrate operation and prints Ready to the terminal.
3. in the endless while loop (mainloop)
   it reads ADC into unsigned **i**,
   It reads switch status into **SW**, and converts **SW** into binary ASCII string **j[]**.
   It displays **i** and **j** on LCD;
   Whenever **SW1** is pressed, it prints **i** to serial port when switch is pushed (only once).
   It updates past switch status to **SWP** for next pass to detect when SW1 is pushed.
   It stays in delayloop for 200 ms.

9.3 About Keil C51 compiler

REG51.H declares the ports, special function registers, and special function bits of the 8051 processor. **STDIO.H** provides declarations of the procedures **_getkey getchar ungetchar putchar printf sprintf vprintf vsprintf *gets scanf sscanf puts** which are necessary to format the integer and char types into the desired strings.

The **sbit** type is used to declare single bits of special function registers such as **EN**, **RS**, **ADCS, ADRD**. A **bit** variable declares bits in RAM (i.e., **RSF**). The **char** type is used for 8-bit signed integers, **int** is used for 16-bit signed integers. The type qualifier **unsigned** makes both **char** and **int** an unsigned number. The type qualifier **const** makes them constants allocated in RAM area. They are initialized only once at the start of the program. The qualifier **code** allocates the constants in ROM. For example:

```
code char test[] = "This is a text string in ROM";
```

allocates the character string test[] in ROM, along with the program code. The type qualifier **volatile** allocates them in registers, and can be used for very short term temporary purposes.
The **_at_** keyword allows you to specify the address for uninitialized variables in your C source files. It can be used to overlap a memory location for two different data types.

Keep the conditional tests as simple as possible. Use complement (**~**), and (**&**), or (**|**), and ex-or (**^**) for bitwise operations between the **char** and **int** variables or constants. not (**!**) operation complements a single bit, or a relational result. You can test the bits of a **char** variable **S** by using a proper and-mask, i.e., **S&0x40** is nonzero if bit-6 of **S** is high, and similarly **~S&0x40** is nonzero if bit-6 of **S** is low.

9.4 Design Requirements

You will work in Keil C51 microVision-3 environment. You shall set the target options of your microVision project to have

device: Generic **8051**
target: Xtal **11.06 MHz** ; Memory **Small**; Code ROM Size **Small**; Op.Sys. **None**.

output: **Create Hex file**, Name of Executable "**proj**"
listing: check **C compiler listing**, check **Assembly Code**.
C51: add the project folder to the **include path**

so that it will generate **proj.hex** and **proj.lst** files which contains complete assembler coding of the C source using the modified stdio.h.

You will design a service terminal system for restaurants that will have a scale to weigh one of three kinds of food labeled **A**, and **B**. The electronic scale has its own zeroing system, with output voltage in millivolts **Vsc** = **5** M_L , where M_L is the mass on the scale in grams. It is connected to analog input of ADC801. The restaurant uses only one kind of dish plate, which is 100 gram in weight. The ADC801 circuit has **Vref**=4.8V.

In explaining the requirements, we will use the following symbols

NPlate = ADC reading of the food plate, (unsigned char)
GrPlate = Weight of the food plate in grams, (unsigned char)
GrFoodPlate = Measured Weight of the plate with food (int in grams).
GrEmptyPlate = Measured Weight of the empty plate (int in grams).
WeightCoeff = 16*Weight coefficient to calculate weight from ADC reading.
i.e. **GrPlate = NPlate*WeightCoeff/16**
GrFoodA = Weight of the food-A (integer in grams).
GrFoodB = Weight of the food-B (integer in grams).
KrPer10GrA = Price of 10 gram food-A (integer in kr)
KrPer10GrB = Price of 10 gram food-B (integer in kr)
KrPlateA = Price of food A (integer in kr)
KrPlateB = Price of food B (integer in kr)
KrTotal = Total price of the food in the plate (integer in kr).
NewCustomer = New Customer bit. (a flag not to delete the last transmitted readings.)

Your software and hardware design shall satisfy the following requirements.

-The reading **NPlate** is not in grams. It needs to be converted to **GrPlate** using the voltage steps **ΔVA=18.75mV** and the coefficient of the scale output (**Vsc/M**$_L$=**5**), .

GrPlate = WeightCoeff × NPlate /256 = 18.75/15 ×NPlate

Thus,

WeightCoeff =16*16*(GrPlate/NPlate)* 1.25 =20,

For example, the net weight of food-A can be obtained by calculating **GrPlate** for the plate with food into **GrFoodPlate**, and then drop **GrEmptyPlate** from the calculated value.

GrFoodA = GrFoodPlate – GrEmptyPlate .

-Each food type will have a pre-determined constant (Kr (Kurus) per 10 gram) price declared in integer form, typically A is 1.5 Kr/gram (**KrPer10GrA =15**) , B is 2.5 Kr/gram (**KrPer10GrB =25**). The price of the plate shall be calculated depending on the food type. For example, the food-B plate price will be

KrPlateA = GrFoodA × KrPer10GrA /10 .

The following algorithm may be used in coding these requirements.

-Before the main loop your code shall initialize LCD print "**Ready\r\r**" to the terminal, and set **GrFoodA =0, GrFoodB =0, KrTotal =0**.

In the main loop, it shall test the switches for the following actions:

-read **ADC** to get **NPlate** , calculate **GrPlate ,** display it on the first line of the LCD (Add some extra blanks to clear the previously written text, and set the cursor to the beginning of the second line).

-if **SW1** is pressed (it indicates that a plate of food-A is on the scale),

-Store **GrPlate** into **GrFoodPlateA**. Calculate **GrFoodA**.

Display **GrFoodA** on LCD, set **NewCustomer**,

-if **SW2** is pressed, it points that a plate of food B is on the scale,

-Store **GrPlate** into **GrFoodPlateB**. Calculate **GrFoodB**.

Display **GrFoodB** on LCD, set **NewCustomer**,

-if **SW3** is pressed, it means that the total price shall be reported to cashbox,

-Calculate **KrPlateA** using **KrPer10GrA** and **GrFoodA.** Also calculate **KrPlateB** similarly. Find **KrTotal =KrPlateA +KrPlateB** , and print the following report to the serial port

```
A- #### gr
B- #### gr
#### Kr
Bon Appetite.
```

-if **SW4** is pressed, it means the empty plate will be stored,

-Store **GrPlate** into **GrEmptyPlate**, and display **"Empty** " on the second line of the LCD. (The extra space characters aim to clear that part of the LCD.)

-continue to looping in the mainloop forever.

There are some challenges in this design. You shall keep the LCD messages short and easy to understand. Student version of Keil-C51 compiles maximum 2.06k code. The template already consumes 1.4 k code. You shall code your program in code efficient manner to complete the project in 2.1 k code. The followings are remedies for code efficient programming:

1- Do not pass more than a single argument to a procedure, and do not return values from a procedure. Instead, use all variables global, so that you can address them in the procedures freely.

2- Write procedures for all repeating parts of the code, for example to test the switch conditions.

3- PrintLCD, sprintf, and printf uses lots of code. Combine them to each other; i.e., instead of

```
printf("A= %u gr\r",WFA); printf("B= %u gr\r",WFB); use
printf("A= %u gr\rB= %u gr\r",WFA,WFB);
```

9.5 Reporting

You shall write a very short report into the file **proj.txt** about :

- Goal of the developed system.
- Any *difficulties you faced in writing your project code*.
- Any *explanations for the software coding*.
- Any *ideas to improve this project in hardware and in software*.
- A **conclusion** about the *contributions of each member to the project*.

Enumerate the team leader and members, and denote each statement by (idea-owner, editor) in the following manner.

Team Leader: (1) Ibrahim Kisaparmak 012345
Members (2) Rustem Habersiz 054321
(3) Hanefi Hamamci 053412

………..

author other supporters

………..

Combining the LCD messages saved large amount of code memory (231). The calibration of the weight might create problem because the sequence of the multiplication and division operations are critical in calculating WFP (11).

……….

Here, the statement “Combining … (23).” is Rustem’s idea, and Hanefi is author or editor of the statement. Next statement “The calibr…. WFP (11)”. is owned by Ibrahim both in idea and in wording.

After you complete the project, please pack the **-.txt** report file **C code** (**-.C** and **-.H** files), the **-.hex** file, the **-.lst** file, and the **-.DSN** file of your project into a zip file with the name proj.zip and e-mail it using your student e-mail account to **cmpe323lab@gmail.com** with the subject line **“proj”** before the Final Exam Day.

Last day of delivery is Final Exam Day. No excuse acceptable.

10.
Design and Coding of an Intelligent Human Weight Scale

10.1 Objective

The aim of this project is to use an A/D converter, four switches, an LCD and the serial output port of an 8051 to construct an intelligent **`Body Mass Index`** (**`BMI`**) calculator.

10.2 Introduction

This project needs the hardware system and template files described in Chapter 9 for a restaurant terminal design application. Please apply from Sections 9.2 to (including) 9.3 for the preliminary of the project. The technical project specifications of the Body Mass Index Calculator will start from Section 10.4.

10.2.1. Installing KC51 on your drive

Please see Section 9.2.1.

10.2.2. Starting a project in KC51 for 8051 or 8052 projects.

Please see Section 9.2.2.

10.2.3. LCD display

Please see Section 9.2.3.

10.2.4. Serial Port

Please see Section 9.2.4.

10.2.5. ADC interfacing

Please see Section 9.2.5.

10.2.6. Switches and Operation of the System

Please see Section 9.2.6.

10.3 About Keil C51 compiler

Please see Section 9.3.

10.4 Design Requirements

You shall develop a human body weight scale that shall measure the weight of a person by the ADC reading into the 8-bit integer **ADCVAL**.

There are four switches (**SW1, SW2, SW3, SW4**) in the system hardware. The switches **SW1 SW2** and **SW3** shall be used to set the 8-bit integer height **Height**. They shall act only once they are pushed down. The switch **S1** shall toggle an 8-bit integer **StepSize** between **1** and **10**, that is, if switch is pressed while **StepSize** =1, then **StepSize** shall be set to 10. Similarly if switch is pressed while **StepSize** =10, then **StepSize** shall be set to 1. The switch **SW2** shall decrement the body height setting **Height, StepSize** amount down to 120. The switch **SW3** shall increment the body height setting **Height, StepSize** amount up to 210.

The LCD module of the unit shall display the following information

Line1: `W=120 kg  BMI= 53`
Line2: `H=150 cm  *`

where, the height **H** is the value set by switches **SW1, SW2** and **SW2,** the weight **Weight** is calculated from the ADC reading **ADCVAL** by the expression

Weight = (ADCVAL+80)/2 ,

which gives minimum 40 kg while **ADCVAL**=0, and maximum 167 kg while **ADCVAL**=255. Considering the overflow of 16- bit integers, the **BMI** value shall be calculated as

BMI = 100*Weight /Height*100/Height;

The star "*****" on line 2 shall be displayed only if **StepSize** =10, and shall be replaced by a dot "**.**" if **StepSize** =1.

The switch **SW4** shall print a report to the mini printer which is connected to the serial terminal. The contents of the report shall be

```
Date:
Name:
W=120 kg
H=150 cm
BMI = 53
-----------
```

where the empty entries for date and name is going to be filled by the health officer who places the report into the medical file of the person.

There are some challenges in this design. Student version of Keil-C51 compiles maximum 2.06k code. The template already consumes 1.4 k code. You shall code your program in code efficient manner to complete the project in 2.1 k code. The followings are remedies for code efficient programming:

1- Do not pass more than a single argument to a procedure, and do not return values from a procedure. Instead, use all variables global, so that you can address them in the procedures freely.

2- Write procedures for all repeating parts of the code, for example to test the switch conditions.

3- PrintLCD, sprintf, and printf uses lots of code. Combine them to each other; i.e., instead of

printf("W= %u kg\r",WW); printf("H= %u cm\r",HH); use
printf("W= %u kg\rH= %u cm\r",WW,BB);

4- Avoid using single letter variables A, B, … since they are predefined for 8051 registers.

10.5 Reporting

You shall write a short team report into the file **proj.txt** . Each team member shall have at least one or two sentences in the report. The report shall start with

- Team members, and team leaders name, surname and student numbers, in enumerated listing.

  i.e: Team leader: 098760 Kevin Kostner (1),
  Members: 098761 Cameron Diaz (2),
  098762 Robert Redford (3),
  098763 Brad Pitt (4)

  At the end of each sentence give the number of the author and other supporters of that sentence, i.e."**In this project we used a pre-designed hardware for the development of a body weight scale that calculates the Body Mass Index, BMI (134). The software is written in Keil C for a 8051 processor (321). ….** " . Here, the idea of the first sentence has been proposed by Kevin (1), and supported by Robert and Brad. Similarly, idea of the second sentence is owned by Robert, and it is supported both by Cameron and Kevin.

The remaining part of the report shall contain

- Goal of the developed system.
- Any difficulties you faced in writing your project code.
- Any explanations for the software coding.
- Any ideas to develop this project in hardware and in software.
- A conclusion about the contributions of each member to the project.

After you complete the project,

- Please pack the report **proj.txt**, the **C code** (**-.C** and **-.H** files), the **-.hex** file, the **-.lst** file, and the **-.DSN** file of your project into a zip file with the name **proj.zip** and e-mail it using your student e-mail account to **cmpe323lab@gmail.com** with the subject line **"proj"** before the June 15, 2010 midnight .
- Please submit a hardcopy of only proj.txt file (no code, only verbal report) to your instructor, or to lab assistant.

Enjoy the project.

Last day of delivery is Final Exam Day. No excuse acceptable.

11.

APPENDIX

Complete 8086 instruction set

Mnemonics

AAA	CMPSB	JA	JNBE	JPO	MOV	RCL	SCASB
AAD	CMPSW	JAE	JNC	JS	MOVSB	RCR	SCASW
AAM	CWD	JB	JNE	JZ	MOVSW	REP	SHL
AAS	DAA	JBE	JNG	LAHF	MUL	REPE	SHR
ADC	DAS	JC	JNGE	LDS	NEG	REPNE	STC
ADD	DEC	JCXZ	JNL	LEA	NOP	REPNZ	STD
AND	DIV	JE	JNLE	LES	NOT	REPZ	STI
CALL	HLT	JG	JNO	LODSB	OR	RET	STOSB
CBW	IDIV	JGE	JNP	LODSW	OUT	RETF	STOSW
CLC	IMUL	JL	JNS	LOOP	POP	ROL	SUB
CLD	IN	JLE	JNZ	LOOPE	POPA	ROR	TEST
CLI	INC	JMP	JO	LOOPNE	POPF	SAHF	XCHG
CMC	INT	JNA	JP	LOOPNZ	PUSH	SAL	XLATB
CMP	INTO	JNAE	JPE	LOOPZ	PUSHA	SAR	XOR
	IRET	JNB			PUSHF	SBB	

Operand types:

immediate: `5, -24, 3Fh, 10001101b`, etc...
Registers **REG**: `AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP`
Segment Registers **SREG**: `DS, ES, SS`, and only as second operand: `CS`.
memory: `[BX], [BX+SI+7],` variable, etc....

Notes:

When two operands are required for an instruction they are separated by comma, i.e.,
`REG, memory`
When there are two operands, both operands must have the same size (except shift and rotate instructions). For example:

```
registers
  AL, DL
  DX, AX
m1 DB ?
  AL, m1
m2 DW ?
  AX, m2
```

Some instructions allow several operand combinations. For example:

```
memory, immediate
REG, immediate
memory, REG
REG, SREG
```

These marks are used to show the state of the flags:
1 - instruction sets this flag to 1.
0 - instruction sets this flag to 0.
r - flag value depends on result of the instruction.
u - flag value is undefined (maybe 1 or 0).
n – flag value is not changed.

Some instructions generate exactly the same machine code, so disassembler may have a problem decoding to your original code. This is especially important for Conditional Jump instructions

Instructions in alphabetical order:

|Only selected instructions are explained in detail.

AAA No operands	**ASCII Adjust after Addition.**
	Corrects result in AH and AL after addition when working with BCD values.
	if low nibble of AL > 9 or AF = 1 **then**
	AL = AL + 6; AH = AH + 1; AF = 1 ; CF = 1 ;
	else AF = 0 ; CF = 0 **endif**
	AL = AL & 0x0F;
	Example:
	`MOV AX, 15   ; AH = 00, AL = 0Fh`
	`AAA          ; AH = 01, AL = 05`
	Flags: **r{C, A}**

AAD No operands	**ASCII Adjust before Division**.
	Prepares two BCD values for division.
	AL = (AH * 10) + AL ; AH = 0 ;
	Example:
	`MOV AX, 0105h   ; AH = 01, AL = 05`
	`AAD             ; AH = 00, AL = 0Fh (15)`
	Flags: **r{Z,S,A}**

AAM No operands	**ASCII Adjust after Multiplication**.
	Corrects the result of multiplication of two BCD values.
	AH = AL / 10 ; AL = remainder ;
	Example:
	`MOV AL, 15   ; AL = 0Fh`
	`AAM          ; AH = 01, AL = 05`
	Flags: **r{Z,S,P}**

AAS No operands	**ASCII Adjust after Subtraction**.
	Corrects result in AH and AL after subtraction when working with BCD values.
	if low nibble of AL > 9 or AF = 1 **then**:
	AL = AL – 6; AH = AH – 1 ; AF = 1 ; CF = 1 ;
	else AF = 0 ; CF = 0 **endif**
	AL = AL & 0x0F;
	Example:
	`MOV AX, 02FFh  ; AH = 02, AL = 0FFh`
	`AAS            ; AH = 01, AL = 09`
	Flags: **r{C, A}**

<table>
<tr><td>ADC op1,op2
REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td><td>Add with Carry.
operand1 = operand1 + operand2 + CF
Example:
STC ; set CF = 1
MOV AL, 5 ; AL = 5
ADC AL, 1 ; AL = 7
Flags: r{C,Z,S,O,P,A}</td></tr>
<tr><td>ADD op1,op2
REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td><td>Add.
operand1 = operand1 + operand2
Example:
MOV AL, 5 ; AL = 5
ADD AL, -3 ; AL = 2
Flags: r{C,Z,S,O,P,A}</td></tr>
<tr><td>AND op1,op2
REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td><td>Logical AND between all bits of two operands.
Result is stored in operand1.

These rules apply:
1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0

Example:
MOV AL, 'a' ; AL = 01100001b
AND AL, 11011111b ; AL = 01000001b ('A')
Flags: 0{C,O}, r{Z,S,P}</td></tr>
<tr><td>CALL addr
procedure name
label
4-byte address</td><td>Transfers control to procedure,
IP (return address) is pushed to stack.
For 4-byte address first value is a segment second value is an offset (this is a far call, so CS is also pushed to stack).
Example:
ORG 100h ; for COM file.
CALL p1
ADD AX, 1
. . . ; continue to code.
p1 PROC ; procedure declaration.
MOV AX, 1234h
RET ; return to caller.
p1 ENDP
Flags: not changed</td></tr>
<tr><td>CBW No operands</td><td>Convert byte into word.
if high bit of AL = 1 then AH = 255 (0FFh) else AH = 0 endif
Example:
MOV AX, 0 ; AH = 0, AL = 0
MOV AL, -5 ; AX = 000FBh (251)
CBW ; AX = 0FFFBh (-5)
Flags: not changed</td></tr>
<tr><td>CLC No operands</td><td>Clear Carry flag.
CF = 0
Flags: C=0</td></tr>
<tr><td>CLD No operands</td><td>Clear Direction flag. SI and DI will be incremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW.
Flags 0{D}</td></tr>
</table>

CLI No operands	**Clear Interrupt enable flag**. This disables hardware interrupts. Flags: **0{I}**
CMC No operands	**Complement Carry flag.** Inverts value of CF. Flags: **r{C}**

CMP op1,op2
REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate
	Compare.
	operand1 - operand2
	result is not stored anywhere, flags are set (OF, SF, ZF, AF, PF, CF) according to result.
	Example:
	`MOV AL, 5`
	`MOV BL, 5`
	`CMP AL, BL  ; AL = 5, ZF = 1 (so equal!)`
	Flags: **r{C,Z,S,O,P,A }**

CMPSB No operands	**Compare bytes**: ES:[DI] from DS:[SI]. Flags: **r{C,Z,S,O,P,A }**
CMPSW No operands	**Compare words**: ES:[DI] from DS:[SI]. Flags: **r{C,Z,S,O,P,A }**

CWD No operands	**Convert Word to Double word**.
	if high bit of AX =1 **then** DX=65535 (0FFFFh) **else** DX = 0
	endif
	Example:
	`MOV DX, 0   ; DX = 0`
	`MOV AX, 0   ; AX = 0`
	`MOV AX, -5  ; DX AX = 00000h:0FFFBh`
	`CWD         ; DX AX = 0FFFFh:0FFFBh`
	Flags: not changed

DAA No operands	**Decimal adjust After Addition**.
	Corrects the result of addition of two packed BCD values.
	
	Algorithm:
	if low nibble of AL > 9 or AF = 1 **then** AL = AL+6; AF = 1; **endif**
	if AL > 9Fh or CF = 1 **then** AL = AL+60h ; CF =1; **endif**
	Example:
	`MOV AL, 0Fh   ; AL = 0Fh (15)`
	`DAA           ; AL = 15h`
	Flags: **r{C,Z,S,O,P,A }**

DAS No operands	**Decimal adjust After Subtraction**.
	Corrects the result of subtraction of two packed BCD values.
	if low nibble of AL > 9 or AF=1 **then** AL =AL-6; AF = 1;
	endif;
	if AL > 9Fh or CF = 1 **then** AL = AL - 60h ; CF = 1; **endif**
	Example:
	`MOV AL, 0FFh   ; AL = 0FFh (-1)`
	`DAS            ; AL = 99h, CF = 1`
	Flags: **r{C,Z,S,O,P,A }**

DEC op
REG
memory
	Decrement.
	operand = operand - 1
	Example:
	`MOV AL, 255   ; AL = 0FFh (255 or -1)`
	`DEC AL        ; AL = 0FEh (254 or -2)`
	Flags: **r{Z,S,O,P,A }**, **n{C}** Carry flag is not changed.

DIV op
REG
memory
	Unsigned divide.
	when operand is a byte:
	AL = AX / operand; AH = remainder (modulus)
	when operand is a word:
	AX = (DX AX) / operand ; DX = remainder (modulus)
	Example:
	`MOV AX, 203   ; AX = 00CBh`
	`MOV BL, 4`
	`DIV BL        ; AL = 50 (32h), AH = 3`
	Flags: All Unknown

HLT No operands	**Halt the System**.

IDIV op
REG
memory
	Signed divide.
	when operand is a byte:
	AL = AX / operand; AH = remainder (modulus)
	when operand is a word:
	AX = (DX AX) / operand ; DX = remainder (modulus)
	Example:
	`MOV AX, -203 ; AX = 0FF35h`
	`MOV BL, 4`
	`IDIV BL  ; AL = -50 (0CEh), AH = -3 (0FDh)`
	Flags: All Unknown

IMUL op
REG
memory
	Signed multiply.
	when operand is a byte: AX = AL * operand.
	when operand is a word: (DX AX) = AX * operand.
	Example:
	`MOV AL, -2`
	`MOV BL, -4`
	`IMUL BL      ; AX = 8`
	Flags: **0{C, O }, u{ Z,S,P,A }**
	when result fits into operand of IMUL then **0{C,O}** .

IN op1,op2 AL, im.byte AL, DX AX, im.byte AX, DX	**Input from port** into AL or AX. Second operand is a port number. If required to access port number over 255 - DX register should be used. Flags not affected

INC op
REG
memory
	Increment.
	Algorithm: operand = operand + 1
	Example:
	`MOV AL, 4`
	`INC AL       ; AL = 5`
	Flags **r{Z,S,O,P,A}, n{C}**

INT imm
immediate byte
	Interrupt numbered by immediate byte (0..255).
	Push to stack: flags register , CS , IP . IF = 0 .
	Transfer control to interrupt procedure
	Example:
	`MOV AH, 4Ch  ; Terminate and Exit to DOS.`
	`INT 21h      ; BIOS interrupt.`
	Flags **n{ C,Z,S,O,P,A,I}**

INTO No operands	**Interrupt 4** if Overflow flag is 1.
IRET No operands	**Interrupt Return.** Pop from stack: IP , CS, flags register Flags C,Z,S,O,P,A,I popped from stack

JA addr label	**Jump if Above.** Short Jump relative to IP for **Unsigned** compare. Jump if first operand is **Above** second operand when used after CMP instruction. **if** (CF = 0) and (ZF = 0) **then** jump **endif** `Flags not changed`
JAE addr label	**Jump if Above or Equal.** Short Jump relative to IP for **Unsigned** compare. Jump if first operand is **Above or Equal to** second operand when used after CMP instruction. **if** CF = 0 **then** jump **endif** `Flags not changed`
JB addr label	**Jump if Below.** Short Jump relative to IP for **Unsigned** compare. Jump if first operand is **Below** second operand when used after CMP instruction. **if** CF = 1 **endif** jump **endif** `Flags not changed`
JBE addr label	**Jump if Below or Equal.** Short Jump relative to IP for **Unsigned** compare. Jump if first operand is **Below or Equal to** second operand when used after CMP instruction. **if** CF = 1 or ZF = 1 **then** jump **endif** `Flags not changed`
JC addr label	**Jump on Carry.** Short Jump if Carry flag is set to 1. **if** CF = 1 **then** jump **endif** `Flags not changed`
JCXZ addr label	**Jump if CX is Zero.** **if** CX = 0 **then** jump **endif** `Flags not changed`
JE addr label	**Jump if Equal.** Short Jump relative to IP for **Signed** and **Unsigned** compare. Jump if first operand is **Equal to** second operand when used after CMP instruction. **if** ZF = 1 **then** jump **endif** `Flags not changed`
JG addr label	**Jump if Greater than.** Short Jump relative to IP for **Signed** compare. Jump if first operand is **Greater than** second operand when used after CMP instruction. **if** (ZF = 0) and (SF = OF) **then** jump **endif** `Flags not changed`
JGE addr label	**Jump if Greater than or Equal to.** Short Jump relative to IP for **Signed** compare. Jump if first operand is **Greater than or Equal to** second operand when used after CMP instruction. **if** SF = OF **then** jump **endif** `Flags not changed`
JL addr label	**Jump if Less than .** Short Jump relative to IP for **Signed** compare. Jump if first operand is **Less than** second operand when used after CMP instruction. **if** SF <> OF **then** jump **endif** `Flags not changed`
JLE addr label	**Jump if Less than or Equal to.** Short Jump relative to IP for **Signed** compare. Jump if first operand is **Less than or Equal to** second operand when used after CMP instruction. **if** SF <> OF or ZF = 1 **then** jump **endif** `Flags not changed`

JMP addr label 4-byte address	**Jump Always.** This unconditional jump transfers control to another part of the program. 4-byte address may be entered in this form: 1234h:5678h, first value is a segment second value is an offset. `Flags not changed`
JNA addr label	**Jump if Not Above .** Same as JB (jump below or equal) instruction. `Flags not changed`
JNAE addr label	**Jump if Not Above or Equal .** Same as JB (jump below) instruction. `Flags not changed`
JNB addr label	**Jump if Not Below .** Same as JAE (jump above or equal) instruction. `Flags not changed`
JNBE addr label	**Jump if Not Below or Equal .** Same as JA (jump above) instruction. `Flags not changed`
JNC addr label	**Jump if No Carry.** Short Jump if Carry flag is zero. **if** CF = 0 **then** jump **endif** `Flags not changed`
JNE addr label	**Jump if Not Equal .** Short Jump relative to IP for **Signed** or **Unsigned** compare. Jump if first operand is **Not Equal to** second operand when used after CMP instruction. **if** ZF = 0 **then** jump **endif** `Flags not changed`
JNG addr label	**Jump if Not Greater than .** Same as JLE (jump less or equal) instruction. `Flags not changed`
JNGE addr label	**Jump if Not Greater than or Equal .** Same as JL (jump less than) instruction. `Flags not changed`
JNL addr label	**Jump if Not Less than .** Same as JGE (jump greater or equal) instruction. `Flags not changed`
JNLE addr label	**Jump if Not Less or Equal .** Same as JG (jump greater) instruction. `Flags not changed`
JNO addr label	**Short Jump if Not Overflow.** `Flags not changed`
JNP addr label	**Short Jump if No Parity.** Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. **if** PF = 0 **then** jump **endif** `Flags not changed`
JNS addr label	**Short Jump if Not Signed (positive).** Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. **if** SF = 0 **then** jump **endif** `Flags not changed`
JNZ addr label	**Short Jump if Not Zero.** Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions **if** ZF = 0 **then** jump **endif** `Flags not changed`
JO addr label	**Short Jump if Overflow.** **if** OF = 1 **then** jump **endif** `Flags not changed`
JP addr label	**Short Jump if Parity (even).** Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. **if** PF = 1 **then** jump **endif** `Flags not changed`
JPE addr label	**Short Jump if Parity Even.** Same as JP (Jump if Parity) instruction `Flags not changed`

JPO addr label	**Short Jump if Parity Odd.** Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Same as JNP (jump if no parity) instruction. `Flags not changed`
JS addr label	**Short Jump if Signed (if negative).** Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. **if** SF = 1 **then** jump **endif** `Flags not changed`
JZ addr label	**Short Jump if Zero (equal).** Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. **if** ZF = 1 **then** jump **endif** `Flags not changed`
LAHF No operands	**Load AH from 8 low bits of Flags register.** AH = flags register flag bits**: `7:SF, 6:ZF, 5:0, 4:AF, 3:0, 2:PF, 1:1, 0:CF`** bits 1, 3, 5 are reserved. `Flags not changed`
LDS op,mem REG, memory	**Load memory double word into word register and DS.** REG = first word DS = second word `Flags not changed`
LEA op,mem REG, memory	**Load Effective Address.** REG = address of memory (offset) Example: **`MOV BX, 35h`** **`MOV DI, 12h`** **`LEA SI, [BX+DI]     ; SI = 35h + 12h = 47h`** Assembler may replace LEA with a more efficient MOV where possible. `Flags not changed`
LES op,mem REG, memory	**Load memory double word into word register and ES.** `Flags not changed`
LODSB No operands	**Load byte at DS:[SI] into AL. Update SI.** `Flags not changed`
LODSW No operands	**Load word at DS:[SI] into AX. Update SI.** `Flags not changed`
LOOP addr label	**Decrease CX, jump to label if CX not zero.** CX = CX – 1 **if** CX <> 0 **then** jump **else** no jump, continue **endif** `Flags not changed`
LOOPE addr label	**Decrease CX, jump to label if CX not zero and Equal** (ZF = 1). CX = CX – 1 **if** (CX <> 0) and (ZF = 1) **then** jump **else** no jump, continue **endif** `Flags not changed`
LOOPNE addr label	**Decrease CX, jump to label if CX not zero and Not Equal** (ZF = 0). CX = CX – 1 **if** (CX <> 0) and (ZF = 0) **then** jump **else** no jump, continue **endif** `Flags not changed`
LOOPNZ addr label	**Same as LOOPNE** `Flags not changed`
LOOPZ addr label	**Same as LOOPE** `Flags not changed`

MOV op1,op2 REG, memory memory, REG REG, REG memory, immediate REG, immediate SREG, memory memory, SREG REG, SREG SREG, REG	**Copy operand2 to operand1.** operand1 = operand2 Restrictions: The MOV instruction cannot set the value of the CS and IP registers. Copying value of one segment register to another segment register requires first copying to a general register. Copying an immediate value to a segment register requires first copying to a general register first. `Flags not changed`
MOVSB No operands	**Copy byte at DS:[SI] to ES:[DI]. Update SI and DI.** ES:[DI] = DS:[SI] **if** DF = 0 **then** SI = SI + 1, DI = DI + 1, **else** SI = SI – 1, DI = DI – 1, **endif** `Flags not changed`
MOVSW No operands	**Copy word at DS:[SI] to ES:[DI]. Update SI and DI.** ES:[DI] = DS:[SI] **if** DF = 0 **then** SI = SI + 2, DI = DI + 2 , **else** SI = SI – 2 , DI = DI – 2 **endif** `Flags not changed`

MUL op
REG
memory
	Unsigned multiply.
	when operand is a byte: AX = AL * operand.
	when operand is a word: (DX AX) = AX * operand.
	Example:
	`MOV AL, 200   ; AL = 0C8h`
	`MOV BL, 4`
	`MUL BL        ; AX = 0320h (800)`
	Flags **r{C, O}. 0{CF,OF}** when high section of the result is zero.

NEG op REG memory	**Negate. Makes operand negative (two's complement).** Invert all bits of the operand. Add 1 to inverted operand Flags **r{C,Z,S,O,P,A}**
NOP No operands	**No Operation.** Flags not changed
NOT op REG memory	**Invert each bit of the operand.** Flags not changed
OR op1,op2 REG, memory memory, REG REG, REG memory, immediate REG, immediate	**Logical OR** between all bits of two operands. Result is stored in first operand. Flags **0{C, O}, r{ Z,S, P,A}**
OUT op1,op2 immediate-byte, AL immediate-byte, AX DX, AL DX, AX	**Output from AL or AX to port.** First operand is a port number. If required to access port number over 255 - DX register should be used. Flags not changed
POP op REG SREG memory	**Get 16 bit value from the stack**. operand = SS:[SP] (top of the stack) SP = SP + 2 Flags not changed

POPA No operands (80186 +)	**Pop all general purpose registers DI, SI, BP, SP, BX, DX, CX, AX from the stack** (SP value is ignored, it is Popped but not set to SP register). it works with 80186 and later POP DI POP SI POP BP POP xx (SP value ignored) POP BX POP DX POP CX POP AX Flags not changed
POPF No operands	**Get flags register from the stack.** flags = SS:[SP] (top of the stack) SP = SP + 2 Flags popped from stack
PUSH op REG SREG memory immediate (80186 +)	**Store 16 bit value in the stack.** PUSH immediate works only on 80186 CPU and later! Flags not changed
PUSHA No operands (80186 +)	**Push all general purpose registers AX, CX, DX, BX, SP, BP, SI, DI in the stack.** Original value of SP register (before PUSHA) is used. Note: this instruction works only on 80186 CPU and later! PUSH AX PUSH CX PUSH DX PUSH BX PUSH SP PUSH BP PUSH SI PUSH DI Flags not changed
PUSHF No operands	**Push flags register in the stack.** SP = SP - 2 SS:[SP] (top of the stack) = flags Flags not changed
RCL op1,op2 memory, immediate REG, immediate memory, CL REG, CL	**Rotate operand1 left through Carry Flag.** The number of rotates is set by operand2. When immediate is greater then 1, assembler generates several RCL xx, 1 instructions because 8086 has machine code only for this instruction . shift all bits left, the bit that goes off is set to CF and previous value of CF is inserted to the right-most position. Flags **r{C,O}** . **O{OF}** if first operand keeps original sign.
RCR op1,op2 memory, immediate REG, immediate memory, CL REG, CL	**Rotate operand1 right through Carry Flag**. The number of rotates is set by operand2. When immediate is greater then 1, assembler generates several RCL xx, 1 instructions because 8086 has machine code only for this instruction . shift all bits right, the bit that goes off is set to CF and previous value of CF is inserted to the left-most position. Flags **r{C,O}** . **O{OF}** if first operand keeps original sign.

REP chain instruct	**Repeat following MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW instructions CX times**.
	if CX<>0 **then**
	do repeat
	execute next chain instruction; CX = CX – 1;
	until CX==0 **enddo**
	endif
	Flag **r{Z}**

REPE chain instruct
REPZ chain instruct
	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Equal), maximum CX times.
	if CX<>0 **then**
	do repeat
	execute next chain instruction;
	CX = CX – 1;
	until ZF==0 && CX==0 **enddo**
	endif
	Flag **r{Z}**

REPNE chain instruct
REPNZ chain instruct
	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Equal), maximum CX times.
	if CX<>0 **then**
	do repeat
	execute next chain instruction;
	CX = CX – 1;
	until ZF==1 && CX==0 **enddo**
	endif
	Flag **r{Z}**

RET No operands or even immediate	**Return from near procedure.** Pop from stack: IP **if** immediate operand is present: **then** SP = SP + operand **endif** Flags not changed
RETF No operands or even immediate	**Return from Far procedure**. Pop from stack: IP, CS **if** immediate operand is present: **then** SP = SP + operand **endif** Flags not changed
ROL op1,op2 memory, immediate REG, immediate memory, CL REG, CL	**Rotate operand1 left.** The number of rotates is set by operand2. When immediate is greater then 1, assembler generates several ROL xx, 1 instructions because 8086 has machine code only for this instruction . shift all bits left, the bit that goes off is set to CF and the same bit is inserted to the right-most position. Flags **r{C, O},** OF=0 if first operand keeps original sign.
ROR op1,op2 memory, immediate REG, immediate memory, CL REG, CL	**Rotate operand1 right.** The number of rotates is set by operand2. When immediate is greater then 1, assembler generates several ROR xx, 1 instructions because 8086 has machine code only for this instruction . shift all bits right, the bit that goes off is set to CF and the same bit is inserted to the left-most position. Flags **r{C, O},** OF=0 if first operand keeps original sign
SAHF No operands	**Store AH register into low 8 bits of Flags register.** flags register = AH flag bits**: 7:SF, 6:ZF, 5:0, 4:AF, 3:0, 2:PF, 1:1, 0:CF** bits 1, 3, 5 are reserved. Flags **r{C,Z,S,O,P,A}**

SAL op1,op2 memory, immediate REG, immediate memory, CL REG, CL	**Shift Arithmetic operand1 Left.** The number of shifts is set by operand2. When immediate is greater then 1, assembler generates several SAL xx, 1 instructions because 8086 has machine code only for this instruction . Shift all bits left, the bit that goes off is set to CF. Zero bit is inserted to the right-most position. Flags **C, O** updated. OF=0 if first operand keeps original sign.
SBB op1, op2 REG, memory memory, REG REG, REG memory, immediate REG, immediate	**Subtract with Borrow.** operand1 = operand1 - operand2 - CF Flags: **r{C,Z,S,O,P,A}**. **CF** is used as Borrow-flag.
SCASB No operands	**Compare bytes: AL from ES:[DI].** AL - ES:[DI]; set flags according to result: OF, SF, ZF, AF, PF, CF **if** DF = 0 **then** DI = DI + 1 **else** DI = DI - 1 **endif** Flags: **r{C,Z,S,O,P,A}**
SCASW No operands	Compare words: AX from ES:[DI]. AX - ES:[DI]; set flags according to result: OF, SF, ZF, AF, PF, CF **if** DF = 0 **then** DI = DI + 2 **else** DI = DI - 2 **endif** Flags: **r{C,Z,S,O,P,A}**
SHL op1,op2 memory, immediate REG, immediate memory, CL REG, CL	**Shift operand1 Left**. The number of shifts is set by operand2. When immediate is greater then 1, assembler generates several SHL xx, 1 instructions because 8086 has machine code only for this instruction . Shift all bits left, the bit that goes off is set to CF. Zero bit is inserted to the right-most position. Flags **C, O** updated. OF=0 if first operand keeps original sign.
SHR op1,op2 memory, immediate REG, immediate memory, CL REG, CL	**Shift operand1 Right.** The number of shifts is set by operand2. When immediate is greater then 1, assembler generates several SHR xx, 1 instructions because 8086 has machine code only for this instruction . Shift all bits right, the bit that goes off is set to CF. Zero bit is inserted to the left-most position. Flags **r{C, O}** OF=0 if first operand keeps original sign.
STC No operands	**Set Carry flag.** Flags: **1{C}**
STD No operands	**Set Direction flag.** SI and DI will be decremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Flags: **1{D}**
STI No operands	**Set Interrupt enable flag.** This enables hardware interrupts. Flags: **1{I}**
STOSB No operands	**Store byte in AL into ES:[DI]. Update DI.** ES:[DI] = AL **if** DF = 0 **then** DI = DI + 1 **else** DI = DI - 1 **endif** Flags are not changed
STOSW No operands	**Store word in AX into ES:[DI]. Update DI.** ES:[DI] = AX **if** DF = 0 **then** DI = DI + 2 **else** DI = DI - 2 **endif** Flags are not changed

SUB op1,op2 REG, memory memory, REG REG, REG memory, immediate REG, immediate	**Subtract.** operand1 = operand1 - operand2 Flags: **r{C,Z,S,O,P,A}**
TEST op1,op2 REG, memory memory, REG REG, REG memory, immediate REG, immediate	**Logical AND between all bits of two operands for flags only**. These flags are effected: ZF, SF, PF. Result is not stored anywhere. Flags: **0{C,O}, r{Z,S,P}**
XCHG op1,op2 REG, memory memory, REG REG, REG	**Exchange values of two operands.** operand1 < - > operand2 Flags are not changed
XLATB No operands	**Translate byte from table.** Copy value of memory byte at DS:[BX + unsigned AL] to AL register. AL = DS:[BX + unsigned AL] Flags are not changed
XOR op1,op2 REG, memory memory, REG REG, REG memory, immediate REG, immediate	**Logical XOR (Exclusive OR) between all bits of two operands.** Result is stored in first operand. Flags: **0{C,O}, r{Z,S,P}**. AF is unknown.

Summary Sheet for Assembly Programming

INT 10h BIOS services

AH=00h Set video mode
AL=03, CGA text mode
AL=04, 320x200 graphics mode 4-color.
AL=06, 640x200 hi-res graph.mode B/W.
AL=07, monochrome text mode
AH=02h Set cursor location
DH=row, DL=col. BH=page
AH=03h Get cursor location, It returns
DH=row, DL=col. BH=page. CX=cursor
AH=06h Clearing the screen
AL=0 for entire page, BH=7 attributes,
CH=0, CL=0, row and col to start.
DH=24, DL=80, row and col to end.
AH=0Ch Set a pixel in graphics screen.
AL=(0 black, or 1 for white),
CX=col; DX=row,

INT 21h DOS services

AH=01h wait and echo a single character.
AL returns char keyed to the keyboard.
AH=02h display a character on the monitor
L= ASCII coded char to be displayed.
AH=09h display a string to the monitor
DX= offset of ASCII string ending with “$”.
AH=0Ah wait a string input ending with <cr>
DX=offset of buffer area .
Input returns in the buffer
buffer area = {size, length, contents}
05 00 20 20 20 20 20 is buffer of 5 char,
05 03 33 32 38 0D 20 contains “328<cr>”.

INT 16h Keyboard Service

AH=01h (checks if any key is pressed)
ZF=1 if no keys pressed).
AH=00h (it is used only after AH=01h, it returns the pressed key in AL).

Assembler Directives

.model [tiny|small|compact|medium|large|huge]
.data (defines the start of data segment)
.code (defines the start of code sector)
.stack n (defines the size of stack segment)
@data (data segment allocated by OS.)
<name> equ value (assigns name=value)
db value (allocate byte with value)
dw, dd, dq alloc.word, double-word, quad-word
dt (allocate 10 digit unpacked-BCD.)
n dup(value) (duplicate value for n times.)
<proclabel> proc [short|near|far]…endp (define procedure)
end (end of assembler source.)
<macrolabel> macro argumentlist…endm (define macro).
include filename.extension (include a file)

80x86 Instruction formats

mov dst,src (move data)
movsx - movzx (move 8-bit into16-bit reg, 386)
cbw reg (convert byte to sign ext. word, 386)
cwd reg (convert word to sign ext. double, 386)
cbw (convert byte al to word ax)
cwd (convert word ax to doubleword dx:ax)
clc / stc (clear / set carry flag)

add dst,src (dst=dst+src ; add)
adc dst,src (dst=dst+src+CF ; add with carry)
sub dst,src (dst=dst - src ; subtract)
sbb dst, src (dst=dst - src - CF; sub with carry)
daa (decimal adjust add);
das (dec.adjust sub)
aaa (ascii or unpacked-BCD adjust addition.)
and dst,src ; or dst,src ; xor dst,src (logical)
neg dst (negation of binary by 2’s complement)
shl dst,1 - shl dst,cl (shift left 1-bit, cl bits);
shr dst,1 - shr dst,cl (shift right 1-bit, cl bits);
sar dst,1 - sar dst,cl (arithmetic shift right)

ror dst, 1 ; ror dst, cl ; rol dst, 1 ; rol dst, cl ; (rotate right and rotate left)
rcr dst, 1 ; rcr dst, cl ; rcl dst, 1 ; rcl dst, cl ; (rotate right and left, over carry flag)
mul op (unsigned ax=al × op or dx:ax=ax × op)
div op (unsigned al=ax/op, ah=reminder, or ax=dx:ax/op dx=reminder)
imul op (signed ax=al × op or dx:ax=ax × op)
idiv op (signed, execution is similar to div)

loop nearaddress (decrement cx, if not zero then go to nearaddress.)
jmp nearaddress (jump to near address)

test op1,op2 sets flags by op1 AND op2
cmp op1, op2 (compare operands for branch)
jxx shortaddress (jump for equal, above, below, greater-than, less-than, and flag conditions)
signed and unsigned: **je, jne**
signed: **jg, jng, jge, jnge, jl, jnl. jle, jnle**
unsigned: **ja, jna, jae, jnae, jb, jnb, jbe,jnbe**
on-flags **jz, jnz, jc, jnc, js, jns, jo, jno, jp, jpo**,

call procaddr (calls near subroutine procaddr)
ret [n] (removes n bytes from stack and returns from subroutine)
push rx - pop rx (push - pop16-bit reg. on stack.)
pushf - popf (push - pop flags onto stack)
xchg dst,src (swaps registers dst and src)

Some ASCII control characters

07h =<BEL> (bell) ; 08h =<BS> (backspace) ; 09h =<TAB>; 0Ah =<LF> Linefeed; 0Ch =<FF> formfeed ; 0Dh =<CR> Carriage-Return;

Printable ASCII Table:

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
2-		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3-	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4-	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5-	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	-
6-	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7-	p	q	r	s	t	u	v	w	x	y	z	{	|	}	→	←

8255 PPI

Mode-0 Control Byte:
(for PA, PCH, PB, PCL use 0:output, 1:input)

b7	b6	b5	b4	b3	b2	b1	b0
1	0	0	PA	PCH	0	PB	PCL

8251 USART

Mode Register format for asynchronous mode:

b7 b6	= { S2S1:	nr.of stop bits	**00:** invalid / **01:**1stop / **10:** 1.5stop / **11:** 2stop },
b5	= { EP:	parity type	**0:** odd / **1:** even },
b4	= { PEN:	parity enable	**0:** no-parity-bits / **1:** parity-bits-present },
b3 b2	= { L2L1:	nr.of data bits	**00:** 5-bit / **01:** 6-bit / **10:** 7-bit / **11:** 8-bit },
b1 b0	= { B2B1:	baud rate factor	**00:** sync-mode / **01:** /1 / **10:** /16 / **11:** /64 }

Control Register format for asynchronous mode:

b7	= { EH:	Enter hunt mode	**1:** enable / **0:** disable }
b6	= { IR:	Internal reset	**1:** resets the 8251A }
b5	= { RTS:	Request to send,	**1:** RTS-output-forced-to-low }
b4	= { ER:	Error Reset	**1:** reset error flags PE,OE,FE}
b3	= { SBRK:	Send break char	**1:** forces TxD low }
b2	= { RxE:	Receiver enable	**1:** enable, **0:** disable }
b1	= { DTR:	Data terminal ready	**1:** DTR-output-forced-to-low }
b0	= { TxE:	Transmitter enable	**1:** enable, **0:** disable }

Status Register format for asynchronous mode:

b7	= {DSR	**1:** DSR pin is active (low)}
b6	= {SY/BD	**1:** sync-or-break char detected}
b5	= {FE	**1:** Framing error detected}
b4	= {OE	**1:** Overrun error detected}
b3	= {PE	**1:** Parity error detected}
b2	= {TxE	**1:** Tx finished transmitting all data}
b1	= {RxRDY	**1:** Data-in buffer is full}
b0	= {TxRDY	**1:** Data-out buffer is empty}