Laboratory Manual for

EC-615 – Advanced Microprocessors

B. Tech.

SEM. VI (EC)

Department of Electronics & Communication
Faculty of Technology
Dharmsinh Desai University
Nadiad

TABLE OF CONTENTS

PART – I LABORATORY MANUAL

PART – II APPENDIX (DATASHEETS)

PART – III SESSIONAL PAPERS

P A R T I

LAB MANUAL

EXPERIMENT – 1

DEBUG Commands

OBJECTIVE:

- To study DEBUG commands and their use in debugging assembly language programs.

THEORY: Refer list of common DEBUG commands given below.

1) ASSEMBLE A [ADDRESS]:-

It is useful to give the starting offset address value in instruction pointer register. Then it will be create effective address for starting instruction.

EXAMPLE:-

```
-a 0100
0B0E:0100
```

Here, in example 0100 is the value which is stored in to instruction pointer (IP) register while 0B0E is code segment base then effective address will be 0B0E0+0100=0B1E0 of first instruction.

2) COMPARE C RANGE ADDRESS:-

It is useful for the comparison between the address range which is shown below:

EXAMPLE:-

```
-a 0100
0B0E:0100   mov ax, 1234
0B0E:0103   mov bx, 5678
0B0E:0106
-t=0100 2
AX=1234 BX=5678
-c 0100 0102 0103
0B0E:0100 B8       BB 0B0E:0103
0B0E:0101 34       78 0B0E:0104
0B0E:0102 12       56 0B0E:0105
```

Here, from example we conclude that compare instruction is used for comparing the two values stored in different locations. In example it compares the values which are at 0100 & 0103 onwards. In this command we should provide stating comparison address, range of the address & ending comparison address (-c 0100 0102 0103).

3) DUMP D [RANGE]:-

It is useful to show the value store at particular address. It also shows at right side the characters whose ASCII value stores in address.

EXAMPLE:-

```
-a 0100
```

0B0E:0100 mov ax, 1234
-d 0100 0102
0B0E:0100 B8 34 12 .4.

In given example we can say at 0B1E0H the opcode of move instruction is stored. While at 0B1E1H & 0B1E2H, 34 & 12 is respectively stored. At right side .4. is shown which shows the ASCII value of 4 is stored at address 0B1E1H.

If we write only,

-d 0100

Then it will take by default range of address from starting at IP=0100H.

4) ENTER E ADDRESS [LIST]:-

It is used to change the stored value at particular address. It is only used for changing the 8-bit. If we change opcode then instruction will be changed if code is valid. If code isn't valid, then it won't give any error. So, programmer should take care about it.

EXAMPLE:-

```
-a 0100
0B0E:0100   mov ax, 12
0B0E:0103
-r                          ; to show registers
Ax=0000                       ; all other registers aren't shown here only 4
                               ; Ex.
-t 0100                     ; execution of an instruction
AX=0012
-d 0100 0103
0B0E:0100 B8 12 00 4f
-e 0100
0B0E:0100 B8.05             ; 05 is opcode of ADD instruction which
                               ; replaces the Opcode of move instruction
                               ; (B8)
-t 0100                     ; execution of (add ax, 12)
AX=0024                      ; after execution AX=0012+0012
-e 0100
0B0E:0100 05.06             ; 06 isn't opcode of any instruction
-t 0100
AX=0024                      ; doesn't affect data in ax register
```

From, this example we conclude that if opcode will be change then totally instruction will be change. So, at run time we can change operation.

5) Go G [=ADDRESS] [ADDRESS]:-

It is used for executing no. of instructions at a time by providing the starting & ending address of the instructions. If we provide in between address then it will come out from debug (which is cerrent).

EXAMPLE:-

-a 0100
0B0E:0100 mov ax, 12
0B0E:0103 mov bx, 34
0B0E:0106
-g = 0100 0106 ; execution of two instructions
AX=0012 BX=0034 ; not shown all other regs 4 ex
-g=0100 0105 ; come out from current DEBUG
D :\> ; whatever C or D drive

From this example we conclude that if we provide 0100 t0 0106 address then it will execute more than one instruction (here 2), but if we provide 0100 to 0105 then it will comes out from current debug it means that which isn't valid address range.

6) MOVE M RANGE ADDRESS:-

It is used for copying the opcode & data from giving address range to starting address at where we want to copy it up to same range of address.

EXAMPLE:-

-a 0100
0B0E:0103
-d 0100 0105
0B0E:0100 B8 12 00 4F A0 32
-m 0100 0102 0103 ; move instruction
-d 0100 0105
0B0E:0100 B8 12 00 B8 12 00

From this example we conclude that at 0103 we have also same instruction because it is copied at that address by using move instruction.

7) NAME N [PATH NAME] [ARG LIST]:-

It is used for creating a file.

For creation of File:-

EXAMPLE:-

-n filename.com
-r bx
BX=0000
: 0000
-r cx
CX=0000
: byte size
-w
Writing…. No. of bytes

For opening a File:-

-n filename.com
-l ; loading a File

-u ; unassembling a File

8) PROCEED P [=ADDRESS] [NUMBER]:-

It is used for the execution of instructions by different way.

EXAMPLE:-

-p n ; where n is an integer no.

It will execute n no. of instructions from where it stops.

EXAMPLE:-

-p = starting address n

It will execute n no. of instructions from starting address which is given.

EXAMPLE:-

-p = starting address

It will execute only one instruction from starting address which is given.

9) QUIT Q:-

It is used for coming out from given current debug session.

10) REGISTER R [REGISTER]:-

It is used for showing the status of all registers as well as the flags.

EXAMPLE:-

-r

This command shows the status of different registers (AX, BX, CX, DX, SP, BP, SI, DI, SS, DS, CS, ES, IP) & as well as all flags (NV,UP,EI,PL,NZ,NA,PO,NC).

11) TRACE T [=ADDRESS] [VALUE]:-

It is used for executing instructions in different ways.

EXAMPLE:-

-t = starting address

It will start the execution of instructions from a given starting address in IP register.

EXAMPLE:-

-t = starting address n ; where n is an integer no.

It will execute n no. of instructions from a given starting address.

EXAMPLE:-

-t n ; where n is an integer no.

It will execute n no. of instructions from current address in IP register.

12) UNASSEMBLE U [RANGE]:-

If we have code & data stored at some address & we unassemble it then it will show instructions in an assembly.

EXAMPLE:-

-a 0100
0B0E:0100 mov ax, 12
0B0E:0103
-d 0100 0102

0B0E:0100 B8 12 00
-u 0100 0102 ; unassembled an instruction
0B0E:0100 B8 12 00 mov ax, 12

SAVING A PROGRAM FROM WITHIN DEBUG:

To use DEBUG to write a very small machine language program that you now want to save, follow these steps

- Request DEBUG program.
- Use A (Assemble) and E (Enter) commands to create a program.
- Name the program: N filename.com
- Tell debug the size of program in bytes using BX:CX pair.
- Write the revised program: W (enter).

For example:

C:\BC\MASM>debug

-a
0BFF:0100 mov ax,bx
0BFF:0102 mov cx,dx
0BFF:0104 add ax,cx
0BFF:0106 nop
0BFF:0107
-n first.com
(Size of the program here is from 0100 to 0106 i.e. 7 bytes. Put this size in BX:CX pair.)
-r bx
BX 0000
:0000
-r cx
CX 0000
:0007
-w
Writing 00007 bytes
-q
C:\BC\MASM>

LOADING A PROGRAM IN DEBUG:

To use DEBUG to load a machine language program that you have saved on the disk, follow these steps

- Request DEBUG program.
- Name the program: N filename.com
- Load the program using L command.

For example:

C:\BC\MASM>debug

-n first.com
-l

-u
0BFF:0100 mov ax,bx
0BFF:0102 mov cx,dx
0BFF:0104 add ax,cx
0BFF:0106 nop
0BFF:0107
-q

CONCLUSION:

EXERCISE:

1. Assemble simple assembly language program in DEBUG to add two numbers placed at memory locations 50H and 60H and put the result at 70H. Trace the program using DEBUG commands.

2. Write assembly language program for subtraction of two numbers (signed) and see the effect on CY and OV flags. The two numbers are : a) FFH, 1BH b) 64H, 32H
a) 2DH, 4BH b) E2H, CEH c) BAH, BAH

3. Explain the status of various flags as displayed in DEBUG.

4. Use commands **N** (Name) and **W** (Write) to save the program written above on the hard disk. Use **L** (Load) command to load the same program in debug and run it.

REFERENCES: 1. IBM PC assembly language programming
Author: Peter Abel

EXPERIMENT - 2

DOS Interrupts

OBJECTIVE:

- To study different DOS interrupt functions and use them to write user friendly programs.

THEORY:

Commonly used DOS interrupts for input from keyboard, display on screen are discussed below.

1. To read a single character from keyboard (function 01)

```
-a  100
0572: 0100     mov ah,01
 0572: 0102    int 21
0572: 0104     mov cx,1234
-t = 100
AX=0100    BX=0000    CX=0000
-p
a
AX=0161    BX=0000    CX=0000
-t
AX=0161    BX=0000    CX=1234
```

In this program ,character a is read from key-board. So it will give its equivalent ASCII value in AL register. Here INT 21h is inbuilt function to execute all the instructions of it proceed is used.

2. Display a single character on screen(function 02):

```
-a=0100
00A7=0100 mov ah,02
00A7=0102 mov dl,61
00A7=0106 int 21h
00A7=0109 mov cx,12
-t = 100
AX=0200    BX=0000    CX=0000    DX=0000
-t
AX=0200    BX=0000    CX=0000    DX=0061
-p
a
```

AX=0261 BX=0000 CX=0012 DX=0061

From the example we should provide function value in AH register. ASCII value of character in DL register.So we show a character whose ASCII value stored in DL register on as screen & ASCII value of it in AL register.

3. To read a single character without ECO(Don't see the character on screen)(function 07):

```
-a 0100
0B0E: 0100 mov ah,07
0B0E: 0100 int 21h
0B0E: 0100 mov cx,12
0B0E: 0100
-t=0100
AX=0700  BX=0000  CX=0000
-p
AX=0762  BX=0000  CX=0000
-t
AX=0762  BX=0000  CX=0012
```

In this case whatever character read from a keyboard that will not display on a screen but its ASCII value will be stored in to AL register.

4. To display a string on a screen(function 09):

We should first start a string at some address & at the end of the string '$' is used to show the ending of the string.then address of that string should be stored in to DX register.

```
-e 0200 "raj$"
-a 0100
0B0E: 0100 mov ah,09
0B0E: 0102 mov dx,0200
0B0E: 0105 int 21h
0B0E: 010A
-t=0100
AX=0900  BX=0000  CX=0000  DX=0000
-t
AX=0900  BX=0000  CX=0000  DX=0200
-p
raj
AX=0900  BX=0000  CX=0000  DX=0200
-t
AX=0900  BX=0000  CX=1234  DX=0200
```

Here string " raj$ " which is stored at 200 will be shown on the screen.

5. To read a string from key-board(function 0A):

```
-e    0200    4
-a    0100
0B0E: 0100 mov ah,0A
0B0E: 0102 mov dx,0200
0B0E: 0105 int 21h
0B0E: 0107 mov cx,12
0B0E: 010A
-t
AX=0A00     BX=0000     CX=0000     DX=0000
-t
AX=0A00     BX=0000     CX=0000     DX=0200
-p
raj
AX=0A00     BX=0000     CX=0000     DX=0200
-t
AX=0A00     BX=0000     CX=0012     DX=0200
-d    200   206
0B0E:200   04 03 72 61 6A 0d 3c ...raj..
```

From this example we can only read a string up to (define size -1) character. String will be start after two type where we define size of it. so, total three bytes (1^{st} = total size, 2^{nd} =actual size & 3^{rd} =for enter). So, total three will be wasted. If we want to display the string then we must put '$' at the end of it while it reads..

CONCLUSION:

EXERCISE:

1. Write an assembly language program in DEBUG to take an input string from the keyboard and display the same string on the screen using INT 21H functions. Save the program using DEBUG commands.

2. Write an assembly language program to take two 2 digit numbers from the keyboard and display the result of addition of the numbers on the screen.

3. Write an assembly language program to create a file; write some data in that and save it using file handles. Also open the same file; read it, modify the data written in it and save it.

4. What are DOS – BIOS interrupts?

5. What is the difference between the DOS interrupt and BIOS interrupt?

REFERENCES:

1. IBM PC Assembly Language Programming
Author: Peter Abel

EXPERIMENT - 3

Program Organization And Development

OBJECTIVE:

(a) To study various Assembler directives and use them to write simple programs.

THEORY: Refer list of common assembler directives given below

- **ASSUME:**
  The assume directive is used to tell the assembler the name of the logical statement it should use for a specified segment. The statement ASSUME CS:CODE, for example tells the assembler that the instruction for a program in a logical statement named code.

- **DB:**
  The DB directive is used to declare a byte type variable or set aside one or more storage location of type byte in memory. The statement A DB 5H, for example, tells the assembler to reserve one byte of memory for the variable A and put the value 5 in that memory space.

- **DW:**
  The DW directive is used to declare 2 byte type variable or set aside two storage location of type word in memory. The statement A DW 2345H, for example, tells the assembler to reserve two bytes of memory for the variable A and put the value 45 in first memory location and then 23 in the next address location that memory space.

- **DD:**
  The DW directive is used to declare a variable of type double word or set aside storage location in memory which can be accessed as type double word. The statement A DD 11225566H, for example, tells the assembler to reserve 4 byte of memory for the variable A and put them in that memory space such that lower word 5566h,will be put at a lower memory address than the high word 1122h.

- **END:**
- The end directive is put at the last statement of the program to tell the assembler that this is the end of the program module. The assembler will ignore any statement written after this END directive. A carriage return is required after this statement.

- **ENDP:**
  This directive is used along with the name of the procedure to indicate the end of the procedure to the assembler. This directive is used with PROC directive.

- **EQU:**
  It is used to give name to some name or symbol. Each time the assembler find the given name in the program, it will replace the name or the symbol with the value or that you equated with that name.

- **ENDS:**
  This directive is used with the name of the segment to indicate he end of the logical statement. This directive is used along with the SEGMENT directive.

- **EXTERN:**
  This directive is used to tell the assembler that the name or the labels following the directive are in some other assembly module. For example, if you want to call a procedure which is in the program module assembled at different time from that which contains the CALL instruction, you must tell the assembler that procedure is external. The assembler then put information in the object file so that linker can connect the two modules together.

- **GLOBAL:**
  This directive can be used in place of a PUBLIC or in place of an EXTERN directive. For a name or symbol defined in the current assembly module. For a name or symbol defined in the current assembly module, the GLOBLE directive is used to make the symbol available for the other
  modules.

- **LABLE:**
  As the assembler assembles a section of data declaration or instruction statements, it uses a location counter to keep the track record of how many bytes it is from the start of the segment at a time. The LABEL directive is used to give a name to the current value in the location counter. If label is going to be used with JUMP or CALL instruction then the label must be specified as type near or type far.

- **OFFSET:**
  It is an operator which tells the assembler to determine offset or displacement of a named data item or procedure from the start of the segment that contains it.

- **ORG:**
  As the assembler assembles a section of data declaration or instruction statements, it uses a location counter to keep the track record of how many bytes it is from the start of the segment at a time. The location counter is automatically set to 0000h when the assembler starts reading a statement. The ORG directive allows you to set a location counter to a desired value anywhere in the program.

- **PROC**:

This directive is used to identify the start of a procedure. It follows a name you give the procedure. After the PROC directive, the term near or the term far is used to specify the type of the procedure.

- **PTR:**
  The ptr operator is used to assign a specific type to a variable or to a label.It is necessary to do this in any instruction where the type of the operand is not clear.

- **SEGEMENT:**
  This directive is used to indicate the start of a logical statement .Preceding the segment directive is the name you want to give the segment.

CONCLUSION:

EXERCISE:

1. Write an assembly language program that multiply two 8-bit numbers X and Y defined in data segment and places the result at memory location Z in extra segment. Assemble and run the program in DEBUG. Show the Memory Map for all the segments used in the program.

2. Write an assembly language program to find the largest number from a given array in data segment.

3. Write an assembly language program to take two 2 digit numbers input from the keyboard and display the results of addition, subtraction, multiplication and division of the numbers on the screen.

4. Explain the use of various program development tools like: Editor, Assembler, Linker, Compiler, Locator, Debugger and Emulator.

5. Why is it necessary to initialize DS and ES registers in the program even after using ASSUME directive? Why CS need not to be initialized?

6. How 8086 instruction set is made compatible with higher level language? Explain with respect to addressing modes available.

REFERENCES:

1. The Intel Microprocessors
Author: Barry B. Brey

2. IBM PC Assembly language programming
Author: Peter Abel

(b): To study program organization and development process.

To develop any assembly language program using “TASM” follow the procedure given below.

- Start the command prompt on your computer.
- Go to “TASM” directory using ***cd*** command.
- Write command ***edit filename.asm*** to make the source file of your program. This command will start the dos editor.
- Write the source code of your program in that editor and save the file.
- Assemble the program using command ***tasm/l filename.asm.*** If there are no errors in your program, then the object file will be generated by the assembler. If there is any error then you can detect it from list file and assemble it again after correcting the error.
- After successfully assembling the file write command ***tlink filename.obj.*** This command will create an executable file and a map file from the object file.
- Check the location of code segment from the map file.
- Write the command ***debug filename.exe*** debug or simulate the program. Trace the programme from the starting address of the code segment specified in the map file.

Here for reference a generated list file is given.

1) First column gives the serial number of the lines of list file.
2) Second column gives the address of the instructions.
3) Third column gives the op-code of the program
4) Fourth column gives the description of the instructions.

Turbo Assembler Version 3.0 01/12/08 12:54:26 Page 1
add_00.asm

```
1   0000                    data segment
2   0000  12 23 44 55 66     x db 12h,23h,44h,55h,66h
3   0005  05*(??)            y db 5 dup(?)
4   000A                    ends
5
6   0000                    code segment
7
```

```
 8                                      assume cs:code,ds:data
 9   0000  B8 0000s                          mov ax,data
10   0003  8E D8                       mov ds,ax
11   0005  BE 0000r                          mov si,offset x
12   0008  BF 0005r                          mov di,offset y
13   000B  B9 0005                           mov cx,05h
14
15   000E  8A 44 04                     back: mov al,[si]+4
16   0011  88 05                       mov [di],al
17   0013  4E                          dec si
18   0014  47                          inc di
19   0015  E2 F7                       loop  back
20
21   0017                              ends
22                                     end
```

Symbol Table

Symbol Name	Type	Value
??DATE	Text	"01/12/08"
??FILENAME	Text	"add_00 "
??TIME	Text	"12:54:26"
??VERSION	Number	0300
@CPU	Text	0101H
@CURSEG	Text	CODE
@FILENAME	Text	ADD_00
@WORDSIZE	Text	2
BACK	Near	CODE:000E
X	Byte	DATA:0000
Y	Byte	DATA:0005

Groups & Segments	Bit Size	Align	Combine	Class
CODE	16	0017	Para	none
DATA	16	000A	Para	none

EXPERIMENT - 4

Array Processing

OBJECTIVE:

- To perform various operations on array in data as well as extra segment.
- To study the string instructions i) Lods ii) Stos iii) Movs iv) Cmps v) Scas

SAMPLE PROGRAM:-

PROGRAM-1:- Transfer data from data segment to extra segment.

```
DATA SEGMENT
        X DB 1 , 2, 3, 4, 5                 ;TO INITIALIZE ARRAY
                                            ;OF 5 ELEMENT HAVING
ENDS
EXTRA SEGMENT
        Y DB 5 DUP(?)
ENDS
CODE SEGMENT
        ASSUME CS:CODE, DS:DATA, ES:EXTRA;
        MOV AX, EXTRA;
        MOV DS, AX;
        MOV AX, DATA;
        MOV ES, AX;
        MOV SI, OFFSET X;                   ;ASSIGNING THE BASE
                                            ;ADDRESS OF STRING TO
        MOV DI, OFFSET Y;
        MOV CX, 05;
BACK:
        MOV AL, [SI];                       ;COPY DATA FROM ONE
                                            ; TO OTHER LOCATION
        MOV ES: [DI], AL;
        INC SI;
        INC DI;
        LOOP BACK;
ENDS
END
```

PROGRAM-2:- Transfer data from extra segment to data segment.

```
DATA SEGMENT
        X DB 1 , 2, 3, 4, 5                 ;TO INITIALIZE ARRAY
                                            ;OF 5 ELEMENT
```

```
ENDS
EXTRA SEGMENT
        Y DB 5 DUP(?)
ENDS
CODE SEGMENT
        ASSUME CS:CODE, DS:DATA, ES:EXTRA;
        MOV AX, EXTRA;
        MOV DS, AX;
        MOV AX, DATA;
        MOV ES, AX;
        MOV SI, OFFSET X;                       ;ASSIGNING THE BASE
                                                ;ADDRESS OF STRING
        MOV DI, OFFSET Y;
        MOV CX, 05;
BACK:
        MOV AL, ES:[SI];                        ;COPY DATA FROM ONE
                                                ; TO OTHER LOCATION
        MOV [DI], AL;
        INC SI;
        INC DI;
        LOOP BACK;
ENDS
END
```

PROGRAM-3

;Here the reference of the comparison of two strings are taken to understand the various string ;instuctions

```
DATA SEGMENT

        X DB "ABC"                      ;STORAGE OF  DEFAULT
        STRING
        A DB "STRING ARE SAME$"                 ;MESSAGES TO BE DISPLAYED
        B DB "STRINGS ARE DIFFERENT$"
ENDS

EXTRA SEGMENT
        Y DB "ABC"                       ;STORAGE OF ANOTHER
                                        ; STRING TO BE COMPARED
        M DB ?
ENDS

CODE SEGMENT
```

```
        ASSUME CS:CODE; DS:DATA; ES:EXTRA;
        MOV AX,EXTRA
        MOV ES,AX
        MOV AX,DATA
        MOV DS,AX
        MOV SI,OFFSET X              ;MOVE ADDRESS OF 1ST STRING IN SI
        MOV DI,OFFSET Y              ;MOVE ADDRESS OF SECOND STRING TO DI
        MOV CL,OFFSET A-OFFSET X     ;FIND STRING LENGTH OF 1ST STRING
        MOV BL,CL                    ;STORE AT SOME OTHER PLACE
        MOV CL,OFFSET M-OFFSET Y     ;FIND LENGTH OF 2ND STRING
        CMP BL,CL                    ;COMPARE TWO STRING LENGTH
        JNZ OVER2
AGAIN:  REPE CMPSB

                JZ OVER1
                JMP OVER2
OVER1:
                MOV AH,09
                MOV DX,OFFSET A      ;DISPLAY MESSAGE THAT
                                     ; STRING ARE SAME
                INT 21H
                JMP LAST             ;JUMP TO END OF CODE
OVER2:          MOV AH,09
                MOV DX,OFFSET B      ;DISPLAY MESSAGE THAT
                                     ;ARE NOT SAME
                INT 2
 LAST:              ENDS
                END                  ;END OF CODE
```

PROGRAM-4

; Here the reference of password generation and verification are taken to understand the ; various string instuction

```
DATA SEGMENT

        X DB "ENTER PASSWORD$"       ;TO DISPLAY MESSAGE AT THE  START
        PASSWORD DB 30 DUP(?)        ;TO ALLOCATE MEMORY FOR USER'S
                                     ; PASSWORD
        STR1 DB "CORRECT PASSWORD$"
        STR2 DB "WRONG PASSWORD$"
```

```
ENDS

EXTRA SEGMENT

        STORE1 DB "HELLO"               ;STORED PASSWORD
        TEMP DB ?

ENDS

CODE SEGMENT

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

        MOV AX,DATA
        MOV DS,AX
        MOV AX,EXTRA
        MOV ES,AX

        MOV AH,09H                      ;TO PRINT APPROPRIATE
                                        ;INSTRUCTION TO GUIDE THE USER
        MOV DX,OFFSET X
        INT 21H
        MOV  BX,OFFSET PASSWORD         ;MOVE STRATING ADDRESS OF
                                        ; USER'S PASSWORD
        BACK:
                MOV AH,08H              ;TO GET THE PASSWORD
                                        ; CHARACTER BY CHARACTER
                INT 21H                 ;FROM USER AND COMPARE
                                        ; ALSO CHARACTER BY
                                        ; CHARACTER
                MOV TEMP,AL
                CMP TEMP,'$'
                JZ CHECK
                MOV [BX],AL             ;TO POINT TO NEXT STRATING ADDRESS
                INC BX
                MOV DL,'*'              ;TO DISPLAY '*'ON THE
                                        ; SCREEN FOR EACH
                                        ; CHARCTER
                MOV AH,02H
                INT 21H
                JMP BACK
        CHECK:
                 MOV CX,OFFSET TEMP-OFFSET STORE1
```

```
                                ;CALCULATE LENGTH OF
                                ; DEFAULT PASSWORD
        MOV SI,OFFSET PASSWORD
        MOV DI,OFFSET STORE1
        REPE CMPSB              ;TO COMPARE CHARCTER BY CHARACTER
        JNZ OVER2
OVER1:
        MOV AH,09
        MOV DX,OFFSET STR1      ;TO DISPLAY MESSAGE THAT
                                ; STRINGS ARE SAME
        INT 21H
        JMP LAST
OVER2:
        MOV AH,09
        MOVDX,OFFSET STR2       ;TO DISPLAY MESSAGE THAT
                                ; STRINGS ARE DIFFERENT
        INT 21H
LAST:
         ENDS
        END                     ;END OF CODE
```

CONCLUSION:

EXERCISE:

1. Modify above program to transfer array from i) one data segment location to other data segment location.
2. Modify above program to reverse array in i) same array ii) other array
3. What is segment override prefix?
4. What are the conditions when segment can not be over ridden?
5. Define a string in the data segment. For that
   - Calculate the length of the string
   - Move the string from one place to the other
   - Reverse the string
   - Calculate number of character ‘a’ in the string
   - Define another string in the data segment and concatenate both
6. Define a password in the data segment. Read the input string as a password from the keyboard. While inputting the password, it should not be displayed on the monitor, instead ‘*’ should be displayed for each character.Also if backspace key is pressed it should replace previous character with new character and should function as normal delete key operation
7. How the opcode for the prefixes REP, REPE and REPNE are generated?
8. Explain with reference to the string instructions, how 8086 instruction set is compatible with the higher level programming?

EXPERIMENT – 5

INTRODUCTION TO KEIL µVISION 4

OBJECTIVES:

To study the environment of Keil µVision 4.

THEORY:

The µVision4 IDE is a Windows-based software development platform that combines a robust editor, project manager, and makes facility. µVision4 integrates all tools including the C compiler, macro assembler, linker/locator, and HEX file generator. µVision4 helps expedite the development process of your embedded applications by providing the following:

- Full-featured source code editor
- Device database for configuring the development tool setting
- Project manager for creating and maintaining your projects
- Integrated make facility for assembling, compiling, and linking your embedded applications,
- Dialogs for all development tool settings,
- True integrated source-level Debugger with high-speed CPU and peripheral simulator.
- Advanced GDI interface for software debugging in the target hardware and for connection to KeilULINK
- Flash programming utility for downloading the application program into Flash ROM,
- Links to development tools manuals, device datasheets & user's guides.

The µVision4 IDE offers numerous features and advantages that help you quickly and successfully develop embedded applications. They are easy to use and are guaranteed to help you achieve your design goals.
The µVision4 IDE and Debugger is the central part of the Keil development tool chain. µVision4 offers a Build Mode and a Debug Mode.
In the µVision4 Build Mode you maintain the project files and generate the application.
In the µVision4 Debug Mode you verify your program either with a powerful CPU and peripheral simulator or with the Keil ULINK USB-JTAG Adapter (or other AGDI drivers) that connect the debugger to the target system. The ULINK allows you also to download your application into Flash ROM of your target system. The figure below shows the Keil µVision4.

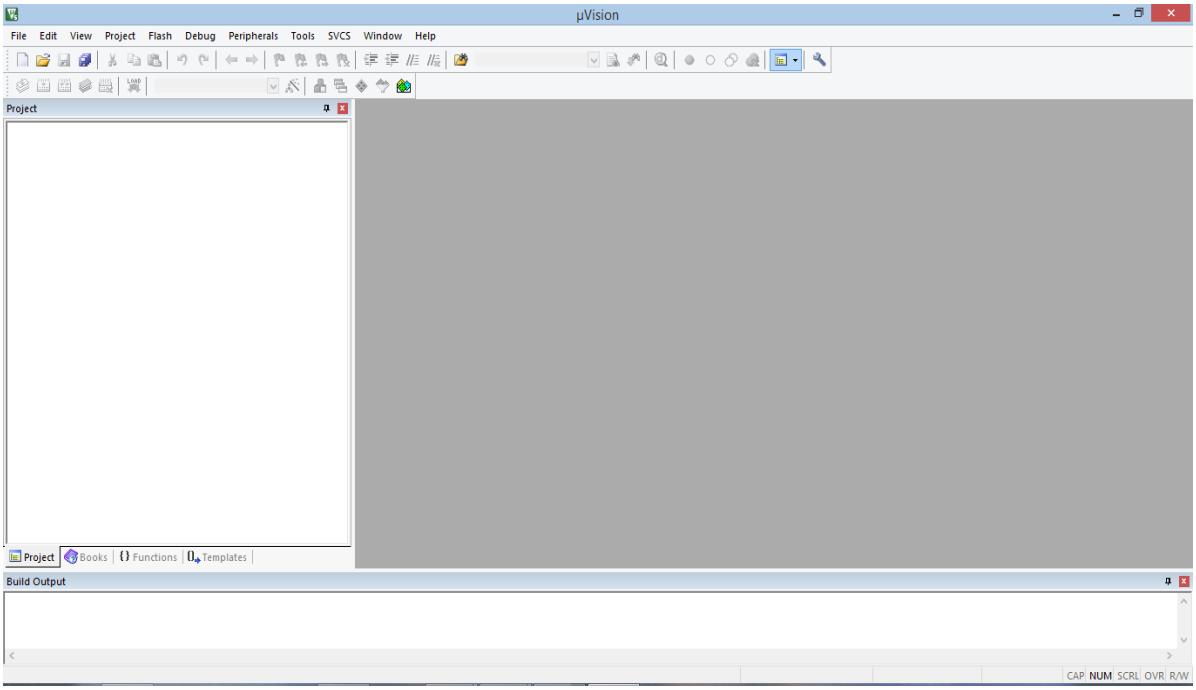


Fig.1.1 Menu Commands, Toolbars, and Shortcuts

Menu Commands, Toolbars, and Shortcuts

The menu bar provides you with menus for editor operations, project maintenance, development tool option settings, program debugging, external tool control, window selection and manipulation, and on- line help.

The toolbar buttons allow you to rapidly execute µVision4 commands. A Status Bar provides editor and debugger information. The various toolbars and the status bar can be enabled or disabled from the View Menu commands.

The following sections list the µVision4 commands that can be reached by menu commands, toolbar buttons, and keyboard shortcuts. The µVision4 commands are grouped mainly based on the appearance in the menu bar:

- File Menu and File Commands
- Edit Menu and Editor Commands
- Outlining Menu
- Advanced Menu
- Selecting Text Commands
- View Menu
- Project Menu and Project Commands
- Debug Menu and Debug Commands
- Flash Menu
- Peripherals Menu
- Tools Menu
- SVCS Menu
- Window Menu
- Help Menu

Creating Applications

This part describes the Build Mode of µVision4 and is grouped into the following sections:

- Create a Project: explains the steps required to setup a simple application and to generate HEX output.
- Project Target and File Groups: shows how to create application variants and organized the files that belong to a project.
- Tips and Tricks: provides information about the advanced features of the µVision4 Project Manager.

Create Project File Folder and Specify Project Name

To create a new project file select from the µVision4 menu Project – New – µVision Project. This opens a standard Windows dialog that asks you for the new project file name. You should you use a separate folder for each project. You can simply use the icon Create New Folder in this dialog to get a new empty folder.

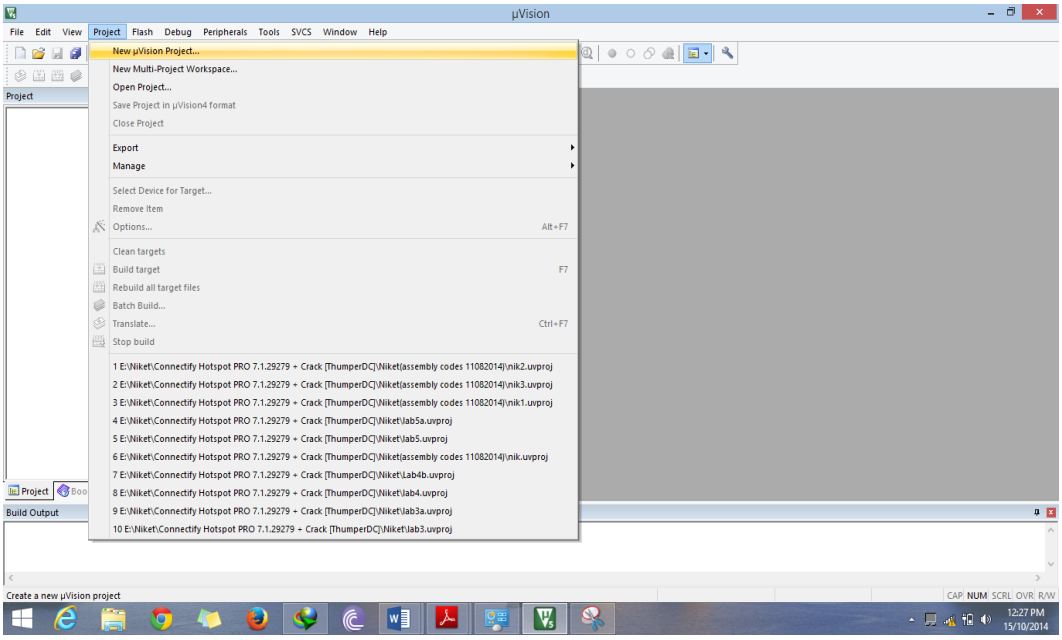


Fig.1.2 Project Menu

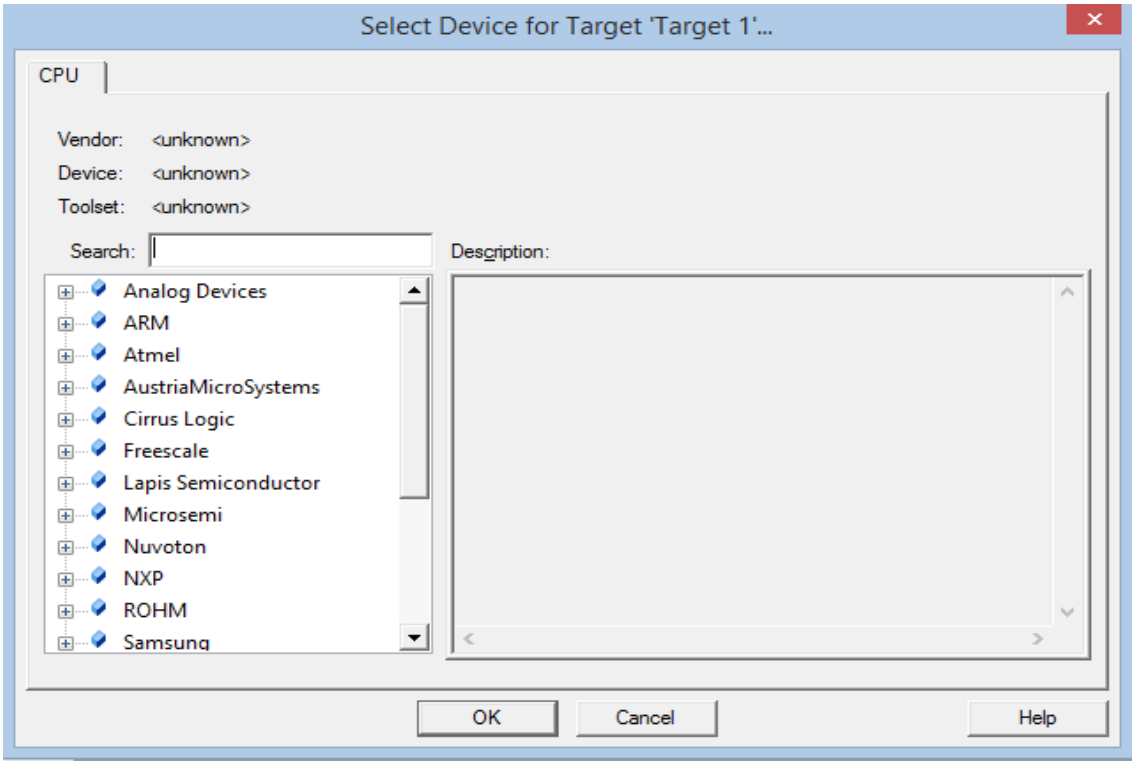


Fig.1.3 Vendor selection window

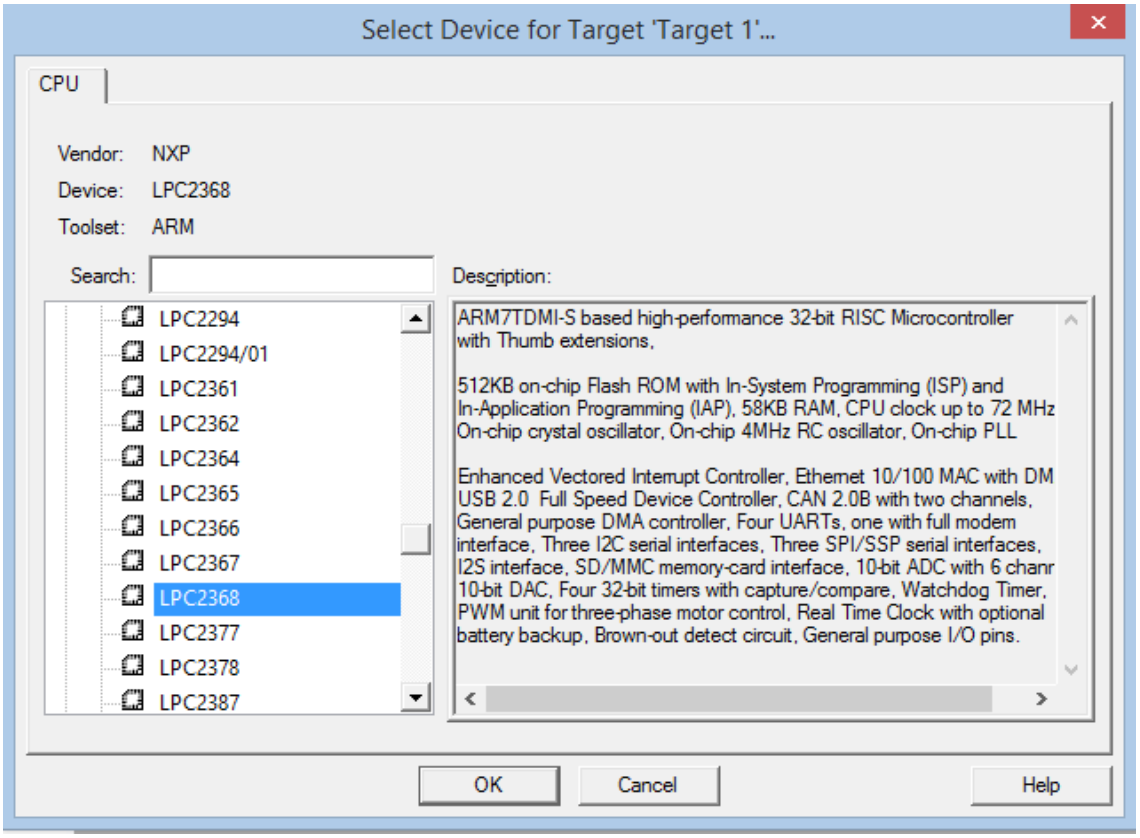


Fig.1.4 Description of selected device

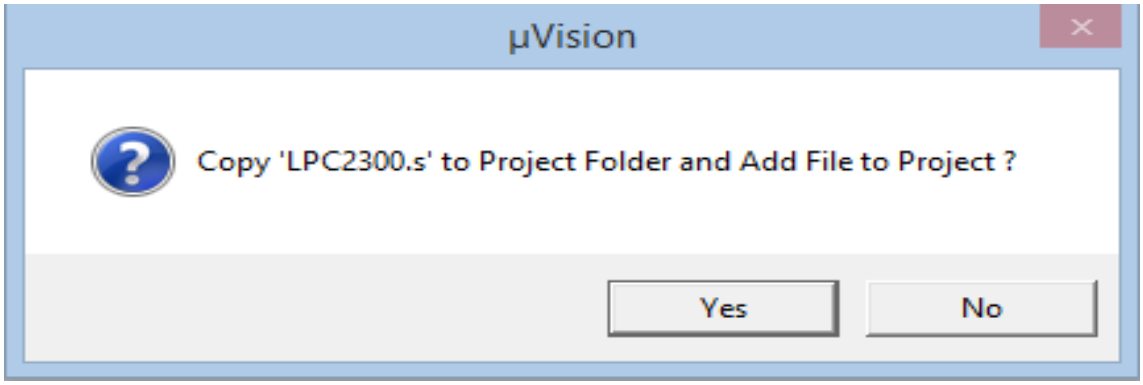


Fig.1.5 Copy LPC233.S to a project folder

Select this folder and enter the file name for the new project, i.e. Project1. µVision4 creates a new project file with the name PROJECT1.UV2 which contains a default target and file group name. You can see these names in the Project Workspace – Files.

Copy and Add the CPU Startup Code
An embedded program requires CPU initialization code that needs to match the configuration of your hardware design. This Startup Code depends also on the tool chain that you are using. Since you might need to modify that file to match your target hardware, the file should be copied to your project folder. For most devices, µVision4 asks you to copy the CPU specific Startup Code to your project. This is required on almost all projects (exceptions are library projects and add-on projects). The Startup Code performs configuration of the microcontroller device and initialization of the compiler run-time system.

Create New Source Files
You may create a new source file with the menu option File – New. This opens an empty editor window where you can enter your source code. µVision4 enables the C color syntax highlighting when you save your file with the dialog File – Save As… under a filename with the extension *.C. We are saving our example file under the name MAIN.C.

Add Source Files to Project
Once you have created your source file you can add this file to your project. µVision4 offers several ways to add source files to a project. For example, you can select the file group in the Project Workspace – Files page and click with the right mouse key to open a local menu. The option Add Files opens the standard files dialog. Select the file MAIN.C you have just created.

Set Tool Options for Target
µVision4 lets you set options for your target hardware. The dialog Options for Target opens via the toolbar icon or via the Project - Options for Target menu item. In the Target tab you specify all relevant parameters of your target hardware and the on-chip components of the device you have selected. The following dialog shows the settings for our example.

Build Project
Typical, the tool settings under Options – Target are all you need to start a new application. You may translate all source files and link the application with a click on the Build Target toolbar icon. When you build an application with syntax errors, µVision4 will display errors and warning messages in the Output Window – Build page. A double click on a message line opens the source file on the correct location in a µVision4 editor window.

The next steps are:

Test Programs with the µVision4 Debugger. The µVision4 Debugger offers two operating modes: simulator that allows you to verify your application on your PC, or Target Debugging with an Evaluation Board or your hardware platform

Program your application into Flash ROM. µVision4 integrates command-line driven Flash Utilities or can use the ULINK USB-JTAG Adapter for Flash programming. You may need to create a HEX file to use Flash programming utilities.

Create HEX File
Once you have successfully generated your application you can start debugging. After you have tested your application, it is required to create an Intel HEX file to download the software into an EPROM programmer or simulator. µVision4 creates HEX files with each build process when Create HEX file under Options for Target – Output is enabled. The FLASH Fill Byte, Start and End values direct the OH166 utility to generate a sorted HEX files; sorted files are required for some Flash programming utilities.

Test Programs with the µVision4 Debugger
This chapter describes the Debug Mode of µVision4 and shows you how to use the user interface to test a sample program. Also discussed are simulation mode and the different options available for program debugging. You can use µVision4 Debugger to test the applications you develop. The µVision4 Debugger offers two operating modes that are selected in the Options for Target – Debug dialog. Use Simulator configures the µVision4 Debugger as software-only product that simulates most features of a microcontroller without actually having target hardware. You can test and debug your embedded application before the hardware is ready. µVision4 simulates a wide variety of peripherals including the serial port, external I/O, and timers. The peripheral set is selected when you select a CPU from the device database for your target.

Debug Windows and Dialogs

- During Debug Mode µVision4 offers additional Debug Windows and Dialogs that are summarized below
- The Breakpoint Dialog allows you to define stop conditions for program execution.
- The Code Coverage Window provides execution statistic information of execute and not executed program parts.
- The CPU Registers may be reviewed and modified in the Regs page of the Project Workspace window.
- The Disassembly Window allows program testing at the level of assembly instructions.
- The Logic Analyzer provides a graphical display for value changes of peripheral registers and variables.
- The Memory Window may be used to review and modify memory content.
- The Serial Window displays the UART communication with the application program.
- The Symbol Window shows debug symbol information of the application program.
- The Toolbox provides configurable buttons for debug command and debug function execution.
- The Watch Window lets you view and modify program variables and lists the current function call nesting.

Flash Programming
µVision4 integrates Flash Programming Utilities in the project environment. All configurations are saved in context with your current project. You may use external command-line driven utilities (usually provided by the chip vendor) or the Keil ULINK USB-JTAG Adapter. The Flash Programming Utilities are configured under Project - Options - Utilities. Flash Programming may be started from the Flash Menu or before starting the µVision4 Debugger when you enable Project - Options - Utilities - Update Target before Debugging.

EXPERIMENT – 6

PROGRAM DEVELOPMENT PROCESS

OBJECTIVE:

Write a program in 'C' that adds and multiply two numbers.

THEORY:

PROGRAM DEVELOPMENT PROCESS

First, create a new project named first with extension ".Uv2".We will select the device from NXP (founded by Phillips) family. The board available with us is LPC 2368. So we will select every time this hardware. After giving the name adds the startup code of LPC 2300.S to the project file. After that the source codes save as text1.C. Add this file to the Source Group 1. Build the project and Debug the project in simulator by selecting the option for target 1. You can see the contents of user registers while running the program step by step.

SAMPLE PROGRAM:

```
#include<stdio.h>
int main()
{
        int a=1,b=2;
        int c;
        c=a+b;
}
```

OUTPUT:

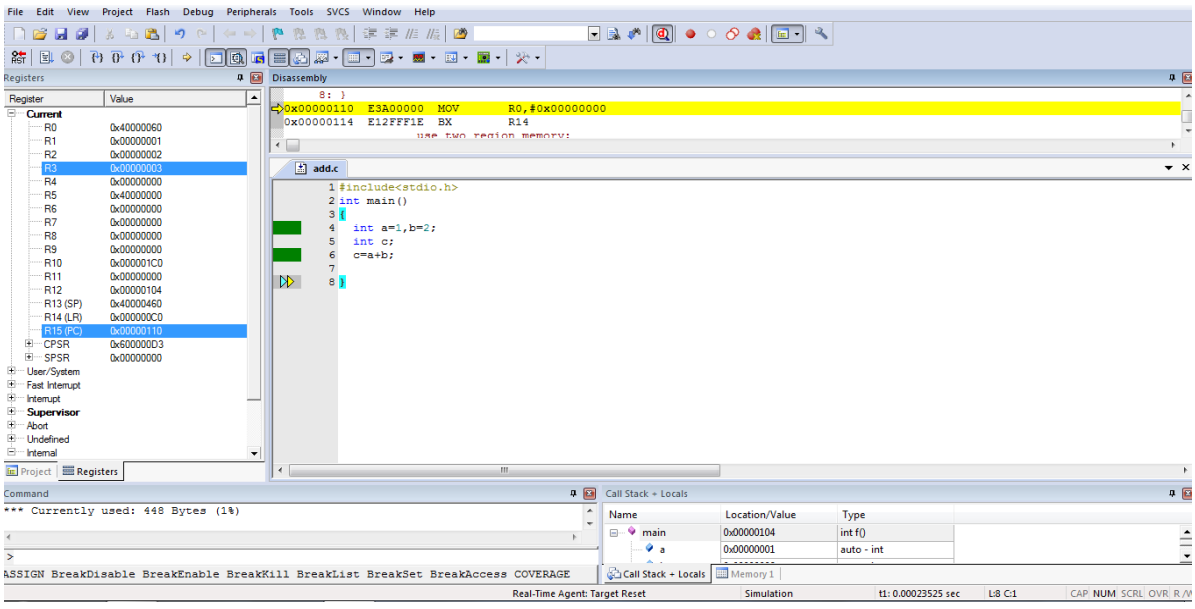


Fig. 2.1 Sample Program Execution

SAMPLE PROGRAM:

```
#include<stdio.h>
```

```
int main( )
{
        int a=2,b=2;
        int c;
        c=a*b;
 }
```

OUTPUT:

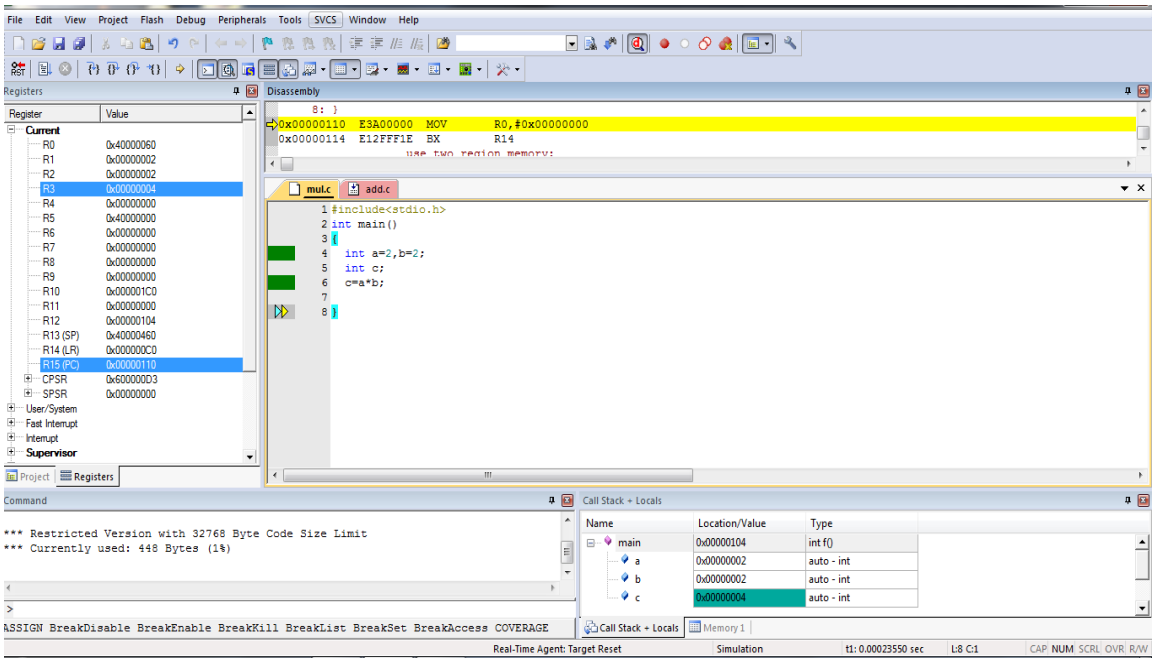


Fig.2.2 Sample Program Output

MODIFICATION:

(1) Write a program to perform subtraction and division on two numbers.

EXERCISE:

(1) What is significance of disassembly window?
(2) What is significance of memory window? How to open it and modify it?
(3) What information is contained in project workspace?

EXPERIMENT – 7

GENERAL PURPOSE I/O

OBJECTIVE:

To do programming of General Purpose Input Output Ports

THEORY:

Special Function Registers for GPIO:
FIOXDIR, FIOXMASK, FIOXSET, FIOXCLR and FIOPIN

Description - Special Function Registers for GPIO:

Table 107. GPIO register map (local bus accessible registers - enhanced GPIO features)

Generic Name	Description	Access	Reset value[1]	PORTn Register Address & Name
FIODIR	Fast GPIO Port Direction control register. This register individually controls the direction of each port pin.	R/W	0x0	FIO0DIR - 0x3FFF C000 FIO1DIR - 0x3FFF C020 FIO2DIR - 0x3FFF C040 FIO2DIR - 0x3FFF C060 FIO2DIR - 0x3FFF C080
FIOMASK	Fast Mask register for port. Writes, sets, clears, and reads to port (done via writes to FIOPIN, FIOSET, and FIOCLR, and reads of FIOPIN) alter or return only the bits enabled by zeros in this register.	R/W	0x0	FIO0MASK - 0x3FFF C010 FIO1MASK - 0x3FFF C030 FIO2MASK - 0x3FFF C050 FIO3MASK - 0x3FFF C070 FIO4MASK - 0x3FFF C090
FIOPIN	Fast Port Pin value register using FIOMASK. The current state of digital port pins can be read from this register, regardless of pin direction or alternate function selection (as long as pins are not configured as an input to ADC). The value read is masked by ANDing with inverted FIOMASK. Writing to this register places corresponding values in all bits enabled by zeros in FIOMASK. **Important:** if a FIOPIN register is read, its bit(s) masked with 1 in the FIOMASK register will be set to 0 regardless of the physical pin state.	R/W	0x0	FIO0PIN - 0x3FFF C014 FIO1PIN - 0x3FFF C034 FIO2PIN - 0x3FFF C054 FIO3PIN - 0x3FFF C074 FIO4PIN - 0x3FFF C094
FIOSET	Fast Port Output Set register using FIOMASK. This register controls the state of output pins. Writing 1s produces highs at the corresponding port pins. Writing 0s has no effect. Reading this register returns the current contents of the port output register. Only bits enabled by 0 in FIOMASK can be altered.	R/W	0x0	FIO0SET - 0x3FFF C018 FIO1SET - 0x3FFF C038 FIO2SET - 0x3FFF C058 FIO3SET - 0x3FFF C078 FIO4SET - 0x3FFF C098
FIOCLR	Fast Port Output Clear register using FIOMASK0. This register controls the state of output pins. Writing 1s produces lows at the corresponding port pins. Writing 0s has no effect. Only bits enabled by 0 in FIOMASK0 can be altered.	WO	0x0	FIO0CLR - 0x3FFF C01C FIO1CLR - 0x3FFF C03C FIO2CLR - 0x3FFF C05C FIO3CLR - 0x3FFF C07C FIO4CLR - 0x3FFF C09C

Fig.3.1 Special Function Registers Description

Features of Digital GPIO Ports: GPIO PORT0 and PORT1 are ports accessible via either the group of registers providing enhanced features and accelerated port access or the legacy group of registers. PORT2/3/4 is accessed as fast ports only.

Accelerated GPIO Functions:

- GPIO registers are relocated to the ARM local bus so that the fastest possible I/O timing can be achieved Mask registers allow treating sets of port bits as a group, leaving other bits unchanged All GPIO registers are byte and half-word addressable Entire port value can be written in one instruction
- Bit-level set and clear registers allow a single instruction set or clear of any number of bits in one port
- Direction control of individual bits
- All I/O default to inputs after reset
- Backward compatibility with other earlier devices is maintained with legacy registers appearing at the original addresses on the APB bus

SAMPLE PROGRAM: Program to generate continuous ON-OFF LED flashing pattern on a hardware board LPC 2368.

```
#include <LPC23xx.H>
void LED_init (void)                        // Function that initializes LEDs
{
        PINSEL10 = 1;                   // Disable ETM interface, enable LEDs
        FIO2DIR  = 0x000000FF;           // P2.0..7 defined as Outputs
        FIO2MASK = 0xFFFFFF00;          // enable all pins for modification for port 2
}

void LED_on (unsigned int num)          // Function that turns on requested LED
{
        FIO2SET =(1 <<num);
}

void LED_off (unsigned int num)             //Function that turns off requested LED
{
        FIO2CLR = (1 <<num);
}

void delay (void)                           // delay
 {
        int i;
        for ( i=0; i<7099999; i++) { }
 }

void main (void)
 {
                int k;
                LED_init ();                        // Initialize the port for output
                delay ();
                while(1)
                {
                        for ( k=0; k<8; k=k+1)
                        {
                                LED_on (k);     // make k th LED on
                                delay ();
                                LED_off (k);    // make k th LED off
                                delay ();
                        }
                }
```

}

Flashing the Program on a Hardware Kit:

After writing any program, it can be flashed into ARM controller kit by following the steps below:
Open Project → Options for target 'target 1' → Debug → Select proper debugger as shown below.

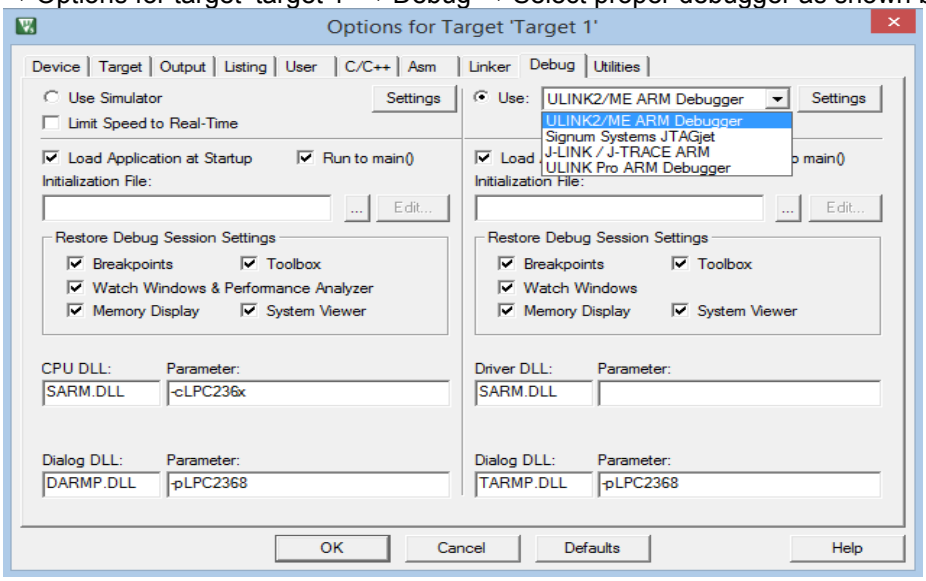


Fig.3.2 Selection of ARM debugger

Open Flash → Download to download the program to the kit

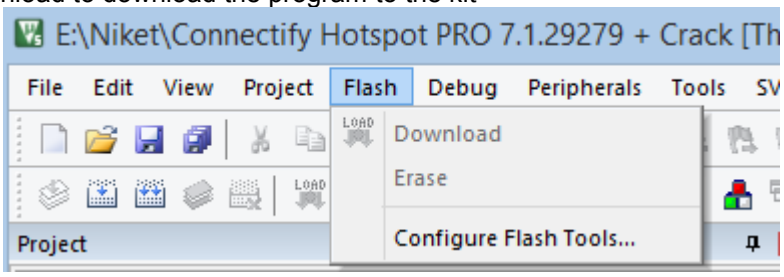
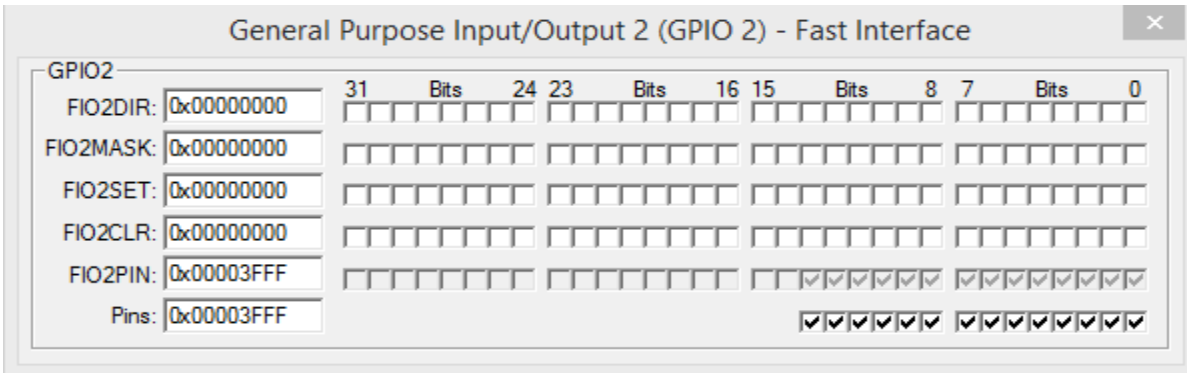
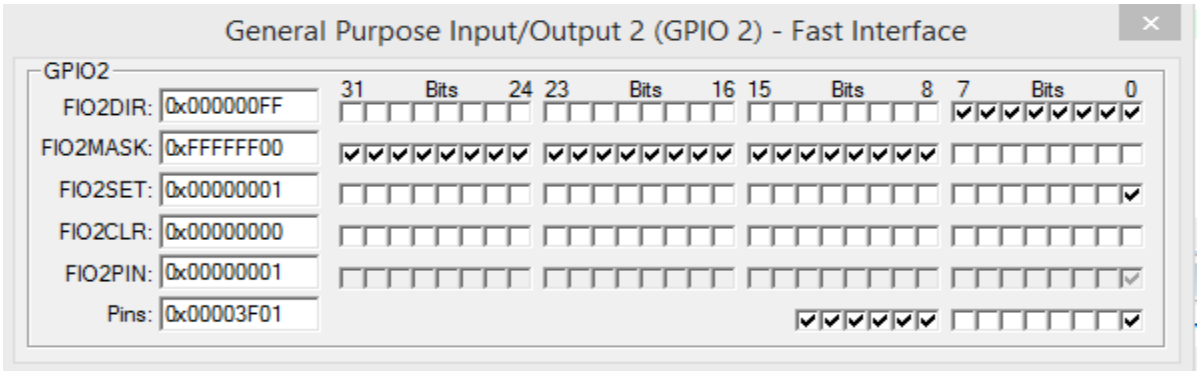
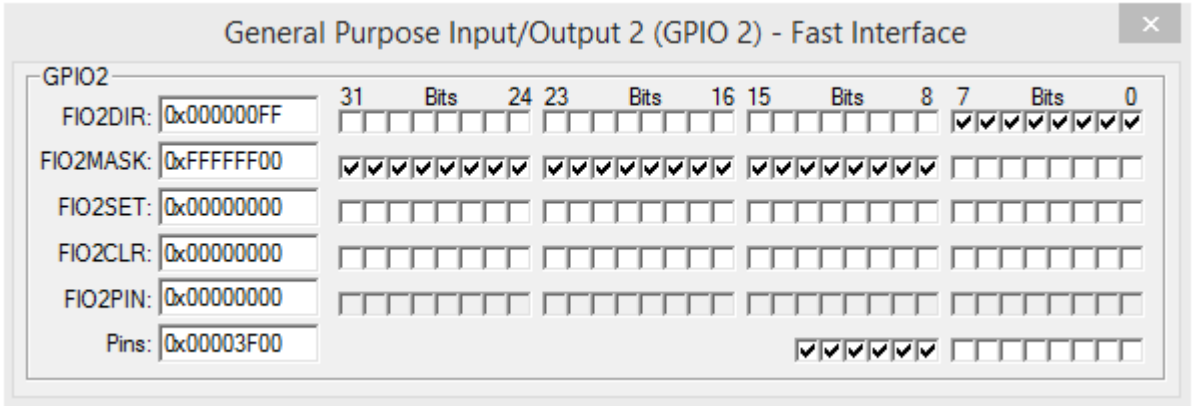


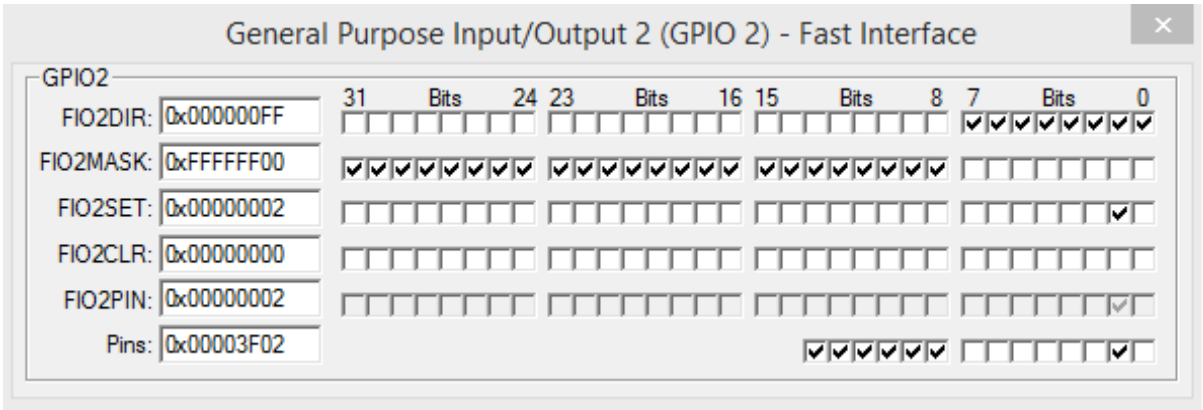
Fig.3.3 Flash window

OUTPUT:


General Purpose Input/Output 2 (GPIO 2) - Fast Interface
GPIO2
FIO2DIR: 0x00000000
FIO2MASK: 0x00000000
FIO2SET: 0x00000000
FIO2CLR: 0x00000000
FIO2PIN: 0x00003FFF
Pins: 0x00003FFF
31 Bits 24 23 Bits 16 15 Bits 8 7 Bits 0


General Purpose Input/Output 2 (GPIO 2) - Fast Interface
GPIO2
FIO2DIR: 0x000000FF
FIO2MASK: 0xFFFFFF00
FIO2SET: 0x00000001
FIO2CLR: 0x00000000
FIO2PIN: 0x00000001
Pins: 0x00003F01
31 Bits 24 23 Bits 16 15 Bits 8 7 Bits 0


General Purpose Input/Output 2 (GPIO 2) - Fast Interface
GPIO2
FIO2DIR: 0x000000FF
FIO2MASK: 0xFFFFFF00
FIO2SET: 0x00000000
FIO2CLR: 0x00000000
FIO2PIN: 0x00000000
Pins: 0x00003F00
31 Bits 24 23 Bits 16 15 Bits 8 7 Bits 0


General Purpose Input/Output 2 (GPIO 2) - Fast Interface
GPIO2
FIO2DIR: 0x000000FF
FIO2MASK: 0xFFFFFF00
FIO2SET: 0x00000002
FIO2CLR: 0x00000000
FIO2PIN: 0x00000002
Pins: 0x00003F02
31 Bits 24 23 Bits 16 15 Bits 8 7 Bits 0

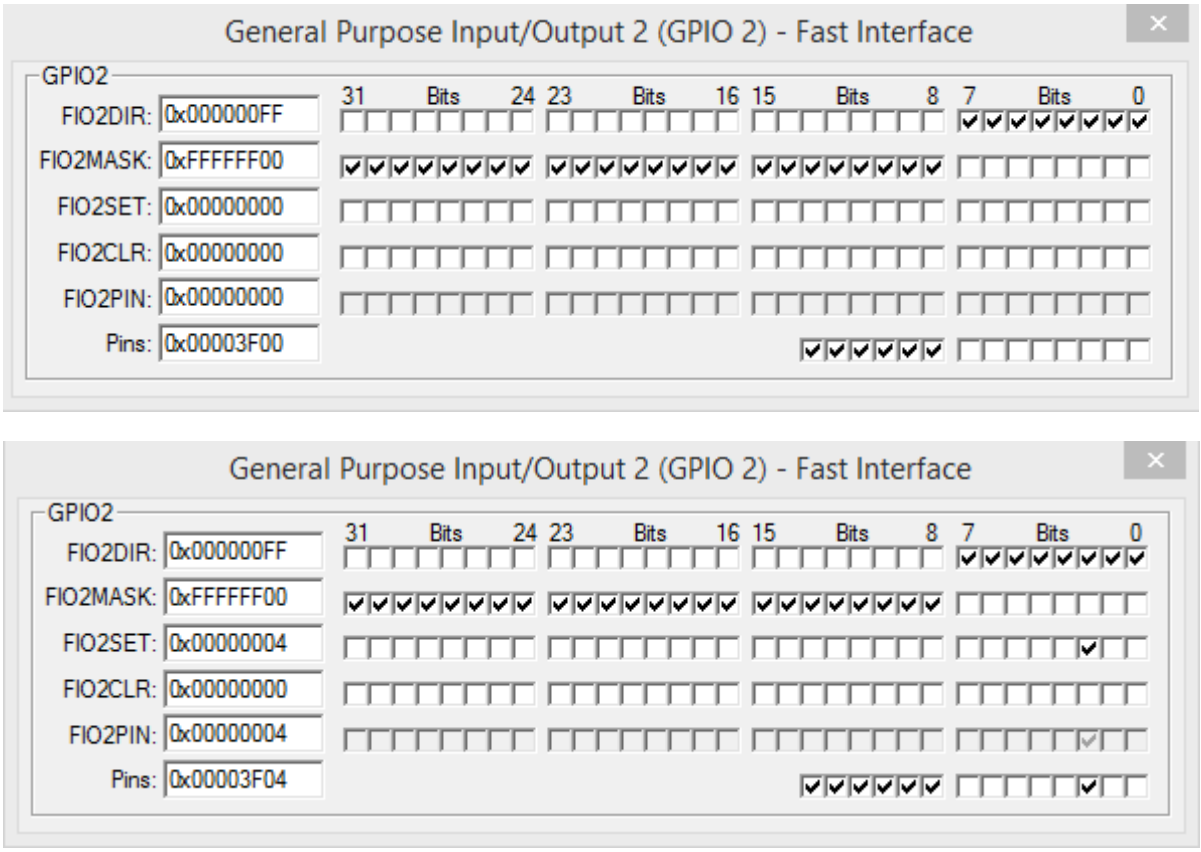


Fig. 3.4 Sample Program Output

EXERCISE:

(1) Write a program to generate even positioned LED flashing pattern on hardware board LPC 2368.
(2) Write a program to generate odd positioned LED flashing pattern on hardware board LPC 2368.

EXPERIMENT – 8

A/D AND D/A CONVERSION

OBJECTIVE (I):

To study Analog to Digital converter and its programming.

THEORY:

Special Function Registers for A to D Converter:
AD0CR, AD0GDR, AD0STAT and ADDR0-7

Features of the available A To D Converter on LPC2368:

- 10 bit successive approximation analog to digital converter
- Input multiplexing among 6 pins (LPC2364/66/68) or 8 pins (LPC2378)
- Power down mode
- Measurement range 0 to 3 V
- 10 bit conversion time >= 2.44 us
- Burst conversion mode for single or multiple inputs
- Optional conversion on transition on input pin or Timer Match signal
- Individual result registers for each A/D channel to reduce interrupt overhead

DESCRIPTION:
Basic clocking for the A/D converters is provided by the APB clock (PCLK). A programmable divider is included in each converter, to scale this clock to the 4.5 MHz (max) clock needed by the successive approximation process. A fully accurate conversion requires 11 of these clocks.

OPERATION
Hardware-Triggered Conversion
If the BURST bit in the ADCR is 0 and the START field contains 010-111, the A/D converter will start a conversion when a transition occurs on a selected pin or Timer Match signal. The choices include conversion on a specified edge of any of 4 Match signals, or conversion on a specified edge of either of 2 Capture/Match pins. The pin state from the selected pad or the selected Match signal, XORed with ADCR bit 27, is used in the edge detection logic.

INTERRUPTS
An interrupt is requested to the Vectored Interrupt Controller (VIC) when the ADINT bit in the ADSTAT register is 1. The ADINT bit is one when any of the DONE bits of A/D channels that are enabled for interrupts (via the ADINTEN register) are one.
Software can use the Interrupt Enable bit in the VIC that corresponds to the ADC to control whether this results in an interrupt. The result register for an A/D channel that is generating an interrupt must be read in order to clear the corresponding DONE flag.

Special Function Register Description:

Name	Description	Access	Reset Value[1]	Address
AD0CR	A/D Control Register. The AD0CR register must be written to select the operating mode before A/D conversion can occur.	R/W	0x0000 0001	0xE003 4000
AD0GDR	A/D Global Data Register. Contains the result of the most recent A/D conversion.	R/W	NA	0xE003 4004
AD0STAT	A/D Status Register. This register contains DONE and OVERRUN flags for all of the A/D channels, as well as the A/D interrupt flag.	RO	0	0xE003 4030
AD0INTEN	A/D Interrupt Enable Register. This register contains enable bits that allow the DONE flag of each A/D channel to be included or excluded from contributing to the generation of an A/D interrupt.	R/W	0x0000 0100	0xE003 400C
ADDR0	A/D Channel 0 Data Register. This register contains the result of the most recent conversion completed on channel 0	R/W	NA	0xE003 4010
ADDR1	A/D Channel 1 Data Register. This register contains the result of the most recent conversion completed on channel 1.	R/W	NA	0xE003 4014
ADDR2	A/D Channel 2 Data Register. This register contains the result of the most recent conversion completed on channel 2.	R/W	NA	0xE003 4018
ADDR3	A/D Channel 3 Data Register. This register contains the result of the most recent conversion completed on channel 3.	R/W	NA	0xE003 401C
ADDR4	A/D Channel 4 Data Register. This register contains the result of the most recent conversion completed on channel 4.	R/W	NA	0xE003 4020
ADDR5	A/D Channel 5 Data Register. This register contains the result of the most recent conversion completed on channel 5.	R/W	NA	0xE003 4024
ADDR6	A/D Channel 6 Data Register. This register contains the result of the most recent conversion completed on channel 6.	R/W	NA	0xE003 4028
ADDR7	A/D Channel 7 Data Register. This register contains the result of the most recent conversion completed on channel 7.	R/W	NA	0xE003 402C

Fig. 4.1 Special Function Registers of A/D converter

Table 456: A/D Control Register (AD0CR - address 0xE003 4000) bit description

Bit	Symbol	Value	Description	Reset Value
7:0	SEL		Selects which of the AD0.7:0 pins is (are) to be sampled and converted. For AD0, bit 0 selects Pin AD0.0, and bit 7 selects pin AD0.7. In software-controlled mode, only one of these bits should be 1. In hardware scan mode, any value containing 1 to 8 ones. All zeroes is equivalent to 0x01.	0x01
15:8	CLKDIV		The APB clock (PCLK) is divided by (this value plus one) to produce the clock for the A/D converter, which should be less than or equal to 4.5 MHz. Typically, software should program the smallest value in this field that yields a clock of 4.5 MHz or slightly less, but in certain cases (such as a high-impedance analog source) a slower clock may be desirable.	0
16	BURST	0	Conversions are software controlled and require 11 clocks.	0
		1	The AD converter does repeated conversions at the rate selected by the CLKS field, scanning (if necessary) through the pins selected by 1s in the SEL field. The first conversion after the start corresponds to the least-significant 1 in the SEL field, then higher numbered 1 bits (pins) if applicable. Repeated conversions can be terminated by clearing this bit, but the conversion that's in progress when this bit is cleared will be completed. **Important:** START bits must be 000 when BURST = 1 or conversions will not start.	
19:17	CLKS		This field selects the number of clocks used for each conversion in Burst mode, and the number of bits of accuracy of the result in the LS bits of ADDR, between 11 clocks (10 bits) and 4 clocks (3 bits).	000
		000	11 clocks / 10 bits	
		001	10 clocks / 9 bits	
		010	9 clocks / 8 bits	
		011	8 clocks / 7 bits	
		100	7 clocks / 6 bits	
		101	6 clocks / 5 bits	
		110	5 clocks / 4 bits	
		111	4 clocks / 3 bits	
20			Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
21	PDN	1	The A/D converter is operational.	0
		0	The A/D converter is in power-down mode.	
23:22	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
26:24	START		When the BURST bit is 0, these bits control whether and when an A/D conversion is started:	0
		000	No start (this value should be used when clearing PDN to 0).	
		001	Start conversion now.	
		010	Start conversion when the edge selected by bit 27 occurs on P2.10/EINT0.	
		011	Start conversion when the edge selected by bit 27 occurs on P1.27/CAP0.1.	
		100	Start conversion when the edge selected by bit 27 occurs on MAT0.1[1].	
		101	Start conversion when the edge selected by bit 27 occurs on MAT0.3[1].	
		110	Start conversion when the edge selected by bit 27 occurs on MAT1.0[1].	
		111	Start conversion when the edge selected by bit 27 occurs on MAT1.1[1].	

Table 456: A/D Control Register (AD0CR - address 0xE003 4000) bit description

Bit	Symbol	Value	Description	Reset Value
27	EDGE		This bit is significant only when the START field contains 010-111. In these cases:	0
		1	Start conversion on a falling edge on the selected CAP/MAT signal.	
		0	Start conversion on a rising edge on the selected CAP/MAT signal.	
31:28	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Fig. 4.2 A/D Control Register

Table 458: A/D Status Register (ADSTAT - address 0xE003 4030) bit description

Bit	Symbol	Description	Reset Value
7:0	Done7:0	These bits mirror the DONE status flags that appear in the result register for each A/D channel.	0
15:8	Overrun7:0	These bits mirror the OVERRRUN status flags that appear in the result register for each A/D channel. Reading ADSTAT allows checking the status of all A/D channels simultaneously.	0
16	ADINT	This bit is the A/D interrupt flag. It is one when any of the individual A/D channel Done flags is asserted and enabled to contribute to the A/D interrupt via the ADINTEN register.	0
31:17	Unused	Unused, always 0.	0

Fig. 4.3 A/D status register

Table 459: A/D Interrupt Enable Register (ADINTEN - address 0xE003 400C) bit description

Bit	Symbol	Description	Reset Value
7:0	ADINTEN 7:0	These bits allow control over which A/D channels generate interrupts for conversion completion. When bit 0 is one, completion of a conversion on A/D channel 0 will generate an interrupt, when bit 1 is one, completion of a conversion on A/D channel 1 will generate an interrupt, etc.	0x00
8	ADGINTEN	When 1, enables the global DONE flag in ADDR to generate an interrupt. When 0, only the individual A/D channels enabled by ADINTEN 7:0 will generate interrupts.	1
31:9	Unused	Unused, always 0.	0

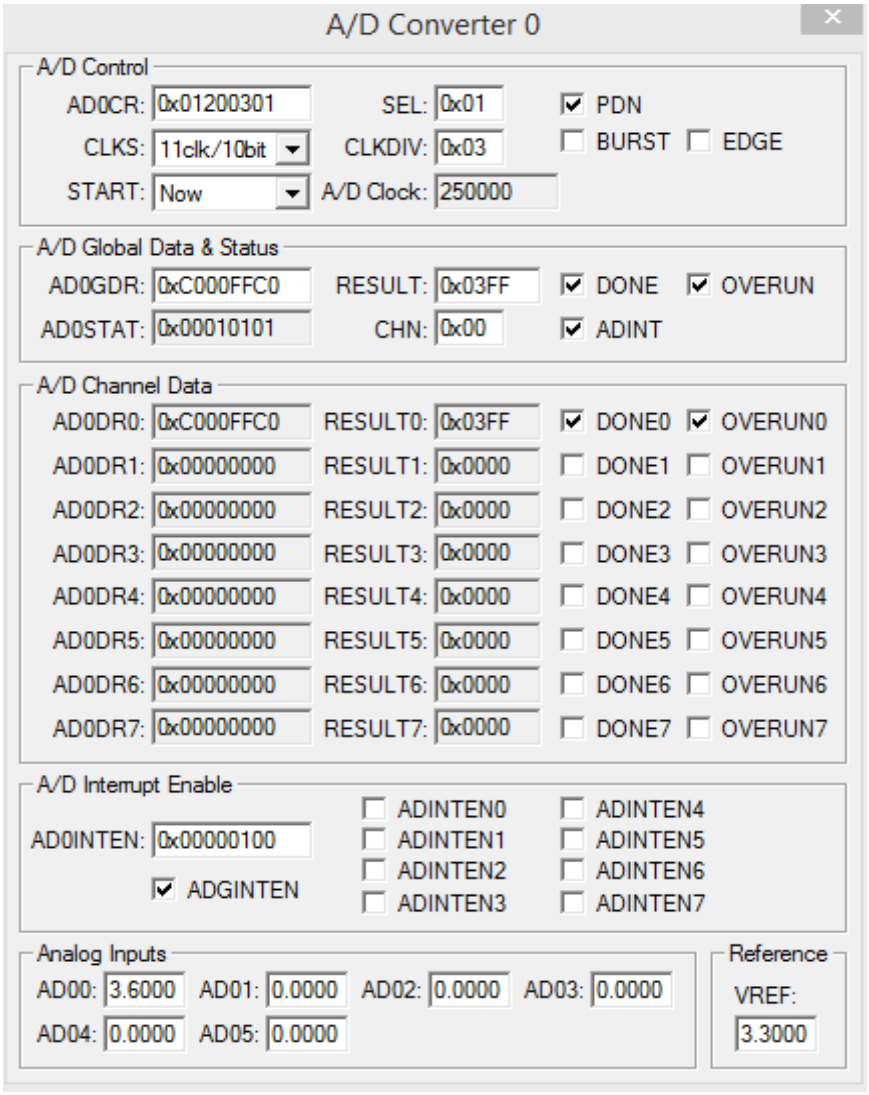
Fig. 4.4 A/D Interrupt Enable Register

SAMPLE PROGRAM: A/D conversion for a single channel

```
#include<LPC23xx.h>
int main()
{
        inti,j;
        AD0CR=0X00200301;   //pdn enabled, sel:00, clkdiv:03,
                                    //clks:11clks/10bits,start:none
        AD0CR=0X01200301;           //set start as :now
        while((AD0STAT&0x00000001)!=0x00000001) // monitor done flag in AD0STAT
        { }
        i= AD0GDR/0x00000040;
        j=i&0x00003ff;
        return 0;
}
```

OUTPUT:

By providing input value=3.6V the digital equivalent is calculated.



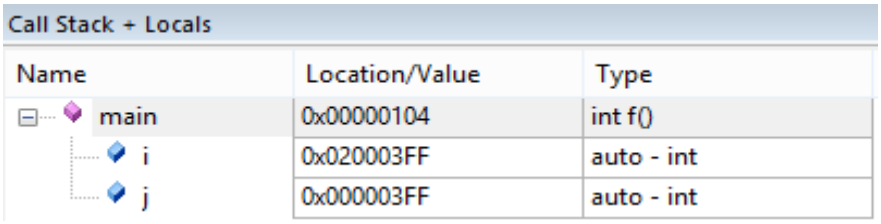


Fig. 4.5 Sample Program Output

SAMPLE PROGRAM2: A/D conversion for six channels

```
#include<LPC23xx.h>
int main( )
{
        int  i[6], j[6],k,l;
        k=0x00000001;
        for(l=0;l<6;l++)
```

```
        {       AD0CR=0X00200300+k;
                AD0CR=0X01200300+k;
                while((AD0STAT&k)!=k)
                {
                        i[l]= AD0GDR/0x00000040;
                        j[l]=i[l]&0x000003ff;
                        k=k*2;
                }
        return 0;
}
```

OUTPUT:

By providing different values, the digital equivalent is calculated.

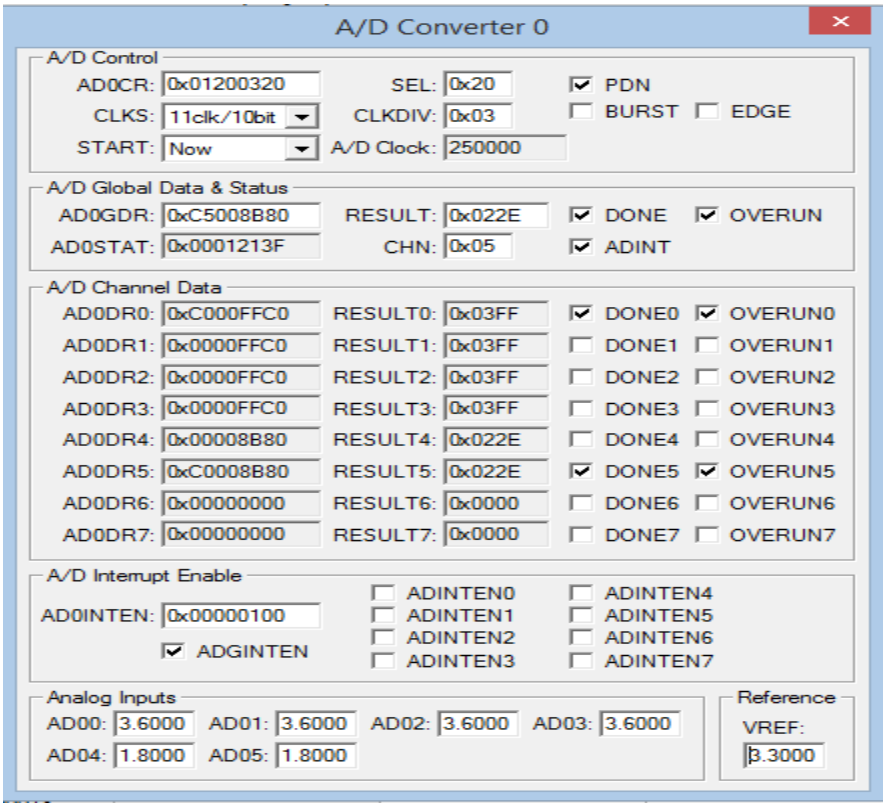


Fig. 4.6 Sample Program Output

OBJECTIVE (II):

To study Digital to Analog converter and its programming.

THEORY:

Special Function Register of Digital to Analog Converter:
DACR

Special Function Register Description:

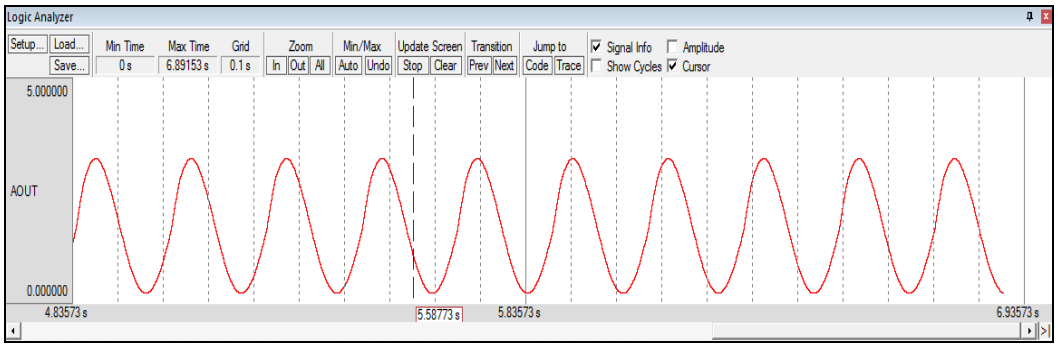
Bit	Symbol	Value	Description	Reset Value
5:0	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
15:6	VALUE		After the selected settling time after this field is written with a new VALUE, the voltage on the A_{OUT} pin (with respect to V_{SSA}) is VALUE/1024 × VREF.	0
16	BIAS	0	The settling time of the DAC is 1 μs max, and the maximum current is 700 υA.	0
		1	The settling time of the DAC is 2.5 μs and the maximum current is 350 μA.	
31:17	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Fig. 4.7 D/A Special Function Register

SAMPLE PROGRAM 1: Generation of sine wave using sin() function

```
#include <LPC23xx.h>
#include <math.h>
int main ()
{
        int x, y;
        PINSEL1=(1<<21);                                // selecting pin no 21
        while(1)
        {
                for(x=0;x<360;x=x++)
                {
                        y=(int)(512+512*sin(x*3.14/180));
                        DACR=y<<6;
                }
        }
        return 0;
}
```

OUTPUT:
By opening the logic analyzer window, the sine wave can be observed as below:



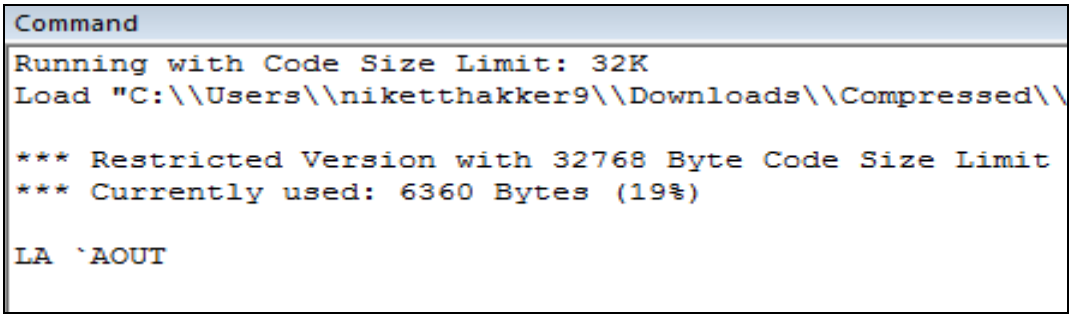


Fig. 4.8 Sample Program Output

SAMPLE PROGRAM 2: Generation of sine wave using lookup table

```
#include<lpc23xx.h>
#include<math.h>
int main()
{
        inti,j,n;
        int
        arr[37]={512,601,687,768,841,904,955,993,1016,1023,1016,955,933,904,841,768,687,601,512,4
        23,336,256,183,120,69,31,8,0,8,31,69,120,183,256,336,423,512};
        // creating lookup table
        i=0;
        n=0;
        PINSEL1=(1<<21);        //selecting pin no 21
label:
        for(i=0;i<36;i++)
         {
                n=arr[i];
                 DACR=n<<6;  //keeping the value of n and shifting by  6 places
           }
        goto label;
}
```

OUTPUT:

By opening the logic analyser window, the sine wave can be observed as below:

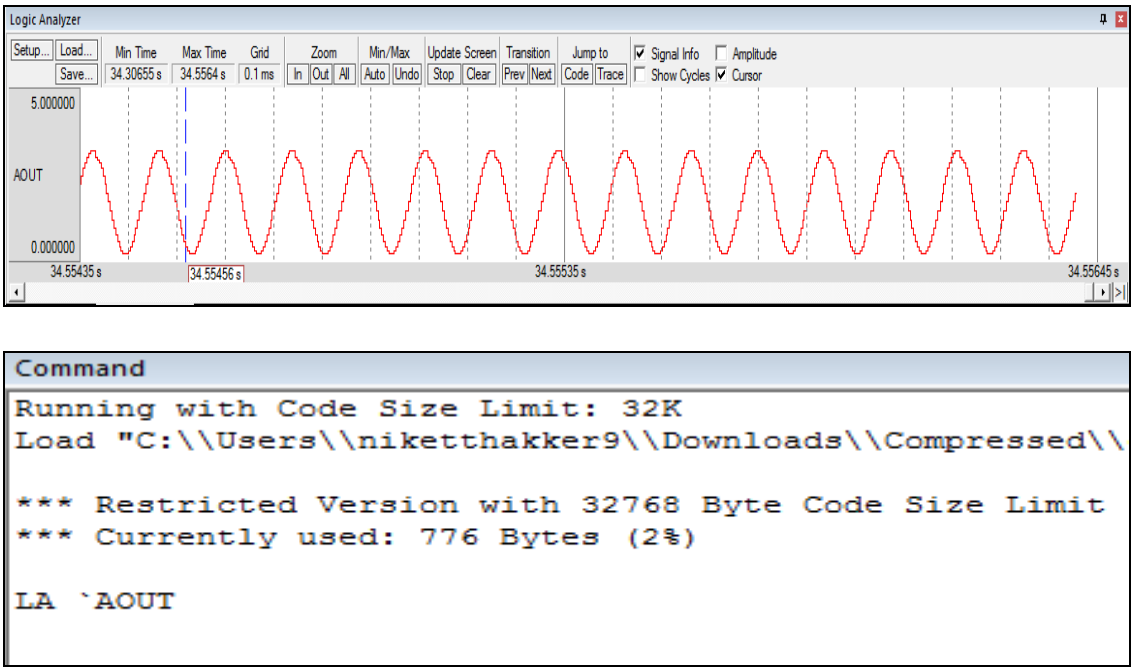


Fig. 4.9 Sample Program Output

EXERCISE:

(1) Explain working of SAR.
(2) Explain conversion Equation of ADC.
(3) What are the specifications of A/D converter?
(4) What are the popular architectures of A/D converter?
(5) How speed of conversion can be compromised with resolution?
(6) What are the specifications of D/A converter?
(7) What are the popular architectures of D/A converter?
(8) What is logic Analyzer? How it is useful?
(9) How amplitude of the waveform generated can be varied?

EXPERIMENT – 9

ARRAY PROCESSING

OBJECTIVES:
To study start-up file for LPC2300 and programming of ARM assembly language.

THEORY:

A start-up code performs stack initialization and the microcontroller setup, before an arm microcontroller can execute main program. The start-up file LPC2300.s code is executed after CPU reset.

The file LPC2300.s is an assembler module provided by Keil. As its name implies, the start-up code is located to run from the reset vector. It provides the exception vector table as well as initializing the stack pointer for the different operating modes. It also initializes some of the on-chip system peripherals and the on-chip RAM before it jumps to the main function in c code. The start-up code will vary depending on which arm7 device you are using and which compiler you are using, so for your own project it is important to make sure that you are using the correct start-up file.

First of all the start-up provides the exception Vector table as shown below. The vector table is located. At 0x00000000 and provides a jump to interrupt service routines (ISR) on each vector to ensure that the full Address range of the processor is available, the LDR (Load Register) instruction is used. The area command is used by the linker to the Place the vector table at the correct start address. For a single chip use this is always 0x00000000, however if you are using the external bus and want to boot from external memory, the vector table must be located at 0x80000000.

SECTIONS OF A START-UP FILE:

1. Interrupt Vector Table
2. Clock Selection and PLL Configuration
3. Peripheral Configuration
4. Stack Definition

START-UP FILE: lpc2300.s

```
Mode_USR      EQU    0x10
Mode_FIQ      EQU    0x11
Mode_IRQ      EQU    0x12
Mode_SVC      EQU    0x13
Mode_ABT      EQU    0x17
Mode_UND      EQU    0x1B
Mode_SYS      EQU    0x1F
I_Bit         EQU    0x80        ; when I bit is set, IRQ is disabled
F_Bit         EQU    0x40        ; when F bit is set, FIQ is disabled
; Startup Code must be linked first at Address at which it expects to run.
        AREA    RESET, CODE, READONLY
        ARM
; Exception Vectors
; Mapped to Address 0
; Absolute addressing mode must be used
; Dummy Handlers are implemented as infinite loops which can be modified.
; Vectors
        LDR     PC, Reset_Addr
        LDR     PC, Undef_Addr
        LDR     PC, SWI_Addr
        LDR     PC, PAbt_Addr
```

```
        LDR     PC, DAbt_Addr
        NOP
; Reserved Vector ;
        LDR     PC, IRQ_Addr
        LDR     PC, [PC, #-0x0120]      ; Vector from VicVectAddr
        LDR     PC, FIQ_Addr
Reset_Addr      DCD     Reset_Handler
Undef_Addr      DCD     Undef_Handler
SWI_Addr        DCD     SWI_Handler
PAbt_Addr       DCD     PAbt_Handler
DAbt_Addr       DCD     DAbt_Handler
                DCD     0
; Reserved Address
IRQ_Addr        DCD     IRQ_Handler
FIQ_Addr        DCD     FIQ_Handler
Undef_Handler   B       Undef_Handler
SWI_Handler             B       SWI_Handler
PAbt_Handler            B       PAbt_Handler
DAbt_Handler            B       DAbt_Handler
IRQ_Handler             B       IRQ_Handler
FIQ_Handler             B       FIQ_Handler
; Reset Handler
        EXPORT  Reset_Handler
; Reset_Handler
; Enter the C code
        IMPORT  __main
        LDR R0, =__main
        BX R0                                   // branch to main program
        IF :DEF:__MICROLIB
        EXPORT  __heap_base
        EXPORT  __heap_limit
        ELSE ; User Initial Stack & Heap
        AREA    |.text|, CODE, READONLY
        IMPORT  __use_two_region_memory
        EXPORT  __user_initial_stackheap
         __user_initial_stackheap
        LDR     R0, =  Heap_Mem
        LDR     R1, = (Stack_Mem + USR_Stack_Size)
        LDR     R2, = (Heap_Mem +      Heap_Size)
        LDR     R3, = Stack_Mem
        BX LR
        ENDIF
        END
```

PROCEDURE:
In built ARM start-up file (lpc2300.s) is written for high level language applications, i.e. for C Programming Language. To write the low level language program (assembly program), it is necessary to make certain modifications in a start-up file.

Steps are mentioned below to write ARM assembly language program / application.

1. In a start-up file, replace actual code from “enter the c code module” to end as shown below.

```
        IMPORT  START
        LDR R0, =START
        BX R0
        END
```

Here, “start” is a label to a block of area where an assembly language program / application’s code exists.

2. An assembly program / application code will start with AREA directive, which instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.
   - "TEST" is a name of block where program code resides, instead of "TEST", any name can be given.
   - CODE means it is program code.
   - READONLY means given code area is read-only.
   - "START" is starting point of program.
   - "END" is end point directive of a program.

SAMPLE PROGRAM: Copy an array of size 10 bytes to another memory location.

```
        AREA TEST, CODE, READONLY
        EXPORT START
START
        MOV R3,#10 ; COUNTER
        MOV R0,#0X40000000
        MOV R2,#0X40000030;
        LOOP
        LDRB R1,[R0];
        STRB R1,[R2];
        ADD R0,R0,#0X01;
        ADD R2,R2,#0X01;
        SUB R3,R3,#0X01;
        CMP R3,#0;
        BNE LOOP;
        END
```

OUTPUT:

Initially the memory window looks like below.

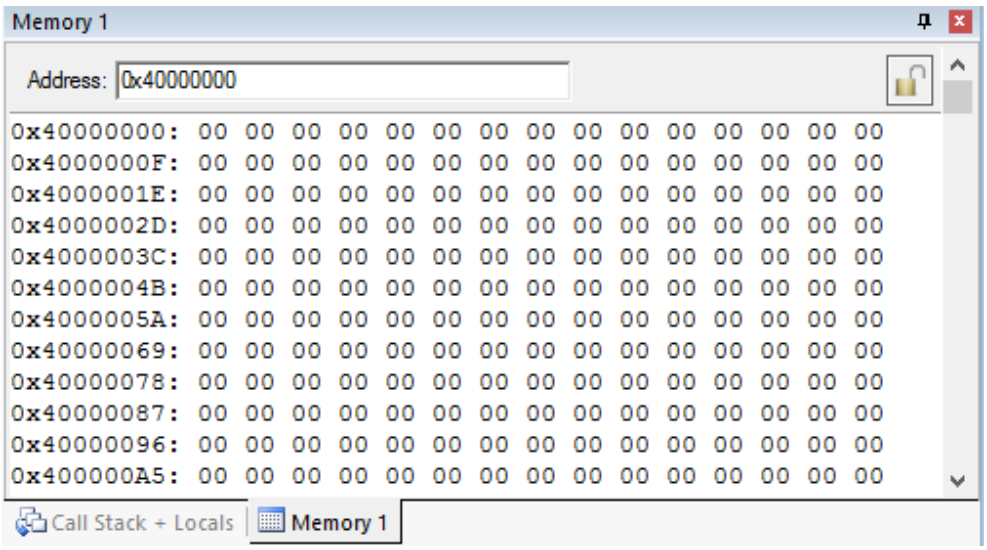


Fig 5.1 Memory Window

So first of all an array is created on the required memory locations by double clicking on that location. The created array looks like below:

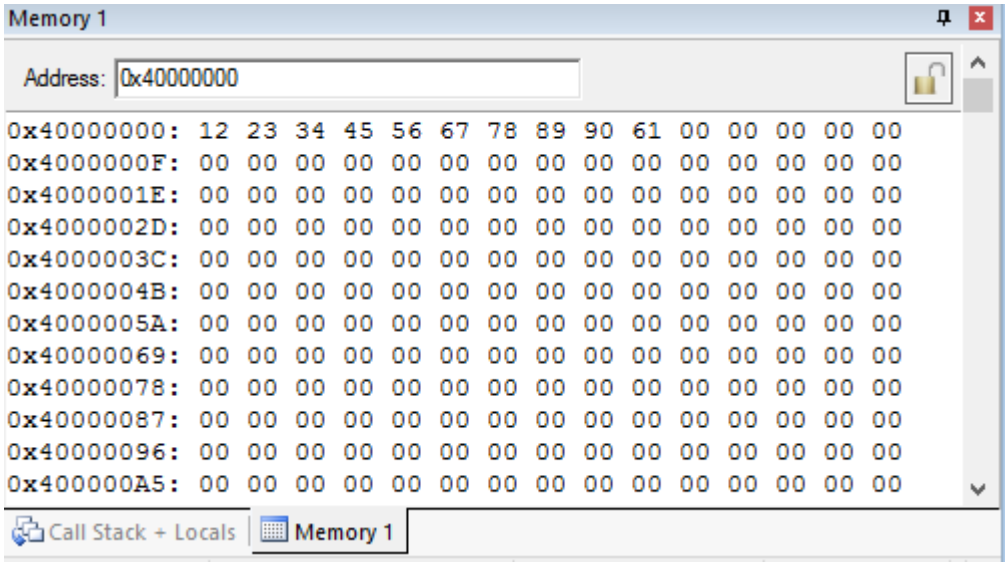


Fig 5.2 Memory Window – After Inserting Value

After the successful execution of program, the array gets copied to the required memory location which is as shown below:

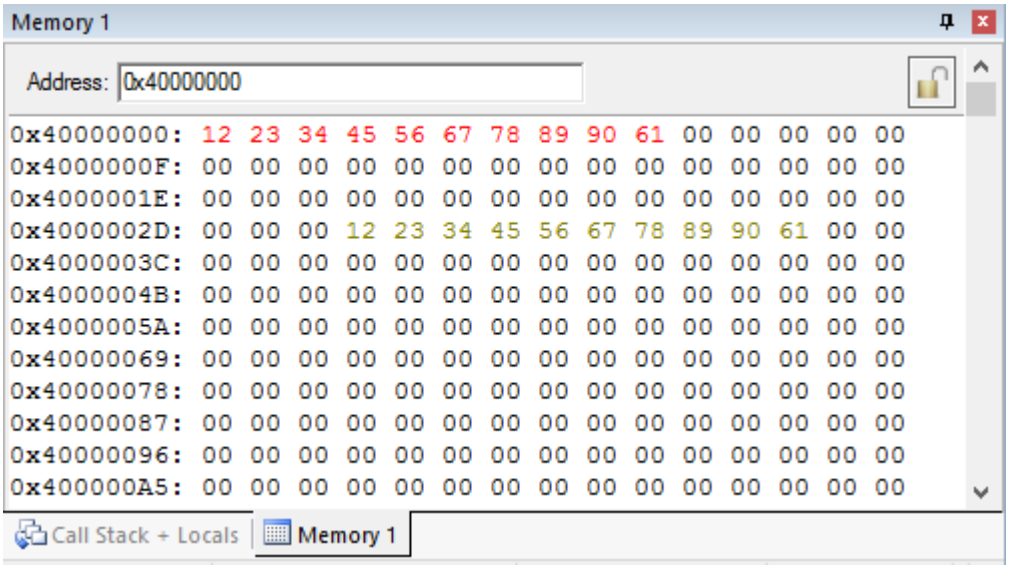


Fig 5.3 Sample Program Output

MODIFICATION:

(1) Modify the above code to find the count of even numbers from an array of size 10 bytes.

CONCLUSION:

(b)To perform an array processing in ARM assembly language programming.

PROCEDURE:
In built ARM start-up file (lpc2300.s) is written for high level language applications, i.e. for C Programming Language. To write the low level language program (assembly program), it is necessary to make certain modifications in a start-up file.

Steps are mentioned below to write ARM assembly language program / application.

1. In a start-up file, replace actual code from “enter the c code module” to end as shown below.

```
IMPORT  START
LDR R0, =START
BX R0
END
```

Here, “start” is a label to a block of area where an assembly language program / application’s code exists.

2. An assembly program / application code will start with AREA directive, which instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.
   - “TEST” is a name of block where program code resides, instead of “TEST”, any name can be given.
   - CODE means it is program code.
   - READONLY means given code area is read-only.
   - “START” is starting point of program.
   - “END” is end point directive of a program.

SAMPLE PROGRAM: Sort an array in ascending order

```
                AREA TEST, CODE, READONLY
                EXPORT START
START
                MOV R0,#0X40000000
                MOV R1,R0
                MOV R4,#4
                MOV R5,#4
HERE2
                LDR R2,[R0]
HERE1
                ADD R1,R1,#4
                LDR R3,[R1]
                CMP R2,R3
                BLT HERE
                MOV R6,R2
                MOV R2,R3
                MOV R3,R6
                STR R2,[R0]
                STR R3,[R1]
HERE
                SUB R4,R4,#1
                CMP R4,#0
                BNE HERE1
                ADD R0,R0,#4
                MOV R1,R0
                SUB R5,R5,#1
                MOV R4,R5
```

```
CMP R5,#0
BNE HERE2
END
```

OUTPUT:

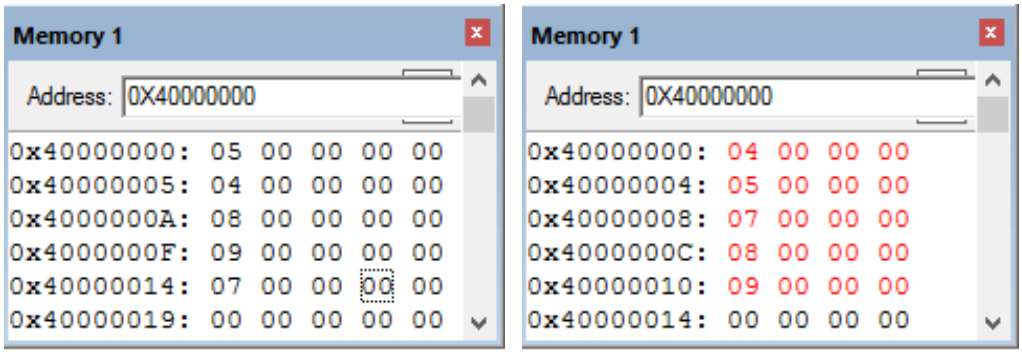


Fig 6.1 Sample Program Output

MODIFICATION:

(1) Implement the sorting of an array in descending order.

CONCLUSION:

EXERCISE:

(1) Find the count of even and odd numbers from an array of size 20 bytes.
(2) Find the count of 1's from an array of size 5 bytes.

EXPERIMENT – 10

INLINE ASSEMBLY & THUMB STATE

OBJECTIVE:

(I) To write and execute a c language program that allows inline assembly instructions.
(II) To write and execute an assembly language program in arm state and thumb state.

THEORY:

Inline assembly is a special provision in the ARM processors which allows user to write ARM assembly language program in higher language. Basic advantage of such provision is to get high code density. This provision is mostly used where code storage capacity of processor is less compared to the requirement.
Restrictions on inline assembly operations: There are a number of restrictions on the operations that can be performed in inline assembly code. These restrictions provide a measure of safety, and ensure that the assumptions in compiled C and C++ code are not violated in the assembled assembly code.

MISCELLANEOUS RESTRICTIONS:
The inline assembler has the following restrictions:

- The inline assembler is a high-level assembler, and the code it generates might not always be exactly what you write. Do not use it to generate more efficient code than the compiler generates. Use embedded assembler or the ARM assembler armasm for this purpose.
- Some low-level features that are available in the ARM assembler armasm, such as branching and writing to PC, are not supported. x Label expressions are not supported.
- You cannot get the address of the current instruction using dot notation (.) or {PC}.
- The & operator cannot be used to denote hexadecimal constants. Use the 0x prefix instead. For example:__asm { AND x, y, 0xF00 }
- The notation to specify the actual rotate of an 8-bit constant is not available in inline assembly language. This means that where an 8-bit shifted constant is used, the C flag must be regarded as corrupted if the NZCV flags are updated.
- You must not modify the stack. This is not necessary because the compiler automatically stacks and restores any working registers as required. The compiler does not permit you to explicitly stack and restore work registers.

REGISTERS:
Registers, such as r0-r3, sp, lr, and the NZCV flags in the CPSR must be used with caution. If you use C or C++ expressions, these might be used as temporary registers and NZCV flags might be corrupted by the compiler when evaluating the expression.
The pc, lr, and sp registers cannot be explicitly read or modified using inline assembly code because there is no direct access to any physical registers. However, you can use the following intrinsics to access these registers:

- current_pc in the Compiler Reference Guide
- current_sp in the Compiler Reference Guide
- return_address in the Compiler Reference Guide.

THUMB INSTRUCTION SET:
The inline assembler is not available when compiling C or C++ for Thumb state, and the inline assembler does not assemble Thumb instructions. Instead, the compiler switches to ARM state automatically.

UNSUPPORTED INSTRUCTIONS:
The following instructions are not supported in the inline assembler:

- BKPT , BX , BXJ, and BLX instructions
- SVC instruction x LDR Rn, =expression pseudo-instruction. Use MOV Rn, expression instead (this can generate a load from a literal pool)
- LDRT, LDRBT, STRT, and STRBT instructions
- MUL, MLA, UMULL, UMLAL, SMULL, and SMLAL flag setting instructions
- MOV or MVN flag-setting instructions where the second operand is a constant
- user-mode LDM instructions
- ADR and ADRL pseudo-instructions.

SAMPLE PROGRAM: illustration of inline assembly language instructions in c program

```
int main( )
{
        int a=10,b=10,c,d,e,f,g;
        c=a+b;
        _ _asm
        {
                mov e,0x04;
                mov f,0x02;
                add g,e,f;
        }
        d=a*b;
        return 0;
}
```

OUTPUT:

The disassembly & register window for the above program showing the C statements into their corresponding assembly language statements is shown below:

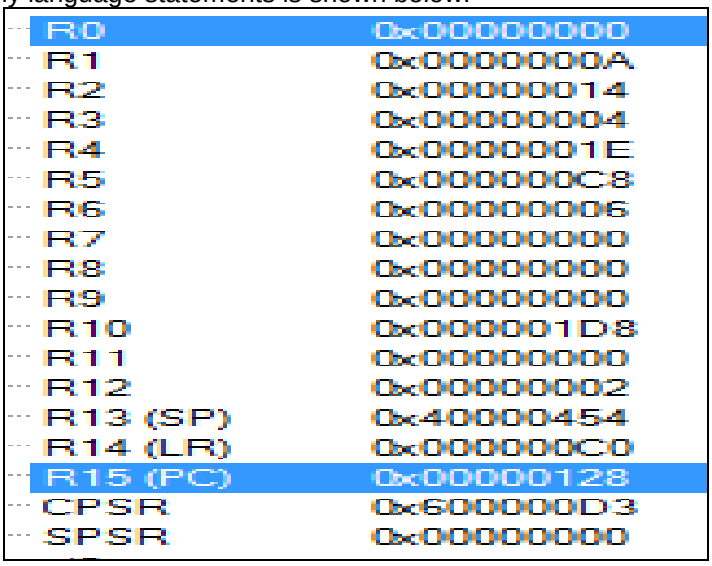


Fig 7.1 Window: Register

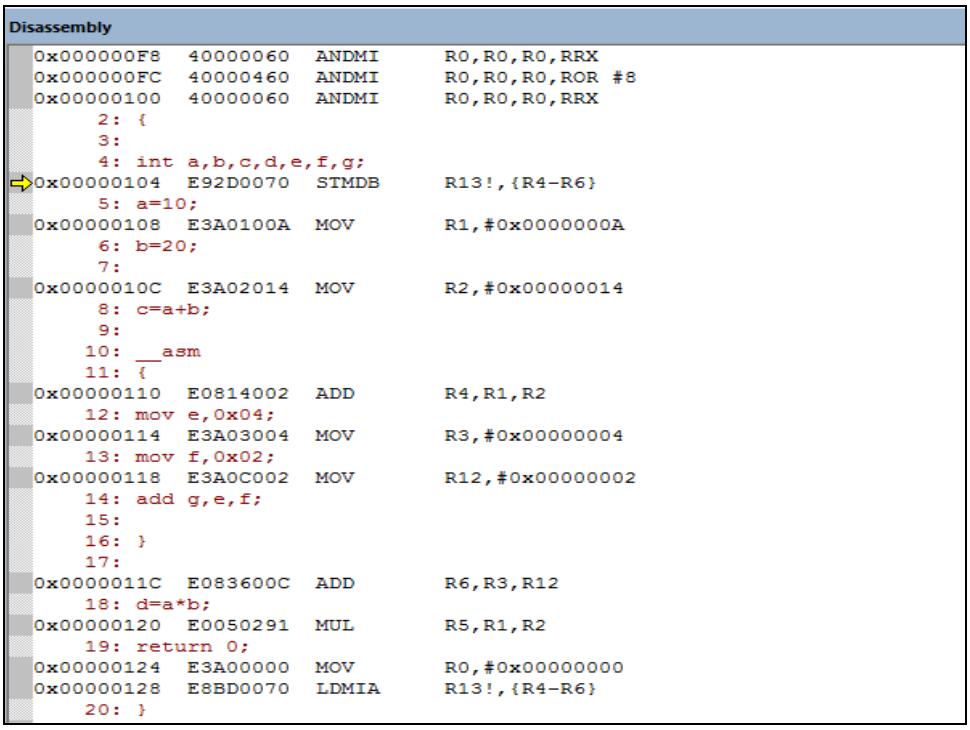


Fig 7.2 Window: Disassembly

(II) To study, write and execute an assembly language program executing in an arm state and in a thumb state.

THEORY:

The Thumb instruction set addresses the issue of code density. It may be viewed as a compressed form of a subset of the ARM instruction set. Thumb instructions map onto ARM instructions, and the Thumb programmer's model maps onto the ARM programmer's model. Implementations of Thumb use dynamic decompression in an ARM instruction pipeline and then instructions execute as standard ARM instructions within the processor.

Thumb is not a complete architecture; it is not anticipated that a processor would execute Thumb instructions without also supporting the ARM instruction set. Therefore the Thumb instruction set need only support common application functions, allowing recourse to the full ARM instruction set where necessary (for instance, all exceptions automatically enter ARM mode). Thumb is fully supported by ARM development tools, and an application can mix ARM and Thumb subroutines flexibly to optimize performance or code density on a routine-by-routine basis.

SAMPLE PROGRAM:

```
        AREA TEST, CODE, READONLY
        EXPORT SQUARE
SQUARE
        MOV R1,#&44
        MOVS R2,#44
        ADCS R3,R2,R1,ASR#5
        SUBS R7,R2,R1,LSL#4
        LDR R5,=FUN
        MOVS LR,PC
        BX R5
        ADD R1,#1
        ADD R1,#1
        ADD R1,#1
        B FUNCTION
        THUMB
FUN
        LDR R6,=0X40000000
        MOVS R7,R6
        LDR R0,=65534
        LDR R1,=280
        MOVS R2,#54
        MOVS R3,#0
        STMIA R6!,{R0-R3}
        MOVS R2,#0
LOOP
        LDR R3,[R7]
        ADDS R2,R3
        ADDS R7,#4
        CMP R6,R7
        BGT LOOP
        BXLR
        ARM
FUNCTION
        ADD R5,R0,R1
        END
```

OUTPUT:

By checking CPSR [5] (TBIT), we can always say whether the processor is in ARM state or Thumb state. If TBIT=1, it is in Thumb state otherwise in ARM state. The figure below shows the initial execution of program while the processor is still in ARM state.

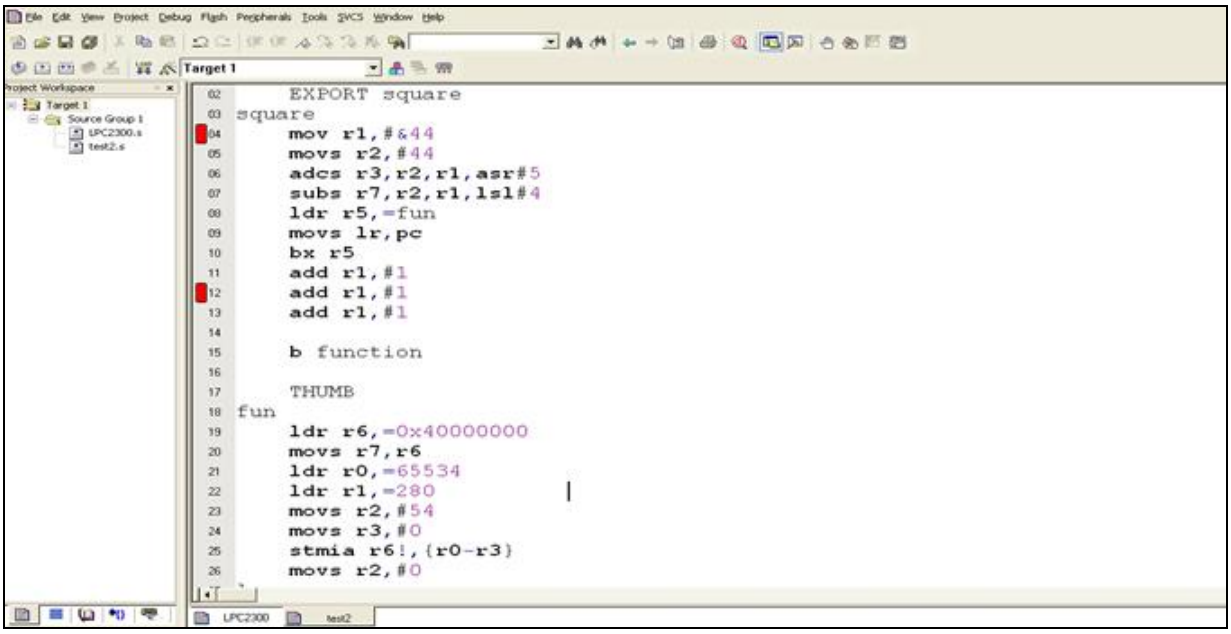


Fig 7.3 Program Execution Window 1

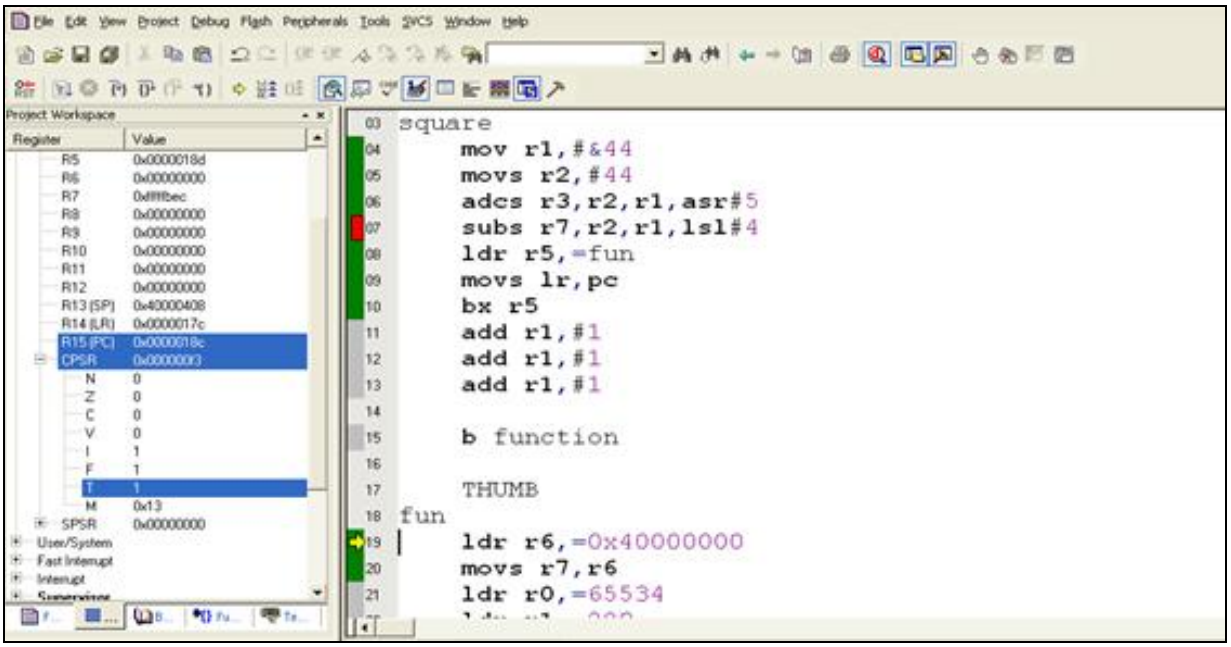


Fig 7.4 Program Execution Window 2 (Observe : T bit)

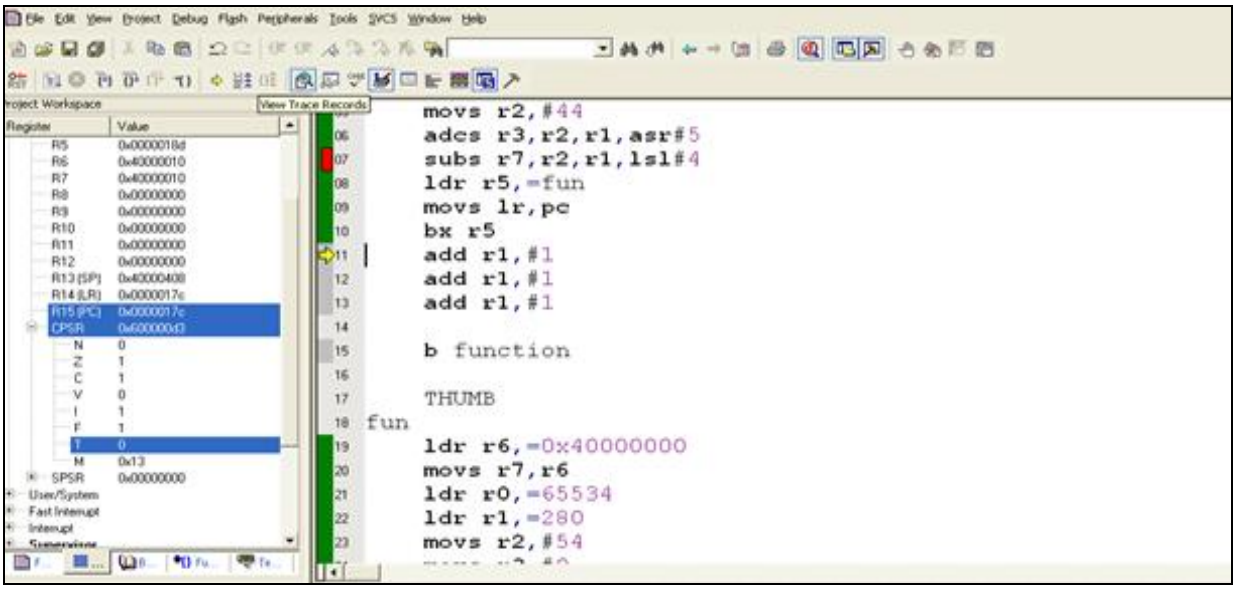


Fig 7.5 Program Execution Window 3 (Observe : T bit)

CONCLUSION:

EXPERIMENT – 11

SUBROUTINES AND SOFTWARE INTERRUPTS

OBJECTIVE:

To write and execute an assembly language program having subroutine.

THEORY:

The use of subroutine is to define a function which may be required to call number of times. The subroutine may be defined within the file or in separate assembly file. The reusability of a piece of code is possible if subroutine is defined in separate file. IMPORT and EXPORT directives plays an important role while writing subroutines in a separate file.

SAMPLE PROGRAM:

```
        AREA TEST, CODE, READONLY
        EXPORT START
DO_ADD
        ADD R4, R0, R1
        BX LR
DO_MUL
        MUL R5, R2, R1
        BX LR
START
        MOV R0, #10
        MOV R1, #3
        BL DO_ADD
        MOV R2, R4
        BL DO_MUL
        MOV R3, R5
HERE
        B HERE
        END
```

OUTPUT: Observe the execution of above program.

MODIFICATION:

(1) Implement the above mentioned subroutines in a separate file.

CONCLUSION:

EXERCISE:

(1) Write an assembly language program which calculates the sum and the average of an array of size 10 bytes. (Note: For calculating sum and average create separate file subroutines.)

(2) Write an assembly language program which will have find ascending and descending order of an array size 10 bytes. (Note: For finding ascending and descending order of an array make separate file subroutines.)

EXPERIMENT – 12

IRQ AND FIQ EXCEPTION HANDLING

OBJECTIVES:

(i) To understand IRQ Exception Handling mechanism and its programming.

(ii) To understand FIQ Exception Handling mechanism and its programming.

THEORY:

Special Function Registers of Vectored Interrupt Controller
VICIRQStatus, VICFIQStatus , VICRawintr, VICIntSelect, VICIntEnable, VICIntEnClr, VICSoftInt and VICSoftIntClr

Special Function Registers Description:

Table 63. VIC register map

Name	Description	Access	Reset value[1]	Address
VICIRQStatus	IRQ Status Register. This register reads out the state of those interrupt requests that are enabled and classified as IRQ.	RO	0	0xFFFF F000
VICFIQStatus	FIQ Status Requests. This register reads out the state of those interrupt requests that are enabled and classified as FIQ.	RO	0	0xFFFF F004
VICRawIntr	Raw Interrupt Status Register. This register reads out the state of the 32 interrupt requests / software interrupts, regardless of enabling or classification.	RO	-	0xFFFF F008
VICIntSelect	Interrupt Select Register. This register classifies each of the 32 interrupt requests as contributing to FIQ or IRQ.	R/W	0	0xFFFF F00C
VICIntEnable	Interrupt Enable Register. This register controls which of the 32 interrupt requests and software interrupts are enabled to contribute to FIQ or IRQ.	R/W	0	0xFFFF F010
VICIntEnClr	Interrupt Enable Clear Register. This register allows software to clear one or more bits in the Interrupt Enable register.	WO	-	0xFFFF F014
VICSoftInt	Software Interrupt Register. The contents of this register are ORed with the 32 interrupt requests from various peripheral functions.	R/W	0	0xFFFF F018
VICSoftIntClear	Software Interrupt Clear Register. This register allows software to clear one or more bits in the Software Interrupt register.	WO	-	0xFFFF F01C
VICProtection	Protection enable register. This register allows limiting access to the VIC registers by software running in privileged mode.	R/W	0	0xFFFF F020
VICSWPriorityMask	Software Priority Mask Register. Allows masking individual interrupt priority levels in any combination.	R/W	0xFFFF	0xFFFF F024
VICVectAddr0	Vector address 0 register. Vector Address Registers 0-31 hold the addresses of the Interrupt Service routines (ISRs) for the 32 vectored IRQ slots.	R/W	0	0xFFFF F100

Fig 9.1 Special Function Register Description

The ARM processor core has two interrupt inputs called Interrupt Request (IRQ) and Fast Interrupt request (FIQ). The Vectored Interrupt Controller (VIC) takes 32 interrupt request inputs and program assigns them as FIQ or vectored IRQ types. The programmable assignment scheme means that priorities of interrupts from the various peripherals can be dynamically assigned and adjusted. Vectored IRQ's, which include all interrupt requests that are not classified as FIQs, have a programmable interrupt priority. When more than one interrupt is assigned the same priority and occur simultaneously, the one connected to the lowest numbered VIC channel will be serviced first.

The VIC ORs the requests from all of the vectored IRQs to produce the IRQ signal to the ARM processor. The IRQ service routine can start by reading a register from the VIC and jumping to the address supplied by that register. IRQ is the Interrupt mode for general purpose interrupt handling.

SAMPLE PROGRAM:

```
#include <lpc23xx.h>
_ _irq void IRQ_Handler (void)   //IRQ handler definition
{
        int a=10,b=20,c;
        c=a+b;
        VICSoftIntClr= 0x00004000;
}
int main(void)
{
        EXTMODE= 0x00000001;        //Interrupt configuration, edge triggered
        EXTPOLAR=0x00000001;        //positive going interrupt
        VICIntEnable=0x00004000;    //Enable interrupt
        VICIntSelect=0x00000000;    //set interrupt as IRQ
        VICVectAddr14 = (unsigned long) IRQ_Handler;
        VICSoftInt=0x00004000;      //Software interrupt generation
        while(1)
        { }       //wait here
        return 0;
}
```

OUTPUT:

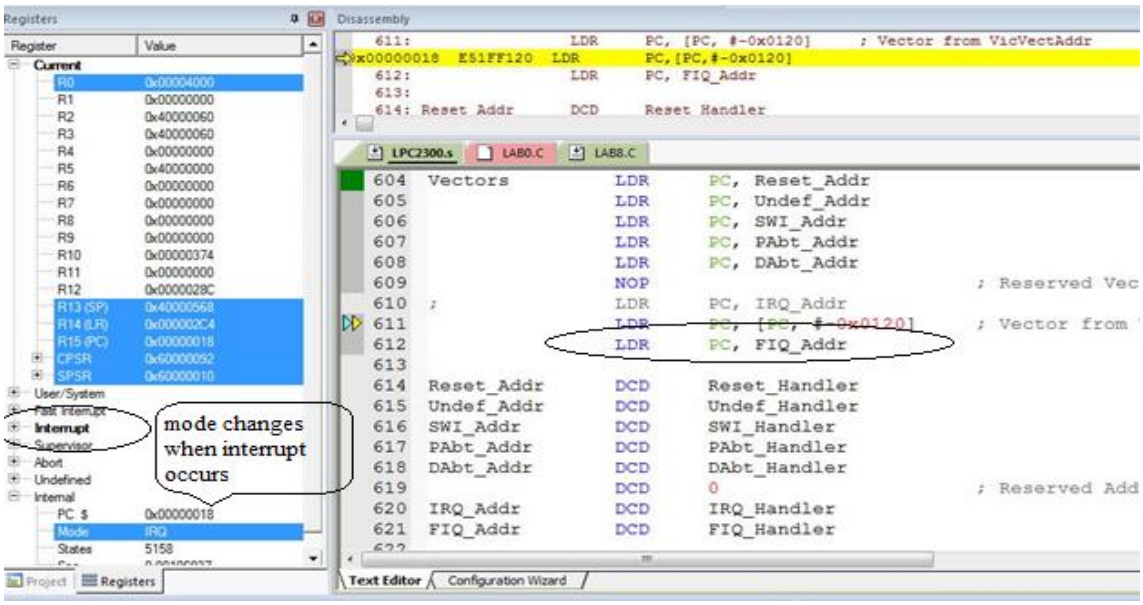


Fig 9.1 Sample Program Execution Window

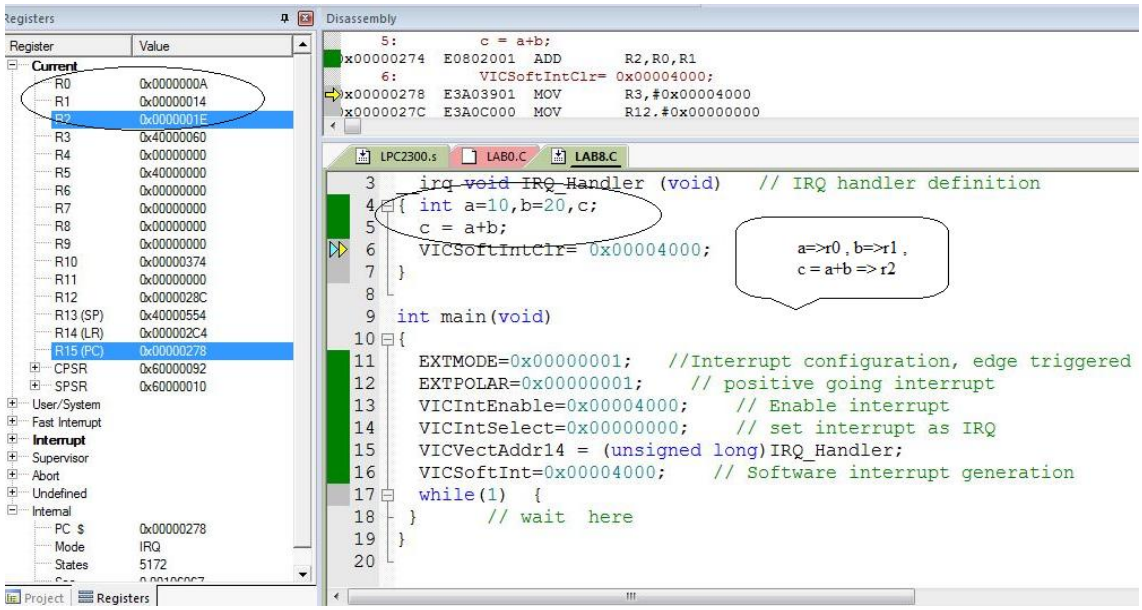


Fig 9.2 Sample Program Output

(ii) To understand FIQ Exception Handling mechanism and its programming

THEORY:

Fast Interrupt request (FIQ) requests have the highest priority. If more than one request is assigned to FIQ, the VIC ORs the requests to produce the FIQ signal to the ARM processor. The fastest possible FIQ latency is achieved when only one request is classified as FIQ, because then the FIQ service routine can simply start dealing with that device. But if more than one request is assigned to the FIQ class, the FIQ service routine can read a word from the VIC that identifies which FIQ source(s) is (are) requesting an interrupt.

SAMPLE PROGRAM:

```
#include <lpc23xx.h>
_ _irq void FIQ_Handler (void)    //FIQ handler definition
{
        int a=10,b=30,c;
        c=a*b;
        VICSoftIntClr= 0x00008000;
}
int main(void)
{
        EXTMODE= 0x00000001;        //Interrupt configuration, edge triggered
        EXTPOLAR=0x00000001;        //positive going interrupt
        VICIntEnable=0x00008000;    //Enable interrupt
        VICIntSelect=0x00008000;    //set interrupt as FIQ
        VICSoftInt=0x00008000;      //Software interrupt generation
        while(1)
        {                           //Wait Here
        }
        return 0;
}
```

OUTPUT:

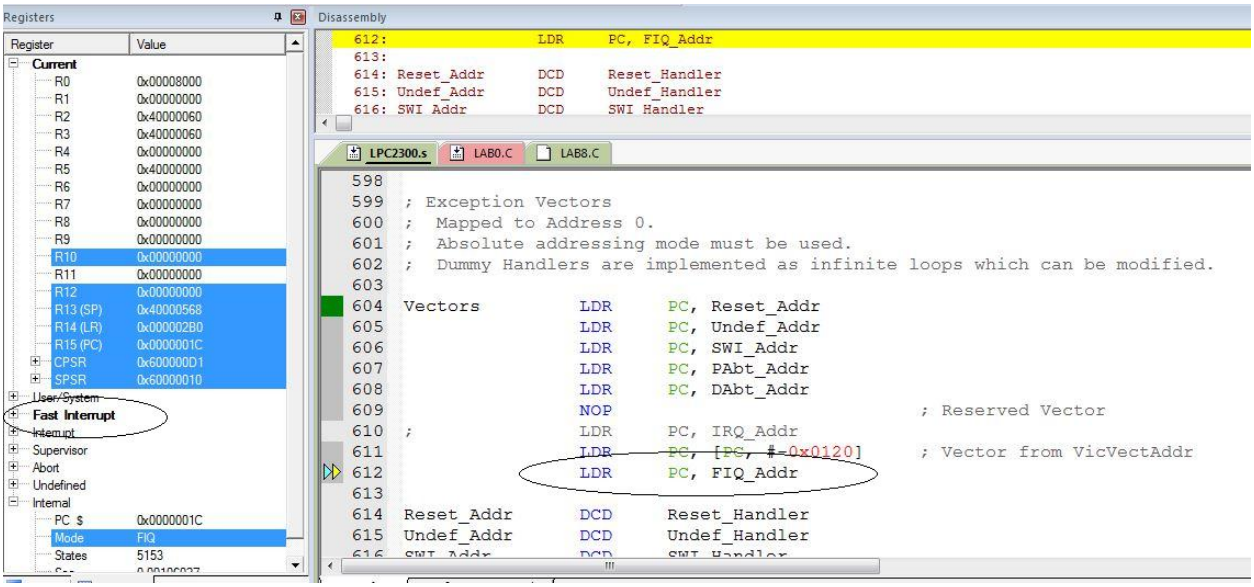


Fig 9.3 Sample Program Execution Window

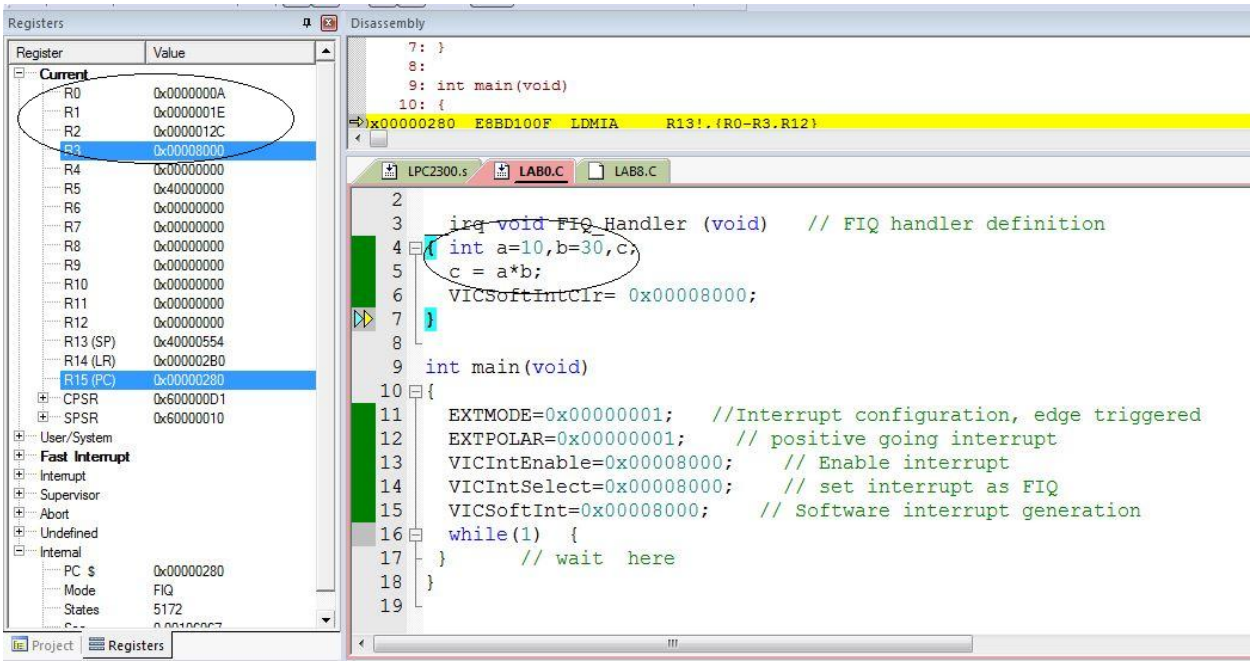


Fig 9.4 Sample Program Output

CONCLUSION:

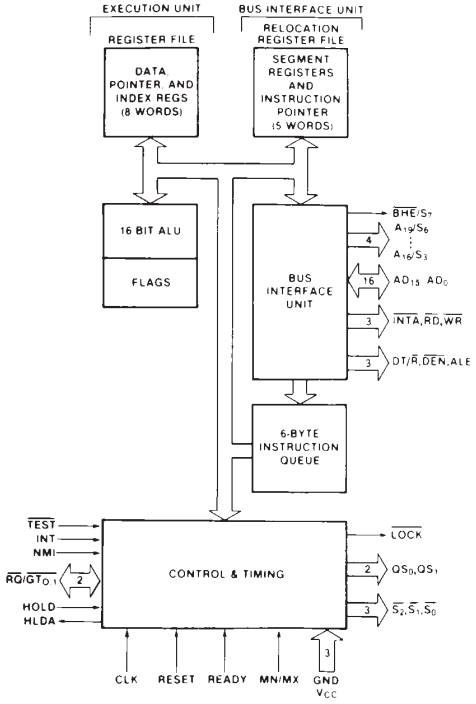
P A R T II

APPENDIX (DATASHEETS)

8086
16-BIT HMOS MICROPROCESSOR
8086/8086-2/8086-1

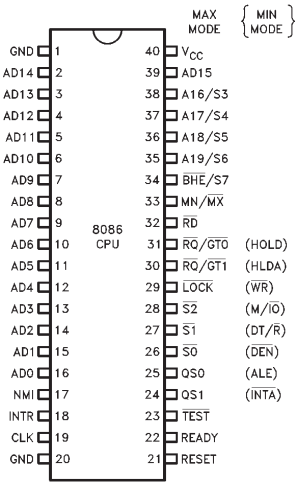
- **Direct Addressing Capability 1 MByte of Memory**
- **Architecture Designed for Powerful Assembly Language and Efficient High Level Languages**
- **14 Word, by 16-Bit Register Set with Symmetrical Operations**
- **24 Operand Addressing Modes**
- **Bit, Byte, Word, and Block Operations**
- **8 and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal Including Multiply and Divide**
- **Range of Clock Rates: 5 MHz for 8086, 8 MHz for 8086-2, 10 MHz for 8086-1**
- **MULTIBUS System Compatible Interface**
- **Available in EXPRESS — Standard Temperature Range — Extended Temperature Range**
- **Available in 40-Lead Cerdip and Plastic Package**
  (See Packaging Spec. Order #231369)

The Intel 8086 high performance 16-bit CPU is available in three clock rates: 5, 8 and 10 MHz. The CPU is implemented in N-Channel, depletion load, silicon gate technology (HMOS-III), and packaged in a 40-pin CERDIP or plastic package. The 8086 operates in both single processor and multiple processor configurations to achieve high performance levels.



231455–1

Figure 1. 8086 CPU Block Diagram



231455–2

40 Lead

Figure 2. 8086 Pin Configuration

Table 1. Pin Description

The following pin function descriptions are for 8086 systems in either minimum or maximum mode. The ''Local Bus'' in these descriptions is the direct multiplexed bus interface connection to the 8086 (without regard to additional bus buffers).

Symbol	Pin No.	Type	Name and Function
AD_{15}–AD_0	2–16, 39	I/O	**ADDRESS DATA BUS:** These lines constitute the time multiplexed memory/IO address (T_1), and data (T_2, T_3, T_W, T_4) bus. A_0 is analogous to $\overline{BHE}$ for the lower byte of the data bus, pins D_7–D_0. It is LOW during T_1 when a byte is to be transferred on the lower portion of the bus in memory or I/O operations. Eight-bit oriented devices tied to the lower half would normally use A_0 to condition chip select functions. (See $\overline{BHE}$.) These lines are active HIGH and float to 3-state OFF during interrupt acknowledge and local bus ''hold acknowledge''.
A_{19}/S_6, A_{18}/S_5, A_{17}/S_4, A_{16}/S_3	35–38	O	**ADDRESS/STATUS:** During T_1 these are the four most significant address lines for memory operations. During I/O operations these lines are LOW. During memory and I/O operations, status information is available on these lines during T_2, T_3, T_W, T_4. The status of the interrupt enable FLAG bit (S_5) is updated at the beginning of each CLK cycle. A_{17}/S_4 and A_{16}/S_3 are encoded as shown. This information indicates which relocation register is presently being used for data accessing. These lines float to 3-state OFF during local bus ''hold acknowledge.''
			A_{17}/S_4 / **A_{16}/S_3** / **Characteristics** 0 (LOW) / 0 / Alternate Data 0 / 1 / Stack 1 (HIGH) / 0 / Code or None 1 / 1 / Data S_6 is 0 (LOW)
$\overline{BHE}/S_7$	34	O	**BUS HIGH ENABLE/STATUS:** During T_1 the bus high enable signal ($\overline{BHE}$) should be used to enable data onto the most significant half of the data bus, pins D_{15}–D_8. Eight-bit oriented devices tied to the upper half of the bus would normally use $\overline{BHE}$ to condition chip select functions. $\overline{BHE}$ is LOW during T_1 for read, write, and interrupt acknowledge cycles when a byte is to be transferred on the high portion of the bus. The S_7 status information is available during T_2, T_3, and T_4. The signal is active LOW, and floats to 3-state OFF in ''hold''. It is LOW during T_1 for the first interrupt acknowledge cycle.
			$\overline{BHE}$ / **A_0** / **Characteristics** 0 / 0 / Whole word 0 / 1 / Upper byte from/to odd address 1 / 0 / Lower byte from/to even address 1 / 1 / None
$\overline{RD}$	32	O	**READ:** Read strobe indicates that the processor is performing a memory or I/O read cycle, depending on the state of the S_2 pin. This signal is used to read devices which reside on the 8086 local bus. $\overline{RD}$ is active LOW during T_2, T_3 and T_W of any read cycle, and is guaranteed to remain HIGH in T_2 until the 8086 local bus has floated. This signal floats to 3-state OFF in ''hold acknowledge''.

Table 1. Pin Description (Continued)

Symbol	Pin No.	Type	Name and Function
READY	22	I	**READY:** is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer. The READY signal from memory/IO is synchronized by the 8284A Clock Generator to form READY. This signal is active HIGH. The 8086 READY input is not synchronized. Correct operation is not guaranteed if the setup and hold times are not met.
INTR	18	I	**INTERRUPT REQUEST:** is a level triggered input which is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt acknowledge operation. A subroutine is vectored to via an interrupt vector lookup table located in system memory. It can be internally masked by software resetting the interrupt enable bit. INTR is internally synchronized. This signal is active HIGH.
$\overline{TEST}$	23	I	**$\overline{TEST}$:** input is examined by the ''Wait'' instruction. If the $\overline{TEST}$ input is LOW execution continues, otherwise the processor waits in an ''Idle'' state. This input is synchronized internally during each clock cycle on the leading edge of CLK.
NMI	17	I	**NON-MASKABLE INTERRUPT:** an edge triggered input which causes a type 2 interrupt. A subroutine is vectored to via an interrupt vector lookup table located in system memory. NMI is not maskable internally by software. A transition from LOW to HIGH initiates the interrupt at the end of the current instruction. This input is internally synchronized.
RESET	21	I	**RESET:** causes the processor to immediately terminate its present activity. The signal must be active HIGH for at least four clock cycles. It restarts execution, as described in the Instruction Set description, when RESET returns LOW. RESET is internally synchronized.
CLK	19	I	**CLOCK:** provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing.
V_{CC}	40		**V_{CC}:** +5V power supply pin.
GND	1, 20		**GROUND**
MN/$\overline{MX}$	33	I	**MINIMUM/MAXIMUM:** indicates what mode the processor is to operate in. The two modes are discussed in the following sections.

The following pin function descriptions are for the 8086/8288 system in maximum mode (i.e., MN/$\overline{MX}$ = V_{SS}). Only the pin functions which are unique to maximum mode are described; all other pin functions are as described above.

$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$	26–28	O	**STATUS:** active during T_4, T_1, and T_2 and is returned to the passive state (1, 1, 1) during T_3 or during T_W when READY is HIGH. This status is used by the 8288 Bus Controller to generate all memory and I/O access control signals. Any change by $\overline{S_2}$, $\overline{S_1}$, or $\overline{S_0}$ during T_4 is used to indicate the beginning of a bus cycle, and the return to the passive state in T_3 or T_W is used to indicate the end of a bus cycle.

Table 1. Pin Description (Continued)

Symbol	Pin No.	Type	Name and Function
$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ (Continued)	26–28	O	These signals float to 3-state OFF in ''hold acknowledge''. These status lines are encoded as shown.

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Characteristics
0 (LOW)	0	0	Interrupt Acknowledge
0	0	1	Read I/O Port
0	1	0	Write I/O Port
0	1	1	Halt
1 (HIGH)	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

$\overline{RQ}/\overline{GT_0}$, $\overline{RQ}/\overline{GT_1}$	30, 31	I/O	**REQUEST/GRANT:** pins are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. Each pin is bidirectional with $\overline{RQ}/\overline{GT_0}$ having higher priority than $\overline{RQ}/\overline{GT_1}$. $\overline{RQ}/\overline{GT}$ pins have internal pull-up resistors and may be left unconnected. The request/grant sequence is as follows (see Page 2-24): 1. A pulse of 1 CLK wide from another local bus master indicates a local bus request (''hold'') to the 8086 (pulse 1). 2. During a T_4 or T_1 clock cycle, a pulse 1 CLK wide from the 8086 to the requesting master (pulse 2), indicates that the 8086 has allowed the local bus to float and that it will enter the ''hold acknowledge'' state at the next CLK. The CPU's bus interface unit is disconnected logically from the local bus during ''hold acknowledge''. 3. A pulse 1 CLK wide from the requesting master indicates to the 8086 (pulse 3) that the ''hold'' request is about to end and that the 8086 can reclaim the local bus at the next CLK. Each master-master exchange of the local bus is a sequence of 3 pulses. There must be one dead CLK cycle after each bus exchange. Pulses are active LOW. If the request is made while the CPU is performing a memory cycle, it will release the local bus during T_4 of the cycle when all the following conditions are met: 1. Request occurs on or before T_2. 2. Current cycle is not the low byte of a word (on an odd address). 3. Current cycle is not the first acknowledge of an interrupt acknowledge sequence. 4. A locked instruction is not currently executing. If the local bus is idle when the request is made the two possible events will follow: 1. Local bus will be released during the next clock. 2. A memory cycle will start within 3 clocks. Now the four rules for a currently active memory cycle apply with condition number 1 already satisfied.
$\overline{LOCK}$	29	O	**$\overline{LOCK}$:** output indicates that other system bus masters are not to gain control of the system bus while $\overline{LOCK}$ is active LOW. The $\overline{LOCK}$ signal is activated by the ''LOCK'' prefix instruction and remains active until the completion of the next instruction. This signal is active LOW, and floats to 3-state OFF in ''hold acknowledge''.

Table 1. Pin Description (Continued)

Symbol	Pin No.	Type	Name and Function		
QS_1, QS_0	24, 25	O	**QUEUE STATUS:** The queue status is valid during the CLK cycle after which the queue operation is performed. QS_1 and QS_0 provide status to allow external tracking of the internal 8086 instruction queue.		
			QS_1	**QS_0**	**Characteristics**
			0 (LOW)	0	No Operation
			0	1	First Byte of Op Code from Queue
			1 (HIGH)	0	Empty the Queue
			1	1	Subsequent Byte from Queue

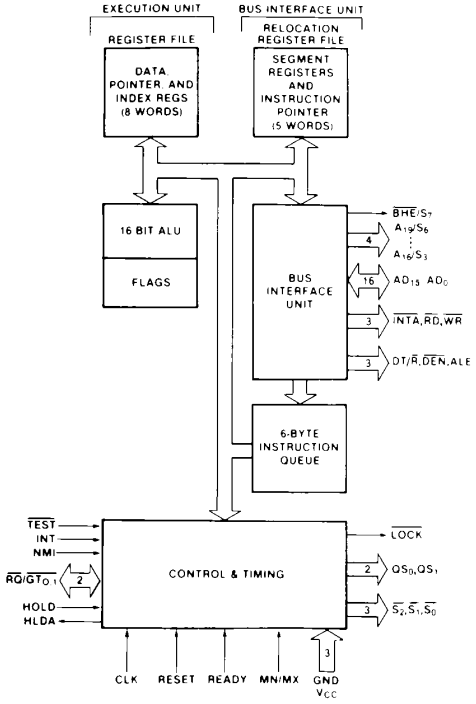
The following pin function descriptions are for the 8086 in minimum mode (i.e., MN/$\overline{MX}$ = V_{CC}). Only the pin functions which are unique to minimum mode are described; all other pin functions are as described above.

M/$\overline{IO}$	28	O	**STATUS LINE:** logically equivalent to S_2 in the maximum mode. It is used to distinguish a memory access from an I/O access. M/$\overline{IO}$ becomes valid in the T_4 preceding a bus cycle and remains valid until the final T_4 of the cycle (M = HIGH, IO = LOW). M/$\overline{IO}$ floats to 3-state OFF in local bus "hold acknowledge".
$\overline{WR}$	29	O	**WRITE:** indicates that the processor is performing a write memory or write I/O cycle, depending on the state of the M/$\overline{IO}$ signal. $\overline{WR}$ is active for T_2, T_3 and T_W of any write cycle. It is active LOW, and floats to 3-state OFF in local bus "hold acknowledge".
$\overline{INTA}$	24	O	**$\overline{INTA}$:** is used as a read strobe for interrupt acknowledge cycles. It is active LOW during T_2, T_3 and T_W of each interrupt acknowledge cycle.
ALE	25	O	**ADDRESS LATCH ENABLE:** provided by the processor to latch the address into the 8282/8283 address latch. It is a HIGH pulse active during T_1 of any bus cycle. Note that ALE is never floated.
DT/$\overline{R}$	27	O	**DATA TRANSMIT/RECEIVE:** needed in minimum system that desires to use an 8286/8287 data bus transceiver. It is used to control the direction of data flow through the transceiver. Logically DT/$\overline{R}$ is equivalent to $\overline{S_1}$ in the maximum mode, and its timing is the same as for M/$\overline{IO}$. (T = HIGH, R = LOW.) This signal floats to 3-state OFF in local bus "hold acknowledge".
$\overline{DEN}$	26	O	**DATA ENABLE:** provided as an output enable for the 8286/8287 in a minimum system which uses the transceiver. $\overline{DEN}$ is active LOW during each memory and I/O access and for INTA cycles. For a read or $\overline{INTA}$ cycle it is active from the middle of T_2 until the middle of T_4, while for a write cycle it is active from the beginning of T_2 until the middle of T_4. $\overline{DEN}$ floats to 3-state OFF in local bus "hold acknowledge".
HOLD, HLDA	31, 30	I/O	**HOLD:** indicates that another master is requesting a local bus "hold." To be acknowledged, HOLD must be active HIGH. The processor receiving the "hold" request will issue HLDA (HIGH) as an acknowledgement in the middle of a T_4 or T_i clock cycle. Simultaneous with the issuance of HLDA the processor will float the local bus and control lines. After HOLD is detected as being LOW, the processor will LOWer the HLDA, and when the processor needs to run another cycle, it will again drive the local bus and control lines. Hold acknowledge (HLDA) and HOLD have internal pull-up resistors. The same rules as for $\overline{RQ}$/$\overline{GT}$ apply regarding when the local bus will be released. HOLD is not an asynchronous input. External synchronization should be provided if the system cannot otherwise guarantee the setup time.

8086
16-BIT HMOS MICROPROCESSOR
8086/8086-2/8086-1

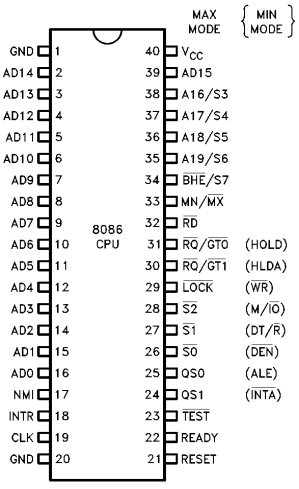
- **Direct Addressing Capability 1 MByte of Memory**
- **Architecture Designed for Powerful Assembly Language and Efficient High Level Languages**
- **14 Word, by 16-Bit Register Set with Symmetrical Operations**
- **24 Operand Addressing Modes**
- **Bit, Byte, Word, and Block Operations**
- **8 and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal Including Multiply and Divide**
- **Range of Clock Rates: 5 MHz for 8086, 8 MHz for 8086-2, 10 MHz for 8086-1**
- **MULTIBUS System Compatible Interface**
- **Available in EXPRESS — Standard Temperature Range — Extended Temperature Range**
- **Available in 40-Lead Cerdip and Plastic Package**
  (See Packaging Spec. Order #231369)

The Intel 8086 high performance 16-bit CPU is available in three clock rates: 5, 8 and 10 MHz. The CPU is implemented in N-Channel, depletion load, silicon gate technology (HMOS-III), and packaged in a 40-pin CERDIP or plastic package. The 8086 operates in both single processor and multiple processor configurations to achieve high performance levels.



231455–1

Figure 1. 8086 CPU Block Diagram



231455–2

40 Lead

Figure 2. 8086 Pin Configuration

Table 1. Pin Description

The following pin function descriptions are for 8086 systems in either minimum or maximum mode. The ''Local Bus'' in these descriptions is the direct multiplexed bus interface connection to the 8086 (without regard to additional bus buffers).

Symbol	Pin No.	Type	Name and Function
AD_{15}–AD_0	2–16, 39	I/O	**ADDRESS DATA BUS:** These lines constitute the time multiplexed memory/IO address (T_1), and data (T_2, T_3, T_W, T_4) bus. A_0 is analogous to $\overline{BHE}$ for the lower byte of the data bus, pins D_7–D_0. It is LOW during T_1 when a byte is to be transferred on the lower portion of the bus in memory or I/O operations. Eight-bit oriented devices tied to the lower half would normally use A_0 to condition chip select functions. (See $\overline{BHE}$.) These lines are active HIGH and float to 3-state OFF during interrupt acknowledge and local bus ''hold acknowledge''.
A_{19}/S_6, A_{18}/S_5, A_{17}/S_4, A_{16}/S_3	35–38	O	**ADDRESS/STATUS:** During T_1 these are the four most significant address lines for memory operations. During I/O operations these lines are LOW. During memory and I/O operations, status information is available on these lines during T_2, T_3, T_W, T_4. The status of the interrupt enable FLAG bit (S_5) is updated at the beginning of each CLK cycle. A_{17}/S_4 and A_{16}/S_3 are encoded as shown. This information indicates which relocation register is presently being used for data accessing. These lines float to 3-state OFF during local bus ''hold acknowledge.''
			A_{17}/S_4 / **A_{16}/S_3** / **Characteristics** 0 (LOW) / 0 / Alternate Data 0 / 1 / Stack 1 (HIGH) / 0 / Code or None 1 / 1 / Data S_6 is 0 (LOW)
$\overline{BHE}/S_7$	34	O	**BUS HIGH ENABLE/STATUS:** During T_1 the bus high enable signal ($\overline{BHE}$) should be used to enable data onto the most significant half of the data bus, pins D_{15}–D_8. Eight-bit oriented devices tied to the upper half of the bus would normally use $\overline{BHE}$ to condition chip select functions. $\overline{BHE}$ is LOW during T_1 for read, write, and interrupt acknowledge cycles when a byte is to be transferred on the high portion of the bus. The S_7 status information is available during T_2, T_3, and T_4. The signal is active LOW, and floats to 3-state OFF in ''hold''. It is LOW during T_1 for the first interrupt acknowledge cycle.
			$\overline{BHE}$ / **A_0** / **Characteristics** 0 / 0 / Whole word 0 / 1 / Upper byte from/to odd address 1 / 0 / Lower byte from/to even address 1 / 1 / None
$\overline{RD}$	32	O	**READ:** Read strobe indicates that the processor is performing a memory or I/O read cycle, depending on the state of the S_2 pin. This signal is used to read devices which reside on the 8086 local bus. $\overline{RD}$ is active LOW during T_2, T_3 and T_W of any read cycle, and is guaranteed to remain HIGH in T_2 until the 8086 local bus has floated. This signal floats to 3-state OFF in ''hold acknowledge''.

Table 1. Pin Description (Continued)

Symbol	Pin No.	Type	Name and Function
READY	22	I	**READY:** is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer. The READY signal from memory/IO is synchronized by the 8284A Clock Generator to form READY. This signal is active HIGH. The 8086 READY input is not synchronized. Correct operation is not guaranteed if the setup and hold times are not met.
INTR	18	I	**INTERRUPT REQUEST:** is a level triggered input which is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt acknowledge operation. A subroutine is vectored to via an interrupt vector lookup table located in system memory. It can be internally masked by software resetting the interrupt enable bit. INTR is internally synchronized. This signal is active HIGH.
$\overline{TEST}$	23	I	**$\overline{TEST}$:** input is examined by the ''Wait'' instruction. If the $\overline{TEST}$ input is LOW execution continues, otherwise the processor waits in an ''Idle'' state. This input is synchronized internally during each clock cycle on the leading edge of CLK.
NMI	17	I	**NON-MASKABLE INTERRUPT:** an edge triggered input which causes a type 2 interrupt. A subroutine is vectored to via an interrupt vector lookup table located in system memory. NMI is not maskable internally by software. A transition from LOW to HIGH initiates the interrupt at the end of the current instruction. This input is internally synchronized.
RESET	21	I	**RESET:** causes the processor to immediately terminate its present activity. The signal must be active HIGH for at least four clock cycles. It restarts execution, as described in the Instruction Set description, when RESET returns LOW. RESET is internally synchronized.
CLK	19	I	**CLOCK:** provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing.
V_{CC}	40		**V_{CC}:** +5V power supply pin.
GND	1, 20		**GROUND**
MN/$\overline{MX}$	33	I	**MINIMUM/MAXIMUM:** indicates what mode the processor is to operate in. The two modes are discussed in the following sections.

The following pin function descriptions are for the 8086/8288 system in maximum mode (i.e., MN/$\overline{MX}$ = V_{SS}). Only the pin functions which are unique to maximum mode are described; all other pin functions are as described above.

$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$	26–28	O	**STATUS:** active during T_4, T_1, and T_2 and is returned to the passive state (1, 1, 1) during T_3 or during T_W when READY is HIGH. This status is used by the 8288 Bus Controller to generate all memory and I/O access control signals. Any change by $\overline{S_2}$, $\overline{S_1}$, or $\overline{S_0}$ during T_4 is used to indicate the beginning of a bus cycle, and the return to the passive state in T_3 or T_W is used to indicate the end of a bus cycle.

Table 1. Pin Description (Continued)

Symbol	Pin No.	Type	Name and Function
$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ (Continued)	26–28	O	These signals float to 3-state OFF in ''hold acknowledge''. These status lines are encoded as shown.

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Characteristics
0 (LOW)	0	0	Interrupt Acknowledge
0	0	1	Read I/O Port
0	1	0	Write I/O Port
0	1	1	Halt
1 (HIGH)	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

$\overline{RQ}/\overline{GT_0}$, $\overline{RQ}/\overline{GT_1}$	30, 31	I/O	**REQUEST/GRANT:** pins are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. Each pin is bidirectional with $\overline{RQ}/\overline{GT_0}$ having higher priority than $\overline{RQ}/\overline{GT_1}$. $\overline{RQ}/\overline{GT}$ pins have internal pull-up resistors and may be left unconnected. The request/grant sequence is as follows (see Page 2-24): 1. A pulse of 1 CLK wide from another local bus master indicates a local bus request (''hold'') to the 8086 (pulse 1). 2. During a T_4 or T_1 clock cycle, a pulse 1 CLK wide from the 8086 to the requesting master (pulse 2), indicates that the 8086 has allowed the local bus to float and that it will enter the ''hold acknowledge'' state at the next CLK. The CPU's bus interface unit is disconnected logically from the local bus during ''hold acknowledge''. 3. A pulse 1 CLK wide from the requesting master indicates to the 8086 (pulse 3) that the ''hold'' request is about to end and that the 8086 can reclaim the local bus at the next CLK. Each master-master exchange of the local bus is a sequence of 3 pulses. There must be one dead CLK cycle after each bus exchange. Pulses are active LOW. If the request is made while the CPU is performing a memory cycle, it will release the local bus during T_4 of the cycle when all the following conditions are met: 1. Request occurs on or before T_2. 2. Current cycle is not the low byte of a word (on an odd address). 3. Current cycle is not the first acknowledge of an interrupt acknowledge sequence. 4. A locked instruction is not currently executing. If the local bus is idle when the request is made the two possible events will follow: 1. Local bus will be released during the next clock. 2. A memory cycle will start within 3 clocks. Now the four rules for a currently active memory cycle apply with condition number 1 already satisfied.
$\overline{LOCK}$	29	O	**$\overline{LOCK}$:** output indicates that other system bus masters are not to gain control of the system bus while $\overline{LOCK}$ is active LOW. The $\overline{LOCK}$ signal is activated by the ''LOCK'' prefix instruction and remains active until the completion of the next instruction. This signal is active LOW, and floats to 3-state OFF in ''hold acknowledge''.

Table 1. Pin Description (Continued)

Symbol	Pin No.	Type	Name and Function		
QS_1, QS_0	24, 25	O	**QUEUE STATUS:** The queue status is valid during the CLK cycle after which the queue operation is performed. QS_1 and QS_0 provide status to allow external tracking of the internal 8086 instruction queue.		
			QS_1	**QS_0**	**Characteristics**
			0 (LOW)	0	No Operation
			0	1	First Byte of Op Code from Queue
			1 (HIGH)	0	Empty the Queue
			1	1	Subsequent Byte from Queue

The following pin function descriptions are for the 8086 in minimum mode (i.e., MN/$\overline{MX}$ = V_{CC}). Only the pin functions which are unique to minimum mode are described; all other pin functions are as described above.

M/$\overline{IO}$	28	O	**STATUS LINE:** logically equivalent to S_2 in the maximum mode. It is used to distinguish a memory access from an I/O access. M/$\overline{IO}$ becomes valid in the T_4 preceding a bus cycle and remains valid until the final T_4 of the cycle (M = HIGH, IO = LOW). M/$\overline{IO}$ floats to 3-state OFF in local bus "hold acknowledge".
$\overline{WR}$	29	O	**WRITE:** indicates that the processor is performing a write memory or write I/O cycle, depending on the state of the M/$\overline{IO}$ signal. $\overline{WR}$ is active for T_2, T_3 and T_W of any write cycle. It is active LOW, and floats to 3-state OFF in local bus "hold acknowledge".
$\overline{INTA}$	24	O	**$\overline{INTA}$:** is used as a read strobe for interrupt acknowledge cycles. It is active LOW during T_2, T_3 and T_W of each interrupt acknowledge cycle.
ALE	25	O	**ADDRESS LATCH ENABLE:** provided by the processor to latch the address into the 8282/8283 address latch. It is a HIGH pulse active during T_1 of any bus cycle. Note that ALE is never floated.
DT/$\overline{R}$	27	O	**DATA TRANSMIT/RECEIVE:** needed in minimum system that desires to use an 8286/8287 data bus transceiver. It is used to control the direction of data flow through the transceiver. Logically DT/$\overline{R}$ is equivalent to $\overline{S_1}$ in the maximum mode, and its timing is the same as for M/$\overline{IO}$. (T = HIGH, R = LOW.) This signal floats to 3-state OFF in local bus "hold acknowledge".
$\overline{DEN}$	26	O	**DATA ENABLE:** provided as an output enable for the 8286/8287 in a minimum system which uses the transceiver. $\overline{DEN}$ is active LOW during each memory and I/O access and for INTA cycles. For a read or $\overline{INTA}$ cycle it is active from the middle of T_2 until the middle of T_4, while for a write cycle it is active from the beginning of T_2 until the middle of T_4. $\overline{DEN}$ floats to 3-state OFF in local bus "hold acknowledge".
HOLD, HLDA	31, 30	I/O	**HOLD:** indicates that another master is requesting a local bus "hold." To be acknowledged, HOLD must be active HIGH. The processor receiving the "hold" request will issue HLDA (HIGH) as an acknowledgement in the middle of a T_4 or T_i clock cycle. Simultaneous with the issuance of HLDA the processor will float the local bus and control lines. After HOLD is detected as being LOW, the processor will LOWer the HLDA, and when the processor needs to run another cycle, it will again drive the local bus and control lines. Hold acknowledge (HLDA) and HOLD have internal pull-up resistors. The same rules as for $\overline{RQ}$/$\overline{GT}$ apply regarding when the local bus will be released. HOLD is not an asynchronous input. External synchronization should be provided if the system cannot otherwise guarantee the setup time.

FUNCTIONAL DESCRIPTION

General Operation

The internal functions of the 8086 processor are partitioned logically into two processing units. The first is the Bus Interface Unit (BIU) and the second is the Execution Unit (EU) as shown in the block diagram of Figure 1.

These units can interact directly but for the most part perform as separate asynchronous operational processors. The bus interface unit provides the functions related to instruction fetching and queuing, operand fetch and store, and address relocation. This unit also provides the basic bus control. The overlap of instruction pre-fetching provided by this unit serves to increase processor performance through improved bus bandwidth utilization. Up to 6 bytes of the instruction stream can be queued while waiting for decoding and execution.

The instruction stream queuing mechanism allows the BIU to keep the memory utilized very efficiently. Whenever there is space for at least 2 bytes in the queue, the BIU will attempt a word fetch memory cycle. This greatly reduces ''dead time'' on the memory bus. The queue acts as a First-In-First-Out (FIFO) buffer, from which the EU extracts instruction bytes as required. If the queue is empty (following a branch instruction, for example), the first byte into the queue immediately becomes available to the EU.

The execution unit receives pre-fetched instructions from the BIU queue and provides un-relocated operand addresses to the BIU. Memory operands are passed through the BIU for processing by the EU, which passes results to the BIU for storage. See the Instruction Set description for further register set and architectural descriptions.

MEMORY ORGANIZATION

The processor provides a 20-bit address to memory which locates the byte being referenced. The memory is organized as a linear array of up to 1 million bytes, addressed as 00000(H) to FFFFF(H). The memory is logically divided into code, data, extra data, and stack segments of up to 64K bytes each, with each segment falling on 16-byte boundaries. (See Figure 3a.)

All memory references are made relative to base addresses contained in high speed segment registers. The segment types were chosen based on the addressing needs of programs. The segment register to be selected is automatically chosen according to the rules of the following table. All information in one segment type share the same logical attributes (e.g. code or data). By structuring memory into relocatable areas of similar characteristics and by automatically selecting segment registers, programs are shorter, faster, and more structured.

Word (16-bit) operands can be located on even or odd address boundaries and are thus not constrained to even boundaries as is the case in many 16-bit computers. For address and data operands, the least significant byte of the word is stored in the lower valued address location and the most significant byte in the next higher address location. The BIU automatically performs the proper number of memory accesses, one if the word operand is on an even byte boundary and two if it is on an odd byte boundary. Except for the performance penalty, this double access is transparent to the software. This performance penalty does not occur for instruction fetches, only word operands.

Physically, the memory is organized as a high bank (D_{15}–D_8) and a low bank (D_7–D_0) of 512K 8-bit bytes addressed in parallel by the processor's address lines A_{19}–A_1. Byte data with even addresses is transferred on the D_7–D_0 bus lines while odd addressed byte data (A_0 HIGH) is transferred on the D_{15}–D_8 bus lines. The processor provides two enable signals, $\overline{BHE}$ and A_0, to selectively allow reading from or writing into either an odd byte location, even byte location, or both. The instruction stream is fetched from memory as words and is addressed internally by the processor to the byte level as necessary.

Memory Reference Need	Segment Register Used	Segment Selection Rule
Instructions	CODE (CS)	Automatic with all instruction prefetch.
Stack	STACK (SS)	All stack pushes and pops. Memory references relative to BP base register except data references.
Local Data	DATA (DS)	Data references when: relative to stack, destination of string operation, or explicitly overridden.
External (Global) Data	EXTRA (ES)	Destination of string operations: explicitly selected using a segment override.

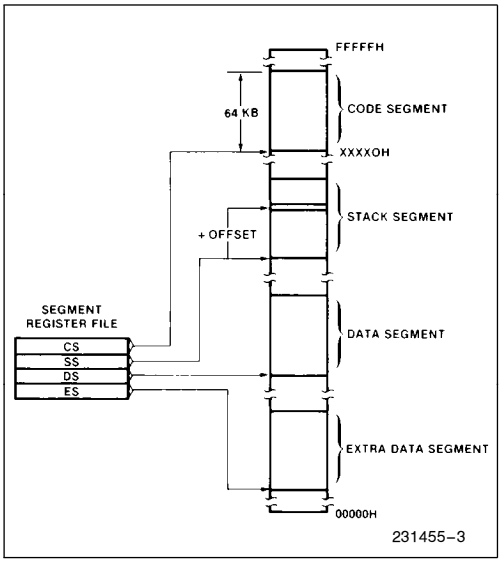


Figure 3a. Memory Organization

In referencing word data the BIU requires one or two memory cycles depending on whether or not the starting byte of the word is on an even or odd address, respectively. Consequently, in referencing word operands performance can be optimized by locating data on even address boundaries. This is an especially useful technique for using the stack, since odd address references to the stack may adversely affect the context switching time for interrupt processing or task multiplexing.

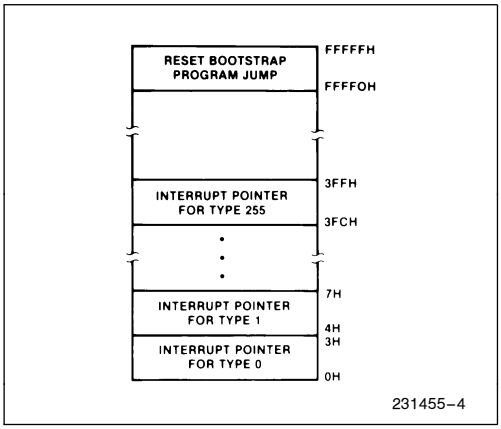


Figure 3b. Reserved Memory Locations

Certain locations in memory are reserved for specific CPU operations (see Figure 3b). Locations from address FFFF0H through FFFFFH are reserved for operations including a jump to the initial program loading routine. Following RESET, the CPU will always begin execution at location FFFF0H where the jump must be. Locations 00000H through 003FFH are reserved for interrupt operations. Each of the 256 possible interrupt types has its service routine pointed to by a 4-byte pointer element consisting of a 16-bit segment address and a 16-bit offset address. The pointer elements are assumed to have been stored at the respective places in reserved memory prior to occurrence of interrupts.

MINIMUM AND MAXIMUM MODES

The requirements for supporting minimum and maximum 8086 systems are sufficiently different that they cannot be done efficiently with 40 uniquely defined pins. Consequently, the 8086 is equipped with a strap pin (MN/$\overline{MX}$) which defines the system configuration. The definition of a certain subset of the pins changes dependent on the condition of the strap pin. When MN/$\overline{MX}$ pin is strapped to GND, the 8086 treats pins 24 through 31 in maximum mode. An 8288 bus controller interprets status information coded into $\overline{S_0}$, $\overline{S_2}$, $\overline{S_2}$ to generate bus timing and control signals compatible with the MULTIBUS architecture. When the MN/$\overline{MX}$ pin is strapped to V_{CC}, the 8086 generates bus control signals itself on pins 24 through 31, as shown in parentheses in Figure 2. Examples of minimum mode and maximum mode systems are shown in Figure 4.

BUS OPERATION

The 8086 has a combined address and data bus commonly referred to as a time multiplexed bus. This technique provides the most efficient use of pins on the processor while permitting the use of a standard 40-lead package. This "local bus" can be buffered directly and used throughout the system with address latching provided on memory and I/O modules. In addition, the bus can also be demultiplexed at the processor with a single set of address latches if a standard non-multiplexed bus is desired for the system.

Each processor bus cycle consists of at least four CLK cycles. These are referred to as T_1, T_2, T_3 and T_4 (see Figure 5). The address is emitted from the processor during T_1 and data transfer occurs on the bus during T_3 and T_4. T_2 is used primarily for changing the direction of the bus during read operations. In the event that a "NOT READY" indication is given by the addressed device, "Wait" states (T_W) are inserted between T_3 and T_4. Each inserted "Wait" state is of the same duration as a CLK cycle. Periods

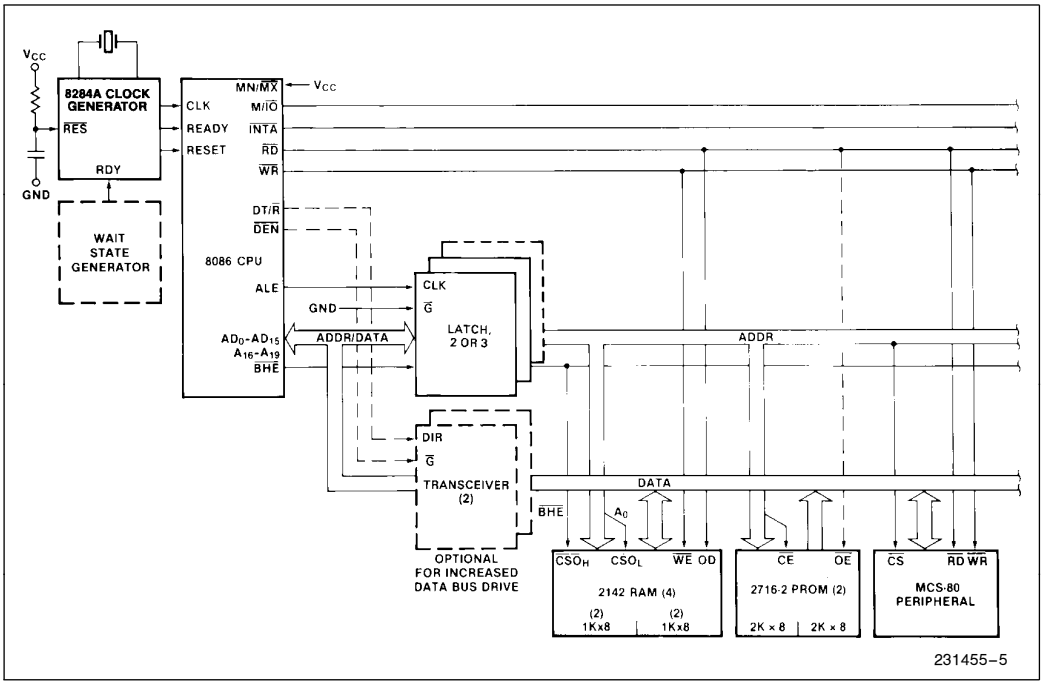


Figure 4a. Minimum Mode 8086 Typical Configuration

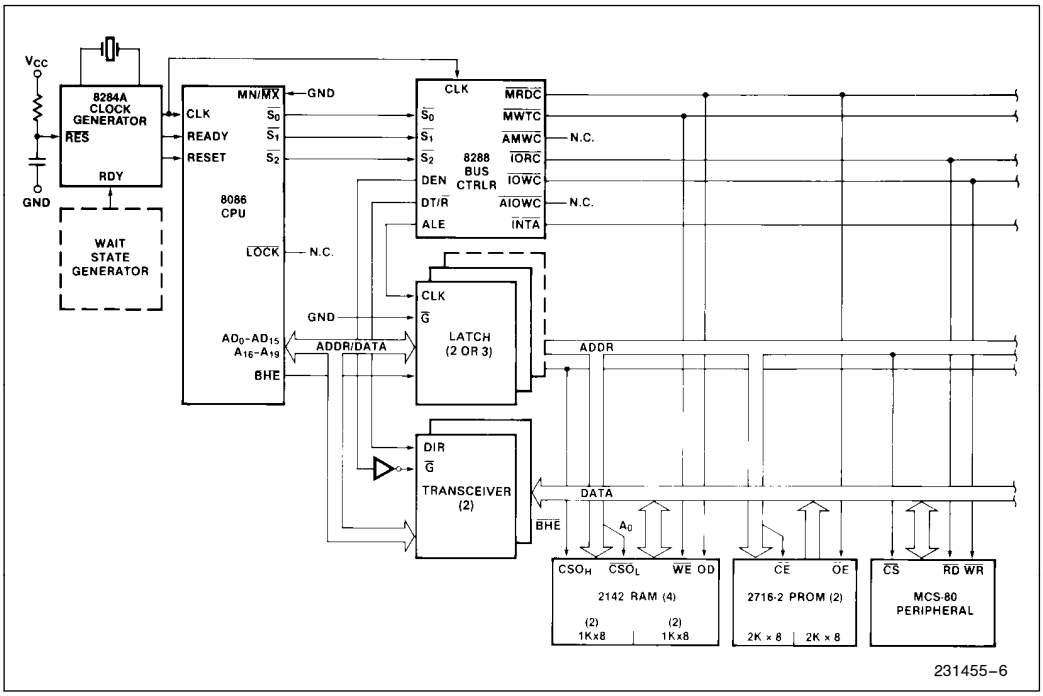


Figure 4b. Maximum Mode 8086 Typical Configuration

can occur between 8086 bus cycles. These are referred to as ''Idle'' states (T_i) or inactive CLK cycles. The processor uses these cycles for internal housekeeping.

During T_1 of any bus cycle the ALE (Address Latch Enable) signal is emitted (by either the processor or the 8288 bus controller, depending on the MN/$\overline{MX}$ strap). At the trailing edge of this pulse, a valid address and certain status information for the cycle may be latched.

Status bits $\overline{S_0}$, $\overline{S_1}$, and $\overline{S_2}$ are used, in maximum mode, by the bus controller to identify the type of bus transaction according to the following table:

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Characteristics
0 (LOW)	0	0	Interrupt Acknowledge
0	0	1	Read I/O
0	1	0	Write I/O
0	1	1	Halt
1 (HIGH)	0	0	Instruction Fetch
1	0	1	Read Data from Memory
1	1	0	Write Data to Memory
1	1	1	Passive (no bus cycle)

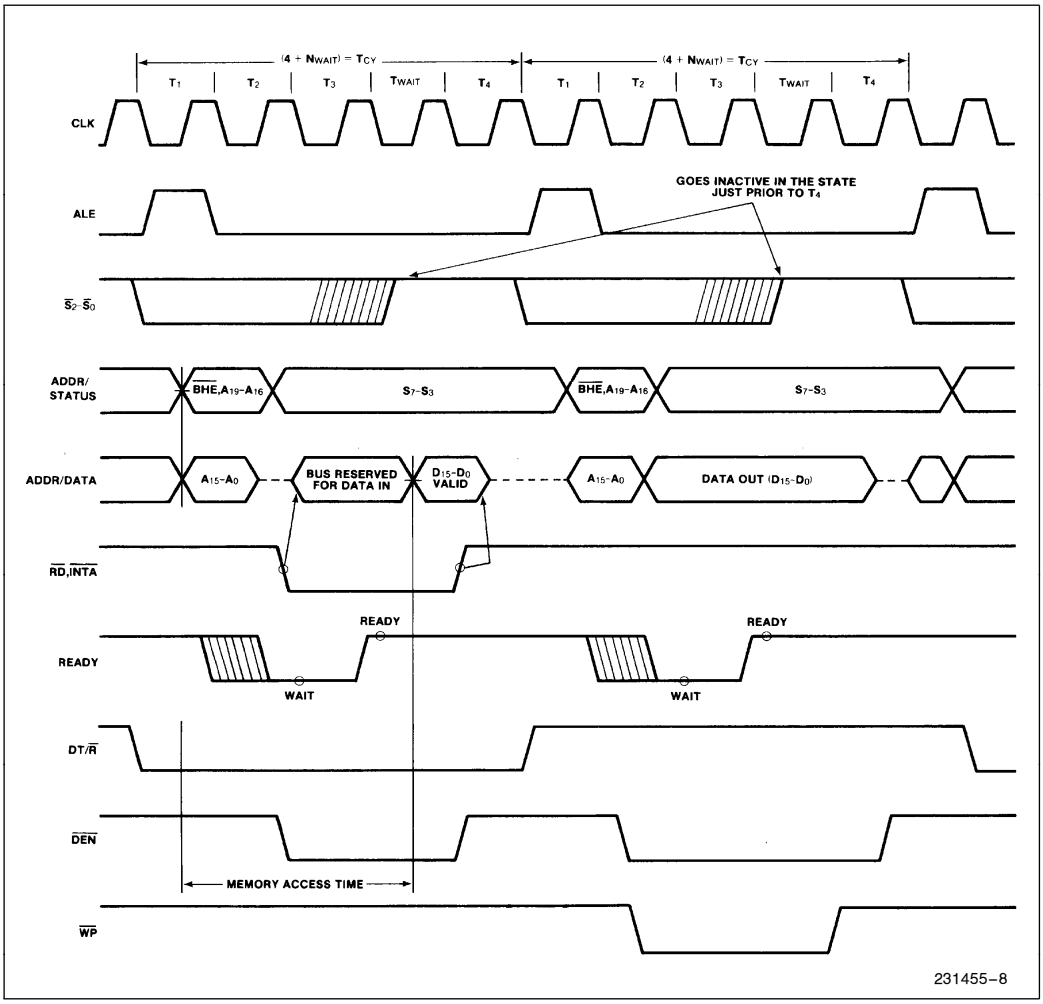


Figure 5. Basic System Timing

Status bits S_3 through S_7 are multiplexed with high-order address bits and the $\overline{BHE}$ signal, and are therefore valid during T_2 through T_4. S_3 and S_4 indicate which segment register (see Instruction Set description) was used for this bus cycle in forming the address, according to the following table:

S_4	S_3	Characteristics
0 (LOW)	0	Alternate Data (extra segment)
0	1	Stack
1 (HIGH)	0	Code or None
1	1	Data

S_5 is a reflection of the PSW interrupt enable bit. $S_6 = 0$ and S_7 is a spare status bit.

I/O ADDRESSING

In the 8086, I/O operations can address up to a maximum of 64K I/O byte registers or 32K I/O word registers. The I/O address appears in the same format as the memory address on bus lines A_{15}–A_0. The address lines A_{19}–A_{16} are zero in I/O operations. The variable I/O instructions which use register DX as a pointer have full address capability while the direct I/O instructions directly address one or two of the 256 I/O byte locations in page 0 of the I/O address space.

I/O ports are addressed in the same manner as memory locations. Even addressed bytes are transferred on the D_7–D_0 bus lines and odd addressed bytes on D_{15}–D_8. Care must be taken to assure that each register within an 8-bit peripheral located on the lower portion of the bus be addressed as even.

External Interface

PROCESSOR RESET AND INITIALIZATION

Processor initialization or start up is accomplished with activation (HIGH) of the RESET pin. The 8086 RESET is required to be HIGH for greater than 4 CLK cycles. The 8086 will terminate operations on the high-going edge of RESET and will remain dormant as long as RESET is HIGH. The low-going transition of RESET triggers an internal reset sequence for approximately 10 CLK cycles. After this interval the 8086 operates normally beginning with the instruction in absolute location FFFF0H (see Figure 3b). The details of this operation are specified in the Instruction Set description of the MCS-86 Family User's Manual. The RESET input is internally synchronized to the processor clock. At initialization the HIGH-to-LOW transition of RESET must occur no sooner than 50 μs after power-up, to allow complete initialization of the 8086.

NMI asserted prior to the 2nd clock after the end of RESET will not be honored. If NMI is asserted after that point and during the internal reset sequence, the processor may execute one instruction before responding to the interrupt. A hold request active immediately after RESET will be honored before the first instruction fetch.

All 3-state outputs float to 3-state OFF during RESET. Status is active in the idle state for the first clock after RESET becomes active and then floats to 3-state OFF. ALE and HLDA are driven low.

INTERRUPT OPERATIONS

Interrupt operations fall into two classes; software or hardware initiated. The software initiated interrupts and software aspects of hardware interrupts are specified in the Instruction Set description. Hardware interrupts can be classified as non-maskable or maskable.

Interrupts result in a transfer of control to a new program location. A 256-element table containing address pointers to the interrupt service program locations resides in absolute locations 0 through 3FFH (see Figure 3b), which are reserved for this purpose. Each element in the table is 4 bytes in size and corresponds to an interrupt ''type''. An interrupting device supplies an 8-bit type number, during the interrupt acknowledge sequence, which is used to ''vector'' through the appropriate element to the new interrupt service program location.

NON-MASKABLE INTERRUPT (NMI)

The processor provides a single non-maskable interrupt pin (NMI) which has higher priority than the maskable interrupt request pin (INTR). A typical use would be to activate a power failure routine. The NMI is edge-triggered on a LOW-to-HIGH transition. The activation of this pin causes a type 2 interrupt. (See Instruction Set description.)

NMI is required to have a duration in the HIGH state of greater than two CLK cycles, but is not required to be synchronized to the clock. Any high-going transition of NMI is latched on-chip and will be serviced at the end of the current instruction or between whole moves of a block-type instruction. Worst case response to NMI would be for multiply, divide, and variable shift instructions. There is no specification on the occurrence of the low-going edge; it may occur before, during, or after the servicing of NMI. Another high-going edge triggers another response if it occurs after the start of the NMI procedure. The signal must be free of logical spikes in general and be free of bounces on the low-going edge to avoid triggering extraneous responses.

MASKABLE INTERRUPT (INTR)

The 8086 provides a single interrupt request input (INTR) which can be masked internally by software with the resetting of the interrupt enable FLAG status bit. The interrupt request signal is level triggered. It is internally synchronized during each clock cycle on the high-going edge of CLK. To be responded to, INTR must be present (HIGH) during the clock period preceding the end of the current instruction or the end of a whole move for a block-type instruction. During the interrupt response sequence further interrupts are disabled. The enable bit is reset as part of the response to any interrupt (INTR, NMI, software interrupt or single-step), although the FLAGS register which is automatically pushed onto the stack reflects the state of the processor prior to the interrupt. Until the old FLAGS register is restored the enable bit will be zero unless specifically set by an instruction.

During the response sequence (Figure 6) the processor executes two successive (back-to-back) interrupt acknowledge cycles. The 8086 emits the LOCK signal from T_2 of the first bus cycle until T_2 of the second. A local bus "hold" request will not be honored until the end of the second bus cycle. In the second bus cycle a byte is fetched from the external interrupt system (e.g., 8259A PIC) which identifies the source (type) of the interrupt. This byte is multiplied by four and used as a pointer into the interrupt vector lookup table. An INTR signal left HIGH will be continually responded to within the limitations of the enable bit and sample period. The INTERRUPT RETURN instruction includes a FLAGS pop which returns the status of the original interrupt enable bit when it restores the FLAGS.

HALT

When a software "HALT" instruction is executed the processor indicates that it is entering the "HALT" state in one of two ways depending upon which mode is strapped. In minimum mode, the processor issues one ALE with no qualifying bus control signals. In maximum mode, the processor issues appropriate HALT status on $\overline{S}_2$, $\overline{S}_1$, and $\overline{S}_0$; and the 8288 bus controller issues one ALE. The 8086 will not leave the "HALT" state when a local bus "hold" is entered while in "HALT". In this case, the processor reissues the HALT indicator. An interrupt request or RESET will force the 8086 out of the "HALT" state.

READ/MODIFY/WRITE (SEMAPHORE) OPERATIONS VIA LOCK

The $\overline{LOCK}$ status information is provided by the processor when directly consecutive bus cycles are required during the execution of an instruction. This provides the processor with the capability of performing read/modify/write operations on memory (via the Exchange Register With Memory instruction, for example) without the possibility of another system bus master receiving intervening memory cycles. This is useful in multi-processor system configurations to accomplish "test and set lock" operations. The $\overline{LOCK}$ signal is activated (forced LOW) in the clock cycle following the one in which the software "LOCK" prefix instruction is decoded by the EU. It is deactivated at the end of the last bus cycle of the instruction following the "LOCK" prefix instruction. While $\overline{LOCK}$ is active a request on a RQ/GT pin will be recorded and then honored at the end of the LOCK.

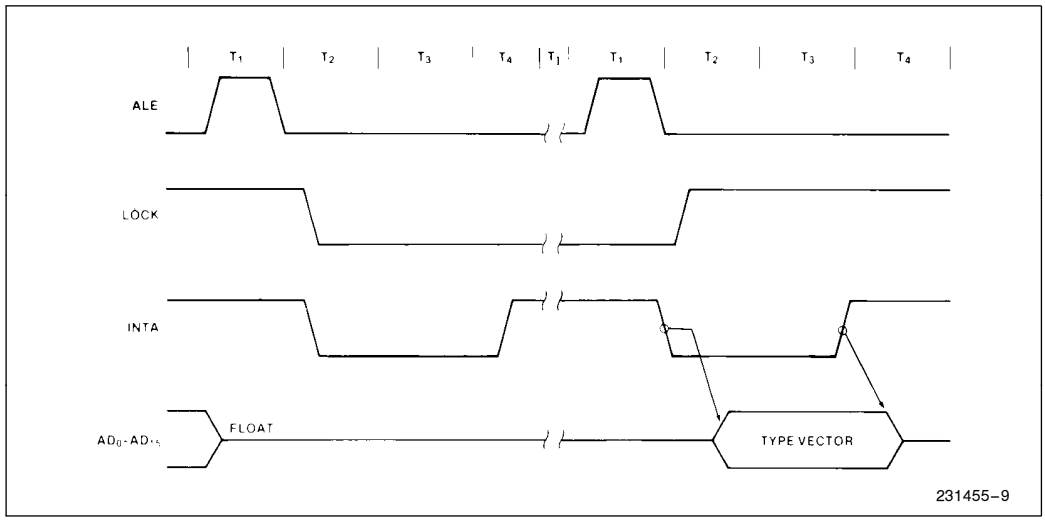


Figure 6. Interrupt Acknowledge Sequence

EXTERNAL SYNCHRONIZATION VIA TEST

As an alternative to the interrupts and general I/O capabilities, the 8086 provides a single software-testable input known as the $\overline{TEST}$ signal. At any time the program may execute a WAIT instruction. If at that time the $\overline{TEST}$ signal is inactive (HIGH), program execution becomes suspended while the processor waits for $\overline{TEST}$ to become active. It must remain active for at least 5 CLK cycles. The WAIT instruction is re-executed repeatedly until that time. This activity does not consume bus cycles. The processor remains in an idle state while waiting. All 8086 drivers go to 3-state OFF if bus "Hold" is entered. If interrupts are enabled, they may occur while the processor is waiting. When this occurs the processor fetches the WAIT instruction one extra time, processes the interrupt, and then re-fetches and re-executes the WAIT instruction upon returning from the interrupt.

Basic System Timing

Typical system configurations for the processor operating in minimum mode and in maximum mode are shown in Figures 4a and 4b, respectively. In minimum mode, the MN/$\overline{MX}$ pin is strapped to V_{CC} and the processor emits bus control signals in a manner similar to the 8085. In maximum mode, the MN/$\overline{MX}$ pin is strapped to V_{SS} and the processor emits coded status information which the 8288 bus controller uses to generate MULTIBUS compatible bus control signals. Figure 5 illustrates the signal timing relationships.

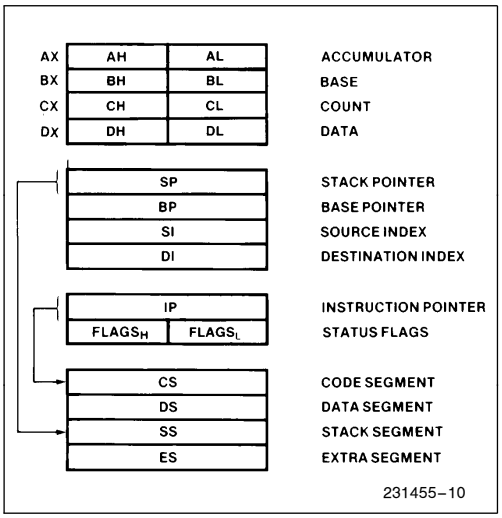


Figure 7. 8086 Register Model

SYSTEM TIMING—MINIMUM SYSTEM

The read cycle begins in T_1 with the assertion of the Address Latch Enable (ALE) signal. The trailing (low-going) edge of this signal is used to latch the address information, which is valid on the local bus at this time, into the address latch. The $\overline{BHE}$ and A_0 signals address the low, high, or both bytes. From T_1 to T_4 the M/$\overline{IO}$ signal indicates a memory or I/O operation. At T_2 the address is removed from the local bus and the bus goes to a high impedance state. The read control signal is also asserted at T_2. The read ($\overline{RD}$) signal causes the addressed device to enable its data bus drivers to the local bus. Some time later valid data will be available on the bus and the addressed device will drive the READY line HIGH. When the processor returns the read signal to a HIGH level, the addressed device will again 3-state its bus drivers. If a transceiver is required to buffer the 8086 local bus, signals DT/$\overline{R}$ and $\overline{DEN}$ are provided by the 8086.

A write cycle also begins with the assertion of ALE and the emission of the address. The M/$\overline{IO}$ signal is again asserted to indicate a memory or I/O write operation. In the T_2 immediately following the address emission the processor emits the data to be written into the addressed location. This data remains valid until the middle of T_4. During T_2, T_3, and T_W the processor asserts the write control signal. The write ($\overline{WR}$) signal becomes active at the beginning of T_2 as opposed to the read which is delayed somewhat into T_2 to provide time for the bus to float.

The $\overline{BHE}$ and A_0 signals are used to select the proper byte(s) of the memory/IO word to be read or written according to the following table:

$\overline{BHE}$	A0	Characteristics
0	0	Whole word
0	1	Upper byte from/to odd address
1	0	Lower byte from/to even address
1	1	None

I/O ports are addressed in the same manner as memory location. Even addressed bytes are transferred on the D_7–D_0 bus lines and odd addressed bytes on D_{15}–D_8.

The basic difference between the interrupt acknowledge cycle and a read cycle is that the interrupt acknowledge signal ($\overline{INTA}$) is asserted in place of the read ($\overline{RD}$) signal and the address bus is floated. (See Figure 6.) In the second of two successive INTA cycles, a byte of information is read from bus

lines D_7-D_0 as supplied by the inerrupt system logic (i.e., 8259A Priority Interrupt Controller). This byte identifies the source (type) of the interrupt. It is multiplied by four and used as a pointer into an interrupt vector lookup table, as described earlier.

BUS TIMING—MEDIUM SIZE SYSTEMS

For medium size systems the MN/$\overline{MX}$ pin is connected to V_{SS} and the 8288 Bus Controller is added to the system as well as a latch for latching the system address, and a transceiver to allow for bus loading greater than the 8086 is capable of handling. Signals ALE, DEN, and DT/$\overline{R}$ are generated by the 8288 instead of the processor in this configuration although their timing remains relatively the same. The 8086 status outputs ($\overline{S_2}$, $\overline{S_1}$, and $\overline{S_0}$) provide type-of-cycle information and become 8288 inputs. This bus cycle information specifies read (code, data, or I/O), write (data or I/O), interrupt acknowledge, or software halt. The 8288 thus issues control signals specifying memory read or write, I/O read or write, or interrupt acknowledge. The 8288 provides two types of write strobes, normal and advanced, to be applied as required. The normal write strobes have data valid at the leading edge of write. The advanced write strobes have the same timing as read strobes, and hence data isn't valid at the leading edge of write. The transceiver receives the usual DIR and $\overline{G}$ inputs from the 8288's DT/$\overline{R}$ and DEN.

The pointer into the interrupt vector table, which is passed during the second INTA cycle, can derive from an 8259A located on either the local bus or the system bus. If the master 8259A Priority Interrupt Controller is positioned on the local bus, a TTL gate is required to disable the transceiver when reading from the master 8259A during the interrupt acknowledge sequence and software ''poll''.

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias . . . . . 0°C to 70°C
Storage Temperature . . . . . −65°C to +150°C
Voltage on Any Pin with Respect to Ground . . . . . −1.0V to +7V
Power Dissipation 2.5W

NOTICE: This is a production data sheet. The specifications are subject to change without notice.

**WARNING: Stressing the device beyond the "Absolute Maximum Ratings" may cause permanent damage. These are stress ratings only. Operation beyond the "Operating Conditions" is not recommended and extended exposure beyond the "Operating Conditions" may affect device reliability.*

D.C. CHARACTERISTICS

(8086: $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$)
(8086-1: $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 5\%$)
(8086-2: $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 5\%$)

Symbol	Parameter	Min	Max	Units	Test Conditions
V_{IL}	Input Low Voltage	−0.5	+0.8	V	(Note 1)
V_{IH}	Input High Voltage	2.0	V_{CC} + 0.5	V	(Notes 1, 2)
V_{OL}	Output Low Voltage		0.45	V	$I_{OL} = 2.5$ mA
V_{OH}	Output High Voltage	2.4		V	$I_{OH} = -400\ \mu A$
I_{CC}	Power Supply Current: 8086 8086-1 8086-2		340 360 350	mA	$T_A = 25°C$
I_{LI}	Input Leakage Current		±10	μA	$0V \le V_{IN} \le V_{CC}$ (Note 3)
I_{LO}	Output Leakage Current		±10	μA	$0.45V \le V_{OUT} \le V_{CC}$
V_{CL}	Clock Input Low Voltage	−0.5	+0.6	V	
V_{CH}	Clock Input High Voltage	3.9	V_{CC} + 1.0	V	
C_{IN}	Capacitance of Input Buffer (All input except AD_0–AD_{15}, $\overline{RQ}/\overline{GT}$)		15	pF	fc = 1 MHz
C_{IO}	Capacitance of I/O Buffer (AD_0–AD_{15}, $\overline{RQ}/\overline{GT}$)		15	pF	fc = 1 MHz

NOTES:
1. V_{IL} tested with MN/$\overline{MX}$ Pin = 0V. V_{IH} tested with MN/$\overline{MX}$ Pin = 5V. MN/$\overline{MX}$ Pin is a Strap Pin.
2. Not applicable to $\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$ (Pins 30 and 31).
3. HOLD and HLDA I_{LI} min = 30 μA, max = 500 μA.

A.C. CHARACTERISTICS

(8086: T_A = 0°C to 70°C, V_{CC} = 5V ± 10%)
(8086-1: T_A = 0°C to 70°C, V_{CC} = 5V ± 5%)
(8086-2: T_A = 0°C to 70°C, V_{CC} = 5V ± 5%)

MINIMUM COMPLEXITY SYSTEM TIMING REQUIREMENTS

Symbol	Parameter	8086		8086-1		8086-2		Units	Test Conditions
		Min	Max	Min	Max	Min	Max		
TCLCL	CLK Cycle Period	200	500	100	500	125	500	ns	
TCLCH	CLK Low Time	118		53		68		ns	
TCHCL	CLK High Time	69		39		44		ns	
TCH1CH2	CLK Rise Time		10		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10		10	ns	From 3.5V to 1.0V
TDVCL	Data in Setup Time	30		5		20		ns	
TCLDX	Data in Hold Time	10		10		10		ns	
TR1VCL	RDY Setup Time into 8284A (See Notes 1, 2)	35		35		35		ns	
TCLR1X	RDY Hold Time into 8284A (See Notes 1, 2)	0		0		0		ns	
TRYHCH	READY Setup Time into 8086	118		53		68		ns	
TCHRYX	READY Hold Time into 8086	30		20		20		ns	
TRYLCL	READY Inactive to CLK (See Note 3)	−8		−10		−8		ns	
THVCH	HOLD Setup Time	35		20		20		ns	
TINVCH	INTR, NMI, $\overline{\text{TEST}}$ Setup Time (See Note 2)	30		15		15		ns	
TILIH	Input Rise Time (Except CLK)		20		20		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12		12		12	ns	From 2.0V to 0.8V

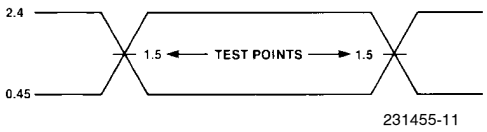
A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

Symbol	Parameter	8086		8086-1		8086-2		Units	Test Conditions
		Min	Max	Min	Max	Min	Max		
TCLAV	Address Valid Delay	10	110	10	50	10	60	ns	*C_L = 20–100 pF for all 8086 Outputs (In addition to 8086 selfload)
TCLAX	Address Hold Time	10		10		10		ns	
TCLAZ	Address Float Delay	TCLAX	80	10	40	TCLAX	50	ns	
TLHLL	ALE Width	TCLCH-20		TCLCH-10		TCLCH-10		ns	
TCLLH	ALE Active Delay		80		40		50	ns	
TCHLL	ALE Inactive Delay		85		45		55	ns	
TLLAX	Address Hold Time	TCHCL-10		TCHCL-10		TCHCL-10		ns	
TCLDV	Data Valid Delay	10	110	10	50	10	60	ns	
TCHDX	Data Hold Time	10		10		10		ns	
TWHDX	Data Hold Time After WR	TCLCH-30		TCLCH-25		TCLCH-30		ns	
TCVCTV	Control Active Delay 1	10	110	10	50	10	70	ns	
TCHCTV	Control Active Delay 2	10	110	10	45	10	60	ns	
TCVCTX	Control Inactive Delay	10	110	10	50	10	70	ns	
TAZRL	Address Float to READ Active	0		0		0		ns	
TCLRL	$\overline{RD}$ Active Delay	10	165	10	70	10	100	ns	
TCLRH	$\overline{RD}$ Inactive Delay	10	150	10	60	10	80	ns	
TRHAV	$\overline{RD}$ Inactive to Next Address Active	TCLCL-45		TCLCL-35		TCLCL-40		ns	
TCLHAV	HLDA Valid Delay	10	160	10	60	10	100	ns	
TRLRH	$\overline{RD}$ Width	2TCLCL-75		2TCLCL-40		2TCLCL-50		ns	
TWLWH	$\overline{WR}$ Width	2TCLCL-60		2TCLCL-35		2TCLCL-40		ns	
TAVAL	Address Valid to ALE Low	TCLCH-60		TCLCH-35		TCLCH-40		ns	
TOLOH	Output Rise Time		20		20		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12		12		12	ns	From 2.0V to 0.8V

NOTES:
1. Signal at 8284A shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T2 state. (8 ns into T3).

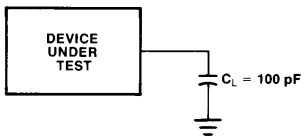
A.C. TESTING INPUT, OUTPUT WAVEFORM



231455-11

A.C. Testing: Inputs are driven at 2.4V for a Logic "1" and 0.45V for a Logic "0". Timing measurements are made at 1.5V for both a Logic "1" and "0".

A.C. TESTING LOAD CIRCUIT

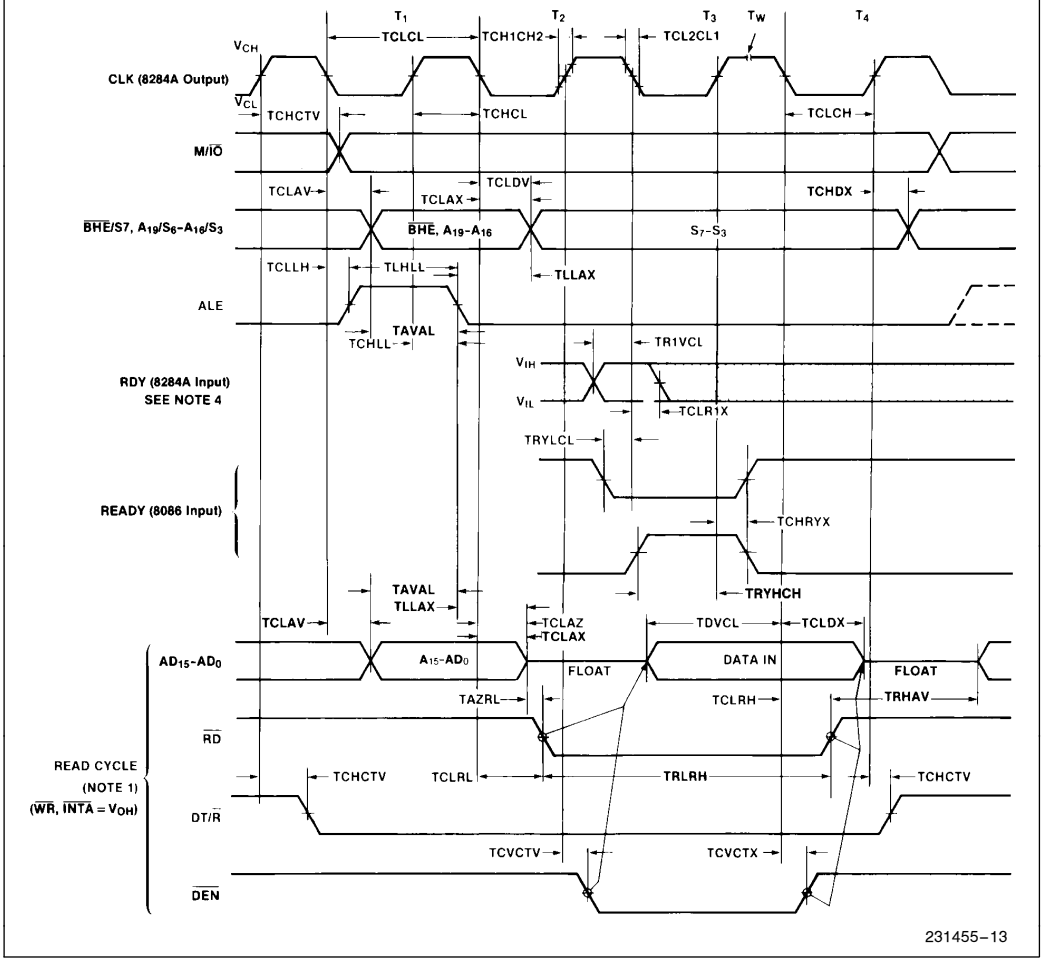


231455-12

C_L Includes Jig Capacitance

WAVEFORMS

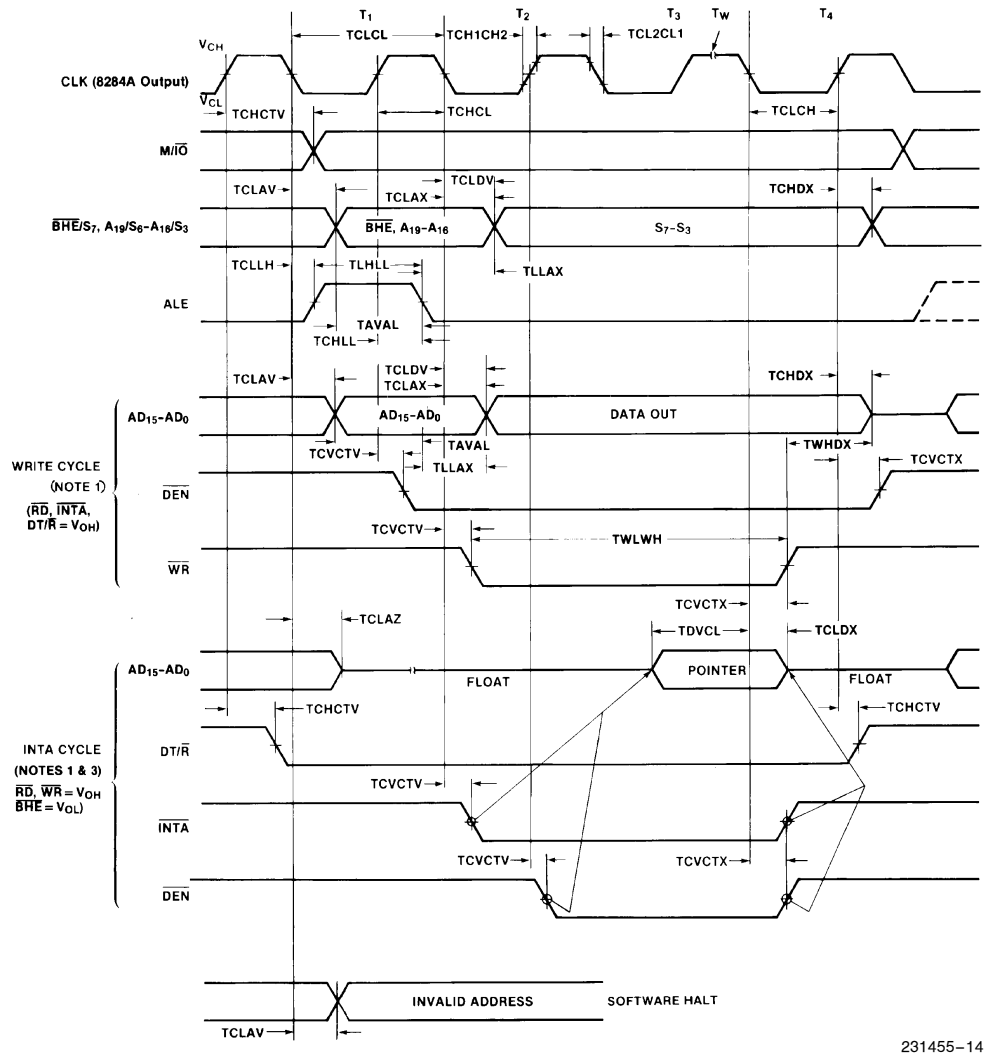
MINIMUM MODE



231455-13

WAVEFORMS (Continued)

MINIMUM MODE (Continued)



SOFTWARE HALT—
RD, WR, INTA = V_{OH}
DT/R = INDETERMINATE

NOTES:
1. All signals switch between V_{OH} and V_{OL} unless otherwise specified.
2. RDY is sampled near the end of T_2, T_3, T_W to determine if T_W machines states are to be inserted.
3. Two INTA cycles run back-to-back. The 8086 LOCAL ADDR/DATA BUS is floating during both INTA cycles. Control signals shown for second INTA cycle.
4. Signals at 8284A are shown for reference only.
5. All timing measurements are made at 1.5V unless otherwise noted.

A.C. CHARACTERISTICS

MAX MODE SYSTEM (USING 8288 BUS CONTROLLER)
TIMING REQUIREMENTS

Symbol	Parameter	8086		8086-1		8086-2		Units	Test Conditions
		Min	Max	Min	Max	Min	Max		
TCLCL	CLK Cycle Period	200	500	100	500	125	500	ns	
TCLCH	CLK Low Time	118		53		68		ns	
TCHCL	CLK High Time	69		39		44		ns	
TCH1CH2	CLK Rise Time		10		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10		10	ns	From 3.5V to 1.0V
TDVCL	Data in Setup Time	30		5		20		ns	
TCLDX	Data in Hold Time	10		10		10		ns	
TR1VCL	RDY Setup Time into 8284A (Notes 1, 2)	35		35		35		ns	
TCLR1X	RDY Hold Time into 8284A (Notes 1, 2)	0		0		0		ns	
TRYHCH	READY Setup Time into 8086	118		53		68		ns	
TCHRYX	READY Hold Time into 8086	30		20		20		ns	
TRYLCL	READY Inactive to CLK (Note 4)	−8		−10		−8		ns	
TINVCH	Setup Time for Recognition (INTR, NMI, $\overline{\text{TEST}}$) (Note 2)	30		15		15		ns	
TGVCH	$\overline{\text{RQ}}/\overline{\text{GT}}$ Setup Time (Note 5)	30		15		15		ns	
TCHGX	$\overline{\text{RQ}}$ Hold Time into 8086	40		20		30		ns	
TILIH	Input Rise Time (Except CLK)		20		20		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12		12		12	ns	From 2.0V to 0.8V

A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

Symbol	Parameter	8086		8086-1		8086-2		Units	Test Conditions
		Min	Max	Min	Max	Min	Max		
TCLML	Command Active Delay (See Note 1)	10	35	10	35	10	35	ns	C_L = 20–100 pF for all 8086 Outputs (In addition to 8086 self-load)
TCLMH	Command Inactive Delay (See Note 1)	10	35	10	35	10	35	ns	
TRYHSH	READY Active to Status Passive (See Note 3)		110		45		65	ns	
TCHSV	Status Active Delay	10	110	10	45	10	60	ns	
TCLSH	Status Inactive Delay	10	130	10	55	10	70	ns	
TCLAV	Address Valid Delay	10	110	10	50	10	60	ns	
TCLAX	Address Hold Time	10		10		10		ns	
TCLAZ	Address Float Delay	TCLAX	80	10	40	TCLAX	50	ns	
TSVLH	Status Valid to ALE High (See Note 1)		15		15		15	ns	
TSVMCH	Status Valid to MCE High (See Note 1)		15		15		15	ns	
TCLLH	CLK Low to ALE Valid (See Note 1)		15		15		15	ns	
TCLMCH	CLK Low to MCE High (See Note 1)		15		15		15	ns	
TCHLL	ALE Inactive Delay (See Note 1)		15		15		15	ns	
TCLMCL	MCE Inactive Delay (See Note 1)		15		15		15	ns	
TCLDV	Data Valid Delay	10	110	10	50	10	60	ns	
TCHDX	Data Hold Time	10		10		10		ns	
TCVNV	Control Active Delay (See Note 1)	5	45	5	45	5	45	ns	
TCVNX	Control Inactive Delay (See Note 1)	10	45	10	45	10	45	ns	
TAZRL	Address Float to READ Active	0		0		0		ns	
TCLRL	RD Active Delay	10	165	10	70	10	100	ns	
TCLRH	RD Inactive Delay	10	150	10	60	10	80	ns	

A.C. CHARACTERISTICS (Continued)

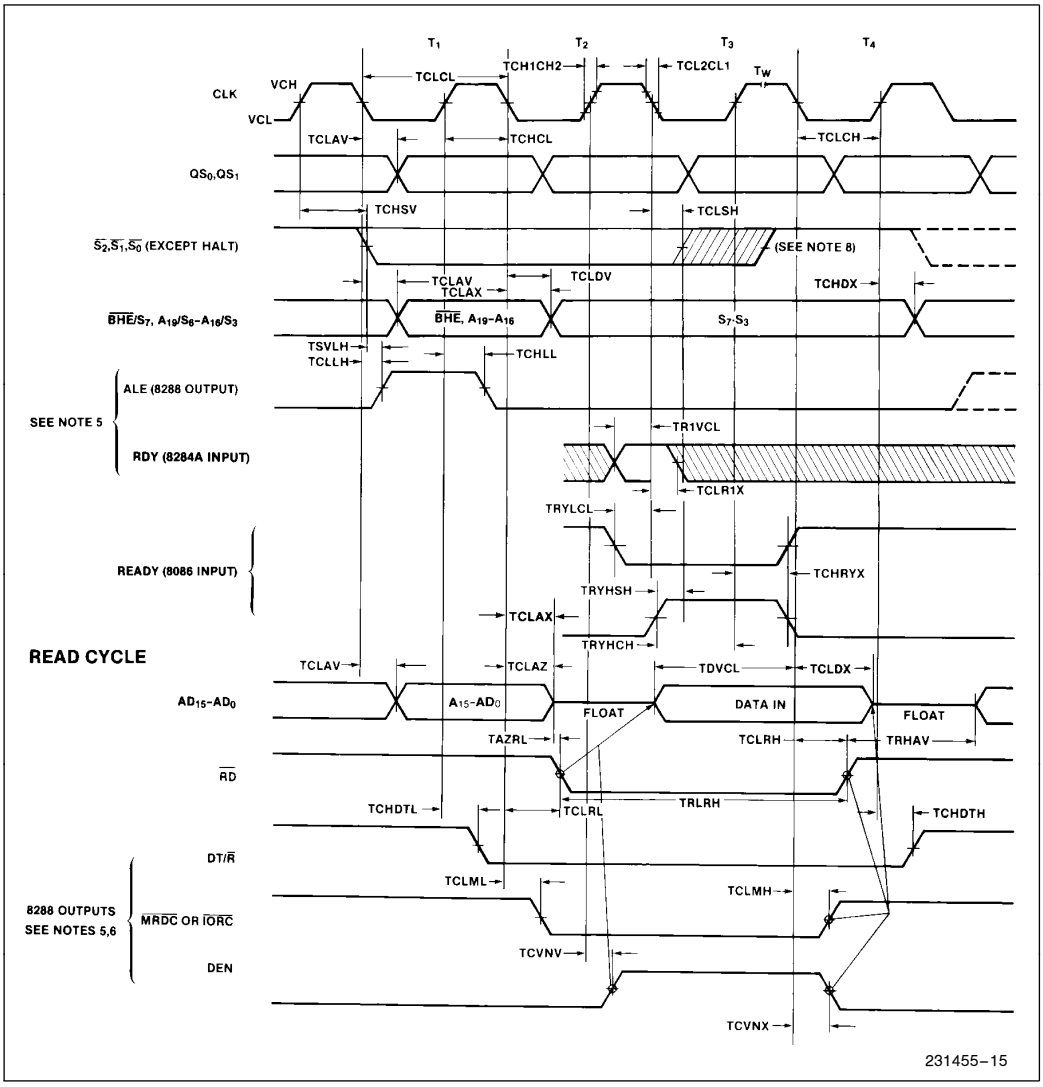
TIMING RESPONSES (Continued)

Symbol	Parameter	8086		8086-1		8086-2		Units	Test Conditions
		Min	Max	Min	Max	Min	Max		
TRHAV	RD Inactive to Next Address Active	TCLCL-45		TCLCL-35		TCLCL-40		ns	C_L = 20–100 pF for all 8086 Outputs (In addition to 8086 self-load)
TCHDTL	Direction Control Active Delay (Note 1)		50		50		50	ns	
TCHDTH	Direction Control Inactive Delay (Note 1)		30		30		30	ns	
TCLGL	GT Active Delay	0	85	0	38	0	50	ns	
TCLGH	GT Inactive Delay	0	85	0	45	0	50	ns	
TRLRH	RD Width	2TCLCL-75		2TCLCL-40		2TCLCL-50		ns	
TOLOH	Output Rise Time		20		20		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12		12		12	ns	From 2.0V to 0.8V

NOTES:
1. Signal at 8284A or 8288 shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T3 and wait states.
4. Applies only to T2 state (8 ns into T3).

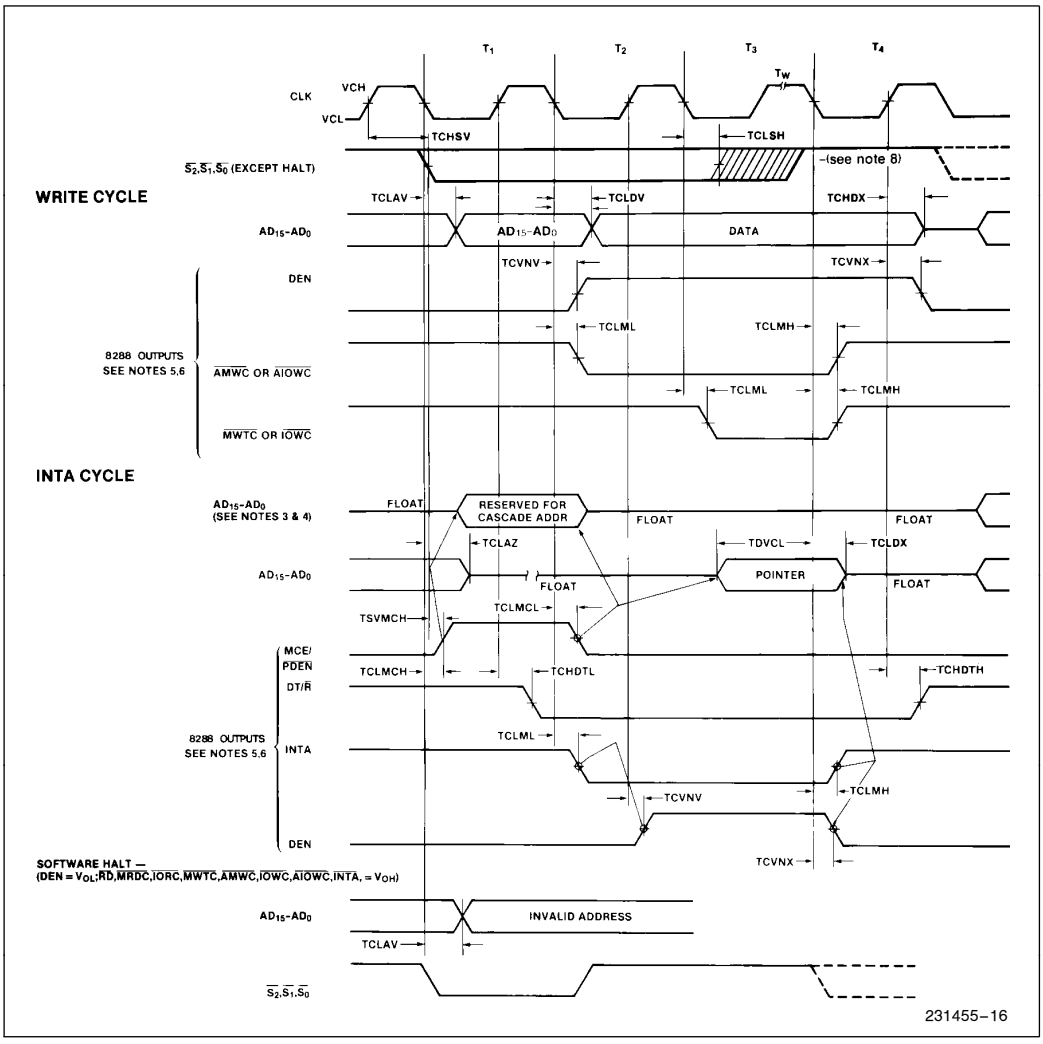
WAVEFORMS

MAXIMUM MODE



WAVEFORMS (Continued)

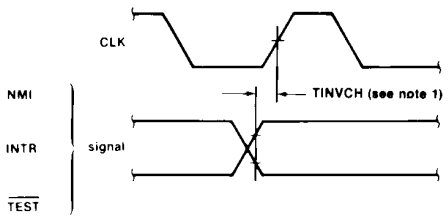
MAXIMUM MODE (Continued)



NOTES:
1. All signals switch between V_{OH} and V_{OL} unless otherwise specified.
2. RDY is sampled near the end of T_2, T_3, T_W to determine if T_W machines states are to be inserted.
3. Cascade address is valid between first and second INTA cycle.
4. Two INTA cycles run back-to-back. The 8086 LOCAL ADDR/DATA BUS is floating during both INTA cycles. Control for pointer address is shown for second INTA cycle.
5. Signals at 8284A or 8288 are shown for reference only.
6. The issuance of the 8288 command and control signals ($\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$, $\overline{IOWC}$, $\overline{AIOWC}$, $\overline{INTA}$ and DEN) lags the active high 8288 CEN.
7. All timing measurements are made at 1.5V unless otherwise noted.
8. Status inactive in state just prior to T_4.

WAVEFORMS (Continued)

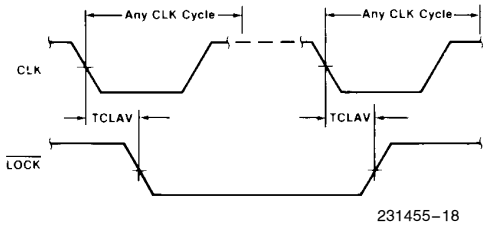
ASYNCHRONOUS SIGNAL RECOGNITION



231455–17

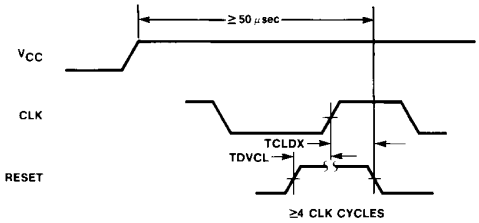
NOTE:
1. Setup requirements for asynchronous signals only to guarantee recognition at next CLK.

BUS LOCK SIGNAL TIMING (MAXIMUM MODE ONLY)



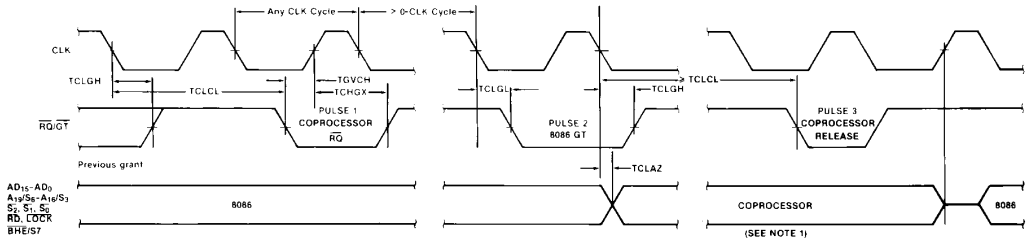
231455–18

RESET TIMING



231455–19

REQUEST/GRANT SEQUENCE TIMING (MAXIMUM MODE ONLY)



231455–20

NOTE:
The coprocessor may not drive the buses outside the region shown without risking contention.

WAVEFORMS (Continued)

HOLD/HOLD ACKNOWLEDGE TIMING (MINIMUM MODE ONLY)

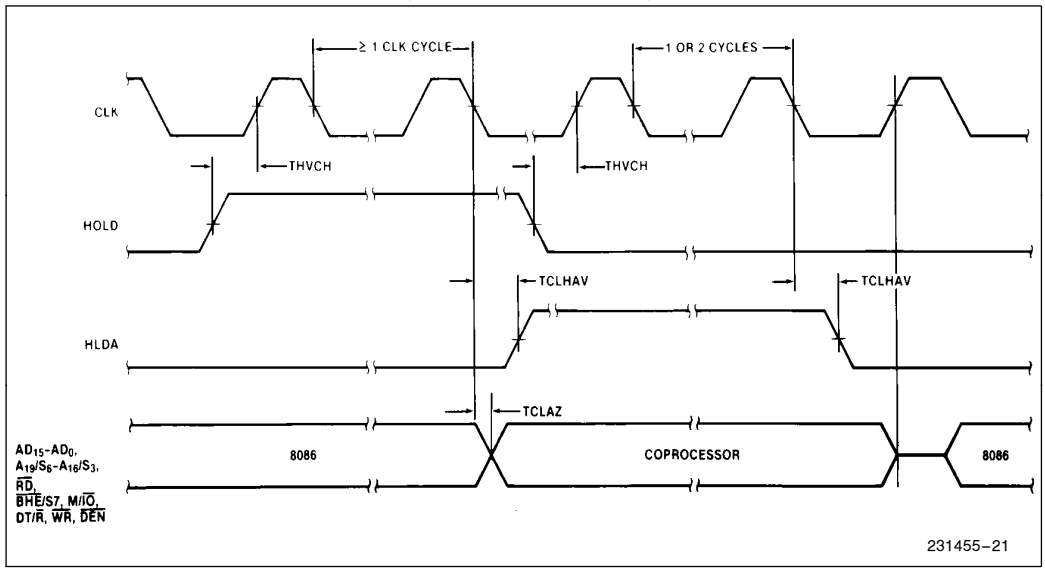


Table 2. Instruction Set Summary

Mnemonic and Description	Instruction Code			
DATA TRANSFER				
MOV = Move:	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1	
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		
PUSH = Push:				
Register/Memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m		
Register	0 1 0 1 0 reg			
Segment Register	0 0 0 reg 1 1 0			
POP = Pop:				
Register/Memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m		
Register	0 1 0 1 1 reg			
Segment Register	0 0 0 reg 1 1 1			
XCHG = Exchange:				
Register/Memory with Register	1 0 0 0 0 1 1 w	mod reg r/m		
Register with Accumulator	1 0 0 1 0 reg			
IN = Input from:				
Fixed Port	1 1 1 0 0 1 0 w	port		
Variable Port	1 1 1 0 1 1 0 w			
OUT = Output to:				
Fixed Port	1 1 1 0 0 1 1 w	port		
Variable Port	1 1 1 0 1 1 1 w			
XLAT = Translate Byte to AL	1 1 0 1 0 1 1 1			
LEA = Load EA to Register	1 0 0 0 1 1 0 1	mod reg r/m		
LDS = Load Pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m		
LES = Load Pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m		
LAHF = Load AH with Flags	1 0 0 1 1 1 1 1			
SAHF = Store AH into Flags	1 0 0 1 1 1 1 0			
PUSHF = Push Flags	1 0 0 1 1 1 0 0			
POPF = Pop Flags	1 0 0 1 1 1 0 1			

Table 2. Instruction Set Summary (Continued)

Mnemonic and Description	Instruction Code			
ARITHMETIC	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**
ADD = Add:				
Reg./Memory with Register to Either	0 0 0 0 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s: w = 01
Immediate to Accumulator	0 0 0 0 0 1 0 w	data	data if w = 1	
ADC = Add with Carry:				
Reg./Memory with Register to Either	0 0 0 1 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s: w = 01
Immediate to Accumulator	0 0 0 1 0 1 0 w	data	data if w = 1	
INC = Increment:				
Register/Memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m		
Register	0 1 0 0 0 reg			
AAA = ASCII Adjust for Add	0 0 1 1 0 1 1 1			
BAA = Decimal Adjust for Add	0 0 1 0 0 1 1 1			
SUB = Subtract:				
Reg./Memory and Register to Either	0 0 1 0 1 0 d w	mod reg r/m		
Immediate from Register/Memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s w = 01
Immediate from Accumulator	0 0 1 0 1 1 0 w	data	data if w = 1	
SSB = Subtract with Borrow				
Reg./Memory and Register to Either	0 0 0 1 1 0 d w	mod reg r/m		
Immediate from Register/Memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s w = 01
Immediate from Accumulator	0 0 0 1 1 1 w	data	data if w = 1	
DEC = Decrement:				
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
NEG = Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		
CMP = Compare:				
Register/Memory and Register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with Register/Memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s w = 01
Immediate with Accumulator	0 0 1 1 1 1 0 w	data	data if w = 1	
AAS = ASCII Adjust for Subtract	0 0 1 1 1 1 1 1			
DAS = Decimal Adjust for Subtract	0 0 1 0 1 1 1 1			
MUL = Multiply (Unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
IMUL = Integer Multiply (Signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
AAM = ASCII Adjust for Multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
DIV = Divide (Unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
IDIV = Integer Divide (Signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
AAD = ASCII Adjust for Divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
CBW = Convert Byte to Word	1 0 0 1 1 0 0 0			
CWD = Convert Word to Double Word	1 0 0 1 1 0 0 1			

Table 2. Instruction Set Summary (Continued)

Mnemonic and Description	Instruction Code			
LOGIC	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
NOT = Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m		
SHL/SAL = Shift Logical/Arithmetic Left	1 1 0 1 0 0 v w	mod 1 0 0 r/m		
SHR = Shift Logical Right	1 1 0 1 0 0 v w	mod 1 0 1 r/m		
SAR = Shift Arithmetic Right	1 1 0 1 0 0 v w	mod 1 1 1 r/m		
ROL = Rotate Left	1 1 0 1 0 0 v w	mod 0 0 0 r/m		
ROR = Rotate Right	1 1 0 1 0 0 v w	mod 0 0 1 r/m		
RCL = Rotate Through Carry Flag Left	1 1 0 1 0 0 v w	mod 0 1 0 r/m		
RCR = Rotate Through Carry Right	1 1 0 1 0 0 v w	mod 0 1 1 r/m		
AND = **And:**				
Reg./Memory and Register to Either	0 0 1 0 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w = 1
Immediate to Accumulator	0 0 1 0 0 1 0 w	data	data if w = 1	
TEST = **And Function to Flags, No Result:**				
Register/Memory and Register	1 0 0 0 0 1 0 w	mod reg r/m		
Immediate Data and Register/Memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate Data and Accumulator	1 0 1 0 1 0 0 w	data	data if w = 1	
OR = **Or:**				
Reg./Memory and Register to Either	0 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w = 1
Immediate to Accumulator	0 0 0 0 1 1 0 w	data	data if w = 1	
XOR = **Exclusive or:**				
Reg./Memory and Register to Either	0 0 1 1 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w = 1
Immediate to Accumulator	0 0 1 1 0 1 0 w	data	data if w = 1	
STRING MANIPULATION				
REP = Repeat	1 1 1 1 0 0 1 z			
MOVS = Move Byte/Word	1 0 1 0 0 1 0 w			
CMPS = Compare Byte/Word	1 0 1 0 0 1 1 w			
SCAS = Scan Byte/Word	1 0 1 0 1 1 1 w			
LODS = Load Byte/Wd to AL/AX	1 0 1 0 1 1 0 w			
STOS = Stor Byte/Wd from AL/A	1 0 1 0 1 0 1 w			
CONTROL TRANSFER				
CALL = **Call:**				
Direct within Segment	1 1 1 0 1 0 0 0	disp-low	disp-high	
Indirect within Segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m		
Direct Intersegment	1 0 0 1 1 0 1 0	offset-low	offset-high	
		seg-low	seg-high	
Indirect Intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m		

Table 2. Instruction Set Summary (Continued)

Mnemonic and Description	Instruction Code		
JMP = Unconditional Jump:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within Segment	1 1 1 0 1 0 0 1	disp-low	disp-high
Direct within Segment-Short	1 1 1 0 1 0 1 1	disp	
Indirect within Segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
Direct Intersegment	1 1 1 0 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect Intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	
RET = Return from CALL:			
Within Segment	1 1 0 0 0 0 1 1		
Within Seg Adding Immed to SP	1 1 0 0 0 0 1 0	data-low	data-high
Intersegment	1 1 0 0 1 0 1 1		
Intersegment Adding Immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high
JE/JZ = Jump on Equal/Zero	0 1 1 1 0 1 0 0	disp	
JL/JNGE = Jump on Less/Not Greater or Equal	0 1 1 1 1 1 0 0	disp	
JLE/JNG = Jump on Less or Equal/ Not Greater	0 1 1 1 1 1 1 0	disp	
JB/JNAE = Jump on Below/Not Above or Equal	0 1 1 1 0 0 1 0	disp	
JBE/JNA = Jump on Below or Equal/ Not Above	0 1 1 1 0 1 1 0	disp	
JP/JPE = Jump on Parity/Parity Even	0 1 1 1 1 0 1 0	disp	
JO = Jump on Overflow	0 1 1 1 0 0 0 0	disp	
JS = Jump on Sign	0 1 1 1 1 0 0 0	disp	
JNE/JNZ = Jump on Not Equal/Not Zero	0 1 1 1 0 1 0 1	disp	
JNL/JGE = Jump on Not Less/Greater or Equal	0 1 1 1 1 1 0 1	disp	
JNLE/JG = Jump on Not Less or Equal/ Greater	0 1 1 1 1 1 1 1	disp	
JNB/JAE = Jump on Not Below/Above or Equal	0 1 1 1 0 0 1 1	disp	
JNBE/JA = Jump on Not Below or Equal/Above	0 1 1 1 0 1 1 1	disp	
JNP/JPO = Jump on Not Par/Par Odd	0 1 1 1 1 0 1 1	disp	
JNO = Jump on Not Overflow	0 1 1 1 0 0 0 1	disp	
JNS = Jump on Not Sign	0 1 1 1 1 0 0 1	disp	
LOOP = Loop CX Times	1 1 1 0 0 0 1 0	disp	
LOOPZ/LOOPE = Loop While Zero/Equal	1 1 1 0 0 0 0 1	disp	
LOOPNZ/LOOPNE = Loop While Not Zero/Equal	1 1 1 0 0 0 0 0	disp	
JCXZ = Jump on CX Zero	1 1 1 0 0 0 1 1	disp	
INT = Interrupt			
Type Specified	1 1 0 0 1 1 0 1	type	
Type 3	1 1 0 0 1 1 0 0		
INTO = Interrupt on Overflow	1 1 0 0 1 1 1 0		
IRET = Interrupt Return	1 1 0 0 1 1 1 1		

Table 2. Instruction Set Summary (Continued)

Mnemonic and Description	Instruction Code	
	76543210	76543210
PROCESSOR CONTROL		
CLC = Clear Carry	11111000	
CMC = Complement Carry	11110101	
STC = Set Carry	11111001	
CLD = Clear Direction	11111100	
STD = Set Direction	11111101	
CLI = Clear Interrupt	11111010	
STI = Set Interrupt	11111011	
HLT = Halt	11110100	
WAIT = Wait	10011011	
ESC = Escape (to External Device)	11011xxx	mod x x x r/m
LOCK = Bus Lock Prefix	11110000	

NOTES:
AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value
Greater = more positive;
Less = less positive (more negative) signed values
if d = 1 then ''to'' reg; if d = 0 then ''from'' reg
if w = 1 then word instruction; if w = 0 then byte instruction
if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16 bits, disp-high is absent
if mod = 10 then DISP = disp-high; disp-low
if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)
*except if mod = 00 and r/m = 110 then EA = disp-high; disp-low.

if s w = 01 then 16 bits of immediate data form the operand
if s w = 11 then an immediate data byte is sign extended to form the 16-bit operand
if v = 0 then ''count'' = 1; if v = 1 then ''count'' in (CL)
x = don't care
z is used for string primitives for comparison with ZF FLAG

SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

REG is assigned according to the following table:

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:
FLAGS = X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

DATA SHEET REVISION REVIEW

The following list represents key differences between this and the -004 data sheet. Please review this summary carefully.

1. The Intel 8086 implementation technology (HMOS) has been changed to (HMOS-III).
2. Delete all ''changes from 1985 Handbook Specification'' sentences.

P A R T III

SESSIONAL PAPERS

DHARMSINH DESAI UNIVERSITY, NADIAD
FACULTY OF TECHNOLOGY
FIRST SESSIONALEXAMINATION
SUBJECT: (EC601) ADVANCED MICROPROCESSORS

Examination	:B. TECH. SEMESTER VI [EC]	**Seat No**	**: _____**
Date	**:** 08/01/2018	**Day**	**:**Monday
Time	**:** 1hr 15min	**Max. Marks**	**:** 36

INSTRUCTIONS:

1. Figures to the right indicate maximum marks for that question.
2. Assume suitable data, if required & mention them clearly.
3. Draw neat sketches wherever necessary.

Q.1 Do as directed. **[6]**

(A) Choose the most appropriate alternate (s)

(i) Privileged modes in ARM7 are [1]
a) abort b)supervisor c)IRQ d)all

(ii) When exception is generated, the return address is saved in the [1]
a) R13 of the existing mode b) R13 of the new mode
c) R14 of the existing mode d)R14 of the new mode

(iii) In ARM7 architecture, same priority is assigned to [1]
a) *SWI* and *undefined* instructions b) *SWI* instruction and *FIQ*
c) *IRQ* and *FIQ* d) *IRQ* and *undefined* instructions

(iv) The address lines used in ARM mode are [1]
a) A31-A0 b) A31-A1 c) A31-A2 d) A30-A2

(B) "*ARM7* core is biendian". Justify the statement. [2]

Q.2 Attempt *any TWO* of the following **[12]**

(A) Compare conventional RISC and CISC architecture with reference to followings: [6]
a) Number of cycles for execution. b) Format of instruction set architecture.
c) Available general purpose registers d) Number of bytes of an instruction

(B) (i)What architectural support is provided by ARM7 core to make FIQ a Fast interrupt? [3]
(ii) Explain the exception exit mechanism for all modes of ARM7 [3]

(C) List out the visible registers in *ARM* and *Thumb* modes of *ARM7 TDMI* processor core. Discuss the significance of *SPSR* register during exception handling. [6]

Q.3 Do as directed. **[06]**

(A) State at least one situation in which the 8086 pipeline will not be useful. [1]

(B) What is the principle benefit of address manipulation in the 8086? [1]

(C) How the locations of different registers of the 8086 are divided in the architecture? [2]

(D) Compare the instructions. MOV AL,[SI] & MOV CL,[SI] [2]

Q.4 Do as directed. **[12]**

(A) Specify the addressing mode and segments accessed by each of the following instructions: (i) *MOV [BX+SI],SP* (ii) *MOV ES,DATA1* (iii) *MOV CL, [BP]* [3]

(B) Explain how BIU and EU of the 8086 cooperate with each other during instruction execution? [3]

(C) Write set of instruction(s) of To copy content of 11100H physical address in to AL register using 3 different addressing modes. [6]

OR

Q.4 Do as directed. **[12]**

(A) Find the validity of following instructions. Give reason if it is invalid. [3]
(i) *MOV DX,BL* (ii) *MOV BP,[BX]* (iii) *MOV [BP+BX],SI*

(B) Discuss briefly how segmentation of memory of the 8086 can support multiprogramming environment. [3]

(C) Consider that following instructions are written in the code segment. Discuss the validity of each of the instruction. Also determine the result after execution of each instruction. [6]

```
MOV BL, X
MOV CX, Z
MOV DX, DATA
MOV SI, OFFSET STRING1
MOV DX, [SI]
MOV AL, [SI+2]
```

Consider that the data segment has been defined as follows:

```
DATA SEGMENT
X DB 23, 24
Y DW 23, 24
Z DB 25
STRING1 DB 'DDU'
 ENDS
```

DHARMSINH DESAI UNIVERSITY, NADIAD
FACULTY OF TECHNOLOGY
SECOND INTERNAL EXAMINATION
SUBJECT: (EC615) ADVANCED MICROPROCESSORS

Examination	:B.TECH. SEMESTER VI [EC]	**Seat No**	**: _____**
Date	**: 12-2-2018**	**Day**	**:**
Time	**: 1 hr.15 min.**	**Max. Marks**	**:** 36

INSTRUCTIONS:

1 Figures to the right indicate maximum marks for that question.
2 Assume suitable data, if required & mention them clearly.
3. Write comments in the programs.

Q.1 Do as directed. **[06]**

(A) **State true/false with reason(s).** [3]

(i) Because the 8086 is a 16 bit processor, it is not possible to perform data transfer operations of a byte size.

(ii) The READY signal must be raised high at appropriate time during a wait state, if no further wait states are needed in the bus cycle.

(B) The condition that will terminate the instruction REPE CMPSB is [1]

a) CX=0 b) compared bytes are equal
c) compared bytes are not equal d) CX=0 or compared bytes are not equal

(C) Justify true/false with reason. "RET instruction do have more than one opcodes." [2]

Q.2 Attempt the following. **[12]**

(A) What is the function of following pin outs of the 8086? [3]
(i)IO/M' (ii) ALE (iii) DT/R'

(B) State the status of BHE' signal in the following cases. Also mention the corresponding data lines to be used at that time. [3]
(i) The 8086 is currently accessing a memory byte at address 98765H.
(ii) The 8086 is currently accessing a memory word of 16bits from address A9876H.
(iii) The 8086 is currently accessing a memory byte at address 87654H.

(C) (i) What will be content of AX after execution of the following instructions? [3]
MOV AL,38H
ADD AL, 34H
AAA
Give comment on the result

(D) Consider following sequence of events (calls & returns) occur in a main program (assume CS=5000H of main program, SP=0050H, SS=2000H, DS=3000,ES=4000). [3]
1. Main program calls NEAR procedure PROC_A (return address IP=1000)
2. PROC_A calls FAR procedure PORC_B (return address IP=2500)
3. return is made to PROC_A
4. return is made to main program.

Consider only stack activity during execution of call and return instructions. Illustrate the stack activities by series of stack diagrams.

OR

Q.2 Attempt the following. **[12]**

(A) Answer followings in brief for the 8086. [3]
(i) What are the addresses corresponding the firs byte and the last byte of the total maximum amount memory?
(ii) How many address lines are required to be demultiplexed? Which is the other signal also required to be demultiplexed along with the address lines?
(iii) Mention at least two signals that can be used for detecting the T state when the status of READY signal has to be changed in case of slower devices.

(B) Certain signals are incorrectly connected in the circuit shown in **fig.1,** for generating a single wait state. Point out necessary corrections with reason. [3]

(C) Determine the machine code for the instruction MOV CX, [1234H], What will be the change in the machine code if the segment override prefix is used in the instruction? [6]
Template for *MOV* instruction is as follows:
100010 DW MOD-REG-R/M LOW DISP HIGH DISP

Q.3 Do as directed. **[6]**

(A) Choose the most appropriate alternate (s)

i) The result of MLA R4,R3,R2,R1 stored in [1]
a) R4 b) R4:R3 c) R3 d) R3:R2

ii) The operation performed by instruction ADD R5,R5,R3,LSL R2 is [1]
i) $R5=R5^{R2}+R3$ ii) $R5= R5+ R3x\ 2^{R2}$ iii) $R5=R5+R3 \times R2^2$ iv) $R5=R5 \times R3 +2^{R2}$

(B) Differentiate between LDR R0, [R1,#4] and LDR R0, [R1],#4 instructions. [2]

(C) Implement given high level language statement using assembly language instruction(s). [2]
if ((a≠ b) && (c=d)) e--;

Q.4 Attempt any *TWO* of the following. [12]

(A) Mention the changes in the registers and/or memory locations after executing given instructions. Consider initial values of registers before execution of each of the following instructions as R1=0x38, R2=0x51,R3=0x23,R5=0x94, R6=0x4000000,R7=0x67. [6]
i) STMIA R6,{R1-R3} ii) STMFD R6, {R5-R7}
iii) STMFA R6, {R5-R7} iv) STMDB R6, {R1-R3}

(B) Implement jump tables using at least two different methods. Compare them with respect to [6]
i) memory requirements ii) speed of execution

(C) Write assembly language program to add two 16 bit arrays element wise and store result in to third array. The size of the array is available in link register. [6]

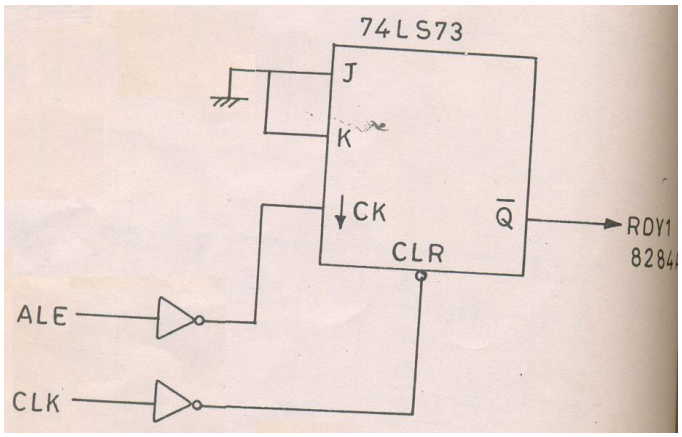


FIG.1

DHARMSINH DESAI UNIVERSITY, NADIAD
FACULTY OF TECHNOLOGY
THIRD INTERNAL EXAMINATION
SUBJECT: (EC615) ADVANCED MICROPROCESSORS

Examination	:B.TECH. SEMESTER VI [EC]	**Seat No**	**: _____**
Date	**: 26-3-2018**	**Day**	**:**
Time	**: 1 hr.15 min.**	**Max. Marks**	**:** 36

INSTRUCTIONS:

1 Figures to the right indicate maximum marks for that question.
2 Assume suitable data, if required & mention them clearly.
3. Write comments in the programs.

Q.1 Do as directed. **[06]**

(A) **State true/false with reason(s).** [3]
i) The 8086 always executes a far call for ISR of an interrupt.
ii) For any access into the memory, the Intel advanced processor always needs to fetch a descriptor from the descriptor table from physical memory.

(B) Which signal can be ignored while interfacing ROM in the 8086 system? [1]
(a) ALE (b)A0 (c) M/IO' (d) RD'

(C) With the help of diagram, Show how M/IO' signal can be utilized while interfacing the memory with 8086. [2]

Q.2 Attempt the following.(*Any two*) **[12]**

(A) Draw a schematic diagram to interface 128KB of RAM and 64KB of ROM with the 8086 system bus at appropriate address. Use 64KB of RAM and 32KB of ROM memory chips. Also show the starting and ending address of each of memory chips in design. [6]

(B) (i)While interfacing Memory with Pentium-IV, Which of address line will be ignored in connection with address lines of memory chips? Why? [3]
(ii) List the priority order of 8086 interrupts. Also explain clearly, which ISR will be completed first if divide-0 and NMI interrupt occurred simultaneously [3]

(C) (i)Answer the following with respected to **fig.1**
(1) What is the role of INTA? [1]
(2) Mention the range of interrupt type number that can be generated from the DIP switch setting. [2]
(ii) Answer the followings with respect to protected virtual mode. [1]
(1).What will be the maximum size of segment in case of the descriptor's G bit is equal to
(a) 0 and (b) 1
(2) Give at least two differences related to the segments in 8086 and the Advanced processor. [1]
(3) What are Current Privilege Level (CPL) AND Descriptor Privilege Level (DPL)? [1]

Q.3 Do as directed. **[6]**

(A) Choose the most appropriate alternate (s)
i) The number of instructions required for instruction ADD R1,R2,R3 are [1]
a) 1 b) 2 c) 3 d) 4
ii) The instruction that perform the operation R1=[R2+(R3*4)] is [1]
a) LDR R1,[R2,R3,LSL #2] b) LDR R1,[R2,R3,LSL #1]
c) LDR R1,[R2,R3,LSR #2] d) LDR R1,[R2,R3,LSR #1]
iii) The instruction used for switching between ARM and THUMB mode is [1]
a) B LABEL b) BX Rn c) BL LABEL d) all of above
iv) Number of interrupt sources that should be configured as FIQ are [1]
a) 1 b) 2 c) 3 d) n

(B) Show with calculations how an immediate number 4000 is represented in binary encoding of data processing instructions. [2]

Q.4 Attempt the following. **[12]**

(A) The offset field for the instruction "B TARGET" is 0x10. Find the address of the label "TARGET" if above instruction is executed in i) ARM mode ii)THUMB mode. [6]
Note: The offset field for the branch instruction is 24 bits wide in ARM mode and 11 bits wide in THUMB mode

(B) Explain execution of STR instructions with help of datapath organization of ARM7 core. [6]

OR

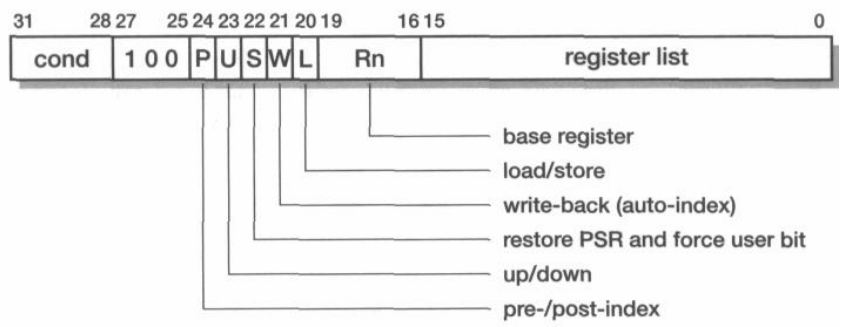
OR

Q.4 Attempt the following. **[12]**

Compare ARM and THUMB instructions with respect to [6]

(A) a) no. of machine cycles b) conditional execution c) available operand registers d) updation of flags e) no.of operands f) size of operand registers

(B) Obtain machine code (in HEX) for the instruction LDMIA R3!, {R5-R8}.The opcode encoding for the instruction is as given below. [6]



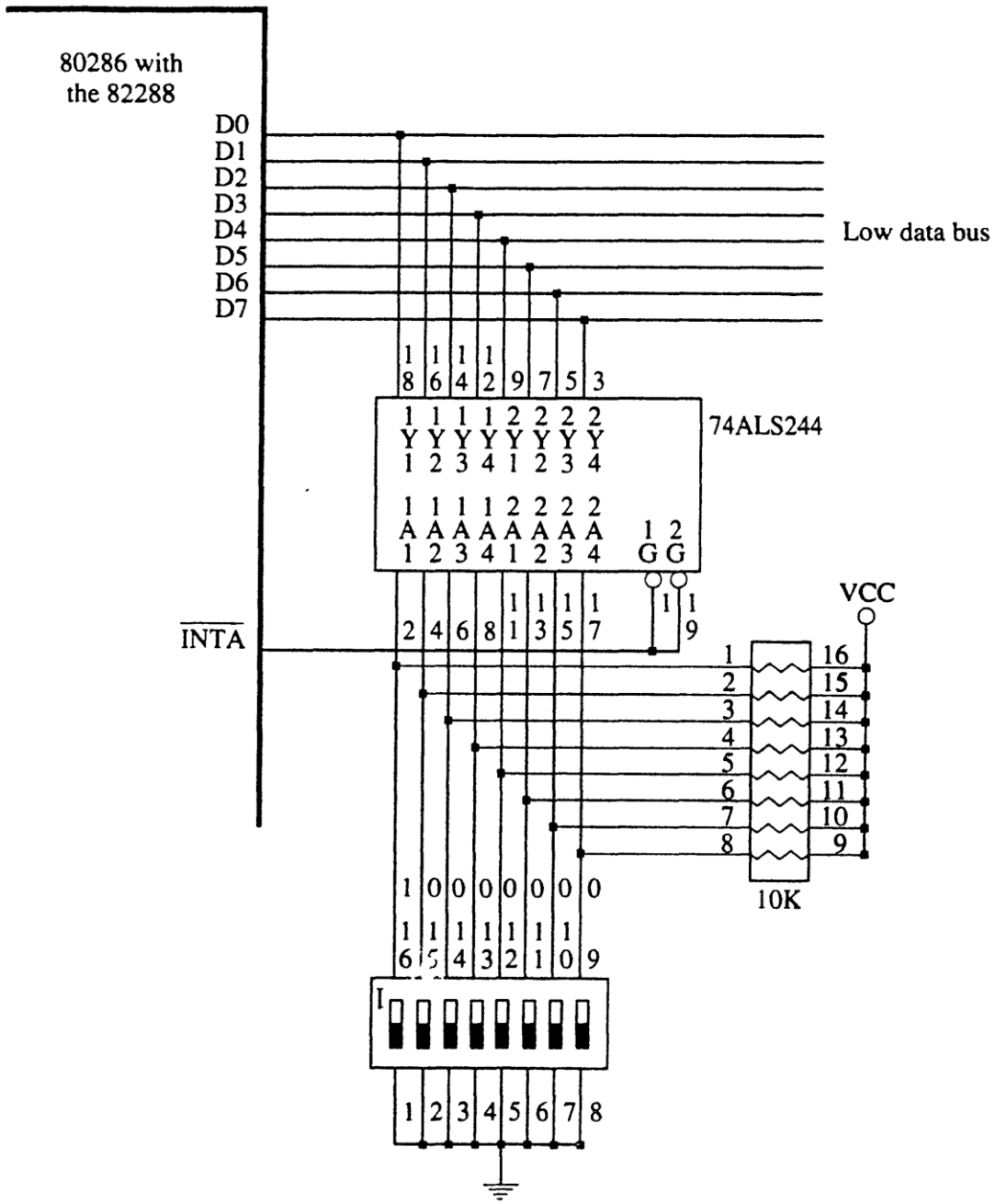


Fig.1