



**MANIPAL INSTITUTE OF TECHNOLOGY**  
**MANIPAL**  
*(A constituent unit of MAHE, Manipal)*

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**



**CERTIFICATE**

This is to certify that Ms./Mr. ....

Reg. No. ..... Section: ..... Roll No: ..... has satisfactorily completed the lab exercises prescribed for Embedded systems 1ab [CSE 2263] of Second Year B. Tech. in Computer Science and Engineering Degree at MIT, Manipal, in the academic year 2019-2020.

Date: .....

Signature  
Faculty in Charge



## **CONTENTS**

<b>LAB NO</b>	<b>TITLE</b>	<b>PAGENO.</b>
	COURSE OBJECTIVES AND OUTCOMES	i
	EVALUATION PLAN	i
	INSTRUCTIONS TO THE STUDENTS	ii – iii
	SAMPLE LAB OBSERVATION NOTE PREPARATION	4
1	INTRODUCTION TO KEIL $\mu$ VISION-4 AND PROGRAMS ON DATA TRANSFER INSTRUCTIONS	5
2	PROGRAMS ON ARITHMETIC INSTRUCTIONS	21
3	PROGRAMS ON ARITHMETIC AND LOGICAL INSTRUCTIONS	23
4	BRANCHING AND LOOPING, CODE CONVERSION	26
5	SORTING, SEARCHING PROGRAMS	28
6	INTERFACING LED TO ARM MICROCONTROLLER.	31
7	PROGRAMS ON MULTIPLEXED SEVEN SEGMENT DISPLAY	37
8	LIQUID CRYSTAL DISPLAY (LCD) AND KEYBOARD INTERFACING	44

9	ANALOG TO DIGITAL CONVERTOR PROGRAM	55
10	PROGRAM ON DIGITAL TO ANALOG CONVERTOR (DAC)	59
11	PROGRAM ON PULSE WIDTH MODULATION (PWM)	62
12	PROGRAM ON STEPPER MOTOR	66
	APPENDIX A	70
	APPENDIX B	74
	APPENDIX C	78

## **Course Objectives**

- To gain knowledge about assembly language and Embedded C programming
- To implement the programs using ARM instruction set
- To understand various interfacing circuits necessary for various applications and programming using ARM.

## **Course Outcomes**

On the completion of this laboratory course, the students will be able to:

- Gain knowledge about simulator for an embedded system and to execute simple programs.
- Comprehend the software development for ARM cortex-M microcontroller using assembly language.
- Develop embedded C program for ARM cortex-M microcontroller by interfacing various modules to ARM kit

## **Evaluation plan**

- Internal Assessment Marks : 60%
  - ✓ Continuous evaluation component (for each experiment):10 marks
  - ✓ The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce
  - ✓ Total marks of the 12 experiments reduced to marks out of 60
- End semester assessment of 2 hour duration: 40 %

## **INSTRUCTIONS TO THE STUDENTS**

### **Pre- Lab Session Instructions**

1. Students should carry the Class notes, Lab Manual and the required stationery to every lab session
2. Be in time and follow the Instructions from Lab Instructors
3. Must Sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum

### **In- Lab Session Instructions**

- Follow the instructions on the allotted exercises given in Lab Manual
- Show the program and results to the instructors on completion of experiments
- On receiving approval from the instructor, copy the program and results in the Lab record
- Prescribed textbooks and class notes can be kept ready for reference if required

### **General Instructions for the exercises in Lab**

- The programs should meet the following criteria:
  - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
  - Use meaningful names for variables and procedures.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty during evaluation.
- The exercises for each week are divided under three sets:
  - Solved exercise
  - Lab exercises - to be completed during lab hours
  - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill
- In case a student misses a lab class, he/ she must ensure that the experiment is completed at students end or in a repetition class (if available) with the permission of the faculty concerned but credit will be given only to one day's experiment(s).

- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.
- A sample note preparation is given later in the manual as a model for observation.

*Sample lab observation note preparation*

LAB NO: 1

Date:

**Title: INTRODUCTION TO KEIL  $\mu$ VISION-4 AND PROGRAMS ON DATA TRANSFER INSTRUCTIONS**

Add two immediate values in the registers and store the result in third register.

**Program:**

AREA RESET, DATA, READONLY

EXPORT \_\_Vectors

\_\_Vectors

    DCD 0X10001000

    DCD Reset\_Handler

    ALIGN

AREA mycode, CODE, READONLY

ENTRY

EXPORT Reset\_Handler

Reset\_Handler

    MOV R0, #10

    MOV R1, #3

    ADD R2, R0, R1

END

**Sample output:**

R0	0x0000000A
R1	0x00000003
R2	0x0000000D

## INTRODUCTION TO KEIL µVISION-4 AND PROGRAMS ON DATA TRANSFER INSTRUCTIONS

### Objectives:

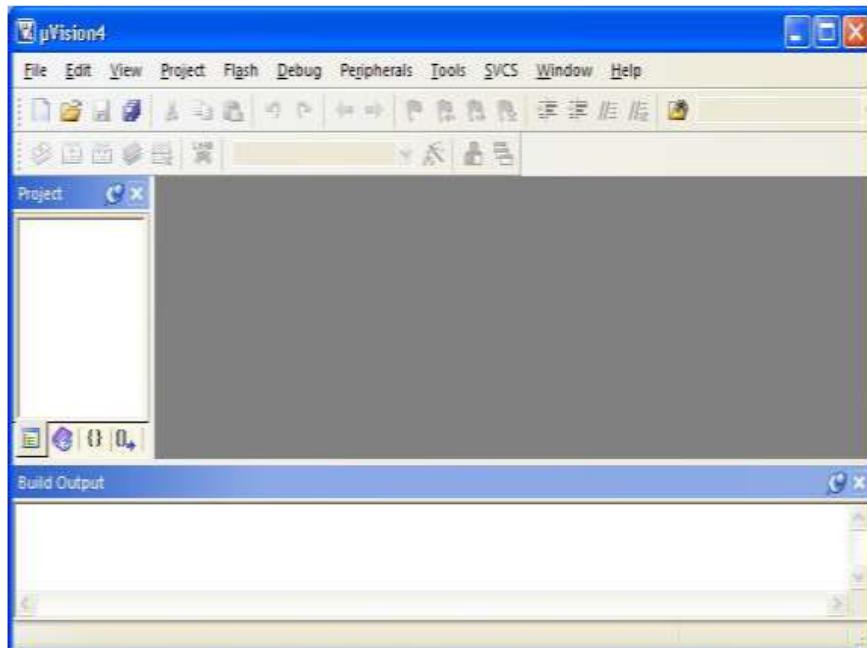
In this lab, students will be able to

- Understand the usage of Keil u Vision 4 software for assembly language.
- Write, build and execute assembly language programs in Keil u Vision 4.
- Use different data transfer instructions of ARM processor

### I. Running an assembly language program in Keil u Vision 4

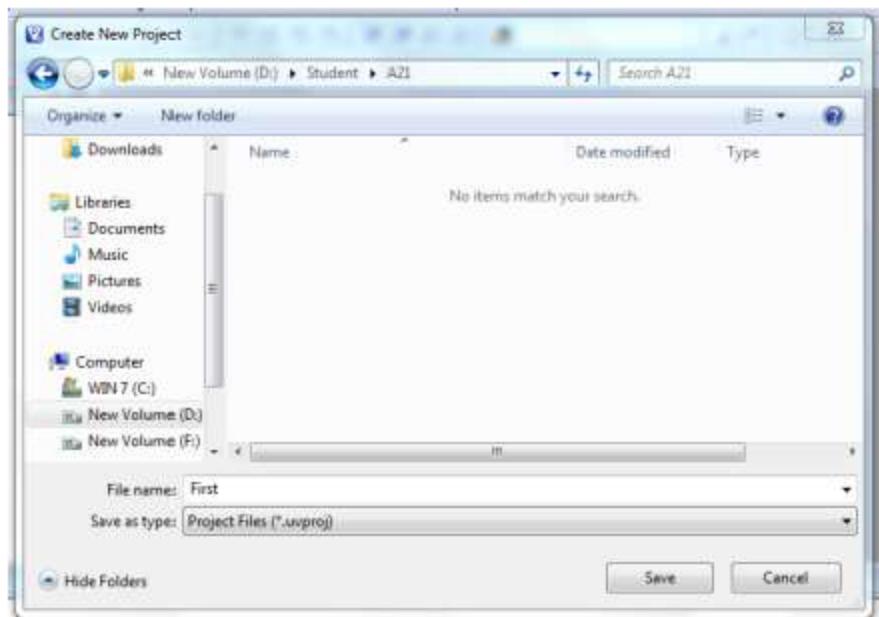
#### Step 1:

- Create a directory with section followed by roll number (to be unique); e.g. A21
- Start up uVision4 by clicking on the icon  from the desktop or from the "Start" menu or "All Programs". The following screen appears.

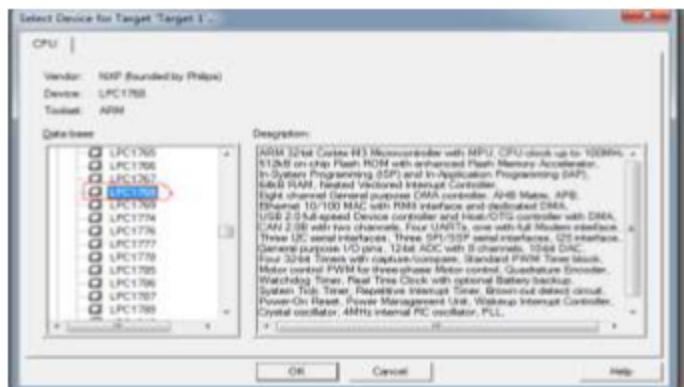


**Step 2:** Create a project

To create a project, click on the "Project" menu from the uVision4 screen and select "New uVision Project". Then, select the folder you have created already, give project name and save.



From the "Select Device for Target Target 1..." window, select "NXP" as the vendor. In that select LPC1768 ARM controller, then click on OK button. Some general information of the chip is shown in the description box.



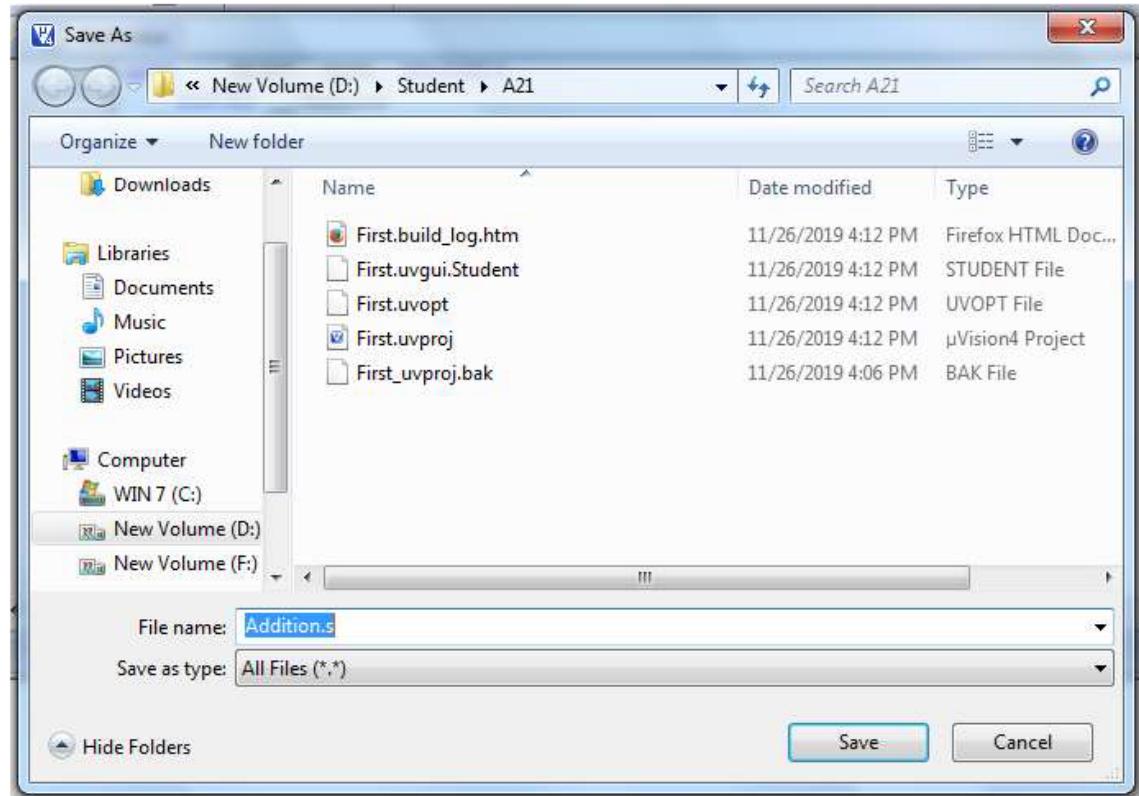
Make sure you click on "NO" for the following pop up window.

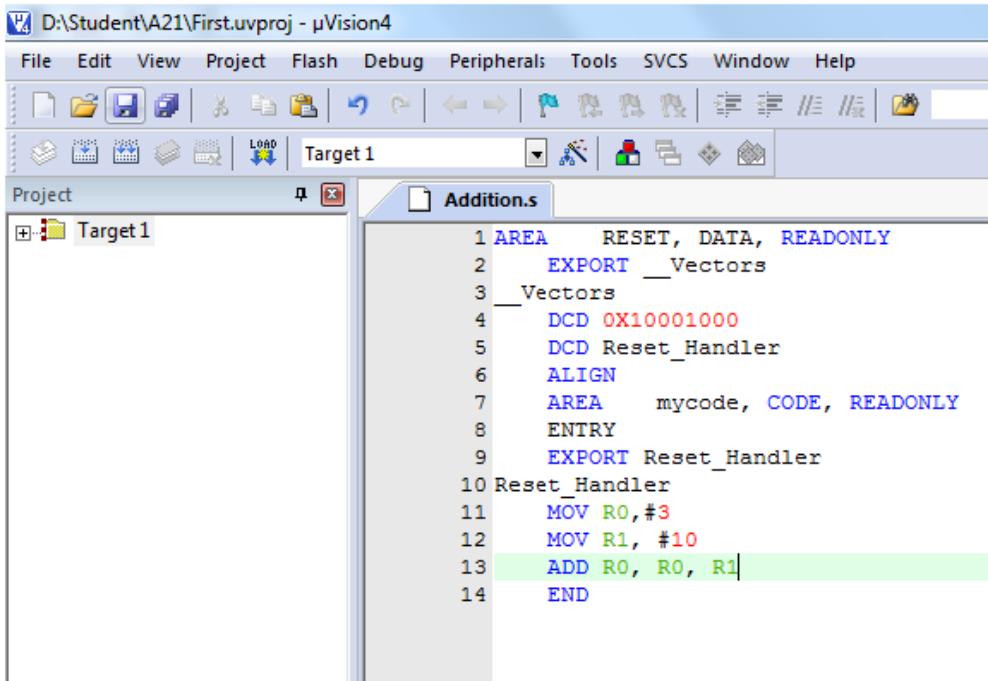


### Step 3: Create Source File

From the "File" menu, select "New", to get the editor window. Type the program here. (Note: give a tab space at the beginning). Save the program with .s extension in the directory.

## LAB NO 2





The screenshot shows the Keil uVision4 IDE interface. The title bar reads "D:\Student\A21\First.uvproj - μVision4". The menu bar includes File, Edit, View, Project, Flash, Debug, Peripherals, Tools, SVCS, Window, and Help. The toolbar contains various icons for file operations like Open, Save, and Build. The project tree on the left shows a folder named "Target1". The main workspace displays an assembly file named "Addition.s" with the following code:

```

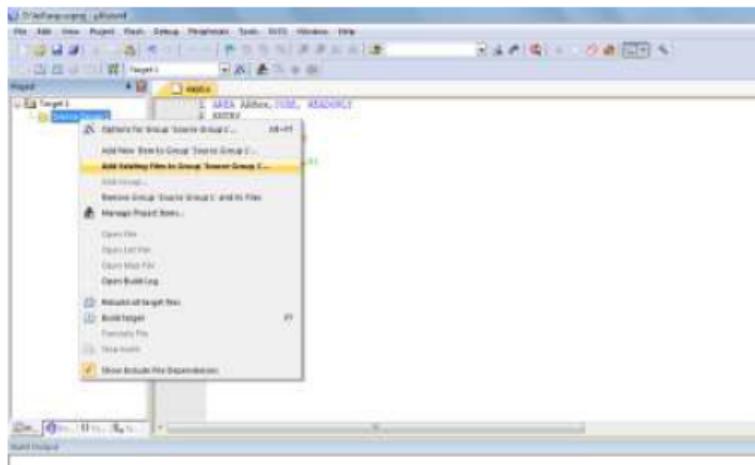
1 AREA      RESET, DATA, READONLY
2   EXPORT __Vectors
3 __Vectors
4   DCD 0X10001000
5   DCD Reset_Handler
6   ALIGN
7   AREA      mycode, CODE, READONLY
8   ENTRY
9   EXPORT Reset_Handler
10 Reset_Handler
11   MOV R0,#3
12   MOV R1, #10
13   ADD R0, R0, R1
14 END

```

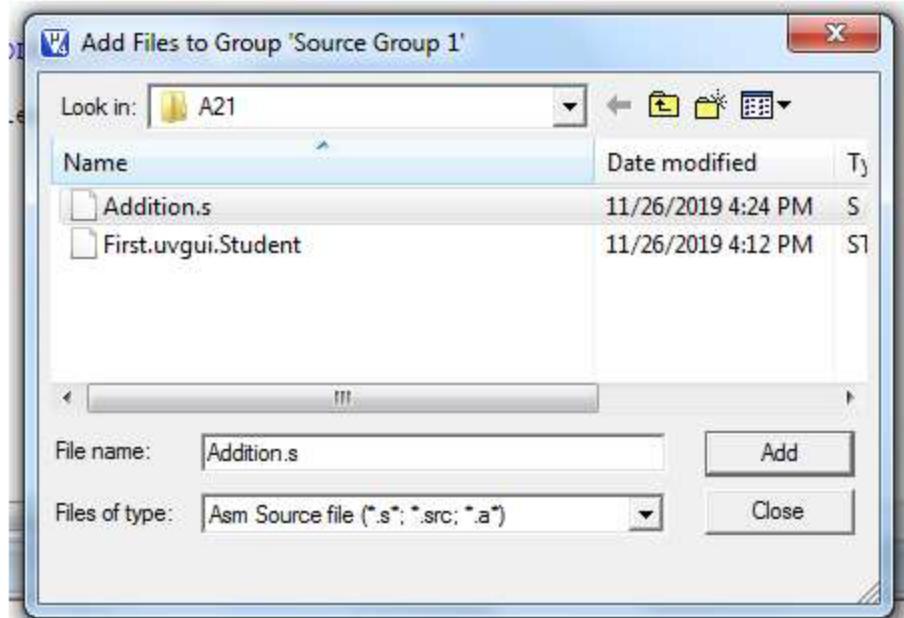
The line "ADD R0, R0, R1" is highlighted with a green background.

#### Step 4: Add Source File to the Project

Click on the + symbol near the Target 1 in the top left corner of the window. Right click on the "Source Group 1", select "Add Existing Files to Group 'Source Group 1'".



Select "Files of type" as "asm Source file (\*.s; \*.src; \*.a)", then select the file. Click on "Add", and then click on "Close".



### **Step 5: Build your project**

Click on the "+" beside the "Source Group 1", you will see the program “ Addition.s”  
Click on the "Build" button or from the "Project" menu, you will see the following screen.

The screenshot shows the Keil µVision 4 IDE interface. The top menu bar includes File, Edit, View, Project, Flash, Debug, Peripherals, Tools, SVCS, Window, and Help. Below the menu is a toolbar with various icons. The project tree on the left shows 'Target 1' with a single source group 'Source Group 1' containing a file named 'Addition.s'. The main workspace displays the assembly code for this file:

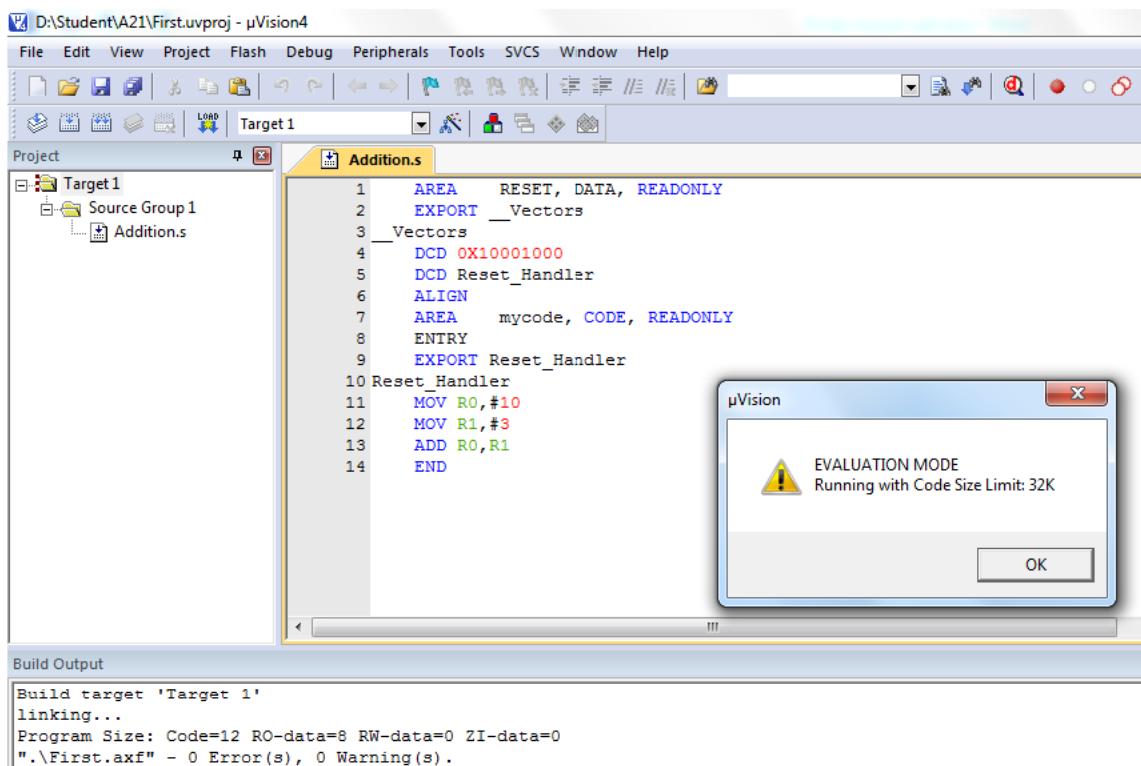
```
1 AREA      RESET, DATA, READONLY
2 EXPORT __Vectors
3 __Vectors
4 DCD 0X10001000
5 DCD Reset_Handler
6 ALIGN
7 AREA      mycode, CODE, READONLY
8 ENTRY
9 EXPORT Reset_Handler
10 Reset_Handler
11 MOV R0,#10
12 MOV R1,#3
13 ADD R0,R1
14 END
```

Below the workspace is a 'Build Output' window showing the build process and results:

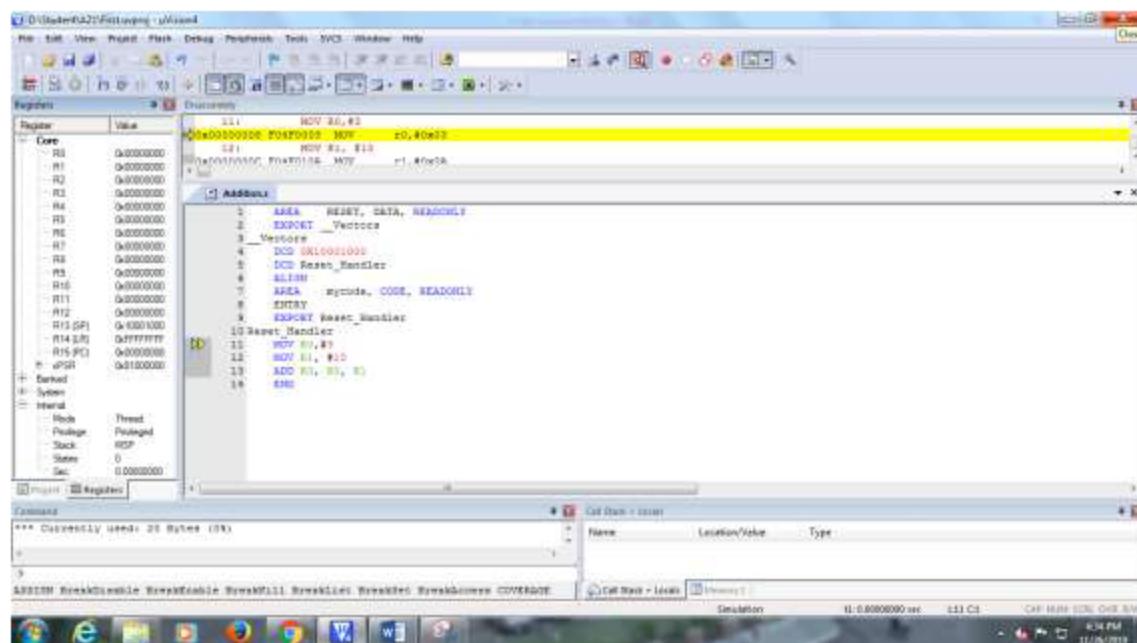
```
Build target 'Target 1'
linking...
Program Size: Code=12 RO-data=8 RW-data=0 ZI-data=0
".\First.axf" - 0 Error(s), 0 Warning(s).
```

### Step 6: Run the program

Run the program through "Debug" menu.

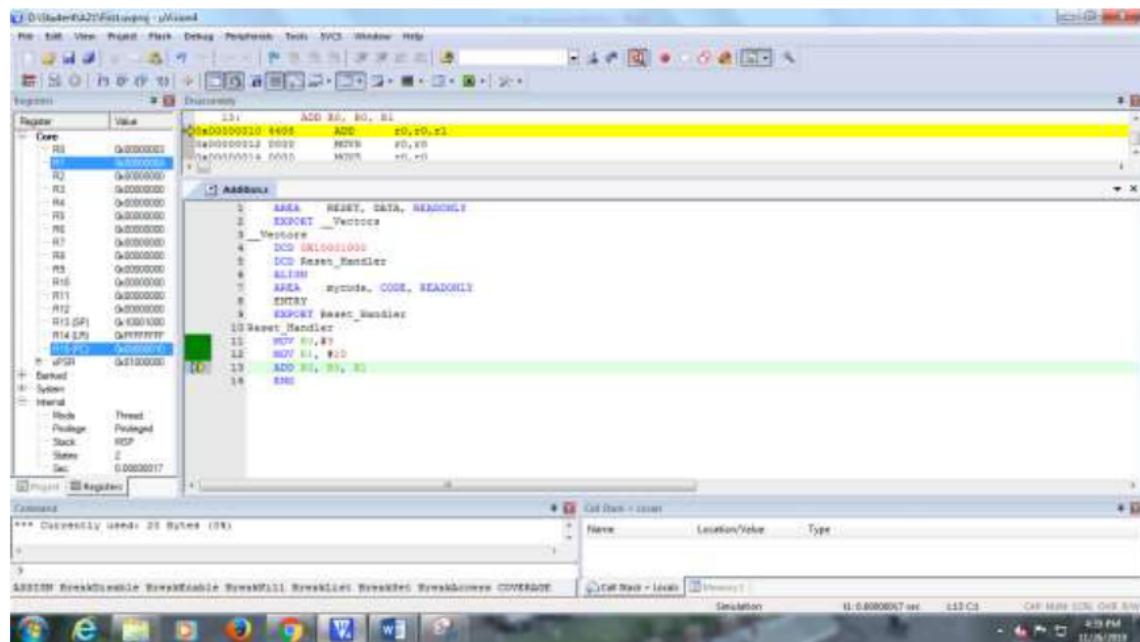


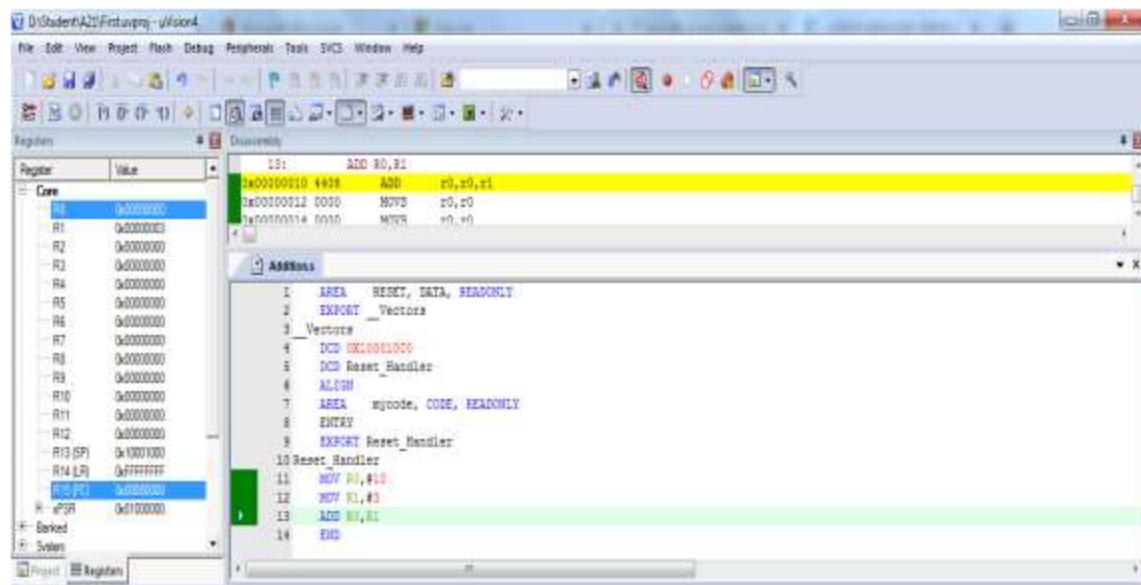
Click on "OK" for the pop up window showing "EVALUATION MODE, Running with Code Size Limit: 32K". You will see the following window.



Open uVision4 to full screen to have a better and complete view. The left hand side window shows the registers and the right side window shows the program code. There are some other windows open. Adjust the size of them to have better view. Run the program step by step; observe the change of the values in the registers.

Run the program using the **Step Over** button or click on **Step Over** from the Debug menu. It executes the instructions of the program one after another. To trace the program one can use the **Step** button, as well. The difference between the **Step Over** and **Step** is in executing functions. While **Step** goes into the function and executes its instructions one by one, **Step Over** executes the function completely and goes to the instruction next to the function. To see the difference between them, trace the program once with **Step Over** and then with **Step**. When PC is executing the function and want the function to be executed completely one can use **Step Out**. In this case, the instructions of the function will be executed, it returns from the function, and goes to the instruction which is next to the function call.





Click on the "Start/Stop Debug Session" again to stop execution of the program.

## II. ARM assembly language module

An ARM assembly language module has several constituent parts.

These are:

- Extensible Linking Format (ELF) sections (defined by the AREA directive).
- Application entry (defined by the ENTRY directive).
- Program end (defined by the END directive).

### Assembler Directives

- Assembler directives are the commands to the assembler that direct the assembly process.
- They do not generate any machine code i.e. they do not contribute to the final size of machine code and they are assembler specific

#### AREA:

The AREA directive tells the assembler to define a new section of memory. The memory can be code (instructions) or data and can have attributes such as READONLY,

READWRITE and so on. This is used to define one or more blocks of indivisible memory for code or data to be used by the linker. The following is the format:

**AREA sectionname attribute, attribute, ...**

The following line defines a new area named mycode which has CODE and REASDONLY attributes:

**AREA mycode, CODE, READONLY**

Commonly used attributes are CODE, DATA, READONLY, READWRITE, ALIGN and END.

### **READONLY:**

It is an attribute given to an area of memory which can only be read from. It is by default for CODE. This area is used to write the instructions.

### **READWRITE:**

It is attribute given to an area of memory which can be read from and written to. It is by default for DATA.

### **CODE:**

It is an attribute given to an area of memory used for executable machine instructions. It is by default READONLY memory.

### **DATA:**

It is an attribute given to an area of memory used for data and no instructions can be placed in this area. It is by default READWRITE memory.

### **ALIGN:**

It is an attribute given to an area of memory to indicate how memory should be allocated according to the addresses. When the ALIGN is used for CODE and READONLY, it is aligned in 4-bytes address boundary by default since the ARM instructions are 32 bit word. If it is written as ALIGN = 3, it indicates that the information should be placed in memory with addresses of  $2^3$ , that is for example 0x50000, 0x50008, 0x50010, 0x50018 and so on.

**EXPORT:**

The EXPORT directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files.

**DCD (Define constant word):**

Allocates a word size memory and initializes the values. Allocates one or more words of memory, aligned on 4-byte boundaries and defines initial run time contents of the memory.

**ENTRY:**

The ENTRY directive declares an entry point to the program. It marks the first instruction to be executed. In applications using the C library, an entry point is also contained within the C library initialization code. Initialization code and exception handlers also contain entry points

**END:**

It indicates to the assembler the end of the source code. The END directive is the last line of the ARM assembly program and anything after the END directive in the source file is ignored by the assembler.

**Example:**

```
AREA RESET, DATA, READONLY
```

```
EXPORT __Vectors
```

```
__Vectors
```

```
    DCD 0X10001000 ;stack pointer value when stack is empty  
                      ;The processor uses a full descending stack.  
                      ;This means the stack pointer holds the address of the last  
                      ;stacked item in memory. When the processor pushes a new item  
                      ;onto the stack, it decrements the stack pointer and then  
                      ;writes the item to the new memory location.
```

```
    DCD Reset_Handler ; reset vector. The program linker requires Reset_Handler
```

```
ALIGN  
AREA mycode, CODE, READONLY  
ENTRY  
EXPORT Reset_Handler  
  
Reset_Handler  
;;;;;;User Code Starts from the next line;;;;;  
MOV R0, #10  
MOV R1, #3  
ADD R0, R1  
END ;End of the program
```

### **III. Introduction to ARM addressing modes**

Data can be transferred into and out of ARM controller using different addressing modes. There are different ways to specify the address of the operands for any given operations such as load, add or branch. The different ways of determining the address of the operands are called addressing modes. Different addressing modes used in ARM are listed in Appendix A.

#### **Solved Exercise:**

Write an ARM assembly language program to copy 32 bit data from code memory to data memory.

Source: SRC= 0X00000008 at location pointed by R0

Destination: DST = 0X00000008 at location pointed by R1 after the execution

#### **Program:**

```
AREA RESET, DATA, READONLY  
EXPORT __Vectors
```

Vectors

```

    DCD 0x10001000 ; stack pointer value when stack is empty
    DCD Reset_Handler ; reset vector
    ALIGN
    AREA mycode, CODE, READONLY
    ENTRY
    EXPORT Reset_Handler

```

## Reset\_Handler

```

    LDR R0, =SRC      ; Load address of SRC into R0
    LDR R1, =DST      ; Load the address of DST onto R1
    LDR R3, [R0]       ; Load data pointed by R0 into R3
    STR R3,[R1]       ; Store data from R3 into the address pointed by R1
STOP
    B STOP          ; Be there
SRC DCD 8        ;SRC location in code memory
    AREA mydata, DATA, READWRITE
DST DCD 0        ;DST location in data memory
END

```

**Observations to be made**

- Data storage into the memory:** Click on Memory window and go to Memory1 option. Type address pointed by R0 in address space and observe how the data are stored into the memory.
- Data movement from one memory to another memory:** Click on Memory window and go to Memory2 option. Type address pointed by R1 in address space and observe data movement to another location before execution and after execution.

**Lab Exercises:**

- Write an ARM assembly language program to transfer a 32 bit number from one location in the data memory to another location in the data memory.
- Write an ARM assembly language program to transfer block of ten 32 bit numbers from one location to another

- a. When the source and destination blocks are non-overlapping (from code memory to data memory)

- b. When the source and destination blocks are overlapping

Hint: Use Register indirect addressing mode or indexed addressing mode.

3. Reverse an array of ten 32 bit numbers in the memory.

**Additional Exercises:**

1. Store two byte values in memory when
  - a) No ALIGN directive is used
  - b) ALIGN = 2
  - c) ALIGN = 4
2. Repeat 2. a) above using pre indexing mode.

**LAB NO: 2****Date:****PROGRAMS ON ARITHMETIC INSTRUCTIONS****Objectives:**

In this lab, students will be able to

- Identify and use the instructions required to perform addition and subtraction
- Debug and trace the programs.

Refer Appendix A for instruction details.

**Solved Exercise:**

Write a program to add two 32 bit numbers available in the code memory. Store the result in the data memory

```
AREA RESET, DATA, READONLY  
EXPORT __Vectors
```

```
__Vectors  
  DCD 0x40001000 ; stack pointer value when stack is empty  
  DCD Reset_Handler ; reset vector
```

```
ALIGN  
AREA mycode, CODE, READONLY  
ENTRY  
EXPORT Reset_Handler
```

```
Reset_Handler  
  LDR R0, =VALUE1 ;pointer to the first value1  
  LDR R1, [R0]      ;load the first value into R1  
  LDR R0, =VALU2   ;pointer to the second value  
  LDR R3, [R0]      ;load second number into r3  
  ADDS R6, R1, R3  ;add two numbers and store the result in r6  
  LDR R2, =RESULT  
  STR R6, [R2]
```

```
STOP
B STOP
VALUE1 DCD 0X12345678      ; First 32 bit number
VALUE2 DCD 0XABCDEF12      ; Second 32 bit number
AREA data, DATA, READWRITE
RESULT DCD 0
END
```

**Lab Exercises:**

1. Write a program to add ten 32 bit numbers available in code memory and store the result in data memory.
2. Write a program to add two 128 bit numbers available in code memory and store the result in data memory.  
Hint: Use indexed addressing mode.
3. Write a program to subtract two 32 bit numbers available in the code memory and store the result in the data memory.
4. Write a program to subtract two 128 bit numbers available in the code memory and store the result in the data memory.

**Additional Exercises:**

1. Write a program to find the 2's complement of a 64 bit data in R0 and R1 register. The R0 hold the lower 32 bit.
2. Add and subtract two 32 bit numbers and check all the flags. Take appropriate data to check all the flags.

**PROGRAMS ON ARITHMETIC AND LOGICAL INSTRUCTIONS****Objectives:**

In this lab, students will be able to

- Identify and use the instructions required to perform multiplication, division and logical operations
- Debug and trace the programs.

Refer Appendix A for instruction details.

**Solved Exercise:**

Write an assembly program to multiply two 32 bit numbers

AREA RESET, DATA, READONLY

EXPORT \_\_Vectors

\_\_Vectors

    DCD 0x40001000 ; stack pointer value when stack is empty

    DCD Reset\_Handler ; reset vector

    ALIGN

AREA mycode, CODE, READONLY

ENTRY

EXPORT Reset\_Handler

Reset\_Handler

    LDR R1, =VALUE1 ;pointer to the first value1

    LDR R2,=VALUE2 ;pointer to the second value

    UMULL R3, R4, R2, R1 ;Multiply the values from R1 and R2 and store  
                               ;least significant 32 bit number into R3 and most  
                               ;significant 32 bit number into R4.

    LDR R2, =RESULT

    STR R4, [R2]

```
ADD R2, #4
STR R3, [R2] ; store result in memory
STOP
B STOP
VALUE1 DCD 0X54000000 ; First 32 bit number
VALUE2 DCD 0X10000002 ; Second 32 bit number
AREA data, DATA, READWRITE
RESULT DCD 0
```

**Note: If the result within 32 bits, use MUL instruction.**

### Lab Exercises:

1. Write a program to multiply two 32 bit numbers using repetitive addition

Hint: If two numbers are in R0 and R1 Registers then use following algorithm  
Sum=0;

```
do { sum=sum+R0; R1--; ;Use ADDS instruction for addition and use ADD
      ;instruction to increment a register by 1
```

if carry then

```
R2++; ;Increment carry value by one.
```

```
} while(R1!=0); ;Use Compare instruction to check greater
                  ;than or not. And Brach instructions for loop
```

Result= R2 and R0

2. Repeat the above program for BCD multiplication
3. Find the sum of ‘n’ natural numbers using MLA instruction.
4. Write an assembly language program to find GCD of two numbers

Hint:

```
While(a!=b)
```

```
{
```

```
  If(a>b)
```

```
    a=a-b;
```

```
  else
```

```
    b=b-a;
```

```
} Return (a);
```

5. Write an assembly language program to find LCM of two numbers

**Additional Exercises:**

1. Write an assembly language program to generate Fibonacci series.
2. Write an assembly language program to divide a 32-bit number by 16-bit number by repetitive subtraction
3. Check whether a given number is even or odd.

**BRANCHING AND LOOPING, CODE CONVERSION****Objectives:**

In this lab, students will be able to

- Learn different kinds of branching instructions.
- Understand looping code conversion programs

**Solved Exercise:**

Write an assembly program to convert a 2 digit hexadecimal number into unpacked ASCII.

```
AREA RESET, DATA, READONLY
EXPORT __Vectors
```

\_\_Vectors

```
CD 0x40001000 ; stack pointer value when stack is empty
CD Reset_Handler ; reset vector
ALIGN
AREA mycode, CODE, READONLY
ENTRY
EXPORT Reset_Handler
Reset_Handler
    LDR R0,=NUM
    LDR R3,=RESULT
    LDRB R1,[R0] ; load hex number into register R1
    AND R2,R1,#0x0F ; mask upper 4 bits
    CMP R2,#09 ; compare the digit with 09
    BLS DOWN ; if it is lower than 9 then jump to down label
    ADD R2,#07 ;else add 07 to that number
DOWN
    ADD R2,#0x30 ; Add 30H to the number, Ascii value of first digit
    STRB R2,[R3]
    AND R4,R1,#0xF0 ; Mask the second digit
```

```
MOV R4,R4,LSR#04 ; Shift right by 4 bits
CMP R4,#09          ; check for >9 or not
BLS DOWN1
ADD R3,#07

DOWN1
ADD R4,#0x30      ; Ascii value of second digit
STRB R4,[R3,#01]
NUM DCD 0x0000003A
AREA data, DATA, READWRITE
RESULT DCD 0
END
```

**Lab Exercises:**

1. Write an ARM assembly language program to convert 2-digit hexadecimal number in ASCII unpacked form into its equivalent packed hexadecimal number
2. Write an ARM assembly language program to convert a 32 bit number in the unpacked form into packed form.
3. Write an assembly language program to convert a 2-digit BCD number in to its equivalent hexadecimal number.
4. Write an assembly language program to convert a 2-digit hex number in to its equivalent BCD number

**Additional Exercises:**

1. Write an assembly language program to unpack a 32 bit BCD number into eight 32-bit numbers
2. Write an assembly language program to unpack a 32 bit BCD number into eight 32-bit ASCII numbers
3. Write an assembly language program to find the sum of bits (no. of 1's) of a 32 bit number available in the memory.

**LAB NO: 5****Date:****SORTING AND SEARCHING PROGRAMS****Objectives:**

In this lab, students will be able to

- Perform advanced list operations in a given list or array.
- Use different branch instructions.

**Solved Exercise:**

Write an ARM ALP to sort a list using bubble sort.

```

AREA  RESET, DATA, READONLY
EXPORT __Vectors
__Vectors
    DCD 0x40001000 ; stack pointer value when stack is empty
    DCD Reset_Handler ; reset vector
    ALIGN
    AREA ascend, code, readonly
    ENTRY
Reset_Handler
    mov r4,#0
    mov r1,#10
    ldr r0, =list
    ldr r2, =result
    up   ldr r3, [r0,r4]
          str r3, [r2,r4]
          add r4, #04
          sub r1,#01
          cmp r1,#00
          bhi up
          ldr r0, =result

```

```

        mov r3, #10           ; inner loop counter
        sub r3, r3, #1
        mov r9, r3           ; R9 contain no of passes
                                ; outer loop counter
outer_loop
        mov r5, r0
        mov r4, r3           ; R4 contains no of comparison in a pass
inner_loop
        ldr r6, [r5], #4
        ldr r7, [r5]
        cmp r7, r6
                                ; swap without swap instruction
        strls r6, [r5]
        strls r7, [r5, #-4]
        subs r4, r4, #1
        bne inner_loop
        sub r3, #1
        subs r9, r9, #1
        bne outer_loop
list dcd 0x10,0x05,0x33,0x24,0x56,0x77,0x21,0x04,0x87,0x01
        AREA data1, data, readwrite
result DCW 0,0,0,0,0,0,0,0,0,0
        end

```

**Lab Exercises:**

1. Write an assembly program to sort an array using selection sort
2. Write an assembly program to find the factorial of a unsigned number using recursion
3. Write an assembly program to search an element in an array of ten 32 bit numbers using linear search.
4. Assume that ten 32 bit numbers are stored in registers R1-R10. Sort these numbers in the fully ascending stack using selection sort and store the sorted array back into the registers. Use STM and LDMDB instructions wherever necessary.

**Additional Exercises:**

1. Repeat question 4 for fully descending stack using STMDB and LDM instruction wherever necessary.
2. Write an 8086 ALP that contains a list of numbers and makes a count of
  - a) Even and Odd numbers.
  - b) Numbers greater than 10

**INTERFACING LED TO ARM MICROCONTROLLER.****Objectives:**

In this lab, students will be able to

- Interface LEDs to the ARM cortex LPC1768 microcontroller using ALS interfacing board.

**Steps to be followed****Project Creation in Keil uvision4 IDE:**

- Create a project folder before creating NEW project.
- Use separate folder for each project
- Open Keil uVision4 IDE software by double clicking on “Keil Uvision4” icon.
- Select “Project” then to “New Project” and save it with a name in the respective Project folder, which is already you created.
- Select the device as “NXP (founded by Philips)” Select “LPC1768” then Press “OK” and then press “YES” button to add “system\_LPC17xx.s” file.
- Go to “File” select “New” to open an editor window. Create a source file and use the header file “LPC17xx.h” in the source file and save the file. Color syntax highlighting will be enabled once the file is saved with a Recognized extension such as “.C”.
- Right click on “Source Group 1” and select the option “Add Files to Group 'Source Group 1' “add the. C source file(s) to the group.
- Again right click on Source Group 1 and select the option “Add Files to Group 'Source Group 1' “add the file - C:Keil\ARM\startup\NXP\LPC17xx\system\_LPC17xx.c
- Any changes made to this file at current project will directly change the source system\_LPC17xx.C file. As a result other project settings may get altered. So it is recommended to copy the file C:Keil\ARM\startup\NXP\LPC17xx\system\_LPC17xx.c to the project folder and add to the source group.
- **Important: This file should be added during each project creation.**
- Select “Project” then select “Translate” to compile the File (s).

- Select “Project”, select “Build Target” for building all source files such as “.C”, “.ASM”, “.h”, files, etc... This will create the hex file if there are no warnings & no errors.

**Solved Exercise:**

Write a program to turn on/off LED serially.

**Note: Before writing the program please check GPIO port pins available in the kit (Refer Appendix C.)**

```
#include <LPC17xx.h>

unsigned int i,j;
unsigned long LED = 0x00000010;

int main(void)
{
    SystemInit();           ;Add these two function for its internal operation
    SystemCoreClockUpdate();

    LPC_PINCON->PINSEL0 &= 0xFF0000FF
                           ;Configure Port0 PINS P0.4-P0.11 as GPIO function

    LPC_GPIO0->FIODIR |= 0x00000OFF0;
                           ;Configure P0.4-P0.11 as output port

    while(1)
    {
        LED = 0x00000010; Initial value on LED
        for(i=1;i<9;i++)      //On the LED's serially
        {
            LPC_GPIO0->FIOSET = LED;
                           ; Turn ON LED at LSB(LED connected to p0.4)
```

```
for(j=0;j<10000;j++);a random delay  
LED <<= 1; Shift the LED to the left by one unit  
  
}  
; loop for 9 times  
  
LED = 0x00000010;  
  
for(i=1;i<9;i++) //Off the LED's serially  
{  
    LPC_GPIO0->FIOCLR = LED;  
  
    ;Turn OFF LED at LSB(LED connected ;to p0.4)  
  
    for(j=0;j<10000;j++);  
    LED <<= 1;  
}  
}  
}
```

**Some Settings to be done in KEILUV4 for Executing C programs :**

- In Project Window Right click “TARGET1” and select “options for target ‘TARGET1’ select to option “Target” in that select
  1. XTAL 12.0MHz
  2. Select IROM1 (starting 0×0 size 0×8000).
  3. Select IRAM1 (starting 0×10000000 size 0×8000).
- Then go to option “Output”  
Select “Create Hex file”.
- Then go to option “Linker”  
Select use memory layout from target dialog

**Settings to be done at configuration wizard of system\_LPC17xx.c file**

- There are three clock sources for CPU. Select Oscillator clock out of three. This selection is done by CLKSRCSEL register.
- If we disable the PLL0 System clock will be bypassed directly into CPU clock divider register.

- Use CCLKCFG register for choosing the division factor of 4 to get 3MHz out of 12 MHz Oscillator frequency
- For any other peripherals use the PCLK same as CCLK.

**Follow the steps specified below to carry out the settings.**

- Double click on system\_LPC17xx.c file at project window
- Select the configuration wizard at the bottom
- Expand the icons
- Select Clock configuration
- Under System controls and Status registers  
OSCRANGE: Main Oscillator range select 1MHz to 20MHz  
OSCEN: Main oscillator enable ✓
- Under Clock source select register (CLKSRCSEL)  
CLKSRC: PLL clock source selection Main oscillator
- Disable PLL0 configuration and PLL1 configuration
- Under CPU Clock Configuration register(CCLKCFG)  
CCLKSEL: Divide value for CPU clock for PLL0 4
- Under USB Clock configuration register (USBCLKCFG)  
USBSEL: Divide value for USB clock for PLL0 4
- Under Peripheral clock selection register 0 (PCLKSEL0) and 1 (PCLKSEL1)  
select Pclk = Cclk for all.
- Under Power control for peripherals (PCONP)  
Enable the power for required peripherals
- If CLKOUT to be studied configure the Clock output configuration register as below  
CLKOUTSEL : Main Oscillator  
CLKOUTDIV : 1  
CLKOUT\_EN : ✓
- Call the functions  
SystemInit();  
SystemCoreClockUpdate(); at the beginning of the main function without missing. These functions are defined in system\_LPC17xx.c where the actual clock and other system control registers are configured.

## **LAB NO 6:**

- A small change is required in the file system \_LPC17xx.c after installation. Go to text editor:

```
#define PLL0_SETUP      0  
#define PLL1_SETUP      0  
if the above #defines are 1 then make 0
```

### **Components required**

- ALS-SDA-ARMCTXM3-01 : 1 No.
- Power supply (+5V) : 1 No.
- Cross cable for programming and serial communication: 1 No
- One working USB port in the host computer system and PC for downloading the software.
- 10 core FRC cables of 8 inch length 2 No
- USB to B type cable 1 No

### **Some Settings for downloading the program in FLASH MAGIC:**

Step1.Connect 9 pin DSUB cross cable from PC to CN9 at the board.

Step2.On the 2 way dip switch SW21. Short jumper JP3

Step3.Open flash magic 6.01

Step4.Make following setting in Flash magic(Only once)

a. Communications:

Device: LPC1768  
Com Port: COM1  
Baud Rate: 9600  
Interface: None(ISP)  
Oscillator: 12MHz

b. ERASE:

Select “Erase Blocks Used By Hex File”.

c. Hex file:

Browse and select the Hex file which you want to download.

d. Options:

Select “Verify After Programming”.

### **Go to Options -> Advanced Options->communications**

Do not select High Speed Communications, keep baud rate 115200.

**Options -> Advanced Options->Hardware config**

Select Use DTR & RTS to control RST & ISP Pin.

Select Keep RTS asserted while COM Port open.

T1 = 50ms. T2 = 100ms.

**Step5.Start:**

Click “Start” to download the hex file to the controller.

**Step6.** Connect one end of 10 pin FRC cable to CNA1, Short other end to CNA

**Step7.** Press reset controller switch SW1 and Check output on the LEDs connected to CNA1.

**Lab Exercises:**

1. Write a C program to display 8-bit binary up counter on the LEDs.
2. Write a C program to read a key and display an 8-bit up/down counter on the LEDs.  
**Hint:** Use key SW2(if SW2=1, up counter else down counter), which is available at CNB1 pin 7. Connect CNB1 to any controller connector like CNB, CNC etc. Configure corresponding port pin as GPIO using corresponding PINSEL register and input pin using corresponding FIODIR register.
3. Write a program to simulate an 8- bit ring counter with key press (SW2).

**LAB NO: 7.****Date:****PROGRAMS ON MULTIPLEXED SEVEN SEGMENT DISPLAY****Objectives**

In this lab students will be able to

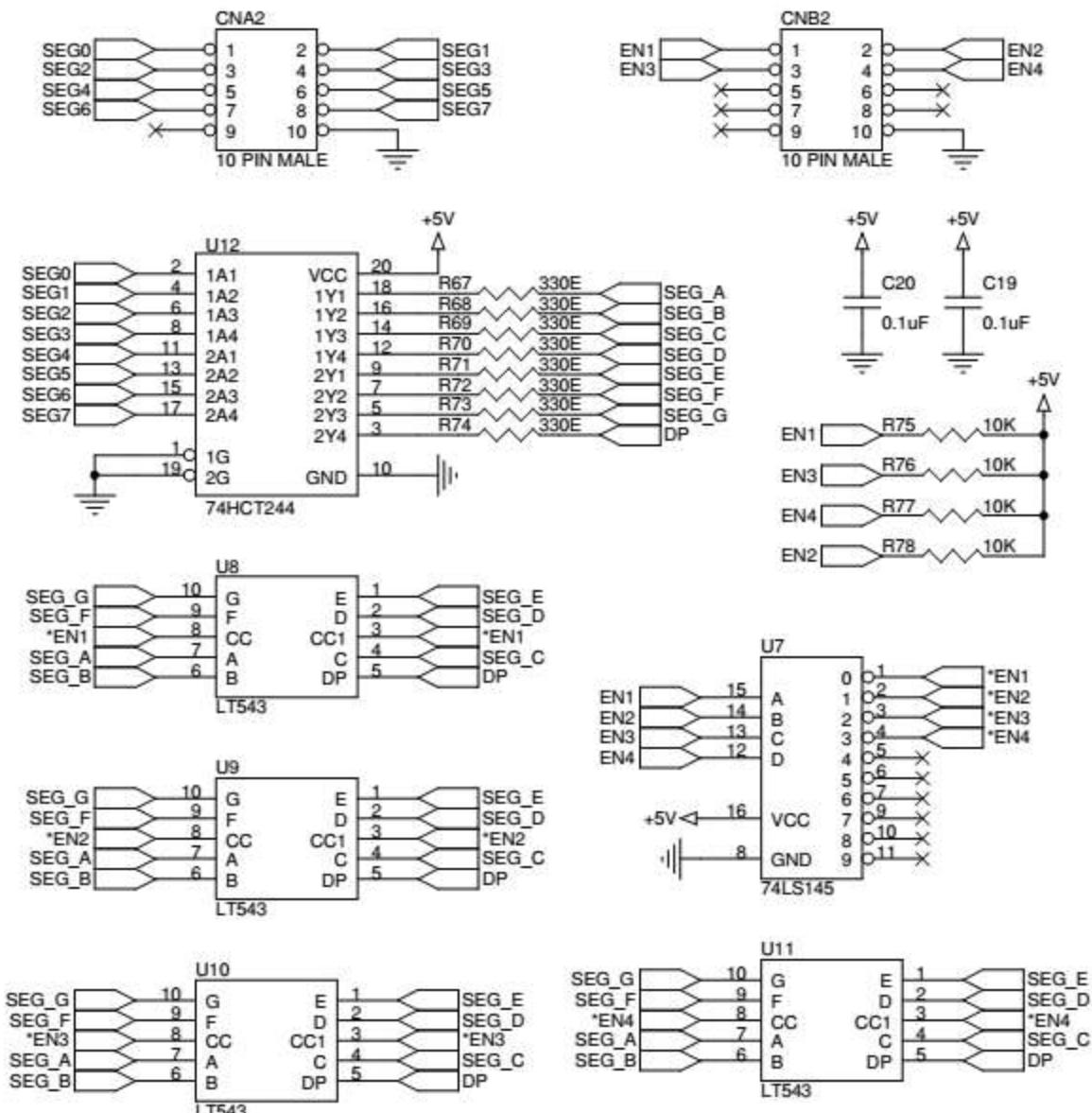
- Interface and understand the working of multiplexed seven segments display

**Introduction:**

There are four multiplexed 7-segment display units (U8, U9, U10 and U11) on the board. Each display has 8-inputs SEG\_A (Pin-7), SEG\_B (Pin-6), SEG\_C (Pin-4), SEG\_D (Pin-2), SEG\_E (Pin-1), SEG\_F (Pin-9), SEG\_G (Pin-10) and SEG\_H (Pin-5) and the remaining pins pin-3 & pin-8 are Common Cathode CC. These segments are common cathode type hence active high devices.

At power on all the segments are pulled up. A four bits input through CNB2 is used for multiplexing operation. A 1-of-10 Decoder/Driver U7 is used to accept BCD inputs and provide appropriate outputs for enabling the required display.

8 bits data is provided in this block using CNA2. All the data lines are taken buffered at U12 before giving to the displays.



SEVEN SEGMENT DISPLAY

At controller end any 2 connector are required for interfacing this block.

Lookup Table for displaying 0,1,2,3 to 9

value= h g f e d c b a	On 7-SEG U8,U9,U10 & U11.
* 0x3F = 0 0 1 1 1 1 1 1	-> Displaying '0'
* 0x06 = 0 0 0 0 0 1 1 0	-> Displaying '1'
* 0x5B = 0 1 0 1 1 0 1 1	-> Displaying '2'
* 0x4F = 0 1 0 0 1 1 1 1	-> Displaying '3'
* 0x66 = 0 1 1 0 0 1 1 0	-> Displaying '4'
* 0x6D = 0 1 1 0 1 1 0 1	-> Displaying '5'
* 0x7D = 0 1 1 1 1 1 0 1	-> Displaying '6'
* 0x07 = 0 0 0 0 0 1 1 1	-> Displaying '7'
* 0x7F = 0 1 1 1 1 1 1 1	-> Displaying '8'
* 0x6F = 0 1 1 0 1 1 1 1	-> Displaying '9'

### Solved Exercise:

WAP to simulate 4-digit BCD up counter on the multiplexed seven segment display.

```
#include <LPC17xx.h>
#include <stdio.h>

#define FIRST_SEG 0xF87FFFFF
#define SECOND_SEG 0xF8FFFFFF
#define THIRD_SEG 0xF97FFFFF
#define FOURTH_SEG 0xF9FFFFFF
#define DISABLE_ALL 0xFA7FFFFF

unsigned int dig1=0x00,dig2=0x00,dig3=0x00,dig4=0x00;
unsigned int twenty_count = 0x00,dig_count=0x00,temp1=0x00;
unsigned char
array_dec[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
unsigned char tmr0_flg = 0x00,one_sec_flg = 0x00;
unsigned long int temp2 = 0x00000000,i=0;
unsigned int temp3=0x00;
void delay(void);
void display(void);

int main(void)
{
```

```

SystemInit();
SystemCoreClockUpdate();

LPC_PINCON->PINSEL0 &= 0xFF0000FF;           //P0.4 to P0.11
                                                //GPIO data lines
LPC_PINCON->PINSEL3 &= 0xFFC03FFF;           //P1.23 to P1.26
                                                //GPIO enable lines

LPC_GPIO0->FIODIR |= 0x00000FF0;           //P0.4 to P0.11 output
LPC_GPIO1->FIODIR |= 0x07800000;           //P1.23 to P1.26 output

while(1)
{
    Delay();
    dig_count +=1;
    if(dig_count == 0x05)
        { dig_count = 0x00;
          one_sec_flg =0xFF;
        }
    if(one_sec_flg == 0xFF)
    {
        one_sec_flg = 0x00;
        dig1 +=1;

        if(dig1 == 0x0A)
        {
            dig1 = 0;
            dig2 +=1;

            if(dig2 == 0x0A)
            {
                dig2 = 0;
                dig3+=1;

                if(dig3 == 0x0A)
                {
                    dig3 = 0;
                    dig4 += 1;

                    if(dig4 == 0x0A)

```

```

        {
            dig4 = 0;
        } //end of dig4

    } //end of dig3

} //end of dig2

} //end of dig1

} //end of one_sec if

Display();

} //end of while(1)

}//end of main

void Display(void) //To Display on 7-segments
{
    if(dig_count == 0x01)      // For Segment U8
    {
        temp1 = dig1;
        LPC_GPIO1->FIOPIN = FIRST_SEG;
    }

    else if(dig_count == 0x02) // For Segment U9
    {
        temp1 = dig2;
        LPC_GPIO1->FIOPIN = SECOND_SEG;
    }

    else if(dig_count == 0x03) // For Segment U10
    {
        temp1 = dig3;
        LPC_GPIO1->FIOPIN = THIRD_SEG;
    }
}

```

```

}

else if(dig_count == 0x04) // For Segment U11
{
    temp1 = dig4;
    LPC_GPIO1->FIOPIN = FOURTH_SEG;

}
temp1 &= 0x0F;
temp2 = array_dec[temp1]; // Decoding to 7-segment
temp2 = temp2 << 4;
LPC_GPIO0->FIOPIN = temp2; // Taking Data Lines for 7-Seg
for(i=0;i<500;i++);
LPC_GPIO0->FIOCLR = 0x00000FF0;
// LPC_GPIO1->FIOPIN = DISABLE_ALL;//disable all the segments
}
Void delay(void)
{ unsigned int i;
    For(i=0;i<1000;i++);
    if(twenty_count == 1000) //multiplied by 500x2msec for
        //1 Sec
    {
        one_sec_flg = 0xFF;
        twenty_count = 0x00;
    }
    else twenty_count += 1;
}
}

```

### Components required

- ALS-SDA-ARMCTXM3-01 : 1 No.
- Power supply (+5V) : 1 No.
- Cross cable for programming and serial communication : 1 No
- One working USB in the host computer system and PC for downloading the software.
- 10 core FRC cables of 8 inch length 2 No
- USB to B type cable 1 No

**Hardware setup:** Connect a 10 core FRC cable from CNA to CNA2 and CNB to CNB2.

**Working procedure:** After software download and hardware setup, press the reset, Observe the count from 0000 to 9999 on the display.

**Lab Exercises:**

1. Write a C program to simulate a 4 digit BCD down counter. Use timer for a delay
2. Write a C program for 4 digit BCD up/down counter on seven segment using a switch and timer with a delay of 1-second between each count.
3. Write a program for 4 digit Hexadecimal up/down counter on seven segment using a switch and timer with a delay of 1-second between each count.

LAB NO: 8:

Date:

**LIQUID CRYSTAL DISPLAY (LCD) AND KEYBOARD INTERFACING****Objectives:**

In this lab students will be able to

- Interface and understand the working of LCD and matrix key board

**Introduction:**

LCD: A  $16 \times 2$  alphanumeric LCD can be used to display the message from controller.

16 pin small LCD has to be mounted to the connector CN11. 10 pin connector CNAD is used to interface this LCD from controller. Only higher 4 data lines are used among the 8 LCD data lines. Use POT3 for contrast adjustment and Short the jumper JP16 to use this LCD. LCD connector CN11 is described in this table. CN11 is single row 16 pin female berg.

<b>Pin no CN11</b>	<b>Description</b>
1	Ground
2	+5V
3	LCD contrast
4	RS
5,7,8,9,10	NC
6	En
11 to 14	Data 4 to 7
15	Back light anode
16	Back light cathode

Connection from CNAD to LCD connector CN11 is shown below

<b>Pin no at CNAD</b>	<b>Description</b>	<b>Pin no at CN11</b>
1	L0 – Data line 4 of LCD	11
2	L1 – Data line 5 of LCD	12
3	L2 – Data line 6 of LCD	13
4	L3 – Data line 7 of LCD	14
5	L5 – Command line of LCD	4
6	L5 – Enable line of LCD	6

## LAB NO 8

Instruction	RS	R/W	Code								Execution Time (max) (when $f_{op}$ or $f_{osc}$ is 270 kHz)	
			DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	
Write data to CG or DDRAM	1	0	Write data								Writes data into DDRAM or CGRAM.	37 µs $t_{add} = 4 \mu s^*$
Read data from CG or DDRAM	1	1	Read data								Reads data from DDRAM or CGRAM.	37 µs $t_{add} = 4 \mu s^*$
			I/D = 1: Increment								DDRAM: Display data RAM	Execution time changes when frequency changes
			I/D = 0: Decrement								CGRAM: Character generator RAM	Example:
			S = 1: Accompanies display shift								ACG: CGRAM address	When $f_{op}$ or $f_{osc}$ is 250 kHz,
			S/C = 1: Display shift								ADD: DDRAM address (corresponds to cursor address)	$37 \mu s \times \frac{270}{250} = 40 \mu s$
			S/C = 0: Cursor move								AC: Address counter used for both DD and CGRAM addresses	
			R/L = 1: Shift to the right									
			R/L = 0: Shift to the left									
			DL = 1: 8 bits, DL = 0: 4 bits									
			N = 1: 2 lines, N = 0: 1 line									
			F = 1: 5 × 10 dots, F = 0: 5 × 8 dots									
			BF = 1: Internally operating									
			BF = 0: Instructions acceptable									

Note: — indicates no effect.

- \* After execution of the CGRAM/DDRAM data write or read instruction, the RAM address counter is incremented or decremented by 1. The RAM address counter is updated after the busy flag turns off. In Figure 10,  $t_{add}$  is the time elapsed after the busy flag turns off until the address counter is updated.

Instruction	RS	R/W	Code								Execution Time (max) (when $f_{op}$ or $f_{osc}$ is 270 kHz)	
			DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 µs
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 µs
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.	37 µs
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 µs
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 µs
Set DDRAM address	0	0	1	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 µs						
Read busy flag & address	0	1	BF	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.	0 µs						

**Solved Exercise:**

WAP to display message on LCD

```
#include <lpc17xx.h>

#define RS_CTRL 0x08000000 //P0.27
#define EN_CTRL 0x10000000 //P0.28
#define DT_CTRL 0x07800000 //P0.23 to P0.26 data lines

void lcd_init(void);
void wr_cn(void);
void clr_disp(void);
void delay_lcd(unsigned int);
void lcd_com(void);
void wr_dn(void);
void lcd_data(void);
void clear_ports(void);
void lcd_puts(unsigned char *);

extern unsigned long int temp1 , temp2;

unsigned long int temp1=0, temp2=0 ;

int main(void)
{
    unsigned long adc_temp;
    unsigned int i;
    float in_vtg;
    unsigned char vtg[7],dval[7];
    unsigned char Msg3[11] = {"MIT"};
    unsigned char Msg4[12] = {"Department of ICT:"};

    SystemInit();
    SystemCoreClockUpdate();
    lcd_init();
    temp1 = 0x80;
        lcd_com();
        delay_lcd(800);
        lcd_puts(&Msg3[0]);
}
```

```
temp1 = 0xC0;
lcd_com();
delay_lcd(800);
lcd_puts(&Msg4[0]);
}

//lcd initialization
void lcd_init()
{
    /* Ports initialized as GPIO */
    LPC_PINCON->PINSEL3 &= 0xFC003FFF; //P0.23 to P0.28

    /* Setting the directions as output */
    LPC_GPIO0->FIODIR |= DT_CTRL;
    LPC_GPIO0->FIODIR |= RS_CTRL;
    LPC_GPIO0->FIODIR |= EN_CTRL;

    clear_ports();
    delay_lcd(3200);

    temp2 = (0x30<<19);
    wr_cn();
    delay_lcd(30000);

    temp2 = (0x30<<19);
    wr_cn();
    delay_lcd(30000);

    temp2 = (0x30<<19);
    wr_cn();
    delay_lcd(30000);

    temp2 = (0x20<<19);
    wr_cn();
    delay_lcd(30000);

    temp1 = 0x28;
    lcd_com();
    delay_lcd(30000);
```

```
temp1 = 0x0c;
lcd_com();
delay_lcd(800);

temp1 = 0x06;
lcd_com();
delay_lcd(800);

temp1 = 0x01;
lcd_com();
delay_lcd(10000);

temp1 = 0x80;
lcd_com();
delay_lcd(800);

return;
}

void lcd_com(void)
{
    temp2 = temp1 & 0xf0;//move data (26-8+1) times : 26 - HN
    //place, 4 - Bits
    temp2 = temp2 << 19; //data lines from 23 to 26
    wr_cn();
    temp2 = temp1 & 0x0f; //26-4+1
    temp2 = temp2 << 23;
    wr_cn();
    delay_lcd(1000);

return;
}

// command nibble o/p routine
void wr_cn(void)           //write command reg
{
    clear_ports();
    LPC_GPIO0->FIOPIN = temp2;    // Assign the value to the data
    //lines
    LPC_GPIO0->FIOCLR = RS_CTRL;      // clear bit RS
    LPC_GPIO0->FIOSET = EN_CTRL;      // EN=1
    delay_lcd(25);
```

```

LPC_GPIO0->FIOCLR = EN_CTRL;      // EN =0
return;

}

// data o/p routine which also outputs high nibble first
// and lower nibble next
void lcd_data(void)
{
    temp2 = temp1 & 0xf0;
    temp2 = temp2 << 19;
    wr_dn();
    temp2= temp1 & 0x0f;
    temp2= temp2 << 23;
    wr_dn();
    delay_lcd(1000);
    return;
}

// data nibble o/p routine
void wr_dn(void)
{
    clear_ports();

    LPC_GPIO0->FIOPIN = temp2;          // Assign the value to the
                                         //data lines
    LPC_GPIO0->FIOSET = RS_CTRL;        // set bit RS
    LPC_GPIO0->FIOSET = EN_CTRL;        // EN=1
    delay_lcd(25);
    LPC_GPIO0->FIOCLR = EN_CTRL;        // EN =0
    return;
}

void delay_lcd(unsigned int r1)
{
    unsigned int r;
    for(r=0;r<r1;r++);
    return;
}

```

```
void clr_disp(void)
{
    temp1 = 0x01;
    lcd_com();
    delay_lcd(10000);
    return;
}
void clear_ports(void)
{
/* Clearing the lines at power on */
    LPC_GPIO0->FIOCLR = DT_CTRL; //Clearing data lines
    LPC_GPIO0->FIOCLR = RS_CTRL; //Clearing RS line
    LPC_GPIO0->FIOCLR = EN_CTRL; //Clearing Enable line

    return;
}

void lcd_puts(unsigned char *buf1)
{
    unsigned int i=0;

    while(buf1[i]!='\0')
    {
        temp1 = buf1[i];
        lcd_data();
        i++;
        if(i==16)
        {
            temp1 = 0xc0;
            lcd_com();
        }
    }
    return;
}
```

**Components required**

- ALS-SDA-ARMCTXM3-01 : 1 No.
- Power supply (+5V) : 1 No.

- Cross cable for programming and serial communication: 1 No
- One working USB port in the host computer system and PC for downloading the software.
- 10 core FRC cables of 8 inch length 2 No
- USB to B type cable 1 No

**Hardware setup:**

Connect 10 pin FRC cable from CND to CNAD. Short the jumper JP16 & JP5. Use POT3 for contrast adjustment.

**Working procedure:** After software download and hardware setup, press the reset. A fixed message will display on LCD.

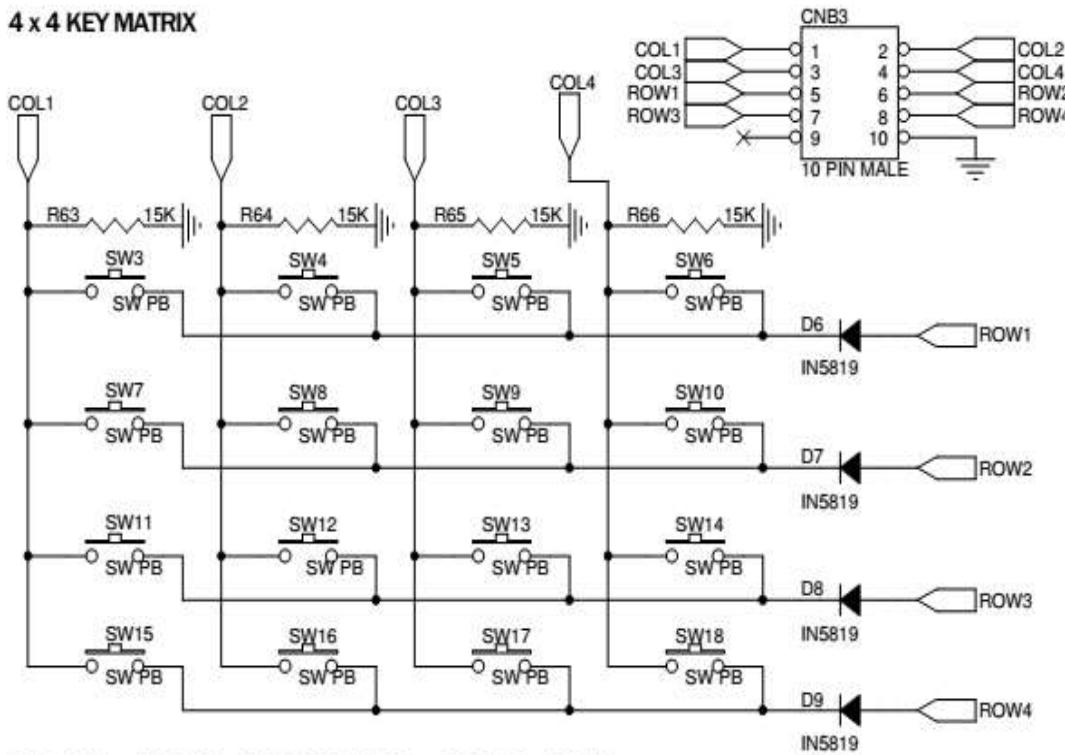
**Exercise Questions:**

1. Simulate DIE tossing on LCD

Hint: Program reads the external interrupt using the key SW2. A random number between 0-6 should be displayed on the LCD upon keypress.

**Keyboard connection:** The switches SW3 to SW18 are organized as 4 rows X 4 columns matrix. One end of all the switches are configured as columns. The other end of the matrix configured as rows. A row line will be always an output from the controller. Column lines are pulled to ground. A high level sent from the row will appear at column end if the switch is pressed.

Connector CNB3 is used for interfacing this block with controller. At the controller end any connector can be used to interact this connector CNB3.

**4 x 4 KEY MATRIX**

C1 to C4 &gt; P1.23 to P1.26. R1 to R4 &gt; P2.10 to P2.13

**Solved Exercise:**

WAP to read a key from the matrix keyboard and display its key code on the LCD.

```
#include <LPC17xx.h>
void scan(void);

unsigned char col,row,var,flag,key,*ptr;
unsigned long int i,var1,temp,temp3;

int main(void)
{
    SystemInit();
    SystemCoreClockUpdate();

    LPC_PINCON->PINSEL3 &= 0xFFC03FFF; //P1.23 to P1.26 MADE
                                         //GPIO
```

```

LPC_PINCON->PINSEL3 &= 0xF00FFFF; //P2.10 t P2.13 made
                                //GPIO
LPC_GPIO2->FIODIR |= 0x00003C00; //made output P2.10 to
                                //P2.13 (rows)
LPC_GPIO1->FIODIR &= 0xF87FFFF; //made input P1.23 to
                                //P1.26 (cols)

while(1)
{
    while(1)
    {
        for(row=1;row<5;row++)
        {
            if(row == 1)
                var1 = 0x00000400;
            else if(row == 2)
                var1 = 0x00000800;
            else if(row == 3)
                var1 = 0x00001000;
            else if(row == 4)
                var1 = 0x00002000;

            temp = var1;

            LPC_GPIO2->FIOCLR = 0x00003C00;
            LPC_GPIO2->FIOSET = var1;

            flag = 0;
            scan();
            if(flag == 1)
                break;

        } //end for(row=1;row<5;row++)

        if(flag == 1)
            break;
    } //2nd while(1)
}

void scan(void)

```

```

{
    unsigned long temp3;

    temp3 = LPC_GPIO0->FIOPIN;
    temp3 &= 0x0780000;
    if(temp3 != 0x00000000)
    {
        flag = 1;
        if (temp3 ==0x0080000)
            col=0;
        else if (temp3==0x0100000)
            col=1;
        else if (temp3==0x00200000)
            col=2;
        else if (temp3==0x0400000)
            col=3;

        } //1st if(temp3 != 0x00000000)
    } //end scan
Display(key) //write display function to display the keycode
    //on the LCD or on seven segment display
}

```

**Components required**

- ALS-SDA-ARMCTXM3-01 : 1 No.
- Power supply (+5V) : 1 No.
- Cross cable for programming and serial communication : 1 No
- One working USB port in the host computer system and PC for downloading the software.
- 10 core FRC cables of 8 inch length 2 No
- USB to B type cable 1 No

**Hardware setup:** Connect 10 core FRC cable from CNB to CNB3, short JP4(1, 2)  
 Connect another 10 core FRC cable from CND to CNAD, Short the jumper JP16 & JP5. Use POT3 for contrast.

**Working procedure:** After software download and hardware setup, use the reset. Identity of key pressed (0 to F) will be displayed on LCD.

## **LAB NO 8**

### **Lab Exercise:**

1. Write a program to input an expression of the type A operator B =, from the key board, where A and B are the single digit BCD numbers and operator may be + or - . Display the result on the LCD.

**ANALOG TO DIGITAL CONVERTOR PROGRAM****Objectives:**

In this lab students will be able to

- Understand the working of a 12 bit internal Analog-to-Digital Converter (ADC)

**Introduction:** The LPC1768 contains a single 12-bit successive approximation ADC with eight channels and DMA support. 12-bit ADC with input multiplexing among eight pins, conversion rates up to 200 kHz, and multiple result registers. The 12-bit ADC can be used with the GPDMA controller. On board there are two interfaces for internal ADC's. AD0.5 (pin P1.31) of controller is used to convert the analog input voltage varied using POT1 to digital value. AD0.4(Pin 1.30) used convert the analog voltage varied using POT4. A input voltage range of 0 to 3.3V is accepted. 000 to FFF is the converted digital voltage range here. Short JP18 (2, 3) to use AD0.4.

**Solved Exercise:**

WAP to configure and read analog data from ADC channel no 5, and display the digital data on the LCD

```
#include<LPC17xx.h>
#include<stdio.h>
#include"AN_LCD.h"
#defineRef_Vtg          3.300
#defineFull_Scale      0xFFFF           //12 bit ADC

int main(void)
{
    unsigned long adc_temp;
    unsigned int i;
    float in_vtg;
    unsigned char vtg[7],dval[7];
    unsigned char Msg3[11] = {"ANALOG IP:"};
    unsigned char Msg4[12] = {"ADC OUTPUT:"};

    SystemInit();
}
```

```

SystemCoreClockUpdate();

LPC_SC->PCONP |= (1<<15); //Power for GPIO block
lcd_init();
LPC_PINCON->PINSEL3 |= 0xC0000000; //P1.31 as AD0.5
LPC_SC->PCONP |= (1<<12);      //enable the peripheral ADC

SystemCoreClockUpdate();

temp1 = 0x80;
lcd_com();
delay_lcd(800);
lcd_puts(&Msg3[0]);

temp1 = 0xC0;
lcd_com();
delay_lcd(800);
lcd_puts(&Msg4[0]);

while(1)
{
    LPC_ADC->ADCR = (1<<5)|(1<<21)|(1<<24); //0x01200001;
    //ADC0.5, start conversion and operational
    //for(i=0;i<2000;i++); //delay for conversion
    while((adc_temp = LPC_ADC->ADGDR) == 0x80000000);
    //wait till 'done' bit is 1, indicates conversion complete
    adc_temp = LPC_ADC->ADGDR;
    adc_temp >= 4;
    adc_temp &= 0x00000FFF;           //12 bit ADC
    in_vtg = (((float)adc_temp * (float)Ref_Vtg))/((float)Full_Scale);
    //calculating input analog
    //voltage
    sprintf(vtg,"%3.2fV",in_vtg);
    //convert the readings into string to display on LCD
    sprintf(dval,"%0x",adc_temp);
    for(i=0;i<2000;i++);

    temp1 = 0x8A;
    lcd_com();
    delay_lcd(800);
}

```

```
lcd_puts(&vtg[0]);  
  
temp1 = 0xCB;  
lcd_com();  
delay_lcd(800);  
lcd_puts(&dval[0]);  
  
for(i=0;i<200000;i++);  
for(i=0;i<7;i++)  
    vtg[i] = dval[i] = 0x00;  
    adc_temp = 0;  
    in_vtg = 0;  
}  
}
```

**Components required**

- ALS-SDA-ARMCTXM3-01 : 1 No.
- Power supply (+5V) : 1 No.
- Cross cable for programming and serial communication : 1 No
- One working COM port (Ex: COM1) in the host computer system and PC for downloading the software.
- 10 core FRC cables of 8 inch length 2 No
- USB to B type cable 1 No

**Hardware Setup:** Do the setup related to LCD

**Working procedure:** Vary POT1 and observe the corresponding analog and digital voltage values on LCD.

**Exercise question**

- IV. Write a c program to display the digital value representing the difference in analog voltages at ADC channel 4 and channel 5 on LCD. Using BURST and Software mode

**LAB NO: 10****Date:****PROGRAM ON DIGITAL TO ANALOG CONVERTOR (DAC)****Objectives:**

In this lab students will be able to

- Understand the working of a 10 bit DAC and check the waveform on Cathode Ray Oscilloscope (CRO).

**Introduction:** LPC1768 has 10 bit internal DAC with dedicated conversion timer and DMA support. The DAC allows to generate a variable analog output. The maximum output value of the DAC is VREFP. The equation to calculate output voltage value is given as below.

$$AOUT = DACR \text{ value} \times ((VREFP - VREFN)/1024) + VREFN$$

An analog output from the controller can be observed in this block at TP8. Open JP5 to use this feature and use CRO to watch analog output value.

**Solved Exercise:**

WAP to generate a sawtooth waveform using DAC and display it on CRO.

```
#include <lpc17xx.h>
```

```
#define DAC_BIAS (0x1<<16) //maximum update rate of 400KHz  
#define DATA_LENGTH 0x400 //Maximum value is 0xCFF in 10 bit DAC
```

```
void DAC_Init(void);
```

```
int main (void)  
{  
    unsigned int m,i=0;  
  
    SystemInit();  
    SystemCoreClockUpdate();  
    LPC_PINCON->PINSEL1 = 0x00200000; /* set p0.26 to DAC output */  
  
    /* Initialize DAC */  
    DAC_Init();
```

```
while ( 1 )
{
    LPC_DAC->DACR = (i << 6);
//| DAC_BIAS; // AOUT = DACR value x ((VREFP - VREFN)/1024) + VREFN
    i=i+50;
    for(m = 100; m > 1; m--);
    if ( i == DATA_LENGTH ) //Maximum value is 0xCFF in 10 bit DAC
    {
        i = 0;
    }
}
}

void DAC_Init( void )
{
/* Note that the DAC does not have a control bit in the PCONP register.
   To enable the DAC, its output must be selected to appear on the
   related pin, P0.26, by configuring the PINSEL1 register */

/* setup the related pin to DAC output */

LPC_DAC->DACCNTVAL = 0x00FF;
LPC_DAC->DACCTRL = (0x1<<1)|(0x1<<2);
return;
}
```

**Components required**

- ALS-SDA-ARMCTXM3-01 : 1 No.
- Power supply (+5V) : 1 No.
- Cross cable for programming and serial communication : 1 No

## **LAB NO 10**

- One working USB port in the host computer system and PC for downloading the software.
- 10 core FRC cables of 8 inch length 2 No
- USB to B type cable 1 No

### **Hardware setup:**

Open the jumper JP5

Connect TP8 pin to CRO positive wire and TP3 to CRO negative wire. Scale the CRO to the proper display

**Working procedure:** Reset the controller and observe the analog output waveform on CRO.

### **Lab Exercises:**

1. Using DAC generate a triangular waveform with maximum possible peak-peak amplitude.
2. Using DAC, generate a variable frequency sine waveform. Use ROW-0 of keyboard for frequency variation

LAB NO: 11

Date:

**PROGRAM ON PULSE WIDTH MODULATION (PWM)****Objectives:**

In this lab students will be able to

- Interface and understand the working of PWM

**Introduction:** The PWM is based on the standard Timer block and inherits all of its features, although only the PWM function is pinned out on theLPC1768. The Timer is designed to count cycles of the system derived clock and optionally switch pins, generate interrupts or perform other actions when the specified timer values occur, based on seven match registers. The PWM function is in addition to these features, and is based on match register events. A PWM output from the controller can be observed as an intensity variation of the LED LD10.

**Solved Exercise:**

WAP to vary the intensity of an LED using PWM.

```
#include <LPC17xx.H>
void pwm_init(void);
void PWM1_IRQHandler(void);

unsigned long int i;
unsigned char flag,flag1;

int main(void)
{
    SystemInit();
    SystemCoreClockUpdate();
    pwm_init();

    while(1)
    {
        for(i=0;i<=1000;i++); // delay
    }//end of while
```

```
}//end of main
```

```
void pwm_init(void)
{
    LPC_SC->PCONP |= (1<<6);          //PWM1 is powered
    LPC_PINCON->PINSEL3 &= ~(0x0000C000);   //cleared if any other
                                            //functions are enabled
    LPC_PINCON->PINSEL3 |= 0x00008000; //pwm1.4 is selected for the pin
                                            //P1.23

    LPC_PWM1->PR = 0x00000000; //Count frequency : Fpclk
    LPC_PWM1->PCR = 0x00001000; //select PWM1 single edge
    LPC_PWM1->MCR = 0x00000003; //Reset and interrupt on PWMMR0
    LPC_PWM1->MR0 = 30000; //setup match register 0 count
    LPC_PWM1->MR4 = 0x00000100; //setup match register MR1
    LPC_PWM1->LER = 0x000000FF; //enable shadow copy register
    LPC_PWM1->TCR = 0x00000002; //RESET COUNTER AND PRESCALER
    LPC_PWM1->TCR = 0x00000009; //enable PWM and counter

    NVIC_EnableIRQ(PWM1 IRQn);
    return;
}
```

```
void PWM1_IRQHandler(void)
{
    LPC_PWM1->IR = 0xff; //clear the interrupts

    if(flag == 0x00)
    {
        LPC_PWM1->MR4 += 100;
        LPC_PWM1->LER = 0x000000FF;
```

```

if(LPC_PWM1->MR4 >= 27000)
{
    flag1 = 0xff;
    flag = 0xff;
    LPC_PWM1->LER = 0x000000ff;
}
}

else if(flag1 == 0xff)
{
    LPC_PWM1->MR4 -= 100;
    LPC_PWM1->LER = 0x000000ff;

    if(LPC_PWM1->MR4 <= 0x500)
    {
        flag = 0x00;
        flag1 = 0x00;
        LPC_PWM1->LER = 0X000000ff;
    }
}
}
}

```

**Hardware setup:** Connect 10 pin FRC cable from CNB to CNB1.

**Working procedure:** As the pulse width varies, intensity of LED LD10 varies. Observe the pulses at TP5. Observe the amplitude level at TP6.

### Lab Exercises:

Write a program to set the following intensity levels to the LED connected to PWM output. Use ROW-0 of keyboard for intensity variation

Intensity level	Key pressed
10%	0
25%	1

## **LAB NO 11**

50%	2
75%	3

LAB NO: 12

Date:

### PROGRAM ON STEPPER MOTOR

#### **Objectives**

In this lab students will be able to

- Interface and understand the working of stepper motor

**Introduction:** The Stepper motor can be interfaced to the board by connecting it to the Power Mate PM1. The direction of the rotation can be changed through software. The DC Motor can also be interfaced to the board by connecting it to the Reliamate RM5. The direction of the rotation can be changed through software.

The Relay K2 is switched between ON and OFF state. The LED L12 will toggle for every relay switch over. The contact of NO & NC of the relay can be checked at the MKDSN connector CN12 pins 1 & 2 using a CRO—these contacts can be connected to external devices. Using connector CNA5 micro controller can interface with this block.

Description of the connector pins are given in below table.

<b>Pin @ CNA5</b>	<b>Description</b>
1 to 4	Buffered from U13 used for stepper motor control
5	Buffered from U13 and connected to relay k2 coil. coil other end is connected to +5V
6	Connected to both 1 & 2 of U13; corresponding outputs of U13 are taken to NO and NC contacts of relay K1
7	One end of coil of relay K1
8	Controls the buzzer

PM1—it's a 5 pin straight male power mate. PIN descriptions are as given below.

<b>Pin no</b>	<b>Description</b>
1	+5v supply
2	Phase A
3	Phase B
4	Phase C
5	Phase D

Pin 2 to 5 are phase A to D output for the stepper motor respectively.

**Sample program:** To rotate the stepper motor in clockwise and anticlockwise direction at a particular speed continuously.

```
#include <LPC17xx.H>
void clock_wise(void);
void anti_clock_wise(void);
unsigned long int var1,var2;
unsigned int i=0,j=0,k=0;

int main(void)
{
    SystemInit();
    SystemCoreClockUpdate();

    LPC_PINCON->PINSEL0 = 0xFFFF00FF; //P0.4 to P0.7 GPIO
    LPC_GPIO0->FIODIR = 0x000000F0;    //P0.4 to P0.7 output

    while(1)
    {
        for(j=0;j<50;j++)      // 20 times in Clock wise Rotation
            clock_wise();

        for(k=0;k<65000;k++)   // Delay to show anti_clock Rotation
            anti_clock_wise();
    }
}
```

## Appendix A

```
for(j=0;j<50;j++)      // 20 times in Anti Clock wise Rotation
    anti_clock_wise();

    for(k=0;k<65000;k++);    // Delay to show clock Rotation

} // End of while(1)

} // End of main

void clock_wise(void)
{
    var1 = 0x00000008;      //For Clockwise
    for(i=0;i<=3;i++)      // for A B C D Stepping
    {
        var1 = var1<<1;    //For Clockwise
        var2 = ~var1;
        var2 = var2 & 0x000000F0;

        LPC_GPIO0->FIOPIN = ~var1;
        //LPC_GPIO0->FIOSET = var1;
        //LPC_GPIO0->FIOCLR = var2;

        for(k=0;k<3000;k++); //for step speed variation
    }

}

void anti_clock_wise(void)
{
    var1 = 0x00000100;    //For Anticlockwise
    for(i=0;i<=3;i++)    // for A B C D Stepping
```

```
{  
    var1 = var1>>1; //For Anticlockwise  
    var2 = ~var1;  
    var2 = var2 & 0x000000F0;  
  
    LPC_GPIO0->FIOPIN = ~var1;  
    //LPC_GPIO0->FIOSET = var1;  
    //LPC_GPIO0->FIOSET = var2;  
    for(k=0;k<3000;k++); //for step speed variation  
}  
}
```

**Components required**

- ALS-SDA-ARMCTXM3-01 : 1 No.
- Power supply (+5V) : 1 No.
- Cross cable for programming and serial communication: 1 No
- Stepper motor 1 No
- One working USB port in the host computer system and PC for downloading the software.
- 10 core FRC cables of 8 inch length 2 No
- USB to B type cable 1 No

**Hardware setup:** Connect 10 pin FRC cable from CNA to CNA5. Connect the stepper motor to PM1.

**Working procedure:** Stepper motor will rotate clockwise and in anti-clock wise direction automatically after reset.

**Lab Exercise:**

Write a C program to rotate the stepper motor in the clockwise direction when SW2 is high and anticlockwise direction when SW2 is low.

## Appendix A: Instructions

### Instruction Set Summary

Mnemonic	Operation	Description
<u><a href="#">ADC</a></u>	Rd := Rn + <u>Op2</u> + C	Add with carry
<u><a href="#">ADD</a></u>	Rd := Rn + <u>Op2</u>	Add
<u><a href="#">ADDS</a></u>	Rd := Rn + Op2	Add and update flags
<u><a href="#">ADR</a></u>	Rd := Rn, label	Load register with address
<u><a href="#">AND</a></u>	Rd := Rn AND <u>Op2</u>	AND
<u><a href="#">ANDS</a></u>	Rd := Rn AND <u>Op2</u>	AND update flags
<u><a href="#">ASR</a></u>	Rd := Rn,#LSB,#width	Arithmetic shift right
<u><a href="#">B</a></u>	R15 := address	Branch
<u><a href="#">BIC</a></u>	Rd := Rn AND NOT <u>Op2</u>	Bit Clear
<u><a href="#">BL</a></u>	R14 := address of next instruction, R15 := address	Branch with Link
<u><a href="#">BX</a></u>	R15 := Rn, change to Thumb if address bit 0 is 1	Branch and Exchange
<u><a href="#">CLZ</a></u>	Rd := number of leading zeroes in Rm	Count Leading Zeroes
<u><a href="#">CMN</a></u>	CPSR flags := Rn + <u>Op2</u>	Compare Negative
<u><a href="#">CMP</a></u>	CPSR flags := Rn - <u>Op2</u>	Compare
<u><a href="#">EOR</a></u>	Rd := Rn EOR Op2	Exclusive OR
<u><a href="#">LDM</a></u>	Stack manipulation (Pop)	Load multiple Registers (refer last paragraph of this appendix)
<u><a href="#">LDMIA</a></u>	LDMIA Rn!, {reglist}	Load multiple registers from memory
<u><a href="#">LDR</a></u>	Rd := [address][31:0]	Load 32-bit word from memory.
<u><a href="#">LDRB</a></u>	Rd := ZeroExtend ([address][7:0])	Load register byte value to Memory.

<u>LDRH</u>	Rd := ZeroExtend ([address][15:0])	Load register 16-bit halfword value to Memory.
<u>MCR</u>	cRn:=rRn {<op>cRm}	Move CPU register to coprocessor register
<u>MLA</u>	Rd := (Rm * Rs) + Rn	Multiply Accumulate
<u>MOV</u>	Rd := <u>Op2</u>	Move register or constant
<u>MRS</u>	Rn := PSR	Move PSR status flags to register
<u>MSR</u>	PSR := Rm	Move register to PSR status flags
<u>MUL</u>	Rd := Rm * Rs	Multiply
<u>MVN</u>	Rd := NOT Rm	Move inverted register or constant
<u>NOP</u>	None	No operation
<u>ORR</u>	Rd:=Rn OR <u>Op2</u>	OR
<u>PUSH</u>	PUSH {reg list}	Push registers on to the stack pointed by R13
<u>POP</u>	POP{reg. list}	Pop registers from the stack pointed by R13
<u>RSB</u>	Rd := Op2 – Rn	Reverse Subtract
<u>RSC</u>	Rd := Op2 - Rn - 1+Carry	Reverse Subtract with Carry
<u>RBIT</u>	RBIT Rd, Rn	Reverse the bit order in a 32-bit word
<u>REV</u>	REV Rd, Rn	converts 32-bit big-endian data into little-endian data or 32-bit little-endian data into big-endian data
<u>ROR</u>	Rd: = Rd, Rs	Rotate Rd register by Rs bits
<u>RRX</u>	Rd: = Rd, Rm	Rotate Right with Extend
<u>SBC</u>	Rd := Rn - Op2 - 1+Carry	Subtract with Carry
<u>STM</u>	stack manipulation (Push)	Store Multiple (refer last paragraph of this appendix)
<u>STR</u>	<address>:=Rd	Store register to memory
<u>STRB</u>	[address][7:0] := Rd[7:0]	Store register byte value to Memory.
<u>STRH</u>	[address][15:0] :=Rd[15:0]	Store register 16-bit halfword value to Memory

<u>SUB</u>	Rd := Rn - Op2	Subtract
<u>TEQ</u>	CPSR flags:= Rn EOR Op2	Test bitwise equality
<u>TST</u>	CPSR flags:= Rn AND Op2	Test bits
<u>UMULL</u>	UMULL r0, r4, r5, r6	Unsigned Long Multiply

A conditional instruction is only executed on match of the condition flags in the Program Status Register. For example, the BEQ (B instruction with EQ condition) branches only if the Z flag is set. If the {cond} field is empty the instruction is always executed.

{cond} Suffix	Tested Status Flags	Description
<b>EQ</b>	Z set	equal
<b>NE</b>	Z clear	not equal
<b>CS/HS</b>	C set	unsigned higher or same
<b>CC/LO</b>	C clear	unsigned lower
<b>MI</b>	N set	negative
<b>PL</b>	N clear	positive or zero
<b>VS</b>	V set	overflow
<b>VC</b>	V clear	no overflow
<b>HI</b>	C set and Z clear	unsigned higher
<b>LS</b>	C clear or Z set	unsigned lower or same
<b>GE</b>	N equals V	signed greater or equal
<b>LT</b>	N not equal to V	signed less than
<b>GT</b>	Z clear AND (N equals V)	signed greater than
<b>LE</b>	Z set OR (N not equal to V)	signed less than or equal
<b>AL</b>	(ignored)	always (usually omitted)

## **Addressing Mode for LDM and STM**

The instructions LDM and STM provide four different addressing modes. The addressing mode specifies the behavior of the base register and is explained in the following table.

Addressing Mode	Description
IA	Increment base register after instruction execution.
IB	Increment base register before instruction execution.
DA	Decrement base register after instruction execution.
DB	Decrement base register before instruction execution.

**Examples:** STMDB R2!,{R4,R5,LR}

LDMIA R0!,{R1-R5}

STMDB R6!,{R0,R1,R5}

## Appendix B: Addressing modes

Name	Alternative Name	ARM Examples
Register to register	Register direct	MOV R0, R1
Absolute	Direct	LDR R0, MEM
Literal	Immediate	MOV R0, #15 ADD R1, R2, #12
Indexed, base	Register indirect	LDR R0, [R1]
Pre-indexed, base with displacement	Register indirect with offset	LDR R0, [R1, #4]
Pre-indexed, autoindexing	Register indirect pre-incrementing	LDR R0, [R1, #4]!
Post-indexing, autoindexed	Register indirect post-increment	LDR R0, [R1], #4
Double Reg indirect	Register indirect Register indexed	LDR R0, [R1, R2]
Double Reg indirect with scaling	Register indirect indexed with scaling	LDR R0, [R1, r2, LSL #2]
Program counter relative		LDR R0, [PC, #offset]

### Literal Addressing

In this addressing mode data is a part of instruction. '#' symbol is used to indicate the data. ARM and Thumb instructions can only be 32 bits wide. You can use a **MOV** or **MVN** instruction to load a register with an immediate value from a range that depends on the instruction set. Certain 32-bit values cannot be represented as an immediate operand to a single 32-bit instruction, although you can load these values from memory in a single

instruction, you can load any 32-bit immediate value into a register with two instructions, a **MOV** followed by a **MOVT**. Or, you can use a pseudo-instruction, **MOV32**, to construct the instruction sequence for you. You can also use the **LDR** pseudo-instruction to load immediate values into a register

Examples	Meaning
CMP R0, #22	;Compare Register content R0 with 22
ADD R1, R2, #18 ;Add the content of R2 and 18 then store ;the result in R1	
MOV R1, #30	;copy the data 30 into register R1
MOV R1, #0Xff	;copy the data ff in hexadecimal into R1
MOV R2, #0xFF0000FF	
AND R0, R1, #0xFF000000	
CMN R0, #6400	; update the N, Z, C and V flags
CMPGT SP, R7, LSL #2	; update the N, Z, C and V flags

- **MOV** can load any 8-bit immediate value, giving a range of 0x0-0xFF (0-255). It can also rotate these values by any even number. These values are also available as immediate operands in many data processing operations, without being loaded in a separate instruction.
- **MVN** can load the bitwise complements of these values. The numerical values are - (n+1), where n is the value available in **MOV**.
- A **MOVT** instruction that can load any value in the range 0x0000 to 0xFFFF into the most significant half of a register, without altering the contents of the least significant half.
- The **LDR Rd,=const** pseudo-instruction generates the most efficient single instruction to load any 32-bit number

**Introduction to Register Indirect Addressing :** Register indirect addressing means that the location of an operand is held in a register. It is also called indexed addressing or base addressing.

Register indirect addressing mode requires three read operations to access an operand. It is very important because the content of the register containing the pointer to the operand can be modified at runtime. Therefore, the address is a variable that allows the access to the data structure like arrays.

- Read the instruction to find the pointer register
- Read the pointer register to find the operand address
- Read memory at the operand address to find the operand

Some examples of using register indirect addressing mode:

---

LDR R2, [R0] ; Load R2 with the word pointed by R0

---

STR R2, [R3] ; Store the word in R2 in the location  
; pointed by R3

---

LDR Rd,=label can load any 32-bit numeric value into a register. It also accepts PC-relative expressions such as labels, and labels with offsets

### Register Indirect Addressing with an Offset

ARM supports a memory-addressing mode where the effective address of an operand is computed by adding the content of a register and a literal offset coded into load/store instruction. For example,

Instruction	Effective Address
LDR R0, [R1, #20]	R1 + 20 ; loads R0 with the word ; pointed at by R1+20

---

## ARM's Autoindexing Pre-indexed Addressing Mode

This is used to facilitate the reading of sequential data in structures such as arrays, tables, and vectors. A pointer register is used to hold the base address. An offset can be added to achieve the effective address. For example,

Instruction	Effective Address
LDR R0, [R1, #4]!	R1 + 4 ; loads R0 with the word ;pointed at by R1+4 then ;update the pointer ;by adding 4 to R1

## ARM's Autoindexing Post-indexing Addressing Mode

This is similar to the above, but it first accesses the operand at the location pointed by the base register, then increments the base register. For example,

Instruction	Effective Address
LDR R0, [R1], #4	R1 ;loads R0 with the word ;pointed at by R1 then ;update the pointed by ;adding 4 to R1

## Program Counter Relative (PC Relative) Addressing Mode

Register R15 is the program counter. If you use R15 as a pointer register to access operand, the resulting addressing mode is called PC relative addressing. The operand is specified with respect to the current code location. Please look at this example,

Instruction	Effective Address
LDR R0, [R15, #24]	R15 + 24 ;loads R0 with the word ;pointed at by R1+24

## APPENDIX C

**GPIO extension connectors:**

There are four 10 pin FRC type male connectors, they extends the controllers general purpose port lines for the use of user requirements. Details on each connector is given below:

**CNA** – 10 pin male box type FRC connector. Port lines P0.4 to P0.11 from controller are terminated in this connector. They can be extended to interface few on board or external peripherals. The pins mentioned in the above table are configured to work as a GPIO's at power on. Other alternate functions on those pins needs to be selected using respective PINSEL registers.

Pin CNA	PIN LPC1768	Description
1	81	P0.4/I2SRX_CLK/RD2/CAP2.0
2	80	P0.5/I2SRX_WS/TD2/CAP2.1
3	79	P0.6/I2SRX_SDA/SSEL1/MAT2.0
4	78	P0.7/I2STX_CLK/SCK1//MAT2.1
5	77	P0.8/I2STX_WS/MISO1/MAT2.2
6	76	P0.9/I2STX_SDA/MOSI1/MAT2.3
7	48	P0.10/TXD2/SDA2/MAT3.0
8	49	P0.11/RXD2/SCL2/MAT3.1
9	-	No connection
10	-	Ground

**CNB** – 10 pin male box type FRC connector. Port lines from P1.23 to P1.26 and P2.10 to P2.13 are terminated in this connector.

Description of the connector **CNB**:

<b>Pin CNB</b>	<b>Pin LPC1768</b>	<b>Description</b>
1	37	P1.23/MCI1/PWM1.4/MISO0
2	38	P1.24/MCI2/PWM1.5/MOSI0
3	39	P1.25/MCOA1/MAT1.1
4	40	P1.26/MCOB1/PWM1.6/CAP0.0
5	53	P2.10/EINT0/NMI
6	52	P2.11/EINT1/I2STX_CLK
7	51	P2.12/EINT2/I2STX_WS
8	50	P2.13/EINT3/I2STX_SDA
9	-	No connection
10	-	Ground

CNC – 10 pin male box type FRC connector. Port lines from P0.15 to P0.22 and P2.13 are terminated in this connector.

<b>Pin CNC</b>	<b>Pin LPC1768</b>	<b>Description</b>
1	62	P0.15/TXD1/SCK0/SCK
2	63	P0.16/RXD1/SSEL0/SSEL
3	61	P0.17/CTS1/MISO0/MISO
4	60	P0.18/DCD1/MOSI0/MOSI
5	59	P0.19/DSR1/SDA1
6	58	P0.20/DTR1/SCL1
7	57	P0.21/RI1/RD1
8	56	P0.22/RTS1/TD1
9	50	P2.13/I2STX_SDA
10	-	Ground

CND – 10 pin male box type FRC connector. Port lines from P0.23 to P0.28 and P2.0 to P2.1 are terminated in this connector.

<b>Pin CND</b>	<b>Pin LPC1768</b>	<b>Description</b>
1	9	P0.23/AD0.0/I2SRX_CLK/CAP3.0
2	8	P0.24/AD0.1/I2SRX_WS/CAP3.1
3	7	P0.25/AD0.2/I2SRX_SDA/TXD3
4	6	P0.26/AD0.3/AOUT/RXD3
5	25	P0.27/SDA0/USB/SDA
6	24	P0.28/SCL0/USB_SCL
7	75	P2.0/PWM1.1/TXD1
8	74	P2.1/PWM1.2/RXD1
9	-	No connection
10	-	Ground