IV Semester BTech (E&C)

EC210 MICROPROCESSOR LABORATORY

Laboratory Manual

Prepared By

Dr. Aparna P.

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA, SURATHKAL

SRINIVASNAGAR 575025 KARNATAKA INDIA

JANUARY 2020

ARM Microprocessor (ARM7TDMI)

Objective:

The objective of this course is to study the assembly and high-level programming of an ARM microprocessor. Initial work will be verifying and executing
the programs on a simulator. In the latter part of the lab, students are introduced to ARM evaluation boards where in the programs are downloaded and verified on board. ARM-based microcontrollers are introduced which enables students to learn interfacing few basic peripheral devices to the processor, thus introducing them to SOC concept.

Choice of hardware and software

ARM7TDMI processor.
Keil, µVision Integrated Development Environment
MCB 2140 ARM7 Evaluation Boards-NXP- LPC 2148. Microcontroller

References

1. William Hohl, "ARM assembly Language- Fundamentals and Techniques", CRC PRESS.
2. J.R.Gibson, "ARM Assembly Language-An Introduction", Lulu publications.
3. Steve Furber, "ARM System Architecture", Edison Wesley Longman, 1996.
4. www.arm.com, "ARM Architecture Reference Manual".

General Instruction

- Maintain an observation book and write the programs to be tested before coming to the lab.
- Get the observation book verified before leaving the lab.

Programming Guide Lines:

Writing assembly code is generally not difficult once you've become familiar with the processor's abilities, the instruction available, and the problem you are trying to solve. When writing code for the first time, however, you should keep a few things in mind:

- **Break your problem down into small pieces.** Writing smaller blocks of code can often prove to be much easier than trying to tackle a large problem all at one go.

- **Always run a test case through your finished code, even if the code looks like it will "obviously" work.**

- **Use the software tools to their fullest when writing a block of code**. For example, the Keil tools provide a nice interface for setting breakpoints on instructions and watch points on data so that you can track the changes in registers, memory, and the condition code flags. As you step through your code, watch the changes carefully to ensure your code is doing exactly what you expect.

- **Use proper, meaningful names or labels.**

- **Simplicity is usually the best bet for beginning programs**.

- **Pay attention to initialization**. When your program or modules begin, make a note of what values you expect to find in various registers. Do you need to reset certain parameters at the start of a loop? Check for constants and fixed values that can be stored in memory or in the program itself.

- Your first programs will probably not be optimal and efficient. This is normal. As you gain experience coding, you will learn about optimization techniques and pipeline effects later, so focus on getting the code running without errors first. Optimal code will come with practice.

- **Using flowcharts may be useful in describing algorithms.**

- **Documentation is important.** Be sure to annotate your assembly clearly and with as much detail as required to allow someone else to follow your code.

LAB 1

Introduction to ARM assembly level programming and Tools.

Objective: The aim of this lab is to introduce to ARM assembly levels programming and use of KEIL μvision tools.

Introduction:

This lab introduces the basic program structure and few simple instructions to show how directive and code create an assembly program.

The Tools:

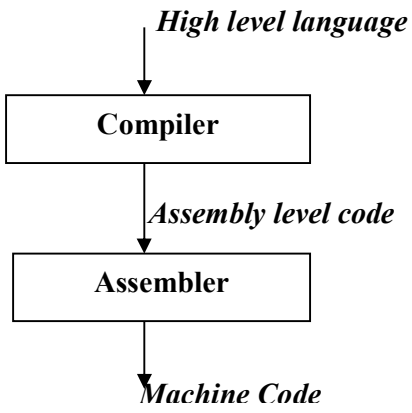


Fig 1.1

Compiler:

All the high level languages like C, C++ and Java have near English descriptions that are translated into native instruction set of the microprocessor. The program that does their translation is called a compiler.

ARM's C and C++ compiles generate optimized code for the 32-bit ARM instruction set and support full ISO standard C and C++. Modern tool sets, like ARM'S Real View Microcontroller Development Kit (RVMDK), which is found at http:// www.keil.com/demo, can normally display both the high-level code and its assembly language equivalent together.

Assembler:

As with most programming, we also need an automated way of translating our assembly language instructions into bit patterns, and this is precisely what an *assembler* does, producing a file that a piece of hardware (or a software simulator) can understand, in machine code using only 1's and 0's. To help out even further, we can give the assembler some pseudo-instructions, or directives (either in the code or with options in the tools), that tell it how to do its job, provided that we follow the particular assembler's rules such as spacing, syntax, the use of certain markers like commas, etc.

Software Development Cycle:

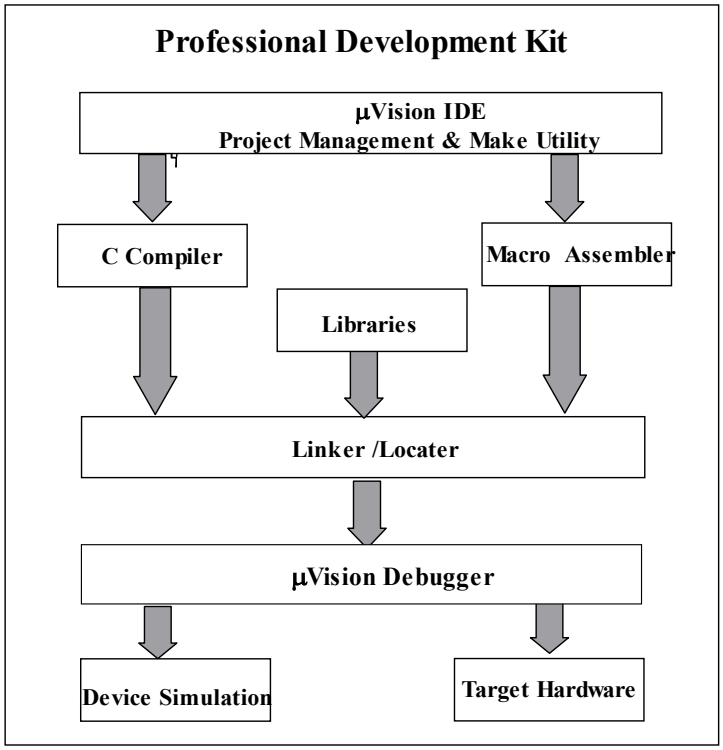


Fig 1.2

Tool Flow:

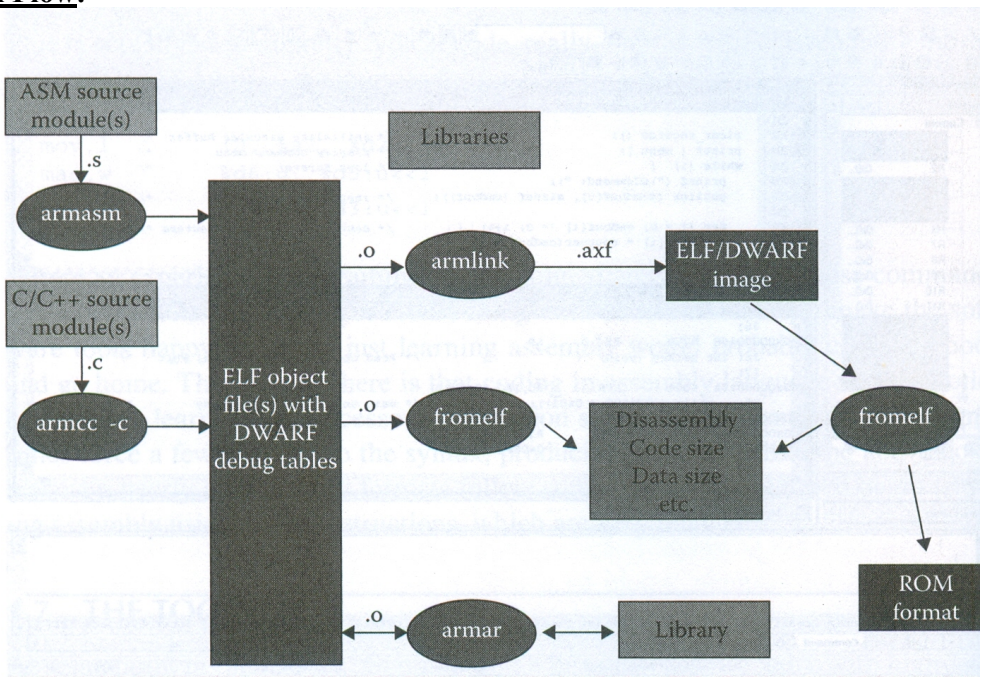


Fig 1.3

An object file is produced by the assembler from our source file, or the file that contains our assembly language program. Note that a compiler will also use a source file, but the code might be C or C++. Object files differ from executable files in that they often contain debugging information, such as program symbols (names of variables and functions) for linking or debugging, and are usually used to build a larger executable. Objective files, which can be produced in different formats, also contain relocation information. Once you've assembled your source files, a *linker* can then be used to combine them into an executable program, even including other object files, say from customized libraries. Under test conditions, you might choose to run these files in a debugger, but usually these executable are run by hardware in the final embedded application. The *debugger* provides access to registers on the chip, views of memory, and the ability to set and clear breakpoints and watch points, which are methods of stopping the processor on memory or instruction accesses. It also provides views of code in both high-level languages and assembly.

In all, RVMDK provides the following:

- C and C++ compilers
- Macro assembler
- Linker
- True integrated source-level debugger with a high-speed CPU and peripheral simulator for popular ARM-based microcontrollers
- µVision3 Integrated Development Environment (IDE), which includes a full-featured source code editor, a project manager for creating and maintaining projects, and an integrated make facility for assembling, compiling, and linking embedded applications
- File conversion utility (to convert an executable file to a HEX file, for example)
- Links to development tools manuals, device datasheets, and user's guides

Tools Used:

Assemblers are usually part of an integrated development system, a software package that runs on a PC. Most development system provides an integrated development environment IDE. Common development systems for ARM are ADS and real view from ARM Holdings or the Keil µVision system which allows the user to use either Real view or the GNU assembler.

IDE from RVMDV, Keil µVision system is used for their lab which provides a graphical user interface.

RVMDV not only simulates an ARM microprocessor but also can simulate a complete microcontroller. A wide variety of microcontrollers are available in the tools.

Note: Evaluation versions of the ARM tools are available on line from *www.keil.com*

Creating a Project:

Step1: Start the RVMDK tools by clicking the icon shown. 

Step2: choose **New µVision Project** from the **Project** menu. Give your project a name. The project file can include source files of code including C, C++ and assembly, library files, header files, etc. along with environment options that you can save

Step 3 :At this point the tools will ask you to specify a device to simulate as shown in Fig 1.4. Select ARM7(Little Endian) say OK.

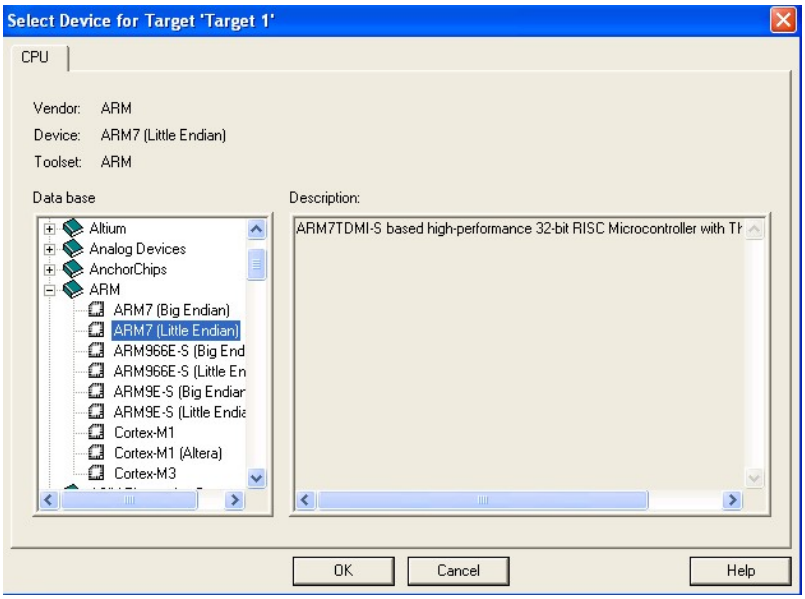


Fig 1.4

Step 4 : Creating Application Code:
Now that the project has been created and a device chosen, you will need to create a source file. From the **File** menu, choose **New** to create your assembly file with the editor.

Step5: Type the program 1.1 given below

Step 6: Choose **save as** from the **File** menu, and give it a name, such as **First_Program** with '.s' extension on the file.

Step 7: The assembly file must be added the project. In the Project Workspace window on the left, click on the plus sign to expand the Target 1 folder. Right click on the Source Group 1 folder, then choose **Add Files to Group 'Source Group 1'.** In the dropdown menu called **Files of Type,** choose **Asm Source** file to show all of the assembly files in your folder. Select the file you just created and saved, Click **Add,** and then **Close.**

Step8: Building the Project and Running Code:
To build the project, select **Build target** or **Rebuild all target files** from the **Project** menu. You will get a warning about the fact that Reset_Handler does not exist, but you can ignore it. From the **Debug** menu, choose **Start/Stop Debug Session.** This puts you into a debug session and produces new windows, including the Register window and the Memory window. You can single-step through the code, watching each instruction execute by clicking on the **Step** from the **Debug** menu. At this point, you can also view and change the contents of the register file, and view and change memory location by typing in the address of interest. When you are finished, choose **Start/Stop Debug Session** again from the **Debug** menu.

Program 1.1

```
                AREA        MyFirst, CODE, READONLY
                EXPORT      Reset_Handler

Reset_Handler

Start           MOV   R1, #10              ; Load a decimal Value
                MOV   R2, #0x11            ; Load an initial Hex Value
                MOV   R3, R1, LSL#1        ; Shift left by 1bit
                ADD   R4, R1, R2

Stop            B     Stop                 ; Stop program
                END
```

AREA, EXPORT, CODE, READONLY and END are the assembler directives that are essential part of an assembly program.

- Directives do not generate any instruction code but they determine how assembler generates code.

- AREA indicates a section of program with additional optional items CODE and READONLY to indicate that the section contains instructions and that the code produced may be put into read only memory.

- The line with the AREA directive provides information that the assembler passes to the linker.

- **MyFirst** is the name for the program (user defined). (Note: It has to be ***reset*** for new KEIL versions.)

- For Keil System the entry point is set by two lines,
  EXPORT *Reset_Handler*
  Rest_Handler
  These two lines set the program starting point. This must be present in one module of multiple module programs.

- END directive indicates that the program is complete, anything after this is ignored.

- *Start* and *stop* are the labels that are user specified. Labels must be positioned such that first letter of the label is in the first column of the line.

ARM assembly language program have a general format given by:
{Label} {Instruction or directive or pseudo instruction} {Comment}

For most ARM instruction, the general format of an instruction is :
Instruction destination, source, source

Debugging:

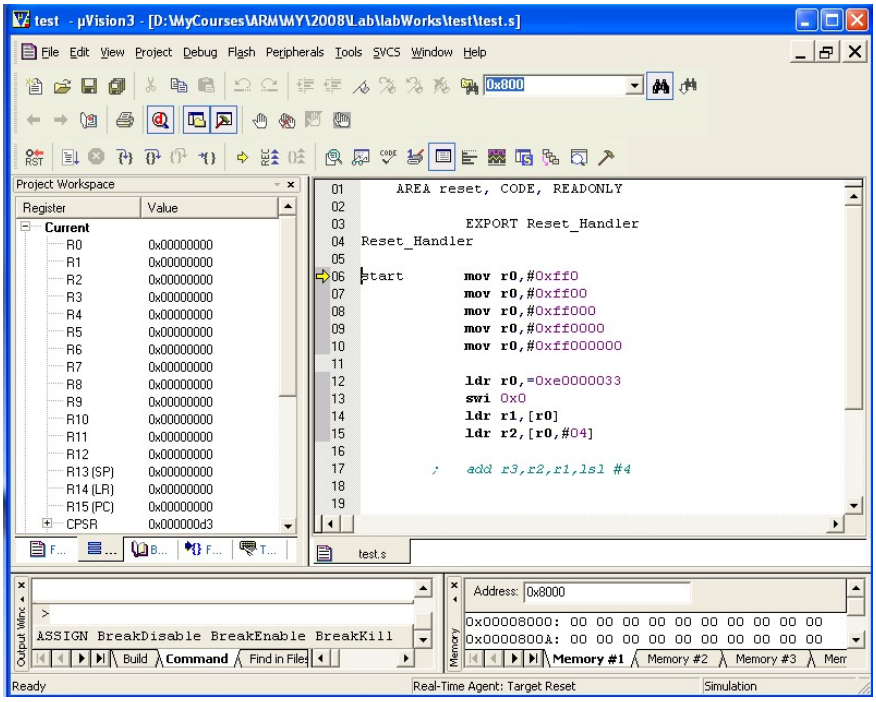


Fig 1.5

• Once you've started the debugger you can single step through the code executing one instruction at a time until the last instruction.

• Choose Disassembly window from the view menu and the code will appear as it appears in memory.

• Upper left hand corner shows the register window, where the entire register bank can be viewed and altered.

• Lower right hand corner shows a memory window that will display the contents of memory locations given a starting address.

• Break points can be set by double clicking the instructions in the grey bar area.

EXERCISE:

1.2. Observe the use of MOV and MVN instructions in loading a 32 bit immediate data into the registers.

1.3. Demonstrate the use of LSL, LSR, ASR, ROR, RRX with MOV, MVN, MOVS and MVNS for different data. Observe the results and conditional code flags in CPSR.

LAB 2

Objective: To demonstrate the use of arithmetic and logical instructions.

Exercises:
Try out the Practice sheets 1 and 2.

2.1 Write an assembly program to take two 32 bit unsigned numbers in to the registers using MOV or MVN instructions and perform the following
(a) Add using ADD, ADDS, ADC
(b) Subtract using SUB, SUBS, SBC
(c) Reverse subtract using RSB, RSC

2.2 (a) Repeat ex 2.1 a,b,c for 32 bit signed numbers.
(b) Repeat ex 2.1 a,b,c for 64 bit signed numbers.
(c) Repeat ex 2.1 a,b,c for 64 bit unsigned numbers.

(A) Note: use conditional execution of instructions wherever required.

2.3 Take two appropriate, 32 bit unsigned numbers and try all the logical instructions on them..

2.4 For the nos given in ex 2.4
perform No1 –No2 ;if No1>No2
perform No2 –No1 ;if No2>No1
using conditional execution of instruction.

2.5 Write a program to arithmetic shift a 64 bit value to right or left.

2.6 Write a program that counts number of ones in a 32 bit value. Store the result in register R3.

LAB – 3

Objective: To demonstrate the use of load and Store instructions and their addressing.

Introduction:

Data and the programs are stored in the memory. With all RISC processors, dedicated instructions are required for loading data from memory and storing data to memory. Memory can be viewed as contiguous storage elements that hold data, each element holding a fixed numbers of bits and having an address. Memory can be of different type and can reside at different addresses.

Address bus of ARM7TDMI consists of 32 bits and hence memory with address 0 to 2^{32} – 1 (i.e. 4GB) can be accessed. But however all the address may not contain the memory elements.

Load and Store instruction

Data is brought in from the external memory using Load instruction and placed in the memory using store instruction.

Format

LDR/STR {<Condition>} {B/H} <R_d>, <addressing mode>

LDR instructions take a single value from the memory and write it to a general purpose register. STR reads a value from the general purpose register and writes it to the memory.

<Condition> → optional condition
B → Load / Store byte (8bits)
H → Load / Store half word (16 bits)
Default → Load / Store word (32 bits)

LDM → Load multiple words
STM → Store multiple words

Endianness

ARM supports both formats knows as little endian and big endian through software and / or hardware mechanisms.

Say a register has a 32 bit value R_1 = 0x0A0B0C0D, which needs to be stored in the memory 0x400 to 0x403

Using LE configuration 0x0A0B0C0D is stored as

0x400	0D
0x401	0C
0x402	0B
0x403	0A

Using BE configuration 0x0A0B0C0D is stored as

0x400	0A
0x401	0B
0x402	0C
0x403	0D

Ex 3.1: Simple program to add the two 8 bit numbers from the memory and store the result in the memory.

```
                AREA        reset, CODE, READONLY
                EXPORT      Reset_Handler

Reset_Handler

Start           MOV  R1     #0x40000000     ; Address of the memory where the
                                            ; number is present
                MOV  R0,    #0              ; R0 holds the result

Loop            LDRB R2, [R1], #1           ; Load the first number & advance ptr
                LDRB R3, [R1], #1           ; Load the second number & advance ptr

                ADD  R0,, R2, R3            ; Add the numbers
                STRH R0, [R1], #2           ; store the result in the next memory

Stop            B    Stop
                END
```

Procedure for Defining the data memory:

- In the above program the memory starting from address 0x40000000 is used as the data memory. This address is available in the memory map as a read write area by default.
- If the data memory address is not in the memory map it should be added .
- Defining the data memory can be done by adding the starting and ending address to the memory map as: ***startAddress,endAdress*** in the read write mode.

For ex: 0x80000000,0x800000FF should be added to memory map and should be defined as read write. By doing this we specify that 256 bytes from address 0x80000000 to 0x800000FF is used as data memory in the simulator.

- The memory map is available in the debugger in the **debug option** .

Try out the Practice sheets 3 and 4.

3.2 Write an assembly program to add two numbers: C = a+ b where
   (a) a and b are both unsigned 32 bit numbers
   (b) a and b are both signed 32 bit numbers
   (c) a and b are both unsigned 64 bit numbers
   (d) a and b are both signed 64 bit numbers

   Load a and b from memory and store the results including carry into new location.

3.3 Write an assembly program to perform C = a – b where
   (a) a and b are both unsigned 32 bit numbers
   (b) a and b are both signed 32 bit numbers
   (c) a and b are both unsigned 64 bit numbers
   (d) a and b are both signed 64 bit numbers

   Load a and b from memory and store C in the memory

3.4 If a and b are two 32 bit numbers stored in consecutive memory location, swap the register content using EOR instruction and store back the results.

3.5 WAAP to add a constant value 0xFF to a block of 10 words stored in the consecutive memory locations using LDM and STM instructions.

3.6 Exchange a block of 32 bit data stored in the processor registers (R0- R7) with that in the memory starting at address 0x40000000.

3.7 Take two Nos from the memory, No1 and No2 as given below

No1= 0xBDF85433 and No2 =0x8FFF5FF3
No1= 0x967ff500 and No2 =0x50000000
No1 =0x74FFF000 and No2 =0x6800AFFF
No1 =0x80000000 and No2 =0xFFFF0000

(B) For each case perform 32 bit signed comparison and set the register R3 to 0xFF is No1 is greater than No2. Else set R3 to 0x00 using conditional execution of instructions.

(C) For each case perform 32 bit unsigned comparison and set the register R3 to 0xFF is No1 is greater than No2. Else set R3 to 0x00 using conditional execution of instructions..

LAB– 4

Objective: To study defining memory area, constant in the assembly program

Introduction:

In assembly program it is necessary to define some memory areas as constant or variable which may be used later during programming. Constant data is included in a program by use of some additional directives:

DCB : Define constant Byte
DCW : Define constant half word
DCD : Define constant data word.

Ex:

```
a1      DCB   34
a2      DCW   1234
a3      DCD   0xABCDEF10
```

Here a_1, a_2, a_3 are labels/names are given to the address of the constant value in memory.

For Ex: If a table of coefficients is needed and each coefficient is represented in 8 bits, then you might declare an area of memory as

```
table   DCB   0xFE, 0xF9, 0x12,  0x34
        DCB   0x11, 0x22, 0x33,   0x44
```

* If we assume that table starts at address 0x4000 the memory would look like

0x4000	0xFE
0x4001	0xF9
0x4002	0x12
0x4003	0x34
0x4004	0x11
0x4005	0x22
0x4006	0x33
0x4007	0x44

Strings : Strings are accessed by the processor by their ASCII value. Strings can be declared by DCB directive with the string given within " ".

For ex :

My_String **DCB "Hello"**

Where *My_String* is the label given to the memory location where the string is stored. ASCII of Hello will be stored in the memory. Each ASCII value takes a byte.

0x4000	0x48
0x4001	0x65
0x4002	0x65
0x4003	0x6C
0x4004	0x6F

Ex 4.1: Program that sums word-length value in memory and stores the result.

```
myDataSize          EQU         1024
abc                 EQU         stop+8
aaa                 EQU         myDataStart
sum                 RN          3
carry               RN          4
num                 RN          1
count               RN          2

                    AREA myData, DATA,NOINIT, READWRITE

myDataStart         SPACE       myDataSize

                    AREA reset, CODE, READWRITE
                    EXPORT      Reset_Handler

Reset_Handler

Start               LDR  R0, =myDataStart
                    LDR  count, N

loop                LDR    num, [R0], #4
                    ADDS sum, sum, num
                    ADDCS carry, carry,#1
                    SUBS  count, count,#1
                    BNE    loop

                    STR    sum, [R0], #4
                    STR    carry,[r0],#4

                     LDR   r4,=aaa
                     LDR   r5,=abc

stop                B       stop

N                   DCB     0xF

                    ALIGN

myConstants  DCD   0xAAAABBBB, 0x12345678, 0x88881111
             DCD   0x00000013, 0x80808080, 0xFFFF0000

                    END
```

Note:

- DATA directive indicates that the section of memory holds variable data values.
- READWRITE shows that running program may alter the values.
- EQU i.e equals directive set the user name defined by the label to a quantity set by the programmer

Pseudo Instructions:

RISC processors have some limitation on the instruction capabilities. In an ARM instruction an immediate operand can only have a restricted range of vale. This is particularly inconvenient when using the MOV instruction to put a fixed value in a register.

To overcome such limitation ARM has introduced pseudo instructions.

LDR {<Condition>} <R_d>, = <immediate>

(A) Define a number in the program and perform the following. If true store 1.Else 0xFF
4.2 Check if the given number is odd or even.
4.3 Find the number of occurrence of a digit in a number.
4.4 Reverse the given number.
4.5 Check whether the given number is a Fibonacci number.
4.6 To generate the Fibonacci numbers up to the given number

(B) String Operations

4.7 A program to move a string of characters from one memory location to other.

```
str_dst         EQU     0x040000000
myDataSize      EQU     1024

                AREA myData, DATA, READWRITE
myDataStart     SPACE   myDataSize

                AREA   reset, CODE, READONLY
                EXPORT Reset_Handler

Reset_Handler
                LDR     R1, =str_src    ; Pointer to the first string
                LDR     R2, =str_dst    ; Pointer to the second string

str_cpy         LDRB    R3, [R1],#1     ; load a byte and update the pointer
                STRB    R3, [R2],#1     ; store byte and update the pointer
                CMP     R3, #0          ; Check for End of string
                BNE str_cpy

stop            BAL stop

str_src         DCB    "This my string start.",0
                END
```

4.8 Reverse the string and check if the string is a palindrome.
4.9 Find the substring in Main string.
4.10 Insert a substring in main string at given position.
4.11 Squeeze a string removing all the blank spaces and store it in the same location

LAB – 5

Objective: To use the branch instructions in assembly level programming efficiently

Introduction

Any event that modifies the program counter can be defined as a change of flow and this can be accomplished by either explicitly modifying the program counter by writing to it or using one of the branch instructions.

Three types of branch instructions on ARM 7 are:

B – Branch. Condition codes may be used to decide whether or not to branch to a new address in the code.

Bx – Branch and exchange. This instruction provides a mechanism to switch from 32-bit ARM instruction to 16-bit thumb instruction.

BL – Branch and Link. Used to execute subroutine and return

Conditional memories:

Memories	Name	Conditional Flags
EQ	Equal	Z
NE	Not Equal	z
CS, HS	Carry Set/ unsigned High or Same	C
CC, LO	Carry Clear / unsigned Lower	c
MI	Minus / negative	N
PL	Plus / positive or zero	n
VS	Overflow	V
VC	No overflow	v
HI	unsigned higher	zC
LS	unsigned lower or same	Z or c
GE	signed greater that or equal	NV or nv
LT	signed less than	Nv or nV
GT	signed greater than	NzV or nzv
LE	signed less than or equal	Z or Nv or nV
AL	always (unconditional)	ignored

* BAL → jump or goto

Ex 5.1: Write a program to have a leading 1 in a binary number in the most significant bit position.

```
        AREA  reset, CODE, READONLY
        EXPORT Reset_Handler

Reset_Handler

main    MOV  R4, # 0        ; clear shift count
        CMP  R3, # 0        ; compare whether original number is zero
                            ; (N≤0)
```

```
            BLE    finish                ; if yes, done
loop        MOVS   R3, R3, LSL # 1       ; shift one bit
            ADD    R4, R4, # 1           ; increment shift counter
            BPL    loop

finish      B finish
            END
```

Exercise: **(A) Arrays & Subroutines:**

5.2 WAAP to find the maximum value in a list of numbers stored in the memory.

5.3 Write an arm assembly program to exchange a block of data from one memory location to another. Optimize your code for minimum number of instructions.

5.4 Marks of two quizzes are stored in memory starting from memory location 'quiz1' and 'quiz2'. Each mark is stored as a byte. Write an assembly program to

(i) Find the maximum and minimum
(ii) To sort the marks in ascending order
(iii) To find the average of the class

5.5 Write an assembly subroutine to do the following:

(a) Compare two signed words present in r_0 & r_1
(b) Swap the contents of r_0 & r_1 if $r_1 > r_0$

5.6 Write a program to check a string. If the string is "Name", sort the list in the database in alphabetical order. If the string is "Age", sort the list according to the age. (Create a database with 10 entries in the RAM with fields Name and age. Limit the name to 3 characters (i.e 3 bytes) and the age to 1 byte. Thus each entry takes 4 bytes.). Call the sorting function as subroutines.

LAB – 6

Objective: To understand and use the multiplication instructions effectively.

Introduction:

Microprocessors and/or DSP engines are often selected based on their ability to perform fast multiplication, especially in the areas of speech and signal processing, signal analysis

Multiply and multiply accumulate instructions available
MUL : 32 x 32 multiply with 32-bit product
MLA : 32 x 32 multiply added to a 32-bit accumulated value
SMULL: signed 32 x 32 multiply with 64 bit product
UMULL : unsigned 32 x 32 multiply with 64 bit product
SMLAL : signed 32 x 32 multiply added to a 64 bit accumulated value
UMLAL : unsigned 32 x 32 multiply added to a 64 bit accumulated value

Exercise

6.1 Write an assembly program to perform multiplication $c = a * b$ where
(a) a and b are both unsigned 32 bit numbers.
(b) a and b are both signed 32 bit numbers.
(c) a and b are both unsigned 64 bit numbers.
(d) a and b are both signed 64 bit numbers.
Load a and b from the memory and store the results into new locations in memory.
Use RN assembler directives to name the registers.

6.2 Assume that a signed long multiplication instruction is not available. Write a program that performs long multiplications, producing 64 bits of result. Use only the UMULL instruction and logical operations such as MVN to invert EOR and ORR

6.3 Write a program to divide (a) 32 bit number by 16 bit number.
(b) 64 bit number by 32 bit number.
By Method I: Repeated subtraction
Method II: shift & subtract
Compare the time taken by two methods for same example.

Ex: $\frac{FFFFFFFF}{FFFD}$, $\frac{FFFFFFED}{FF}$, $\frac{FFFF}{FFF}$, $\frac{FFFF}{FFF}$

6.4 Write a program that takes a 16 bit Hex number and converts it into its BCD equivalent.

6.5 Write a program that takes an 4 digit BCD number and converts it into is Hex equivalent.

6.6 Perform addition of two 8 digit BCD numbers and give the result in BCD.

6.7 Write a program to convert a given (Note: Consider BCD and HX to 4 digit no)
(a) ASCII to BCD
(b) ASCII to HX
(c) HX to ASCII
(d) BCD to ASCII

Lab -7
Thumb Mode of Operation

Objective: To introduce the Thumb mode of operation of an ARM processor and to get familiar with the Thumb instructions.

Introduction:

Thumb technology is an extension to the 32-bit ARM architecture. The Thumb instruction set features a subset of the most commonly used 32-bit ARM instructions which have been compressed into 16-bit wide opcodes. On execution, these 16-bit instructions are decompressed transparently to full 32-bit ARM instructions in real time without performance loss.

Designers can use both 16-bit Thumb and 32-bit ARM instructions sets and therefore have the flexibility to emphasize performance or code size on a sub-routine level as their applications require.

A "Thumb-aware" core is a standard ARM processor fitted with a Thumb decompressor in the instruction pipeline. The designer therefore gets all the underlying power of the 32-bit ARM architecture as well as excellent code density from Thumb, all at 8-bit system cost.

Thumb has better code density than common 8 and 16-bit CISC/RISC Controllers and is at a fraction of the code size of traditional 32-bit architectures. This means that program memory can be smaller and hence cost reduced.

- Thumb Register usage:
    - In thumb state no direct access to all registers.
    - Only R0-R7 is fully accessible.
    - R8-R12 is used only with MOV, ADD, CMP.
    - All data processing instructions & CMP that operate on low registers update the condition flags.
    - No MSR and MRS instructions.
- ARM-Thumb Interworking within a source file:
    - Change from ARM to Thumb state can be implemented by changing the T bit.
    - **BX Rn** (i.e branch and exchange)instruction causes a switch between ARM & Thumb mode while branching to a routine. Processor enters thumb mode only if the bit 0 of register Rn is set to binary 1.
    - **BX LR** returns from a routine, with a state changed if required.
    - The following directives instruct the assembler to assemble instructions from appropriate instruction set.
        - **CODE16**
        - **CODE 32**

Thumb Instruction Set

Mnemonic	Operation	Description
ADC	Rd := Rd + Rm + C	Add with carry
ADD	Rd := Rn + { imm, Rm }	Add constant or register to register
AND	Rd := Rd & Rm	Logical AND
ASR	Rd := (signed) Rm >> { imm5, Rs }	Arithmetic shift right
B	R15 := label	Branch conditional or unconditional
BIC	Rd := Rd AND NOT Rm	Bit Clear
BL	R14 := address of next instruction R15 := label	Branch with link
BX	R15 := Rm AND 0xFFFFFFFE	Branch and exchange
CMN	Set CPSR flags on: Rn + Rm	Compare negated register
CMP	Set CPSR flags on: Rn - { imm8, Rm }	Compare constant or register
EOR	Rd := Rd EOR Rm	Exclusive OR
LDMIA	Load register list	Load multiple registers from memory
LDR	Rd := [address][31:0]	Load 32-bit word from memory
LDRB	Rd := ZeroExtend ([address][7:0])	Load byte from memory
LDRH	Rd := ZeroExtend ([address][15:0])	Load 16-bit halfword from memory
LDRSB	Rd := SignExtend ([address][7:0])	Load signed byte from memory
LDRSH	Rd := SignExtend ([address][15:0])	Load signed 16-bit halfword form memory
LSL	Rd := Rm << { imm5; Rs }	Logical Shift Left
LSR	Rd := (unsigned) Rm >> { imm5, Rs }	Logical Shift Right
MOV	Rd := { imm, Rm }	Move constant or register to register
MUL	Rd := Rm * Rs	Multiply register
MVN	Rd := NOT Rm	Move inverted register to register
NEG	Rd := -Rm	Negate
ORR	Rd := Rd OR Rm	Logical OR
POP	Pop register list	Load multiple registers from stack
PUSH	Push register list	Save multiple registers to stack
ROR	Rd := Rd ROR Rs[7:0]	Rotate right
SBC	Rd := Rd - Rm - NOT C	Subtract with Carry
STMIA	Store register list	Store multiple registers to memory
STR	[address][31:0] := Rd	Store 32-bit word to memory
STRB	[address][7:0] := Rd[7:0]	Store byte to memory
STRH	[address][15:0] :=Rd[15:0]	Store 16-bit halfword to memory
SWI	Software interrupt	Call software interrupt function
SUB	Rd := Rn - { imm; Rm }	Subtract constant or register
TST	Set CPSR flags on: Rn AND Rm	Test bits

Example 7.1

```
                AREA RESET,CODE,READONLY
                EXPORT Reset_Handler

Reset_Handler

main            LDR r0, =ThProg+1   ; Generate branch target address
                BX R0               ; Branch exchange to ThumbProg.

                CODE16              ; Subsequent instructions are Thumb code.

data            dcb     23,12

ThProg          MOV   R2, #0x40     ; Load r2 with value 0x40
                MOV   R3, #24       ; Load r3 with value 3.
                LSL   R2,R2,R3
                LDRH R4,[R1,R2]
                ADD   R2,R3
                MOV   R4,R2

                ADR   R0,ARMProg
                BX    R0

                ALIGN
                CODE32              ; Subsequent instructions are ARM code.

ARMProg         MOV   R5, #4
                MOV   R6, #5
                ADD   R7, R5, R6

stop            BAL stop
                END
```

ARM-Thumb Interworking using veneers

If an ARM subroutine has to call a Thumb subroutine or vice-versa and if these two sections were to be compiled separately, switching from ARM to Thumb mode needs some additional steps to be accomplished. Compiler provides a means to do this. This is called interworking and is enabled by means of a piece of code called **veneers** as given below.

```
ADD     R12,PC,#0x00000001
BX      R12
```

Example 7.2

;ArmProg.s

```
                AREA reset, CODE, READONLY
                EXPORT      Reset_Handler
                IMPORT  subAdd

Reset_Handler
Start           mov r0,#0
                mov r1,#3
                b next
                mov r2,#3
next            mov r2,#2

                bl subAdd

Stop            B       Stop                    ; Stop program
                END
```

;ThumbProg.s

```
                AREA functionAdd, CODE, READONLY
                EXPORT subAdd

                CODE16
subAdd
                mov r1,#0x1
                mov r2,#0x2
                add r3,r1,r2
                mov pc,lr

                END
```

Exercise:

Write Thumb-mode assembly programs for the following and compare their code density and time of execution with Arm mode assembly programs:
(Note: Write the programs similar to arm-mode programs so that it can be compared with thumb-mode programs)

1. Write a program to add two numbers: c = a + b where
   (a) a & b are both unsigned 32 bit numbers
   (b) a & b are both signed 32 bit numbers
   Load a & b from memory and store the results including carry (if any) into new location in memory.

2. Write a program to perform logical operations: c = a(logical operation) b where a & b are both unsigned 32 bit numbers

3. Write a program to perform multiplication: c = a * b where a & b are both unsigned 32 bit numbers. Load a & b from memory and store the results into new location in memory.

Lab 8

System Peripherals

Aim: To learn interfacing few basic peripheral to the microprocessor and write application programs in assembly level to use them.

Description:

A microprocessor can be seen as a computing engine with no peripherals. Very simple processor can be combined with useful extras such as timers, universal synchronous receivers/ transmitters (UARTS), A/D converters etc to produce a microcontroller, which tends to be a very low cost device for use in industrial controllers, displays, automotive applications, toys etc. This lab exercise look into how these devices can be interfaced with a single processor.

Small target boards are available for initial studies of ARM systems which use microcontrollers that incorporate peripheral devices. The simulator provided with Keil software tools emulates the input and output systems of many microcontrollers and is adequate for initial programming. However any program for commercial use must be fully tested using the complete production system.

Tools used:

MCB 2140 – ARM7 Evaluation Board – LPC2148 Microcontroller
Flash Magic Software
Keil MicroVision Realview

Description of LPC2148

LPC 2148 microcontroller is based on 16 bit / 32 bit ARM7TDMI-S CPU with real time emulation and embedded tract support.

Key Features

- 16-bit/32-bit ARM7TDMI-S microcontroller in a tiny LQFP64 package.
- 40 kB of on-chip static RAM and 512 kB of on-chip flash memory.
- 128-bit wide interface/accelerator enables high-speed 60 MHz operation.
- In-System Programming/In-Application Programming (ISP/IAP) via on-chip boot loader software, single flash sector or full chip erase in 400 ms and programming of 256 B in 1 ms
- Embedded ICE RT and Embedded Trace interfaces offer real-time debugging with the on-chip Real Monitor software and high-speed tracing of instruction execution.
- 10-bit ADCs provide a total of 6/14 analog inputs, with conversion times as low as 2.44 ms per channel.
- Single 10-bit DAC provides variable analog output (LPC2142/44/46/48 only).
- Two 32-bit timers/external event counters (with four capture and four compare channels each), PWM unit (six outputs) and watchdog.
- Low power Real-Time Clock (RTC) with independent power and 32 kHz clock input.
- Multiple serial interfaces including two UARTs (16C550), two Fast I2C-bus (400 kbit/s)
- Vectored Interrupt Controller (VIC) with configurable priorities and vector addresses

Block Diagram

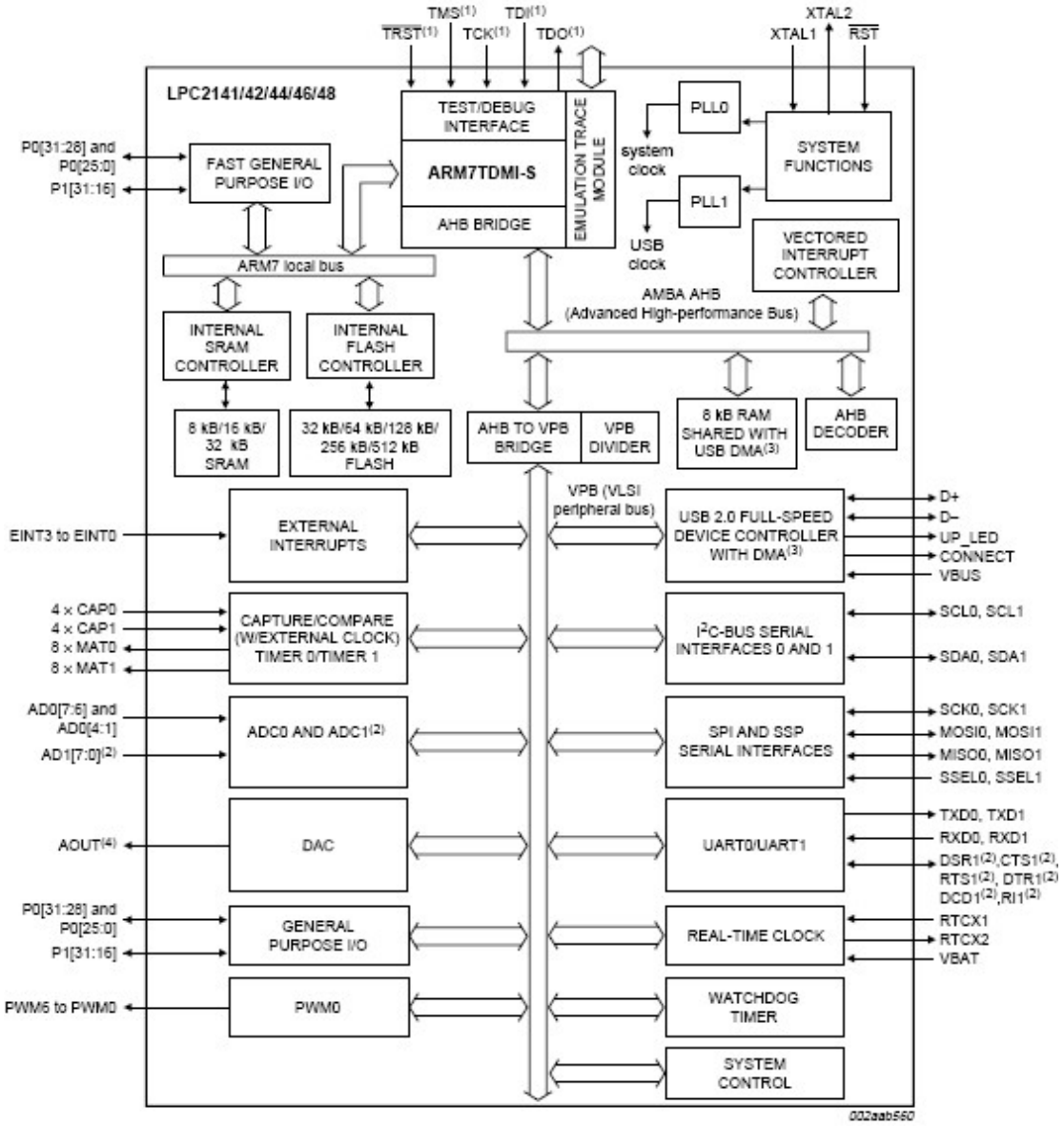


Fig (1): Block Diagram of LPC2148

Bus Structure:

- **AHPB**(Advanced High Performance Bus)

Fastest way of connecting peripherals to ARM7 core is through AHPB.

- **VPB**(VLSI peripheral Bus)

Slower user peripherals are connected to the bus.

- **VPB Bridge**

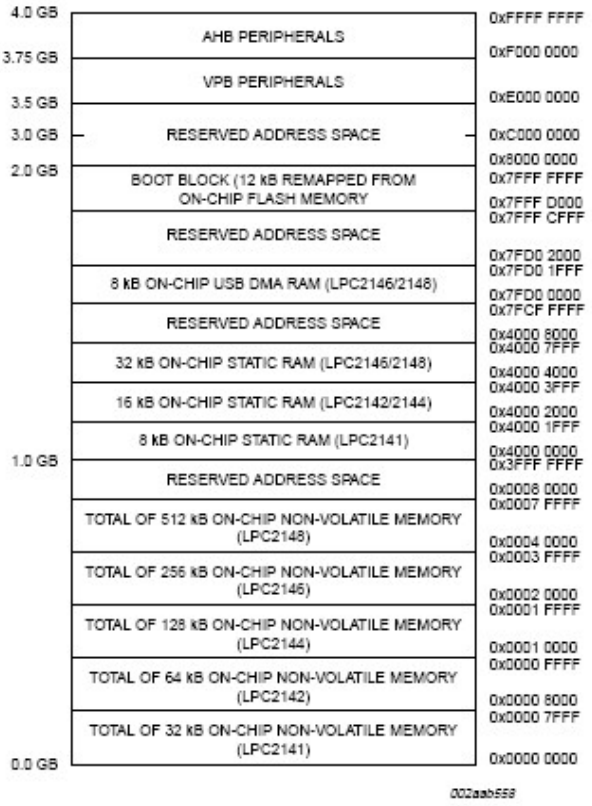
Slower VPB is connected to AHPB through this VPB Bridge

- **Local Bus**

On chip RAM and Flash are connected this to ARM core

Memory Map:

This LPC2148 memory map incorporates several distinct regions as shown in fig (2).



Fig(2) Memory Map for LPC 2100 series

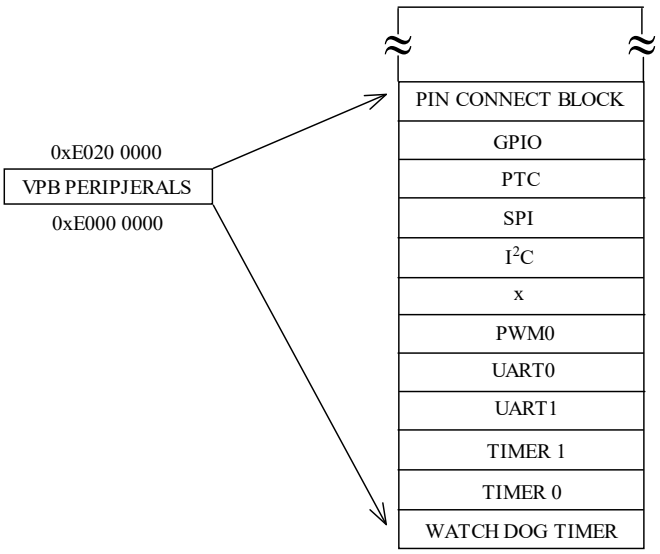
On-chip flash program memory

- The LPC2148 incorporate a 512 KB flash memory system respectively. This memory may be used for both code and data storage.
- Programming of the flash memory may be accomplished in several ways. It may be programmed In System via the serial port.
- Due to the architectural solution chosen for an on-chip boot loader, flash memory available for user's code on LPC2148 is 500 kB respectively.

On-chip static RAM

- On-chip static RAM may be used for code and/or data storage. The SRAM may be accessed as 8-bit, 16-bit, and 32-bit. The LPC2148 provide 32 kB of static RAM respectively.
- In case of LPC2148 only, an 8 kB SRAM block intended to be utilized mainly by the USB can also be used as a general purpose RAM for data storage and code storage and execution.

The system peripherals are all memory mapped IO's and hence shares the 4GB of Address space.



Fig(3): Mapping of VPB peripherals

Exercise 8.1:

Out line

This exercise illustrates how to configure the ports of LPC 2148 for general purpose I/O and use them for simple input and output operation.

Description:

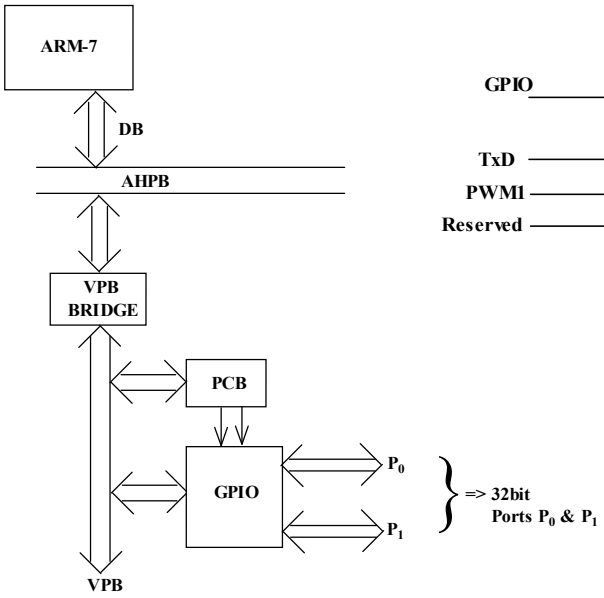


Fig (5) Bus structure

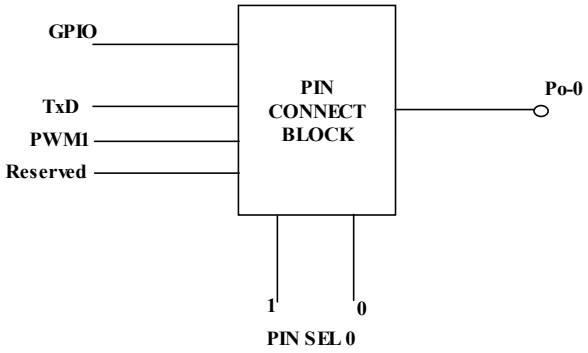


Fig (6) P0.0 configured By PCB

Pin connect Block:

- The pin connect Block allows selected pins of the microcontroller to have more than one function.
- Configuration register control the multiplexers to allow connection between the pin and the on chip peripheral.
- After reset all pins of port0 and port1 are configured as input except during debug or trace mode is enabled.

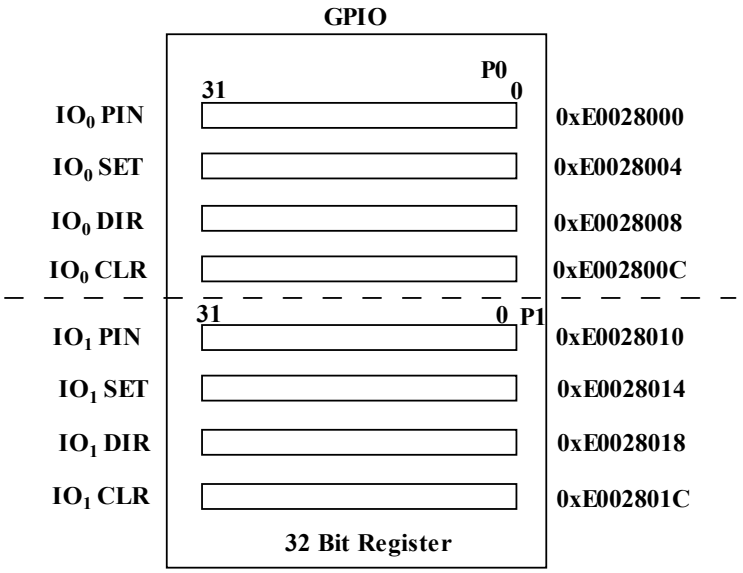
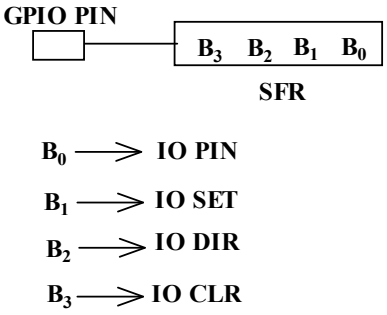


Fig (7) GPIO Registers and their addresses



Fig(8) Single GPO Pin

- On reset the pin connect block configure all the peripheral pins to be general purpose IO.
- GPIO pins are controlled by four registers as shown in fig (7) and fig (8). Each GPIO pin is controlled by a bit in each of the four GPIO register. IO PIN, IOSET, IOCLR, IODIR.
- **IODIR** bit allows each pin to be individually configured as an input (0) or an output (1).
- **IOSET** bit = 1 sets the corresponding pin to 1 when IODIR=1.
- **IOCLR** =1 clears the corresponding pit to 0 when IODIR = 0.
- **IOPIN** register is read to get the status of the pin when IODIR = 0

Note: Detailed explanation of all the peripherals is given in product Data Sheet of LPC2148. Refer to Keil\Arm\Help\MCB2140.chm for help.

Program:

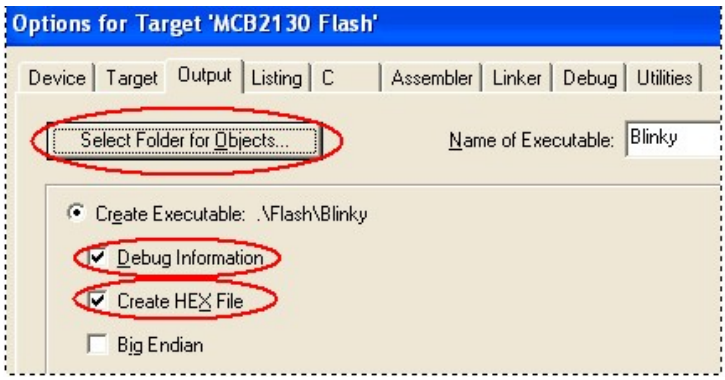
Write an assembly program to blink the LEDs connected to port GPIO P1.16-23 for Philips LPC2100 series verify the assembly program using simulator and download the program to Flash using Flash magic tool.

```
; General Purpose Input Output1(GPIO1)
IO1DIR      EQU   0xE0028018
IO1SET      EQU   0xE0028014
IO1CLR      EQU   0xE002801C

            AREA BlinkyTest, CODE, READONLY
            EXPORT blinky

blinky      LDR   r1, =IO1DIR          ; load the address of the IODIR reg to R1
            LDR   r0, =0x00FF0000      ; To set pins P1.16 to P1.23 as output pins

            STR   r0, [r1]

            LDR   r2, =IO1SET          ; load the address of the IOSET reg to R2
            LDR   r3, =IO1CLR          ; load the address of the IOCLR reg  to R3

repeat      MOV   r4, #0x00010000      ; write a control word to set one bit
next        STR   r4, [r2]

            LDR   r5, =0xF00           ;  Delay program to retain the bit for some time.
delay       SUBS  r5, r5, #1
            BNE   delay

            STR   r4, [r3]             ; write a control word to clear the same bit

            MOV   r4, r4, LSL #1
            CMP   r4, #0x800000
            BNE   next
            B     repeat

            END
```

Procedure:

1. **Create** a **New Project File** with the µVision menu command **Project — New Project**. **Select** a microcontroller LPC 2148 from NXP list from the Device Database.
2. **Add Startup Code**.
   The Startup file provides pre-configured startup code for the MCB2140 board. The Startup file you use varies depending on the toolset you select. **Copy** the appropriate Startup file to your project folder and **add** it to your project.
3. **Add** your own **Source Code (say blinky.s)** to the project
4. Configure the Output Files in the **Project — Options for Target — Output** dialog. The steps for configuring these files are:

- **Select Folder for Objects...** allows you to specify sub-folders for the output files. By default, objects are stored in the same folder as the project file.
- **Enable Debug Information** to store symbolic debug information within the executable file for source level program testing with the µVision debugger.
- **Enable Create HEX File** to generate an Intel HEX file for Flash programming with the NXP LPC2000 Flash Utility.



5. **Start the build**

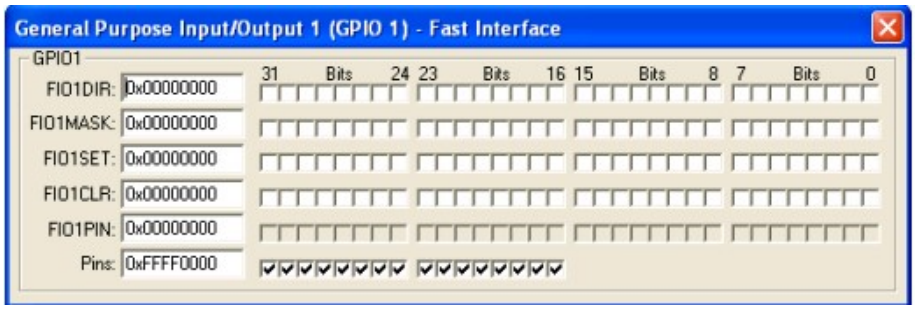
Click on the **Build** toolbar button to generate your application program. µVision translates all your source files and links the project.

6. **Program Files**

- The build creates an **Executable File** in ELF Format in the Output folder you designated earlier. The ULINK USB-JTAG adapter requires this file for downloading to the MCB2140 board using the JTAG or other emulators.
- The build creates an **Intel HEX** executable file. The LPC2000 FLASH ISP Utility requires this file for downloading through the serial port.

7. **Debugging in the simulator**

Debug the BLINKY example program in the simulator as shown below:



The GPIO 1 dialog reflects the status of the I/O ports.

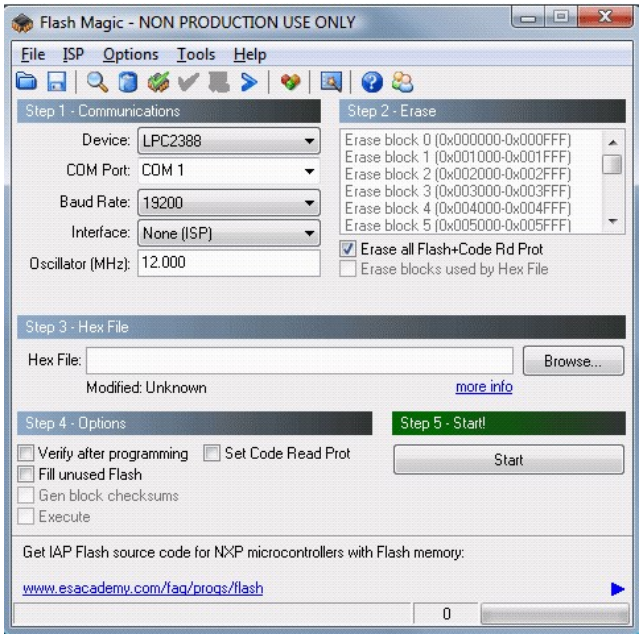
8. **Download the program to the on-chip Flash of the MCB 2140**

Either of the following tools can be used to download programs to the Flash memory of the MCB2140 evaluation board:

- The ULINK USB-JTAG Adapter connects the USB port of your PC to the JTAG port of the MCB2140. It supports programming Flash, emulation, and debug capabilities.
- The Flash Magic Utility connects the COM port of your PC to the serial port of the MCB2140. It may be used only for programming Flash via the ISP Flash Interface.

9. **Flash Programming**

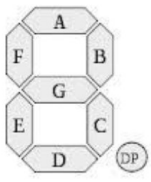
This exercise uses Flash Programming using Flash Magic Tool .

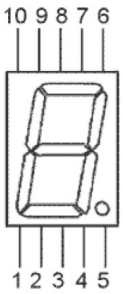


Ex 8.2 Using the same ports as in ex 8.1, write a program to display binary values 00-FF on the LEDs.

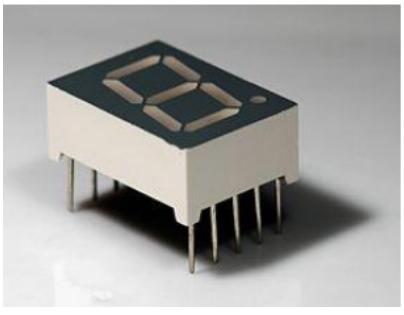
Seven Segment Display

Ex 8.3 Connect a seven segment LED, and display the numbers 0-F continuously.





PIN NO.	E MAN6960
1	Cathode E
2	Cathode D
3	Com. Anode
4	Cathode C
5	Cathode D.P.
6	Cathode B
7	Cathode A
8	Com. Anode
9	Cathode F
10	Cathode G



HEX Combinations of Bus Signals Necessary to Light up from 0 to 9

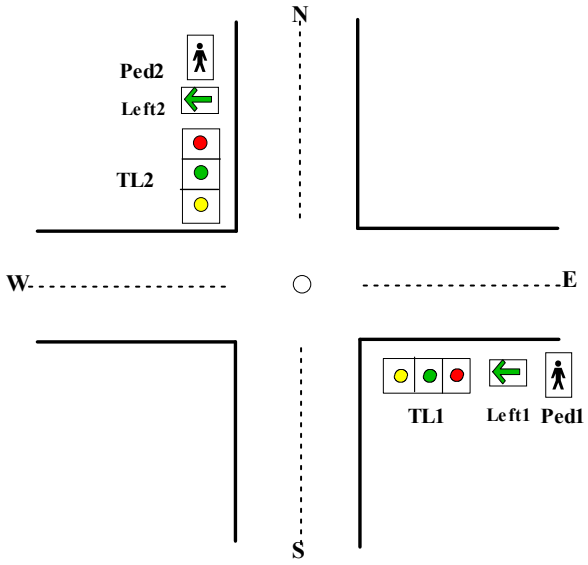
Digit	dp G F E D C B A	HEX Combination
0	1 1 0 0 0 0 0 0	C0
1	1 1 1 1 1 0 0 1	F9
2	1 0 1 0 0 1 0 0	A4
3	1 0 1 1 0 0 0 0	B0
4	1 0 0 1 1 0 0 1	99
5	1 0 0 1 0 0 1 0	92
6	1 0 0 0 0 0 1 0	82
7	1 1 1 1 1 0 0 0	F8
8	1 0 0 0 0 0 0 0	80
9	1 0 0 1 0 0 0 0	90

- **Use common-anode 7-segment display**
- **Pin Numbering (Diagram on next slide)**
  - Face display with decimal point at bottom right
  - Start at Pin 1 at bottom left, move right incrementing pin numbers until Pin 5
  - Then crossover to Pin 6 at top right, move incrementing pin numbers until Pin 10
- **Identify 7-segment Display pins**
  - 8 LED segments, including decimal point
  - Pins 3 or 8 are common-anode pins
  - Pins 1, 2, 4, 5, 6, 7, 9 and 10 are cathodes

GPIO Ports	7segment pin no	7 segment pin name
P0.0	7	A
P0.1	6	B
P0.2	4	C
P0.3	2	D
P0.4	1	E
P0.5	9	F
P0.6	10	G
P0.7	5	dp
P0.8	3 or 8	CA

Project

Design a traffic light controller for a cross road as shown below:



Description :

- Only straight and Left turn are allowed.
- TL1 : For N to S and S to N
- TL2 : For W to E and E to W
- Left1 : For left turn S to W and N to E
- Left2 : For left turn W to N and E to S
- Ped1 : Pedestrian at N and S
- Ped2 : Pedestrian at W and E
- Left and TL straight can be green at same time.

Write a program using GPIO. Check it on simulator. Execute this on the target board with all the H/W setup made. Show the flow chart or the *fsm* for the same.

Duration: 2 Hrs

Lab 9

System Peripherals-UART

Aim: To understand the concept of serial transmission and hence write application program to configure and use UART in assembly level.

Introduction:

- ***Serial Communication:***

Microprocessor is a parallel device, transferring 8/16/32 bits of data simultaneously over the data bus. However in many situations parallel mode of transmission is impracticable or impossible. For Ex: Parallel data communication over long distance becomes expensive., or CRT terminal etc. In such situations serial IO mode is preferred where by one bit is sent at a time over a single line. Serial transmission is commonly used with modems and for non-networked communication between computers, terminals and other devices.

There are two primary forms of serial transmission: Synchronous (USART) and Asynchronous (UART). Depending on the modes that are supported by the hardware, the name of the communication sub-system will usually include a A if it supports Asynchronous communications, and a S if it supports Synchronous communications

- ***Serial UART***

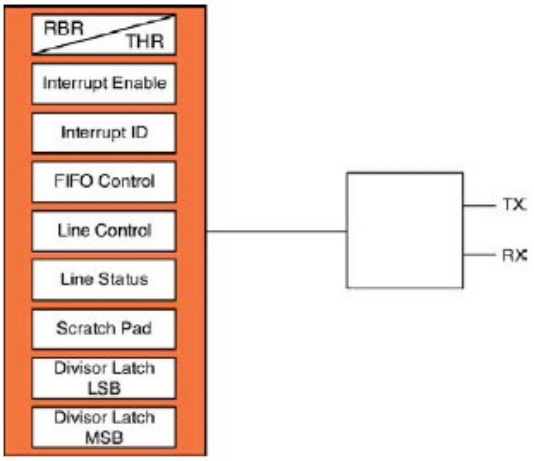
The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. The UART takes bytes of data and transmits the individual bits in a sequential fashion. At the destination, a second UART re-assembles the bits into complete bytes. The UART performs all the tasks, timing, parity checking etc needed for the communication.

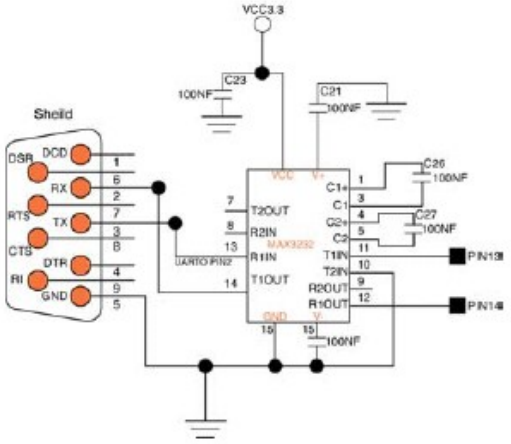
Description:

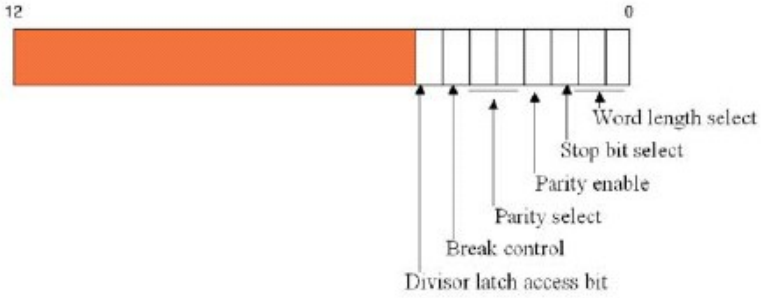
Available registers in an UART:

- RBR, receiver buffer register
- THR, transmitter holding register
- IER, interrupt enable register
- IIR, interrupt identification register
- FCR, FIFO control register
- LCR, line control register
- MCR, modem control register
- LSR, line status register
- MSR, modem status register
- SCR, scratch register
- DLL, divisor latch LSB
- DLM, divisor latch MSB

Block Diagram



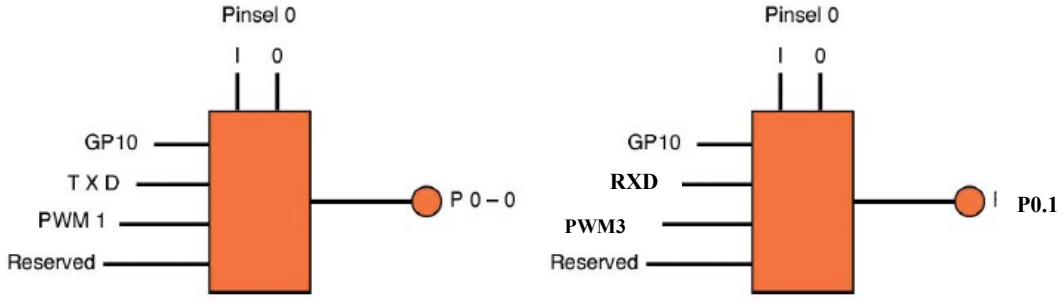




UART Line control register: The LCR configures the format of transmitted data. Setting the DLAB bit allows programming of the BAUD rate generators

Pin Connect Block:

- The pin connect Block allows selected pins of the microcontroller to have more than one function.
- Configuration register control the multiplexers to allow connection between the pin and the on chip peripheral.
- After reset all pins of port0 and port1 are configured as input except during debug or trace mode is enabled.
- Secondary functions are selected through the PINSEL0 register.



Exercise 9.1

Write an assembly program to configure the UART0 on LPC 2148 to accept a character from the keyboard and display the characters on the screen.

Procedure:

- First the ***PIN SELECT BLOCK*** must be programmed to switch the processor pins from GPIO to the

UART functions.

- Next the ***UART LINE CONTROL REGISTER is*** used to configure the format of the transmitter data character.

- In this example the character format is set to 8 bits, no parity and one stop bit.

- In the LCR there is an additional bit called DLAB which is the divisor latch access bit. In order to be

able to program the baud rate generator this bit must be set.

- After setting the baud rate value the DLAB bit in the LCR register must be set back to zero to protect the contents of the divisor registers.

- Once the UART is initialized, characters can be transmitted by writing to the Transmit Holding Register.
- Similarly, characters may be received by reading from the Receive Buffer Register.

In fact both these registers occupy the same memory location, writing a character places the character in the transmit FIFO and reading from this location loads a character from the Receive FIFO.

Program:

```
PINSEL0     EQU   0xE002C000
IOPIN       EQU   0xE0028000

; UART0 registers & their address

U0RBR       EQU   0xE000C000
U0THR       EQU   0xE000C000
U0IER       EQU   0xE000C004
U0IIR       EQU   0xE000C008
U0FCR       EQU   0xE000C008
U0LCR       EQU   0xE000C00C
U0LSR       EQU   0xE000C014
U0SCR       EQU   0xE000C01C
U0DLL       EQU   0xE000C000
U0DLM       EQU   0xE000C004

; control words to configure UART

En_RxTx0    EQU   0x5
S_fmt       EQU   0x83
S_fmt2            EQU   0x3
Speed       EQU   97

; -------sequence to demonstrate UARTO on Serial #I --------

            AREA Serial, CODE, READONLY
            EXPORT __main

__main      BL          setupUART0

;--------To Display the characters to the screen
```

```
                LDR     r0,     =myString1
                LDRB    r1,     [r0],#1                 ;load ASCII code of character

next1           BL              CharOut_0               ; display on Serial#l
                LDRB    r1,     [r0],#1
                CMP     r1,     #0
                BNE             next1

;--------To accept the characters from the keyboard

                MOV     r0,     #0x40000000
inptString      BL              CharIn_0                ; get serial input from Serial#l
                STB     r1,     [r0],#1                 ; put in safe place (memory)
                CMP     r1,      #'*'                   ; check if input was *
                BNE     inptString                      ; repeat loop unless input was *

;--------To Display the characters of my string2 to the screen

                LDR     r0,     =myString2
                LDRB    r1,     [r0],#1                  ;load ASCII code of character

next2           BL              CharOut_0                ; display on Serial#l
                LDRB    r1,     [r0],#1
                CMP     r1,     #0
                BNE             next2

;--------To Display the accepted string

                MOV     r0,     #0x40000000
                LDRB    r1,     [r0],#1

next3           BL              CharOut_0               ;Display on Serial#l
                LDRB    r1,     [r0],#1
                CMP     r1,     #'*'
                BNE             next3


; reset PINSEL0 occurs before UART output completes so add delay

                LDR     r10, =0x8000                    ;simple delay count value
delay           SUBS    r10, r10, #1                    ;count down delay
                BNE     delay                           ;loop until counted down delay


stop            BAL stop

        ;Set up all the SFRs for UART0 requirements only


SetupUART0      STMFD  sp!,     {lr}                    ;not strictly needed but be consistent
                LDR     r4,     =En_RxTx0               ;pattern to enable RxD0 and TXD0
                LDR     r6,     =PINSEL0                ;point to pin control register
                STR     r4,     [r6]                    ;set pins as Tx0 and Rx0

                LDR     r3,     =U0THR                  ;UART0 Base  address
                LDR     r5,     =U0LCR                  ;UART0  LCR address

                MOV     r4,     #S_fmt                  ;for 8 bits, no Parity, 1 Stop bit
                STRB    r4,     [r5]                    ; set format in LCR
```

```
                MOV    r4,     #97              ;for 9600 Baud Rate @ I5MHz VPB Clock
                STRB   r4,     [r3]             ;set baud rate

                MOV    r4,     #S_fmt2          ;set DLAB = 0 so addresses are . .
                STRB   r4,     [r5]             ;.. now TX/RX buffers

                LDMFD sp!,     {pc}             ; return (could use MOV pc,lr)

;----------Output one character using serial port 0

CharOut_0       STMFD sp!,     {lr}             ;not strictly needed here
                LDR    r3,      =U0THR          ;UART base address
                LDR    r5,      =U0LSR          ;UART LSR address

wait_rdy        LDRB   r2,     [r5]             ;get UART       status
                TST    r2,     #0x20            ;check for transmit buffer empty
                BEQ    wait_rdy                 ;loop until TX   buffer empty
                AND    r1,     r1,#0xFF         ;ensure callee  has only set byte
                STRB   r1,     [r3]             ;load transmit  buffer

                LDMFD sp! ,    {pc}             ;return (could   use MOV pc,lr)

;------------Wait for input of one character at serial port 0 then read it

 CharIn_0       STMFD sp!,     {lr}                    ; not strictly needed here
                LDR    r3,      =U0RBR                 ;UART base address
                LDR    r5,      =U0LSR                 ;UART base address


wait_rcd0       LDRB   r2,     [r5]                    ;get UART status
                TST    r2,     #0x1                    ;check if receive buffer has
something
                BEQ    wait_rcd0                       ;loop until RX buffer has value
                LDRB   r1,     [r3]                    ;get received value
                LDMFD sp!,     {pc}                    ;could use MOV pc,lr

myString1       dcb     "\n Enter Your Name: ",0
myString2       dcb     "\n The name you entered is : ",0

                END
```

Exercises

9.2 Using the concept of UART design a calculator that performs (+, -, x, /) of two numbers.
Accept the two 32 bit numbers and the operation from the keyboard and call the respective functions and display the results. (Results should be 64 bit value).

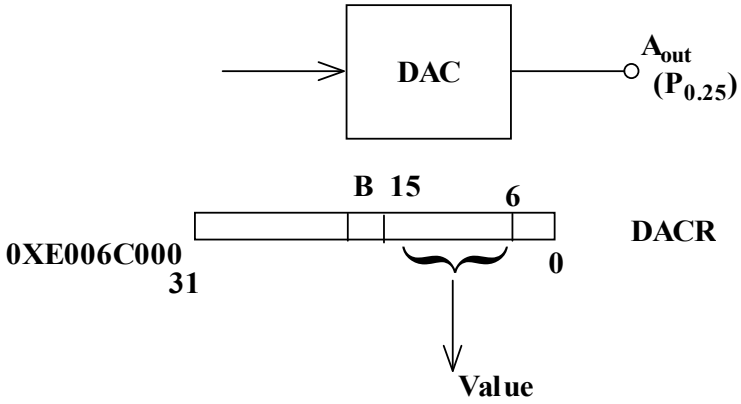
9.3 For the program 9.1 high-level versions is given along with in folder lab9. Go through the c source code and understand the concept.

Lab-10
Digital to Analog Converter

AIM : To use the on chip DAC to generate different waveforms.

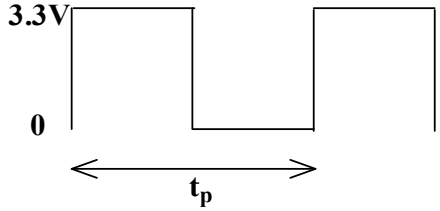
Description:
Basic Operation of a DAC is to take a 10 bit binary value into analog voltage A_{out} which is proportional to the reference voltage $V_{REF.}$

$A_{out} = \frac{Value}{1024} x V_{REF}$ where $V_{REF} = 3.3V$



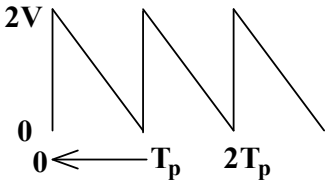
Once A_{out} is enabled, conversion can be started by writing into the value in the control register (15:6)

Ex: 10.1
Write an assembly program to generate a Square Wave.



Hint:
Send a binary value, corresponding to each voltage level and then call a delay corresponding to $t_{p/2}$.

Ex: 10.2
Write an assembly program to generate a Saw Tooth Waveform. Check the program for different step values and observe the waveform.



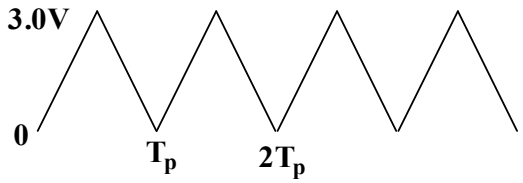
Hint:
Δ = Step Value

N= Total number of Steps = $\frac{(Binary)}{StepValue}$

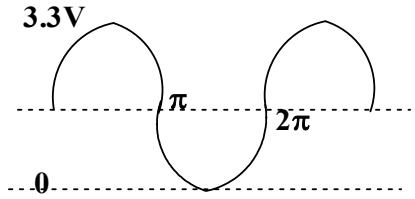
Step Delay = Tp/N

Step1 – Load the Binary value corresponding to 2v
Step2 - Increment/Decrement the binary value in steps ofΔ.
Step3 - Call the Step Delay.
Step4 - Repeat this till the voltage level is reached.
Step5 – Repeat Step1.

Ex: 10.3
Write the assembly program to generate Triangular Waveform



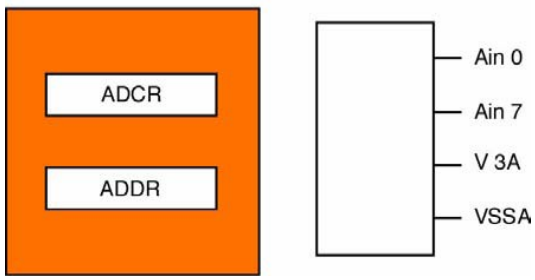
Ex: 10.4
Write an assembly program to generate a Sine Wave



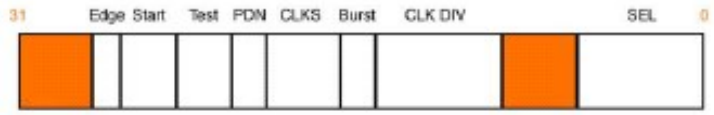
Vout = 512 + 512 Sin(θ)

Lab 11
Analog To Digital Converter (Optional)

The programming interface for the A/D converter is shown below



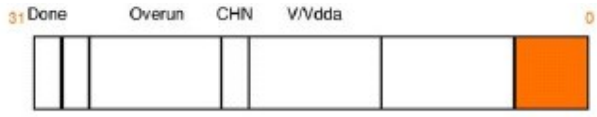
The A/D control register establishes the configuration of the converter and controls the start of conversion. The first step in configuring the converter is to set up the peripheral clock. As with all the other peripherals, the A/D clock is derived from the PCLK. This PCLK must be divided down to equal 4.5MHz.



AD Control register: The control register determines the conversion mode, channel and resolution

PCLK is divided by the value stored in the CLKDIV field plus one. Hence the equation for the A/D clock is as follows: `CLKDIV = ( PCLK/Adclk) - 1`

The A/D has a maximum resolution of 10 bits but can be programmed to give any resolution down to 3 bits. The conversion resolution is equal to the number of clock cycles per conversion minus one. Hence for a 10-bit result the A/D requires 11 ADCLK cycles and four for a 3-bit result. Once you have configured the A/D resolution, a conversion can be made.
The A/D has two conversion modes, hardware and software. The hardware mode allows you to select a number of channels and then set the A/D running. In this mode a conversion is made for each channel in turn until the converter is stopped. At the end of each conversion the result is available in the A/D data register.



AD data register: The data register contains the conversion result, channel overrun error and conversion done flag.

At the end of a conversion the Done bit is set and an interrupt may also be generated. The conversion result is stored in the V/Vdda field as a ratio of the voltage on the analogue channel divided by the voltage on the analogue power supply pin. The number of the channel for which the conversion was made is also stored alongside the result. This value is stored in the CHN field. Finally, if the result of a conversion is not read before the next result is due, it will be overwritten by the fresh result and the OVERUN bit is set to one.

Exercise 11 : Analog To Digital Converter
WAAP that uses the A/D to convert an external voltage source and modulate a bank of LEDs with the result.