Lab Task (Cognitive)

Question 1:

An LED is connected through a resistor to pin PA5 (active-high). The core clock is 16 MHz and one iteration of the software delay loop takes 4 clock cycles.

a) State the sequence of configuration steps required before pin PA5 can be driven as an output. **[3 Marks]**

b) Determine the values (in hexadecimal) that must be written to the BSRR register to set PA5 and to reset PA5. **[3 Marks]**

c) Determine the loop count N required to generate a 500 ms software delay. **[2 Marks]**

d) If the LED is toggled once every 500 ms, determine the blinking period and frequency. **[2 Marks]**

Lab Exercise and Summary

Summary should cover Introduction, Procedure, Data Analysis and Evaluation.

Student's Signature: _____ Date: _____

LABORATORY SKILLS ASSESSMENT (Cognitive)

Obtained Marks	**/10**

Instructor's Signature: _____ Date: _____

LABORATORY SKILLS ASSESSMENT (Psychomotor)

Psychomotor Activity 1: Experimental Setup

S. No	Criteria	Performance	Description	Marks
1	**GPIO Hardware & Software Setup (4)**	Level 4	Correctly prepares the required GPIO hardware and software setup for the experiment independently and according to the laboratory requirements.	4
		Level 3	Correctly prepares the required GPIO hardware and software setup with minor errors or assistance.	3
		Level 2	Partially prepares the required GPIO hardware and software setup and requires considerable assistance.	2
		Level 1	Shows limited ability to prepare the required GPIO hardware and software setup and requires substantial assistance.	1
2	**Pin & Clock Configuration (4)**	Level 4	Correctly prepares the required pin and clock configuration for the experiment independently and according to the laboratory requirements.	4
		Level 3	Correctly prepares the required pin and clock configuration with minor errors or assistance.	3
		Level 2	Partially prepares the required pin and clock configuration and requires considerable assistance.	2
		Level 1	Shows limited ability to prepare the required pin and clock configuration and requires substantial assistance.	1
3	**Build & Debug Parameters (2)**	Level 2	Correctly sets the required build and debug parameters for the experiment.	2
		Level 1	Sets the basic build and debug parameters with minor errors or assistance.	1
Total Obtained				**/10**

If a student is absent:

Criteria	Description	Marks
Absent	Student is absent from the laboratory activity.	**0**

Instructor's Signature: _____ Date: _____

Psychomotor Activity 2: Implementation + Results

S. No	Criteria	Performance	Description	Marks
1	**GPIO C Program & Safe Working Practice (4)**	Level 4	Correctly implements the assigned GPIO C program independently while following appropriate laboratory practices.	4
		Level 3	Implements the assigned GPIO C program with minor errors or assistance and generally follows proper laboratory practices.	3
		Level 2	Partially implements the assigned GPIO C program and requires considerable assistance or repeated guidance.	2
		Level 1	Implementation of the assigned GPIO C program is largely incomplete, requires substantial assistance, or shows poor adherence to proper laboratory and safety practices.	1
2	**Execution & Debugging (2)**	Level 2	Correctly performs the required build, execution and debugging and verifies the expected operation.	2
		Level 1	Performs the basic build, execution and debugging with minor errors or assistance.	1
3	**LED Operation & Output Analysis (4)**	Level 4	Obtains correct LED operation and output analysis and appropriately verifies and interprets the required experimental results.	4
		Level 3	Obtains mostly correct LED operation and output analysis with minor errors in verification or interpretation.	3
		Level 2	Obtains partially correct LED operation and output analysis with noticeable errors or omissions.	2
		Level 1	LED operation and output analysis is largely incorrect or incomplete and requires substantial assistance.	1
Total Obtained				**/10**

If a student is absent:

Criteria	Description	Marks
Absent	Student is absent from the laboratory activity.	**0**

Instructor's Signature: _____ **Date: _____**

LABORATORY SKILLS ASSESSMENT (Affective)

Affective Activity: Lab Report + Viva Voce

S. No	Criteria	Performance	Description	Marks
1	**Lab Report Completeness & Organization (4)**	Level 4	Report is complete, well-organized, properly presented, and demonstrates responsible completion of the assigned laboratory work.	4
		Level 3	Report is complete with minor omissions or organizational issues and demonstrates satisfactory responsibility for the assigned work.	3
		Level 2	Report is partially complete with several omissions and demonstrates limited responsibility for the assigned work.	2
		Level 1	Report has major omissions or poor organization and demonstrates minimal responsibility for the assigned work.	1
2	**GPIO Output & Conclusion (2)**	Level 2	The GPIO output and conclusion are correctly presented and discussed, demonstrating ownership of the assigned work.	2
		Level 1	The GPIO output and conclusion are presented with minor errors or omissions.	1
3	**Viva Voce (4)**	Level 4	Clearly explains the assigned experiment, GPIO configuration, program explanation, working, and results, demonstrating individual responsibility, participation, and effective communication.	4
		Level 3	Explains most of the assigned experiment and results correctly, with minor difficulty, and demonstrates good individual participation.	3
		Level 2	Explains basic aspects of the assigned experiment with some prompting and demonstrates basic participation.	2
		Level 1	Demonstrates limited understanding or involvement and requires substantial prompting to explain the assigned work.	1
Total Obtained				**/10**

The student has not submitted the report and has not appeared in the Viva Voce.

Criteria	Description	Marks
Lab Report and Results Not Submitted	Student has not submitted the report and results file.	**0**
Absent in Viva Voce	Student is absent from the viva voce.	**0**

Instructor's Signature: _____ Date: _____

Experiment No 3

Overview of ARM assembly programming and addressing modes

Objectives

- To learn basics of assembly language and start simulation on Keil
- To learn addressing modes of ARM Instruction Set Architecture

This lab is intended for students who are not familiar with assembler, or have a very distant idea about it. If you have some idea of high-level languages such as Java, Basic C/C++ or Pascal then you will find assembly coding quite easy.

It is assumed that you have some knowledge about number representation in Hexa-decimal and binary. If you are not familiar with numbering systems, then you must go through before starting this lab.

The ARM Architecture

The ARM is a Reduced Instruction Set Computer (RISC) with a relatively simple implementation of a load/store architecture, i.e. an architecture where main memory (RAM) access is performed through dedicated load and store instructions, while all computation (sums, products, logical operations, etc.) is performed on values held in registers. ARM supports a small number of addressing modes with all load/store addresses being determined from registers and instruction fields only.

ARM Software Programming Model

ARM is a load store reducing instruction set computer architecture; it means the core cannot directly operate with the memory. All data operations must be done by registers with the information which is located in the memory. Performing the operation of data and storing the value back to the memory. ARM processor consist of 16 register in which 13 register are data manipulation registers and other 3 registers are special purpose registers

- R13 (Stack Pointer or SP): Holds address of the top of the stack
- R14 (Link Register or LR): Holds return address from subroutine
- R15 (Program Counter or PC): Hold address of the next instruction to be executed

Despite the name of a register, it's the programmer who determines the usage for each general purpose register. The main purpose of a register is to keep a number (variable). The size of the above registers is 32 bits.

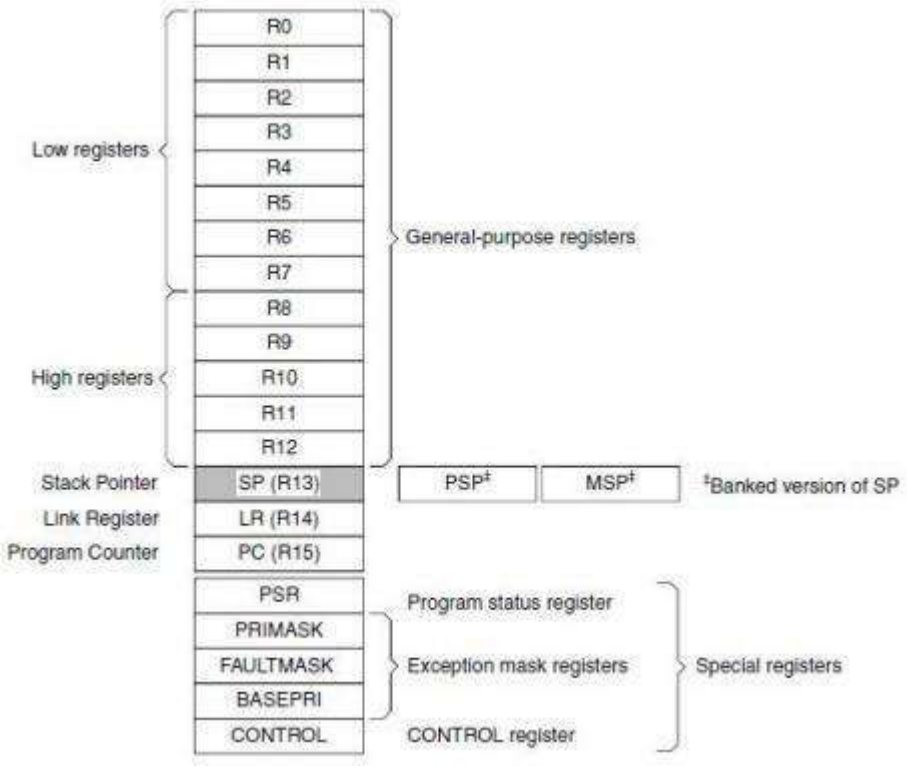


Figure 1 - ARM Registers

Assembly Language

An assembly language is a low-level programming language designed for a specific type of processor. It may be produced by compiling source code from a high-level programming language (such as C/C++) but can also be written from scratch. Assembly code can be converted to machine code using an assembler.

Since most compilers convert source code directly to machine code, software developers often create programs without using assembly language. However, in some cases, assembly code can be used to fine-tune a program. For example, a programmer may write a specific process in assembly language to make sure it functions as efficiently as possible.

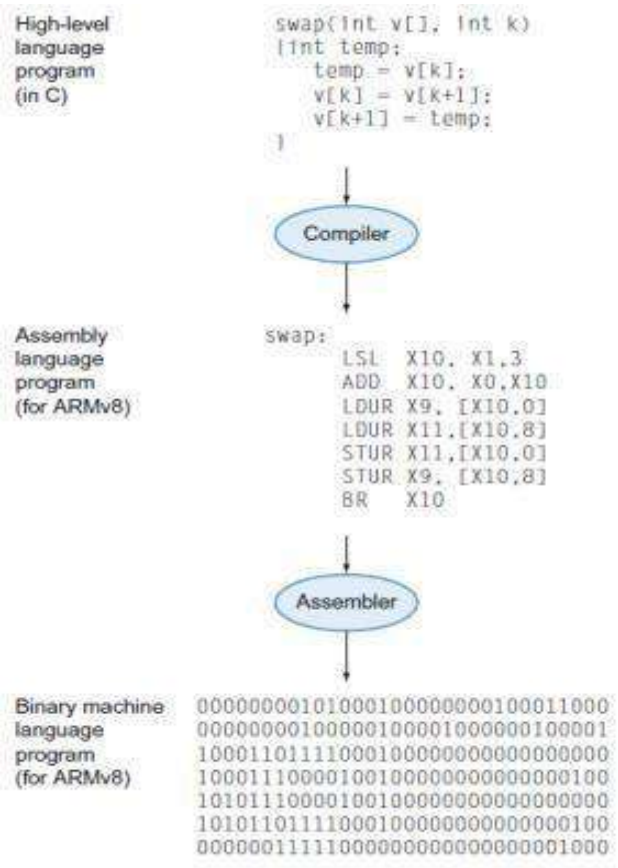


Figure 2 - Compiler and Assembler

Assembly Language Syntax

ARM assembler commonly uses following instruction format:

Normally, the first operand is the destination of the operation. The number of operands in an instruction depends on the type of instruction, and the syntax format of the operand can also be different.

Label

Label is an optional first field of an assembly statement. Labels are alphanumeric names used to define the starting location of a block of statements. Labels can be subsequently used in our program as an operand of some other instruction. When creating the executable file the assembler will replace the label with the assigned value. Labels must be unique in the executable file because an identical label encountered by the Assembler will generate an error.

ARM assembler has reserved first character of a line for the label field and it should be left blank for the instructions with no labels. In some compilers, labels can be optionally ended with colon (:) but it is not accepted by the ARM assembler as an end of the label field.

Defining appropriate labels makes your program look more legible. Program location can be easily found and remembered using labels. It is easier to use certain functionality of your program in an entirely different code .i.e, your code can be used as a library program. You do not have to figure out memory addresses because it is a tedious task especially when the instruction size in your microcontroller is not fixed.

Opcode (Mnemonics)

Opcode is the second field in assembly language instruction. Assembly language consists of mnemonics, each corresponding to a machine instruction. Using a mnemonic you can decide

What operation you want to perform on the operands. Assembler must translate each mnemonic opcode into their binary equivalent.

Operands

Next to the opcode is the operand field which might contain different number of operands. Some of the instructions in ARM Architecture will have no operand while other might have as many as four operands. The number of operands in an instruction depends on the type of instruction, and the syntax format of the operand can also be different. Normally, the first operand is the destination of the operation.

Comments

Comments are messages intended only for human consumption. They have no effect on the translation process and indeed are not acted on by the ARM Assembler. The comment field of an assembly language instruction is also optional. A semicolon (;) signifies that the rest of the line is a comment and is to be ignored by the assembler.

Assembly Language:

In this section we learn to write a simple assembly language program and it can be used as a template for writing any assembly language program. An assembler is required to compile an assembly language program. Keil has an ARM assembler which can compile and build ARM assembly language programs.
Assembly language is only a set of instructions for specific hardware resources. To drive the assembly and linking process, we need to use directives, which are interpreted by the assembler. Some commonly used directives are explained below:

Assembler Directives

Every program to be executed by a computer is a sequence of statements that can be read by the humans. An assembler must differentiate between the statements that will be converted into executable code and that instruct the assembler to perform some specific function.

Assembler directives are commands to the assembler that direct the assembly process. Assembler directives are also called pseudo opcodes or pseudo-operations. They are executed by the assembler at assembly time not by the processor at run time. Machine code is not generated for assembler directives as they are not directly translated to machine language. Some tasks performed by these directives are:

- Assign the program to certain areas in memory.
- Define symbols.
- Designate areas of memory for data storage.
- Place tables or other fixed data in memory.
- Allow references to other programs.

Attribute	Explanation
CODE	Contains machine instructions. READONLY is the default.
DATA	Contains data, not instructions. READWRITE is the default.
READONLY	Indicates that this area should not be written to.
READWRITE	Indicates that this area may be read from or written to.

AREA Directive

AREA directive allows the programmer to specify the memory location to store code and data. Depending on the memory configuration of your device, code and data can reside in different areas of memory. A name must be specified for an area directive. There are several optional comma delimited attributes that can be used with AREA directive. Some of them arediscussed below.

```
AREA    MYCODE, CODE, READONLY
```

EXPORT and IMPORT Directives

A project may contain multiple source files. You may need to use a symbol in a source file that is defined in another source file. In order for a symbol to be found by a different programfile, we need to declare that symbol name as a global variable. The EXPORT directive declares a symbol that can be used in different program files. GLOBAL is a synonym for EXPORT.

The IMPORT directive provides the assembler with a name that is not defined in the current assembly.

```
; Export functions so they can be called from other file

    EXPORT SystemInit
    EXPORT __main
```

Data Reservation Directives (DCB, DCD, DCW)

ARM assembler supports different data definition directives to insert constants in assembly code. This directive allows the programmer to enter fixed data into the program memory and treats that data as a permanent part of the program.

Different variants of these directives are:

1. DCB (Define Constant Byte) to define constants of byte size.

2. DCW (Define Constant Word) allocates one or more halfwords of memory, aligned on two-byte boundaries.

3. DCD (Define Constant Data) allocates one or more words of memory, aligned

on four-byte boundaries.

END Directive

This directive causes the assembler to stop processing the current source file. Every assembly language source module must therefore finish with this directive.

```
__main FUNCTION

    ; Your Assembly Code

    ENDFUNC

    END
```

Addressing Modes

1) Immediate Addressing mode

The addressing mode in which the data operand is a part of the instruction itself is known as immediate addressing mode.

Example

MOV R2, #4929;

MOV R3, #15;

2) Register Addressing mode

By specifying the name of the register as an operand to the instruction, you may access the contents of that register. Consider the ARM Mov (move) instruction:

MOV Destination, Source

This instruction copies the data from the source operand to the destination operand. The 32 bit register are certainly valid operands for this instruction. In this addressing mode both source and destination operand will be registers.

MOV R1, R2 ; valid operands

In this command contents of register R2 will be copied in register R1.

MOV R2, [#123] ; invalid operands

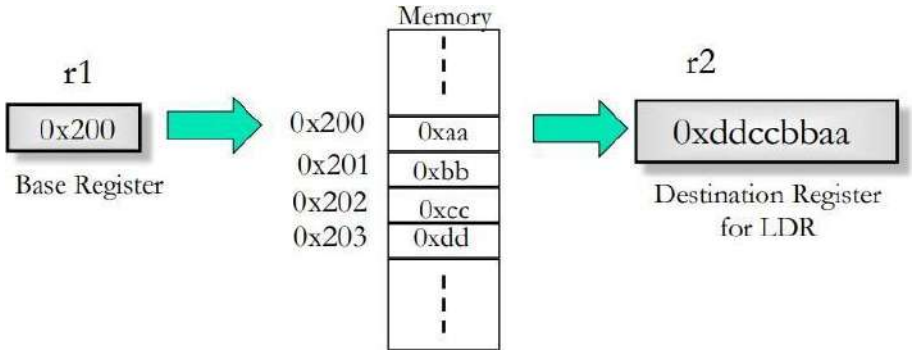
3) Memory Addressing Modes

The ARM is a Load/Store Architecture and it does not support memory to memory data processing operations. Must move data values into registers before using them.
This might sound inefficient, but in practice isn't. Data Transfer instruction is used in MemoryAddressing modes.

- Memory to register or LOAD from memory to register
- Register to memory or STORE from register to memory

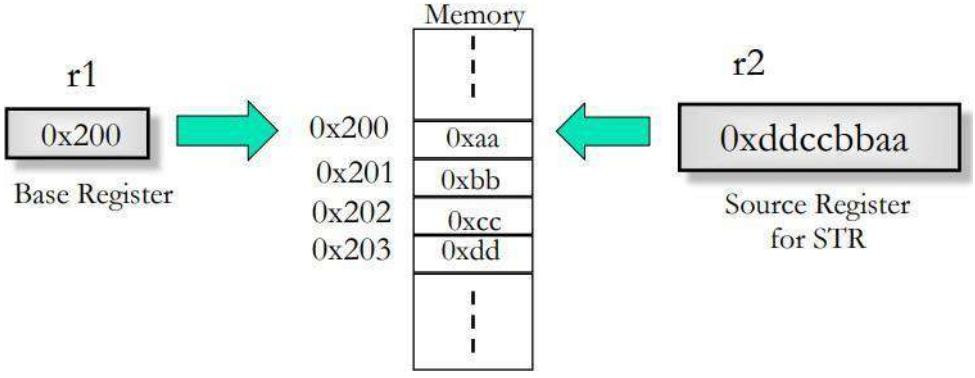
1) LDR R2, [R1]

This instruction will take the address in r1, and then load a 4 byte value from the memory pointed to it into register r2. Note: r1 is called the base register



2) STR R2, [R1]

This instruction will take the address in r1, and then store a 4 byte value from the register r2 to the memory pointed to by r1. Note: r1 is called the base register



3.1 Pre-indexed addressing Mode

The effective address of the operand is the sum of the contents of a base register, R_n, and a signed offset.

Example:

LDR R0, [R1, #12]

This instruction will take the pointer in R1, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register R0

STR R0, [R1, #-8]

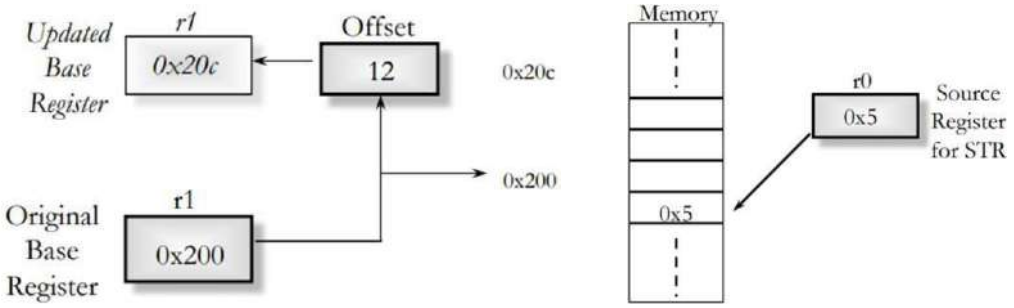
This instruction will take the pointer in R0, subtract 8 bytes from it, and then store the value from register R0 into the memory address pointed to by the calculated sum

R1 is called the base register. #constant is called the offset. Offset is generally used in accessing elements of array or structure: Base register points to beginning of array orstructure. Base register is first updated, the updated address is used.

3.2 Post-indexed addressing Mode

The effective address of the operand is the contents of Rn. The offset is then added to this address and the result is written back into Rn.

Example: STR R0, [R1], #12



3.1 Load/Store Multiple Registers

In ARM Assembly, LDM/STM assembly instructions are used to load/store multiple registers.

LDMxx → Load Multiple Registers

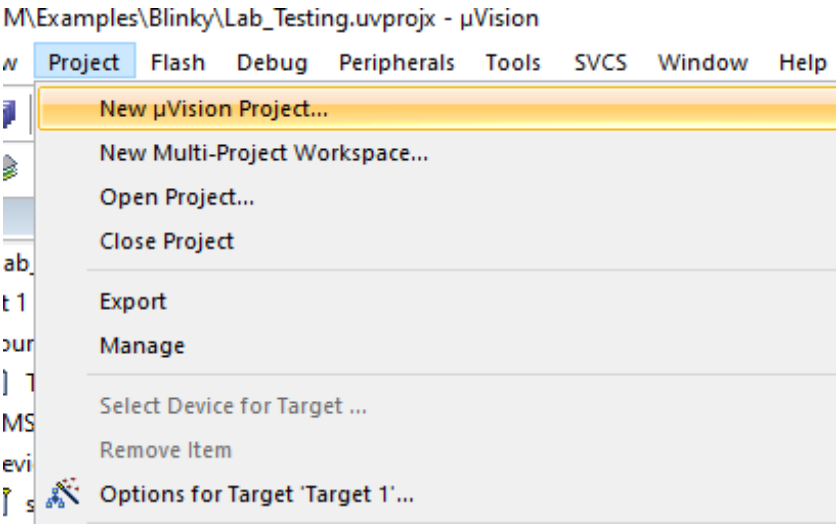
STMxx → Store Multiple Registers

STMFD/LDMFD → Load/Store Multiple Register Full Descending

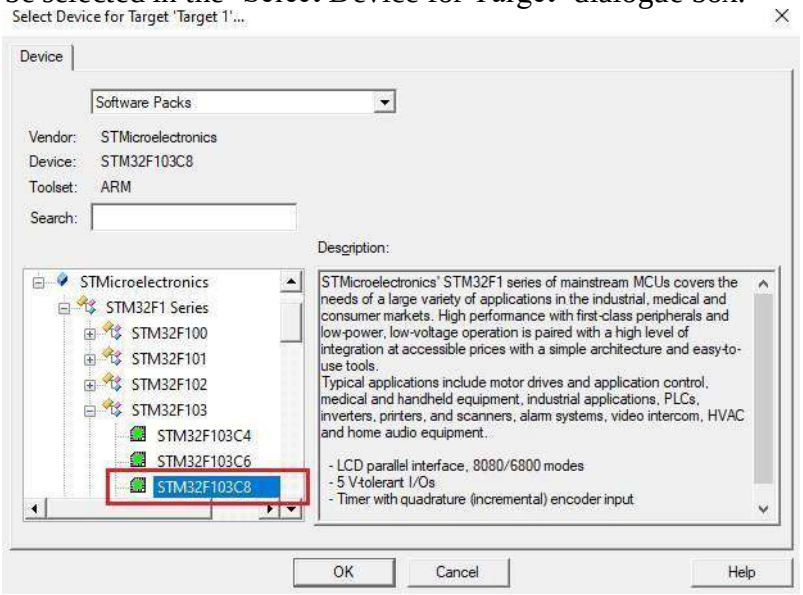
STMIA/LDMIA → Load/Store Multiple Increase After

Assembly Programming in KEIL:

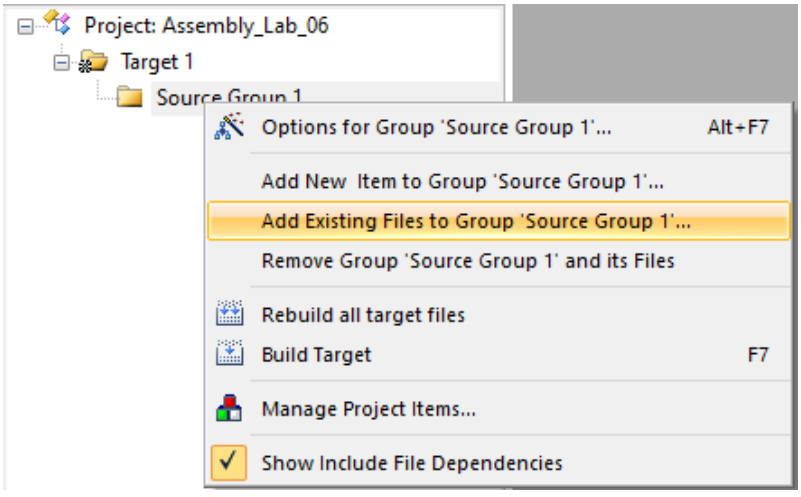
For doing Assembly Programming in Keil, the first step is to create a new uVision Project and selecting the device which is STM32F103C8.



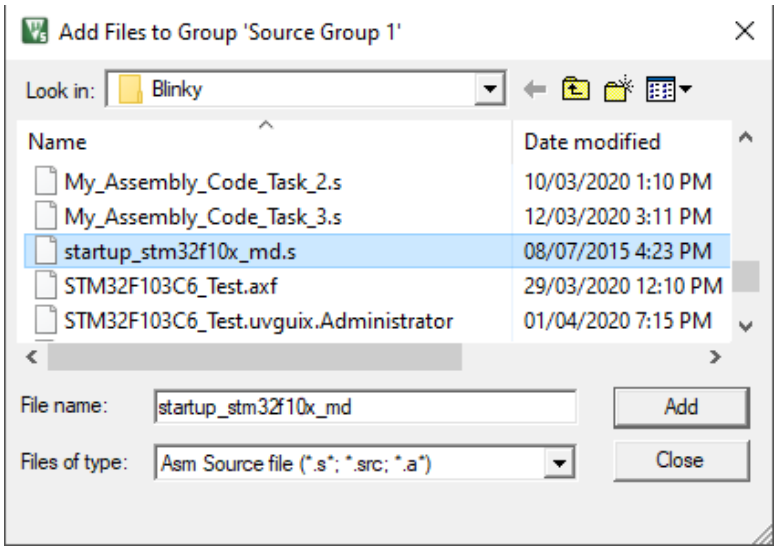
The device will be selected in the 'Select Device for Target' dialogue box.



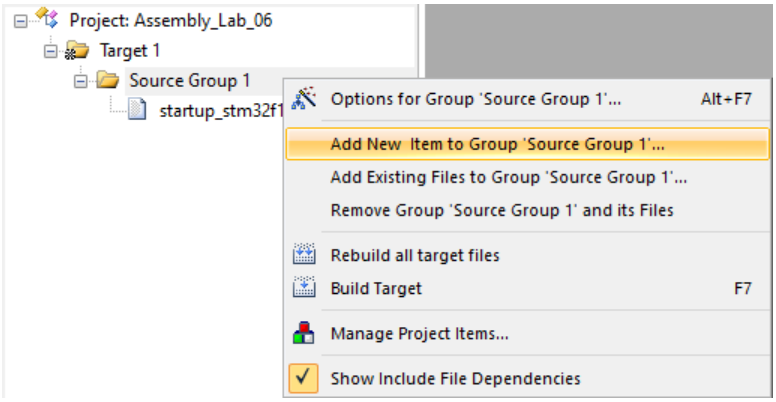
After that, the only requirement is to add the startup file for the respective device.



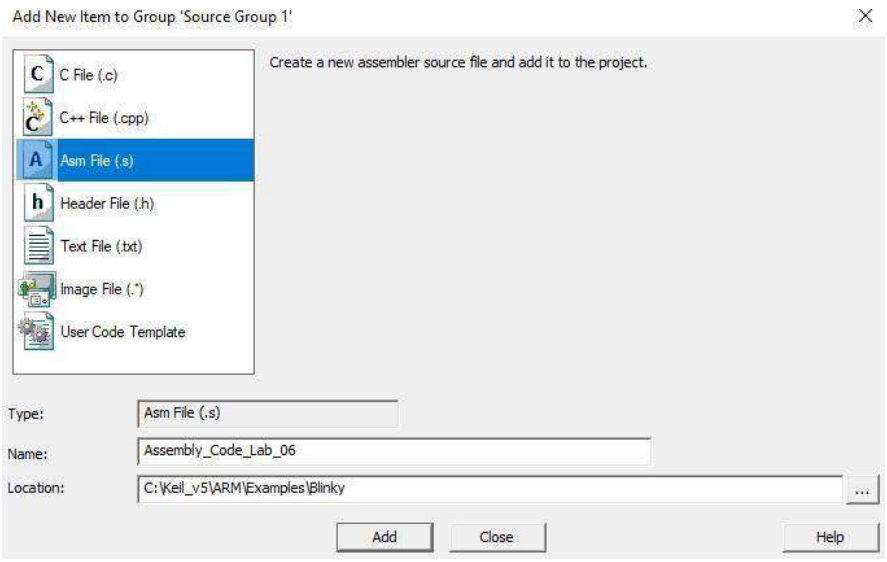
Find the startup file in your PC and add it into the Source Group 1.

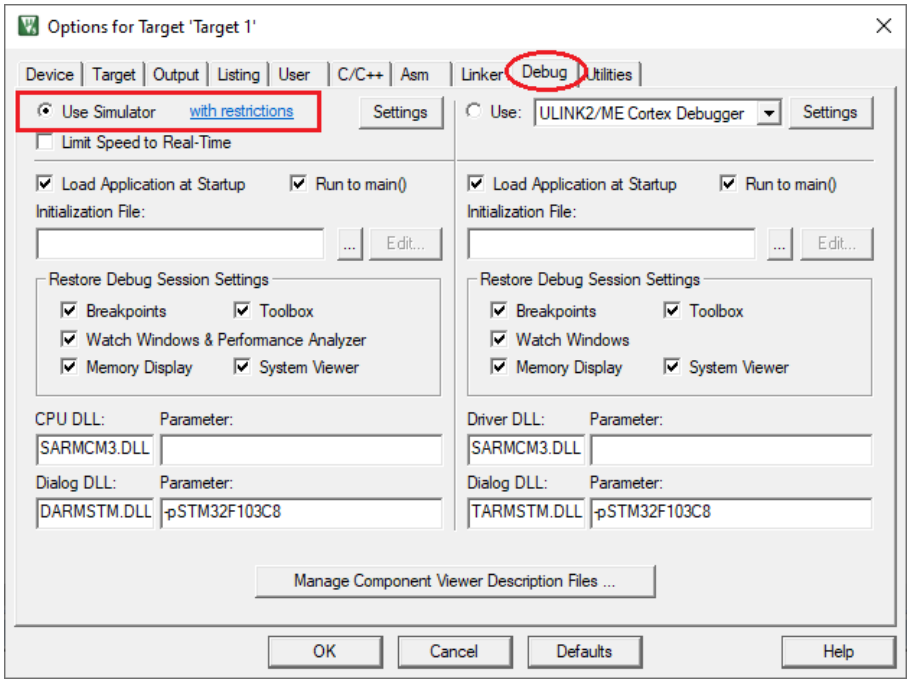


Then add the new Assembly file to write your Assembly Code.



After adding the Assembly file for coding, go to 'Options for target', use the Debug option to select the 'Use Simulator' option for simulation and debugging of your code.




Options for Target 'Target 1'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Use Simulator
with restrictions
Settings
Limit Speed to Real-Time
Load Application at Startup
Run to main()
Initialization File:
Edit...
Restore Debug Session Settings
Breakpoints
Toolbox
Watch Windows & Performance Analyzer
Memory Display
System Viewer
CPU DLL:
Parameter:
SARMCM3.DLL
DARMSTM.DLL
-pSTM32F103C8
Use:
ULINK2/ME Cortex Debugger
Watch Windows
Driver DLL:
TARMSTM.DLL
Manage Component Viewer Description Files ...
OK
Cancel
Defaults
Help

Lab Task (Psychomotor)

Activity 01:

Below is an Assembly Program which defines two 32-bit Arrays having five elements each. The program will add 5 into each element of Array 1 and stores it in Array 2.

```
3 ; Export functions so they can be called from other file
4
5     EXPORT SystemInit
6     EXPORT __main
7
8     AREA    myData, DATA, READWRITE         ; Define a data section
9 ; Define an array with five integers
10 Array2 DCD 0, 0, 0, 0, 0
11
12    AREA    MYCODE, CODE, READONLY
13 Array1 DCD 1, 2, 3, 4, 5
14
15 SystemInit FUNCTION
16
17    BX      LR                  ;Return from function
18
19    ENDFUNC
50 __main FUNCTION
51
52    LDR     R0,=Array1
53    LDR     R1,=Array2
54
55    LDR     R5,[R0], #4
56    ADD     R5,R5, #5
57    STR     R5,[R1],#4
58
59    LDR     R5,[R0], #4
60    ADD     R5,R5, #5
61    STR     R5,[R1],#4
62
63    LDR     R5,[R0], #4
64    ADD     R5,R5, #5
65    STR     R5,[R1],#4
66
67    LDR     R5,[R0], #4
68    ADD     R5,R5, #5
69    STR     R5,[R1],#4
70
71    LDR     R5,[R0], #4
72    ADD     R5,R5, #5
73    STR     R5,[R1],#4
74
75    ENDFUNC
76    END
```

Activity 02:

Write an assembly program which defines two Arrays of 16 bits. Program will copy the contents of first Array into second Array.

- **Contents of both Arrays should be of 16 Bits.**
- **Array-1 contents (9, 9, 9, 0, 1, 4, 3, 1, 2, 6) ; Use your Reg No. for last six values**
- **Array-2 contents (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)**
- **Load the Arrays into Registers of your choice**
- **By using a third Register, load the contents of Array-1 and store it in Array-2**
- **Post-indexed Addressing mode will be used.**

Activity 03:

Write an assembly program that will define two 32-bit Arrays of twenty values in each array. Program will copy the first Array into second Array using Multiple Load Store Instruction.

- **Copy the data by transferring 5 registers at a time.**
- **Copy the data by transferring 10 values at a time.**

Activity 04:

What functions do the following commands perform?

1) LDR R1 , [R2 , #16]

2) MOV R2 , [#123]

3) STR R6 , [R8 , #-4]

4) LDMIA R2! , { R6 – R11 }

Lab Task (Cognitive)

Question 1:

Memory contents (word-aligned, 32-bit words) are shown below. Before each instruction, assume R1 = 0x20000000 (treat the three parts independently).
0x20000000 : 0x0000000A
0x20000004 : 0x00000014
0x20000008 : 0x0000001E
0x2000000C : 0x00000028

a) Apply offset addressing: determine R0 and R1 after LDR R0, [R1, #8]. **[2 Marks]**
b) Apply pre-indexed addressing: determine R0 and R1 after LDR R0, [R1, #4]! **[3 Marks]**
c) Apply post-indexed addressing: determine R0 and R1 after LDR R0, [R1], #4 **[3 Marks]**
d) Name the addressing mode of each instruction and state which of them modify the base register. **[2 Marks]**