DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

15CSE285 Embedded Systems Lab (Keil uVision)

AMRITA SCHOOL OF ENGINEERING, COIMBATORE

AMRITA VISHWA VIDYAPEETHAM COIMBATORE 641 112

Syllabus

B.Tech/II Year CSE/IV Semester **L T P C**

15CSE285/Embedded Systems Lab **0 0 2 1**

Intel 8086 Assembly Program for Arithmetic and logical operations. Intel 8086 Procedures and Macros. ARM Assembly program for Arithmetic and Logical operations, Assembly program for Multi-byte operations, Control manipulation, String manipulation, Assembly program for Thumb Instructions, Embedded C Programming using Keil Simulator – Simple C programs, Port Programming, Peripheral Interfacing – Keypad, Motor, LED.

Course Outcomes:

COs	Course Outcomes	BTL
CO01	Assembly Program using Intel 8086 Arithmetic and Logical Instructions of 8086 Microprocessor	L3
CO02	Design Embedded System Unit making use of MCUs, I/Os, Timers, Interrupts and Communication Interfaces	L4
CO03	Assembly Program using Instruction Set Architecture of ARM 7 TDMI	L3
CO04	Hardware and Software Design of Embedded System using ARM 7 LPC2148 and validating the outcome	L4

CO-PO Mapping

	CO #	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO11	PO12	PSO1	POS2
	CO01	3	2	3	1	2				1	1		1	1	
	CO02	3	2	3	1	2				1	1		1	1	
	CO03	3	3	3	3	2				1	1		3	2	2
	CO04	3	2	3	2	3				1	1		3	2	2

LIST OF EXPERIMENTS

ARM 7 - LPC 2148 Based Experiments

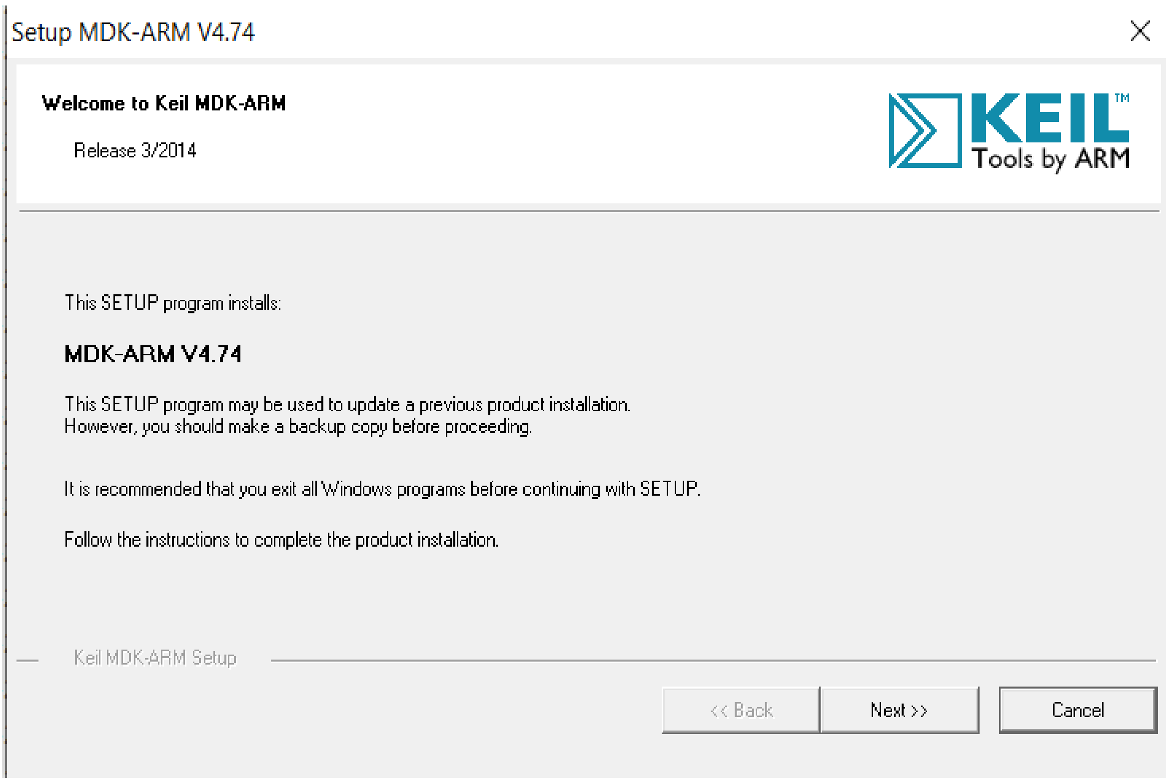
- ARM Assembly Program for Arithmetic and Logical Operations. [CO03]
- ARM Assembly Program for Multi-byte Operations [CO03]
- ARM Assembly Program for Control Manipulations [CO03]
- ARM Assembly Program for String Manipulations [CO03]
- ARM Assembly Program for Thumb Instructions. [CO03]
- Embedded C Program for ARM 7 LPC 2148 using ARM Keil Simulator [CO03]

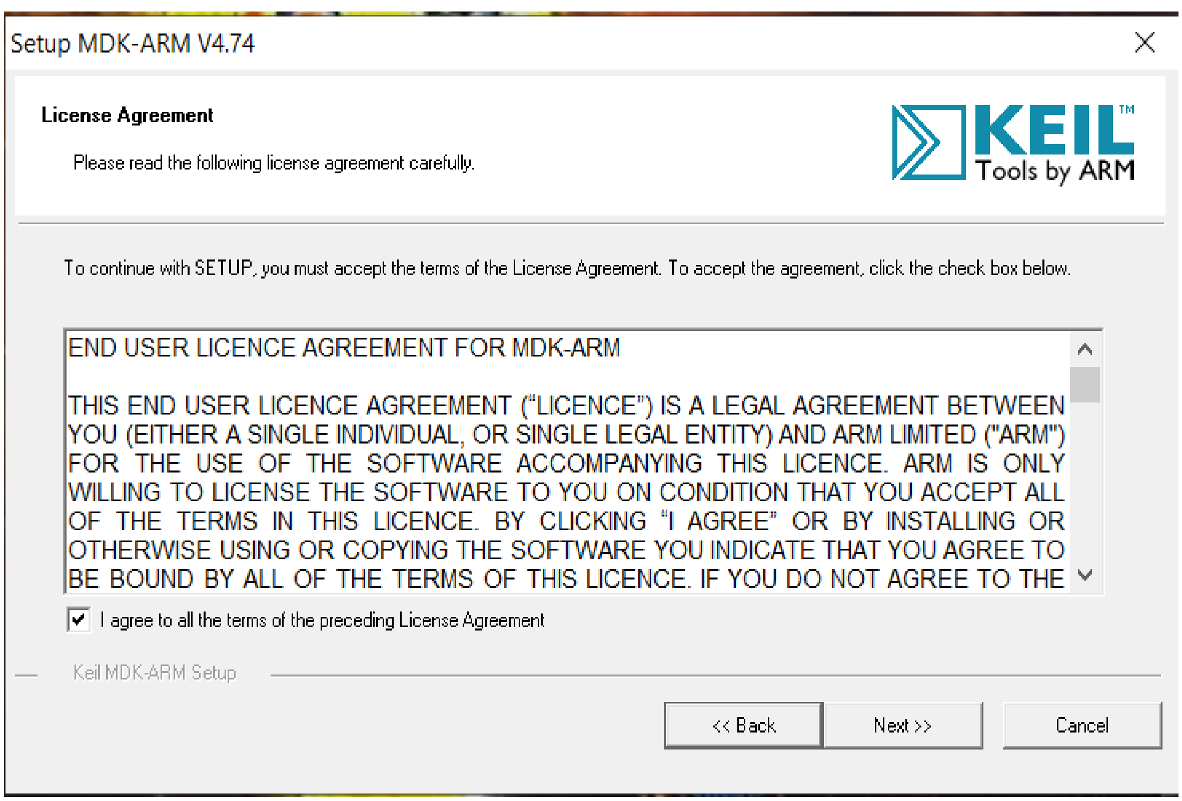
Embedded C Program for Port Programming and Peripheral Devices Interfacing.

- Study of ARM Trainer Board system [CO02]
- Interfacing of real time clock and serial port. [CO02]
- Interfacing keyboard and LCD. [CO02]
- Interfacing of stepper motor and temperature sensor. [CO02]
- Interfacing of Switch and LED / Matrix keypad [CO04]
- Interfacing of DC Motor [CO04]
- Interfacing of ADC and DAC. [CO04]

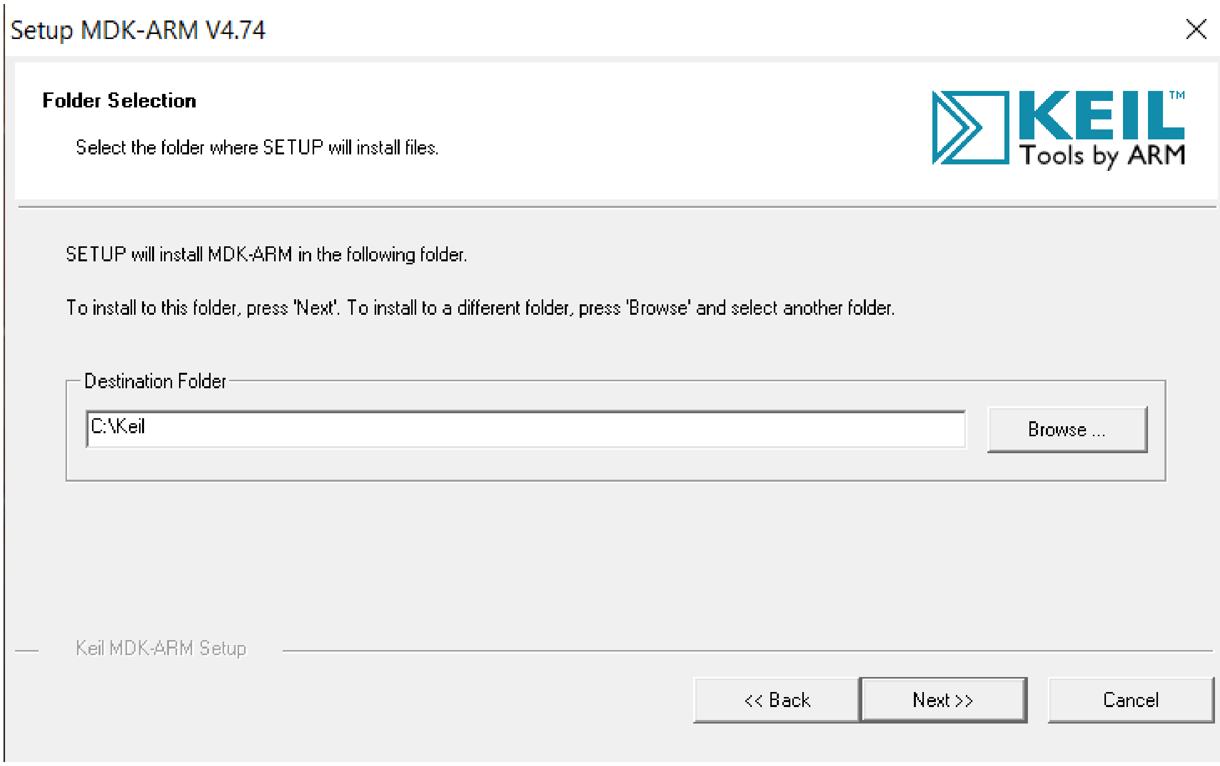
Installation of Keil IDE

Download Keil IDE version 4.74 installer and invoke the application

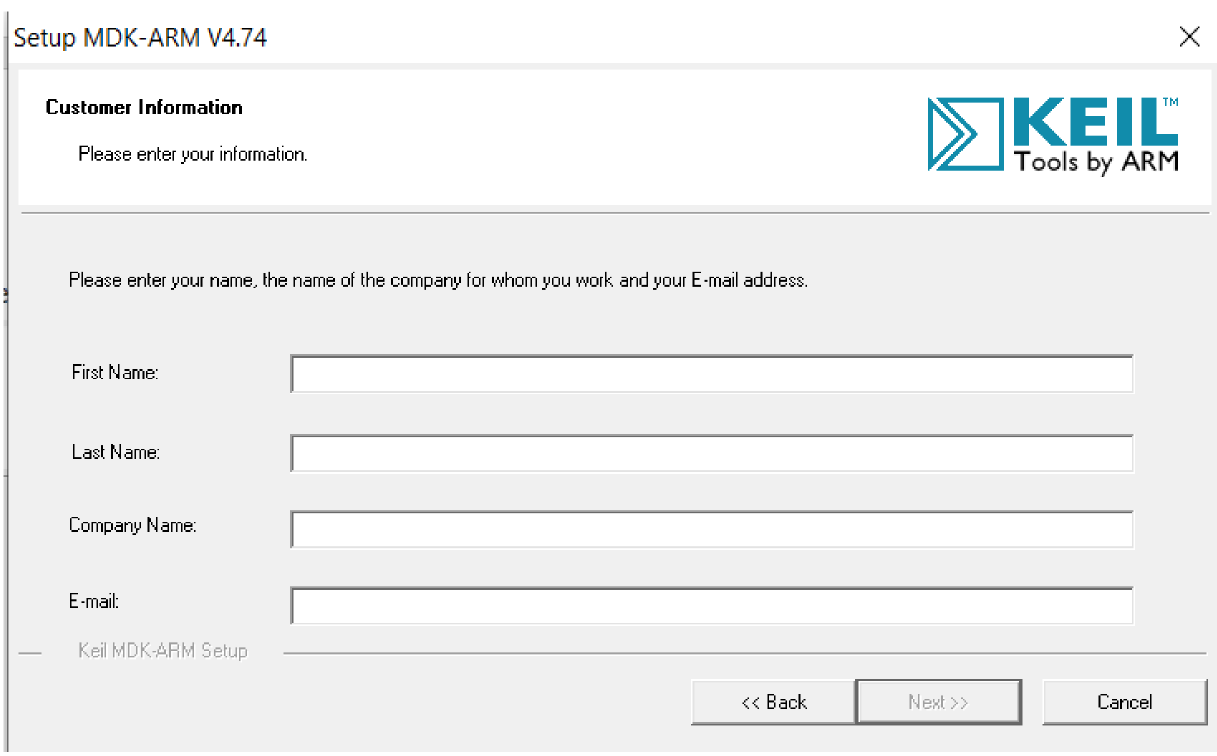




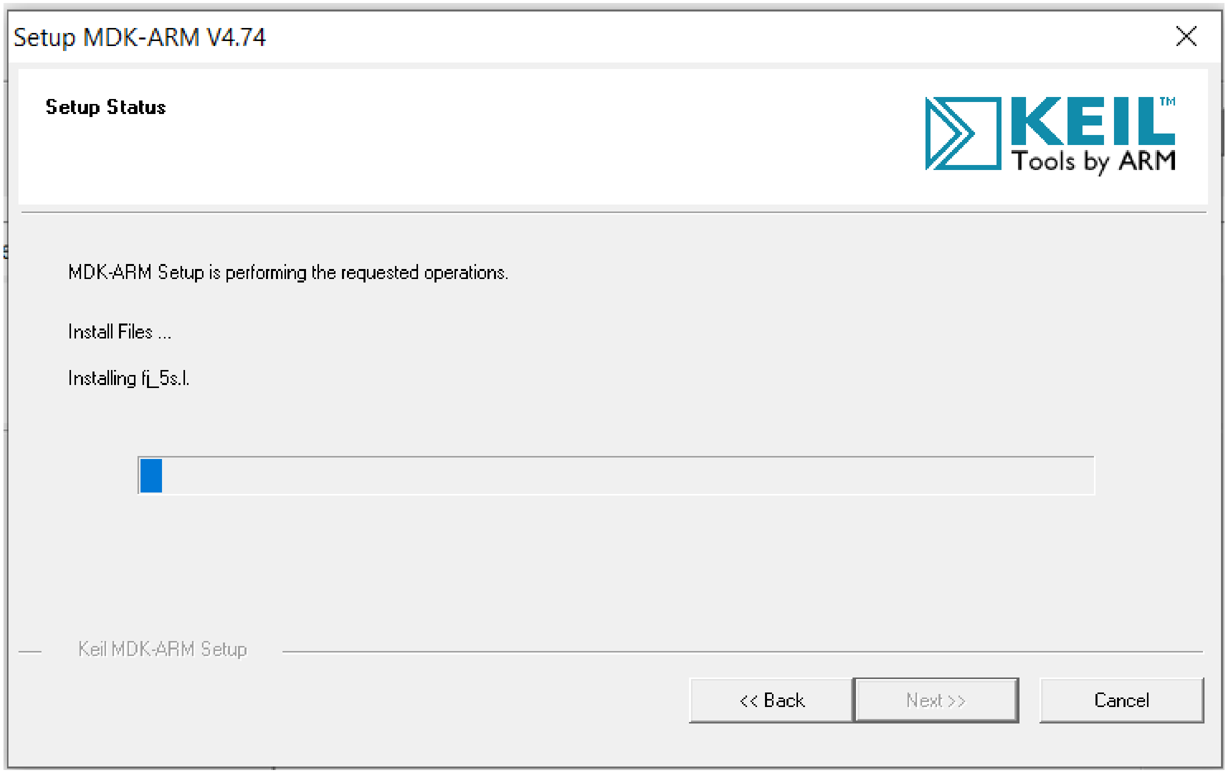
Choose the folder where you want to install Keil

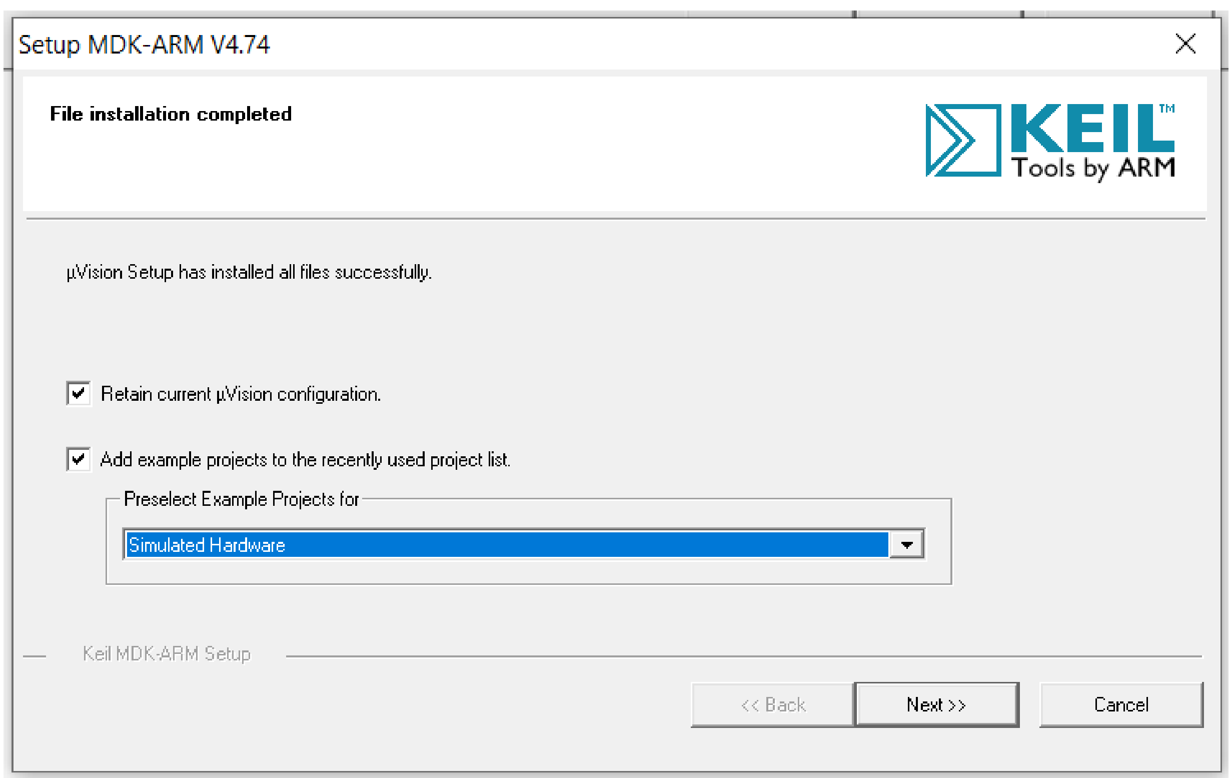


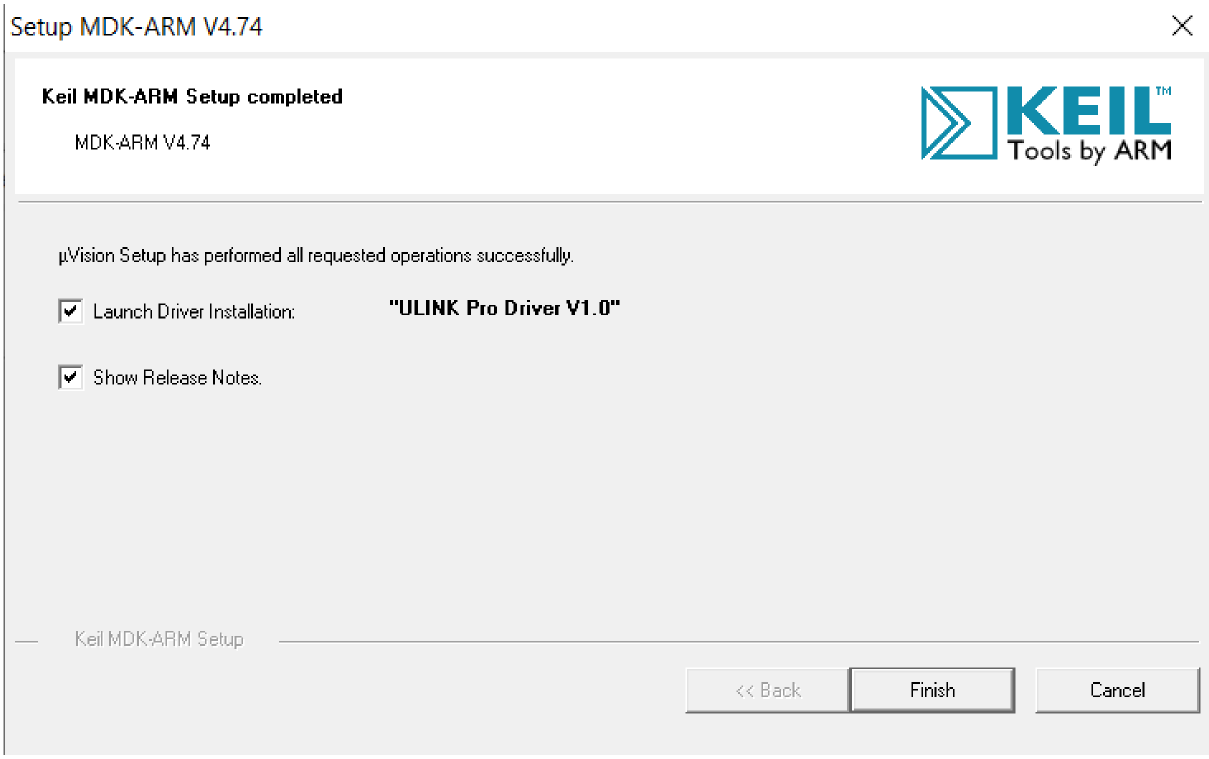
Fill your details



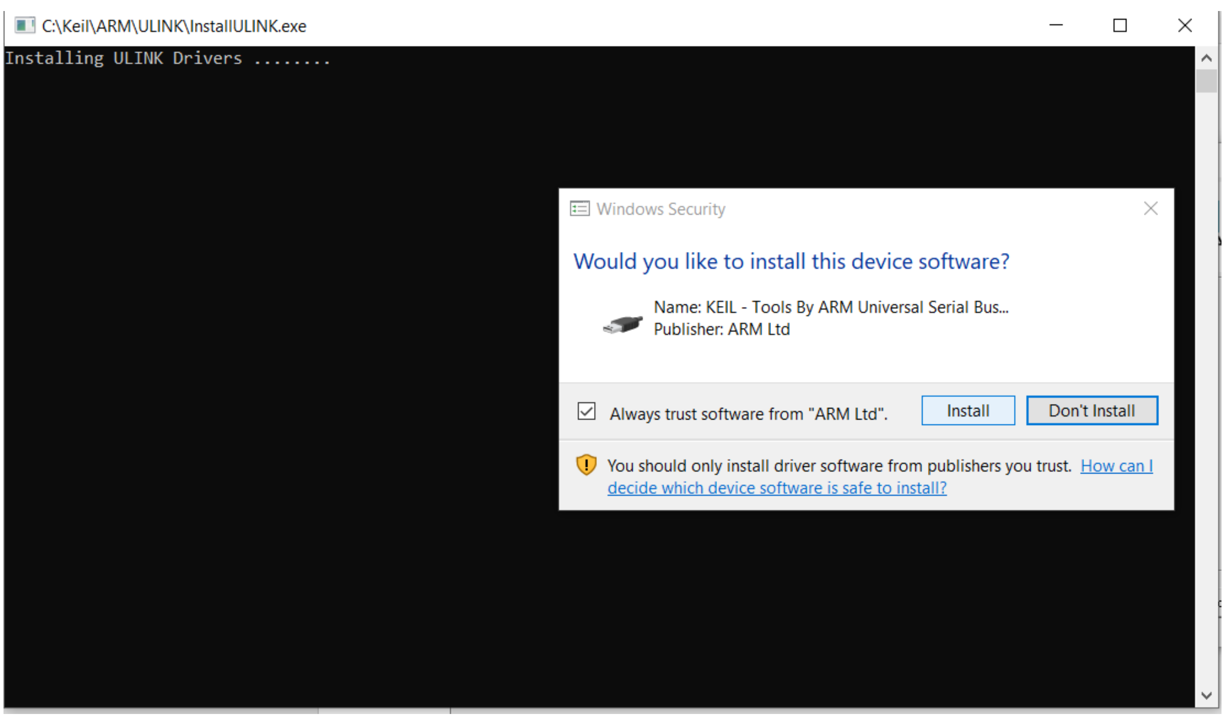
Installation will begin



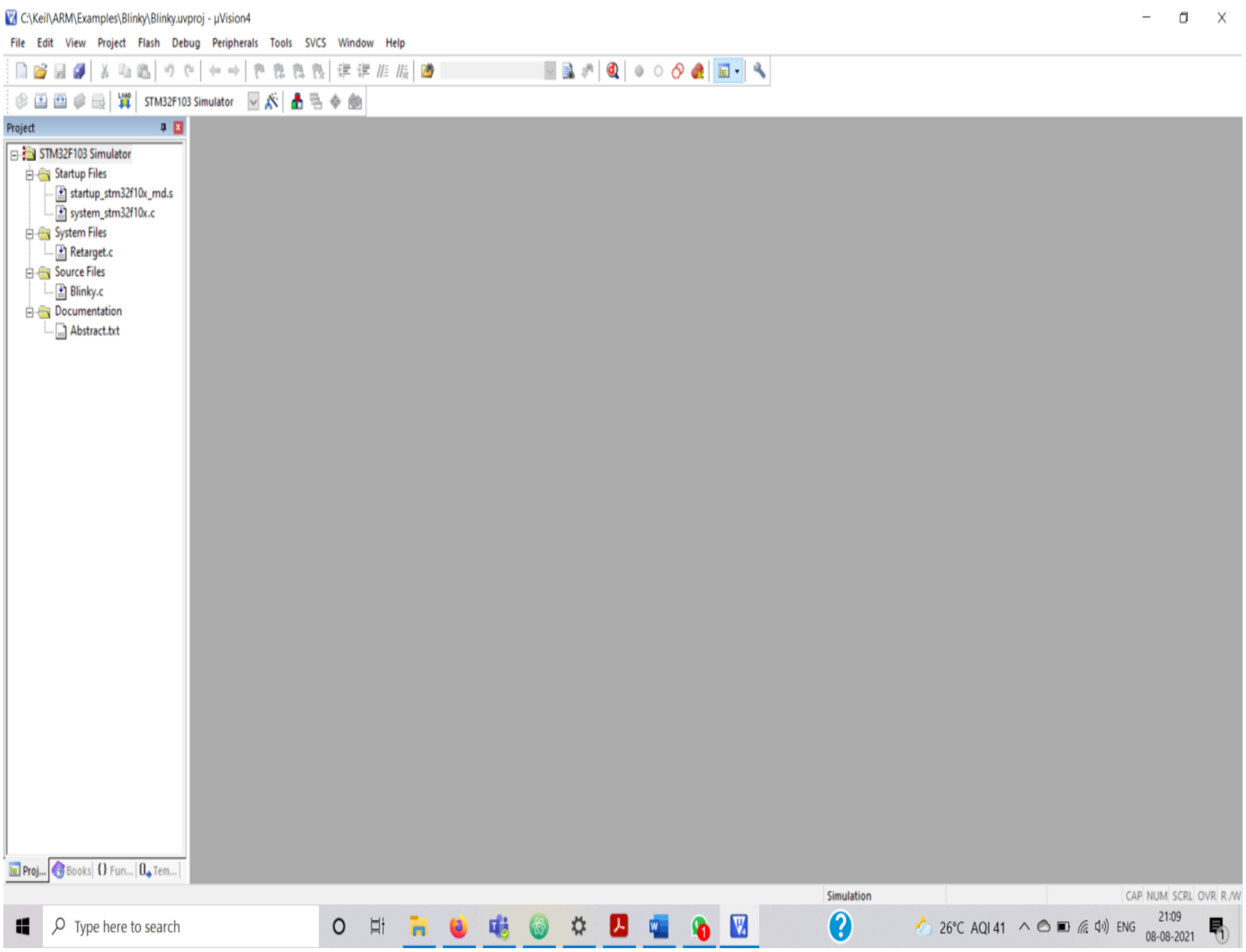




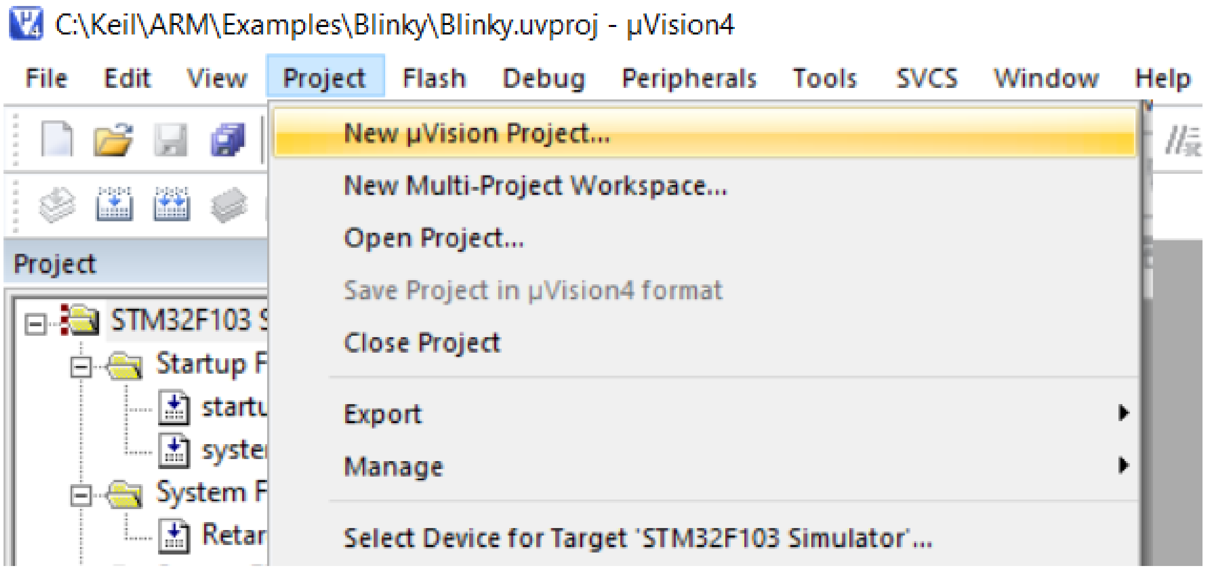
Install the device software



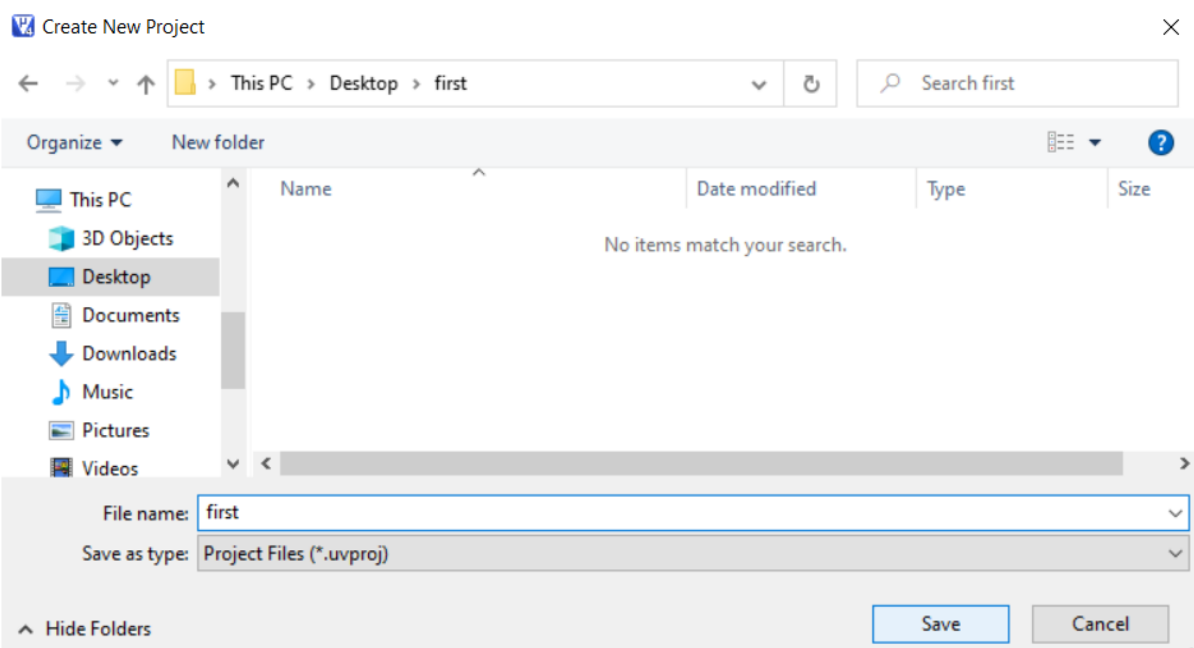
Invoke the Keil uVision4 application



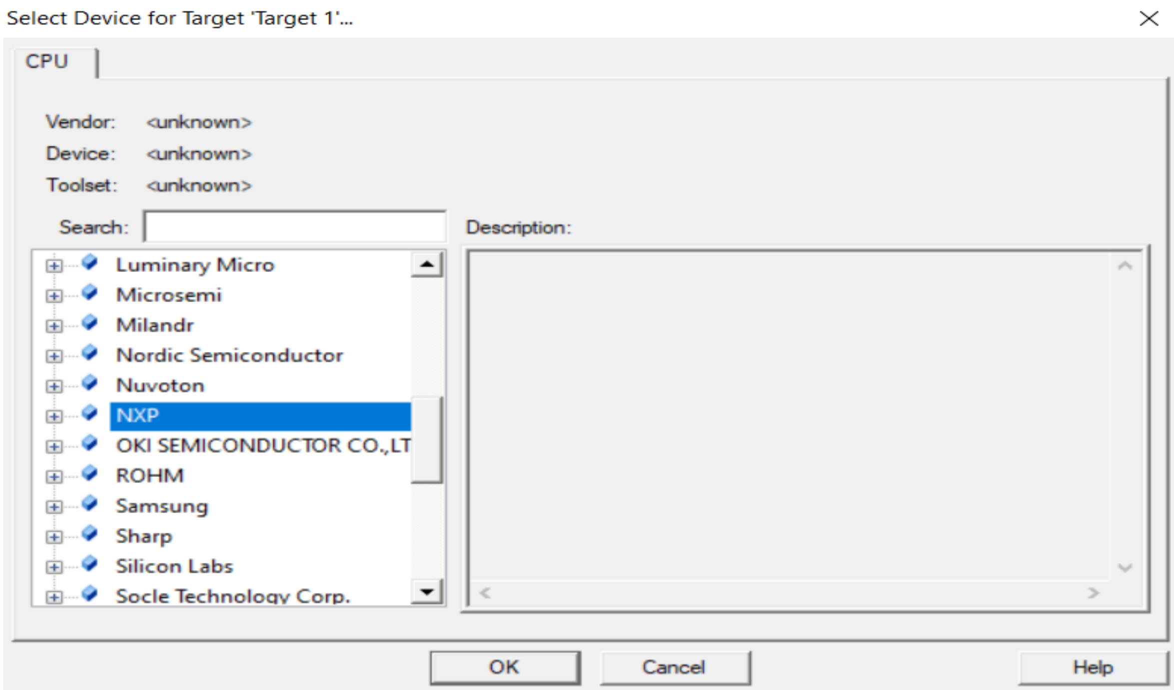
Create a New uVisionProject



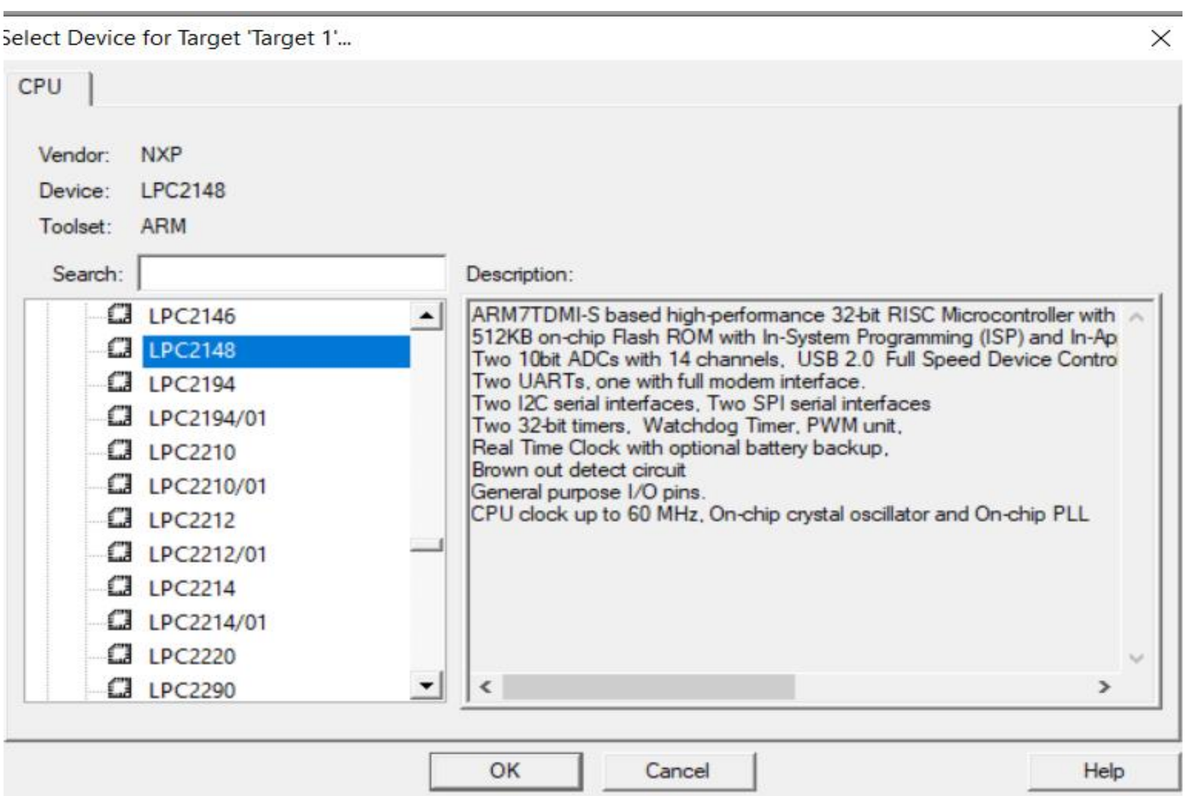
Save it in a folder(preferably empty)

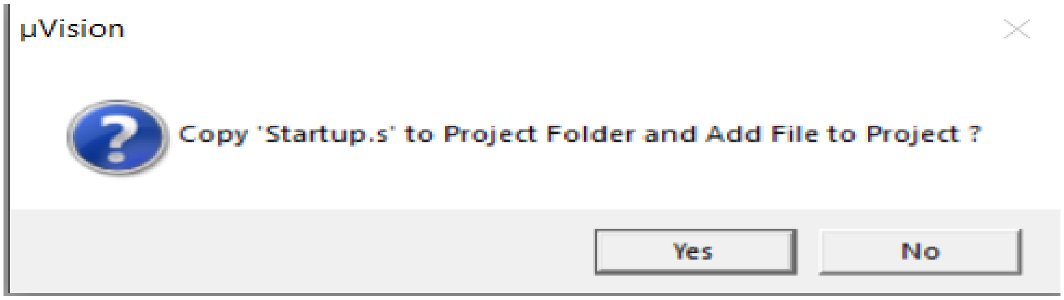


Choose vendor as NXP

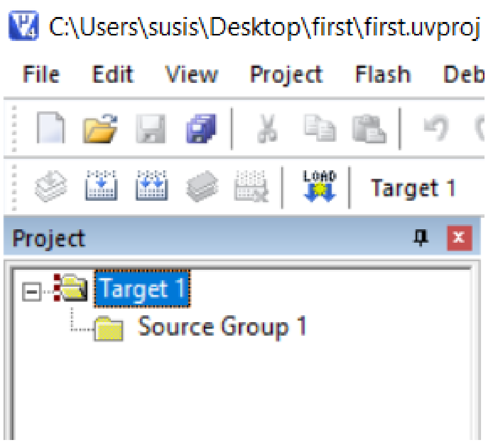


Choose LPC2148 and click OK

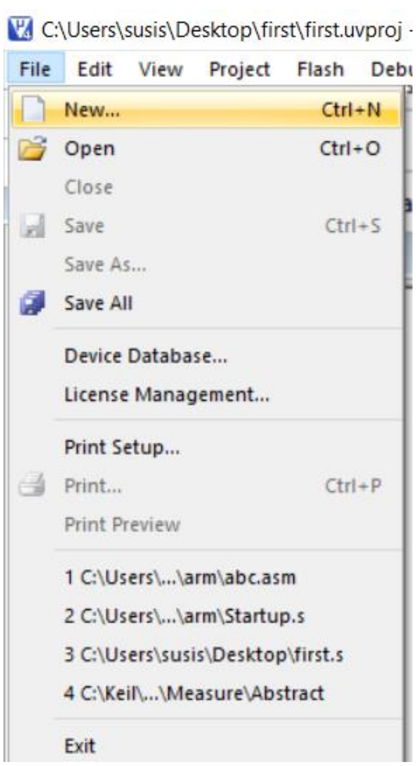


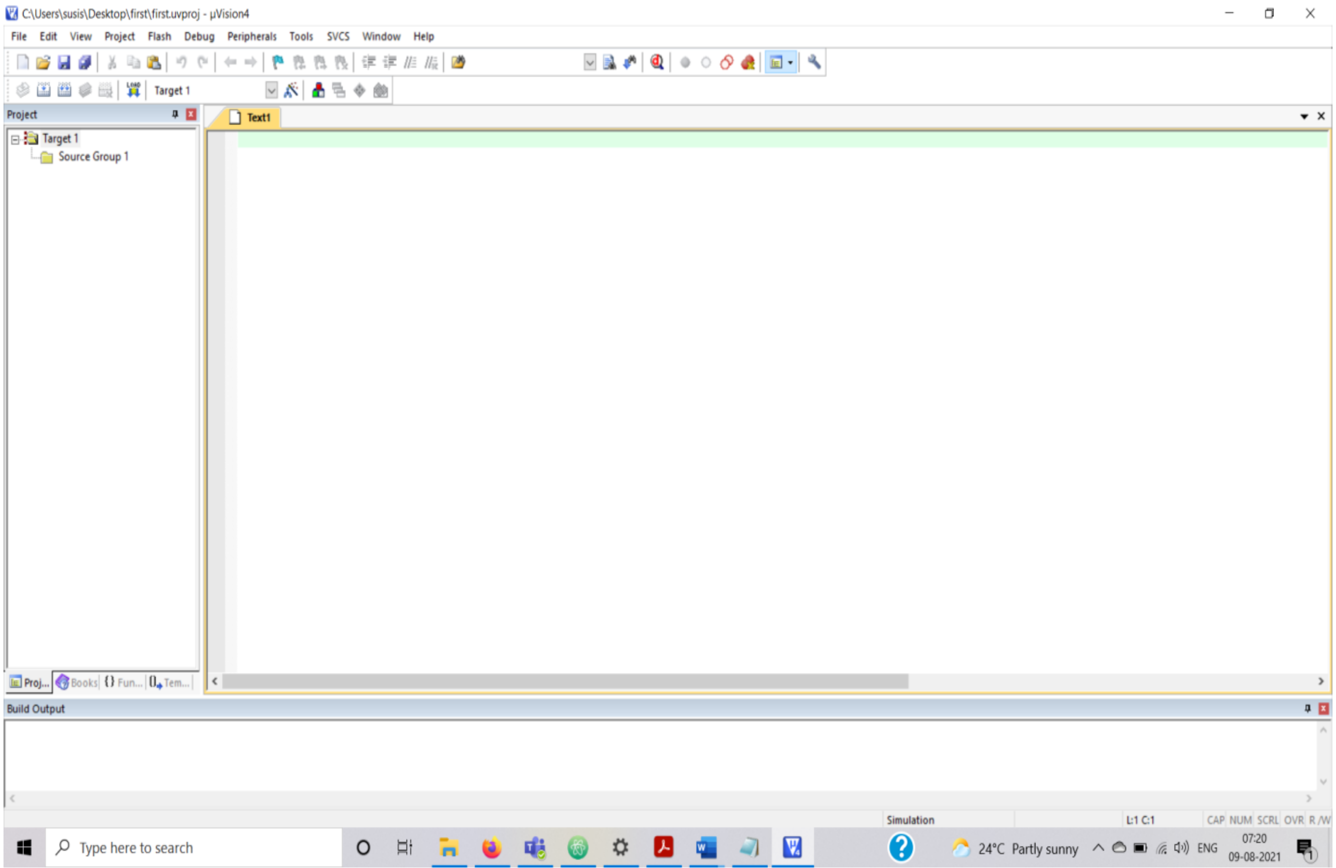


Directory Structure

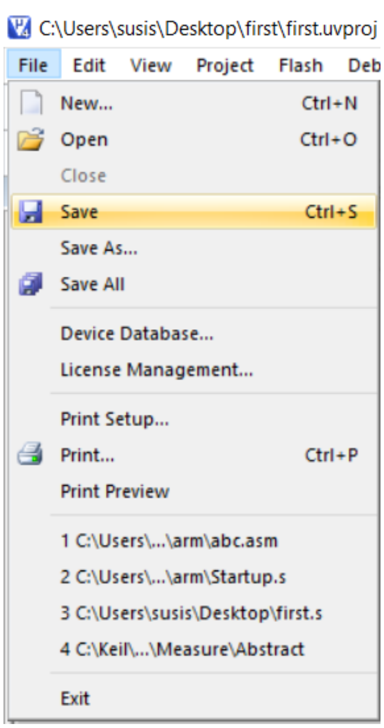


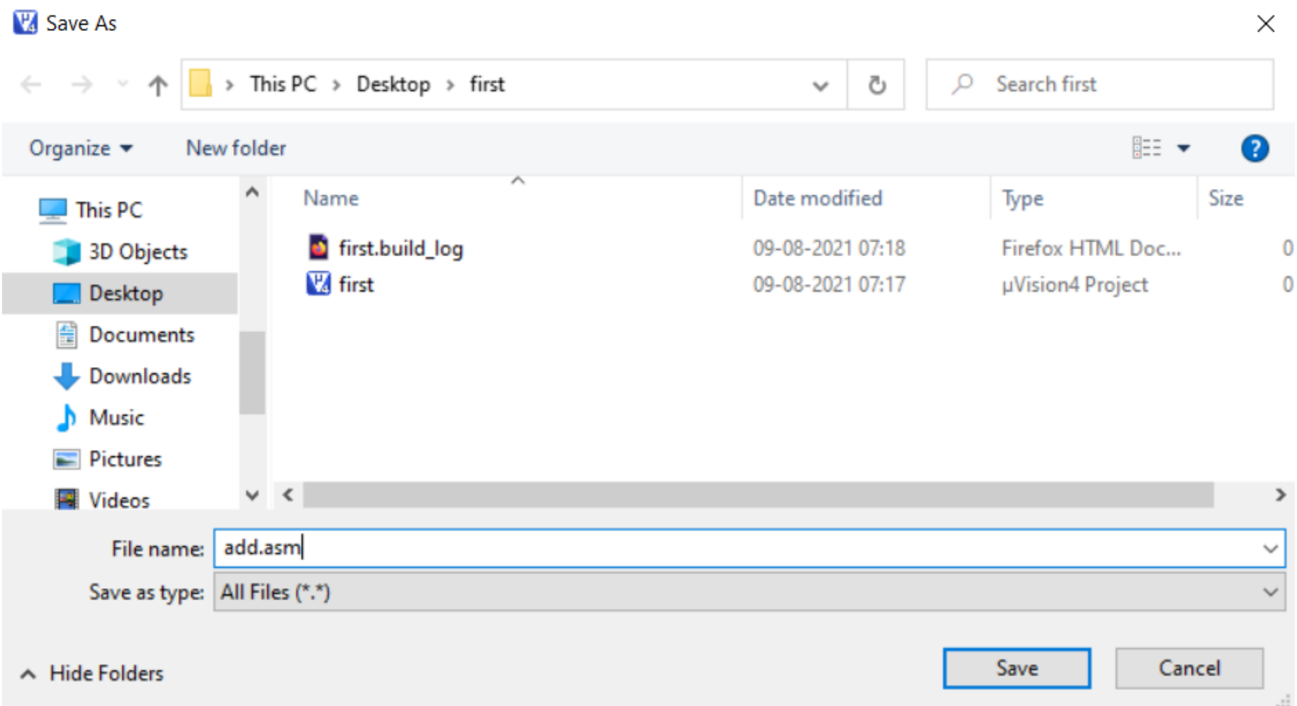
Then create a new empty text



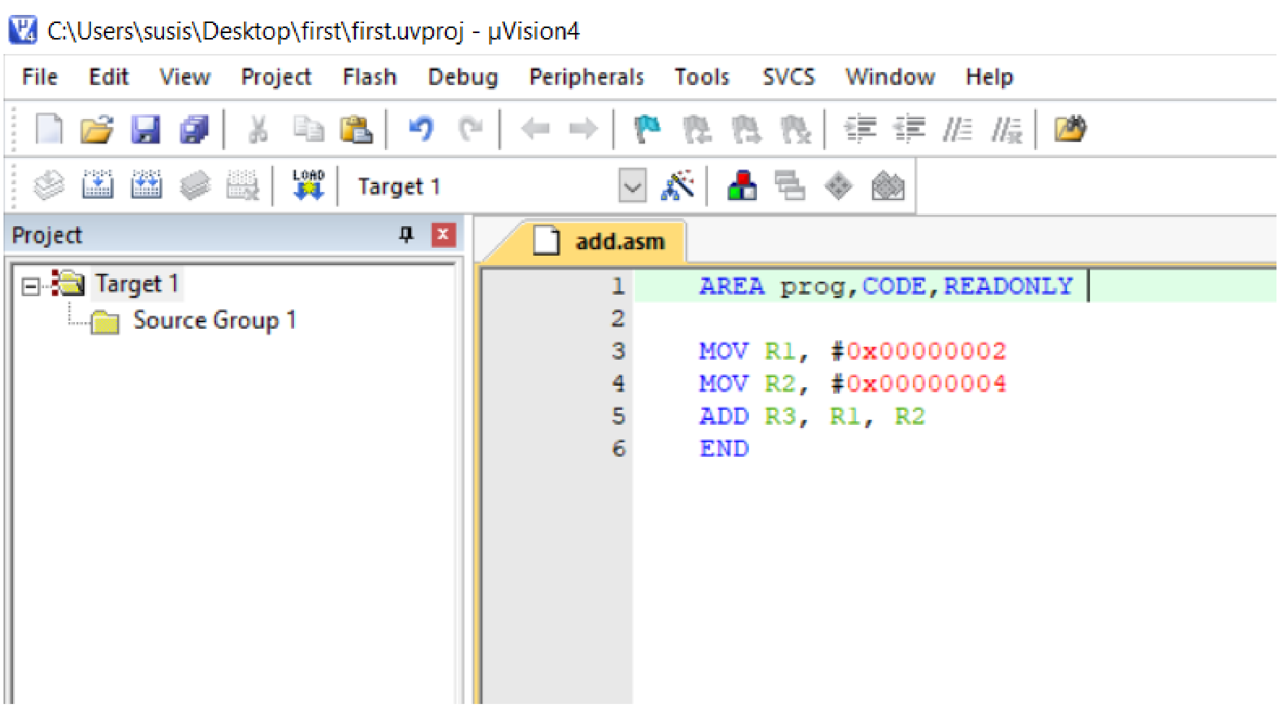


Save it and give an extension of .asm

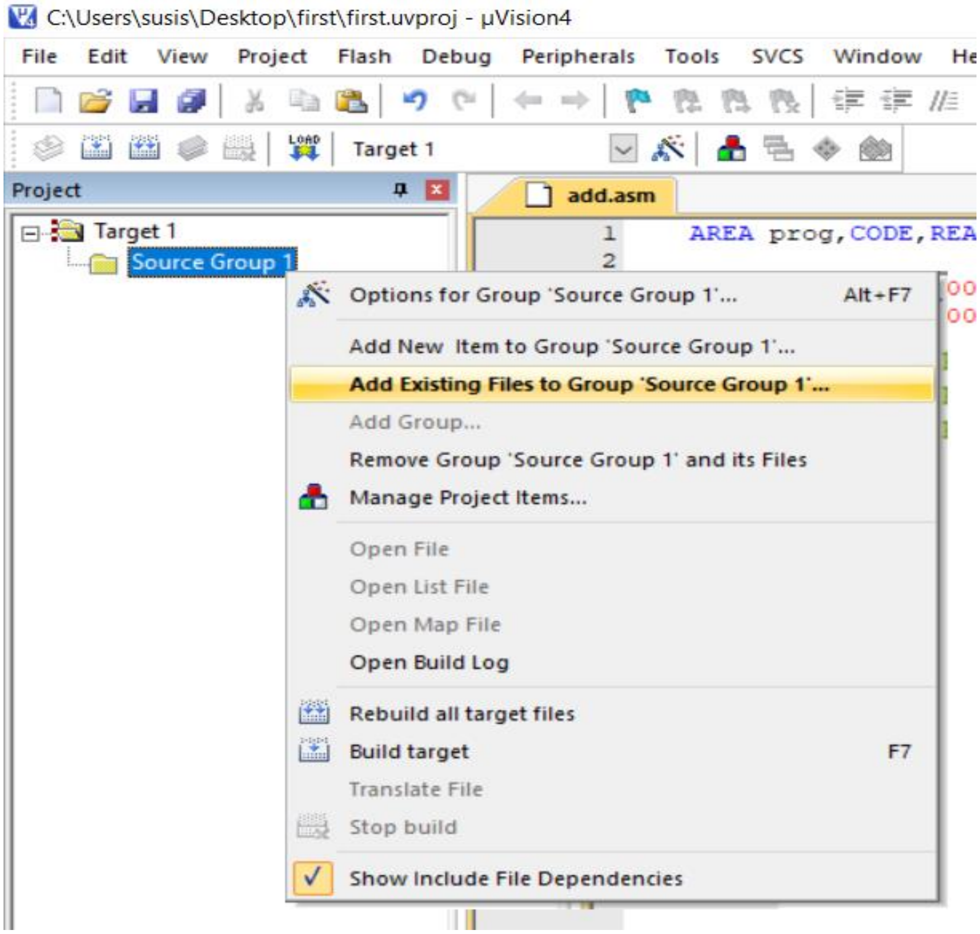




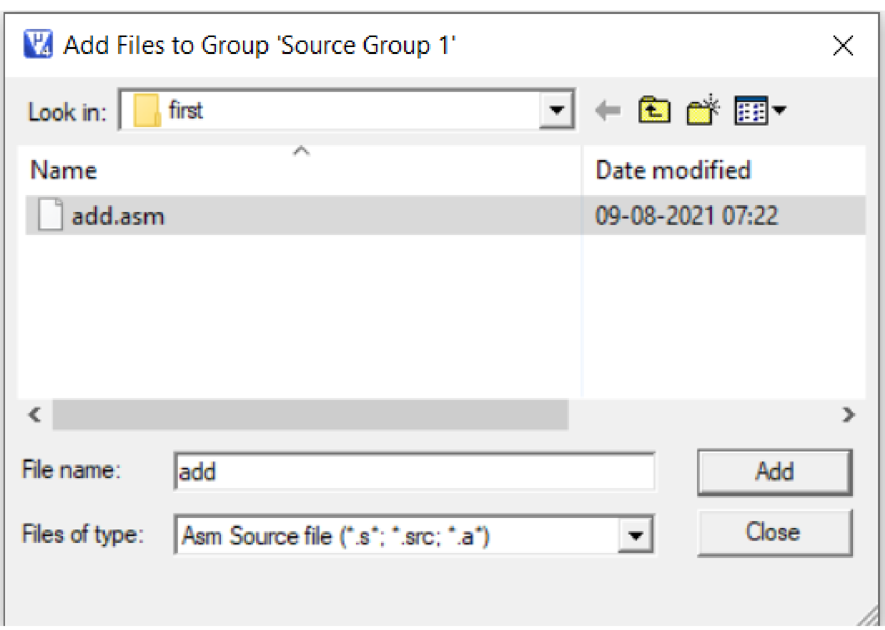
Type the following code



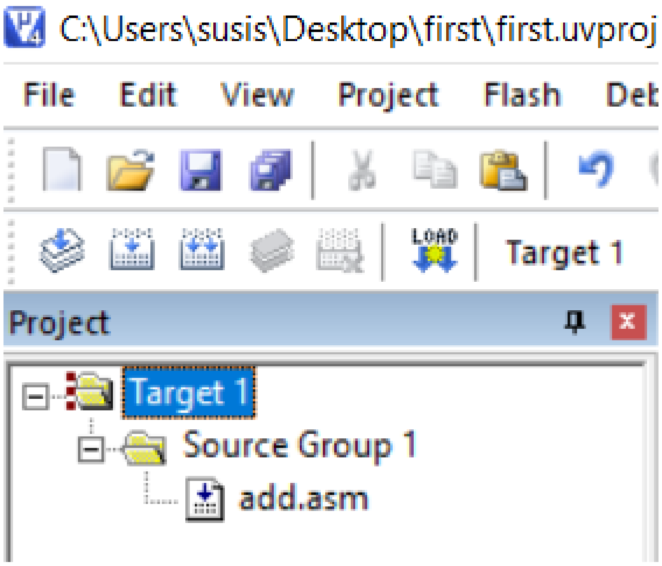
Then add the created .asm file to Source Group 1 by following the below steps



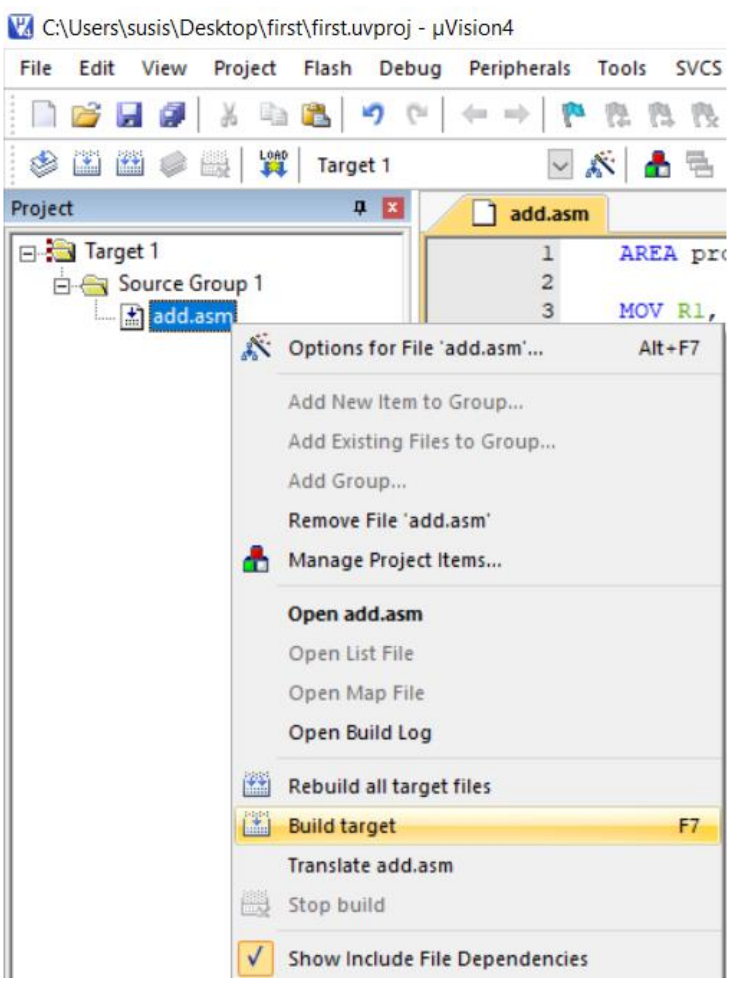
Locate the file and add the .asm file (Save it as add.s)



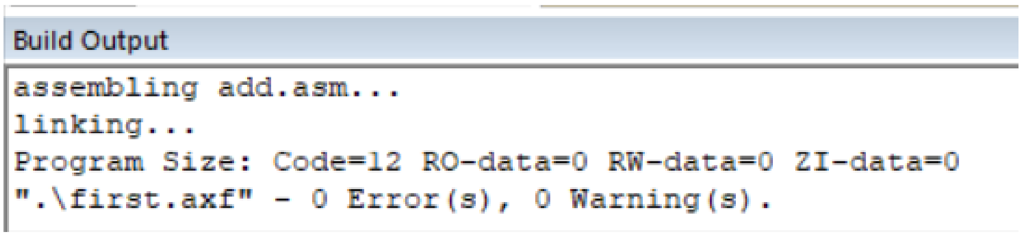
Directory Structure



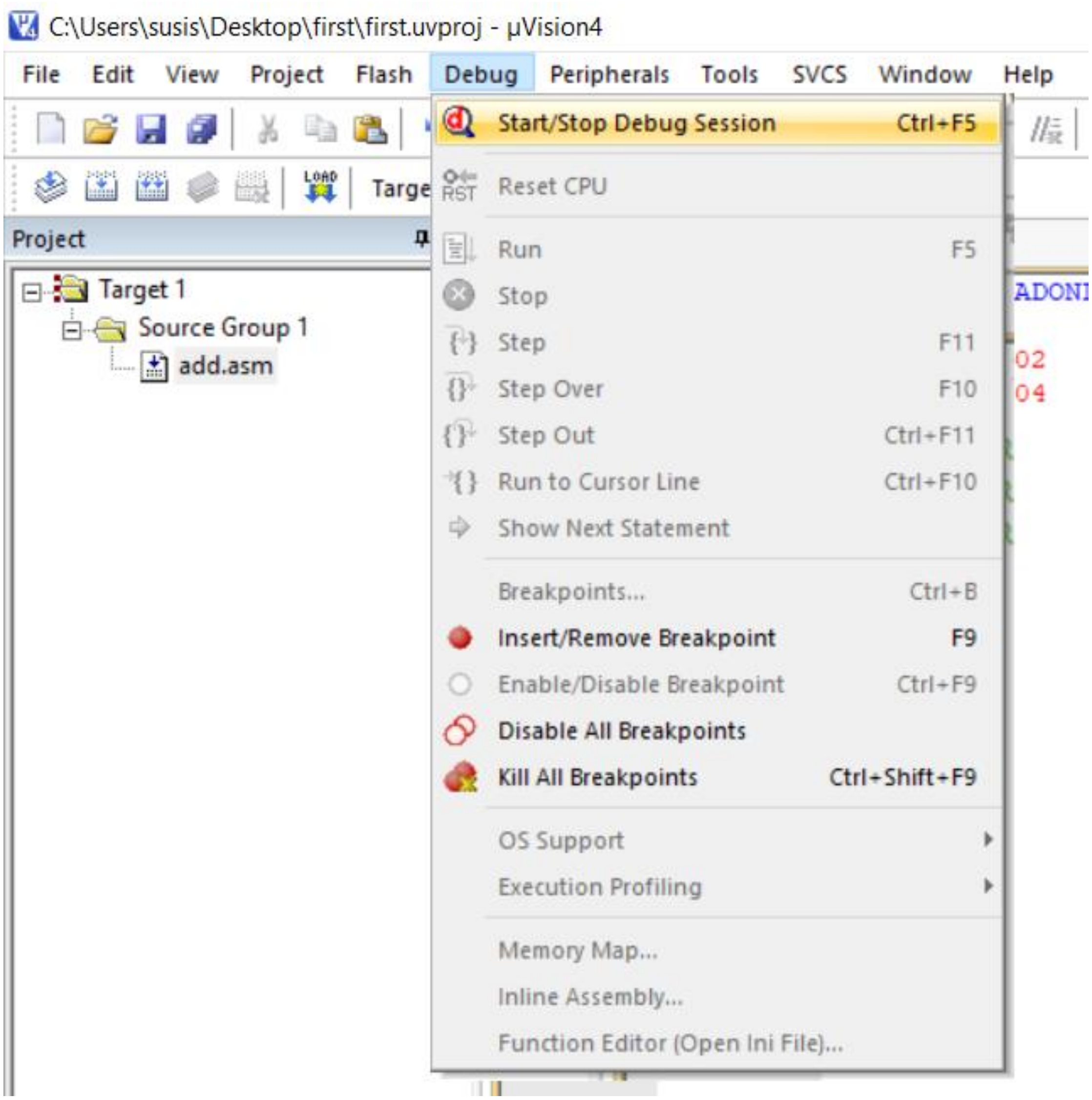
Build target

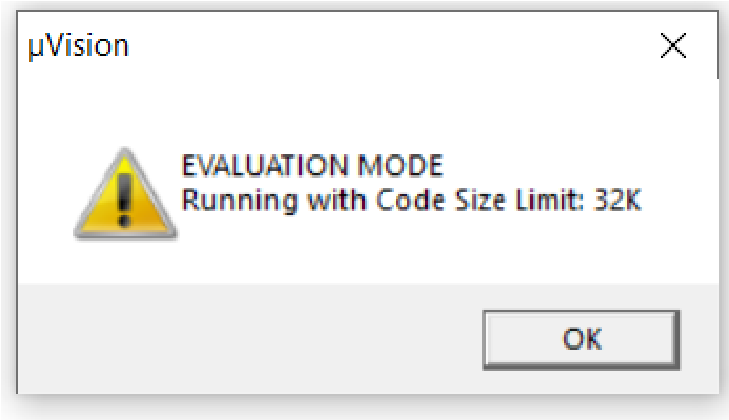


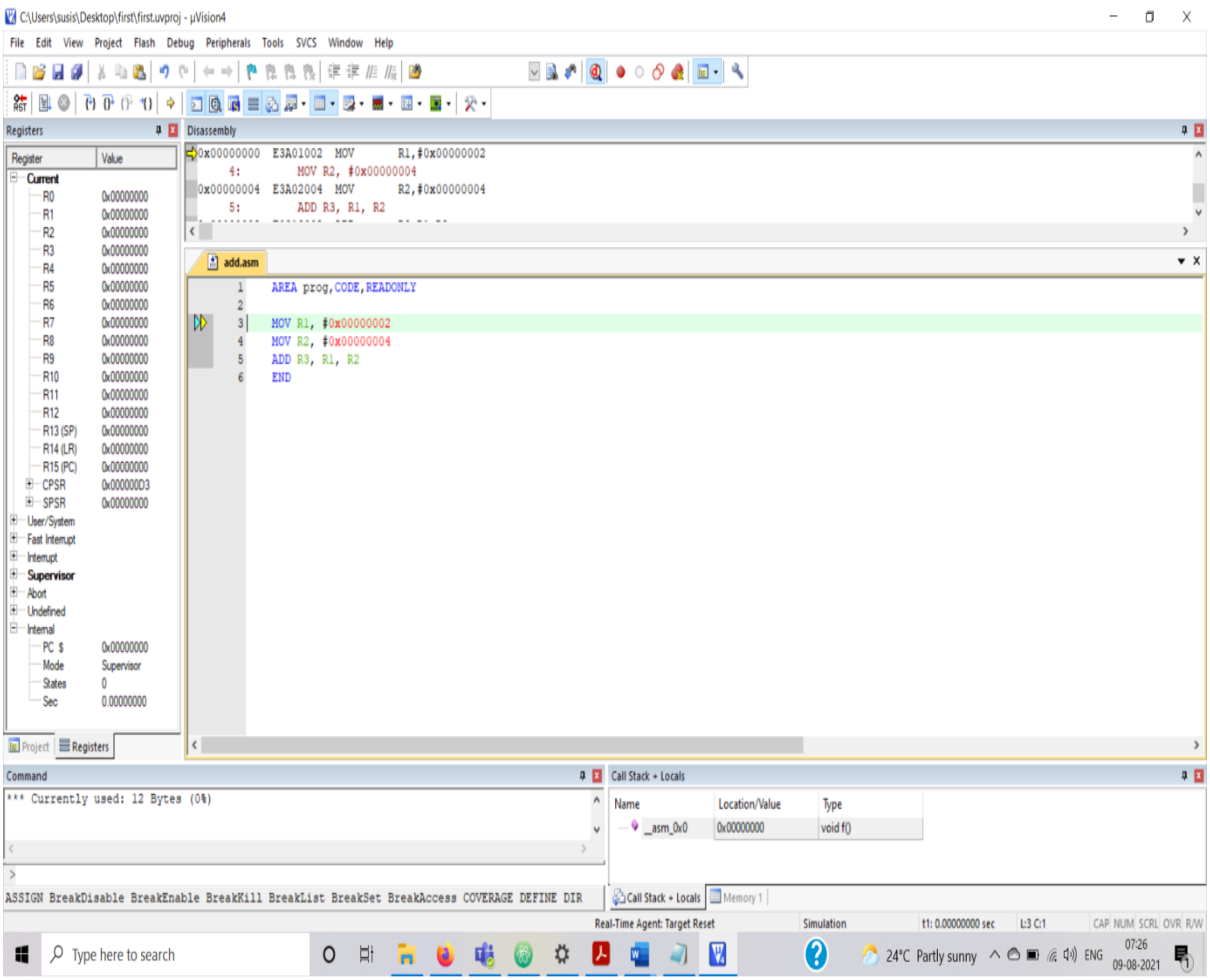
No errors and warnings

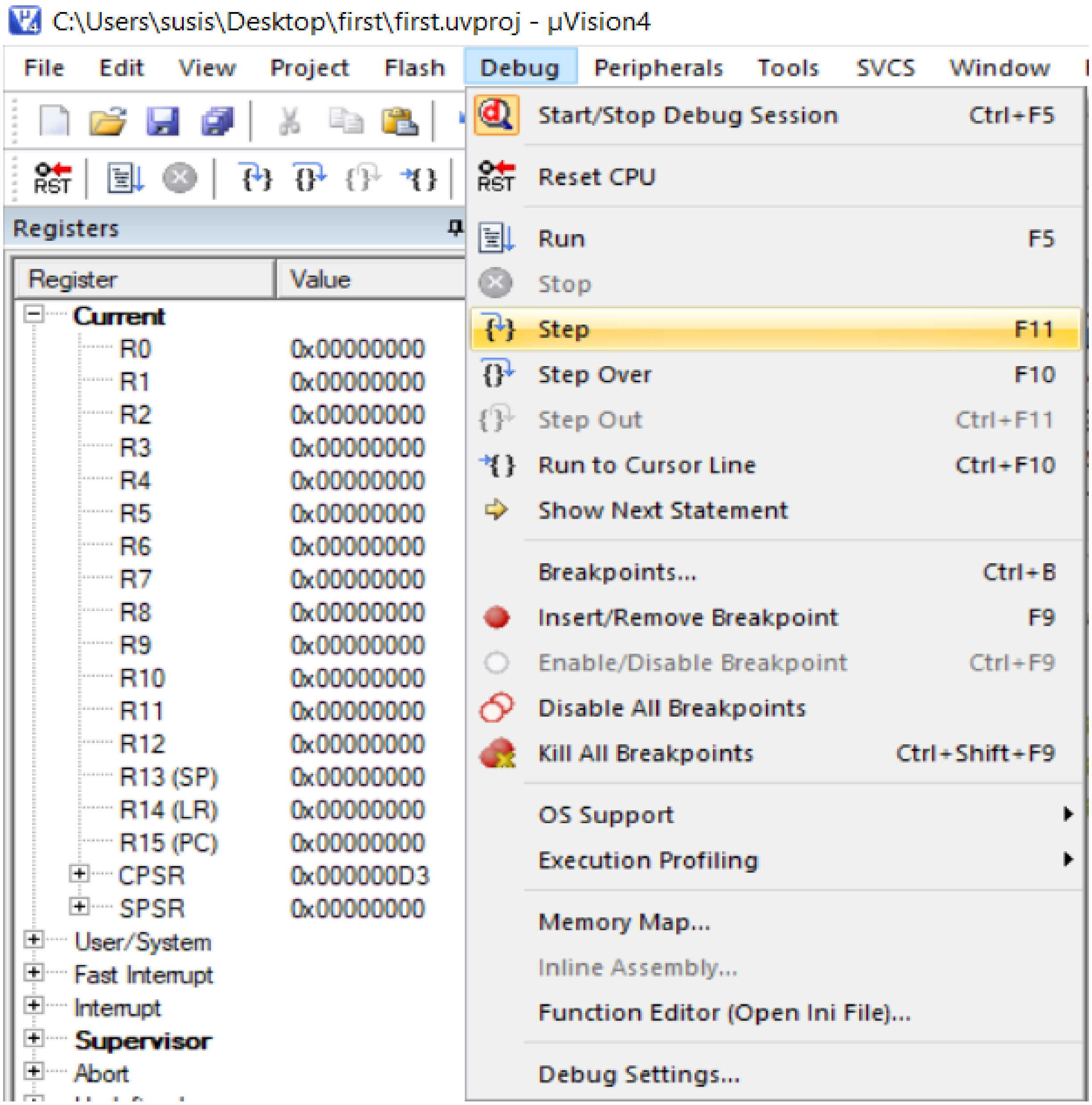


Then Start/Stop Debug Session to begin execution

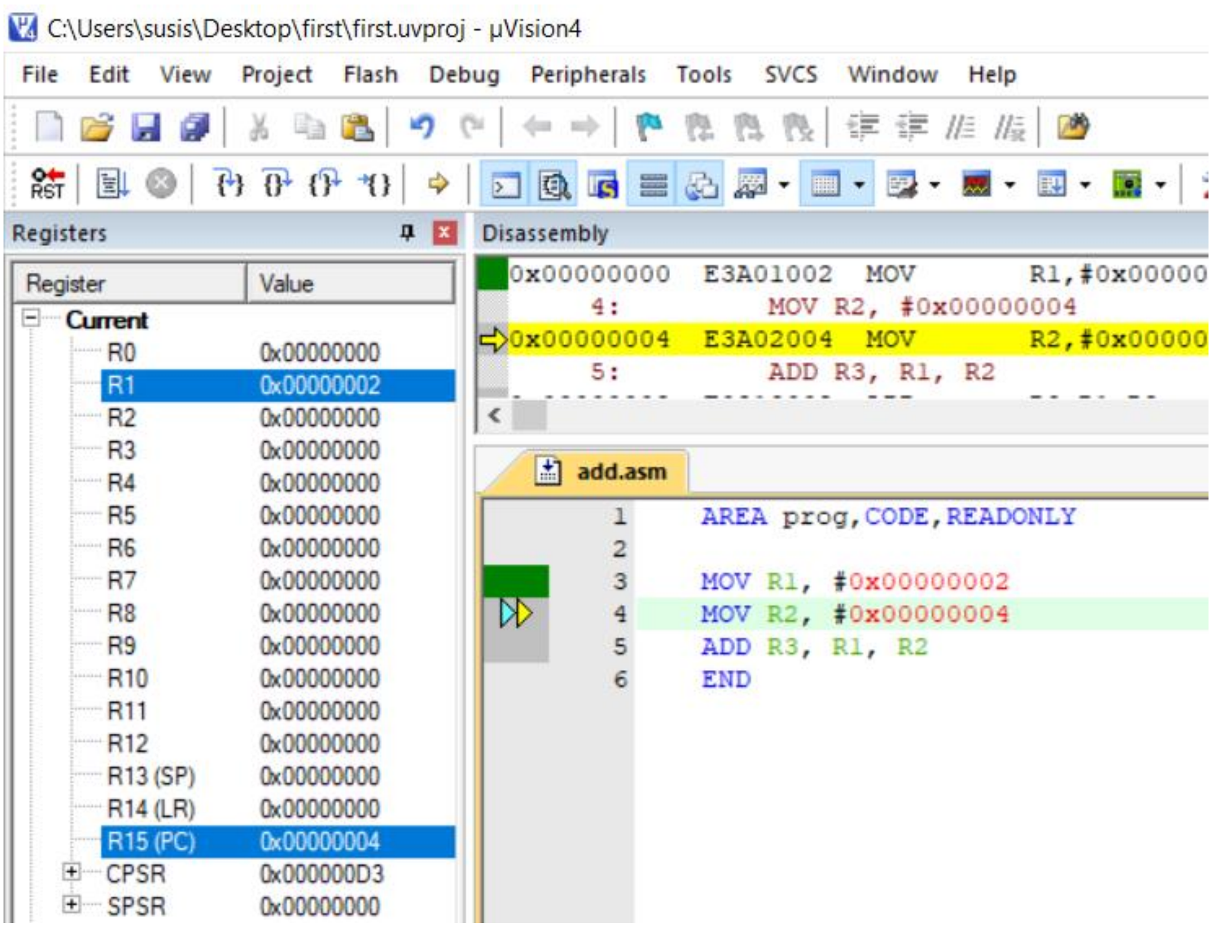



µVision
EVALUATION MODE
Running with Code Size Limit: 32K
OK

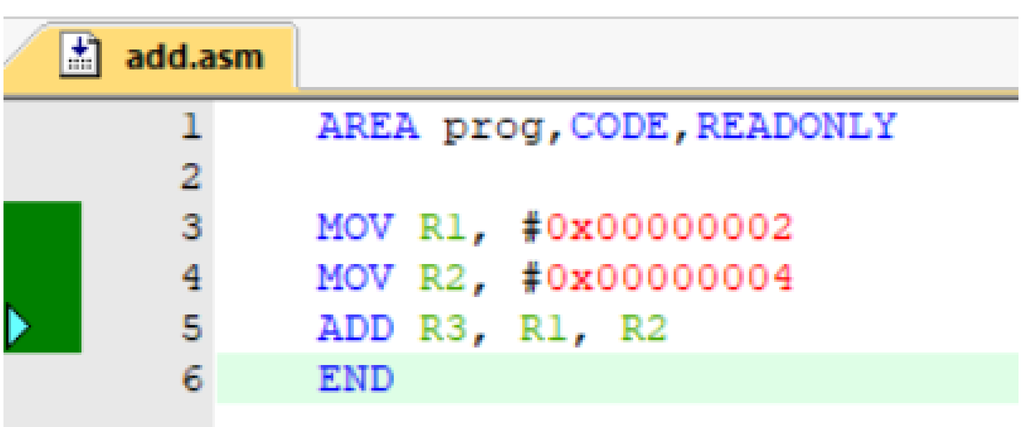

C:\Users\susis\Desktop\first\first.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Registers
Disassembly
0x00000000 E3A01002 MOV R1,#0x00000002
4: MOV R2, #0x00000004
0x00000004 E3A02004 MOV R2,#0x00000004
5: ADD R3, R1, R2
add.asm
AREA prog,CODE,READONLY
MOV R1, #0x00000002
MOV R2, #0x00000004
ADD R3, R1, R2
END
Command
*** Currently used: 12 Bytes (0%)
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE DEFINE DIR
Call Stack + Locals
Memory 1
Simulation
Type here to search
24°C Partly sunny
ENG
07:26
09-08-2021


C:\Users\susis\Desktop\first\first.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window
Registers
Register
Value
Current
R0 0x00000000
R1 0x00000000
R2 0x00000000
R3 0x00000000
R4 0x00000000
R5 0x00000000
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x00000000
R14 (LR) 0x00000000
R15 (PC) 0x00000000
CPSR 0x000000D3
SPSR 0x00000000
User/System
Fast Interrupt
Interrupt
Supervisor
Abort
Start/Stop Debug Session Ctrl+F5
Reset CPU
Run F5
Stop
Step F11
Step Over F10
Step Out Ctrl+F11
Run to Cursor Line Ctrl+F10
Show Next Statement
Breakpoints... Ctrl+B
Insert/Remove Breakpoint F9
Enable/Disable Breakpoint Ctrl+F9
Disable All Breakpoints
Kill All Breakpoints Ctrl+Shift+F9
OS Support
Execution Profiling
Memory Map...
Inline Assembly...
Function Editor (Open Ini File)...
Debug Settings...

It can be seen that R1 is updated with value 2 after execution MOV R1, #0x00000002



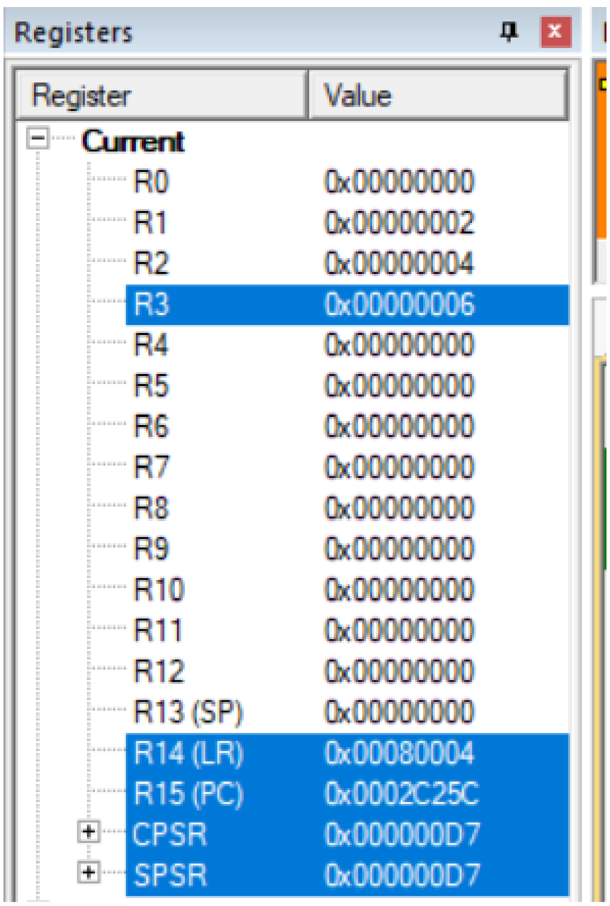
Complete the execution of all the instructions



R1 – 2

R2 – 4

R3 - 6


Registers
Register
Value
Current
R0
0x00000000
R1
0x00000002
R2
0x00000004
R3
0x00000006
R4
0x00000000
R5
0x00000000
R6
0x00000000
R7
0x00000000
R8
0x00000000
R9
0x00000000
R10
0x00000000
R11
0x00000000
R12
0x00000000
R13 (SP)
0x00000000
R14 (LR)
0x00080004
R15 (PC)
0x0002C25C
CPSR
0x000000D7
SPSR
0x000000D7

Lab Sheet 7:

Familiarization of Keil uVision 4 IDE

with simple Embedded C and ARM Assembly code

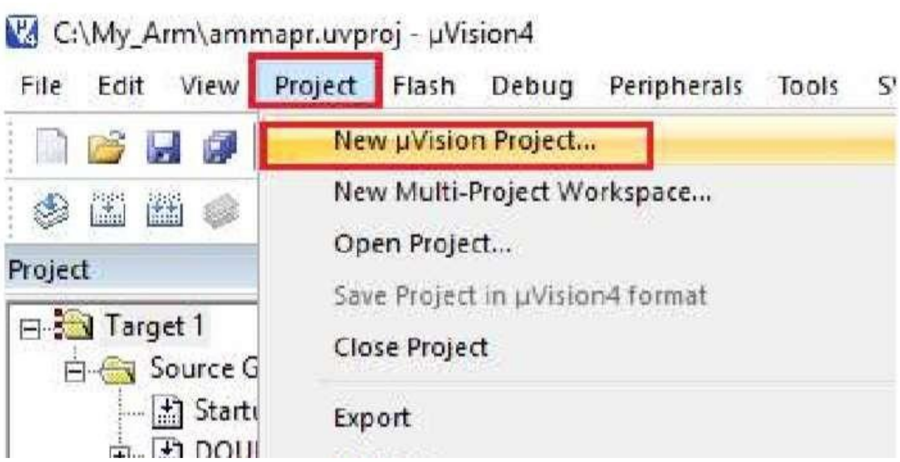
Name: Roll Number :

Keil IDE: Keil Electronics, provides an Integrated Development Environment called Keil µVision (pronounced Micro-Vision) that integrates project manager, editor, compiler, debugger and simulator in a single powerful environment; it provides a high efficiency and clear graphic user interface for embedded software development.

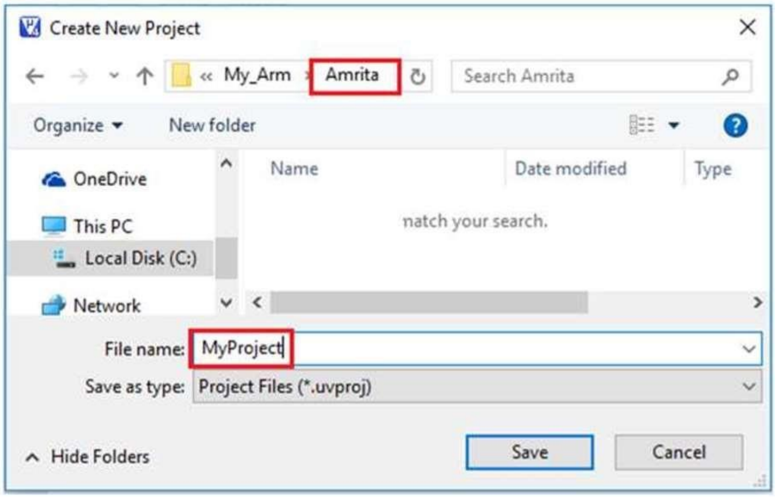
I. Create your first Project: The objective of this lab sheet is to give you an introduction to the world of embedded programming using C and ARM assembly language; here we learn how to use the ARM Keil uVision IDE to create projects, build and debug them.

Step 1: Open the Keil IDE by clicking on its icon on the desktop.

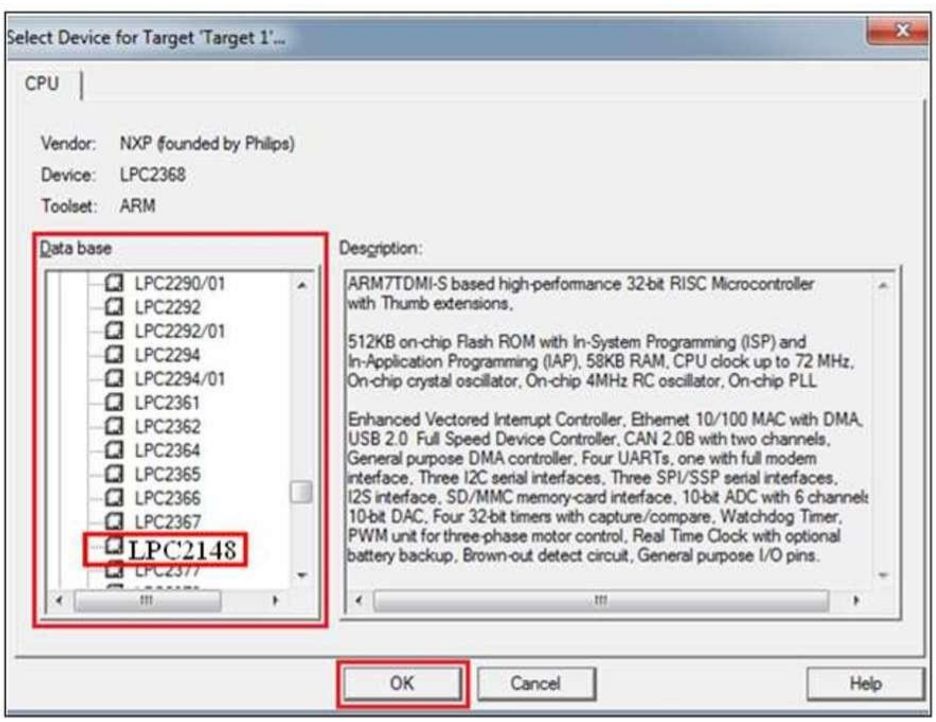
Step 2: Choose **new uVision Project** from the **Project** menu



Step 3: Create a new folder and name it as Amrita; type the name MyProject for the project and click Save.



Step 4: In the Data base tree as shown below, choose the vendor and then the chip you want to use and click OK. As want to use LPC2148, click on the **NXP** then on the **LPC214**8 and press **OK**. (Or in the search box , type LPC2148 directly and click OK)

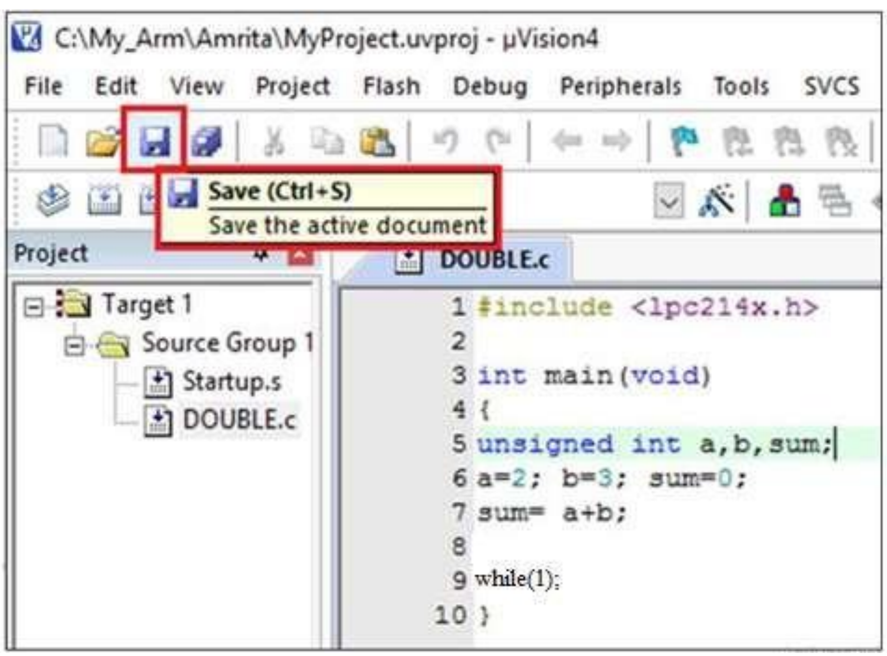


Step 5: A message window will be appeared and asking you that whether to add Startup.s file to the project or not; as we are going to start with C code, click on 'Yes' button to add

startup file. Note that if you want to write Assembly program then your answer should be 'No' (skip adding Startup.s).

Step 6: Create a new file by choosing **New** from the **File** menu; or you may press Ctrl+N, as well.

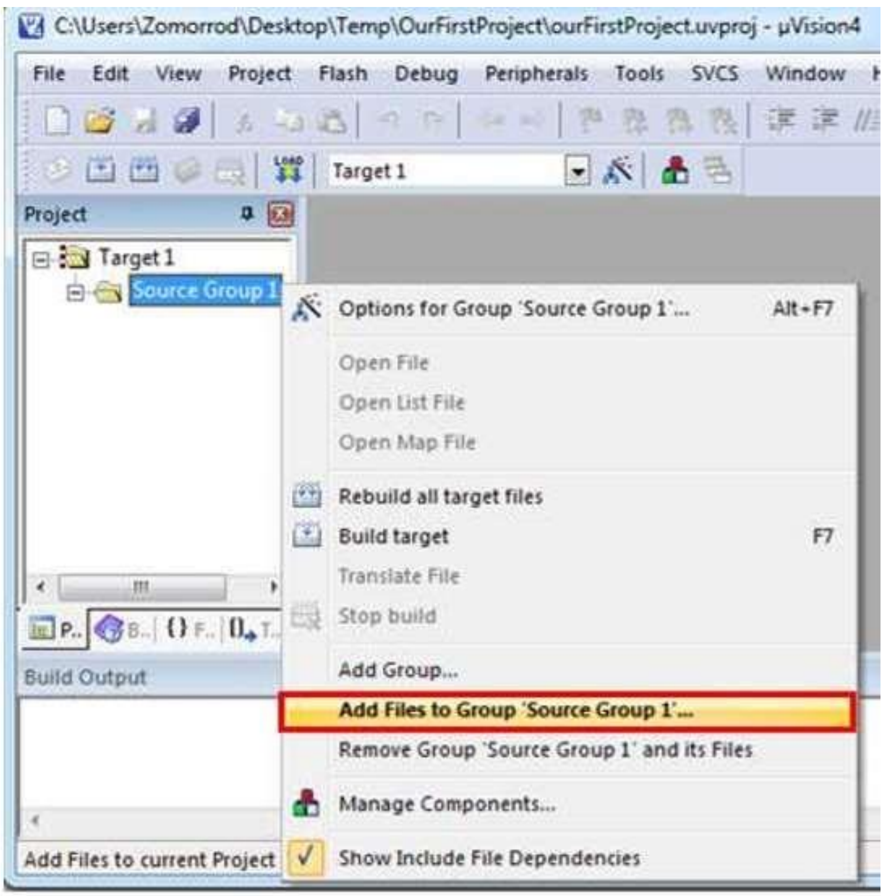
Step 7: Type your first C code in the work space provided as shown in the following figure

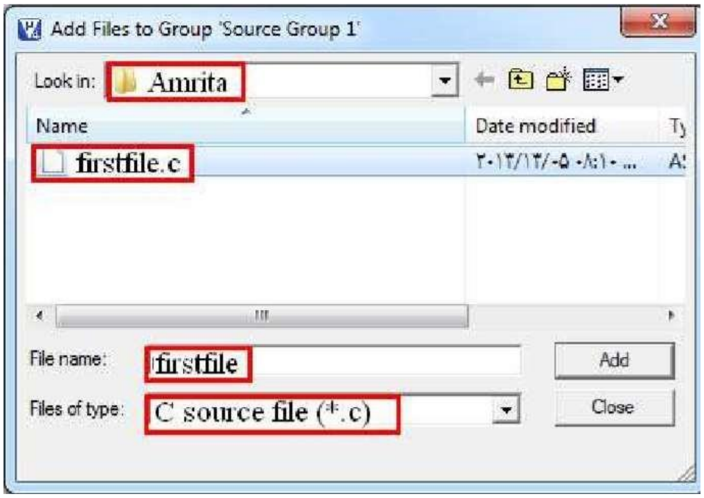


Note that in the text editor window, keywords appear in a different colour; it is to differentiate keywords from variables, constants and others,

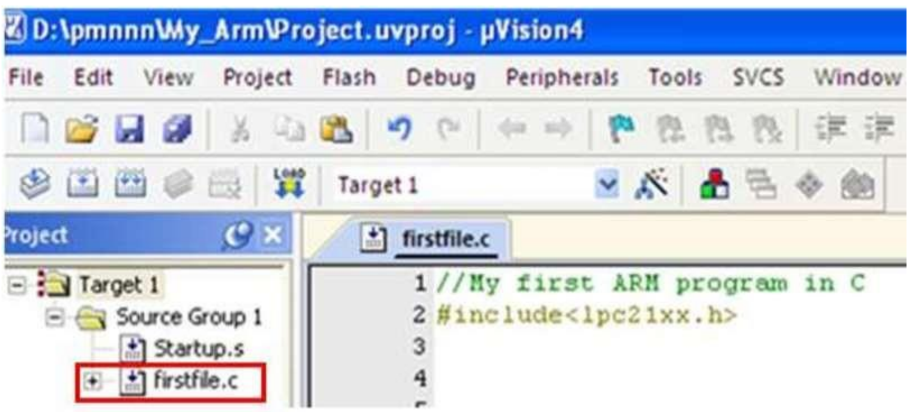
Step 8: Once the coding is completed, press save icon on the file menu (or press Ctrl+S). Name the file as FirstFile.c and save it as a C file in your personal folder Amrita.

Step 9: Add your FirstFile.c to the project. To do so: Expand **Target1** on the left side panel; Right click on the **Source Group1** and choose **Add Existing Files to source Group**; then browse the directory Amrita and choose desired file: FirstFile.c. Finally click on **Add** button and **Close**.

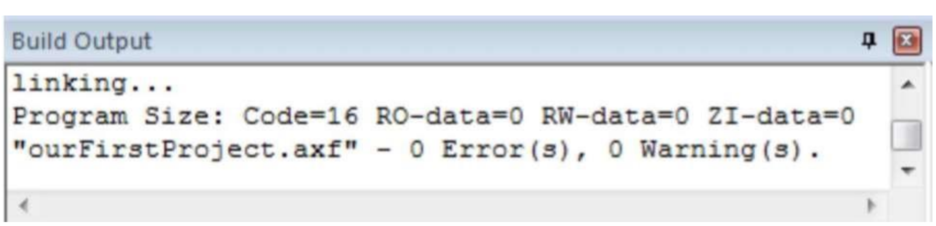




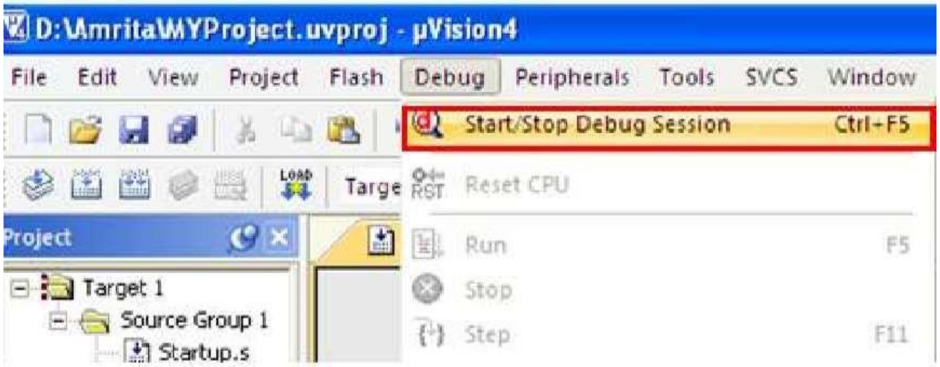
You should make sure that your file is added under the project by expanding the **Source Group** shown under Target (Note that firstfile.c is added under Source Group 1in the left panel)



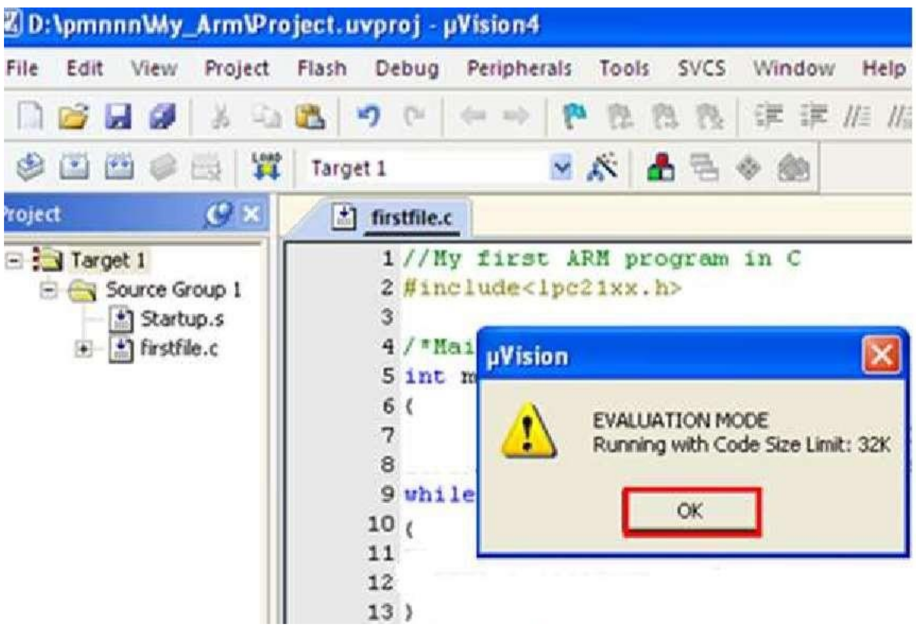
II. Building the Project: To build your code, click on the Build icon or choose **build target** from the **Projec**t menu (Project→Build Target). If the program is built successfully a built output window appears with 0 Error(s) and 0 Warning(s) as shown in the following figure. If the built is unsuccessful (i,e, if the built output window is provided with any error – syntax error), you have to re-edit the source code and build it again until the built become successful.



III. Debugging the Project: Now you need to use a debugger to see what is happening inside your program. To start debugging click on Start/Stop Debug Session icon or choose Start/Stop Debug Session from the Debug menu (or use Ctrl+F5).

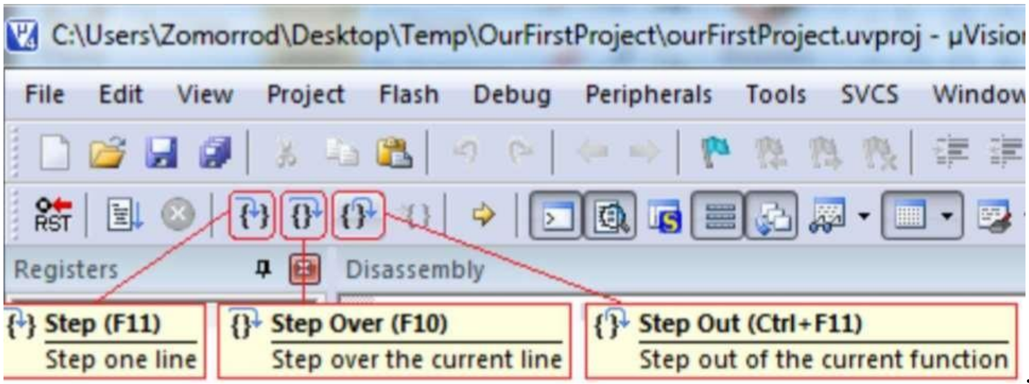


When you select Start Debug Session, a warning that this is an evaluation version shows up; proceed with OK. Then, a new window with several different supporting panels appears where you can see the simulation of the program. The left hand side panel is called Register window.



III. Running the Project: To run your project, select Debug- > Run, or click the Run button on the toolbar; or else you may press F5 key to run. The program runs until the simulator encounters a breakpoint. To stop running the project, again select Debug -> Start/Stop Debug Session, or else click the Stop button on the toolbar

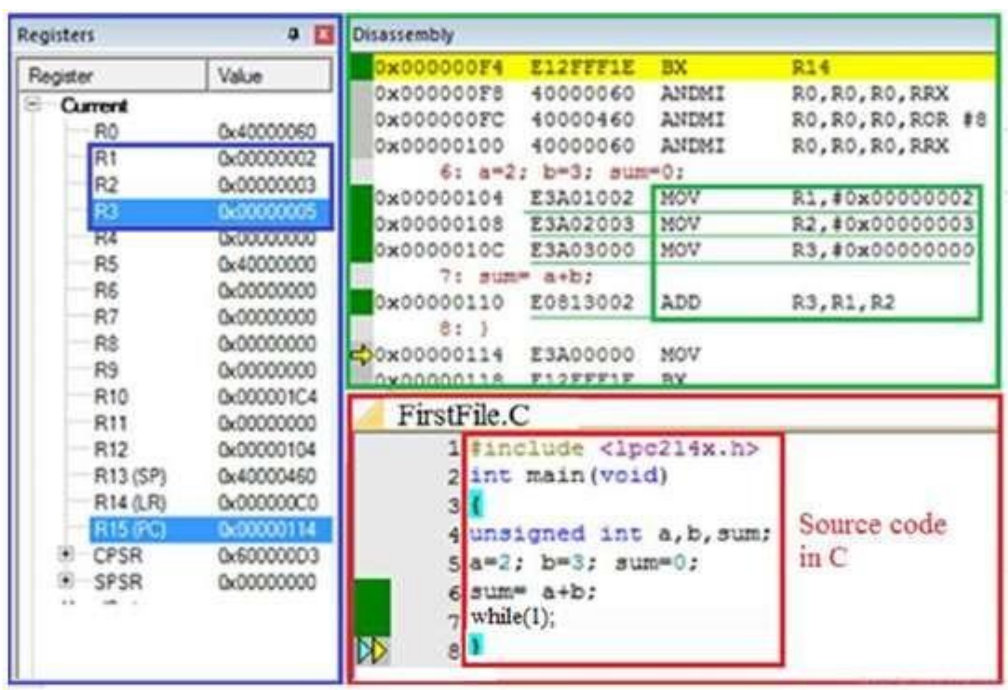
To trace your program you are provided with Step Over button (or click on Step Over from the Debug menu); using this tool one may execute his code line by line (one instruction after another); during trace you may verify or modify register content.



To exit from the debug/run mode press Start/Stop Debug Session. again

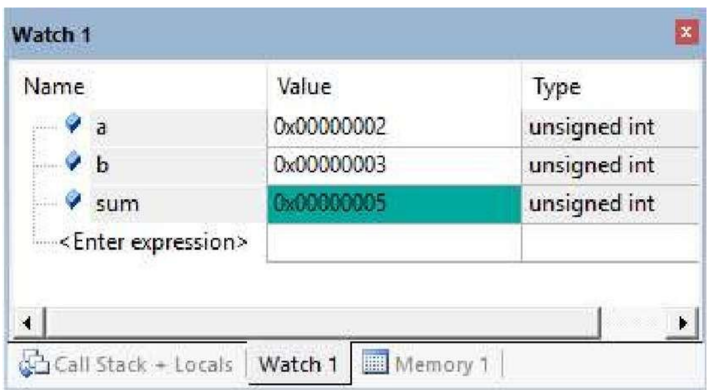
Debugging windows: There are some special windows provided by the Keil IDE, to help you to analyse the code and result through the Registers, Stack, Memory and Ports.

Disassembly window: In the **following** figure, disassembly window is walled in green which shows assembly instructions equivalent to your C code. Let's go through the assembly code line by line; in fact, by reading an assembly code, one can realize that what the processor is doing exactly.



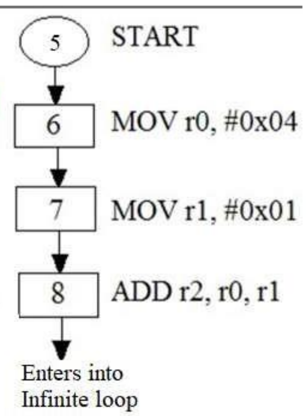
Register Window: The left most panel (in the following figure, it is walled in blue) is called register window that shows 16 general-purpose registers along with Current Program Status Register (CPSR) and SPSR; this window allows you to modify their contents. A register can hold a 32-bit signed, unsigned integer or a memory pointer. All C variables map onto the registers and large data structures like arrays will be in a memory.Register Window is a very handy tool for debugging. Go through required registers and verify their contents.

Watch Window: Go through View-> Watch Windows-> Watch-1; this window allows you to view or monitor the values of the variables or registers which are used in the program during run time (while the program is running); and also you may modify these values without disturbing or terminating the execution. To view the value of any variable that the variable should be added through the <Enter expression> field of the watch window.



Exercise: Execute the following ARM Assembly code using Keil IDE; while typing the code proper indentation must be provided as shown in the figure.

```
; Demo Program - Done at Amrita

      AREA MyCode, CODE, READONLY

START;Begining of code
      MOV r0, #0x04  ;r0 <-- 0x04
      MOV r1, #0x01  ;r1 <-- 0x01
      ADD r2, r0, r1 ;r2 <-- r0 +r1
EXIT  B   EXIT       ;
      END            ;End of code
```



Signature of the faculty with date: ………………………………

Exercise 1: AC

```
;Demo Code - Done at Amrita
;Find biggest among 3 numbers
;a=r1;  b=r2; c==r3; big=r0
;During run load r1, r2 and r3 with values

    AREA    MyCode, CODE, READONLY

START                   ;Begining of code
CheckAB CMP r1,r2       ;if (a>b)
        BLE CheckBC
CheckAC CMP r1,r3       ;     if (a>c)
        BLE CtoBig
        MOV r0,r1       ;     a is big
        B   EXIT

                        ;Else
CheckBC CMP r2,r3       ;      if (b>c)
        BLE CtoBig      ;
        MOV r0,r2       ;      b is big
        B   EXIT
CtoBig  MOV r0,r3       ;

EXIT    B   EXIT
        END             ;End of code
```

Exercise 2: Design a simple calculator with four basic operations such as, addition, subtraction, multiplication and logical AND

```
; Demo Program - Done at Amrita
; Mutiple slection - Switch
; Simple Calculator

        AREA MyCode, CODE, READONLY
START                   ;Begining of code
        CMP R2, #0      ;Choice 1
        BEQ case0
        CMP R2, #1      ;Choice 2
        BEQ case1
        CMP R2, #2      ;Choice 3
        BEQ case2
default                 ;default choice
        AND r0, r1      ;and operation
        B break
case0   ADD r0, r1      ;Addition
        B break
case1   SUB r0, r1      ;Subtraction
        B break
case2   MUL r0, r1, r0 ;Multiplication
break   B   START       ;Go to START
        END             ;End of code
```

tExercise 3: Using the above template write an ARM assembly code to calculate the factorial of a number; let us calculate 5!

```
;Demo Code - Done at Amrita
;Using For Loop
;Computing 5!

      AREA    MyCode, CODE, READONLY

START                       ;Begining of code
      MOV r0,#0x01          ;fact=1
      MOV R1,#0x01          ;i=1: Initialization
      B   CHECK             ;Branch to CHECK
LOOP  ;Body of the loop
      MUL r0, r1, r0        ;fact=fact*i
      ;.................
      ADD  R1,R1,#0x01      ;i++: Increment
CHECK CMP  R1,#0x05         ;Condition check
      BLE  LOOP             ;if(i<=5) then looop

      END                   ;End of code
```

Exercise 4: Convert the following C code into equivalent ARM assembly code; use conditional execution feature.

```
;Demo Code - Done at Amrita
;Conditional Execution

        AREA Mycode, CODE, READONLY

ENTRY ;Begining of code

  CMP   r1,r2     ;Compare r1 and r2
  SUBGT r3,r1,r2  ;If r1>r2 then r3=r1-r2
  ADDEQ r3,r1,r2  ;If equal then r3=r1+r2
  MULLT r3,r1,r2  ;If less then r3=r1*r2
  END             ;End of code

```

Lab Sheet 9:

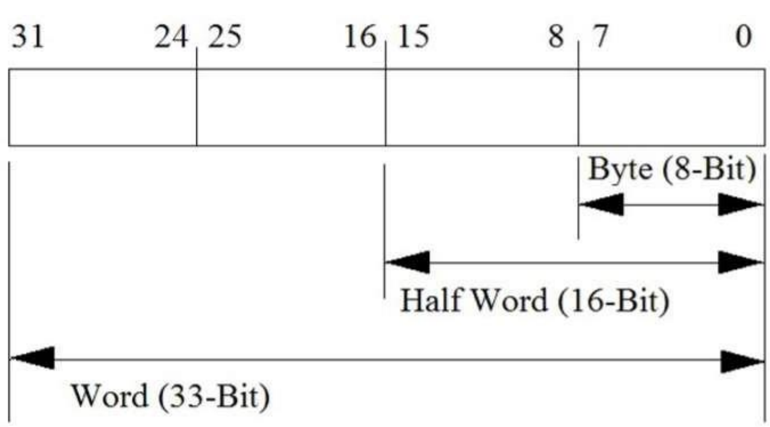
Understanding ARM7 Instruction Set and
Writing Programs in ARM Assembly Language

Name: Roll Number :

ARM Assembly Programming: The ARM7TDMI-S processor has two operating states such as ARM state and THUMB state; with two different formats of instruction sets: 32-Bit long ARM instruction set and 16-Bit THUMB instruction set. ARM instructions are executed in the ARM state whereas Thumb instructions are processed in the THUMB state. Both Instruction sets supports different categories of instructions: Branching instructions, Data processing instructions, Register data transfer instructions, Load and Store instructions and Coprocessor instructions. ARM assembly language instructions are case insensitive; with a basic understanding of the instructions, one may write programs.

The ARM processor supports six data types; obviously a 32-Bit word is a basic data type; but you may use 8-Bit and 16-Bits words as well.

- 8-bit signed and unsigned bytes.
- 16-bit signed and unsigned half-words; these are aligned on 2-byte boundaries.
- 32-bit signed and unsigned words; these are aligned on 4-byte boundaries.



An assembly program may comprise assembler directives and Assembly instruction; The assembler directives are mean to guide the assembler during translation process; but the assembly instructions are actually translated into the machine language. The following are some of commonly used assembler directives or pseudo instructions.

1. EQU: The equate directive associates a symbolic name with a constant (an address or a data). It does not store anything in a memory; but the assembler maintains the names and their associated values in a symbol table. Generally an EQU directive is used in the beginning of program that is identical to the #define in C. Example: TEMP EQU 32 SUM EQU 500
2. INCLUDE: Includes the content of a specified file in the current file
3. AREA: Defines a storage area for code or data chunk with name; it also defines that the type of memory to be used, for example codes are stored at READONLY areas located in a Read Only Memory.
4. CODE: Specifies that the Area to be used for code (read only)
5. DATA: Specifies that the Area to be used for data (read write)
6. ENTRY: This directive specifies the program entry point
7. ADR: Loads an address into a register.
8. Define constant directives (Data Storage Directives): These directives allocates one or more word or half-word or bytes of memory with their initial runtime values. In fact the storage for space for data is allocated in a program memory. For example, VALUE1 DCD 14, it puts 14 into the next available program memory location and the location is named VALUE1.

   DCD (Define Constant Data): Allocates a 32-Bit storage

   DCW: Allocates a half-word storage (for 16-Bit data)

   DCB: Allocates a byte storage. Note that an ALIGN directive should be used to ensure that the data is byte aligned. Default ARM assembler strings are not null terminated. One may define a null terminated string as shown in the following code.
9. END: Indicates an end of source code; any code found after an END directive is ignored by an Assembler.
10. IMPORT: Imports an external routine for use in a current routine
11. EXPORT: Export current routine to use in other routine (refer page 182 – importing and exporting procedure)

The following code computes the 2's Complement of a number; here we use, load register LDR R0, X1 instruction to load register R0 with the contents of memory location X1. Note that the LDR is a pseudo instruction.

```
; Demo code: 2s complement of a number
; Done at Amrita

        AREA    Demo, CODE, READONLY
START
        LDR r0, x1        ; Load data from location x1 into r0
        MVN r1, r0        ; Move not of r0 to r1
        ADD r2, r1, #1    ; r2 = r1 + 1
        STR r2, x2        ; Store result at location x2
LOOP    B LOOP
;DCD (Define constant data) Allocates a 32-Bit storage
x1      DCD 0xFFFFFFFE
x2      DCD 0
        END
```

Control Structures: There are three basic types of control structures may be implemented. By default, execution of a program is sequential.

- Sequential control
- Decision making and selection control
- Loop control

Decision making and selection

Unconditional Jump

```
        B       Label       ; Unconditional jump to Label
        MOV     r1, #6      ; This line never executes
Label   MOV     r1, #5      ; Jumps here, this moves 5 to r1
```

Conditional Jump - Executing a loop five times

```
        MOV     r1, #5      ; Initialize loop counter r0 with 5
Loop    SUBS    r1, #1      ; Decrease counter and change flags accordingly
        BNE     Loop        ; if (counter! = 0) then branch to loop else exit loop
```

Example 1: The following code checks whether the given input value (in register r2) is odd or even; if the value is even then it moves E to register r0 else moves 0. Note that, if you replace AND with ANDS, then the result of the operation may create an impact on the Status Register (CPSR); so that the BNE instruction works well without CMP.

```
;Demo Code - Done at Amrita
;Control Structure - Simple If...Else
;If input number is even then
;move E to r0 else move 0

      AREA    MyCode, CODE, READONLY

START                   ;Begining of code
      MOV  r2, #0x04    ;Input number
      MOV  r1, #0x01    ;Mask value
      AND  r1, r1, r2   ;r1=r1 and r2
TEST  CMP  r1, #0x00    ;If (r1==0)
      BNE  ODD          ;
      MOV  r0, #0xE     ;Then move E and exit
      B    EXIT
ODD   MOV  r0, #0x0     ;Else move 0 and exit
EXIT  B    EXIT         ;
      END               ;End of code
```

Exercise 1: Find the biggest among the three numbers a, b and c which are, to be loaded in registers r0, r1 and r2; copy the biggest one in register r3.

Example 2: Assume that an input value is loaded into register r1; if it is 1 then load register r2 with 0x0A; else if it is 2 then load 0x0B in register r2; else if it is 3 then load 0x0C into register r2; else load 0x0D (default value to be loaded in the register)

```
IF  (R1==1)  THEN R2=0X0A
ELSE IF (R1==2)  THEN R2=0X0B
ELSE IF ( R1==3) THEN R2=0X0C
ELSE R2=0X0D   (Default Case)
```

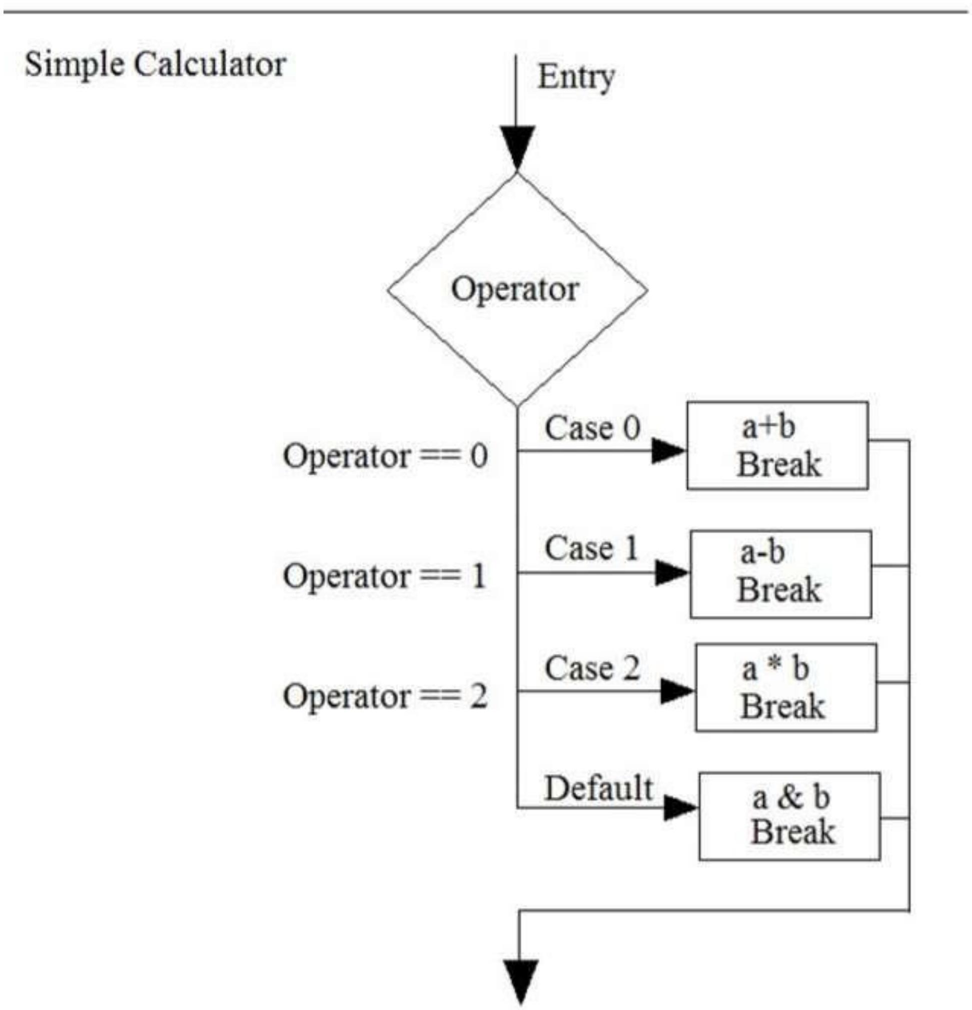
The following code implements the above operation

```
;Demo Code - Done at Amrita
;Else-If ladder
;Input = register r2
;Output = register r1

        AREA    MyCode, CODE, READONLY

START                   ;Begining of code
        MOV R2,#0x03    ;i
TEST1   CMP R2,#0x01    ;if (i==1)
        BNE TEST2
        MOV R1,#0x0A    ;then out=A
        B   EXIT
TEST2   CMP R2,#0x02    ;Else if(i==2)
        BNE TEST3
        MOV R1,#0x0B    ;then out=B
        B   EXIT
TEST3   CMP R2,#0x03    ;Else if(i=3)
        BNE DEFAULT
        MOV R1,#0x0C    ;then out=C
        B   EXIT
DEFAULT MOV R1,#0x0D
EXIT    B EX
        END
```

Exercise 2: Design a simple calculator with four basic operations such as, addition, subtraction, multiplication and logical AND

Opcode	Mnemonic	Operation
0	ADD	Addition
1	SUB	Subtraction
2	MUL	Multiplication
3	AND	Logical And

Assume that the operation code is loaded in register r2; and the operands are loaded in register r0 and r1. The operation code is compared with three integer values (specified by three cases: case0, case1 and case2), if there is a match, depends on the value being matched, it performs an appropriate operation on the operands and loads the result in r0 itself. For example, if the operation code is 2 and matched with case2 then the product of r0 and r1 is calculated and stored in r0. If no case matches, the default logical and operation is performed.



Loop control.: We will discuss how loop structures are implemented in ARM assembly language

Example 3: The following template executes an empty body six times; the loop control variable is assumed to be in register r1, which is initialized with zero and incremented up to 5; i.e. every time value in r1 is compared with 0x05, if it is less than or equal to 5 (i<=5), then it loops otherwise exits.

```
;Demo Code - Done at Amrita
;Implementation of For Loop
;for (i=0; i<=5; i++)

      AREA    MyCode, CODE, READONLY

START                       ;Begining of code

      MOV R1,#0x00          ;i=0: Initialization
      B   CHECK             ;Branch to CHECK
LOOP  ;................
      ;Body of the loop
      ;................
      ADD  R1,R1,#0x01      ;i++: Increment
CHECK CMP  R1,#0x05         ;Condition check
      BLE  LOOP             ;if(i<=5) then looop

      END                   ;End of code
```

Exercise 3: Using the above template write an ARM assembly code to calculate the factorial of a number; let us calculate 5!

Conditional Execution: Most of the ARM instructions can be executed conditionally by post-fixing them with an appropriate condition code.

The condition code may determine whether the ARM core to execute it or not. Prior to the execution, processor compares the condition code with the condition flag in the CPSR register. If it matches, then the instruction is executed; otherwise the instruction is ignored. The following table lists the condition codes.

Condition Code	Interpretation	Status Flag
EQ	Equal / equals zero	Z set
NE	Not equal	Z clear
CS/HS	Carry set / unsigned higher or same	C set
CC/LO	Carry clear / unsigned lower	C clear
MI	Minus / negative	N set
PL	Plus / positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N equals V
LT	Signed less than	N is not equal to V
GT	Signed greater than	Z clear and N equals V
LE	Signed less than or equal	Z set or N is not equal to V
AL	Always	any
NV	Never (do not use!)	none

Example 4: Consider the following demo code, content of r1 is compared with r2; if there is a match (if zero flag is set), then the processor performs add operation; i.e. it adds the content of r3 with of r1 and stores the sum in r0; where the add operation is conditionally performed; the ADD instruction is conditionally executed.

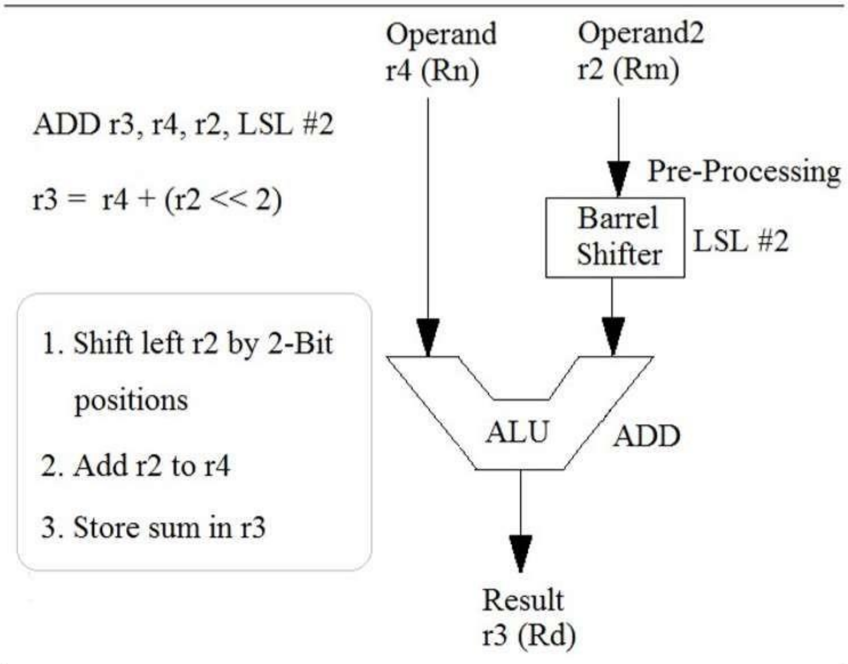
```
;Demo Code - Done at Amrita
;Conditional Execution
;if(r2==r1) then r0=r1+r3

    AREA    MyCode, CODE, READONLY

ENTRY                   ;Begining of code
    MOV r1, #0x02   ;r1=0x02
    MOV r2, #0x02   ;r2=0x02
    MOV r3, #0x03   ;r3=0x03
    CMP r1, r2      ;if (r1==r2) then
    ADDEQ r0, r1,r3 ;compute r0=r1+r3

LOOP B LOOP             ;Stay here
    END                 ;End of code

```

By default, data processing instructions (ADD, SUB, MUL, MOV, AND, etc.) do not affect the condition flags (except compare, CMP does not need S) unless the mnemonic is post-fixed with S. In the following program, SUBS r1, r1, #0x01 decrements the contentment r1 by 1 (subtracts 1 from r1 and stores the difference in r1 itself); and sets the condition flags as well. This improves the code density and performance by reducing the number of forward branch.

Exercise 4: Convert the following C code into equivalent ARM assembly code; use conditional execution feature.

```
/* Demo Code - Done at Amrita
Conditional Excution */

int main(){
int x=20, y=30, z;

if (x>y)
z=x-y;
else if (x==y)
z=x+y;
else
z=x*y;

return(0);
}
```

Shift Operations: The major functional units of the ARM processor are: Arithmetic and Logic Unit, Barrel Shifter, Booth Multiplier or MAC, Register file and Control Unit. Barrel shifter is basically a combinational circuit which can shift or rotate a data to left or right by an arbitrary (or a specified) number of bit positions at once in a single clock cycle Architecturally the Barrel Shifter is associated with an ALU, The Shifter pre-processes the data, before it enters the ALU. Note that in the above figure second operand r2 is shifted left by 2 bit positions and enters into ALU.



Shift Operations	Syntax
Logical shift left by immediate	Rm, LSL #imm
Logical shift left by register	Rm, LSL Rs
Logical shift right by immediate	Rm, LSR #imm
Logical shift right with register	Rm, LSR Rs
Arithmetic shift right by immediate	Rm, ASR #imm
Arithmetic shift right by register	Rm, ASR Rs
Rotate right by immediate	Rm, ROR #imm
Rotate right by register	Rm, ROR Rs
Rotate right with extend	Rm, RRX
Rm and Rs --> Registers	#imm --> Immediate Value

Example 5: Evaluate the expression: S =1+4+8+16+32+64+122. To generate the series, load 0x01 in register r2 and perform logical shift left (LSL) r2 by 2 bit positions.

Signature of the faculty

with date: ………………………………

10. Load and Store
Multiply and Accumulate

Name : Roll Number :

ARM is a Load and Store Architecture, and the data to be processed must be loaded into core registers before processing. For loading a 32-bit value into a general purpose LDR (Load Register) instruction should be used. For example, LDR r0, =0x100011111, loads a 32-Bit value into the register r0; such a 32-bit constant cannot be used with MOV instruction.

Load and Store inductions: There are three forms of load and store instructions:

- Single register load and store instructions
- Multiple register load and store instructions
- Single register swap instructions

Single register load or store instructions transfers a byte or a 32-bit word, 16-Bit half-word between register and memory; whereas multiple register load and store instructions (LDM/STM) are used to transfer a block of data. The basic load and store instructions are LDR & STR (Load & Store Word).

STR r0, [r1] ; r0 → memory[r1]

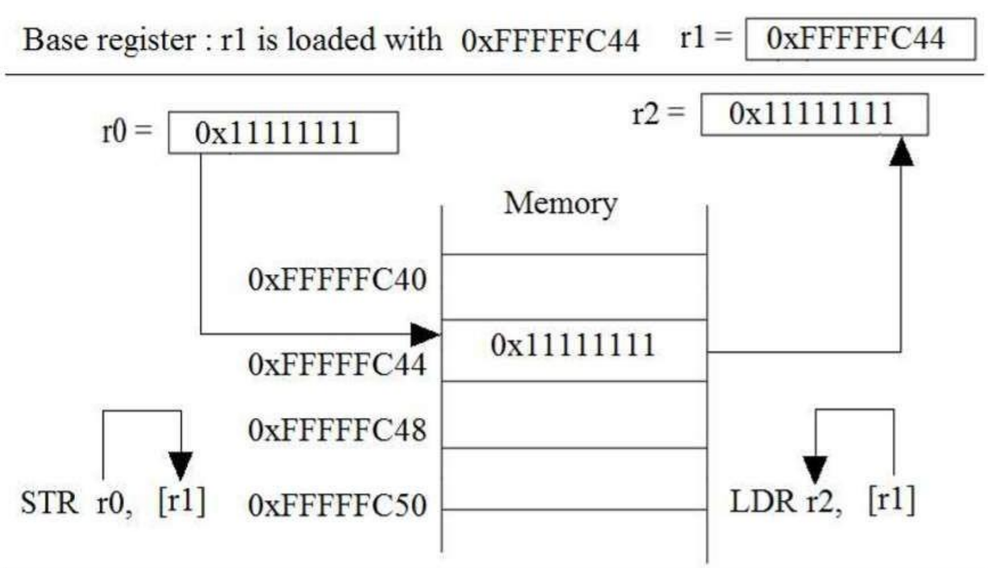
LDR r2, [r1] ; memory[r1] → r2

Address of a memory to be accessed is assumed to be in a register is called as base register, where r1 is considered as a base register. The STR instruction stores the content of source register into a memory location pointed by r1; the LDR loads the destination register with the content of memory location pointed by r1. The following table shows different load and store instructions supported by ARM architecture.

Load and Store Instructions

Instruction	Description
LDRB	Load unsigned 8-Bit value
STRB	Store signed/unsigned 8-Bit vlaue
LDR	Load signed/unsigned 32-Bit vlaue
STR	Store signed/unsigned 32-Bit vlaue
LDRSB	Load signed 8-Bit vlaue
LDRH	Load unsigned 16-Bit vlaue
LDRSH	Load signed 16-Bit vlaue
STRH	Load signed/unsigned 16-Bit vlaue

Consider the following figure, the base register r1 is initialized with an address 0xFFFFFC44; first the STR instruction stores the content of register r0 into the memory pointed by the r1; next the LDR instruction loads the content of memory location pointed by the r1 to register r2; thereafter content of r2 become 0x0x11111111.



Example 1: The following example code assumes a block of memory starts from 0xFFFFFC40, is loaded with five random words; the LDR r3, [r1], #4 instruction reads a word from a memory into a temporary register r3; immediately the base register is incremented by the specified offset value 4. For example, after reading the first value from 0xFFFFFC40 to 0xFFFFFC43, register r1 will become 0xFFFFFC44 to point the next memory location to be read in. The value read into register r3 is added with r0 to calculate the sum. In this way the code reads all five values one by one and adds with r0; at last it stores the overall 32-bit sum at memory location 0xFFFFFC54

```
; Demo code for Post-Index Addressing
; Done at Amrita
; Find the sum of a memory array

      AREA    Demo, CODE, READONLY

START
      MOV r0, #0            ; Initialize sum with Zero
      LDR r1, =0xFFFFFC40   ; Starting address of array
      MOV r2, #5            ; Size of the array
      ; r3 is used as TEMP

SUM   LDR r3, [r1], #4      ; Read 1st value from array
                            ; then update address
                            ; Address = r0+4
      ADD r0, r0, r3        ; Caculate Sum:r0=r0+r3
      SUBS r2, r2, #1       ; Decrement size of array
      BNE SUM               ; If Size!=0 jump to SUM
      STR r0, [r1]          ; Else store sum at
LOOP  B LOOP                ; Stay here forever
      END
```

Exercise 1: Execute the above code and verify output; then modify the code to calculate sum of bytes; use LDRB and STRB (instead of normal LDR and STR); and compare the byte sum with word sum.

Example 2a : Below code computes the sum of a byte array using a Table method; where the base register r0 is loaded with address of a table labelled LIST and register r1 is loaded with size of the table; register r2 to be used to store the sum is initialized with zero; elements of the table are read one by one in a loop to calcuate the sum as shown in the example1.

```
;Demo Code - Done at Amrita
;Sum of bytes of a LIST
;Using DCB, LDRB and STRB

       AREA    MyData, DATA, READWRITE
RESULT DCB 0
       ALIGN

       AREA    MyCode, CODE, READONLY
ENTRY                        ;Begining of code
       LDR r0, =LIST        ;Address of LIST
       MOV r1, #0           ;Initialize sum
       MOV r2, #5           ;Length of LIST
SUM
       LDRB r3,[r0],#1      ;Read memory
       ADD r1, r1, r3       ;Calculate sum
       SUBS r2, r2, #1      ;Decrement length
       BNE SUM              ;If r2!=0, jump to SUM
       LDR r0, =RESULT      ;Address of RESULT
       STRB r1,[r0],#1      ;Write sum in RESULT

EXIT   B EXIT               ;Stay here
LIST   DCB 5, 3, 2, 4, 2
       ALIGN
       END                  ;End of code
```

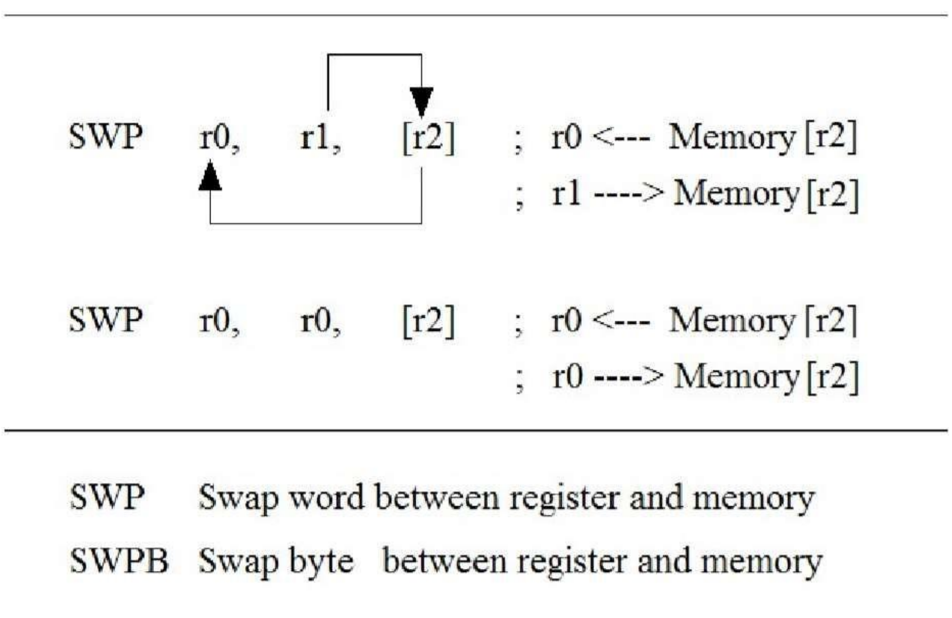
Example 2b: The below code reads data bytes from memory block1 and copies into block2.

```
;Demo Code - Done at Amrita
;Load and store bytes
;Copy bytes from block1 to block2

    AREA    MyCode, CODE, READONLY
ENTRY
    LDR r0, =0xFFFFFF00 ;Memory Block1
    LDR r1, =0xFFFFFF08 ;Memory Block2
    MOV r3, #5          ;Size of each block
LOOP
    LDRB r2, [r0], #1   ;Read byte from block 1
    STRB r2, [r1], #1   ;Write byte on block2
    SUBS r3, r3, #1     ;Decrement block size
    BNE LOOP            ;is size!=0 then loop
EXIT B EXIT             ;Stay here
    END                 ;End of code
```

Exercise 2:

a. Modify the above code to use Table method;
b. Each element read from the source block should be 1s complemented and copied into the destination use MVN instruction. Note that the MVN instruction moves a logical NOT (complement) of a register or 32-bit constant into the destination register.

Swap instructions: Swap is a special type of load store operation, which exchange or swap content of a memory location with a register r0 as shown in the following figure. Generally, this instruction requires two registers and one memory location.



Example 3: The following demo code exchanges the content of register r1 with the content of the memory location pointed by r0; and also exchanges the content of register r4 with the content of the memory location pointed by r3;

```
; Demo code for SWP - Done at Amrita
; Swap the content of r1 with memory[r0]
; and r4 with memory[r3]

      AREA    Demo_Code, CODE, READONLY
START
      LDR r0, =0xFFFFFF00 ;Load r0 with memory1
      LDR r1, =0xAAAAAAAA ;Load r1 Data1

      LDR r3, =0xFFFFFF08 ;Load r3 with memory2
      LDR r4, =0xBBBBBBBB ;Load r4 Data2

      ;Exachange r1 with [r0]
      MOV r2,r1            ; TEMP <= r1
      LDR r1,[r0]          ; r1 <= memory[r0]
      STR r2,[r0]          ; memory[r0] <= TEMP

      ;Exachange r4 with [r3]
      SWP r4, r4, [r3]     ;Swap [r3] with r4

LOOP  B LOOP               ;Stay here
      END
```

Exercise 3: Write an ARM assembly code to create two memory arrays: block1 and block2 for storing data items of size of one byte; load the first block with first six odd numbers and the second block with even numbers. Then swap block1 with block2; as a result, odd array become even and the even array become odd (elements are interchanged). A code written in C is given below for your reference. Use SWPB instruction alone for performing the swap operation.

```
/*Demo Code - Done at Amrita
Swap two 32-Bit integer arrays */

#include<lpc214x.h>
int main(void)
{
//integer arrays
int  array1[6]={1, 3, 5, 7, 9, 11};
int  array2[6]={2, 4, 6, 8, 10, 12};
int i,temp;
for(i=0;i<=5;i++)
  {
    temp=array1[i];
    array1[i]=array2[i];
    array2[i]=temp;
  }//for end
} //main end

```

Multiply and Accumulate (MLA); along with general multiplication, we have MLA instruction that accumulates the result of a multiplication (product) with another register; for example MLA Rd, Rm, Rs, Rn; which is expected to produce a 32-Bit result; where we actually multiply Rm with Rs; and the product is added with Rn; finally the result is stored in Rd (destination register).

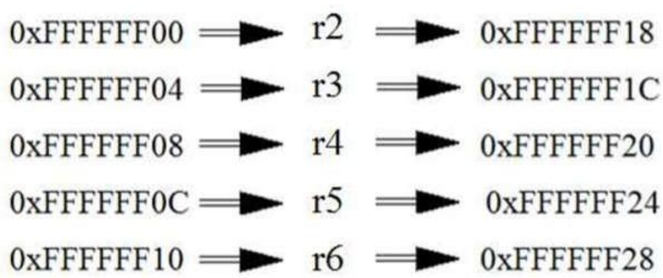
Mnemonic	Descripstion	Example
MLA	Multiply and Accumulate	Rd = (Rm x Rs) +Rn
MUL	Multiply	Rd = Rm x Rs
MLA Rd, Rm, Rs, Rn MUL Rd, Rm, Rs		MLA r0, r1, r2, r3 ; r0 = (r1 x r2) + r3 MUL r0, r1, r2 ; r0 = r1 x r2
SMULL	Signed Multiply Long	{RdHi, RdLo} = {RdHi, RdLo} + (Rm x Rs)
SMLAL	Signed Multiply Accumulate Long	{RdHi, RdLo} = Rm x Rs
UMLAL	Unsigned Multiply Accumulate Long	{RdHi, RdLo} = {RdHi, RdLo} + (Rm x Rs)
UMULL	Unsigned Multiply Long	{RdHi, RdLo} = Rm x Rs
UMULL RdLow, RdHigh, Rm, Rs		UMULL r0, r1, r2, r3 ;{ r1, r0 }= r2 x r3

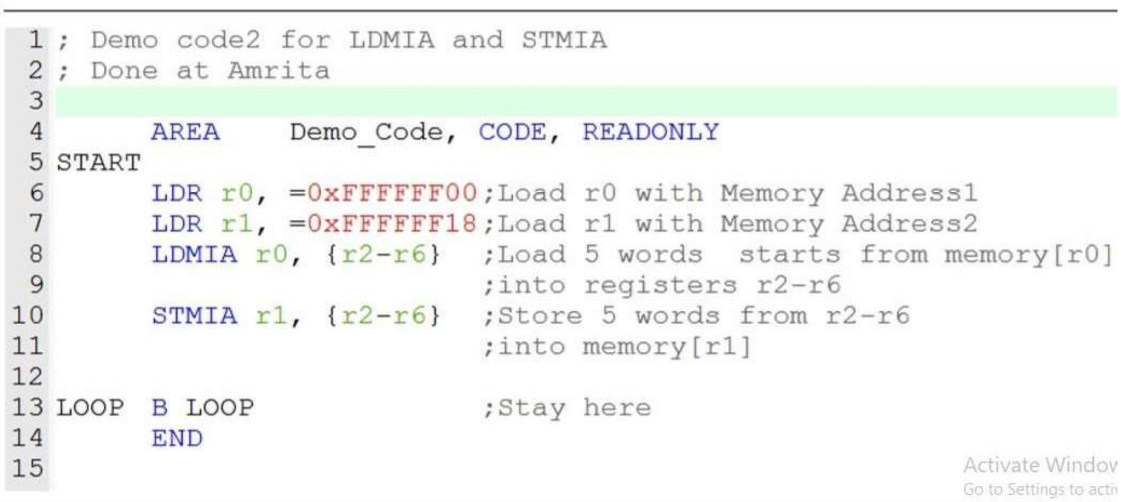
Example 4 Evaluating the expression $s = x^2 + y^2$; following code equates the variables x and y to the constants 3 and 2 respectively; square of the x is calculated in r3; and the MLA instruction calculates the square of y and accumulates with r3.

```
; Demo code done at Amrita
; Equate symbolic names to values

      AREA demo, CODE,READONLY
x     EQU 3                ;Equate x to 1
y     EQU 2                ;Equate y to 2
START
      MOV r1,#x            ;load r1 with x
      MOV r2,#y            ;load r2 with y
      MUL r3, r1, r1       ;r3=x*x
      MLA r3, r2, r2, r3 ;r3=(y*y)+r3

EXIT  B EXIT               ; r3 = ( x^2+ y^2 ) Done
      END
```

Exercise 4a: Evaluate the following expression, S $=1^3+2^3+3^3+4^3+5^3$ that computes the sum

Example 5a: Transferring data from one memory block to another block. Look at the following sequence that copies a memory block of five 32-bit words specified by register r0, into another memory block specified by r1. Note that the five words are read from the first memory block begins at 0xFFFFFF00 into five registers r2 to r6 then transferred to the second memory block starts at 0xFFFFFF18.





```
; Demo code2 for LDMIA and STMIA
; Done at Amrita

      AREA    Demo_Code, CODE, READONLY
START
      LDR r0, =0xFFFFFF00;Load r0 with Memory Address1
      LDR r1, =0xFFFFFF18;Load r1 with Memory Address2
      LDMIA r0, {r2-r6}  ;Load 5 words  starts from memory[r0]
                         ;into registers r2-r6
      STMIA r1, {r2-r6}  ;Store 5 words from r2-r6
                         ;into memory[r1]

LOOP  B LOOP             ;Stay here
      END
```

Example 5b: The following sequence adds two 64-bit integers; it assumes lower and higher bytes of the first value are stored at registers r0 and r1 respectively; similarly, r2 holds the lower byte of the second value and r3 holds the higher byte.

```
; Demo for adding two 64-bit nubers done at Amrita

      AREA    Demo, CODE, READONLY

START LDR r0, =0x80010001 ;Load lower byte of 1st Integer
      LDR r1, =0x01002001 ;Higher lower byte of 1st Integer
      LDR r2, =0x80000020 ;Load lower byte of 2nd Integer
      LDR r3, =0x01003001 ;Load higher byte of 2nd Integer

      ADDS r2, r2, r0     ;Add lower bytes of integers
                          ;(r0+r2)
      ADC r4, r3, r1      ;Add higher bytes integers
                          ; with pervious carry
                          ;(r1+r3+C)
LOOP  B LOOP
      END
```

Exercise 5: Implement the above task (64-Bit addition) by using LDM and STM; add the lower words together, with ADDS; and higher words with ADC. Finally, store the 64-bit sum in memory at address 0xFFFFFF00.

Signature of the faculty

with date: ………………………………