

1. Introduction

In this lab, we introduce the ARM software development toolkit (Keil MDK version 4.12) and the ARM processor emulator. You will also get some familiarity of ARM assembly language programming.

The ARM processor Architecture:

ARM7 Processor is a Mobile Processor of Size 32-bit.

→ It uses 16 registers 32-bits size each:

- Registers from **R0 - R12** → are general-purpose registers and can be used for any purpose.
- Register **R13** that is called: **SP (Stack Pointer)** → (more about this later).
- Register **R14** that is called: **LR (Link Register)** → is used to store the return address at function calls.
- Register **R15** that is called: **PC (Program Counter)** → is the program counter. It holds the address of the next instruction to be fetched for execution.

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13
r14
r15 (PC)

The ARM Instruction set can be divided to six broad classes of instructions:

- Data Movement instructions
- Arithmetic instructions
- Logical and Bit manipulation instructions
- Memory Access instructions
- Flow Control instructions
- System Control/Privileged

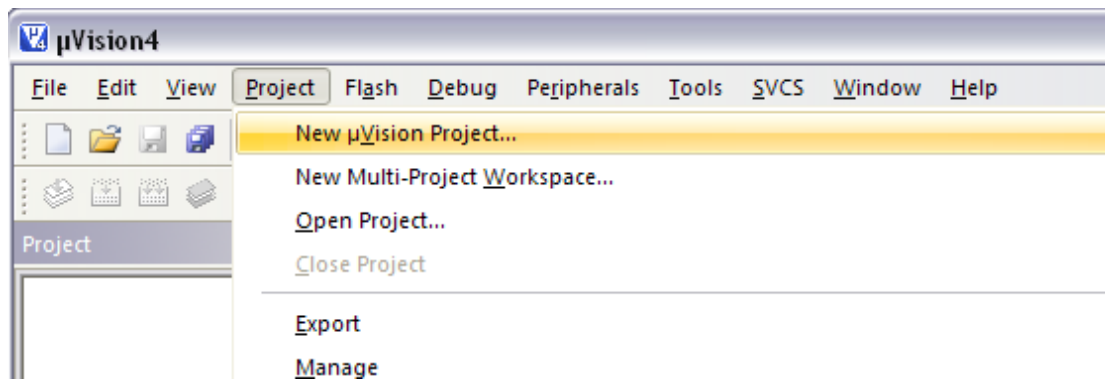
The ARM Instruction Set

Mnemonic	Operation	Mnemonic	Operation
MOV	Move	MVN	Move Not
ADD	Add	ADC	Add with Carry
SUB	Subtract	SBC	Subtract with Carry
RSB	Reverse Subtract	RSC	Reverse Subtract with Carry
CMP	Compare	CMN	Compare Negated
TST	Test	TEQ	Test Equivalence
AND	Logical AND	BIC	Bit Clear
EOR	Logical Exclusive OR	ORR	Logical (inclusive) OR
MUL	Multiply	MLA	Multiply Accumulate
SMULL	Sign Long Multiply	SMLAL	Signed Long Multiply Accumulate
UMULL	Unsigned Long Multiply	UMLAL	Unsigned Long Multiply Accumulate
CLZ	Count Leading Zeroes	BKPT	Breakpoint
MRS	Move From Status Register	MSR	Move to Status Register
B	Branch		
BL	Branch and Link	BLX	Branch and Link and Exchange
BX	Branch and Exchange	SWI	Software Interrupt
LDR	Load Word	STR	Store Word
LDRH	Load Halfword	STRH	Store Halfword
LDRB	Load Byte	STRB	Store Byte
LDRSH	Load Signed Halfword	LDRSB	Load Signed Byte
LDMIA	Load Multiple	STMIA	Store Multiple
SWP	Swap Word	SWPB	Swap Byte
CDP	Coprocessor Data Processing		
MRC	Move From Coprocessor	MCR	Move to Coprocessor
LDC	Load To Coprocessor	STC	Store From Coprocessor

2. Creating a New Project

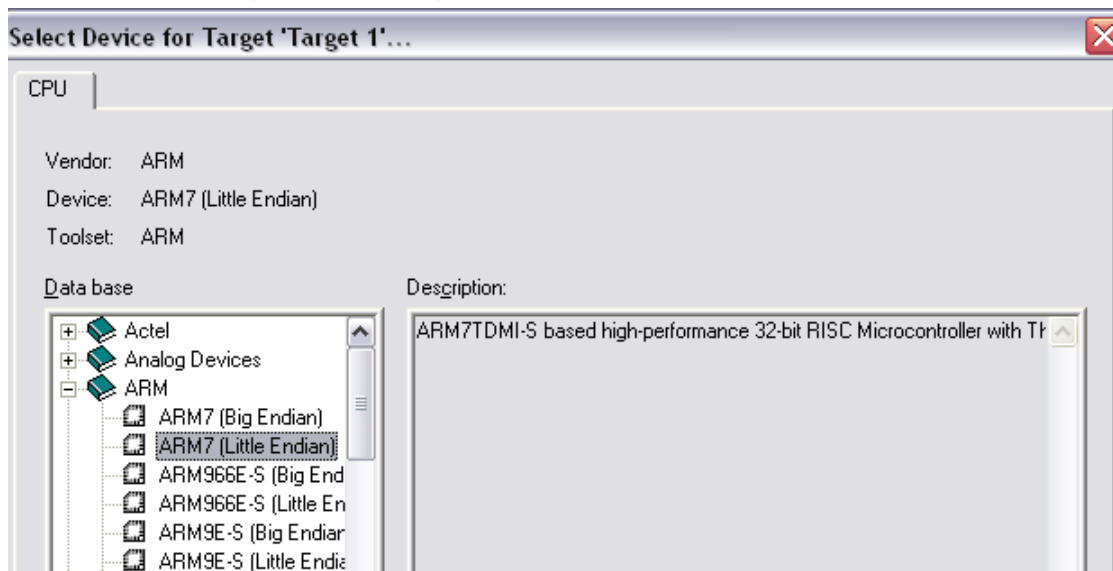
The first step is to run our ARM assembly simulator tool called “**Keil uVision4**” then follow the next steps:

1- Create a new project

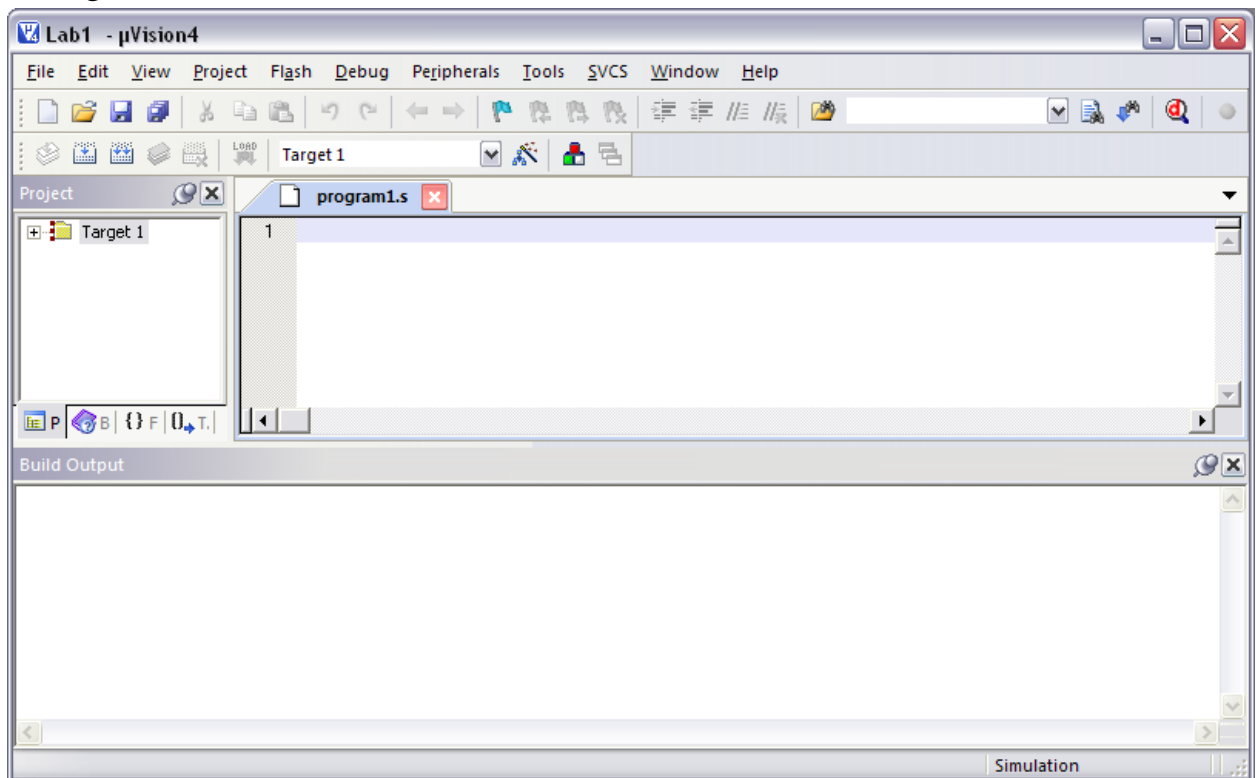


Then create a new folder called “Lab1” that will contain all the project files and source codes. Save your project named “Lab1” into this folder.

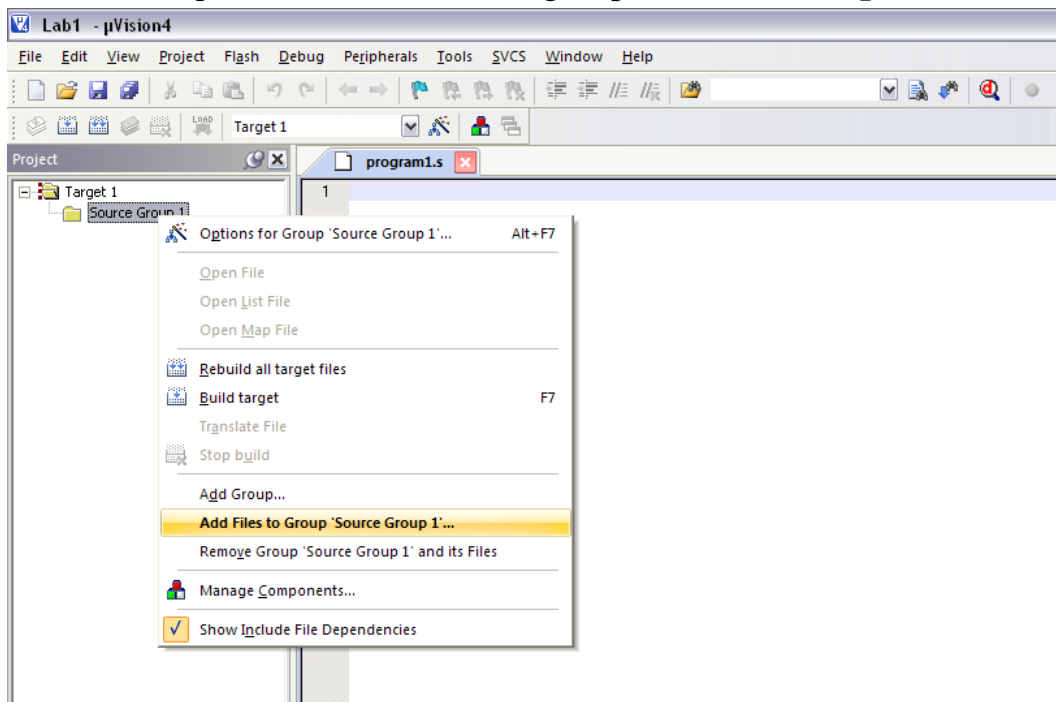
2- The Following window will appear to choose the target architecture which will be: **“ARM -> ARM7 (Little Endian)”**



3- Create a new file and save it in your project folder with name: “program1.s” as following:



4- The final step is to add this file as one of the project files to be assembled and built. Right-click on the source group “**Source Group 1**” as following:



2. Writing your code

➔ Structure of the Program:

```
AREA Lab1, CODE
ENTRY
.....
← Here you insert your code
.....
END
```

The First line, which starts with **AREA** defines a new section in the program with a user given name *lab1* and with an attribute **CODE** specifying that this section contain instructions (and not data).

The second line with the one word **ENTRY** pointing out that **the next instruction is the entry point of the program**, i.e., the start address.

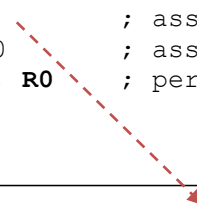
➔ At this step, you should be able to write your ARM assembly code in the supported editor. For a start, we will write a small program that should add two decimal numbers 6 and 10.

➔ **NOTE: It's very important to insert a tab at the beginning of each line, otherwise the assembler will produce errors.**

```
AREA Lab1, CODE
ENTRY

MOV R0, #6      ; assign value (6) to register R0
MOV R1, #10     ; assign value (10) to register R1
ADD R2, R1, R0  ; perform the calculation: R2 = R0 + R1

END
```



Immediate

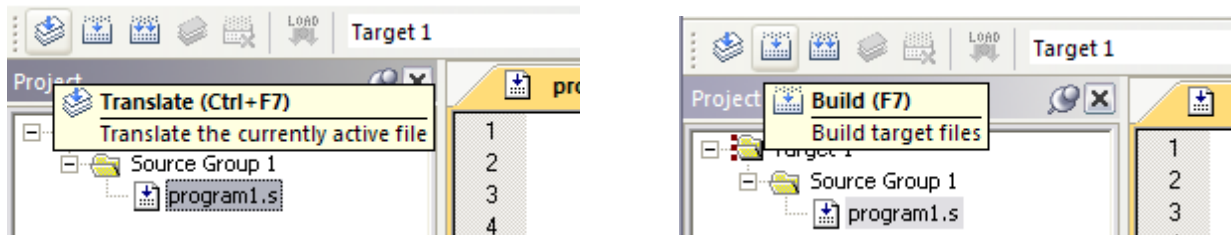
In the above code, we used two instructions illustrated below:

<p>MOV desReg, src</p> <p>Where: desReg is the destination register and src can be constant value (as in the example) or another register.</p> <p>Performs as assignment operator: desReg = src</p>	<p>ADD desReg, op1, op2</p> <p>Where: desReg is the destination register to store the value in, and op1 and op2 are the two operands to be added.</p> <p>Performs as addition operator: desReg = op1 + op2</p>
---	--

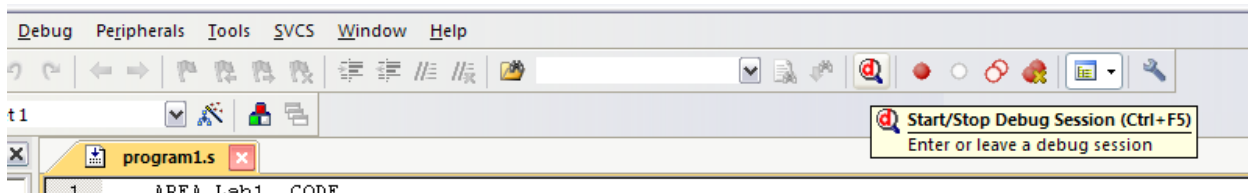
3. Running and debugging the program

Once the program is written, we need to assemble and link it to be executed. The assembler will check your code and state if you have any syntax errors to be fixed before executing the program.

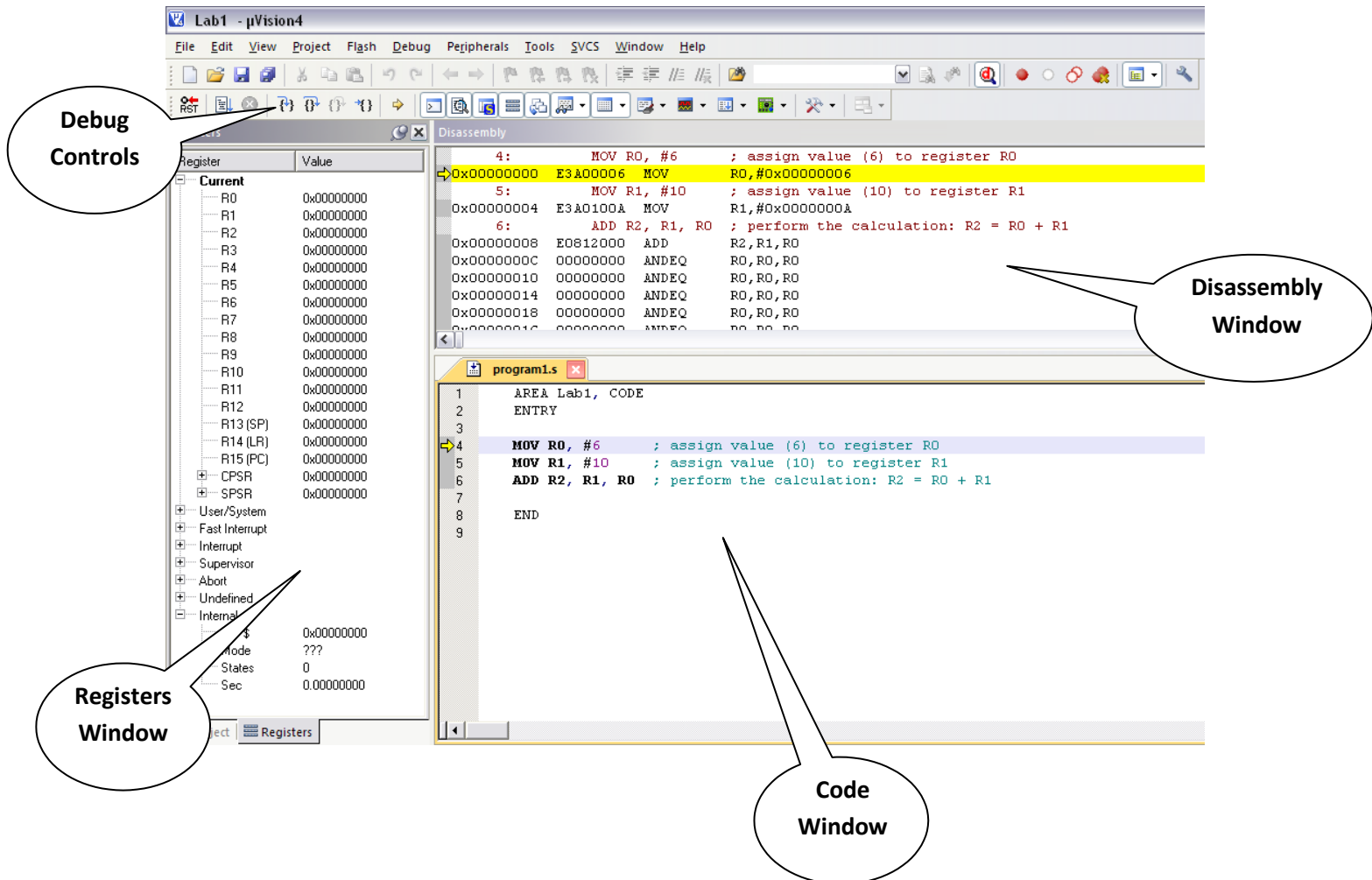
You can use toolbar icons as below, or use the same options from **Project** menu.



If your program is free of errors, you can start executing and debugging it using the icon on the toolbar or from **Debug** menu.



This emulator has a very good debugging tool. It enables you to view all the contents of the used memory and registers with each step executed in the program as following:



As you can see in the above figure, the program code (with the current statement to be executed using yellow arrow) is shown in the code window. The registers window shows the value of each general-purpose registers (**in Hexadecimal**) and the special registers (SP, LR, PC).

Use the debug controls to step into and execute each statement and check your results.