**ASSEMBLY PROGRAMMING WITH** **Keil Microcontroller Development Kit (MDK)**

The purpose of this lab is to introduce you to the layout and structure of assembly language programs and their format, as well as to the use of the Keil Microcontroller Development Kit (MDK). You will write your own programs later on in the semester with similar structure.

**INTRODUCTION TO MDK**

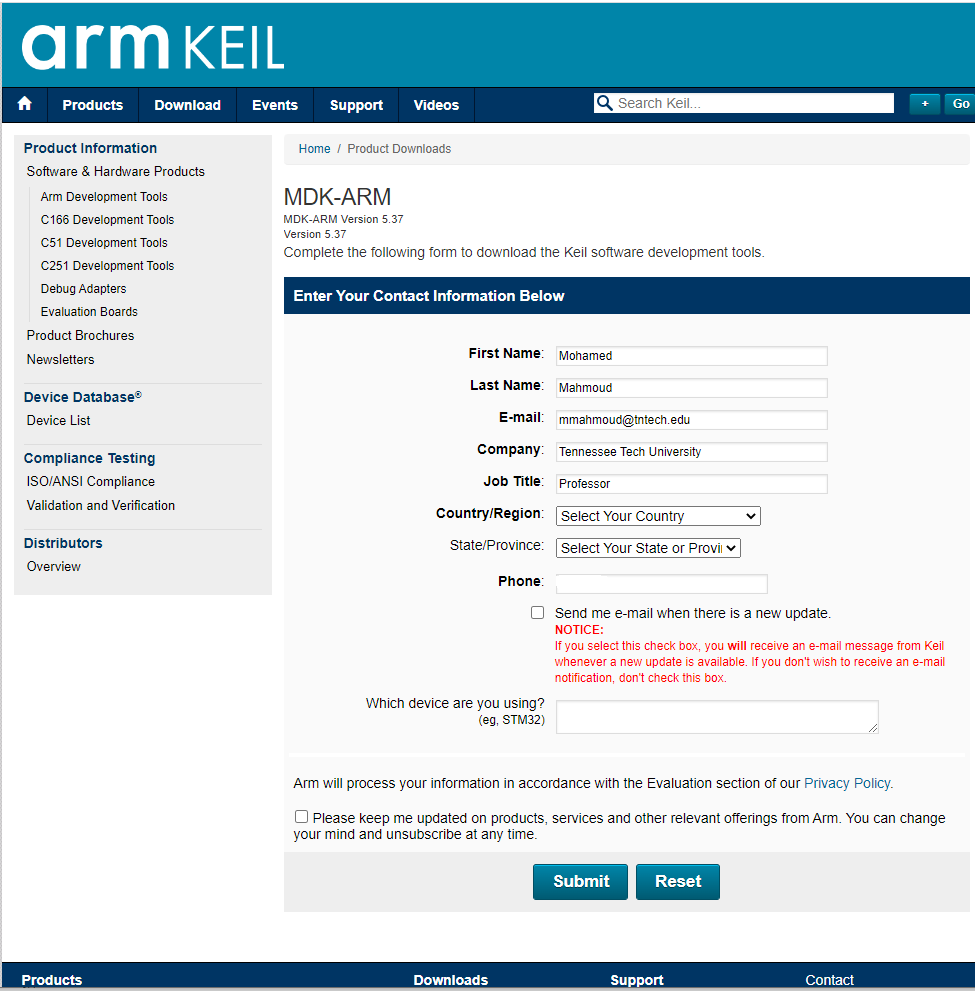
MDK is **a windows-based program**, which allows assembly programmer to assemble, debug, and download a program to ARM-based boards. **If your laptop’s operating system is not Windows, e.g., MAC. You should use Windows virtual machine or dual operating systems to use the code warrior.** This lab familiarizes the student with all the steps involved in assembling, running, and debugging assembly language programs. The students will be required to understand and remember all the steps to download and debug programs in future labs.

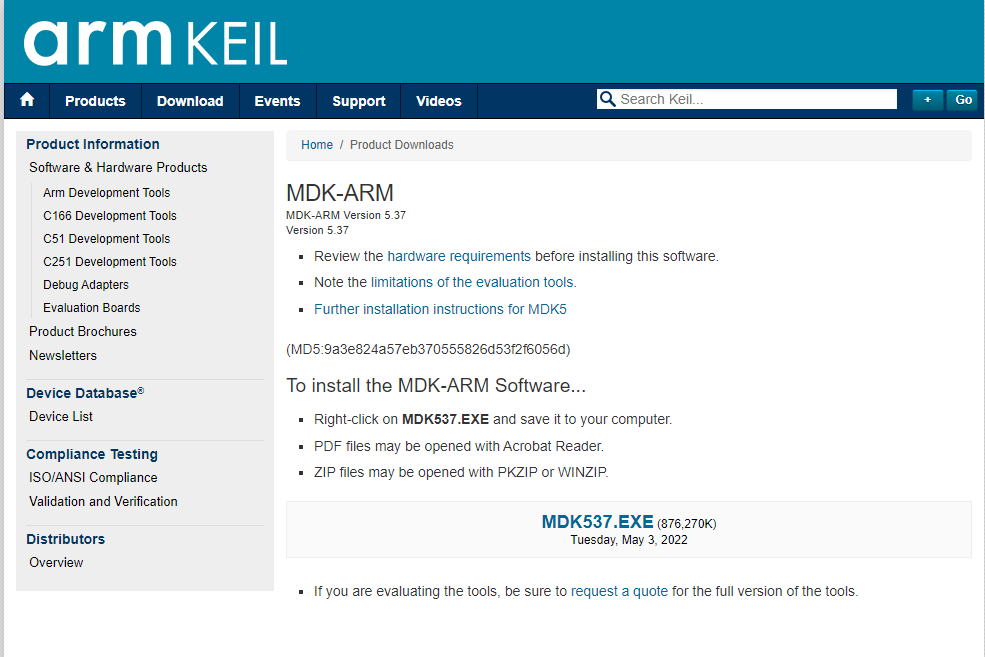
## **Step 1: Install Keil MDK-ARM**

* 1. Download the latest free evaluation version Keil MDK-ARM from the following link:

<https://www.keil.com/demo/eval/arm.htm>

* Keil MDK-ARM contains µVision 5 IDE (Integrated Development Environment) with debugger, flash programmer and the ARM compiler toolchain.
* The limitation of the free version is that programs that generate more than 32 Kbytes of code and data will not compile, assemble, or link.
  1. You may need to enter your contact information to obtain the executable file





* 1. Run the downloaded ***MDK5xx.exe*** and install to the default path. The software takes 2GB disk storage space. You can install it to a different driver, instead of the default C drive, if there is limited space in C drive.

Graphical user interface, text, application, email

Description automatically generated

After the core software is installed, a dialog will show up to install Keil Pack. It automatically downloads selected components (called packs) from <http://www.keil.com/dd2/pack/>

Graphical user interface, text, application, email

Description automatically generated

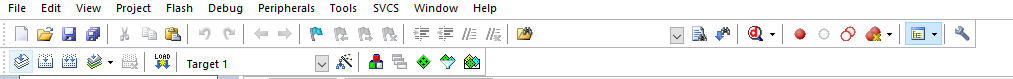
Click OK and then the following window shows up.

**You can also download the program from here**

<https://www.cae.tntech.edu/~mmahmoud/teaching_files/undergrad/ECE3130-ARM/MDK537.zip>

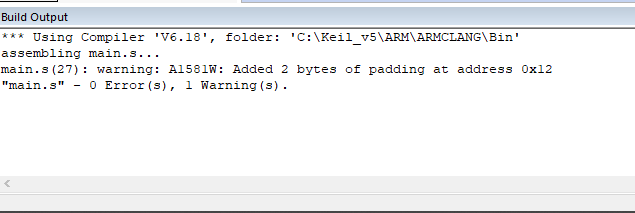
## **Step 2: Run a project**

1. Create a new project
2. To run a program, first click the translate button

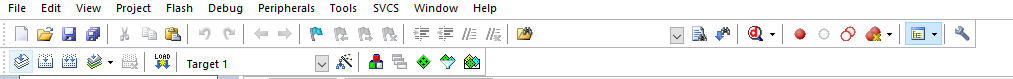


Translate

If there is no error in your program you should see

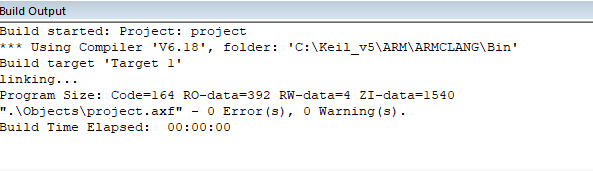


1. Compile and run your project



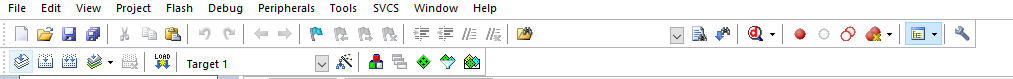
Build

If there is no error in your program you should see

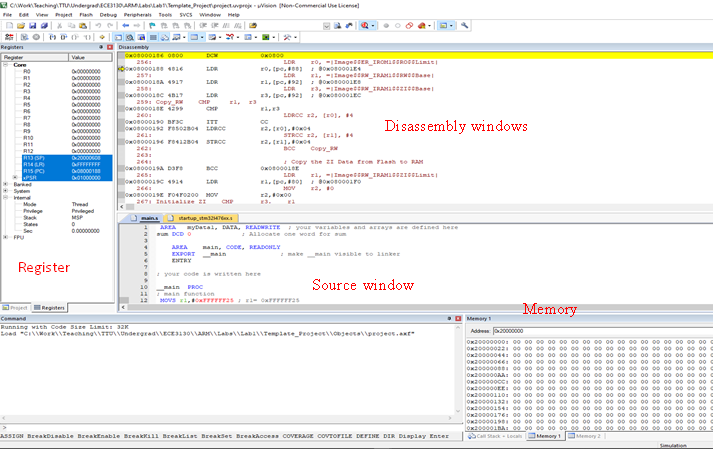
****

1. **Open the Keil Debugger**

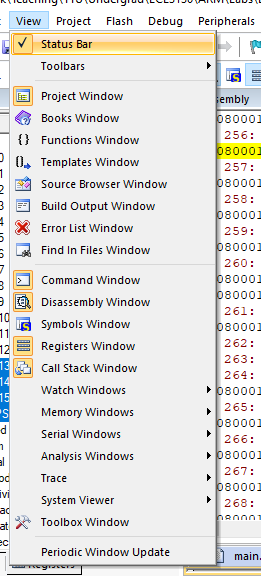
Click the Start/Stop Debug Button to start the debug and click it again to exit the debug.



Start/Stop Debug session

****The debugger windows is shown here

If you do not see any of these windows, activate them from View

****

**Memory Window**

The memory window is used to view the memory content in real time. By default, the address of data memory (RAM) starts at 0x20000000 and the flash memory starts from 0x80000000

The following assembly program defines and allocates an array of four words. Each word consists of four bytes. When we type the memory address 0x20000000, we can see the content of this array.

|  |
| --- |
| AREA myData, DATA, READWRITE  ALIGN  array DCD 1, 2, 3, 4 |

The memory content is displayed in bytes by default. Little-endian is used to store a word in the memory.

Table

Description automatically generated

By right click, we can select different display format. For example, we can show the content as unsigned integers.

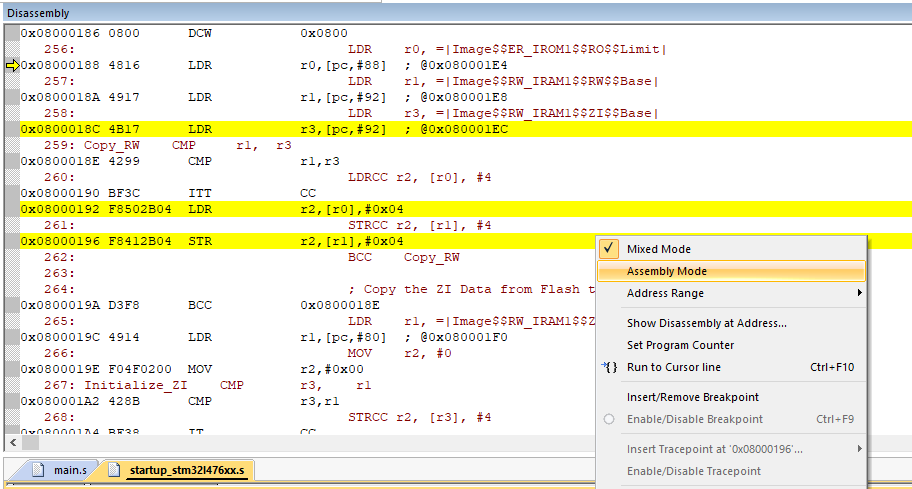
Graphical user interface, application

Description automatically generated

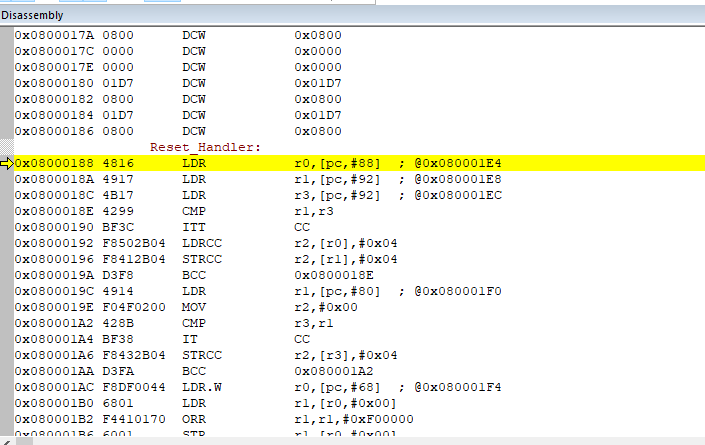
**Processor Registers**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Core Registers:   * Program counter (PC) r15 holds the memory address (location in memory) of the next instruction to be fetched from the instruction memory. * Stack point (SP) r13 holds a memory address that points to the top of the stack. SP is a shadow of either MSP or PSP. * xPSR (Special-purpose program status registers) is a combination of the following three processor status registers: * Application PSR * Interrupt PSR * Execution PSR  |  |  | | --- | --- | | **N** | Negative or less than flag (1 = result negative) | | **Z** | Zero flag (1 = result 0) | | **C** | Carry or borrow flag (1 = Carry true or borrow false) | | **V** | Overflow flag (1 = overflow) | | **Q** | Q Sticky saturation flag | | **T** | Thumb state bit | | **IT** | If-Then bits | | **ISR** | ISR Number ( 6 bits ) |   System:   * Base priority mask register (BASEPRI) defines the minimum priority for exception processing. * Priority mask register (PRIMASK) is used to disable all interrupts excluding hard faults and non-maskable interrupts (NMI). If an interrupt is masked, this interrupt is ignored (i.e. disabled) by the processor. * Control register (CONTROL) sets the choice of main stack or process stack, and the choice of privileged or unprivileged mode. * Fault mask register (FAULTMASK) is used to disable all interrupts excluding non-maskable interrupts (NMI). |

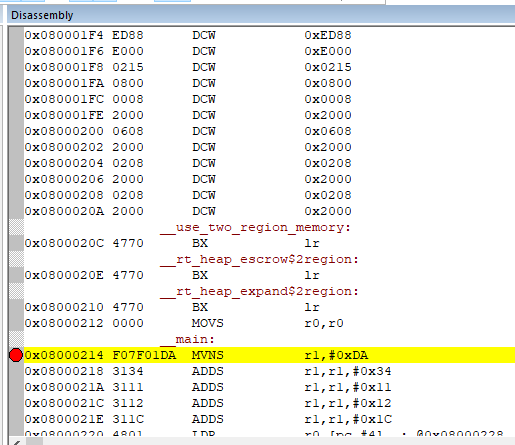
Right click on the disassembly window and select assembly mode



You should see this window



The program starts with an initialization code to initialize the system and then you see you program after label \_\_main:



Click on the gray area next to the first instruction after \_\_main. You see a red spot which is called a break point. When you run the program, it stops at break points. When a program stops at a breakpoint, the corresponding instruction has not been executed yet. Then click Run button to run the initialization code and stop at your code. After that you can run step by step.

The following table summarizes commonly used debug control buttons.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  | Set a Breakpoint | Run | Stop Debug | Step In | Step Over | Step Out |

* **Run**: Continues the execution from the current position until you click ***Stop*** or the program is paused by a breakpoint.
* **Step In**: Execute one step and enter the function if the current step calls a function.
* **Step Over**: Execute one step and run the function all at once if the current step calls a function.
* **Step Out**: Execute until the current function returns.

**Enabling Watch Window for Assembly Programs**

First, all variables you want to watch need to be exported at the top of the code using the EXPORT command, as shown below.

Graphical user interface, text

Description automatically generated with medium confidence

Next, while **in Debug Mode**, go to View, Watch Windows, and click either Watch 1 or Watch 2, it doesn't matter which one.

Graphical user interface, application

Description automatically generated

The window will appear in the same area as the Memory Window appears.

Graphical user interface, text, application

Description automatically generated

From here, double click on <Enter expression>, and then type in the variable that you want to watch. Make sure you spell the variable name properly.

Graphical user interface, text, application, table, Excel

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Repeat the previous step of double clicking <Enter expression> and typing in the variable you want to watch for all variables you want to watch. They will update as you run or step through your program.

The purpose of this lab is to run the following code and debug it. This is the same code you see in the template project

**; your variables and arrays are defined here**

**AREA myData1, DATA, READWRITE**

**sum DCD 0 ; Allocate one word for sum**

**AREA main, CODE, READONLY**

**; make \_\_main visible to linker**

**EXPORT \_\_main**

**ENTRY**

**; your code is written here**

**\_\_main PROC**

**; main function**

**MOVS r1,#0xFFFFFF25 ; r1= 0xFFFFFF25**

**ADDS r1, r1,#0x34 ; r1=r1+0x00000034**

**ADDS r1, r1,#0x11 ; r1=r1+0x00000011**

**ADDS r1, r1,#18 ; r1=r1+0x00000012**

**ADDS r1, r1,#2\_00011100 ; r1=r1+0x0000001C**

**;ADDS r1, r1,#0xFF ; run with and without this line**

**LDR r0,=sum ; r0=the address of sum**

**STR r1,[r0] ; sum=r1**

**stop B stop ; dead loop & program hangs here**

**ENDP**

**END**

**Things to turn in as your Lab1 Report:**

A. Run the code without this instruction ADDS r1, r1,#0xFF . Use single step repeatedly to execute instruction by instruction. *Observe* the content of the registers, as well as four of the flags after each single step execution. *Record* these values in Table 1. **[10 marks]**

Table 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PC | r0 | r1 | [0x20000000] | N | Z | V | C |
| 0x08000214 | 0x08000215 | 0x00F000000 | 00000000 | 0 | 1 | 0 | 1 |
| 0x08000218 | 0x08000215 | 0xFFFFFFF25 | 00000000 | 1 | 0 | 0 | 1 |
| 0x0800021A | 0x08000215 | 0xFFFFFFF59 | 00000000 | 1 | 0 | 0 | 0 |
| 0x0800021C | 0x08000215 | 0xFFFFFFF6A | 00000000 | 1 | 0 | 0 | 0 |
| 0x0800021E | 0x08000215 | 0xFFFFFFF7C | 00000000 | 1 | 0 | 0 | 0 |
| 0x08000220 | 0x08000215 | 0xFFFFFFF98 | 00000000 | 1 | 0 | 0 | 0 |
| 0x08000222 | 0x20000000 | 0xFFFFFFF98 | 00000000 | 1 | 0 | 0 | 0 |
| 0x08000224 | 0x20000000 | 0xFFFFFFF98 | FFFFFFF98 | 1 | 0 | 0 | 0 |

B. The machine code that is equivalent to the assembly program. **[5 marks]**

01DAF07F 31113134 311C3112 60014801 0000E7FE

C. Run the program again after adding the instruction ADDS r1, r1,#0xFF. Answers the following questions:

1. What are the values of *sum* in both cases? Are these values correct? **[5 marks]**
2. What are the flags after executing ADDS r1, r1,#0xFF? Explain why these values are obtained? Is there signed overflow or unsigned overflow? **[10 marks]**
3. Why PC is increment by 4 in the first instruction and then it is incremented by two in the other instructions? [**5 marks**]
4. WITHOUT - 98 WITH - 97
5. Flags after executing ADDS r1, r1,#0xFF are ‘N’ = 0, ‘Z’ = 0, ‘C’ = 1, ‘V’ = 0, ‘N’ means the value was not negative, ‘Z’ means the result is nonzero, ‘C’ means that there was a carry, ‘V’ means that there was no overflow.

There was no overflow whatsoever.

1. Increment by 4 means the computer architecture is 32-bit; Increment by 2 means the architecture is 16-bit.