

LABORATORY MANUAL

ECE 255

Introduction to Computer Architecture

Laboratory Experiment #1

This manual was prepared by

The many dedicated, motivated, and talented graduate students and
faculty members in the Department of Electrical and Computer
Engineering

The laboratory experiments are developed to provide a hands-on introduction to the ARM architecture. The labs are based on the open source tools Eclipse and OpenOCD.

You are expected to read this manual carefully and prepare **in advance** of your lab session. Pay particular attention to the parts that are **bolded and underlined**. You are required to address these parts in your lab report. In particular, all items in the **Prelab** section must be prepared in a written form **before your lab**. You are required to submit your written preparation during the lab, which will be graded by the lab instructor.

Laboratory Experiment 1: Using the Eclipse Integrated Development Environment

1.1 Goal

In this introductory experiment, you will learn the procedure for developing assembly language programs and the basic features of the ARM architecture.

1.2 Objectives

Upon completion of this lab, you will be able to:

- Assemble and link ARM architecture assembly programs.
- Load program onto an ARM development board.
- Run and debug programs.
- Examine / Modify memory locations (including program and data) and registers.
- Set breakpoints and single step execution.

1.3 Prelab

Prelab (In addition to your written submission, your Lab Instructor may ask you these questions during the lab and your answers will be graded individually.)

- What is a cross-assembler?
- What is an exception?
- Comment on the program in 1.4.2 Part 2 (Explain what it does).
- Comment on the program in 1.4.3 Part 3 (Explain what it does).

1.4 Hardware Introduction

The STM32F0DISCOVERY board allows users to discover STM32F0 Cortex-M0 features and to develop applications easily. Based on the **STM32F051R8T6** microcontroller featuring 64KB flash, 8 KB RAM in an LQFP64 package, the board includes an ST-LINK/V2 embedded debug tool, four LEDs (the red LD1 for 3.3V power in; red/green LD2 for USB communication; green LD3 for PC9 output; blue LD4 for PC8 output) and two push buttons (user button and reset button) (see Figure 1.1).

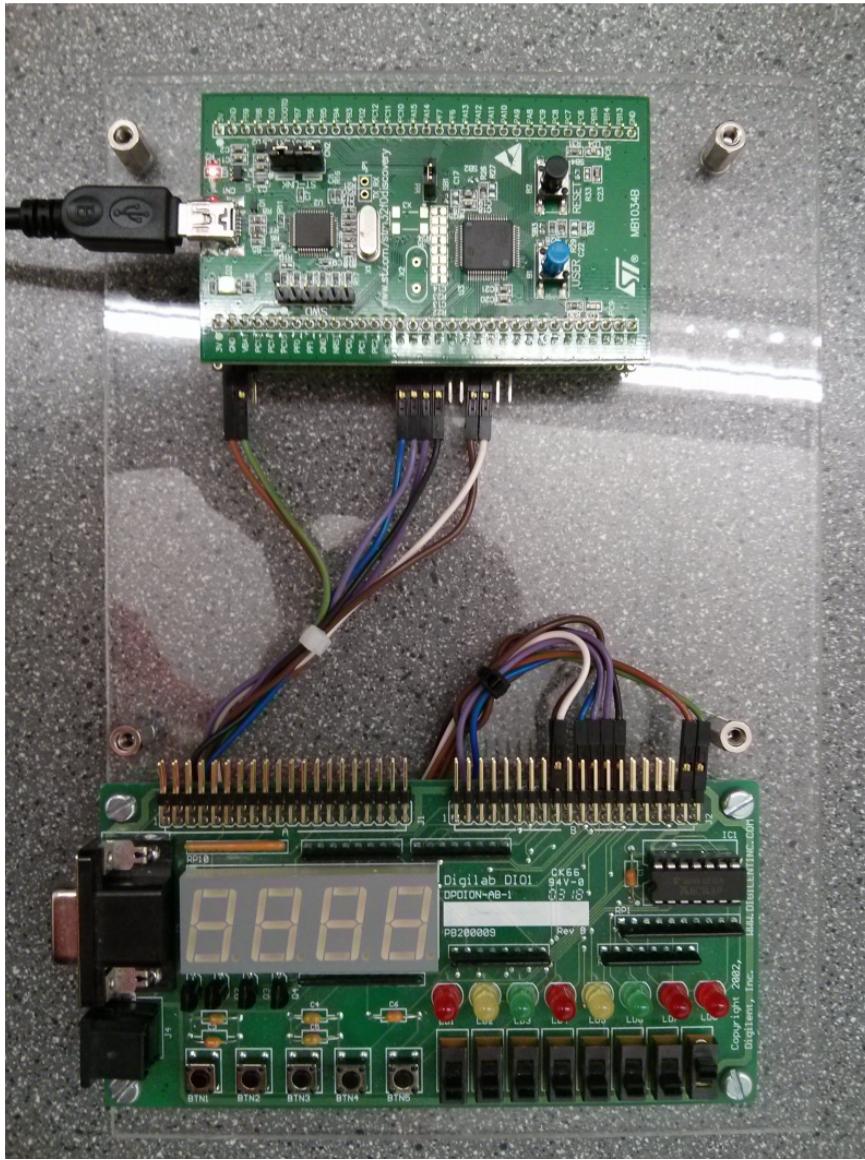


Figure 1.1: Hardware used in the ECE 255 Labs

Figure 1.1 shows the hardware used in this lab. The upper part is the main board for programming in Lab Experiment #1, #2, and #3, while the lower part is specifically designed for Experiment #4 to control and display the results of polling and interrupts with input/output.

For this experiment, we employ the **Eclipse** Integrated Development Environment (IDE), and the debugging tool OpenOCD for this board.

1.4.1 Part 1

1. This part introduces how to import an existing project to Eclipse, and then how to view and read the memory after downloading the executable binary file to the ARM board.

To begin this part, download the project lab1a (available on the lab web page) to your desktop. This project contains the assembly codes along with the supporting files. After the download, unzip it and import it to Eclipse.

To launch Eclipse and set its workspace:

- Double click the Eclipse icon on desktop
- Choose the directory to set the workspace
- Close the welcome window
- Open the zip file and copy the project to your Eclipse workspace.

The importing steps are:

- Choose “Import...” in file menu
- Choose “Existing Projects into workspace” in the General folder and click “Next” in the popup window
- Click “Browse...” to set the “Select root directory” in Import Projects from and click “Finish” to complete the import process

The assembly code in main.asm is shown in Listing 1.1 and is viewable in Eclipse after import. This code describes a simple addition of ‘4+5’. In Listing 1.1, first, the numbers 5 and 4 are stored in register **r4** and register **r5**, respectively, using the **mov** instruction. Then, the **add** instruction is used to sum the value in these two registers and the result is stored in register **r0**. Note the sign # used in front of 5 and 4.

Listing 1.1: Program that stores the results of “4 + 5” into register 0.

```
.text
.global main
main:
    nop
    mov r4, #5          // Load register r4 with the value 5
    mov r5, #4          // Load register r5 with the value 4
    add r0, r4, r5      // Add r5 and r4 and store in r0
stop:
    nop
    b stop
```

2. The assembler translates assembly language programs into object programs and the linker transforms object programs into executable programs. This process is described in Experiment #0: 2. Build the project.

After the build process, the elf file (in this case, it is **lab1a.elf**) will be generated in the **debug** folder. The **elf** file is an acronym for Executable Linking and Formatting and is the file format that is used in the GNU Linux/UNIX world for object files and executable files. It is also the format that the Eclipse IDE uses to store the code that is to be downloaded and debugged on the STM32F0 Discovery board.

Once you have assembled and linked your assembly language program, you can execute it through **Eclipse**. Before execution, the executable program has to be loaded onto the board by the debugging process (see 3. Configure debugging in Experiment #0).

The disassembled code can be viewed in the “**Disassembly**” window in Eclipse when debugging the project. Figure 1.2 shows the disassembled code of **lab1a** project. The first column shows the address of each instruction. The next columns represent the mnemonics of the downloaded program.

```
20000136: stmvs r1, {r1, r2, r3, r5, r7, r11, lr}
2000013a: strl r0, [r3], #-1417 ; 0x589
2000013e: strcs r4, [r0], #-2414 ; 0x96e
5           mov r4, #5          //Load register r4 with the value 5
* 20000142: movs r4, #5
6           mov r5, #4          //Load register r5 with the value 4
20000144: movs r5, #4
7           add r0, r4, r5      //Add r5 and r4 and store in r0
20000146: adds r0, r4, r5
8           stop: b stop
stop:
```

Figure 1.2: Listing for main.asm

Note: make sure the board is properly connected to the workstation and is powered on.

Figure 1.3 shows the memory with the **add** instruction. The ARM architecture is in little-endian

format that has the least significant byte of a word being stored at the lowest-numbered (address) byte, and the most significant byte stored at the highest-numbered byte. For example, the instruction **add r0, r4, r5** is translated to 0x6019 at the address 0x20000146. The translated instructions are stored in the memory as shown in Figure 1.4. The PC register holds the current program address (in this case is 0x20000142 which is the address of the instruction **mov r4, #5**). You can find more details about assembly language and different addressing modes in the PM0215 STM32F0xxx Cortex M0 programming manual.

Address	0 - 3	4 - 7	8 - B	C - F
20000140	00240524	04256019	FEE70000	002103E0
20000150	0A4B5B58	43500431	09480A4B	42189A42
20000160	F6D3094A	02E00023	13600432	074B9A42
20000170	F9D300F0	69F8FFF7	E3FF7047	AC020020
20000180	AC020020	AC020020	AC020020	AC020020
20000190	FEE70000	82B00023	01930093	274B1968
200001A0	80225202	0A431A60	191C8022	9202A020
200001B0	C0010868	13400093	019B0133	0193009B

Figure 1.3: Instruction memory

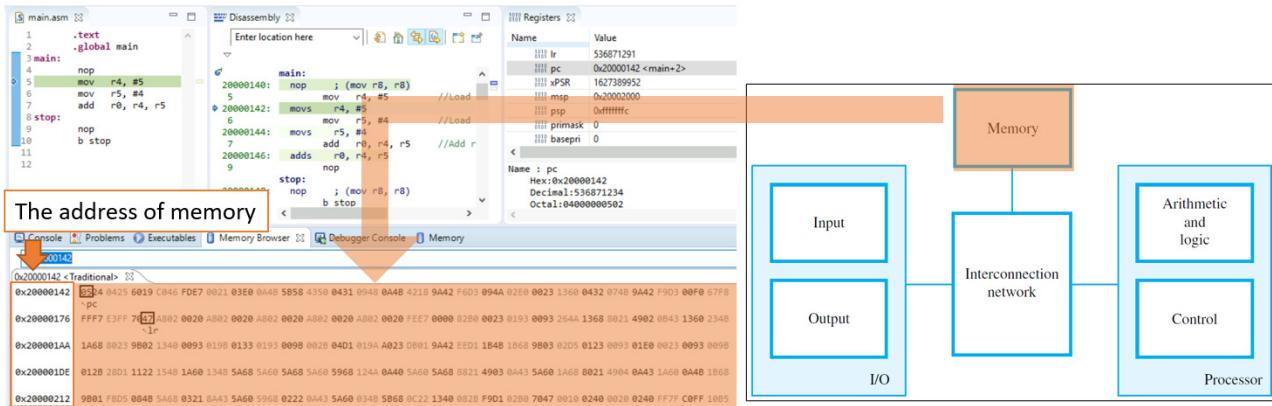


Figure 1.4: Accessing the content of memory

3. Before running the program, examine the contents of the memory and the registers. Figure 1.5 shows the content of the memory after loading the program onto the board. To verify this, use the **Window->Show View->Memory Browser** from the main menu and type the label name with the address character &, plus the variable type and an asterisk, for example, (**int ***)&main, or the address showing in the “Disassembly” view. If you click on this option, you will see the content of the memory as shown in Figure 1.5.

4. Use the step over in the debug window. This option allows us to trace the program line by line and to observe new memory content and register value. In this section, we execute the first instruction to see how the contents of the registers change. Before running the program, we check the content of the register in the **Registers** window.

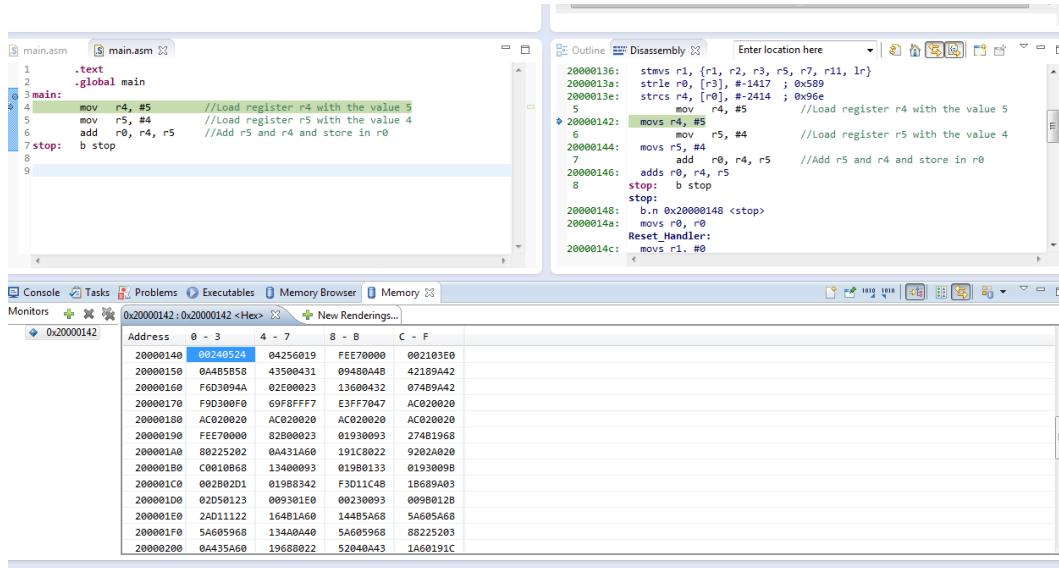


Figure 1.5: Content of the Memory

- Check the PC register (program counter) and the registers (r0-r7)
- Run the program using **Step Over** for two instructions
- Check the PC register
- Check the content of register r4. It is also possible to change the content of a register by selecting it and typing in a new value; you may wish to experiment the addition of different numbers

5. Next, we run the program to termination and check the register's content.

- Set a breakpoint at **stop**.
- Check the content of register **r0** (you should have 0x09 in this register).

Exercise: Change the content of the memory with address “0x20000146” to a “SUBTRACT” instruction and run the program. What are the content of the registers and why?

It is obvious that altering the content of the program memory can change a program's functionality.

6. The contents of some special registers are critical and changing them might result in exceptions (e.g., access to protected areas, or the ability to access non-instruction sections in the memory). Thus, we have to be careful when changing these registers. One of these registers is the PC register.

1.4.2 Part 2

1. The Lab Instructor will provide you with an URL (or get it from the lab web page) where you can download a small zip folder that contains the project **lab1as**. The **main.asm** file in the source folder contains the assembly code for this part of the lab.

To assemble the **lab1a.s** program, follow the steps in Lab #0 manual. The following shows the listing for **main.asm**.

```

main.asm  »13
1      .data
2      .align 4
3 x:    .word 0x2400
4 y:    .word 0x1200
5 sum:   .word 0x00
6
7
8 .text
9
10 .global main
11
12
13 main:
14     nop
15     ldr r3, =x
16     ldr r3, [r3]
17
18     ldr r5, =y
19     ldr r5, [r5]
20
21     add r5, r5, r3
22
23     ldr r6, =sum
24     str r5, [r6]
25 stop:
26     nop
27     b stop
28

08000318: ; <UNDEFINED> instruction: 0xfffffffffb
0800031c: ; <UNDEFINED> instruction: 0xfffffffffb
08000320: ; <UNDEFINED> instruction: 0xffffffffc0
08000324: mcr2 15, 5, pc, cr12, cr15, {7} ; <UNPREDICTABLE>
main:
    nop ; (mov r8, r8)
    ldr r3, =x
    ldr r3, [pc, #16] ; (0x800033c <stop+4>)
    ldr r3, [r3]
    ldr r3, [r3, #0]
    ldr r5, =y
    ldr r5, [pc, #16] ; (0x8000340 <stop+8>)
    ldr r5, [r5]
    ldr r5, [r5, #0]
    add r5, r5, r3
    adds r5, r5, r3
    ldr r6, =sum
    ldr r6, [pc, #12] ; (0x8000344 <stop+12>)
    str r5, [r6]
    str r5, [r6, #0]
    nop
stop:
    nop ; (mov r8, r8)
    b stop
    b.n 0x8000338 <stop>
    ldr r3, =x
    movs r0, r2
    movs r0, #0
    ldr r5, =y
    movs r4, r2
    movs r0, #0
    ldr r6, =sum

```

Figure 1.6: Source code for main.asm

Exercise: Describe the function of this program. The program has two sections, data and code. The data section includes the variables used in the program starting with the **.data** keyword and ending before **.text**. The code section starts with the “**.text**” keyword and ends at the last instruction.

The data section contains the initial value of the variables, if there are any. For example, in this program we define **x** as a word-sized (4 bytes) variable which takes 4 bytes of the memory and its initial value is 0x2400. The source code in the code section are translated and stored in the memory. As can be seen, each instruction is converted into its corresponding object code. For example, the instruction **add r5,r5,r3** is translated into 0xED18.

2. Build the project and configure it to debug. Before running the program with **step over**, examine the contents of the memory and the registers.

Figure 1.7 shows the content of the memory after loading the program onto the board. As can be seen from the file’s content, the code section starts at the location with address **0x08000324**.

The data section starts from address **0x20000010**. As all the variables are defined as **word**, they need four bytes for their storage. The first variable, **x**, has a value equal to 0x2400 and is located at address **0x20000010**. The second variable, **y**, has a value equal to 0x1200 and is located at address

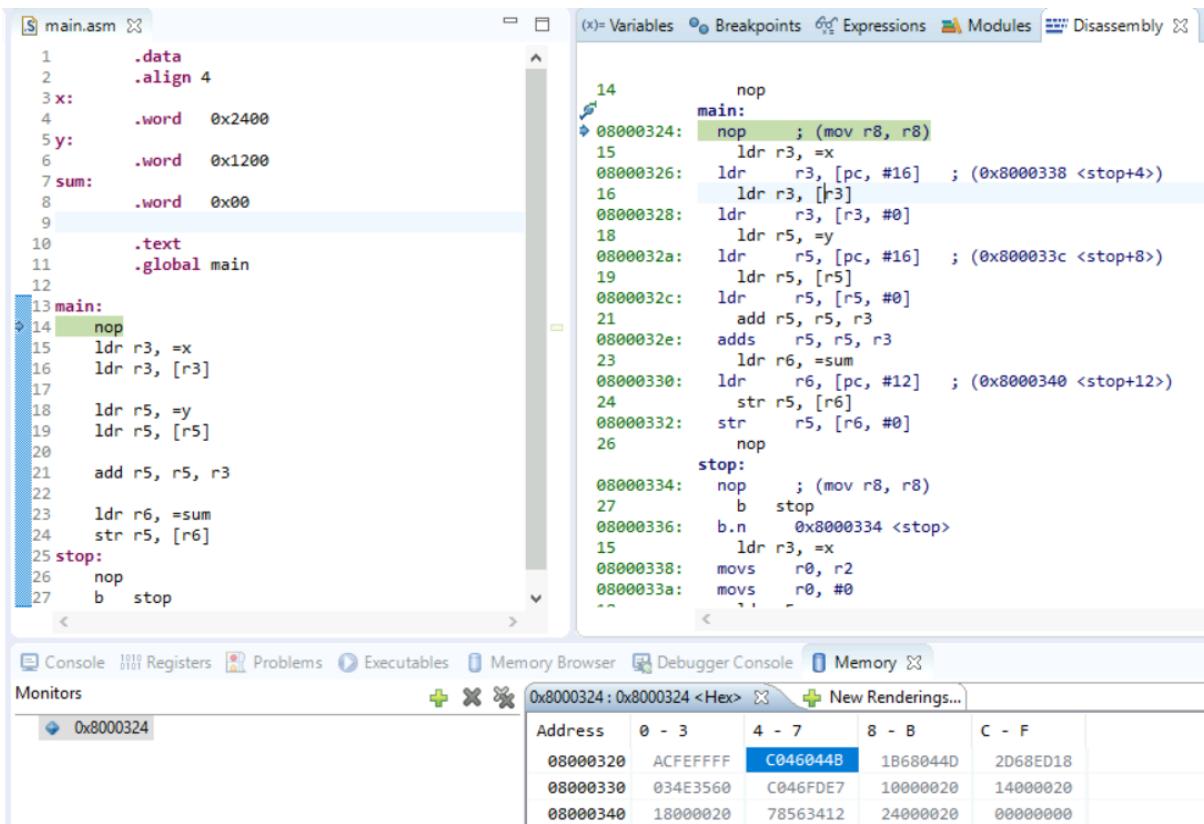


Figure 1.7: Content of the memory after loading the program onto the board

0x20000014. The next variable, `sum`, is located immediately after the variable `y`, and its value is equal to 0. These are shown in Figure 1.8, 1.9, and 1.10.

(int *)&x
0x20000010 - (int *)x <Traditional>
0x20000010 00002400 00001200 00000000 00000000 98765432 00000000 00000000 E7E5D002 60502000 BE004630 200003AC
0x20000040 200003AC 20002000 080001D5 080001DD 080001E1 00000000 00000000 00000000 00000000 00000000 00000000
0x20000070 080001E5 00000000 00000000 080001E9 080001ED 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1
0x200000A0 080001F1 00000000
0x200000D0 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 00000000 080001F1
0x20000100 00000000 F108F85F B083B530 4A324B31 4A326013 22004932 601AE001 428B3304 4B2CD3FB 2B00681B 4B2BD103
0x20000130 2B00681B E7FED000 4A2B4B26 4A2B6013 4B2B6013 492C4A2B 6810E003 33046018 428B3204 4B24D3F9 4B28681A D104429A
0x20000160 681A4B22 429A681B E7FED000 F86EF000 4669A801 F862F000 4D224B21 10AD1AED E0042400 4AE00A3 4798589B 42A53401
0x20000190 4B1DDCF8 1AED4D1D 240010AD 00A3E004 581B4819 34014798 DCF842A5 99009801 F8DCF000 4B161C05 1AE44C16 E00410A4
0x200001C0 00A33C01 589B4A12 2C004798 1C28DCF8 F832F000 CADEBABA 20000024 2000002C 20000024 20000030 20000000 20000000

Figure 1.8: Contents of memory containing the variable x

As you can see in Figure 1.11, if you run the program step by step (using “step into” icon), `r3` will hold the address of `x` in the memory after execution of line 15. After execution of line 16 the content of memory `x` is loaded in `r3` which is shown in Figure 1.12. Also the PC register value is `0x800032a` which is the address of the current instruction.

```
(int *)&y

0x20000014 - (int *)y <Traditional> X
0x20000014 00001200 00000000 00000000 98765432 00000000 00000000 E7E5D002 60502000 BE004630 200003AC 200003AC
0x20000044 20002000 080001D5 080001DD 080001E1 00000000 00000000 00000000 00000000 00000000 00000000 080001E5
0x20000074 00000000 00000000 080001E9 080001ED 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1
0x200000A4 080001F1 080001F1
0x200000D4 080001F1 08000000
0x20000104 F108F85F B083B530 4A324B31 4A326013 4B326013 22004932 601AE001 428B3304 4B2CD3FB 2B00681B 4B2BD103 2B00681B
0x20000134 E7FED000 4A2B4B26 4A2B6013 4B2B6013 492C4A2B 6810E003 33046018 428B3204 4B24D3F9 4B28681A D104429A 681A4B22
0x20000164 429A681B E7FED000 F86EF000 4669A801 F862F000 4D224B21 10AD1AED E0042400 4A1E00A3 4798589B 42A53401 4B1DDCF8
0x20000194 1AED4D1D 240010AD 00A3E004 581B4819 34014798 DCF842A5 99009801 F8DCF000 4B161C05 1AE44C16 E00410A4 00A33C01
0x200001C4 589B4A12 2C004798 1C28DCF8 F832F000 CADEBABA 20000024 2000002C 20000024 20000030 20000000 20000020 20000000
```

Figure 1.9: Contents of memory containing the variable y

```
(int *)&sum

0x20000018 - (int *)sum <Traditional> X
0x20000018 00000000 00000000 00000000 98765432 00000000 00000000 E7E5D002 60502000 BE004630 200003AC 20002000
0x20000048 080001D5 080001DD 080001E1 00000000 00000000 00000000 00000000 00000000 00000000 080001E5 00000000
0x20000078 00000000 080001E9 080001ED 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1
0x200000A8 080001F1 080001F1
0x200000D8 080001F1 08000000
0x20000108 B083B530 4A324B31 4A326013 4B326013 22004932 601AE001 428B3304 4B2CD3FB 2B00681B 4B2BD103 2B00681B E7FED000
0x20000138 4A2B4B26 4A2B6013 4B2B6013 492C4A2B 6810E003 33046018 428B3204 4B24D3F9 4B28681A D104429A 681A4B22 429A681B
0x20000168 E7FED000 F86EF000 4669A801 F862F000 4D224B21 10AD1AED E0042400 4A1E00A3 4798589B 42A53401 4B1DDCF8 1AED4D1D
0x20000198 240010AD 00A3E004 581B4819 34014798 DCF842A5 99009801 F8DCF000 4B161C05 1AE44C16 E00410A4 00A33C01 589B4A12
0x200001C8 2C004798 1C28DCF8 F832F000 CADEBABA 20000024 2000002C 20000024 20000030 20000000 20000020 20000000 08000344
```

Figure 1.10: Contents of memory containing the variable sum

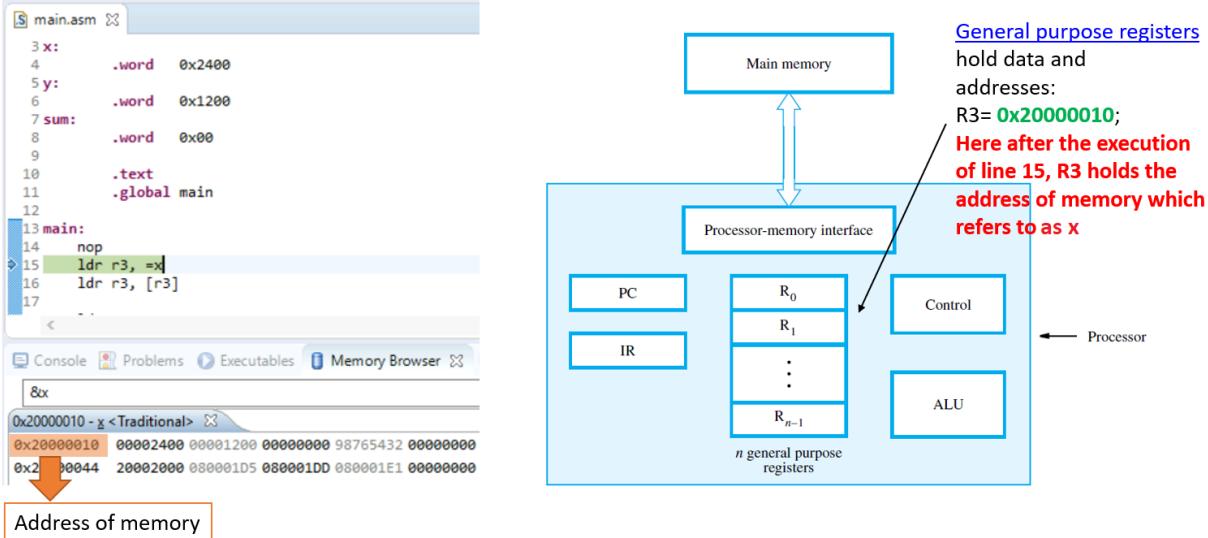


Figure 1.11: Content of r3 after execution of line 15

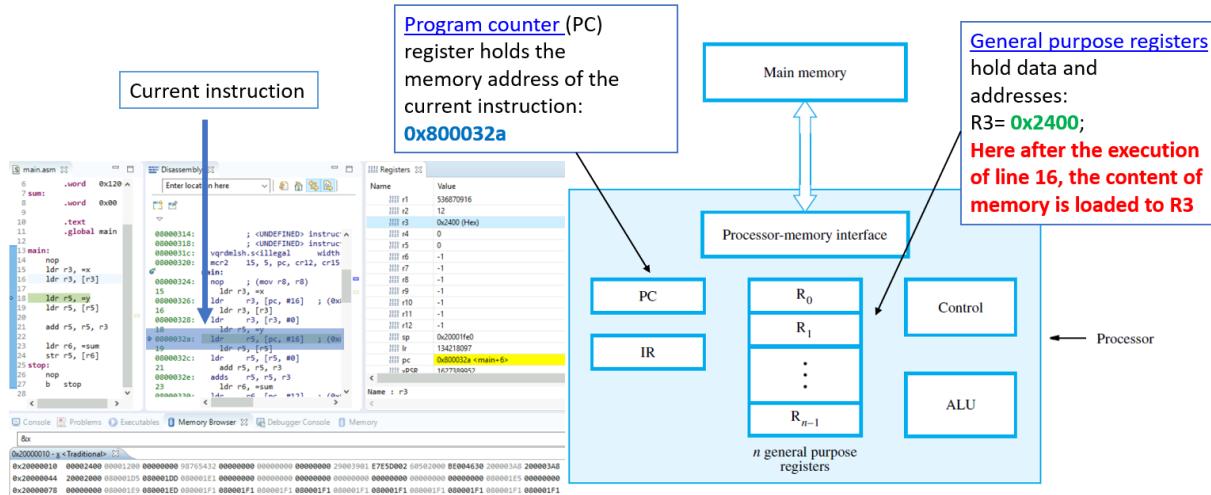


Figure 1.12: Contents of r3 and PC register after execution of line 16

- Resume execution
- Check the contents of the **sum** variable at address **0x20000018** (see Figure 1.13).

Important Note: As can be seen from the Memory and Register Windows in this experiment, you can change the contents of any memory located in ram or register during the execution of a loaded program. This method can be used to debug and check the correctness of the program execution.

(int *)&sum	
0x20000010 -	(int *)sum <Traditional>
0x20000018	00003600 00000000 98765432 00000000 00000000 E7E5D002 60502000 BE004630 200003AC 200003AC 20002000
0x20000048	080001D5 080001DD 080001E1 00000000 00000000 00000000 00000000 00000000 080001E5 00000000
0x20000078	00000000 080001E9 080001ED 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1
0x200000A8	080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 00000000 080001F1 080001F1
0x200000D8	080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 080001F1 00000000 080001F1 00000000 F108F85F
0x20000108	B083B530 4A324B31 4A326013 4B326013 22004932 601AE001 428B3304 4B2CD3FB 2B00681B 4B2BD103 2B00681B E7FED000
0x20000138	4A2B4B26 4A2B6013 4B2B6013 492C4A2B 6810E003 33046018 428B3204 4B24D3F9 4B28681A D104429A 681A4B22 429A681B
0x20000168	E7FED000 F86EF000 4669A801 F862F000 4D224B21 10AD1AED E0042400 4A1E00A3 47985898 42A53401 4B1DDCF8 1AED4D1D
0x20000198	240010AD 00A3E004 581B4819 34014798 DCF842A5 99009801 F8DCF000 4B161C05 1AE44C16 E00410A4 00A33C01 589B4A12
0x200001C8	2C004798 1C28DCF8 F832F000 CADEBABA 20000024 2000002C 20000024 20000030 20000000 20000020 20000000 08000344

Figure 1.13: Contents of memory contents of sum after Execution

1.4.3 Part 3

The following source code is considered for this portion of the lab.

```
1      .data
2      .align 4
3 x_array:    .word   0x24,0x12,0x05,0x66,0x12,0x01,0x08,0x14
4 y_array:    .word   0x12,0x33,0x21,0xA,0x15,0x11,0x25,0x99
5 sum_array:  .space  32
6
7      .text
8      .global main
9
10 main:     nop
11     ldr    r0, =x_array      @ Load register r0 with the value 5
12     ldr    r1, =y_array
13     ldr    r2, =sum_array
14     mov    r3, #0
15 loop:    cmp    r3, #8
16     bge    stop
17     ldr    r4, [r0]        @load content in r0
18     ldr    r5, [r1]        @load content in r1
19     add    r4, r4, r5
20     str    r4, [r2]
21
22     add    r0, r0, #4
23     add    r1, r1, #4
24     add    r2, r2, #4
25     add    r3, r3, #1
26
27     b     loop
28 stop:    nop
29     b     stop
30
31
```

Figure 1.14: Lab part 3 source code

You are required to assemble and link this source file, and run the executable file.

1.5 Deliverable for Part 3:

- Description of the program
- Program Listing
- Snap shots of the data section before and after execution of the program
- The contents of the memory and r3, r4, r5, and r0 registers at the stop breakpoint
- Change the program in such a way that it performs its operation for 32 elements. Is there any limitation in the provided code?

IMPORTANT: Copy your workspace folder from the **C:** drive into the **M:** drive for safe-keeping.

1.6 Summary:

In this lab we have described the steps in the translation process and introduced the basic functions provided by the **Eclipse** Integrated Development Environment (IDE).

The assembler translates an assembly language program into object code. The linker transforms an object code file into an executable program.

Figure 1.15 summarizes the steps used to translate an assembly language program into an executable program.

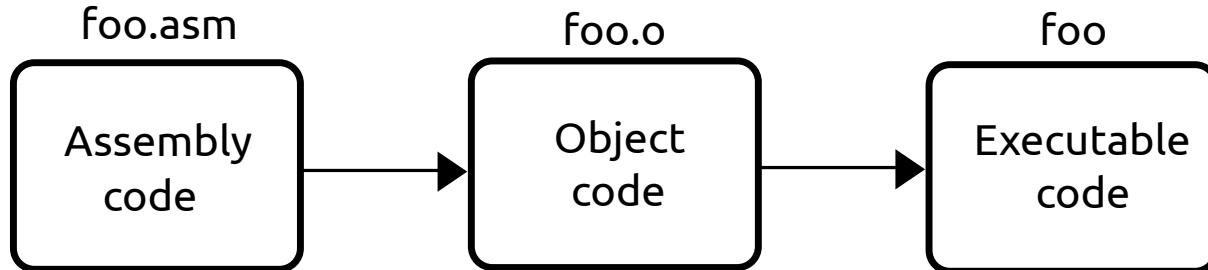


Figure 1.15: From Assembly to Executable