CT Lab: Introduction to Assembly

1 Introduction

In this lab you will work for the first time with assembly programs. You will learn the possibilities of the remote debugger to visualize and change the memory and register content. You will trace the influence of several data transfer commands and their addressing modes with the remote debugger.

2 Learning Objectives

- You can assemble, link, upload and execute an assembly program on the target hardware.
- You know how to use the remote debugger, to visualize and change the content of memory, registers and ports.
- You understand the different addressing modes in simple programs and are able to apply them.

3 Assembling and Loading an Assembly Program

In this chapter you'll learn how to assemble and link a program for the CT Board and how to upload it to the target hardware.

> *Assembling* means to translate the text based source code (coded in assembly language) into op codes used by the target hardware. This process is done by an assembler. The reverse process, translating op code to assembly language, is called *disassembling*.

Open the Assembly Project

Download the given project frame (student_projects.zip) from https://moodle.zhaw.ch. Open the project with uVision.

Assembling and Linking

To assemble and link the project, use the *rebuild* button in uVision. The result of the build process is shown in the output window.

Along with the object files the assembler also creates a so called assembly list file (***transbf.lst***). It is located in the *build* directory. This file also contains the source code in assembly syntax (right column) and a column with line numbers, the addresses of the op codes and the op code (both in hexadecimal notation). The file can be opened with uVision or any text editor such as Notepad++.

3.1 Task 1

Which op codes are generated by the assembler for the following assembly instructions? Search for the corresponding op code in the list file and use the disassembly table to decode the hexadecimal values. Fill in the gaps in the following table based on the example.

<table>
<tr><th>Assembly Code</th><th>Op Code (Hex)</th></tr>
<tr><td rowspan="2">Example
MOVS R1, #0xfe</td><td>0x21FE (from the list file)</td></tr>
<tr><td>Bit 15 … Bit 0
0 0 1 0 0 | 0 0 1 | 1 1 1 1 1 1 1 0
MOVS | R1 | imm8</td></tr>
<tr><td rowspan="2">MOVS R2, #MY_CONST</td><td></td></tr>
<tr><td>Bit 15 … Bit 0
0 0 1 0 0 0 1 0 0 0 0 1 0 0 1 0</td></tr>
<tr><td rowspan="2">MOV R11, R2</td><td></td></tr>
<tr><td>Bit 15 … Bit 0
0 1 0 0 0 1 1 0 1 0 0 1 0 0 1 1</td></tr>
<tr><td rowspan="2">LDR R0, [R7]</td><td></td></tr>
<tr><td>Bit 15 … Bit 0
0 1 1 0 1 0 0 0 0 0 1 1 1 0 0 0</td></tr>
<tr><td rowspan="2">STR R3, [R7,R6]</td><td></td></tr>
<tr><td>Bit 15 … Bit 0
0 1 0 1 0 0 0 1 1 0 1 1 1 0 1 1</td></tr>
</table>

3.2 Task 2

The given program is split into three sections (`AREA`). What are the three sections and what are their properties?

1. MyAsmVar, DATA, READWRITE
2. MyAsmConst, DATA, READONLY
3. MyCode, CODE, READONLY

3.3 Task 3

How many bytes does each section contain?

1. MyAsmVar: 16 Byte
2. MyAsmConst: 20 Byte
3. MyCode: 20 Byte + 36*2 Byte = 94 Byte

(36 Code Zeilen)

After assembling, each section begins at the address **`0x0000'0000`**. The physical addresses are assigned during the linking process.

Uploading onto the Target Hardware

Switch on the target hardware. Ensure that the USB connection on the left side of the target hardware is connected to the host computer.

Start the debugger. The program now gets uploaded into the flash memory of the target hardware and halted at the first instruction in the code section.

> **Caution**: Don't press **"Run (F5)"**, the programm has to be halted for the following manipulations.

4 Memory Content on Target System after Loading

While the ***lst*** files contain the listing of the assembly translation of each module, the project's ***map*** file contains the listing of the linker actions and symbol resolution in the final executable.

4.1 Code Section

Before we run the program we want to take a look at the memory content on the target hardware. We want to see where exactly the program has been loaded.

Memory View

In the right bottom corner of uVision you'll see the call stack or alternatively the memory view (See Figure 1). If this is not the case, go to *View → Memory Windows* and activate *Memory 1*. Now you should see it in the main window.

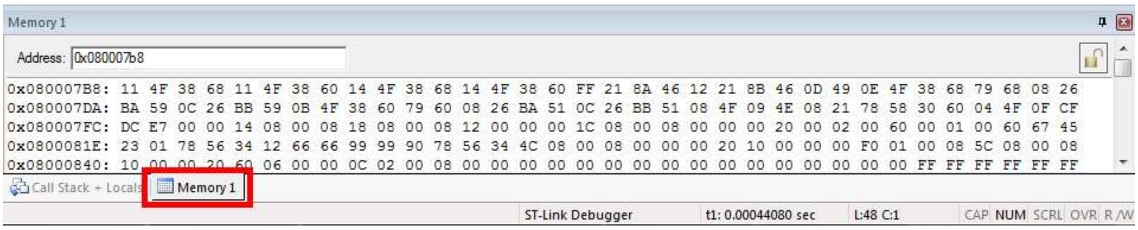


Figure 1 uVision Memory View

4.1.1 Task 4

Open the ***transbf.map*** file in uVision or some text editor and search for the symbol ***MyCode***. The associated address needs to be rounded to an even value.

Click on *Memory 1* and enter the start address of the ***main*** function code section in the field *Address*. Can you locate the op codes from the list file in memory?

Load Address

By default the remote debugger loads the program at address **`0x0800'0000`**. At this address the flash memory is located on the target hardware. The memory space from here to the beginning of the ***main*** function contains the initialization code for the microcontroller.

4.1.2 Task 5 – Deviations

Compared to the list file the order of the two op code bytes is reversed. What could be the reason?

little endian

....

Some bytes will be defined after the creation of the list file (by the linker). At these positions the memory content differs from the list file. Find corresponding bytes! What could be the reason?

Es sind relative Addressen. Der Linker berechnet die absolute Adresse.

....

....

....

4.2 Data Section (Read only)

Definition of Variables

The sample program defines several global variables in the data section. The assembler directives **`DCD`**, **`DCW`** and **`DCB`** reserve memory.

An assembler directive is a directive of the programmer for the translation program, the assembler. The assembler directive does not get translated into executable op codes, i.e. there is no corresponding op code.

4.2.1 Task 6 – Memory View

The debugger will allocate the storage region of the read-only data section right after the read-only code section. The starting address of this section is given by the first constant in the read-only data section. I.e. the first constant is at label ***addr_dip_switch***. Search this symbol in the **transbf.map** file for the start address of the respective memory area and fill in the following table with the values and start addresses of the given constants.

Variable name	Content	Start address
addr_dip_switch	0x60 00 02 00	0x08000f40
const_table[0]	0x01 23 45 67	0x08000f44
const_table[1]	0x12 34 56 78	0x08000f48
const_table[2]	0x99 99 66 66	0x08000f4C
const_table[3]	0x34 56 78 90	0x08000f50

4.3 Data Section (Read Write)

Definition of Variables

The sample program defines a global variable in the read-write data section (RAM). The assembler directive `SPACE` reserves memory space and fills it with zero.

Memory View

The read-write data section of the CT Board begins in the RAM at address `0x2000’0000`. The stack and the heap section are inserted after this section. You find the respective information in the **transbf.map** file by searching for the symbol of the first variable of the read-write data section.

5 Function of the Program

The given program **transbf.s** demonstrates different commands, used to load and store values. It shows how constants are defined and loaded, and how the load and store commands are used.

5.1 Task 7

Study the code in the list file. What are the results of the indicated instructions? Fill in the following table with the expected values of the target registers after the corresponding line of code has been executed.

Line	**Instruction**	**Content of target register**
*** A1 ***	MOVS R1, #0xfe	0x000000fe
*** A2 ***	MOV R11, R2	0x00000012
*** A3 ***	LDR R3, =ADDR_DIP_SWITCH_31_0	0x60000200
*** A4 ***	LDR R7, addr_dip_switch	0x60000200
*** A5 ***	LDR R7, =addr_dip_switch	0x08000F40
*** A6 ***	LDR R1, [R7, #4]	0x12345678
*** A7 ***	LDR R3, [R7, R6]	0x34567890

5.2 Task 8

Execute the program step by step. Check your values in the table. Be aware of the differences between the two lines marked with A4 and A5 (LDR as literal and as pseudo instruction). Do they meet your expectations?

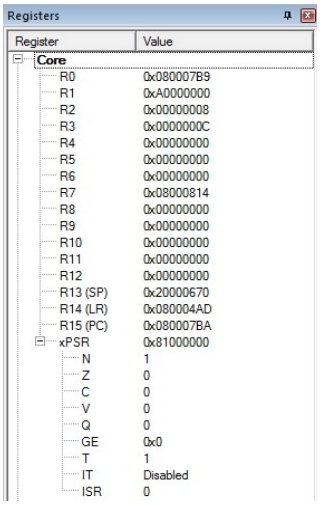
6 Altering the Processor State (Optional)

With the debugger you are not only able to observe the processor state; you are also able to alter it directly. This enables you to comfortably debug your program without changing it.

Altering Memory Content

Within the *Memory 1* window you can directly alter the content of the memory (RAM). Input the address `0x2000'0000` into the address field of the window. This is the start of the RAM section. A double click on a particular value lets you alter its content. If you cannot alter the content, make sure the lock in the top right of the *Memory 1* windows is open. If it's closed you can open it with a left click.

Altering Register Content



The current content of the processor registers can be observed in the upper left part of the main window, as well as the processor state (xPSR, Processor Status Register) with its flags.

You can change the content of these registers as well as the flags in the xPSR with a double click on the corresponding register.

6.1 Task 9 – Customize the Output (optional)

Now change the variable *store_table* in the RAM in such a way, that with the execution of line 101 every second LED on the CT Board is bright. Which memory cell do you need to modify?

0x2000008 = 0xAAAAAAAA
....

....

....

7 Grading

Task	Criteria	Weight
3	The tables are filled in correctly and the questions are answered. You can explain your reasoning.	1/4
4		1/4
5.1		1/4
5.2	You briefly explain how you used the debugger and can answer questions about it.	1/4