



2B1445 Embedded systems

Laboratory 1 – Introduction to ARM¹

1 Introduction

In this lab we introduce the ARM software development toolkit (ARM SDT), the ARM processor simulator and the ARM debugger. You will also get some familiarity of ARM assembly language programming.

The goals of the laboratory exercise are

- to gain familiarity with the program development environment and the ARMulator, the ARM simulator environment
- to practice the ARM instruction set and assembler level programming

2 Preparation Tasks

Read pages 57-82 in the text book. Solve the following ARM related exercises in the course book: 2-3, 2-6, 2-7, 2-10, 2-12.

Read through the entire lab manual and then solve the homework problems before coming to the lab. Your task will be easier if you have your programs in a computer file, either using the lab computers or using a diskette (or CD) that you bring to the lab.

3 Lab room

The labs will take place in the Forum-lab, which can be found on level 8 (*elevator B*) in the Forum building, Isafjordsgatan 39. Use your access card to gain entrance to the lab. The lab is open day and night for your preparation work, but may often be in use for labs at day-time (e.g. Embedded Systems labs). Check the schedule on the door.

¹ Edited for the course year 2004/2005 by Ingo Sander based on an earlier version of Mats Brorsson

4 ARM Overview

The ARM processor architecture is described in the course book. Here we outline some details that are not mentioned in the textbook but which are needed in order to solve the lab problems.

4.1 Registers

The ARM processor architecture uses 16 registers:

- Registers 0-12 are general purpose registers and can be used for any purpose.
- Register 13 is typically used as the stack pointer (more about this later).
- Register 14 is used to store the return address at function calls.
- Register 15 is the program counter. It holds the address of the next instruction to be fetched for execution.

There is also a status register: CPSR. It contains, among other things, the following important status bits:

- N – Negative, set when the result of an arithmetic operation is negative (according to the two's complement encoding of the natural numbers).
- Z – Zero, set when the result of an operation is zero.
- C – Carry, set when an operation results in a carry bit from the most significant position.
- V – oVerflow, set when an operation results in arithmetic error.
- I – IRQ, interrupt enable/disable.
- F – FIQ, fast interrupt enable/disable.
- T – Thumb, enables Thumb mode instruction set².

The CPSR also shows in what mode the processor is currently executing. Some of the available modes are:

- USR – User mode.
- FIQ – Fast interrupt mode.
- IRQ – Interrupt mode.
- SVC – Supervisor mode.
- ABT – Abort mode.
- UND – Undefined mode.

For a more thorough discussion about processor modes and their meaning we refer to one of the ARM manuals available through the link section on the course web site. In the course we will deal only with the User, Interrupt and Supervisor modes.

² The Thumb mode is a special 16-bit instruction set which is a true subset of the ordinary ARM instruction set.

4.2 Some important instructions and directives

The example program used in the homework uses a few instructions and directives not mentioned in the textbook. They are explained here.

```
AREA lab1_text, CODE
ENTRY
```

The line, which starts with `AREA` defines a new section in the program with a name `lab1_text` and with an attribute `CODE` specifying that this section contains instructions (and not data). The second line with only the word `ENTRY` specifies that the next instruction is the entry point of the program, i.e., the start address.

```
MSR CPSR_f, #0
```

This line is an instruction to move (actually copy) a register contents or, as in this case, an immediate constant to the Current Processor Status Register (CPSR). The `_f` specifies that it should only apply to the flags bit field. Effectively, this instruction clears the flags bit field.

Close to the end in the program we find:

```
stop  MOV r0, #0x18
      LDR r1, =0x20026
      SWI 0x123456
```

These instructions perform together the functionality of notifying the operating system that the program has reached its end and that execution should halt. The actual functionality of each and every of these instructions will be explored further in lab 3.

At the very end we find:

```
AREA my_data_area, DATA

my_data    DCD    1, 2, 0
```

Again, the `AREA` directive specifies a new section. This time a data section with the name `my_data_area`. The `DCD` directive reserves space in memory for the data 1, 2 and 0. One word each is reserved. The label `my_data` can be used to access the data 1. Data 2 and 0 are on addresses `my_data+4` and `my_data+8` respectively.

It is strictly speaking not necessary to have a separate section for the data. They could have been declared in the code section. However, the code section is read-only by default and a data section is read/write by default so if the program were to be executed on a version of ARM with memory management, then it might not work if read/write data resides in a code section.

5 Homework tasks

5.1 Homework 1

Make sure that you understand what the following program should do and how it works. Make sure you understand which lines are instructions and which are directives. Perform a mental execution of the program and try to understand how registers are modified by this code. The source code (Lab1_HW1.s) is available on the course homepage.

```
        AREA lab1_text, CODE
        ENTRY
start

init    MOV R0, #0
        MOV R1, #0
        MOV R2, #0
        MOV R3, #0
        MSR CPSR_f, #0

ex1     MOV R0, #3
        MOV R1, #2
        ADDS R2, R0, R1

        MOV R0, #0
        MOV R1, #0
        MOV R2, #0
        MSR CPSR_f, #0

ex2     MOV R0, #4
        MOV R1, #3
        SUBS R2, R0, R1
        SUBS R3, R1, R0

        MOV R0, #0
        MOV R1, #0
        MOV R2, #0
        MOV R3, #0
        MSR CPSR_f, #0

ex3     MOV R0, #1
        MOV R1, #2
        CMP R0, R1
        BLE smaller
greater MOV R2, #2
        B init4
smaller MOV R2, #1

init4   MOV R0, #0
```

```

        MOV R1, #0
        MOV R2, #0
        MSR CPSR_f,#0

ex4     MOV R0,#3
        MOV R1,#1
loop    SUBS R0,R0,R1
        BGE loop

        MOV R0, #0
        MOV R1, #0
        MSR CPSR_f,#0

ex5     ADR R3,my_data
        LDR R0,[R3]
        LDR R1,[R3,#4]!
        ADD R2,R1,R0
        STR R2,[R3,#4]

stop    MOV r0,#0x18
        LDR r1,=0x20026
        SWI 0x123456

        AREA my_data_area, DATA

my_data DCD 1,2,0

        END

```

5.2 Homework 2

Make sure that you understand the following program. The source code (Lab1_HW2.c) is available on the course homepage.

```

#include <stdio.h>

main()
{
    int a, b, c;
    printf("hello, world\n");

    a = 3;
    b = 4;
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
}

```

6 The ARM Project Manager

The ARM Project Manager (APM) is a graphical development tool that automates some of the routine operations of managing source files and building your software development project. APM uses the

concept of a *project* to maintain information of the system you are building. You specify what to build and how to build it. When you have described your system as a project, you can build all of it or just the parts that are needed. If the project output is an executable file, you can execute it or debug it using the ARM debugger.

6.1 Starting APM

This description assumes that you are working on a Windows PC. If the program has been properly installed, you should be able to start the program from the Start button. Click “Start -> Programs -> ARM SDT 2.51 -> ARM Project Manager” and the program should start.

6.2 Creating a new project

To create a new project in the APM you select “File -> New” and then you select “New project”. Make sure that the project is of type ARM Executable Image. Also select a name and a directory to put your project and click on OK when you are done. Remember to put your project in a directory which is on a network drive so that you can access it later on a different computer.

When the project has been created, you can add files to it. In this lab we will only add source code files, but a number of other files can also be associated with a project. The extension `.s` indicates that the file contains assembler code and the extension `.c` indicates that it is a file with C-code. Add a file from menu “Project -> Add Files to Project”.

When you have added your file(s) and edited them, the project is ready to be built. The default project template sets up three different targets: DebugRel, Debug and Release. The DebugRel and Debug targets have debugging information which is essential for running the program through the debugger. The DebugRel-variant also adds some level of optimisation and the Release variant has an even higher degree of optimisation so that debugging is not possible. Optimisations are not relevant for assembler files. The default target is DebugRel. If this is not selected automatically, select it and press the button marked “Build DebugRel”.

7 The ARM Debugger (for Windows) ADW

The ARM Debugger enables you to debug your ARM-targeted program. It can debug programs running on a real hardware platform or using the built-in ARM simulator called ARMulator. This is the way we will use it for most of the labs.

7.1 Starting the debugger

If you developed your program using APM, you should be able to start the debugger simply by pressing the key F5. This will start the debugger and automatically transfer the executable image to the debugger. Otherwise, you can start the debugger from “Start -> Programs -> ARM SDT 2.51 -> ARM Debugger for Windows”.

7.2 Load image

If you started the debugger from APM the image should have been loaded in the debugger automatically. If not, to load an image do “File -> Load Image” and then find the correct image file. It should end with the suffix `.axf`.

If you at some point want to execute your program more than once, the image needs to be reloaded between the executions. This is done with the command “File -> Reload current image”.

7.3 Breakpoints

Breakpoints are used to halt the execution at a certain instruction line in the code. When you set a breakpoint it is marked in red in the left pane of the breakpoints window. There are two methods you can use to set a breakpoint. You can double click on the line where you want to set the breakpoint and then press OK. Or you can select the desired breakpoint line and then press F9 or use the Execute menu.

7.4 Views

In the ARM debugger you can choose between several views (windows) to display what is happening during the execution. The available windows are all listed in the View menu in the ARM debugger. Here we will describe the ones that might be useful for the lab.

7.4.1 Execution window

The *execution window* displays the source code of the program that is currently executing. It is used to execute the entire program at once or to step through it, line by line. The execution window is also used to set, edit, or remove breakpoints.

7.4.2 Console window

The *console window* allows you to interact with the executing program. Anything printed by the program, for example a prompt for user input, is displayed on this window and any input required by the program must be entered here.

7.4.3 Disassembly window

The *disassembly window* displays disassembled code interpreted from a specified area of the memory. Memory addresses are listed in the left-hand pane and disassembled code is displayed in the right-hand pane. The disassembly window can, for example, be used to set, edit, or remove breakpoints.

7.4.4 Low Level Symbols windows

The *low level symbols window* displays a list of all low-level symbols in your program. This window can be used to display the memory or the source/disassembled code pointed to by the selected symbol. The low level symbols window can also be used to set, edit or remove breakpoints.

7.4.5 Memory window

The *memory window* displays the contents of the memory in a specified address area. Addresses are listed in the left-hand pane and the memory content is displayed on the right-hand pane. The memory window can be used to change the contents of memory (just double click on a line).

7.4.6 Registers window

The *registers window* displays the registers and their contents. You can double click on an item to modify the value in the register. The register window can be used to display the memory pointed to by the selected register. It can also be used to edit the contents of a register.

8 Lab exercises

Work your way through these exercises and contact the lab assistant when you have any question or when you have finished to discuss the result.

8.1 Executing simple ARM assembler code

Create a new project in APM and add a file that contains homework 1. Build the DebugRel target and press F5 to load the executable image in the debugger. Configure the debugger in the following way:

- Select “Options -> Configure Debugger...”.
- Make sure Target Environment is Armulate. Press Configure and select ARM9TDMI as variant, set emulated speed to 100 MHz,
- Select tab Debugger and make sure Endian is set to Little. Then disable Remote Startup warning.

This configuration needs to be done only the first time and when you want to modify it. If you move to a different computer, you might need to check the configuration again.

When the program has been loaded and you have selected your views (the default is probably good enough), you can execute the program. Although it is possible to execute the entire program all-at-once, this is not meaningful as the exercise is about looking at single instruction execution. Therefore, we will use stepwise execution, which can be done by pressing F10. If you want to re-execute a certain exercise but without stepping through the previous ones, you can use breakpoints and execute to the breakpoint by pressing F5.

You are also encouraged to experiment with the debugger while you are performing the exercises.

For exercises ex1 to ex5 the enclosed tables should be filled in. After executing a line of code, fill in the values in the various registers and the values of the CPSR status bits. A CPSR status bit is set (1) when the corresponding letter is capitalized, otherwise it is zero (0). The register fields should be filled in with the decimal value and not the hexadecimal value, unless it is an address. For each exercise there are also a couple of questions that need to be answered.

Please note, that after each exercise reset operations are executed. I.e. all used registers are cleared and the status bits are reset. These operations no *not* need to monitored. But there is no harm in doing it.

8.1.1 Ex1

Exercise 1 is an addition. Before the addition operation is performed, the two terms are loaded into registers.

Instruction	CPSR				Registers		
	N	Z	C	V	R0 (dec)	R1 (dec)	R2 (dec)

Questions:

In which register is the result stored?

What is the difference between the instructions ADD and ADDS?

8.1.2 Ex2

Exercise 2 is a subtraction. The two terms are loaded into registers, then two different subtract operations are performed.

Instruction	CPSR				Registers			
	N	Z	C	V	R0 (dec)	R1 (dec)	R2 (dec)	R3(dec)

Questions:

In what order are the registers subtracted?

How is a subtraction executed?

Why is the carry flag set after the first subtraction?

Which flag is set to indicate that the result of an operation is negative?

8.1.3 Ex3

In exercise 3, two operands are compared and based on the comparison, one of two possible branches is executed.

Instruction	CPSR				Registers		
	N	Z	C	V	R0 (dec)	R1 (dec)	R2 (dec)

Questions:

Which branch is executed and why?

How do you think a compare action is conducted?

Which flag does the branch operation, BLE, use?

What values should the flags have in order for the BLE instruction to cause a branch jump? Verify that with an experiment.

8.1.4 Ex4

Exercise 4 illustrates how a loop can be implemented and executed in ARM.

Instruction	CPSR				Registers	
	N	Z	C	V	R0 (dec)	R1 (dec)

Questions:

Which register serves as the loop counter?

How many times is the loop entered?

Which flags does the branch operation, BGE, use?

What values should the flags have in order for the BGE instruction to cause a branch jump?

8.1.5 Ex5

In this exercise we will try to load and store values to and from memory. Memory space for our data is allocated at the label `my_data`. Three variables are defined and initialized.

Use the Low Level Symbols window to find out at what address, space for `my_data` is allocated. In the Memory window changes in the memory during execution can be monitored.

In this assignment, you should fill in the register fields with their hexadecimal values since they are addresses.

Instruction	CPSR				Registers			
	N	Z	C	V	R0 (hex)	R1 (hex)	R2 (hex)	R3 (hex)

Questions:

At what memory address is `my_data`?

What is register R3 used as?

What is the address difference between two sequentially stored values? Why?

At what memory address is the result of the operation stored?

What does the `!` symbol do? (Tip, look at register R3 before and after the LDR with `!` operation, compare with the STR without `!` operation).

8.2 A C program

In this exercise you will see how you can generate executable ARM code from a C source code file. The ARM instructions will be generated by the ARM tools and in the end you can execute your image in the ARM debugger.

Create and build a new project with your C-code according to homework assignment 2. Then use the ARM debugger to execute your program. In order to see the output of the program, you need to open the Console window if it is not already open.

Press F5 to start execution. It will stop at an automatic breakpoint at the beginning of the main function. Open the disassemble window (“View -> Disassembly”) and see if you recognize anything from the C file. Then continue execution line by line using F10. Observe the console output.

9 Conclusions

Before you pass the laboratory exercise, make sure you understand the questions above and that you have an answer to them. Think also about the questions below and then contact the lab assistant.

- Where are the programs you have used executed?
- How can the computer jump to a symbolic label, such as **BGE loop**?
- Which registers are affected by a branch instruction?
- How many bytes are used to store one instruction?
- Can the computer execute an assembly instruction?
- How does the computer know that a bit pattern is an instruction?

2B1445, Embedded Systems

Lab registration form

Name _____

Lab 1 Completed date: _____, Lab assistant signature: _____

Lab 2 Completed date: _____, Lab assistant signature: _____

Lab 3 Completed date: _____, Lab assistant signature: _____

Lab 4 Completed date: _____, Lab assistant signature: _____

Lab 5 Completed date: _____, Lab assistant signature: _____

N.B. Keep this page *safe* and *keep it* until you know that the result of your course has been registered in Ladok (Database system for Swedish university transcripts). The purpose of the page is to provide extra security for your sake.