TDT4258 Low-Level Programming
HS 2022

Lab assignment 1

ARM Assembly-Language Programming

Deadline: Fri 09 Sept 2021, 23:59

Teaching Assistants: Pavel Skipenes, Vetle Harnes
Assignment Coordinator: Roman K. Brunner
Lecturer: Björn Gottschall

Pre-amble

The labs are here for you to deepen your understanding of concepts taught in the lecture. The goal is that you not only develop a theoretical understanding of the matter, but also develop the technical skills to apply it in practice.

Each lab has a **main project**, but we also provide optional exercises for those who want to go beyond the mandatory exercise. To pass a lab, you only need to **hand in the solution to the main project**. The optional exercises are purely for your entertainment. Some of the optional tasks are easier than the main task; some are harder. We indicate the difficulty at the beginning of the problem description. The easier ones can serve as entry points, if you feel that you are not yet ready to tackle the main task. But remember that in the end all that counts is solving the main task, as the optional tasks do not count towards the pass/fail decision.

We assess the lab assignments on a **pass/fail** basis. To be allowed to sit in the exam, you have to pass all three labs. As the assignments are part of the evaluation, they are subject to NTNU's plagiarism rules [1]. We have tools at our disposal and will run all submissions through plagiarism checkers. Copying code from current or past students is considered plagiarism. Hence, we advise only sharing code after the deadline has passed to prevent situations where we have to find out who copied from whom.

While copying each other is disallowed, we still encourage student discussions about your solutions. This will allow you to explore alternative approaches and solutions and learn about the advantages and challenges of particular implementations.

1 Description

This is the first lab out of three for this course. The aim of this lab is to expose you to ARM assembly programming. The program you write will communicate with different input/output devices. You will use CPUlator, an online service, to test your programs.

2 Program development and testing framework

For this assignment, you will use CPUlator, an online service that provides a hardware simulator, editor, and debugger. CPUlator runs directly in a web

[1] https://i.ntnu.no/wiki/-/wiki/English/Cheating+on+exams

browser, so you don't need to install any additional software. You can access it under

https://cpulator.01xz.net/

CPUlator allows you to simulate different ISAs and systems. The detailed documentation on CPUlator is available at:

https://cpulator.01xz.net/doc/

You will configure it to simulate ARMv7 ISA with ARMv7 DE1-SoC system[2]. The documentation for DE1-SoC is available on Blackboard[3]. Note that DE1-SoC is actual hardware with different input/output devices, and CPUlator does not simulate all of them. For this lab, we are only interested in red LEDs and JTAG UART (Universal Asynchronous Receiver-Transmitter), both of which are simulated by CPUlator. If you are interested, refer to the CPUlator documentation to check which DE1-SoC devices are not simulated.

Controlling red LEDs: CPUlator simulates ten red LEDs from DE1-SoC. These LEDs can be turned on and off by controlling the values in a peripheral register at a specific address. The register contains one bit per LED. If you set any of those bits to one, the corresponding LED turns on; otherwise, it is off. You can write to the register by using an STR instruction. You find the register's address in the DE1-SoC manual[3].

Controlling JTAG UART: On DE1-SoC, JTAG UART can be used to transfer data between the host computer and the programs running on ARM cores of DE1-SoC. However, CPUlator simulates this communication with host computer by writing to and reading from a JTAG UART box in simulator window. Like the LEDs, the JTAG UART box has its own address, which you find in the DE1-SoC manual[3]. If you want to write to JTAG UART, you need to write one character at a time to this address, and the written text will appear in JTAG UART box.

2.1 Running a program in CPUlator

To help you get started with CPUlator, we have provided a sample ARM assembly program `test.s`[4]. You can run this program in CPUlator to understand the simulation process and to get familiar with the CPUlator features.

- You can open the test.s file by choosing "open" from "File" drop down menu. Al- ternatively, you can also type in your program in the Editor window.

[2]https://cpulator.01xz.net/?sys=arm-de1soc

[3]Blackboard →Labs →Lab 1 →DE1 SOC Manual.pdf

[4]Blackboard →Labs →Lab 1 →test.s

- Once you are finished writing the program, hit "Compile and Load(F5)". It will compile/assemble your program and any warnings or error messages will appear in "Messages" window at the bottom of the screen. If the compilation is successful, "Compile succeeded" will appear in this window. In addition to compilation, your program will be loaded into the memory and will be ready for execution.

- To execute the program in one go, hit "Continue". It will execute your program, and you can inspect the state of registers (window on the left), memory (one of the tabs in the middle window), and IO devices (window on the right).

- Instead of executing your program at once, you can single step through it by hitting "Step Into". This will execute only one instruction at a time, thus you can inspect the state of registers, memory, and IO devices after every single instruction. This feature is useful for debugging your programs.

For more details on CPUlator features, refer to its documentation.

How to terminate an assembly program: Notice that the last instruction in `test.s` is an unconditional branch "b" that jumps to itself. This, effectively, takes the program into an infinite loop. In its absence, the program would have continued to execute sequentially, treating the data in `.data` section or whatever comes afterwords as instructions until encountering an error condition. It can overwrite the results generated by the legitimate code you want to execute. To avoid that, we put the program in an infinite loop so that you can inspect the results while your program is spinning in the loop.

3 Main Task: Palindrome Finder

You have to write a palindrome finder in ARM assembly that determines whether or not a given input is a palindrome. A palindrome is a number, word, phrase, sentence, or another sequence of characters that reads the same backward or forward. A palindrome is also case-insensitive, and spaces are ignored. For the purpose of this assignment, the input is restricted only to contain alpha-numeric characters and spaces (any combination) without punctuation, special characters, or umlauts. For example, 'ad8dF90' and 'e082 2F01' are valid inputs as they are a sequence of characters and/or spaces, but they are not palindromes.

Some rules to follow on palindromes and inputs:

- The valid characters are as follows: 'a-z', 'A-Z', '0-9' and ' ' (space). Special characters will not be used in test inputs. So it doesn't matter whether your code handles them or not
- A valid palindrome can only be a single word, sentence (words separated by spaces), numbers, or alphanumeric. Examples of valid palindromes: "level", "8448", "step on no pets", "My gym", "Was it a car or a cat I saw". Examples of strings that are not a palindrome: "Palindrome", "First level"
- The shortest palindrome is at least two characters long
- Palindromes are case insensitive, so "KayAk" and "A9c9a" are valid parameters

A good approach to writing an ARM assembly program is to first come up with a high-level algorithmic solution or implementation (C, C++, Java etc.). It is much easier and quicker to correct all the control flow and data manipulations in a high-level representation than in assembly. Once the high-level implementation for a program is correct, you can translate it to an equivalent ARM assembly program, statement-by-statement. Before translating, you should test your high-level language program and ensure that it is working correctly before starting on the ARM code. To help you get started, an outline `palin_finder.s` is supplied. It is not required to stick to the provided structure (see Section 4 for further details). We recommend structuring the code into functions for ease of development and readability.

Note: You only need to submit the ARM assembly code. The high-level language code is just for your own reference. Please comment on your ARM assembly code appropriately. As a model for creating and commenting on your ARM code, have a look at the supplied file `test.s`.

3.1 Input to Palindrome Finder

The input to the Palindrome Finder program is a string, phrase, or sentence. The supplied `palin_finder.s` file already contains example inputs in the .data section. This is the input that is put into memory before execution starts and that you will test for a possible palindrome. Defining the input in the data section with .asciz ensures that it will be null-terminated in memory (a zero byte marks its end). Your program must accept variable width (run-time checked) inputs. We also encourage you to use several different inputs to test your program. **When you submit your code, the input label in the data section must be the same as we supply in** `palin_finder.s`

file. Your program can have undefined behavior for invalid inputs (inputs containing invalid characters outside the definition from the beginning of Section 3).

3.2 Output of Palindrome Finder

Your program should display the outcome in two different ways: 1) light up LEDs and 2) write to JTAG UART box.
Light up red LEDs: If the input word is not a palindrome, you need to light up five leftmost red LEDs; and if it is a palindrome, you should light up the five rightmost red LEDs.
Writing to JTAG UART box: If the input word is not a palindrome, the program should print the message "Not a palindrome" in the JTAG UART box; otherwise, it should print "Palindrome detected".
Refer to Section 2 to check how to light up LEDs and write to JTAG UART box.

3.3 Optional Tasks

- **[EASY]:** Write the numbers from $(0, 100($ using the JTAG UART box. The output should be calculated, not hard coded. If you try to minimize the number of executed instructions, what's the minimum you achieve?

  Sample (partial) output: 0, 1, 2, 3, . . . , 98, 99

- **[EASY]:** Given an input string (allowable characters are [0-9A-Za-z] and the simple whitespace ' '), your program should reverse the order of words (strings separated by whitespace (' ')), as well as reverse the order of characters in each word. Print the final solution using the JTAG UART.

  Sample input: "Hello World"

  Sample output: "dlroW olleH"

- **[MEDIUM]:** Your program should use the stack to store a list of integers `li` and a single integer `d`. You can assume that the list of integers is always sorted from smallest to largest integer. The goal is to find the indices of two entries in the list that sum up to `d`. If your program does find such two integers, output the two indices using the JTAG UART, otherwise output `-1, -1`.

  Sample input: `li:` $[1, 3, 4, 7, 9, 12]$, `d:` 11

  Sample output: `2, 3`

- **[HARD]:** Your program should analyze a string and find the longest palindrome in it. Note that the midpoint for a potential palindrome can lie at any position in the string. What's the most efficient implementation you can come up with in assembly?

  Samples: for "AbbaCCdccA" the solution would be "aCCdccA"

4 Submission

Submit your **commented assembly code file** `palin_finder.s` before the deadline on Blackboard. Ensure you haven't changed the input label (name of the variable) in the .data section!

5 Assessment

This assignment will be evaluated on a pass/fail basis. Your program will be judged on correctness and completeness, so please make sure that all above requirements are respected and functional.

Commenting on your code and keeping it tidy is very important. Helping us understand what you did, supports us in assessing your work – we can only give points for what we understand.

6 Similarity Checking and Plagiarism

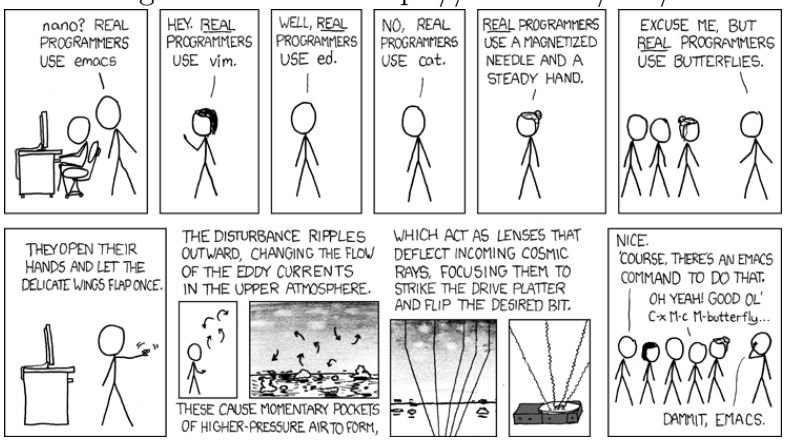
You must submit your **own work**. You must write your **own code** and **not copy** it from anywhere else, including your classmates, internet, and automated tools[5]. Failure to do so is cnosidered plagiarism. Detailed guidelines on what constitutes plagiarism can be found at:

https://innsida.ntnu.no/wiki/-/wiki/English/Cheating+on+exams

We check all submitted code for similarities to other submissions. Plagiarism detection tools have been effective in the past at finding similarities. They have gotten excellent over time, so it is inadvisable to try and outsmart them. So don't do it, not only because we will most likely catch you, but because it is morally wrong and can undermine your academic integrity, even a long time into the future. For more references, see https://www.google.com/search?q=resigns+over+plagiarism+allegations (statistics on the 8th of August: 467'000 results).

[5]This is not an exhaustive list. Don't copy code from any source

Figure 1: Source: https://xkcd.com/378/



7 Questions

If you have any questions about this assignment, we encourage you to ask the question on the course forum on Piazza. By that, you also help other students who have the same questions in the future.