TDT4258 Low-Level Programming
Autumn 2025

Lab assignment 1

ARM Assembly-Language Programming

Handout: Thursday, 21st August 2025, 17:00
Deadline: Friday, 19th September 2025, 17:00

Teaching Assistants: Nicolai Hollup Brand, Callum Gran,
Simon Haug and Joakim Borge Hunskår
Assignment Coordinator: Andreu Girones

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 collect points for a lab, you only need to **hand in the solution to the main project**.

The optional exercises are for possible extended learning and maybe even your entertainment. They can be easier or more difficult than the main task. We indicate their 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 contribute to collecting points.

The lab exercises are compulsory activities in the course. **You need to collect 27 points in total to be allowed to take the exam**. We assess the lab assignments such that a fully approved solution will get 10 points, and we hope that will be the normal case. Submissions with significant defects might get a reduced number of points. Since there will be 4 labs it will be possible to reach the exam threshold without a full score on three first labs.

The labs are subject to NTNU's plagiarism rules [1]. We run all submissions through plagiarism checkers. Copying code from current or past students is considered plagiarism. Hence, we advise you to not share code to prevent situations where we have to find out who copied from whom.

While copying each other is disallowed, we encourage student discussions about the tasks to be solved. This will allow you to explore alternative approaches and learn about the advantages and challenges of different approaches. **As a rule of thumb: Sharing and discussing ideas is fine, sharing code is not.**

1 Description

This lab aims 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. You are allowed to use generative AI for the lab if you prefer to do so. It might help you but also turn out to give difficulties. See the introduction in Lecture 0 and other information given in the course about generative AI.

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

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 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 the host computer by writing to and reading from a JTAG UART box in the simulator window. Like the LEDs, the JTAG UART box has its 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 the JTAG UART box.

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

[3]Blackboard →Labs →Lab 1 →DE1 SOC Manual.pdf (version 17.1)

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, assuming that you first have downloaded the file from Blackboard to your computer. Alternatively, you can also type in your program in the Editor window.
- 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.

[4]Blackboard →Labs →Lab 1 →test.s

3 Main Task: Palindrome Finder

You have to write a program 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 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, with two exceptions, will not be used in test inputs. The exceptions are '?' and '#' that both are wildcards that can match any single valid character except the space, such that "abc?dc#a" is a palindrome, since '?' can be 'd' and '#' can be 'b'. It doesn't matter whether your code handles the other special characters 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, Java, Python 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 `palinfinder.s` is supplied (also on Blackboard). 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 `palinfinder.s` file already contains example inputs in the .data section. This is the input that is put into memory before execution starts and 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 (runtime 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 `palinfinder.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 the JTAG UART box.
Light up red LEDs: If the input word is not a palindrome, you need to light up the 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 the JTAG UART box.

3.3 Optional Tasks

- **A [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

- **B [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"

- **C [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`

- **D [MEDIUM]:** Graphics — VGA-1 (generative AI)
  Introduction and advice
  VGA, or Video Graphics Array, is a computer display standard. It supports many different graphics modes — screens of different sizes, different color palettes and textmodes[5]. Writing a full software driver for a VGA device is a huge undertaking, but the CPUlator makes it easy to write to its VGA window via memory mapped IO.

  The purpose of this task is to train assembler programming and maybe add some color to our daily routine. **We propose that you try to use generative AI such as NTNU copilot for this task**. There will also be optional tasks in Lab 2 and Lab 3 that are follow ups of this task.

  It is easy to get code from copilot to demonstrate the CPUlator VGA with ARM assembler-code, but rather often you get code and explanation that might look convincing, but when testing it, it crashes or nothing happens in the VGA window. It is rather difficult to debug assembly code that you have not written yourself! So we give some initial help:

  – The CPUlator VGA screen is 320 pixels wide and 240 pixels high. (0,0) is top left corner. A pixel is stored as 2 bytes, ie. 16 bits, in

[5] https://en.wikipedia.org/wiki/Video_Graphics_Array

RGB format as Red-value in bits 15-11, Green-value in bits 10-5 and Blue-value in bits 4-0.

- We set a pixel by writing a color value to the two bytes that store the color of that pixel. The address is calculated by adding an offset to the VGA base address shown on top right of the CPUlator window (VGA pixel buffer). Pixels are stored row-wise. 320 x 2 bytes is needed for one row of pixels, but one row fills 1024 bytes. The offset of the pixel at (x,y) is therefore y*1024 + x*2.
- Start with the simplest task (a), and set one pixel to a given color. Debug, single-step, check register values until it works. It is easier to see an actual pixel value by magnifying the VGA display window: Select 2.0 as zoom factor in the drop down menu to the left of the window title (VGA pixel buffer).
- If you want a kick-start, try to run the minimalistic example program `VGAmini2.s`[6]. Trust this code more than copilot. See also more hints below.

Sub-tasks

(a) Write four pixels in the upper left corner with four distinct colors.

(b) Implement a function BlankScreen that sets the entire screen to white.

(c) Write a function BigPixel(x, y, c) where x and y are in the range 0..7 and c is the color. Each big pixel is made up of 100 VGA pixels as a 10x10 square. (0,0) is in the upper left corner of the VGA display, (1,0) starts at position (10,0), (0,1) starts at postion (0,10) and (7,7) starts at (70,70).

Hints on using generative AI

(i) Read the code and the explanation from the copilot and try to understand it before starting to test and debug.

(ii) You can ask copilot to explain single instructions or small pieces of code. (You will typically see more ARM assembler instructions than you have seen in the textbook or lectures, but you can ask it to explain them and use CPUlator to study their effects).

(iii) Be specific in your prompts in general, and especially in describing the use of registers for parameters. Remember to save/restore

[6]Blackboard →Labs →Lab 1 →VGAmini2.s

registers on the stack, including the link register LR, when using nested functions, and follow the ARM register convention on function calls presented in lecture 2.

(iv) In general, copilots are considered to be best at programming language syntax and library semantics, and we will experience that later in the course when asking copilot to help with C or python. However in this task you might experience that syntax is a challenge. The reason is probably that CPUlator specifics have been a rather small part of the training data for the LLM (Large Language Model) used in copilot, compared to C and python. The copilot seems to need more help for the specific CPUlator ARM assembler. For example, we have experienced that you might need to describe how comments are marked, what is the exact format of EQU pragmas (for specifying constants), the correct address of the pixel buffer etc.

(v) It might be productive to give copilot code that already works as part of your prompt to specify the task, eg. parts of or the whole example `VGAmini2.s`.

4 Submission and Assessment

We expect all submissions to meet the following requirements:

- The submission is on time. We provide enough time to solve the labs, but we expect you to start early.
- You submit *a single commented assembly file* `palinfinder.s`
- The variables in the `.data` area have not changed, renamed or commented out.
- The submitted code runs on the provided samples without errors.
- You have submitted an *"AI-statement"* for this lab exercise. It will be anonymized and will not influence on the assessment — but will help us all to gather experience with the use of generative AI in the course. This is further explained in lecture 0 and on blackboard.

Failing to meet any of these requirements will result in the number of points collected by the submission to be reduced, depending on the seriousness of the defect(s). No submission gives zero points.

Commenting 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.

4.1 Evaluation Points

This is the primary evaluation criteria that will be considered for a submitted solution:

- `input` **Variable Name**: The code should respect the specified `input` variable name.
- **Timely Submission**: The solution must be submitted on or before the specified deadline.
- **AI Declaration**: A declaration stating the use (or non-use) of AI tools must be submitted with the code.
- **Input Length Validation**: The code must verify that the `input` string is at least two characters long.
- **Palindrome Detection**: The code must correctly identify both palindromes and non-palindromes for inputs containing only valid characters.
- **Palindrome Detection with Wildcards**: The code must correctly identify both palindromes and non-palindromes for inputs that include valid characters and wildcards.

5 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 or any other sources. We check all submitted code for similarities to other submissions using automated tools.

6 Questions

If you have any questions about this assignment, we encourage you to post it on the discussion forum provided by the course. By that, you also help other students who have the same questions in the future.

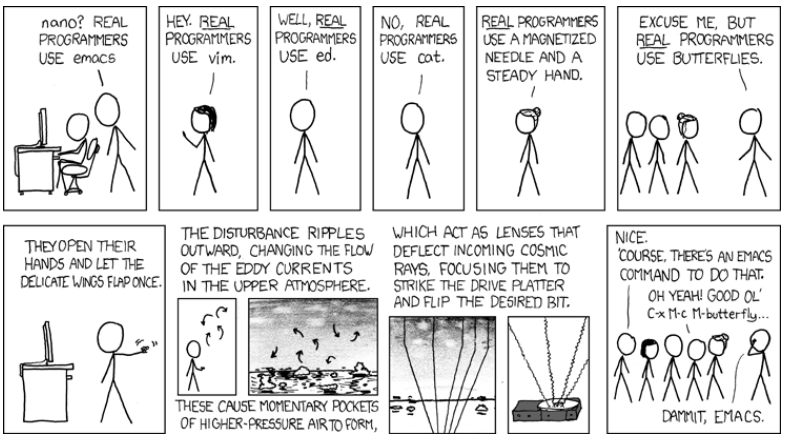

Figure 1: Source: https://xkcd.com/378/ (By the way, your professor recommends using VS code that is available for free on linux, Windows and Mac.)