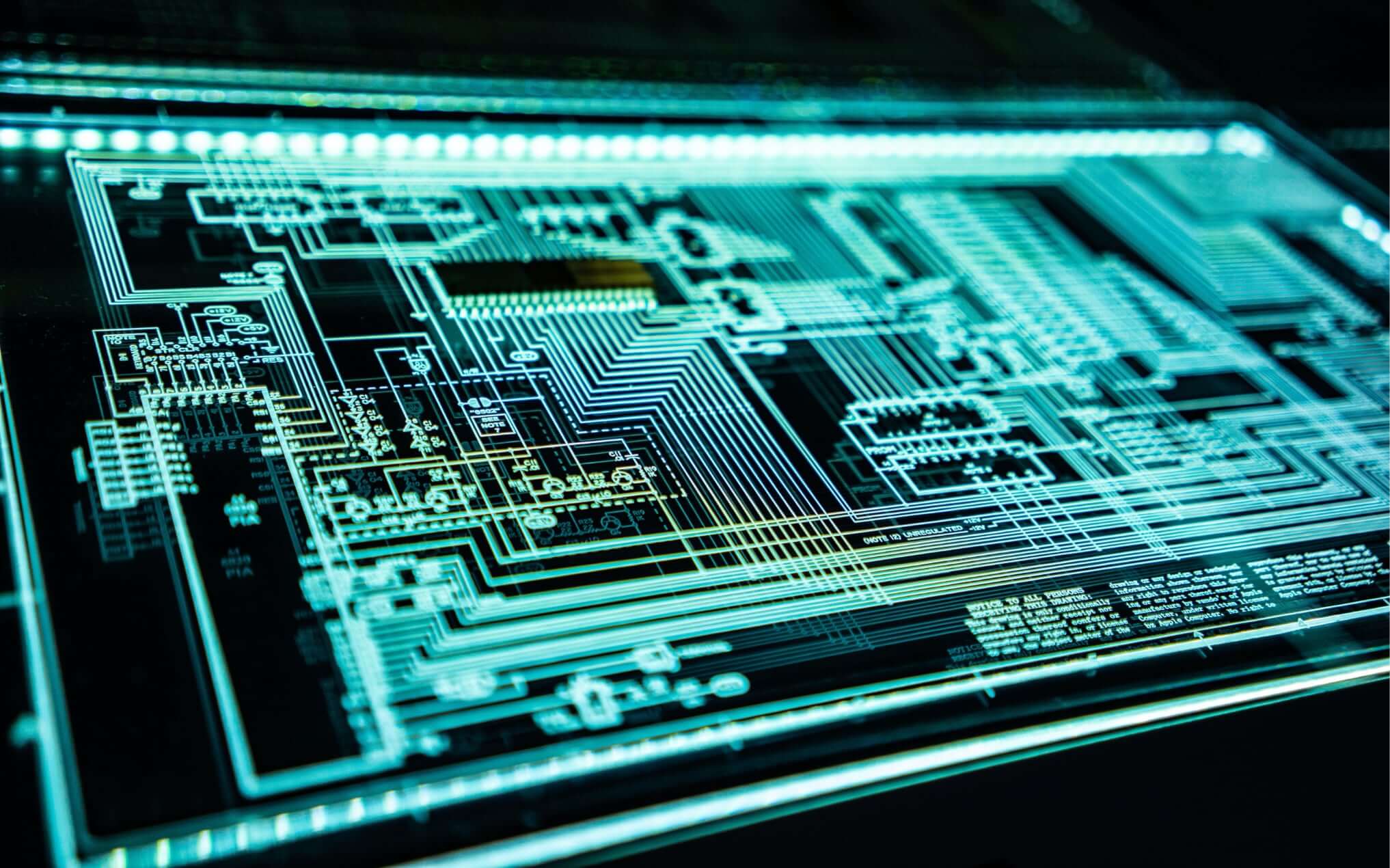


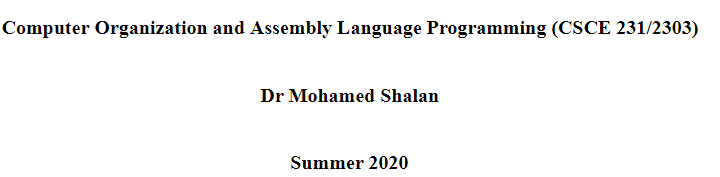
Adham Elkhouly-900171543

Mariam Mousa-900183871

Salma Soliman-900182325



**ARM THUMB Simulator**



Introduction

In this code, we try to implement a simulator that simulates ARM THUMB instruction set, which is a smaller version of ARM ISA. To solve code density problem, ARM THUMB uses 16 bit for an instruction word instead of 32 bit which is used by ARM.

It has 16 registers:

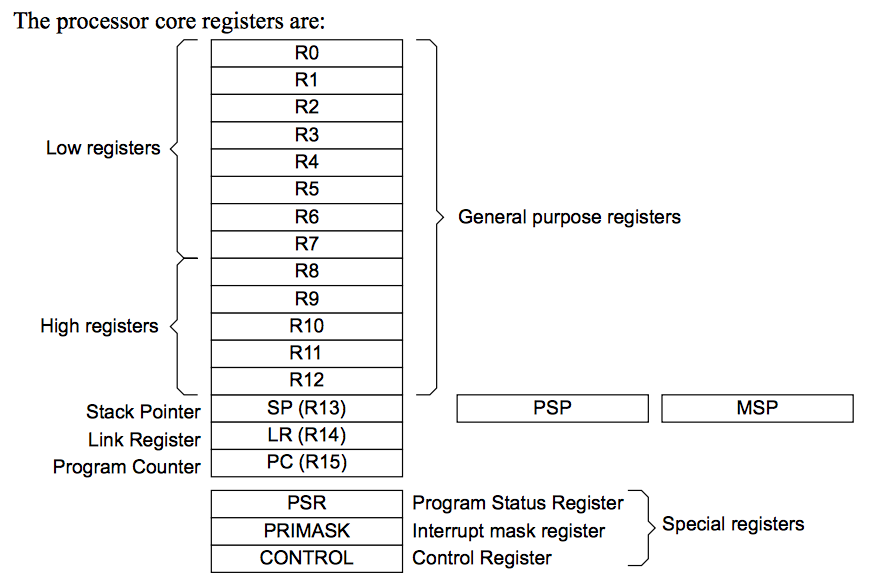
R0- R7: Low registers that are accessed by the programmer.

R8- R12: High registers that are accessed by the programmer for some instruction.

R13: Stack register that contains a pointer to the stack.

R14: Link register that contains the return address.

R15: Program counter register that contain the address of the next instruction to be executed.



Implementation

Algorithm:

Our code is designed to open a certain file by putting its name in the main function. After this binary file is opened, we take the first 32-bit word as the initial value of the stack pointer and the second 32-bit word is the address of the program entry point (PC initial Value). Afterwards, we give an instruction word to function simulate. Each instruction word is disassembled and printed on the terminal. Each modification in registers or the SP or the PC or the LR or the memory is printed also on the screen. PC is adjusted to follow the addresses of the instructions and pointing to the next instruction to be executed. We depended mainly on arrays to represent changes. We have the following arrays:

* unsigned char Mem [1024]: array to simulate memory with 1 k addresses
* unsigned int Regs [16]: array to simulate the 16 register value
* SP Regs [13]: stack pointer
* PC Regs [15]: program counter
* LR Regs [14]: link address
* unsigned int PSR [32]: an array of flags
* unsigned int stack [32]: array for the stack operations (push/pop)
* unsigned int SPSR [32]: saved program status register
* unsigned char Mem[1024]; //array to simulate memory with 1 k addresses

To implement the branching instructions, we used the function fseek, which is a C function that takes an offset and sets the file pointer to that offset. Its syntax is as follows:

fseek (FILE\* fp, int offset, int whence)

whence is used to determine the position from where the offset is added. More can be found on tutorialspoint:

https://www.tutorialspoint.com/c\_standard\_library/c\_function\_fseek.htm

Functions:

We have only one function that is the core of the simulator. Its name is “simulate”. In this function, we receive a single instruction and decode it. Each instruction is differentiated according to the opcode and an additional bit/s – if any-. The opcode is the most three left bits. This function does the disassembly part and the execution one. After disassembling an instruction, it prints out the changes in the registers. We mainly depended on using “if and switch” statements for implementing that. After that, the PC is updated to get the next instruction.

Program Status Register (PSR):

Program Status Register (PSR) holds the program status flags and some additional information. It holds copies of the N, Z, C, and V condition flags. The processor uses them to determine whether to execute conditional instructions.

N (bit 31): Set to 1 when the result of the operation is negative, cleared to 0 otherwise.

Z (bit 30): Set to 1 when the result of the operation is zero, cleared to 0 otherwise.

C (bit 29): Set to 1 when the operation results in a carry, or when a subtraction results in no borrow, cleared to 0 otherwise.

V (bit 28): Set to 1 when the operation causes overflow, cleared to 0 otherwise

-<https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/condition-codes-1-condition-flags-and-codes>

-<https://www.keil.com/pack/doc/CMSIS/Core_A/html/group__CMSIS__CPSR.html>

Bonus part (format 17 & Extension):

We have extended format 17 to support the following:

* reading an integer
* printing an integer
* reading a string
* printing a string
* reading a character
* printing a character
* program termination

|  |  |  |
| --- | --- | --- |
| Instruction name : | input : | Output: |
| Read an integer  SWI 6 | N/A | R0 = integer |
| Print an integer  SWI 1 | R0 = integer | N/A |
| Read a string  SWI 3 | N/A | R0 = length |
| Print a string  SWI 5 | R0 = address of string | N/A |
| Read a character  SWI 4 | N/A | R0 = character |
| Print a character  SWI 2 | R0 = character | N/A |
| Exit the program  SWI 7 | N/A | N/A |

Limitations and challenges:

* Although we implemented the bonus part of the SWI, we could not test it. As we don’t have all fixed numbers taken by the SWI on some samples so that we can test it. We could only test (SWI 1) as this is the one used in the samples.
* We thought of providing a sample made by us to test all SWI instructions but we don't have a THUMB assembler and writing the sample in binary directly was inconvenient.
* Firstly, we did not find specific number in the manual for SWI extension. So, we assumed numbers from 1 to 7.
* SP and PC addresses are not included in all binary files. So, we took their initial value and put it in all samples, which means that we edited a binary sample.
* Although all branch instructions are executed correctly, branching to a far address in (test.bin) is not executed correctly and the file could not disassemble or execute the rest of the instruction. We think it is a very far address that’s out of the scope of the file. We are not sure of this until now.

Output:

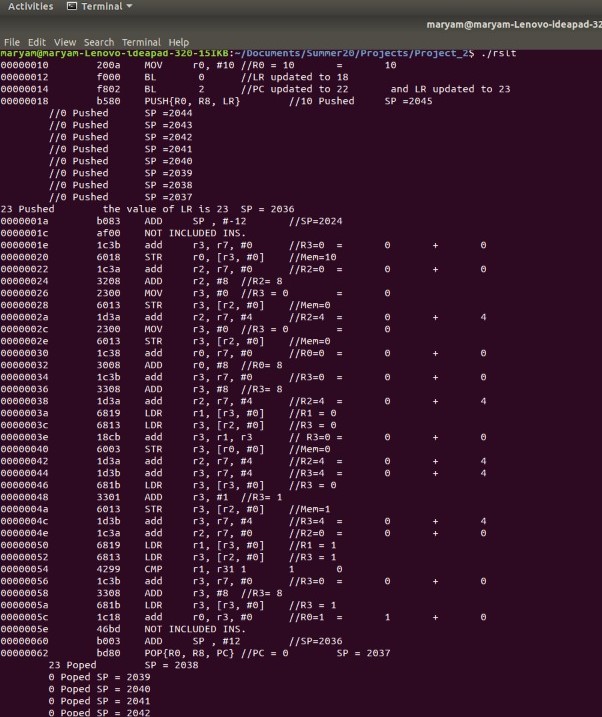


Figure (1): Screenshot (1) from executing sum.bin

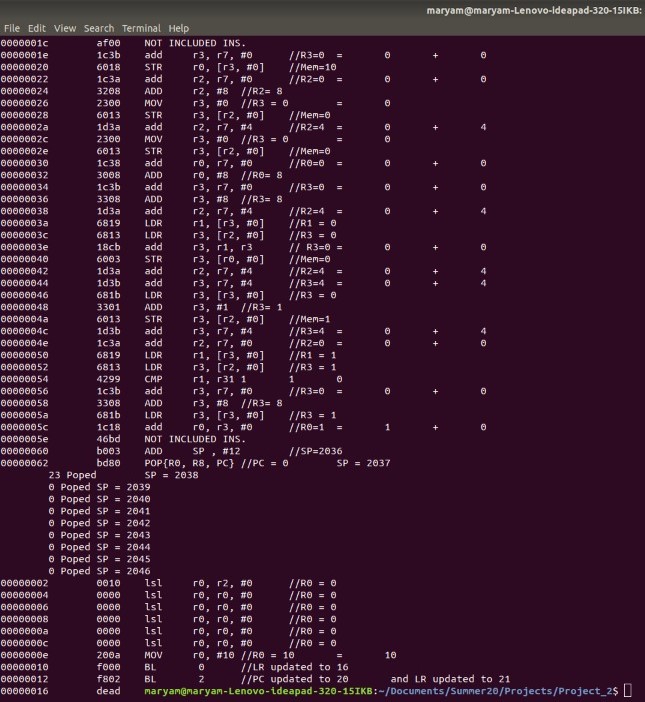


Figure (2) : Screenshot (2) from executing sum.bin

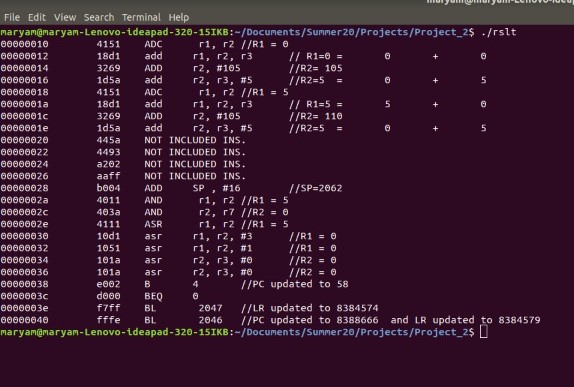


Figure (3): Screenshot from executing test.bin

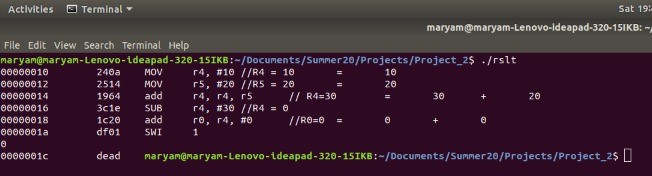


Figure (4): Screenshot from executing test1.bin

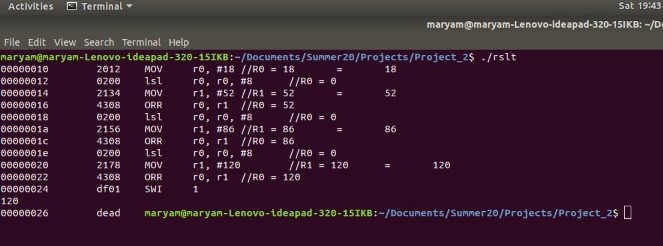


Figure (5): Screenshot from executing test2.bi

Tasks

|  |  |  |
| --- | --- | --- |
| Mariam Mousa | Salma Soliman | Adham Elkhouly |
| * Formats:   16,18,6,7,9,17   * Testing and debugging * Bonus extension | * Formats:   3,4,14,19,17   * Bonus extension * Testing * Report | * Formats:   3,4,6,7  9,13,14   * Printing * Testing and debugging |