

[**CSCE230301 - Comp Org.& Assmbly Lang Prog (2020 Summer)**](https://blackboard.aucegypt.edu/webapps/blackboard/execute/launcher?type=Course&id=_77818_1&url=)

**Project 2**

Andrew Nady

Mahmoud Elshenawy

Mohamed Basuony

Table of Contents

1. Introduction……………...………………………..………….3

……………………………………………………………………

1. Design and Implementation...……………………………….4

……………………………………………………………………

3)Challenges…….. …………………………………………12

……………………………………………………………………

3)Confirming tests ………………………………………….13

……………………………………………………………………

Section 1: Introduction

The goal of this project is to create a virtual ARM CPU that simulates the 16 bit- instructions given to it in the form of a binary file. The CPU deals with 16 registers that handle all the operations happening inside of it. Register 15 and 16 are reserved for the program counter (PC) and link register (LR). Also, Register 13 is reserved for the stack pointer (SP). The memory of our simulator is declared as an array of unsigned characters with a size of 1024. This memory is used to store the addresses of variables when they stored in the memory using store instructions. Another array that was used was an array of flags named bits. It has the size of 32 as there are 32 flags used in ARM ISA. The instructions were divided into parts that represent which register was being handled, the opcode that presents which format was this instruction. The offsets were also isolated and handled using shift instructions and masking using and operator. In order to know which instruction was being executed, a series of switch case statements was used.

Section 2: Design and Implementation

***Format 1 and 2:***

This format was implemented in the skeleton without the part that simulates the instructions. This part was handled. The shift instructions was done using the operator << and operator >> because they are already predefined in c++ to do this.

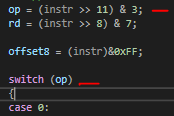
The add instruction was handled also according what was being added. If 2 registers were being added, the implementation used the array of registers to access their values and store them in the destination register.

If the register was added to an offset, the offset was masked and isolated from the 16-bit instruction and stored in the destination register.

The same method was used to handle the sub instruction as it uses the same technique.

The value of rd. was printed after it was changed according to which instruction it was in.

***Format 3:***

This format has an opcode of 1. Therefore, it is handled in case1. After that, bits 11 and 12 determine which instructions in this format is being executed. There are only 4 formats in this instruction so the 2 bits were sufficient. (from 0 to 3).

The first instruction that had op=0 was the MOV instruction. It puts the value of the 8-bit immediate in the destination register.

The second instruction that had op=1 was the CMP instruction. It compares 2 numbers together and sets the flags accordingly.

The third instruction that had op=1 was the ADD instruction. It adds 8-bit immediate value to contents of Rd and place the result in Rd.

The fourth instruction that had op=3 was the SUB instruction. It subtracts 8-bit immediate value to contents of Rd and place the result in Rd.

***Format 4:***

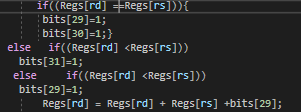
This format shared the same opcode with formats 5, 6 and 7. It is distinguished by the 4 bits from bit 6 to bit 9. These bits determine which instruction will be executed.

When the 4 bits are equal to 0, the instruction AND is executed. The source register is anded with the destination register and the result is placed in the destination register.

When the 4 bits are equal to 1, the instruction EOR is executed. EOR stands for exclusive or. The source register is XORed with the destination register and the result is placed in the destination register.

When the 4 bits are equal to 2, the instruction LSL is executed. LSL stands for left shift logical. The source register is equal to the shifting or rd with the amount of rs.

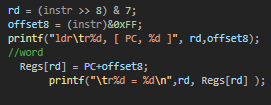
The rest of the instructions are handled the same way. Except ADC.

When the 4 bits are equal to 5, the instruction ADC is executed. This instruction deals with the zero and the carry flags. It sets or clears them according to their values. If rd and rs are equal, it sets both z and c flags. If rd is bigger than rs, negative flag, N, is set. If rd is less than rs, carry flag is set. After that rd and rs are added with the carry bit.

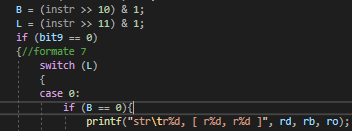
***Format 5:***

This format was only disassembled and not simulated as it was not required in the project. The disassembly was done voluntarily by us.

***Format 6:***

It has the same opcode as 5 and 4. This format handles PC-relative load. The format adds unsigned offset (255 words, 1020 bytes) in offset8 to the current value of the PC. Load the word from the resulting address into Rd.

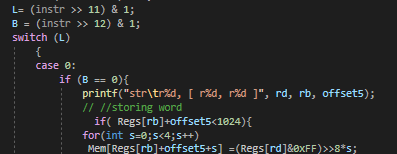
***Format 7:***

This format handles load/store with register offset from memory. The bit number 10 B and 11 L determines which instruction we are handling and what to do with the words/bytes we have. If B=0 and L=0 the instruction STR are executed it calculates the target address by adding together the value in Rb and the value in Ro. Store the contents of Rd at the address. If L=0 and B=1, STRB instruction is executed. It calculates the target address by adding together the value in Rb and the value in Ro. Store the byte value in Rd at the resulting address. The same thing for the other Load instructions except that it loads from memory.

***Format 8:***

This format deals with load and store of sign extended bytes and half words. Bits number 10 and 11 are sign extended flags named S and H respectively. The format is similar to Format 7 with some differences as it deals with half words. It was not required.

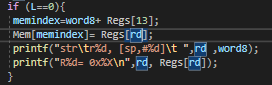
***Format 9:***

This format handles load/store with immediate offset. It has 2 flags named B and L in bits number 10 and 11. These instructions transfer byte or word values between registers and memory using an immediate 5 or 7-bit offset. In the following screenshot is an example from STR instruction.

***Format 10:***

These instructions transfer half word values between a Lo register and memory. Addresses are pre-indexed, using a 6-bit immediate value. The bit number 11 indicated which instruction will be executed. If it was equal to 0 STRH was executed. If it was equal to 1, LDRH was executed.

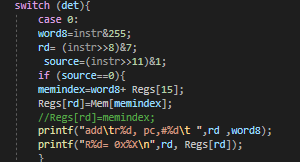
***Format 11:***

This format deals with SP-relative load/store. This format uses Register 13, which is used as a stack pointer to access the memory and load and store from it.

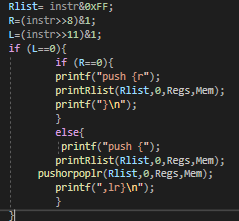
***Format 12:***

This format deals loads addresses. These instructions calculate an address by adding a 10-bit constant to either the PC or the SP, and load the resulting address into a register. Bit 11 acts as an indicator. If it is equal to 0 the instruction Add #Imm to the current value of the program counter (PC) and load the result into Rd. If it is equal to 1 the instruction Add #Imm to the current value of the stack pointer (SP) and load the result into Rd.

***Format 13:***

This format adds an immediate to the stack pointer. Changes the current address that the stack pointer is standing on. It depends on the bit number 6 to determine which instruction to execute.

***Format 14:***

The instructions in this group allow registers 0-7 and optionally LR to be pushed onto the stack, and registers 0-7 and optionally PC to be popped off the stack. The list of registers is in the list is pushed onto the stack. And the stack pointer is updated afterwards.

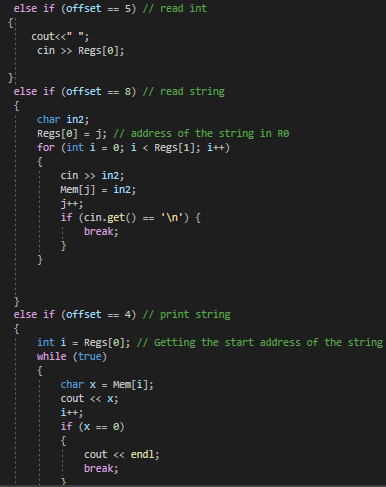
***Format 15:***

These instructions allow multiple loading and storing of Lo registers. The THUMB assembler syntax is shown in the following table. Bit number 11 is the bit that determines which instruction will be executed. If the bit is equal to 0, STMIA instruction is executed. If the bit is equal 1 LDMIA is executed. We did it as extra work.

***Format 16:***

This format handles conditional branches. The bits from 8 to 11 determines which branch condition is going to be executed. The branch instructions depend on the flags that CMP changes their values. In the following code, the flags that determined if the jump to the label will be implemented or not. It branches if Z set, or N set and V clear, or N clear and V set (less than or equal).

***Format 17:***

The SWI instruction performs a software interrupt. On taking the SWI, the processor switches into ARM state and enters Supervisor (SVC) mode. It determines which action the system will do according to the offset in the instruction. We implemented the offsets that handle print int, read int, print string, read string, read char, print char, and program termination.

***Format 18:***

Unconditional branch instruction. .It jumps directly to the label specified. It uses the following formula Branch PC relative +/- Offset11 << 1, where label is PC +/- 2048 bytes.

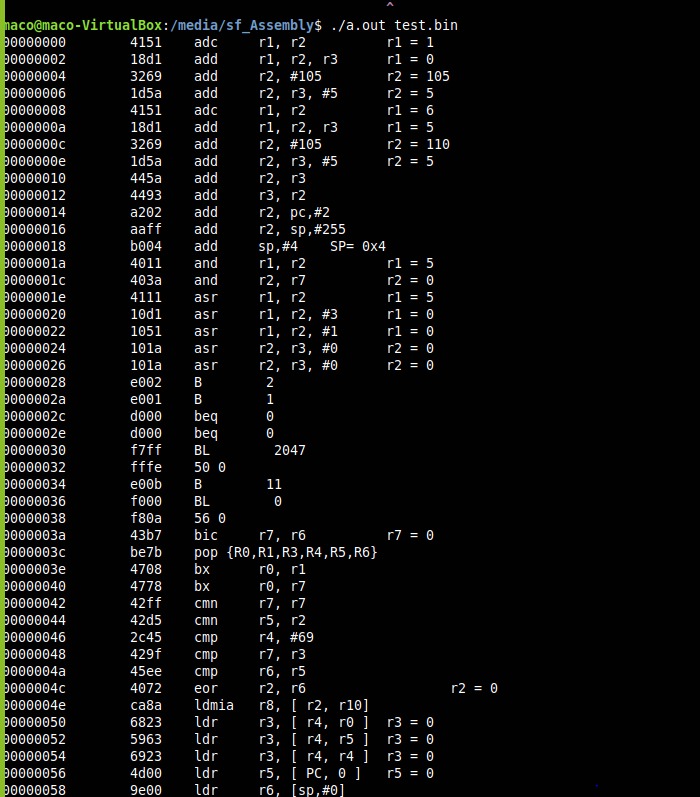
***Format 19:***

This format specifies a long branch with link. The assembler splits the 23-bit two’s complement half-word offset specifed by the label into two 11-bit halves, ignoring bit 0 (which must be 0), and creates two THUMB instructions. It changes the value of the Link Register according to bit number 11.

Section 3: Challenges

1. There were some formats that should be handled the same way, which we solve by creating some helping functions to reduce the complexity and to not mess up the code segments.
2. Sometimes, the PC is not calculated in the right way which makes the program move to a wrong instruction, and thus the code is messed up in small segments of the binary file.
3. There are too many flags to keep track of if we declare to each one of them one corresponding variable. We solved by using array of flags to not make the code hard to read, debug and most importantly to keep track of every single flag.
4. In the original skeleton file, the way we fetch the instruction word was not dependent on the PC, instead the PC change was a result after fetching. What we did is that we change the way in which we fetch the instruction word from the binary file to make it dependent on PC, so if we change PC, the PC will be pointing to another instruction.

Section 4: Confirming tests

Only test sum implements the jump in branch instructions.

