CSE 112 : Computer Organization
Final Report
Anmol Prasad 2016012
Chaitanya Agrawal 2016027
Raghav Bhatia 2016071

MIDWAY PROGRESS

Project 1 - ARM Simulator

Our understanding of the problem:

We need to implement an ARM Simulator which reads input code from .MEM file having ARM commands in hex code format. We will model our simulator such that it reads and executes that code. We also have to focus on and show every stage of the execution of an instruction as done in the ARM architecture, namely Fetch, Decode, Execute, Memory and Writeback. Furthermore, we will try to make this process more efficient by using pipelining (provided we have time left).

Project Implementation

The steps involved in executing an instruction are as follows:

FETCH(IF): The instruction will be read from the input file and program counter register will be updated. The fetched instruction will then be passed to the decode function.

DECODE(ID): The instruction will be decoded by dividing it into parts. The opcode will be used to find out the instruction and type of instruction using a table lookup and the operands will be loaded from the register array on the basis of the opcode using case/if-else conditions. The decoded instruction will then be passed to the execute function. Figure 1 shows one such example.

EXECUTE(EX): The logical/ arithmetic operation will be identified and the function performs the corresponding task. Depending on the instruction type, there will be a conditional code to differentiate between immediate and register as the second operand.

MEMORY(MEM):If the instruction requires memory access(Load/Store etc.), the data will be loaded from the memory using the calculated effective address

WRITEBACK(WB):The data will be written back to memory if needed.

The simulator will exit when the instruction sequence reads “SWI 0x11”

Important ARM Instructions that we are going to potentially support in our simulator:

ADD; SUB; MOV; ADC; AND; ORR; ERR; BIC; MVN; RSB; RSC; SBC; SWP; CMP; CMN; TEQ; TST; B; BL; SWI; LDR; STR

Flags : These store state information about a previous operation. They are in order the 4 most significant bits in an ARM instruction.
They are as follows:
N: Used to denote if result of arithmetic operation was negative(1 if negative)
Z :Used to denote if result of arithmetic or logical operation was 0(1 if zero)
C :Used to denote if result was greater than 32 bits in case of arithmetic operations or after Shift operation 1 was left in carry flag in case of logical operations (1 for both)
V :Used to denote overflow(result>31 bits)

Keeping this in mind, we propose the following plan :

1) Compile the Instruction set for ARM from different resources along with their respective encodings and read the instructions’ binary codes and their breakdown as given in the ARM ISA. (29 October to 30 October)
2) Make an instruction memory which maps the instruction code to the instruction(Fetch Function). (1 November to 3 November)
3) Code a full instruction decoder which completely breaks down the instruction including register variables, constants etc according to the ARM ISA after converting the code from hex to binary. (4 November to 7 November)
4) Prepare the code for execution of instructions according to the output of the decoder. Different functions for each instruction. (8 November to 10 November)
5) Implement the Memory stage, which reads the output from the decode stage and tells whether there is need for memory in this instruction. If yes, it uses the memory from the given input memory file. (11 November to 13 November)
6) Implement the Write Back Stage and writeback changes into the destination register as given in the output of the Decode stage. (14 November to 15 November)
7) Look for possible optimizations for reducing the time of execution. (26 November to 27 November)

8) If time is left, do the following for possible bonus marks. We won't look them up at all until Step 7 is completed.
    a) Implement heap functions, malloc and free.
    b) Modify our code such that it models pipelining.
    c) Include other functionalities and additions.

The project will be coded in JAVA.

FINAL PROJECT REPORT

PROJECT LAYOUT

We have created a java class for this project called ArmDecoder.java. There are additional classes for Registers and Instructions. Register class has attributes like Name and Value, while on the other hand Instruction has attributes like Value and Address. We have created another class namely Operand2Calculator that uses functions like shift and rotate to decode and extract the second operand.

We have created a Hashmap for opcodes that stores each operation binary code along with its corresponding operation name in string format. We have serialized this arraylist and stores it in OPCODES.file and are calling init() every time in order to deserialize and obtain the map. Similarly we have created Register.file that contains all the registers.

STAGEWISE IMPLEMENTATION

- Fetch: In this step, we read all the instructions from the input.MEM file with the help of BufferedReader and put each of the instruction in an ArrayList<>. After this, we read each instruction and using the PC that is stored in R15. Each instruction is then sent to the decode instruction.
- Decode: In this step, we segregate the instruction provided. First, we observe the flags N, Z, C and V in order to determine whether or not to execute the instruction. If the instruction is not to be executed, we move on to the next instruction else we move onto segregate the instruction to identify the various registers, immediate value and check the bit for flags such as I and S. To extract the second operand we use functions like shift and rotate. In this function we also check whether the instruction is of branch type and correspondingly call the branch() function.

- Execute: In this step, we use the opcode passed from the decode stage to identify the type of instruction and then perform an operation accordingly. In each conditional statement we check whether operand2 is a register or an immediate value. For instructions in which memory is not required, we call the writeback with the result and destination register.
- Writeback: In this step, we simply store the value in register passed as an argument to the function
- MEMORY: We have implemented the memory in the form of an array. After calculating the offset, we write/read from that index in the array.

SWI COMMANDS

We have included three basic SWI commands for read, write and exit namely, 0x6a/b/c.

BRANCHING:

The branch instruction is identified in the decode stage and executed using the branch function.The branch function works in the following manner:

- Check if link bit is 0 or 1.
- Set Link register(R14) accordingly.
- The passed 2's complement offset by shifting left by 2(Multiplied by 4) and sign extended to 32 bits. The extended offset is added to the PC. The instruction on the given address is fetched and decoded accordingly.

INSTRUCTION INCLUDED

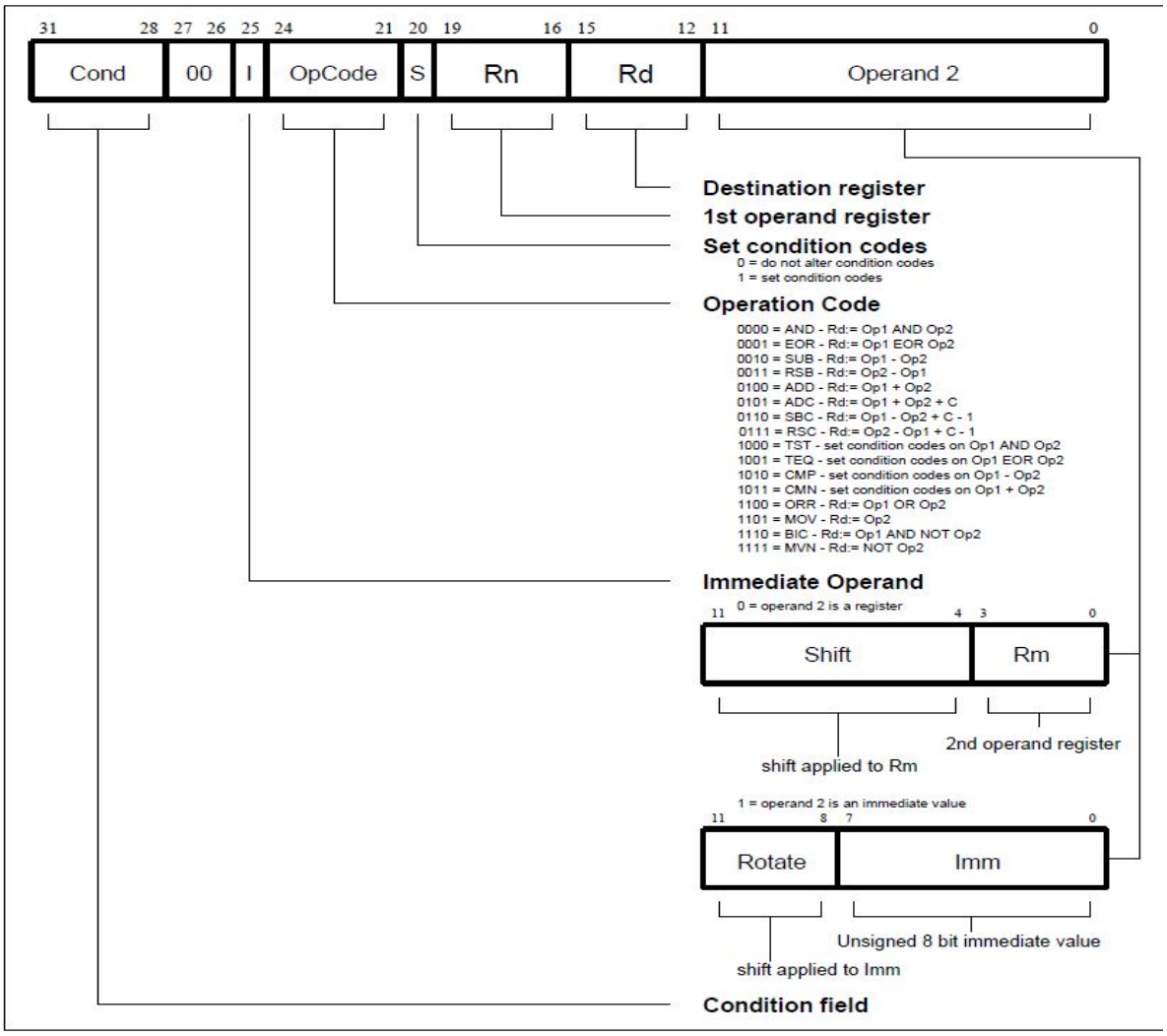
Data Processing: ADD, SUB, MOV, MVN, CMP, CMN, AND, ORR, EOR, BIC, RSB,
Data Transfer: STR, LDR
Branch Instructions: BL, B
Other Instructions: MUL, MLA

VARIOUS FUNCTIONS MADE

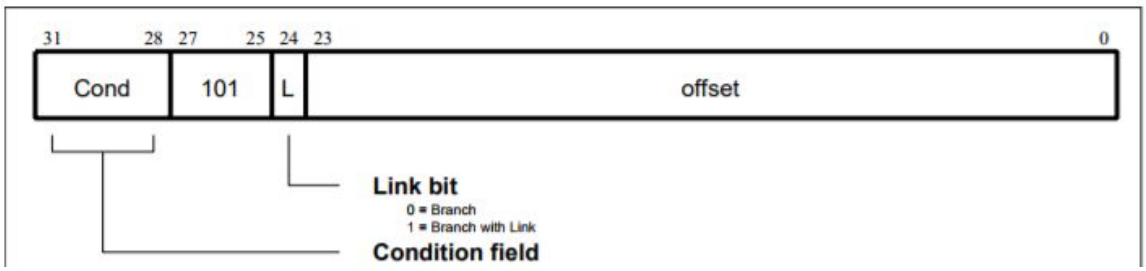
1. Fetch(): To read from input.MEM instruction by instruction.
2. Decode(): To check whether or not to execute the instruction (NZCV) and then identify the kind of instruction (branch, memory, swi etc.).
3. Execute(): Perform any arithmetic/logical operation like add,sub,and,or etc.
4. Writeback(): Writes back the data calculated in the executed stage into the destination register.
5. Get2scomplement(): In order to get the two's complement of a binary string while branching.
6. InvertDigits(): To calculate 1s complement of a binary string.

7. rotate(): To calculate the rotate if any for the 2nd operand value.
8. shift(): To calculate the shift if any for the 2nd operand value.

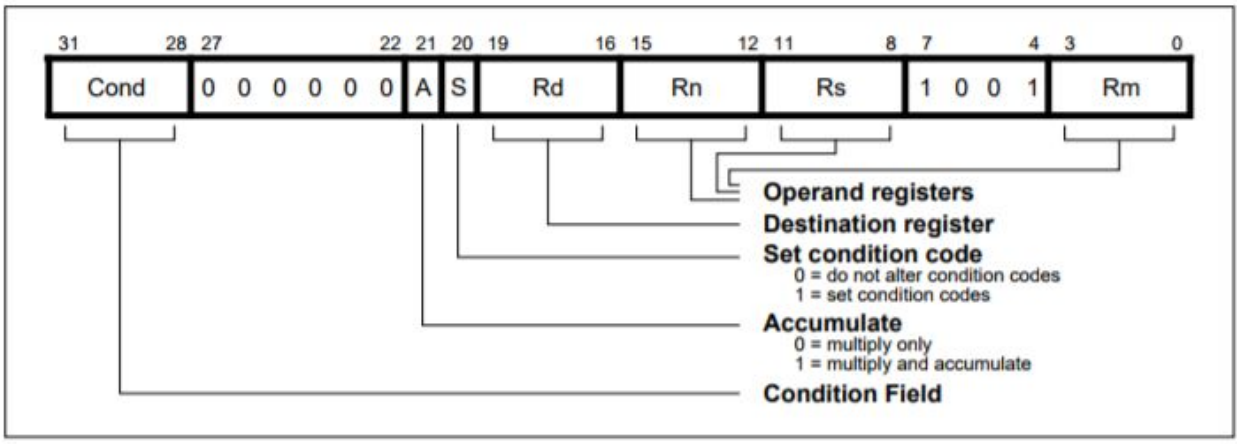
INSTRUCTION BITS INFORMATION



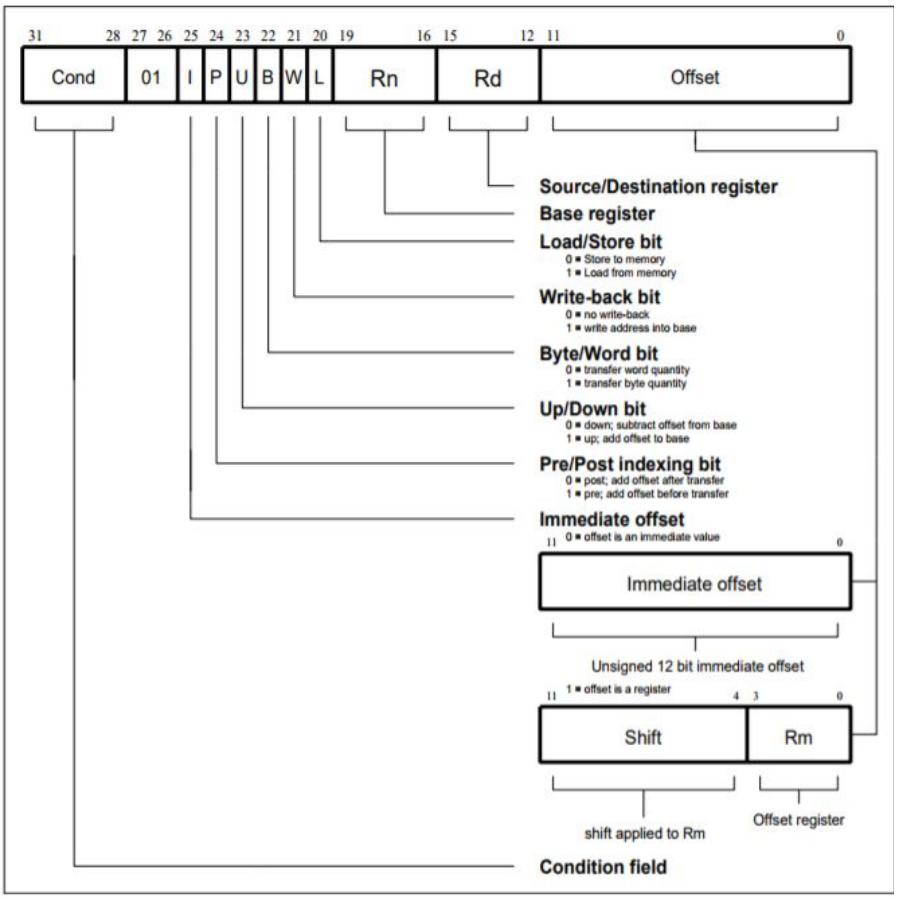
Data Processing Instructions



Branch Instructions



Multiply Instruction



Transfer Instructions

Source : http://bear.ces.cwru.edu/eecs_382/ARM7-TDMI-manual-pt2.pdf