Design Document: Functional Simulator for Subset of ARM instruction set

The document describes the design aspect of myARMSim, a functional simulator for subset of ARM instruction set.

# Design of Simulator

## Data structure

Registers, memories, intermediate output for each stage of instruction execution are declared as global static. Being static, the variables are not visible outside the file, thus, make the data encapsulated in the myARMSim.cpp.

## Input

Input to the simulator is MEM file that contains the encoded instruction and the corresponding address at which instruction is supposed to be stored, separated by space. For example:

0x0 0xE3A0200A

0x4 0xE3A03002

0x8 0xE0821003

## Simulator flow and Output

1. *Reset processor*– Resets the memory to hold NULL values and sets all the flags and registers to NULL.

2. *Load Program* – This function reads the instructions from the .mem file and stores them in the instruction memory. Our memory is divided into two parts: Heap memory (0-1999 indices of MEM array) and instruction memory (2000-3999 indices of MEM array).

3. *Rum ARMSim* – This contains five functions, fetch, decode, execute, memory, write back. These functions keep running until the compiler encounters “SWI 0x11” instruction, i.e “0xEF000011”. At this point the simulator stops and writes the updated memory contents on to a memory text file named data\_out.mem

The five functions are described below:

**FETCH:**

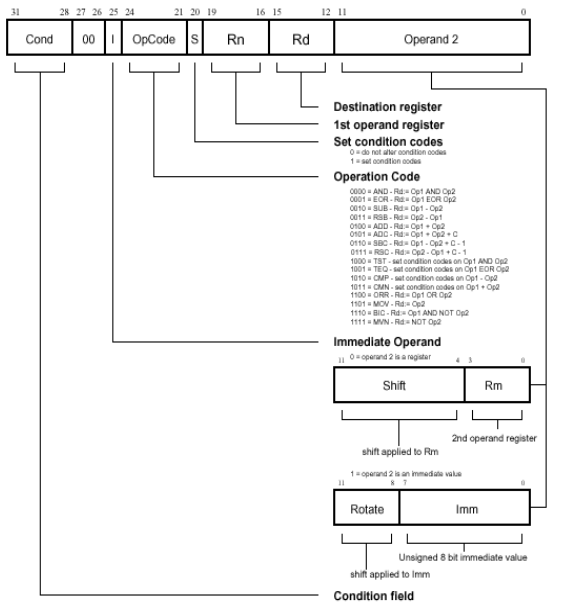
Reads the instruction from the instruction memory one by one and prints -

* + “FETCH: Fetch instruction 0xE3A0200A from address 0x0”

**DECODE:**

Interprets the instruction fetched by the first step based on a few conditions based on the instruction set architecture of ARM. Initially the format [26-27 bits] is decoded form the instruction that divides the instructions into 4 broad categories:

1. Data Processing [format == 0]:



The opcode bits, immediate [I] bits, registers and offset are decoded according to the above-mentioned diagram. According to the opcode and immediate bits, the operation is decoded and it is found whether the second operand comes from a register or is directly addressed in the instruction. The list of operations (both immediate and non-immediate) that has been implemented are:

1. AND
2. EOR
3. SUB
4. ADD
5. ADC
6. CMP
7. ORR
8. MOV
9. MVN

The following message is printed after each operation is decoded:

* + “DECODE: Operation is (1), First operand (2), Second Operand (3), Destination Register (4)”
  + “DECODE: Read register (5), (6)”

1. – Name of operation
2. –Register/immediate value that needs to be used as the first operand according to the operation to be performed and the value of the immediate bits.
3. - Register/immediate value that needs to be used as the second operand according to the operation to be performed and the value of the immediate bits. Not applicable for MOV and MVN operations.
4. - Destination register where the final value will be stored.
5. – Prints the register id and the value of the register, which corresponds to the first operand.
6. – Prints the register id and the value of the register, which corresponds to the second operand/the immediate value of second operand.

2. Data Transfer [format==1]:



The L bit, registers and offset (calculated based on the immediate bit) are decoded based on the above diagram. Based on whether the L bit is 0 or 1 the simulator decodes LDR/STR instruction and prints the following message:

* “DECODE: Operation is (1), First operand (2), offset (3), destination register (4)”
* “DECODE: Read register (5), offset (6)”

1. – The decoded operation based on L bit
2. – Prints the register id for the register corresponding to the first operand
3. – The offset decoded from the instruction fetched.
4. – The destination register where the data is stored or loaded
5. – Prints the register id and value stored in the register corresponding to the first operand
6. - Prints the offset

3. Branch Instruction [format == 2]



The cond bits and offset are decoded. Based on the cond bits, the following branch instructions are implemented:

1. BEQ
2. BNE
3. BGE
4. BLT
5. BGT
6. BLE
7. B

The print statement has the following format:

* “DECODE: (1)”

(1) – Type of branch instruction based on the cond bits.

**EXECUTE:**

According to the format bits there are three execute functions:

1. Data processing:

Performs the specified operation based on opcode and stored the result in a temporary register called temp.

In the case of CMP, the difference of the two operands is found and the N, C, V, Z flags are updated based on the obtained value. The following is printed:

* + “EXECUTE: (1) (2) and (3)”

1. – Operation
2. – First operand
3. – Second operand
4. Data Transfer:

The memory location from where the data is to be stored or loaded is calculated based on the offset decoded from the previous step. The following is printed:

* + “EXECUTE: Add offset (1) to (2) to get memory location”

1. – Offset value
2. – Value stored in register
3. Branch Instruction:

The value of the PC is calculated based on the formula

PC = PC + SignExt32(offset x 4) + 8

The different branch types are executed if the following conditions are satisfied:

|  |  |  |  |
| --- | --- | --- | --- |
| COND | BRANCH TYPE | INTERPRETATION | FLAG STATUS REQUIRED |
| 0000 | BEQ | Equal/ equals zero | Z = 1 |
| 0001 | BNE | Not equal | Z = 0 |
| 1010 | BGE | Greater than or equal | N == V |
| 1011 | BLT | Less than | N != V |
| 1100 | BGT | Greater than | !Z && (N == V) |
| 1101 | BLE | Less than or equal | Z || (N != V) |
| 1110 | B | Always | No condition |

The following message is printed accordingly:

* + “EXECUTE: R[15] gets the memory location (1)”

(1) – HEX code of the Memory location address where the PC will branch to. This memory location is found from the HEAP memory part using the read\_word () function.

**MEMORY:**

For format equal to 0 or 2(i.e. data processing and branch instructions), no memory operation is required. Thus, the function simply prints:

* + “MEMORY: No memory operation”

For format equal to 1 (i.e. data transfer instructions), the required data from the memory location (calculated in the previous step) is stored in or loaded to a temporary register and the following message is printed accordingly:

* + “MEMORY: (1) (2) in/from Heap Memory”

1. – Operation performed (Load/Store)
2. – Value that is stored/loaded

**WRITEBACK:**

The values stored in the temp register are written to the destination register in each format case. Only in the case of branch instructions, CMP and STR, there are no write back operations. The following is printed:

* + “WRITEBACK: write (1) to (2)”

1. – Value to be written
2. – Destination register

**Utility functions and algorithms**

1. *read\_word()* – reads the data from the memory.
2. *write\_word()* – writes data to the instruction memory as per the instructions specified in the .mem file
3. *mem\_write\_word()* – writes data to heap memory part of the MEM array.
4. *Bit extraction*- Bit extraction is performed by the method of bit masking in which the AND operation is performed on the hex code and the required bits are extracted.
5. *Sign Extension*- Extract the most significant bit from the binary code (already stored in the hex format) that needs to be sign extended.

If the most significant bit is equal to 0 then nothing needs to be done as all the other bits are already set to 0.

If the most significant bit is equal to 1 then all the other bits need to be converted to 1 (to get the 2s complement of the number). Hence, we need to simply add the 0xFF000000 to the hex code.

# Test plan

We test the simulator with following assembly programs:

* Simple add. Takes in two integers and provides the sum.
* Fibonacci Program
* Sum of the array of N elements. Initialize an array in first loop with each element equal to its index. In second loop find the sum of this array, and store the result at Arr[N].

# References

* <http://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=arm-instructionset.pdf>
* <http://cas.ee.ic.ac.uk/people/gac1/Architecture/Lecture8.pdf> ~ for branch flag conditions
* <http://simplemachines.it/doc/arm_inst.pdf>
* <http://stackoverflow.com/questions/6215256/sign-extension-from-16-to-32-bits-in-c>
* Lecture notes and backpack resources