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.

# Input/Output

## 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

## Functional Behavior and output

The simulator reads the instruction from instruction memory, decodes the instruction, read the register, execute the operation, and write back to the register file. The instruction set supported is same as given in the lecture notes.

The execution of instruction continues till it reaches instruction “swi 0x11”. In other words as soon as instruction reads “0xEF000011”, simulator stops and writes the updated memory contents on to a memory text file.

The simulator also prints messages for each stage, for example for the third instruction above following messages are printed.

* Fetch prints:
  + “FETCH:Fetch instruction 0xE3A0200A from address 0x0”
* Decode
  + “DECODE: Operation is ADD, first operand R2, Second operand R3, destination register R1”
  + “DECODE: Read registers R2 = 10, R3 = 2”
* Execute
  + “EXECUTE: ADD 10 and 2”
* Memory
  + “MEMORY:No memory operation”
* Writeback
  + “WRITEBACK: write 12 to R1”

# 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.

## Simulator flow:

There are two steps:

1. First memory is loaded with input memory file.
2. Simulator executes instruction one by one.

For the second step, there is infinite loop, which simulates all the instruction till the instruction sequence reads “SWI 0x11”.

Next we describe the implementation of fetch, decode, execute, memory, and write-back function.

FIRST:

We have mentioned the header files

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <string.h>

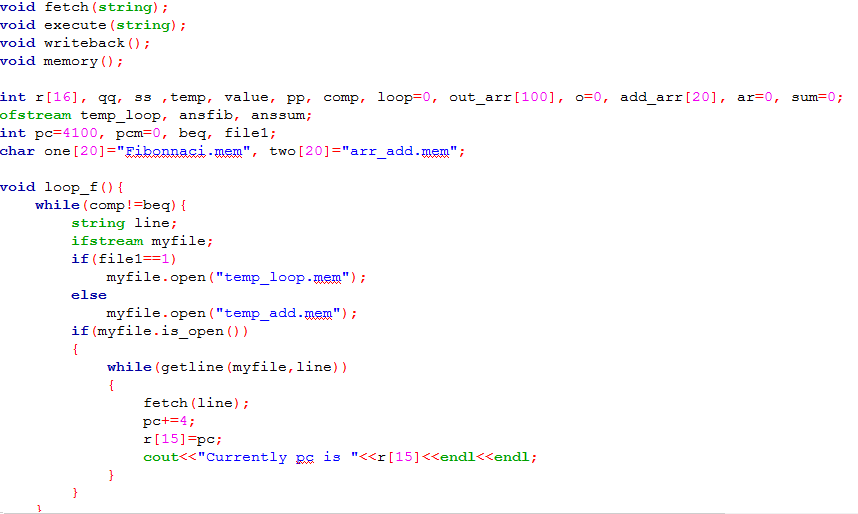
4. #include ”myARMSim.h”

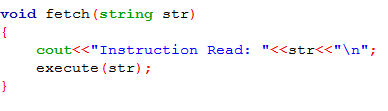
Then we made the array of registers and initialize conditional flags N(negative), Z(Zero), C(Carry or unsigned overflow), V(Signed overflow) and then we initialize a memory block of capacity 4000.

After this we initialize the static unsigned int declaration of instruction word, operand1, operand2, conditional bit, F bit, I bit, opcode, source register, destination register, offset 12, offset, result, readmem, op1, op2 and string s[10].

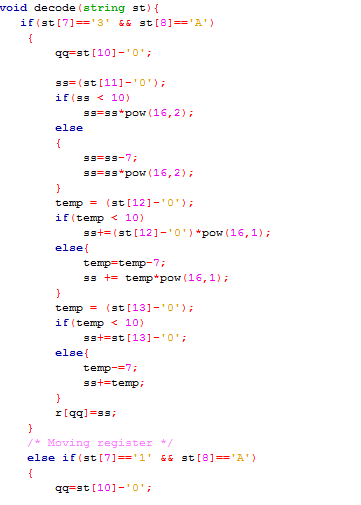
1). Fetch: This function takes advantage of a counter which is attached to the write\_word function which counts the number of instructions. Then we make a for loop till that number, and for each instruction we take the next 8 blocks of memory and print them out, saying that we are accessing this instruction. This 8 character array is then sent to the decode function.

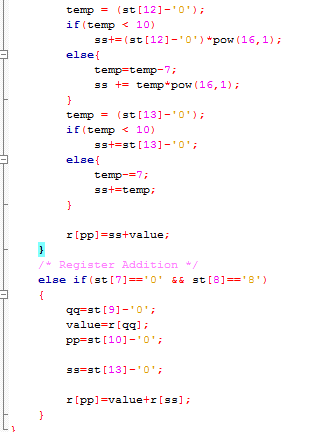
Fetch function initializes or takes the value from memory address of R[15] (PC) and then prints both the instruction and address. And then calls the swi\_exit if if(instruction\_word==0xEF000011) and the increments the PC register by 4.



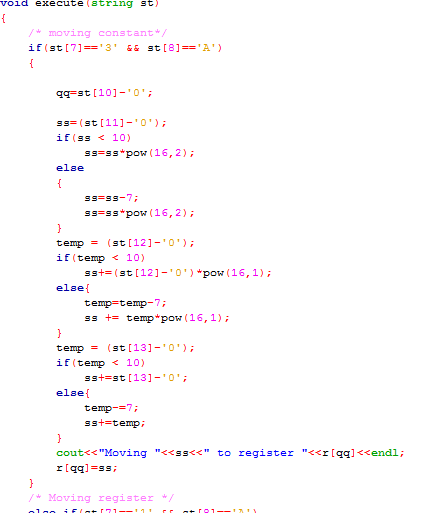


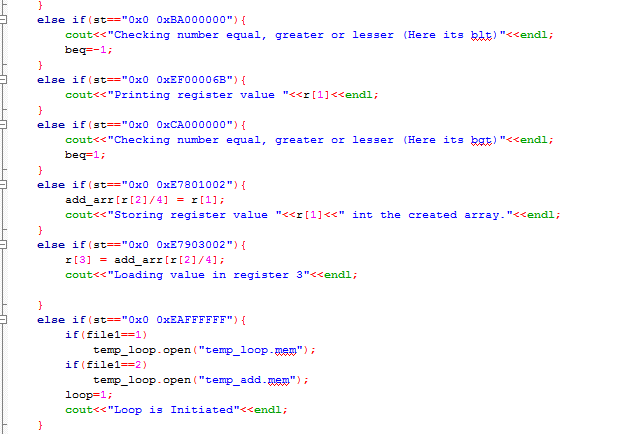
2). Decode: This function first converts the hexadecimal code to binary and then makes choices to find out which instruction it is and what kind of encoding it has depending upon that we get the registers or immediate operands.



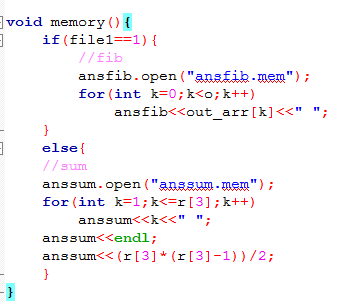


3). Execute: This is where all the calculations happen. We keep variables to store the values and register numbers that are involved in this instruction. We then do the calculations here and show the calculations.

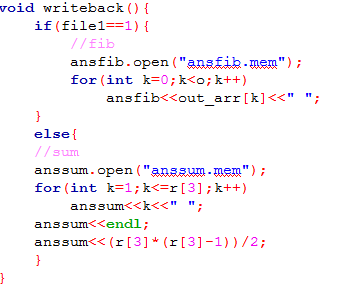




4). Memory: This function gets activated when there is some data written to the memory. The memory gets written when there is a store instruction. The code for the function is given below.5). Writeback: Write back is a [storage](http://searchstorage.techtarget.com/definition/storage) method in which data is written into the [cache](http://searchstorage.techtarget.com/definition/cache) every time a change occurs, but is written into the corresponding location in main [memory](http://searchmobilecomputing.techtarget.com/definition/memory) only at specified intervals or under certain conditions.



5). Writeback: Writeback is used for writing back to registers. It is a [storage](http://searchstorage.techtarget.com/definition/storage) method in which data is written into the [cache](http://searchstorage.techtarget.com/definition/cache) every time a change occurs, but is written into the corresponding location in main [memory](http://searchmobilecomputing.techtarget.com/definition/memory) only at specified intervals or under certain conditions.



# Test Plan:

We test the simulator with following assembly programs:

* 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].

1). Fibonacci: The Fibonacci Sequence is the series of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... The next number is found by adding up the two numbers before it. Similarly, 3 is found by adding the two numbers before it (1+2), and so on.

Code for Fibonacci in hexadecimal representation:

0x0 0xE3A0500A

0x0 0xE3A02000

0x0 0xEF00006B

0x0 0xE3A03001

0x0 0xE1A01003

0x0 0xEF00006B

0x0 0xEAFFFFFF

0x0 0xE0834002

0x0 0xE1A02003

0x0 0xE1A03004

0x0 0xE3A00001

0x0 0xE1A01004

0x0 0xEF00006B

0x0 0xE2866001

0x0 0xE1550006

0x0 0xBA000000

0x0 0xEAFFFFFE

**We have checked our code by calculating Fibonacci series till 5, 10, 20, etc numbers. Our results are correct, signifying we have done it correctly.**

2). Sum of array of N elements: In this we initialize the array with its index and then find sum of all the elements of array.

Code for Sum of array of N elements in hexadecimal representation:

0x0 0xE3A0000C

0x0 0xE3A04000

0x0 0XEF000012

0x0 0xE3A02000

0x0 0xE3A01001

0x0 0xEAFFFFFF

0x0 0xE2888001

0x0 0xE7801002

0x0 0xE7903002

0x0 0xE2822004

0x0 0xE0834004

0x0 0xE2811001

0x0 0xE3580003

0x0 0xCA000000

0x0 0xEAFFFFFE

**We have checked our code by calculating sum till 5, 10, 20, etc numbers. Our results are correct, signifying we have done it correctly.**

These all have been written from Armsim, as it automatically converts our code in hexadecimal representation.