Design Document: Functional Simulator for Subset of ARM instruction set

Roll no.s : Nitika Saran 2014068, Ishita Verma 2014051

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.

In the main program, we first take as input the name of the .mem file and load the instructions in an unsigned char array ‘MEM’. We have an array of 16 unsigned integers R, to represent R0-R15 in ARM. R15 is the program counter, and contains the address of the instruction that is to be executed next.

**Fetch:** Each time we call fetch(), we read the instruction at the address in pc, and store it in *instruction\_word*. Initially, PC is 0 and the first instruction in the MEM array is read, after reading, in fetch itself we increment PC by 4, so that it now points to the next instruction.

**Decode:** In decode, we use bitmasks to compute different fields such as *cond*, *f, opcode* etc. We define an unsigned 32-bit integer *mask*, in which we make those bits one that we wish to read from *instruction\_word*. We then do a bitwise AND operation on *mask* and *instruction\_word*, to make all other bits zero, and shift the result to align the LSB of the result with the 0th bit.

**Execute:** There are 4 different kinds of instructions in ARM- data processing, data transfer, branch and SWI commands. In our implementation, we have 3 different functions to handle the first three cases – *process, mem* and *branch*. Also, before an instruction is executed, the *check()* function checks according to the condition code and current CPSR flags, if the instruction is to be executed. The process function matches data processing opcodes to their operations and stores the result in *acc,* mem() performs LDR/STR (discussed later) and branch() takes in offset, performs a sign extension on it if it is negative and adds the offset to the current value of PC.

**Mem:** To perform memory operations, we have allocated a two-dimensional array data, that has space for 500 integers in each of R0-R15 as the base address. That is, LDR Rn,[Rd, #offset] is performed as Rd = data[Rn][offset]. STR operations are similar.

**Write-back:** The write\_back() function simply stores the value in the temporary accumalator into the destination register.

# Test plan

We test the simulator with following assembly programs:

* Fibonacci Program (file: ‘fibo.mem’)
* 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]. (file: ‘array.mem’)