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.

**Inout/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:

* First memory is loaded with input memory file.
* 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.

Fetch: The InstructionFetch.py File is concerned with fetching the instruction and maintaing the Program Counter. At the beginning of the infinite loop mentioned above, Instruction at current Value of Program Counter is fetched and stored in the variable "instr".

Decode: The InstructionDecode.py file is concerned with decoding instructions. The instruction in variable "instr" is decoded and stored to a variable "midway". This variable contains the name of the instruction, it's format, the registers/immediate value concerned with the instruction. The register numbers in midway for rs1 and rs2, if present, are replaced with the data kept in the corresponding registers at that time. In case of branch instructions, the current PC is also stored in "midway". This operation is done differently for each instruction.

Execute:

1] Except for "jal" and "jalr", all instructions pass through alu.py . "midway" is sent to the alu as input and "opt\_of\_alu" has the output of alu. In the alu.py file, The instruction is passed on to different functions based on it's instruction type. The input format for all instructions of the same type is the same. These functions perform the required operation and return the output.

2] "jal" and "jalr" type instructions are resolved in the main file itself. The address of the immediate next sequential instruction is written to the appropriate register, and the Programme Counter is changed to the appropriate value.

Memory Access: In case of load/store instructions, memory is accessed from memory.py file. Output of alu contains the memory address, data and datatype in case of store instruction, and simply the memory address and datatype in case of load instruction.

Register Writeback: Some instructions require some to be written to the Register file. Registers are stored as an array in the class RegisterTable in registers.py file. They are accessed with the appropriate number in the Register\_Writeback stage, and appropriate data is written back into these registers.

**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].
* Bubble Sort
* Merge Sort
* Factorial