Design Document: Functional Simulator for a Subset of ARM instruction set - Pipelined

The document describes the design aspect of the pipelined version of the RISC-V Simulator, the functional simulator for a subset of the ARM instruction set.

Input/Output

Input

1. A prompt will be displayed on the terminal asking you to enter the following details :
    a. Cache size for data cache in kilobytes:
    b. Block size for data cache in bytes:
    c. Number of ways for SA for data cache:
    d. Cache size for instruction cache in kilobytes:
    e. Block size for instruction cache in bytes:
    f. Number of ways for SA for instruction cache:

2. After entering all the values, a GUI will open.
3. In the Editor tab of GUI, press Select file and browse and select the Input machine code file.
4. You'll notice a number of check boxes for
    a. pipelining
    b. forwarding
    c. printing register values at the end of each cycle
    d. printing pipeline registers
5. Tick the desired box for corresponding result

Input to the simulator is a .mc file that contains the encoded instruction i.e their machine codes and the corresponding address at which instruction is supposed to be stored, separated by space. The data segment, if present, starts after the code segment ends, i.e. after 0xEF000011 (swi 0x11) is read. For example:

```
0x0 0x00000113

0x4 0x00100193

0x8 0x00200213
```

0xC 0xEF000011

0x10000000 0xFFFFFFDD

0x10000004 0xFFFFFFE6

0x10000008 0x00000001

Functional Behavior and output

The simulator reads the instruction from instruction memory, decodes the instruction, reads the register, executes the operation, access the memory when needed and writes back to the register file. The instruction set supported is the same as given in the phase 1 description.

The execution of instruction continues till it reaches the instruction “swi 0x11”. That is when the instruction reads “0xEF000011”, the simulator stops and writes the updated memory contents in the output file.

The simulator also prints messages for each stage in non-pipelined execution, for example for the second instruction above following messages are printed.

- Fetch prints:
    - 'FETCH: Fetch instruction 0x00100193 from address 0x4'
- Decode
    - 'DECODE: Operation is ADDI, first operand x0, destination register x3, immediate value is 1'
    - 'DECODE: Read registers x0 = 0'
- Execute
    - 'EXECUTE: ADD 0 and 1'
- Memory
    - 'MEMORY : No memory operation'
- Writeback
    - 'WRITEBACK: 1 in x3'

The GUI of phase 3 contains these additional tabs/ functionalities with the introduction of cache:

a. The contents of all the sets of the cache (both I$ and D$) which have non-zero data can be viewed after pressing run button in the I_CACHE and D_CACHE tabs respectively
b. For each Fetch , Load , Store the set that is accessed is displayed in SET Tab
c. Upon a miss , the victim block is displayed in VB Tab.
d. A count of the number of main memory accesses , cache accesses , cache hits and cache misses is displayed after execution in STATS Tab

Rest remains the same as previous phases.

Design of Simulator

Data structure

Registers, memories are private variables. Thus, these variables are encapsulated and are not visible outside memory and register files. Interstate buffers are public variables.

Simulator flow:

These are the steps:

1. The memory is loaded with the input memory file. Whenever a value from the memory is to be accessed, cache is used as an intermediate.
2. Simulator executes instructions in a pipelined manner when pipelining is enabled.
3. Simulator executes instructions with stalling if data forwarding is disabled.
4. Simulator executes instructions with data forwarding if data forwarding is enabled.
5. Simulator executes instructions one by one when pipelining is disabled.

After “0xEF000011” instruction is reached, no new instructions are fetched. When 0xEF000011 is intercepted, the remaining instructions in their mid-stages are completed.

Next we describe the implementation of fetch, decode, execute, memory access, write-back function, the main memory, pipelining, HDU, Data-Forwarding, Data-stalling and Cache.

1. FETCH -

(i) Instruction is fetched from the cache using the current PC value. If it is not present in the cache, then the main memory is accessed to update the cache.

(ii) PC is incremented by 4.

(iii) Branch prediction is done using BTB for the control instructions.

2. DECODE -

In the decode the current value of instruction present in IR register is passed as parameter in 32 bit binary format. This instruction is decoded as follows:

(i) The last 7 bits are extracted to get the op code. By using the op code the instruction type or fmt is found.

(ii) Then the fmt is used to find what will be the instruction format and rs1, rs2, rd, immediate, func3 and func7 are extracted accordingly from their positions in the instruction format.

(iii) In R-type: func3 and func7 are extracted and used to find the mnemonic of the instruction. Rs1, rs2 and rd were extracted.

(iv) In I-type: func3 was extracted and was used to find the mnemonic along with opcode. Rs1, rd and immediate were extracted.

(v) In UJ-type: op code was used to find the mnemonic. Rd and immediate were extracted.

(vi) In U-type: op code was used to find the mnemonic. Rd and immediate were extracted.

(vii) In S-type: func3 was extracted and was used to find the mnemonic along with opcode. Rs1, rs2 and immediate were extracted.

(viii) In SB-type: func3 was extracted and was used to find the mnemonic along with opcode. Rs1, rs2 and immediate were extracted.

3. EXECUTE -

In execute output of decode stage is taken as parameter i.e.[ins_fmt, inst, registers, memory, imm] .This instruction has been executed as follows.

1. R-Type : in R-Type instruction operation has been performed on rs1 and rs2 depending upon the type of the instruction and result is stored in a temporary register (ry) , which is then returned to store this value in rd.

2. I-Type: similar to R-Type operation is performed on rs1 and imm and result is stored in ry, which then returned to store this value in rd.
3. S-Type: Effective address is returned as per the definition of S-Type of instructions to store the word,byte or halfword at destination.

4. SB-Type:These are conditional branching instructions, which updates the PC value when condition is satisfied

5. U-Type: lui->sets rd to a 32-bit value with the low 12 bits being 0 and the high 20 bits coming from the U-type immediate .
   auipc-> sets rd to the sum of the current PC and a 32-bit value with the low 12 bits as 0 and the high 20 bits coming from the U-type immediate.

6. UJ-Type:Current PC value is returned in jal instruction and PC us updated as per the definition of jal instruction

4. MEMORY ACCESS -

Other parameters are memory module, register module, data cache module and the address at which the operation is to be performed. Cache is accessed for lw, lb and lh instructions. For store instructions write through policy is used for cache. According to fmt the instructions lw, lh, lb, sw, sh and sb are performed. In case of load operations, the value stored at the address is returned. The methods lw, lh, lb, sw, sh and sb are present in the memory class.

5. WRITE-BACK -

Write-back takes place when the control bit for write-back is non zero. Other parameters are the register module and the data to be written. If the write control is non-zero, then the data is written in the destination register. The “make0()” method ensures that the register x0 always contains the value 0.

6. MAIN MEMORY -

The main memory initializes a dictionary which contains the address and the value stored at the address, byte by byte. The “code_ends()” function writes back the final memory in the output file when the “Dump” is pressed. The load methods take address as their parameter and return the value stored at that address. The store methods take address and data as their parameters and they store the data at the address, i.e. they update the memory.

7. PIPELINING -

Pipelining has two versions i.e stalling and data forwarding. Pipeline provides functionalities like stalling ,flushing as well as data structures like actual_pipeline,disablePC etc.

Pipeline structure is basically a 2D array which will store the information needed to execute the particular instruction.One by one each cycle is taken into consideration and the steps mentioned inside the cycle are executed. Once we fetch the ending instruction program is terminated.

8. HDU

The hazard detection unit or the HDU detects the cases in which data forwarding would be required. The different cases required while data forwarding takes place are EE, ME, MM, MES, ED, MD, EDS, MDS, MDSS. The HDU takes the interstate buffers, list of previous instructions and the knob for data forwarding and d as its inputs. “d” is a dictionary which contains the various paths of forwarding and it will be updated as the output of the HDU.

9. DATA-FORWARDING:

In decode the data hazard and control hazards were checked for the instruction which was current in decode stage using HDU.py. Accordingly it was found which type of data forwarding has to be done for that instruction. If any data forwarding needed stalling then stalling function was called and in the next cycle the data was forwarded accordingly. For data forwarding data_forw function was called and accordingly M-D, E-D, M-M, E-E or M-E was done.

10.DATA-STALLING:

In data-stalling data_forwarding_knob will be off. In this case, for every instruction, if there is data or control hazard, it will be resolved through stalling only. Stalling is done by making disable_PC = 0 which halts the fetching of next instructions. First the instructions in their midstages (from execution stage) are completed till the cycles we have to stall, and then fetch of new instruction is done.

11. CACHE

The cache class implements the working of a Cache module of given specifications. It initialises a multidimensional array for storing the tags, data and other important information like Valid bit, etc. It also initialises another multidimensional array for storing the preference data of each set for LRU working. The *decode_address* function extracts the Tag, index and Block Offset from the address. The *hit_miss* functions checks if the given address is a hit or miss. The *read* function read data from the given address. The *write* function is used in the case of a cache miss. LRU policy is used in the *write* function. This Cache module assumes word alignment and hence, block size must be greater than a word and must be a multiple of word (32 bits). The LRU Policy is implemented in the function *write* and *set_pref.* When there is sw instruction, write through policy is implemented and the *Storedata* function is used to check if the particular block is present in the cache and if present it will get updated. Two different caches Icache and dcache are there for instructions and data respectively.

Test plan

We tested the simulator with following assembly programs:

- Factorial iterative
- Fibonacci iterative