Project 2 report

Tomasulo Algorithm Simulation

CSCE330101: Computer Architecture

Habeeba Sakr - 900183103

Maria Hanna - 900181559

10 December, 2020

Approach:

We wrote some instructions on an external text file, then we started dividing the main objects of the project into several classes, structs, vectors, maps, queues and arrays.

We initialized the single memory as an array of strings of size of 128 KB as required which is 65536 and the memory will be designed to store at one half the data and at the other half the instructions.

Then we did a vector of instructions with parsed instructions.

Implementation:

RS Struct:

- We created a struct called RS for the reservation stations in which we created a boolean to know if the reservation station is available or not, and a string OP to set the type of the instruction, we also initialized Op, Vj, Vk, Qj, Qk and the address and we created a counter to calculate the number of execution cycles .
- we created an array of type RS of size 9. In which we used it at first in the main to set the type of the 9 instructions we need to support in this array. And we also used this array in the main to set the number of execution cycles for each instruction as required.

Instruction Struct:

- We created a struct called instruction that has the source and destination registers members and has the integer variables support1 and support2 that were designed to map the Reservation stations to the instructions, so for instance inst.support1=1 maps to Reservationstation[1]

- And some other variables to calculate the issuing time and the start and end of execution times and the writing time and finally a result variable that stores the result of the instructions according to their type
- Then we created a vector of instructions called instt and two queues of the type Instruction.

Parse Function:

- We opened the file that contains the instructions we wrote and then we started reading line by line and instantly adding each line to a new location in the memory.
- Then for each line we get a substring to read the instruction type and save it at “op”, we save either the 1 or two operands and the immediate and we set the support1 and support2 of the instruction. We pushed all the instructions in a vector to be used later.

Issue function:

- Since we had two reservation stations for some of the instructions, we checked each support separately, so for instance we had to check first on support1 and check if its corresponding reservation station is busy or not. If it is not busy, then we are able to check the instruction itself. We check if it is add or div, etc. and then for each case we check the register status of each operand in the instruction. If it is not equal to zero (operand is busy) then we set Qj of the reservation station of the current instruction and fill it with what’s inside the register status (what we are waiting for). If the operand is not in the register status, we issue the instruction

by putting the value of the operand in the regfile in Vj (and Vk if it exists) and setting Qj to zero.

- In all cases we increment the PC, and set the variable called issuing inside the instruction to the current number of clock cycles. We also set the rd of the instruction in the register status with the number of reservation station we're currently at.
- In the case of the load and store, we have implemented the Load store queue so whenever we issue a load or a store instruction, we push it back to this load/store queue.
- In the BEQ instruction, we set a boolean flag called stall_exec to true, so it is able to stall the execution of any instructions after the Branching, (branch prediction) and we increment the number of branches encountered. We also set a variable inside the instruction called PCissue to the current PC to be used in the future.
- In the case of JALR and RET, a boolean called stall issue is set to true, since we are stalling issuing all instructions since we will be jumping anyway.

Execute function:

- This one was quite straightforward. We looped over the 9 register stations we had, and if we found the reservation station we wanted (busy) we set it to true and set the starting execution time. We then check which instruction it is and act accordingly.
- For each type of instruction we calculate the result of the operation. We first check that Qj and Qk are equal to zero which means that the operands are ready to be executed then we do the actual execution.
- In the case of BEQ, we increment the mispredictor variable whenever we actually take the branch since we follow an always-not taken branch prediction.

Write back function:

- With write back, we declared a databus that we set to true as long as the instruction is writing back, so it doesn't write back more than one thing per cycle. If databus was equal to false, we then set it to true, incremented the written instructions, and then checked for each instruction name
- For the arithmetic operations it was quite easy, we set the value of the result previously executed in the RegFile and zeroed the rd in the register status, we then looped over the reservation stations and checked if Qj or Qk were waiting for this result we just wrote back, and if so, we then wrote it back to its corresponding Vj or Vk.
- For the store, we checked if the instruction was at the front of LSqueue before executing, and if it is we wrote the result in the memory.
- For the load again we checked the LSqueue, and then we looped over the reservation stations and checked if Qj or Qk were waiting for this result we just wrote back, and if so, we then wrote it back to its corresponding Vj or Vk.
- For the BEQ, we put the previously computed result in the PC (whether we jumped or not) and then looped over the instructions to see which ones were issued after the BEQ and we flush them.
- For the Jalr and the return we put result in the PC, and we also set stall_iss and exec_stall to false again to stop stalling.
- We then reset the databus to false.

printtable function:

- This is a simple function that basically loops over the instructions to print their issuing, execution and writing times.

Main:

- We asked the user to input the number of instructions and then we called the function parse to open the file and start to read line by line.

- We initialize here the Register Status, the register file, and the reservation stations.
- We loop over the stages while the PC is less than the number of instructions.
- We first start by incrementing the clock cycles.
- To issue, we check if the boolean issue stall is equal to false, if it is, we send the instruction (from the vector of instructions we have) to issue.
- Next, we loop over the instructions in the vector, we check if the instruction has been issued or not and if there is no stalling to start execution. We created a new int called execloop in which we calculate the total number of loops we should do the execution process. In each while loop we iterate 9 times and check that an instruction has been issued. If the instruction has not started execution yet, set the start execution time with the number of cycles incremented by one.
- We then set the time of the finished execution
- We then call write back and give it the pc-1 since it was incremented in the issuing stage.