**Suyash Singh 2018CS50646**

**Chirag Bansal 2018CS50402**

**Report (Assignment-9)**

***Outline:***

This assignment extends assignment 8 in functionality. We add a proper measure to prevent hazards and stalling is removed as a solution hence reducing the number of clock cycles. A sequential order of execution which forms the base for our ultimate pipelined structure of the program. There are basically 5 components – IF (Instruction Fetch), ID(Instruction Decode), EX(Execution), MEM(Memory Access), WB(Write Back).

***Structure:***

We define our instruction as a struct in C++ which has the following data members – a vector of tokens (it is the instruction broken up), the bool values tell us about the kind of instruction, id is the number of the instruction as they appear, clock\_cycles tell us about the clock cycles it takes to complete the instruction (given in the input), registor[3] array are the values of the registors involved in the function, function is the name of the function and stage\_value is an integer variable which tells us about the stage of the instruction whether it is is IF, ID, EX, MEM or WB.

Memory is of String type, first 1000 being reserved for the instruction memory, checkRegistors is a vector we use for finding out hazards in ID part, and helpCompare and helpErase are just functions helping in the manipulation of the vector.

***Functions:***

* Fetch instantiates the instruction and breaks it into tokens, also gives it its stage\_value.
* Decode checks for the type of instructions and reads data for the registors, this is also the part where we check for hazards and where stalling might be necessary.
* Execute performs particular operations according to the function.
* Memory\_access if only used for store word where it writes into the memory.
* Write\_back has two uses, one is as understood by name, it writes back data into the regitors if needed, also removes from the checkRegistors array all the functions that wrote back in the present time because now they will not be causing hazards in the future.

***Pipelining:***

In the execute function we have two instruction pointers Instruction\_first which always fetches instructions to be pushed in the list of jobs, temp is just a temporary variable.

The jobs queue and buffer stack are used to implement pipelining. The queue contains jobs (instructions at some x stage) being executed at the present time. We go in the while loop and run till this queue is empty, where each instance of an instruction is popped, advanced to the next stage and put in the buffer stack.

Out of the loop we push all the upgraded instructions in the stack back into the queue along with the new fetched instruction which forms the queue for the next iteration of the loop. We also handle stalls accordingly. Once out of the loop we finish with executing any jobs that were left in the queue.

The read\_to\_memory() function just initializes the memory with instructions.

The initialise() function just hard-codes the registor to some initial values.

The main() function calls on read\_to\_memory(), initialise() and execute().

***Stalling and Hazards:***

For data\_hazard stalling we just continue decoding the instruction, not allowing it to go to the execution phase until it is no longer a hazard.

For control\_hazard we prevent the current instruction fetch and decode (in case of branch statements) and update the program-counter.

We use Hash-Tables to check for data-hazards, where we store the register number and corresponding instruction inside the hash-table.

Inside “Instruction-Decode” we check for the source registers inside the hashtable and on a match we toggle our *data\_hazard* variable to true.

Inside *execute()* if the data\_hazard is true then we go through the hash-table and on finding the changed register (which would be in its memory-access stage ‘or’ write stage) we store the value in the register before the current instruction reaches its execution phase.