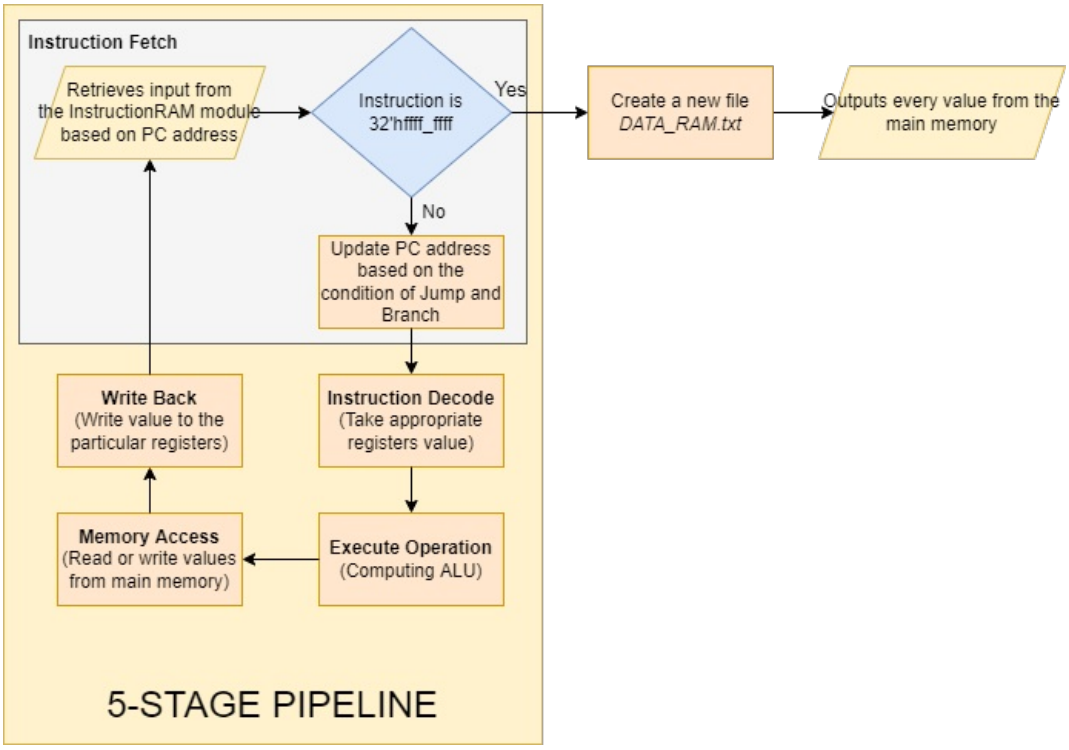
Overview

This project aims to simulate the Central Processing Unit (CPU) using Verilog language. The program requires an input that consists of MIPS machine code instruction lines. The program will execute the instruction based on the classic five-stage pipeline method. The provided module `InstructionRAM.v` and `MainMemory.v` will be used as the primary storage of both instructions and data RAM.

Implementation

The program will take a line of instructions from the InstructionRAM module based on the current PC address. The program first will divide the instructions. After the instructions are separated, the program will analyze the requested instruction. The program then will look for the appropriate values. The values are computed after the instruction. The result is either stored in the memory or used to get data from memory, or directly stored in the destination register. The program then updates the PC for the next instructions line.



The program first will take lines of MIPS instructions and process them to be executed. The program then will use API for the memory provided. As previously mentioned, the program will use the five-stage pipeline. As a result, the program loop must be split into five separate sequential functions. Those functions will run together with the clock cycle in which the clock is set based on the tester's demand. In the implementation, only consider when the clock is in the positive edge, then run each of the five stages with `always @(posedge clock)` command.

The first block is instructed to fetch the instruction from the instruction store. The function will also load the instruction according to the current PC value (in this case, PC is not always a multiple of 4 with the assumption address of PC is already divided by 4 in the first place). The loaded instruction is sent to the `ID` buffer. The loaded instruction will be transferred to the next function for the second stage.

The second block is made to decode the instruction. In this function, instructions are classified into opcodes, function codes, etc. This function then stores the opcode and function code in the `EX` buffer to identify the instruction for the next stage.

The third block is about executing the previous altered information. The arithmetic logical unit executes the program according to the function and opcode. The resulting data is passed to the `MEM` buffer to access the fourth function, accessing the memory part.

The fourth function is responsible for instructions requiring memory access, such as memory read (MemRead) and memory write (MemWrite). Before accessing the data in the main memory, it gets an address from the data in the buffer register and takes action according to the instructions. The registered address is then passed to the `WB` buffer to access the last function. This is a memory writeback where the memory read and loaded by the previous function is rewritten to the register.

Additional registers are used as buffers in this program; those are the `ID`, `EX`, `MEM`, and `WB`. Each buffer consists of 256 bits, which is used in this implementation to save the state of particular instructions. Such as the rs value, rt value, immediate value, etc.

Details

Of course, in the implementation, there might be some mistakes. Consequently, the implementation itself must be finished carefully. Some slight mistakes such as control unit, address, or even index-based value make a different result.

It must be noted that while traversing between stages, the value of control units is usually changing. To solve the previous state of the control unit, it must be saved to additional registers. The buffers `ID`, `EX`, `MEM`, and `WB` will come in handy. Not only that it holds some value related to the instruction, but it also saves the control unit that consists of memory writing, memory reading, register writing, etc.

The use of an address is tricky as usual, especially for jump and branch instructions such as `beq`, `bne`, `j`, `jr`, `jal`. For this, the use of address must be consistent based on the type of instruction. In other words, both `beq` and `bne` must be compatible with each other, and `j`, `jr`, and `jal` must also be consistent with each other. As a side note, it is unnecessary to have the branch and jump instructions to have the same actions. This is due to another trick that is used in the program, which is to put both `ID` and `EX` values to some values so that when meeting branch and jump instructions, the two instructions that were run by the computer will be skipped (as it was intended from the MIPS instruction). The approach still holds the foundation of the 5-stage pipeline.

For the index based is quite similar with the address one. In some cases, when we make the PC address change value to the branch/jump address, sometimes, it will later be added with 4 bytes (or 1 in word) value that creates different results. Because of this, the program was tested many times. One of the methods that are used for debugging is to display every value found or instructions executed. We can analyze and compare that with the one we had manually written.

Output

How to run the program

1. Make sure the tester has installed the *Icarus Verilog* executable application for the compilation.
2. Open `Command Prompt` application and direct it to the PATH folder where the Verilog source codes are downloaded.
3. Put the test case in the format of readable values (preferably bytes) in a file named `instruction.bin`.
4. Run the `Make.txt` file by `make -f Make.txt` command in `Command Prompt`.

Result

After successfully running the make file, a file named `DATA_RAM.txt` will occur, and it will present the stored values in variable `DATA_RAM` in main memory from index 0 to 511 in bytes format.