**ECE-552 Final Report**

**Team Members :**

Abhijith Somashekhar

Megh Doshi

Sidharth Gurbani

**Overview**

The aim of the project is to design a 5 stage pipelined processor with Instruction and Data cache based on the WISC-F19 ISA. The Block diagram of the design has been shown below.

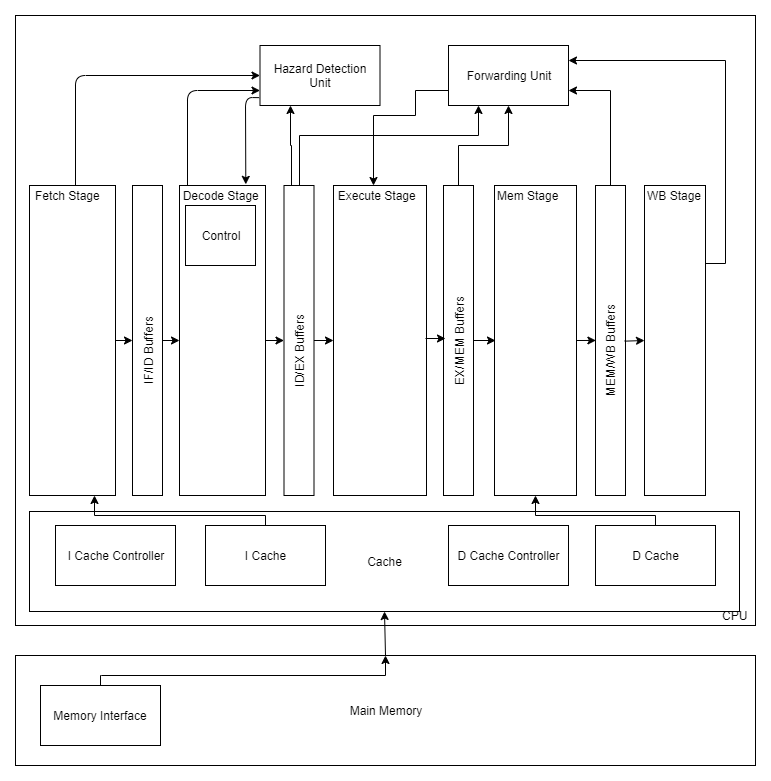


Fig. 1 Block Diagram of the Processor

WISC-F19 ISA specifies 3 types of Instructions - compute, branch and memory instruction and the design has been implemented in accordance with that. A Hazard detection module has been implemented which resolves data and control hazards and branch prediction logic which uses a branch not taken policy. This logic uses stalls/NOPs where necessary to preserve functionality while maintaining the pipelining. It has an ALU that is capable of performing various arithmetic operations such as ADD, SUB, XOR, Partial ADD , Reduction etc as specified by the WISC-F19 ISA. It includes a full forwarding module that implements the EX-to-EX, MEM-to-EX and MEM-to-MEM forwarding logic It also includes separate data cache and a cache controller. The cache is a 2-way set-associative, with cache blocks of 16B each. The cache write policy is write through and write allocate. The following image shows a top level block diagram of our implemented design.

The CPU includes the caches, the cache controllers, all the pipeline stages i.e. from the fetch stage to the memory stage, the hazard detection unit and the forwarding unit. This is interfaced with the main memory.

A 4 cycle main memory module has been provided which has been used in this implementation. The cache controller module is responsible for retrieving data from the main memory and the writing into the cache. The design uses 2 separate caches, I-Cache for caching instructions and a D-Cache for caching program data.

The cache controller FSM loads the data from the main memory into the cache in case of a miss and stalls the pipeline until the data is fetched. A Cache Hit/Miss is decided based on matching of the given tag with the tag stored in either one of the 2 ways. This is done in parallel with loading the data from the cache to retrieve data in a single cycle.

A memory arbitration logic has been implemented that handles simultaneous I-Cache and D-Cache misses. The arbitration logic gives preference to I-Cache misses over D-Cache misses and is serviced.

The flow of data is as follows- once the instruction block is loaded into the I-cache from the main memory by the Cache Controller module, the Fetch stage fetches the instruction from it and stores it in the IF/ID buffer. The decode stage decodes the instruction from this buffer and checks for branches as well as sends out data to the hazard detection unit to detect data and control hazards. It then passes on the data to the execute stage. The execute stage constitutes the ALU and a few muxes which are used to handle data forwarding. The output data is passed onto the memory stage where it is stored or loaded from the memory based on the and then onto the writeback stage where data is written back into the register file.

**Task Breakdown**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Phase 1** | **Phase 2** | **Phase 3** |
| **Abhijith Somashekhar** | Fetch and Memory stages | Forwarding Logic | Cache Controller FSM |
| **Sidharth Gurbani** | Decoder Stage | Control Logic and Branch prediction | Cache data and metadata arrays |
| **Megh Doshi** | ALU design | Buffer Design and integrating modules | Cache Controller FSM |

**Completeness**

Our design meets all requirements of phase 1,2 and 3. During the demo we did not have correct branch prediction logic and an error in the reduction operation in the ALU which has been fixed. The RED operation had a bug due to omission of a bit which has been resolved now.

**Testing**

A modularised approach was followed in implementing the design with each team member designing the blocks as specified in the task breakdown table. Complex modules were validated on separate testbenches to verify correctness. The whole design was validated by running the provided test cases as well as some of our own test cases.

**Results:**

|  |  |  |
| --- | --- | --- |
| Test Number | Cycle Count | Correctness |
| Test 1 | 42 | correct |
| Test 2 | 56 | Correct |
| Test 3 | 52 | Correct |
| Test 4 | 493 | Incorrect |
| Secret Test Case | 118 | Correct |

Test 4 gives incorrect output due to a bug when there is data forwarding from mem to ex stage. The forwarded data appears to be 16’hXXXX which causes the design to break.

**Post-Demo changes :**

The branch prediction logic had a bug with flushing the pipeline which caused the program to not take the branch. There was also a bug with the RED instruction where the output was not sign extended which has now been fixed.