Project design details and description:

This project has got 7 java source class files. Each java file handling a specific task.

List of classes created in this projects:

**Container class:**

This class holds all the globally accessible values. Follows a singleton pattern to basically provide static contents like of current value of registers, number of instructions fetched, total instructions completed, etc. which always need a current value in every step of execution. Changes made to this file gets reflected immediately.

Methods:

Methods related to Memory initialization and register value initialization are also present in this class.

**Instructions class:**

This class contains information specific to an instruction. The structure defined here gets associated with every instruction object that will get created in the future. Information like the program counter (pc), source register names, destination register names, source register values, destination register values etc. gets created when an object of this class gets created.

Information such as dependency, opcode specific to an instruction, literal values also gets stored here.

Apart from this, information specific to a stage such as execution stage complete, memory stage complete also gets stored for specific instructions.

This file also has a bunch of getters and setters to access these fields.

**MyOpcode class:**

This is an Enum class that stores all opcodes required in this projects. Following opcodes are supported :

***ADD***, ***SUB***, ***MUL***, ***AND***,

***OR***, ***EX\_OR***, ***MOVC***, ***LOAD***,

***STORE***, ***HALT***, ***JUMP***, ***BZ***, ***BNZ***,

***BAL***;

Also contains getters and setters for all these enums.

The design factor behind using Enums instead of strings is that, enums can help catch errors at compile time whereas strings can give run time errors if values don’t match.

**Simulator class:**

This is the main driver class that contains the main method. Hence execution starts from here.

First a read buffer is created to store values read from file that will contain the instructions.

User inputs (simulate, initialize and display) as specified in our project requirement is defined in this class.

**SimBackground class:**

This contains helper functions that will be needed in the background to accomplish the pipeline, display of instructions at different stages and forwarding activities. A majority of the work is performed in this class.

Contains various methods to accomplish this task.

All the stages described in the project are implemented in this class.

The following forwarding scenarios are implemented in this project.

**Scenario 1:**

Load can Forward its value to either sources of <op> instruction or <rsrc1> of LOAD instruction or <rscr2> of store instruction which is in D/RF stage.

**Scenario 2:**

Scenario 2 is also similar to Scenario 1 only the forwarding is done from WB stage as the result is being written to the register file.

**Scenario 3:**

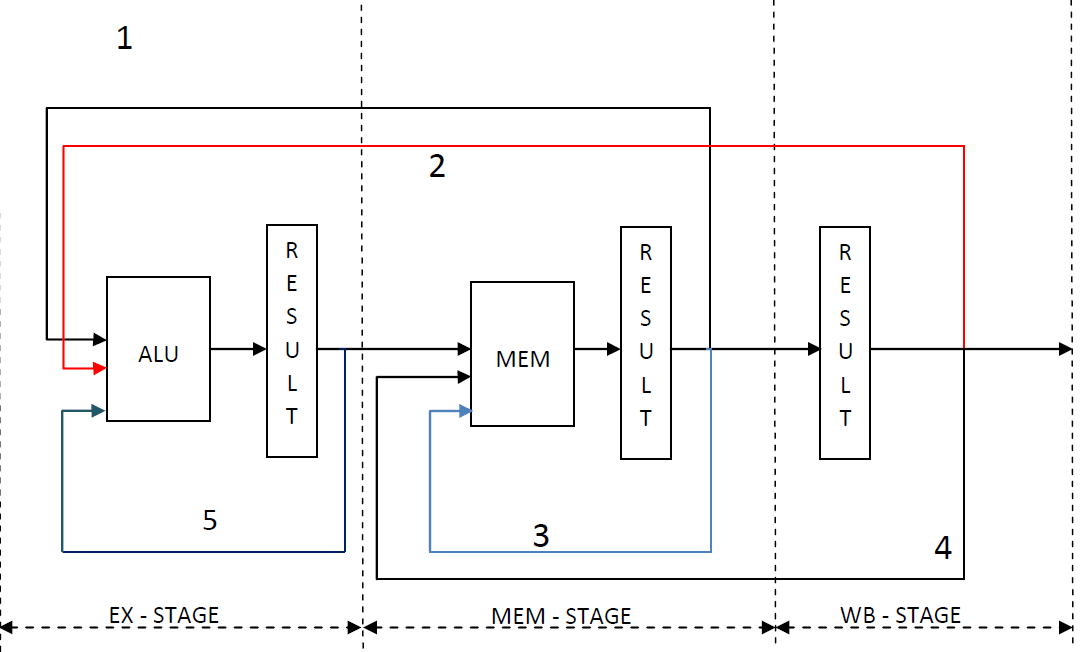
Load can forward the value to <rsrc1> of store instruction, which is coming out of EX-Stage into MEM-Stage.

**Scenario 4:**

Scenario 4 similar to Scenario 3 only from WB stage.

**Scenario 5: EX to EX state forwarding**

Flow dependent Destination values get forwarded as soon as they get calculated in Execution stage.



Working of stages and forwarding:

Every stage information gets stored in a Hash map as a key and value pair since every stage can contain one instruction at any given time. So the key value pair here is a (stage information and its program counter).

To implemented certain forwarding flags are used which are specific to an instruction. Hence this information is stored in a separate class. So every time an object of this class gets created forwarding information can be stored in that object.

These flags and the flow dependency on same register names used as source gets checked with other instruction using the same register name are flagged as dependent instructions and accordingly the dependency is checked in Execution, memory and write back stages.

Any 2 instructions where data forwarding is possible from appropriate stages are treated as non blocking instructions and allowed to pass the decoded stage as the value is guaranteed to be available at the correct stage.

Example: Load Store instruction coming back to back.