The APEX IN-Order Pipeline Simulator has been implemented in JAVA using the following classes:

* Main.java
* Simulator.java
* InitializeDataStructures.java
* Instruction.java
* Operand.java

**Main.java:**

Following are the methods implemented in the Main class along with their brief description:

|  |  |
| --- | --- |
| **Method Name** | **Method Description** |
| public static void main(String[] args) | This is the main method which is running the entire program. It traverses through a switch case for the option we select from the menu on the console. |
| private static void options() | This method prints a menu headed by Simulator for APEX with IN-Order issue and gives us the following options:   1. Initialize: this initializes the program counter to 4000 2. Simulate: this asks the user to enter the number of cycles he/she wishes to simulate the pipeline. 3. Display; this option displays the stages at the end of last cycle simulated 4. Exit: to exit from the menu |

**Simulator.java:**

Following are the methods implemented in Simulator.java along with their brief description:

|  |  |
| --- | --- |
| **Method Name** | **Method Description** |
| public void freePipeline() | The freePipeline method clears the entire lists of all the stages in the pipeline and sets the program counter to the starting value of 4000 |
| public void simulateCycles() | The simulateCycles is one of the most crucial functions of the pipeline code. This function is responsible for simulating through the pipeline cycle after cycle and passing the instruction over the stages. The entire process runs on a loop for the number of cycles entered by the user |
| public void display() | display() function displays the stages after the last cycle has run. |
| private void fetchExecution() | This function gets an instruction from the fetch Stage list into the fetch stage process |
| private void decodeStageExecution() | For decoding the instruction fetched in the fetch stage, the instruction is passed into the decode stage list and the processing for decode happens over here |
| private void aluStageExecution() | If an arithmetic instruction like ADD/SUB or an instruction like LOAD/STORE/MOVC is decoded in the previous stage, then it is passed into the aluStage list for the arithmetic processing. |
| private void mul1StageExecution() | If the instruction decoded is MUL type, then it is sent to the mulStage list instead of the aluStage list for the calculation. |
| private void mul2StageExecution() | Since the latency for MUL type instruction is of 3 cycles, we add an additional MUL stage to decrease the latency to 1. After the mul1StageExecution is complete, mul2StageExecution begins. |
| private void div1StageExecution() | If the instruction is of Div type, then it is sent to the DIV1 stage instead of ALU or MUL stage. The DIV operation takes place in 4 stages. |
| private void div2StageExecution()  private void div3StageExecution()  private void div4StageExecution() | Since the latency for DIV is of 4 cycles, we add 3 stages extra to the DIV1 stage. |
| private void memStageExecution() | After the MUL/ALU execution has happened, the instruction is passed to the MEMORY stage for setting the memory of the destination register |
| private void writeBackExecution() | This is the function responsible for processing the last stage of pipeline. It writes back the value of the destination register and updates it in the register file. |

**InitializeDataStructures.java:**

Following are the methods implemented in InitializeDataStructures.javaalong with their brief description:

|  |  |
| --- | --- |
| **Method Name** | **Method Description** |
| public static Map<> populateInstructionMap() | This method fill in all the instructions in the map |
| public static Map,. initializeArchRegisters() | This function initializes the 16 Architectural Registers to their default value |
| public static int initializeMemory() | This function initializes the code memory for all instructions |
| public static Map initAllInstructions() | This method initializes all the instructions in the map one by one. The destination, sources/literal are set based on the instruction. |

**Instruction.java:**

This function consists of all the variables related to an instruction and certain flags for some stages. It is a helper class where getters and setters are written for these entities:

private String instructionType;

private Operand source1;

private Operand source2;

private Operand literal;

private Operand destination;

private Boolean isMemExecuted = false;

private Boolean isDecoded = false;

private Boolean isALUExecuted = false;

private Boolean isMul1Executed = false;

private Boolean isMul2Executed = false;

private Boolean isDiv1StageExecuted = false;

private Boolean isDiv2StageExecuted = false;

private Boolean isDiv3StageExecuted = false;

private Boolean isDiv4StageExecuted = false;

private int address;

private Boolean isDependent = false;

private Boolean psw = false;

private Boolean isStalled = false;

private Boolean branchFlag = false;

**Operand.java:**

Here we are setting the operand name and value pair for the Register. The variables are:

1. operandName
2. operandValue

The flow of the simulator goes like this:

We first select the 1st option from the menu on screen. This initializes the program counter to 4000. Then we select the 2nd option where we enter the number of cycles to simulate. Upon entering that, the flow goes to the simulator class where it checks for the different lists of the various stages(fetch, decode, alu, etc) and adds the instruction in in-order. The instructions processed in the pipeline are of simple arithmetic operations such as ADD,SUB,MUL,DIV and other operations such as BZ,BNZ,JUMP,HALT and JAL. Upon executing the stage functions, the cycles are printed on the screen. Now when we select the 3rd option from the menu, it calls the display function of simulator.java which will display all the stages at the end of the last cycle.