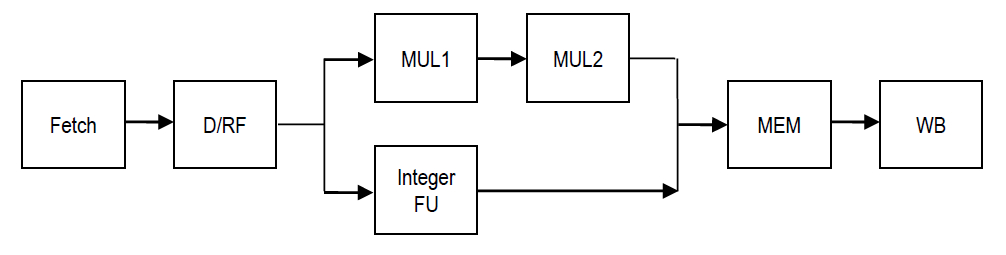
**SIMULATOR FOR AN APEX PIPELINE**

**Problem Statement:**

This project asked to design and code a Simulator for an APEX pipeline which is strictly in-order with two different function units.

Below is the graphical representation of the pipeline:

I have briefly described about the implementation of simulator in the below points:

**Java** is used for implementing the simulator for an in-order Apex pipeline. I have segregated all the stages and **each stage** is a **class**, which basically extends an Abstract class called as **Stage.**

there is a CPUContext Class which basically is the Brain of the code, where registers, memory array for load and store, instruction cache, program counter and instances of each stage is defined.

Our Main method is written in simulator class which basically ask User input from command prompt and act accordingly. On selecting initiate, it will set PC VALUE and reset all the registers, instruction cache, memory array to 0 and in simulation, Code will ask user to provide the number of clock cycle and according to that it will call that number of times to executeCycle method. Which is defined in CPU, this method executes all stages one by one, starting from WB till fetch (Bottom to up).

There is CPUUtils class, it basically has 3 methods, first one, it is used to calculate the registers index, secondly it tells whether the source is a literal or a register and last whether the register is locked or not.

**Algorithm/Flow:** Code have adopted the bottom up strategy, where WriteBack Stage is called first, followed by Memory and so on till it reaches **Fetch** stage. Main idea for starting from WriteBack is that, in this case we don’t need to concern about giving Instruction to next stage in next cycle and parallelly taking a new instruction in the same new cycle (which we need to do if we start executing fetch first).

Whereas, here in my algorithm, Program control starts from WriteBack (WB).

**Dependency:** Code handles true dependency, and for this isSourceLocked method is used which tells the dependency (this is present in CPUUtils) and it is called by D/RF everytime in isInstructionStalled method. Dependency is handled whenever there is a lock due to register.

We also handled the scenario when INT ALU and MUL ALU at same clock trying to move to MEM stage, in our problem MUL will always goes to MEM stage first.

Code makes BZ/BNZ to wait in D/RF stage whenever there is a Arithmetic instruction before it.

**Registers:** There is a register class with name, value and valid as data members, along with this it has 2 Boolean flags. Instance of this registers is created in CPUContext. All the registers are updated using its data member getter and setters. Total 16 registers are present which is defined in CPUContext.

**Memory Array:** A HashMap is created to simulate the RAM memory, here all STORE and LOAD instruction uses the memory location to store and load the data.

**Instructions:** A HashMap is created to store the list of Instruction coming from the input file. Load Instruction method in CPUContext puts the data into instruction map with Key as Program Counter and value as Instruction. Input file is read using scanner. (Each instruction is of 4 bit, thus PC value is incremented by 4). Data Members of Instruction are Instruction, Destination, Source1, Source2, Instruction string.

**ZeroFlag:** It is a Boolean flag used to control BNZ and BZ instruction, it sets whenever any arithmetic instructions give a zero result at WB.

**STAGES:**

**Write Back (WB):** Here Code will set the instruction by taking it from Memory (MEM) stage. If it is null then It will print nothing (which denotes NO-ops) and control will go to Memory stage (which will happen till no instruction reaches Memory instruction and in case of stalls) and but if MEM Stage has an instruction then WB will set that instruction and will write back to the Registers depending upon the instruction and sets the Zero Flag if arithmetic instruction produces a zero value on destination and it also release the locks at the end, After this program control will go to Memory Stage.

**Memory (MEM)**: Here Program’s control will set the instruction from its previous stage i.e. Execution or Multiplication ALU, depending upon the instruction type it will perform memory action. (LOAD and STORE instruction).

**Execution:** This stage will get its instruction from its previous stage i.e. D/RF and do the calculation depending upon the instruction type. There is a different ALU for Multiplication (2 ALU).

**Decode Register Files:** Flow will come to D/RF after Execution stage, here all the instruction will be stalled in case of TRUE dependency. Using **isInstructionStalled** and **isInstructionStalledAtNextStage,** using isSourceLocked method of CPUUtils. Also, here HALT will flush its following instruction once it is decoded. Here In this stage, Code splits Instruction string using regex pattern and stored the split result as instruction’s member in instruction object.

**Fetch:** Program flow reaches here at the last, it will fetch instructions from instruction cache (Implemented in hash map) depending upon the program counter value. Here we also check if fetch instruction is stalled due to Decode RF (which is also stalled due to bubble), This is achieved using **isFetchInstructionStalled** method which checks instruction present in fetch with instruction present in D /RF. Here 2 Boolean Flag are created isBranched and endOfProgram, this isBranched is used in case of Branch case, where we told not to end the program and instead run till the Branch condition is not satisfied, whereas endOfProgram is used to close the program. It will set to true whenever PC gets invalid value or null.

**Project Extension:**

Below are the requirements which are implement with this project:

1. Addition of DIV FU unit. ( Out of Order execution)
2. Forwarding logic for open dependency and PSW Flags.
3. Implementation of JAL instruction.

**Implementation of DIV FU Stage & Output Dependency Handling:**

For DIV FU, we are adding 4 DIV stages same as MUL, with total latency of 4.

Due to presence of DIV FU along with MUL and INT FU, now out of order write back is possible, because of which output dependency needs to be handle. For this, dependent instruction will stall at D/RF until the instruction comes out of WB (Instruction on which other instruction is dependent).

For this, Same system is locking destination register at Execution stage and releasing the same lock at WB back, so whenever an instruction comes to D/RF, it checks its destination registers and if that register is locked by some previous instruction using isDestinationLocked method in CPUUtil class.

Two Methods are present in D/Rf i.e. *isInstructionStalled* and *isInstructionStalledAtNextStage*, if any of these methods returns true, then Instruction need to wait (stall) until, Lock on registers are released from WB stage.

**Implementation of JAL instruction:**

JAL instruction in assembly is used whenever a code is written for calling a function.

JAL has a 3 operands, 2 registers and 1 literal.

Example: JAL R5, R6 #4000

So Here, whenever system decodes It is a JAL instruction, It passes the instruction to INT FU ( if no true dependency is present on R6 with its previous instruction) and In INT ALU it calculates (R6 +4000), this calculated value will be new PC value, so in next cycle, Fetch will take instruction from new PC value and when JAL comes to WB, we are setting the location of next instruction in R5 (current PC value + 4), so this will help us to get back to next instruction once the functions is over, Which is done using JUMP.

**FORWARDING LOGIC:**

Forwarding of data from producers to any dependent instruction will happen only in last unit of ALU (DIV4, MUL2 and IntFU) and only if dependent is present at D/RF,

There is another scenario which is, if any open dependency is present between 2 instructions but the dependent instruction is present at Fetch stage, then no forwarding will happen, and it will stall at D/RF stage till producer instruction comes out of WB.

*Forwarding Implementation:*

We are maintaining buffers in the last stage of each ALU, these buffers hold the values of instruction only till they reach the end of MEM Stage, the instructions in D/RF stage keeps looking for these forwarded values when any of its involved registers are locked. In this way we ensure that only the instruction in D/RF stage can receive the forwarded values.