**CS 520 Computer Architecture and Organization**

**Programming Project 1:**

**Simulator for in-order version of APEX with two separate FUs**

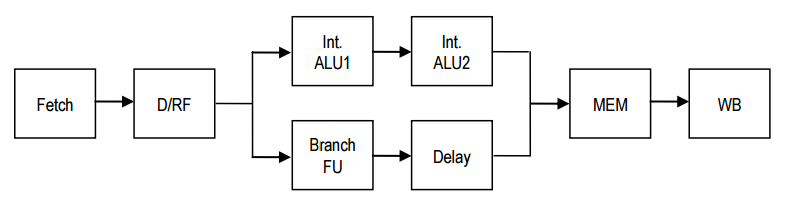
**Design Document**

|  |  |
| --- | --- |
| Rahul Thosar | B00638552 |
| Tasleembanu Halai | B00659229 |

**Prof. Kanad Ghose**

**Objective:**

To implement a cycle-by-cycle simulator for an in-order APEX pipeline with two different function units, as shown below:



*Figure 1.1: APEX pipeline to be implemented*

**Technologies Used:**

Implemented in Java 1.8

Data structures Used:

2D Array – Final array to print result of instruction passing through the pipeline.

Array – Memory of 10000 blocks

Register map – stored as string and integers to represent the registers

Arraylist of Boolean – to store dependencies information

**Apex Processor Description:**

For all instructions, excepting the branches, BAL and JUMP, the operations are performed on a 2-stage integer ALU (with stages Int. ALU1 and Int. ALU2, with a delay of one cycle for each stage). The BZ, BNZ, BAL and JUMP instructions have their target addresses computed in the Branch FU and are all completed in this stage.

**Implementation Logic:**

The data structures used are initialized in the **init()** function where in we have also read the input file with instructions to be simulated.

The normal execution of an APEX pipeline is **Fetch-> Decode -> Execute -> Memory -> Write Back**.

Upon analyzing the process flow in this order, we would require lot many flags and temp variables to enable us to perform an in-order instruction execution. But if we reverse the order and start with write back, this reduces the temp variables.

As per requirement, we have split the stages into 8 stages, as shown in figure 1.1 above and implemented it in the **simulate()** function, which calls the following functions

**Writeback():** Contains logic to write data back to registers according to the instructions. Dependencies set in decode stages are cleared here.

**Memory():** Based on availability of data in delay or alu2 stages, memory function will fetch the instructions from respective stages. It will also perform memory operations in this stage.

**Delay():** This stage takes simply adds a delay of 1 cycle in the branching logic.

**Branch():** The branching logic is implemented in this stage. The branch flags set in decode stage help take a decision whether or not to take a branch.

**ALU2():** Fetching the ALU1 values. Also data forwarding is done from ALU2 to forward data to decode stage for all opcodes other than store.

**ALU1():** Part 1 of execute stage, where in we are performing the arithmetic and load store (Arithmetic) operations.

**Decode():** Contains logic to decode instruction based on the opcode with each instruction. The instructions are decoded in this stage . Based on the register values, the dependencies are also set up here.

**Fetch():** Contains logic to fetch the instructions from arraylist

**How stalls is implemented:**

The instructions are decoded in the decode stage and also the registers in the instructions are marked as dependent. Only once the instructions reach writeback stage, the dependencies are cleared. Untill then, all subsequently fetched instructions in the pipeline are marked as stalled.

**How forwarding is implemented:**

If an instruction is in stall and waiting for a dependency to be cleared up from write back, the info processed in the execute stage is forwarded to the instructions in the decode stage.

**Running the code:**

Simulator is invoked by specifying the name of the executable file for the simulator and the name of the ASCII file that contains the Instruction list to be simulated.

The simulator command interface allows the users to execute the following commands:

* **Initialize**: Initializes the simulator state
* **Simulate**: simulates the number of cycles specified as and waits. Simulation can stop earlier if a HALT instruction is encountered and when the HALT instruction is in the WB stage.
* **Display**: Displays the contents of each stage in the pipeline, all registers (including X) and the contents of the first 100 memory locations containing data, starting with address 0.
* **Exit**:Exits the program.