**CS 520 Computer Architecture and Organization**

**Programming Project 2:**

**Simulator for APEX with four separate FUs**

**Design Document**

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

**Prof. Kanad Ghose**

**Objective:**

For Project 2 we need to extend the simulator developed for Project 1 with a renaming mechanism that uses a unified register file, a centralized IQ and a ROB. There is no need to implement LSQ.

**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 architectural registers, FRT and BRT and URF

Class objects to hold values for ROB, instructions and IQ

Arraylist of Boolean – to store dependencies information

Arraylist of objects for storing IQ entries and ROB entries

**Apex Processor Description:**

Details:

1. Instructions, formats, data memory addressing, instruction accessing etc. are all as in Project1. Forwarding has to be implemented. Be sure to consider all forwarding scenarios! Assume 16 architectural registers (R0 through R15) as before and a unified register file with 32 registers as the default. (The number of registers in the URF can be changed before simulation starts using the Set\_URF\_size command (see next page). The IQ size is 12 and ROB size is 40.
2. The dispatch of a branch or any control flow instruction (BZ, BNZ, BAL, JUMP) stalls till the previous branch or the previous control flow instruction has been issued.
3. In the cycle that a result is being written to a destination register, it can be forwarded to the instruction that needs it. An instruction that needs this result as an input can begin execution in the same cycle in which the result is being written back to the register. That is, forwarding and writeback takes place in the same cycle.
4. Issue of instructions to LSFU take place in program order - issued op stays there, keeping 2nd stage of LSFU busy till the matching in the ROB entry moves to head of ROB. Note that for other FUs, issues can take place out of program order and program order is used only to break any ties in case two awakened instructions need the same FU.
5. Allocate free registers in ascending order of their address. As an example, if P5, P8 and P14 are free, P5 is allocated first. At the end of a cycle, after registers are freed up, they are added to the free list and the free list of physical register is sorted. In the next cycle, the allocation step uses the newly-sorted free list.

**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

**DO\_COMMIT\_ROB():** Once the instruction reached the head of rob, it is then committed and written to the back rename table.

**LSFU():** If store instruction, it will generate the memory address and then store the value at that address. If load instruction, then it will load the value from the specified address.

**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.

**DOfwd():** The Instructions post write back are then checked to have if they have any dependencies set on them, if so, they are cleared in this stage and the data is forwarded to instruction waiting for them.

**WRITEBACKMUL():** The Executed instruction from MUL4() is then written to the registers

**MUL4():** Theexecuted instruction in MUL3 is fetched and passed to WRITEBACKMUL()

**MUL3():** Theexecuted instruction in MUL2 is fetched and passed to mul4()

**MUL2():** Theexecuted instruction in MUL1 is fetched and passed to mul3()

**MUL1():** The instruction passed from IQ are executed in this FU if the opcode is MUL

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

**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.

**PERFORMIQ\_ROB()**: The instructions stay in the ROB for atleast one cycle, until their dependencies are met. Once there is no dependency on the instruction in the IQ, it is passed to the respective FU for further processing.

We also add the instruction in to the ROB queue in this stage.

**Decode2():** The stage 2 of decode, where in the destination registers are renamed and dependencies are set up

**Decode1():** Contains logic to decode instruction based on the opcode with each instruction. The instructions are decoded in this stage. The sources registers are renamed in this stage.

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

**How stall 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
* **Set\_URF\_size <n>**: used before simulation to set the number of registers in the unified register file to n.
* **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.
* **Print\_map\_tables**: prints front rename table and back-end register alias table.
* **Print\_IQ**: prints issue queue entries and their status, one entry per line.
* **Print\_ROB**: prints current ROB contents, one entry per line.
* **Print\_URF**: Prints contents of URF and their status (allocated, committed, free)
* **Print\_Memory <a1> <a2>:** prints out contents of memory locations with addresses ranging from a1 to a2, both inclusive. The addresses a1 and a2 are at 4 Byte boundaries.
* **Print\_Stats**: prints the IPC realized up to the point where this command is invoked, the number of cycles for which dispatched has stalled, the number of cycles for which no issues have taken place to any function unit, number of LOAD and STORE instructions committed (separately).