ECE 401: FINAL PROJECT REPORT
AN OUT OF ORDER SUPER SCALAR PROCESSOR

Introduction: In most of the processors, to increase the performance of the basic MIPS pipeline based processor, instruction level parallelism and structural redundancy is incorporated. In this project, the basic MIPS pipeline (with the memory hierarchy) is expanded to build a dual-issue out of order superscalar processor. The processor is also tested for the IPC with the help of a few benchmarks.

Implementation Details:

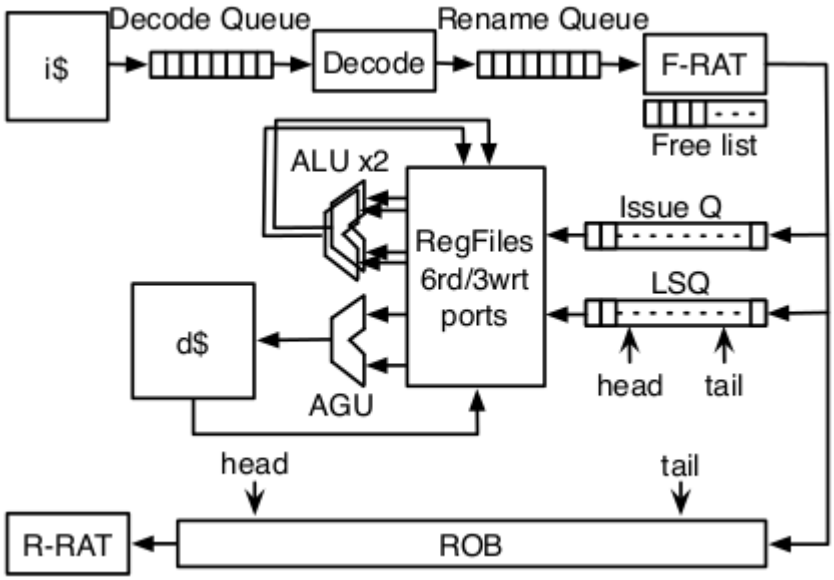

Fig1. The Architecture of the dual issue OoO MIPS processor

- **sim_main.cpp:** Modifications for loading reading two instructions at a time, the FSM was modified to take into account the second instruction too
- **IF.v :** New signals to fetch two instructions in the same cycle and also logic to push data(the fetched instructions + their address) into the decode queue at each clock cycle; a fullflush signal in case of an exception/branch misprediction(this signal is included in all the stages)
- **decode queue:** The basic structure of the queue is implemented in the module called **queue.v**. The queue can push in two sets of data and push out two of its entries at each clock cycle. There 'full' signals indicating whether the queue is full or not. If it is full, then we stall the previous pipeline stage(IF in case of the decode queue and the decode stage in case of the rename queue)
- **iCache.v & instr_cache_core.v:** These modules multi-ported to read two instructions at a time. In most of the cases, the two instructions will belong to the same cacheline; but in case both the instructions miss and their indexes are not the same, then logic to fetch two blocks of data into the instruction cache has been implemented here.

- **ID.v:** In this stage, two instructions from the decode queue is popped and then read into the decoders. There are two decoders (**decode.v** is instantiated) one for each instruction. The output of these is then pushed into the rename queue which is implemented in a similar manner as the decode queue. A mult/div flag is implemented in the decoder module (**decode.v**) and this flag is used in the ID stage. The mult/div flag is used to indicate whether either one of the instructions is a mult or div instruction and if it is, then we drop one instruction (disable the write port of the rename queue for the 2^{nd} instruction) in that cycle so that we have no issues in the rename stage [note that for a mult/div we need to rename the HI and LO registers too; so in order to avoid too many ports to the rename-RAT the above logic is implemented]. When an instruction is dropped, only one instruction is read from the decode queue in the next cycle and then push the dropped and the new instruction read from the decode queue into the decoders.
- **RENAME.v:** In this stage, the decoded instructions are popped from the rename queue and then mapped to the physical register files. The dependencies between the instructions are checked and renamed accordingly. If new physical registers are needed for mapping, then the free list is accessed. The free list is also a FIFO and has 32 entries (since at any point of time 32 of the physical registers will be mapped to the architectural registers and the HI and LO registers). The F-RAT has 34 entries (including the HI and LO) and is indexed using 5 bits. If the pipeline is flushed, then the retirement RAT from the commit stage is written into the F-RAT. We also push the instructions in program order into the Reorder Buffer (ROB).
- **DISPATCH.v:** In this stage, the instructions from the rename stage are dispatched to the issue queue. There is a structure called busy bits which indicates whether a physical register is blocked by an instruction in the issue queue. After the dispatch stage, resources can't be allocated. The dispatch stage is stalled if the ROB/issue queue/ load store queue is full. The required data/signals for both the instructions are pushed into the issue and load-store queue. Note that we have implemented wakeup and selection logic for the issue queue.
- **busybits.v:** multiport the busybits reg; the dependencies are also resolved to avoid conflict between the two instruction's resources which are issued at the same time.
- **issue_queue.v:** In this stage, the required signals for each instruction are entered into the issue queue if the issue queue has at least 2 available entries. There are ready bits for each entry of the queue which are set when the sources become available. A reg called hold_bits(one for each entry in the queue) is also included in the logic in this stage. As discussed in class, this is for the purpose of **mini speculation** for eg. in the case of an instruction which uses a value which is supposed to be loaded by the previous instruction (ld R1← 'X' , add R3 ← R1, R2). If we speculate wrongly, we flush the pipeline.
- **READREG.v:** This stage basically has the 64 physical registers with 3 write ports and 6 read ports. The values from the registers are read in this stage before moving onto the EXE stage.
- **EXE.v:** In this stage, all the instructions are executed. There are two ALUs and one AGU. In case of a mult/div instruction, the ALU0 alone is used. The AGU is used to calculate the effective address for memory instructions. There is forwarding of signals from the retire and commit stage. We also need the hold bits in this stage.

- **MEM.v:** This stage reads in the load/store instructions in program order from the ***load store queue***. We only write into a memory location once the instruction is the head of the ROB. Most of this stage is similar to the basic MIPS pipeline.
- **RETIRE.v:** In this stage, as explained in class, necessary signals like a branch misprediction or syscall, target address, branch/jump flag, memflag(flag to indicate write/read mem operations) are written into the ROB. In case of a branch misprediction, we have to flush the pipeline! We also write back the output values to the register file (kind of similar to the write back stage of a basic MIPS). Two instructions are retired in the same cycle.
- **COMMIT.v:** In this stage, we have the ROB and the retirement RAT. Here also, two instructions are committed at the same time and not just the head of the ROB (but in program order). Only instructions which are marked as ready can be committed. Once instructions are committed, the registers are recycled and returned to the freelist. The committed instructions write the registers into the retirement (declared as replaceRAT) RAT. In case the pipeline needs to be flushed, we copy the register mapping in the replaceRAT to the F-RAT as was mentioned in the rename stage.