CS 520 - Fall 2020 - Ghose

Programming Project 2: Simulator for an out-of-order implementation of APEX

DUE: Sunday, December 6 by 11:59 pm via Blackboard

NO LATE SUBMISSIONS ALLOWED

THIS IS A TEAM PROJECT: UP TO 3 MEMBERS PER TEAM

Demo Dates: TBA

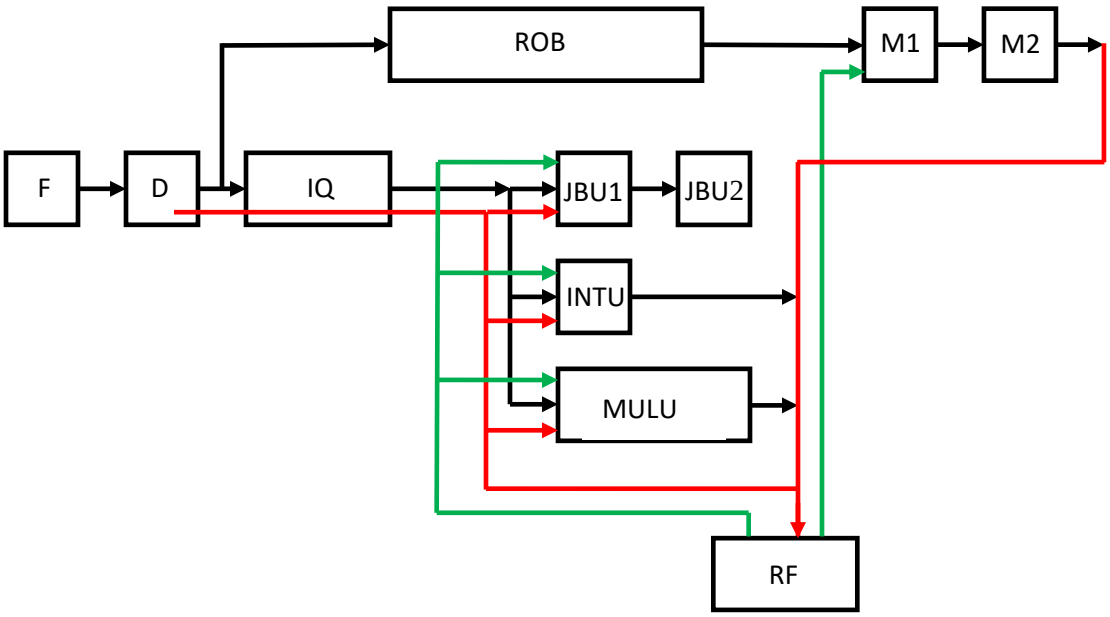
INFORMATION ABOUT SUBMISSIONS AND DEMOS FORTCHCOMING.

DO NOT USE ANY CODE OTHER THAN THE STUB CODE THAT WAS GIVEN OUT AND YOUR OWN CODE. USE OF ANY OTHER CODE WILL BE CONSTRUED AS PLAGIARISM.

This project requires you to implement the simulator for an out-of-order version of APEX with additional instructions. Your submission will have two parts as described later.

PROCESSOR ISA AND DATAPATH OVERVIEW FOR PROJECT 2

The processor to be simulated in Project 2 uses register renaming and is like the processor shown as Variation 3 on Page 203 of the notes, but it **does not** have a LSQ. It is shown in the figure below:



In this diagram, the paths marked in red are used for forwarding and writing a result into their destinations. The paths marked in green are used for reading out operands for the unified register file, RF. Rename tables

and other structures are NOT shown. ***Assume that all of the paths shown in the figure above are capable of transferring as much data/information are needed simultaneously.***

The ISA and specific details of the processor are now described below.

ISA for the Processor to be Simulated for Project 2: The ISA for Project 2 includes all of the instructions present in the ISA for Project 1 as well as the following additional instructions:

- JUMP <src1>, #<signed_literal>: This causes control to flow to the instruction at the address obtained by adding the contents of source register <src1> with the signed literal and all instructions that followed the JUMP instruction into the pipeline are squashed.
- JAL <rdest>, <rsrc1>, #<signed_literal>: This instruction ("jump-and-link") is identical in behavior to the JUMP instruction but before transferring control to the instruction at the address obtained by adding the contents of <src1> register with the signed literal, it saves the address of the instruction immediately following the JAL in the destination register <rdest>.

The JAL instruction is used to implement a function call and saves the return address in the destination register named in the instruction. As an example, the instruction:

JAL R2, R5, #32

Calls the function whose first instruction is at the memory address obtained by adding the contents of R5 with literal 32. If the memory address of the JAL instruction is 64, the instruction following the JAL has the address 68 (= 64+4), and this is saved as the return address in R2. A return from the function called in this case can be implemented by the instruction:

JUMP R2, #0

This assumes that R2 is not altered by the called function and anything else it calls. If R2 is used in the called function, the called function must save it on a stack in memory and restore it to R2 before executing the JUMP R2, #20 instruction to return to the caller's code.

The memory operation for LOADs and STOREs are started when these instructions move up to the head of the ROB and have all operands available to perform a memory operation.

Who can set the Z (Zero) flag: For Project 2, CMP, SUB and ADD can all set the Z flag. Dependencies over the Z flag will have to be handled properly.

PROCESSOR DATAPATH DETAILS

Specific details of this processor are as follows:

Key Structures Supporting Out-of-order Execution: These are as follows:

- The unified physical register file contains 48 registers.
- There is a 64-entry ROB.
- There is a centralized issue queue (IQ) with 24 entries.

LOADs, STOREs and Memory Accesses with the ROB: Register values needed by LOADs and STOREs are forwarded to the respective ROB entries and when the ROB entry for a LOAD or STORE moves up to the head of the ROB and has all register operands available (that is, ready), a memory operation is initiated on a dedicated pipelined memory unit that has latency of 2 cycles (one cycle for computing the memory

address and one cycle for accessing the D-cache. Assume that all D-cache accesses result in a hit for this project.

Issue Queue and Instruction Issue: Physical register tags are used to wake up instructions waiting in the IQ. The IQ uses issue-bound reads and assumes that tags are broadcast one cycle before that data is available, so that source operands are read out of the RF and any register operand the issued instruction is forwarded to it as the issued instruction enter the requisite function unit. *You will **not** need to simulate the exact tag broadcast timing and assume that the instructions begins execution on the required function unit as soon as its last operand is produced. This effectively implements the same timing you will get if tags were broadcasted prior to the availability of the corresponding result.*

Instruction Issue Prioritization: If two instructions waiting in the IQ demand the use of the same function unit at the time of issue, the one that was dispatched earlier is chosen for issue. *To implement this, you will need to add a field to each IQ entry to note the clock cycle when it was dispatched (starting from the beginning of simulation).*

Function Units: The following function units are available:

- Integer, logical operations and the CMP instruction are completed in a single cycle in the INTU shown.
- A MUL(tiply) instruction takes 3 cycles on a monolithic (= non-pipelined) function unit, the MULU. *For this project, assume that the result of a MUL can fit into a single register.*
- A 2-stage pipelined function unit, consisting of stages JBU1 and JBU2 for handling branch and jump instructions. The branches use JBU1 to resolve a branch, compute a target address and transfer control along the actually-determined control flow path. The JUMP and JAL instructions use JBUI and JBU2 to transfer control along the actually-determined control flow path. JBU1 and JBU2 have a delay of one cycle each.
- A 2-stage memory access unit, consisting of stages M1 and M2 for accessing the memory when the LOAD or STORE instruction is at the head of the ROB and when the instruction is ready to access memory. M1 and M2 each have a delay of a single cycle. The LOAD or STORE completes in M2. The LOAD broadcasts its tag and data, updates the destination register at the end of the cycle it is in M2.

Branch Prediction and Speculative Execution: The processor uses a branch predictor that keeps the precomputed target address for branches AFTER the very first time the branch is encountered. Assume a fully-associative BTB (that is, entries are looked up using the branch instruction's address). The BTB lookup proceeds in parallel with an instruction fetch. The BTB entry for branches have the following fields:

- The address of the branch instruction
- A field indicating the actual outcome of the immediately preceding instruction. When the entry for a branch instruction is set up in the BTB, a **default prediction of not taken** is used. This forces the branch instruction to actually use the stage JBU1 and calculate a target address, which is stored in the next field for subsequent use, as well as current use if the default prediction of not taken is wrong (that is, the branch is actually taken).
- Computed branch target address.

For branches, the prediction made is the same as the outcome of the last actual execution, as recorded in the BTB entry.

A speculation depth of 4 is supported. Instructions along the mispredicted path following a branch needs to be squashed. You will need to implement the equivalent of a BTS.

For simplicity in coding the simulator, assume:

- *No replacements are needed from the BTB and it has a capacity of 8 entries to accommodate up to 8 branches at most in the binary being simulated.*
- *BTB entries can be updated without any delay.*
- *All relevant structures (rename table, list of free registers etc.) are checkpointed at the time of dispatching a branch instruction.*
- *Restoration of IQ, ROB, rename table, free list, BTS etc.* ***takes place immediately on resolving a branch****, in the same cycle that branch is in the JBU1 stage.*

JUMP and JAL Instructions: Obviously, no predictions are needed for these. Both instructions require the use of the stages JBU1 and JBU2, as noted earlier.

Data Dependencies: Data dependencies over the Z flag (the only CC-code for this project), the destination register for a JAL and all other architectural registers are handled properly using the renaming mechanism.

Other Timing Details: Assume the following timing for other operations:

- Register file, ROB updates are made at the end of the cycle in which the result is produced. The updates are instantaneous.
- Instruction decoding, renaming and dispatching all take place under appropriate condition and takes place at the end of the (last) cycle an instruction spends in the stage shown as D. The qualifier "last" alludes to the possibility that an instruction may stall in the D stage.
- The instruction fetch stage (F) has a delay of one cycle.
- For instructions other than LOADs or STOREs, tag and data broadcasting, updates to registers etc. takes place when the entry spends its (last) cycle at the head of the ROB. Again, the qualifier "last" alludes to the possibility that an instruction may stall at the head of the ROB stage.

Clarifications/specifications, if any and needed, for any other aspect of timing will be added later.

Simulator Commands and Displaying Results

The requirements are the same as in Project 1, with the following additional commands:

- PrintIQ: this prints out all of the contents of the entries in the IQ.
- PrintROB: this prints out the contents of the four consecutive entries in the ROB, starting with the entry at the head of the ROB.
- PrintBTB: this prints out the contents of the BTB.

You can add other commands to support debugging.

SIMULATOR VARIANTS TO BE SUBMITTED AND OTHER REQUIREMENTS

TWO different versions of the simulator (Project2_Version1 and Project2_Version2) need to be submitted by all teams. These are as follows:

Version 1: This does not support speculative execution and instruction dispatches are stalled on encountering a branch instruction. Dispatching continues once the dispatched branch instruction is resolved.

Version 2: This version supports speculative execution based on BTB predictions as described earlier.

Documentation and submission requirements: You will need to submit a documentation that details the role of every team member, what part of the code they worked on etc. Make sure team members share their responsibilities equally. You will submit a zip file with the code/folders for BOTH versions of the simulator. You can also submit any test code and instructions for running your simulator.

PLEASE START EARLY