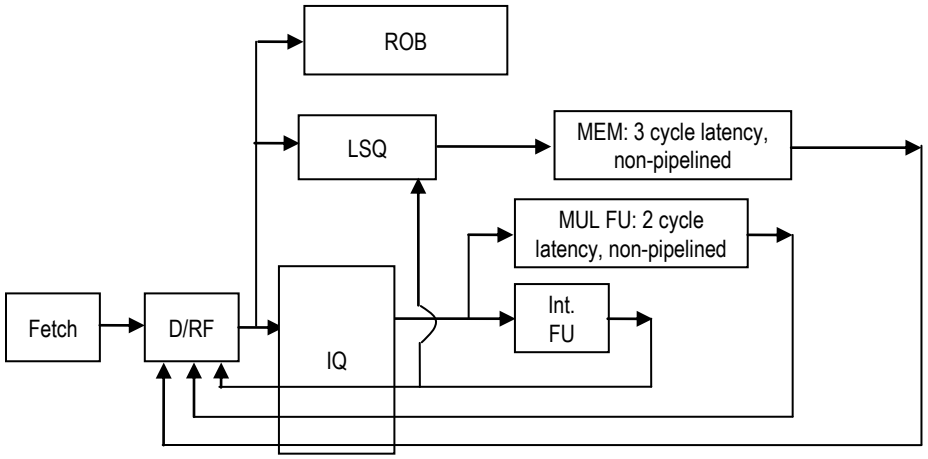
CS 520 – Fall 2018 – Ghose

Project 2 – Team Project – At most 2 Students per team
Due: Monday December 10, 2018. midnight
All demos to be completed between Dec. 12 to 14, 2018

Please see submission requirements at the end of this document

For Project 2, you have to implement the simulator for an out-of-order version of the simulator for the APEX ISA like Variation 3 of the notes. The simulated processor makes use of register renaming with three function units, a centralized issue queue, a LSQ, a ROB and a unified register file (URF). The datapath of this processor is shown in the diagram below:

Connections to the unified register file, front-end and back-end rename tables are **not** shown



1. DATAPATH OVERVIEW

The function units in the processor datapath shown above are as follows:

- An Integer FU that implements all integer operations other than multiply, memory address calculations (for branches, calls and jumps, loads and stores) and bitwise logical operations.
- A multiplier for implementing multiplication operations and this is non-pipelined and has a latency of 2 cycles.
- A memory function unit, which is non-pipelined and has a latency of 3 cycles.

The processor to be simulated is similar to Variation 3 as described in the class notes and has the following features:

- It has the same instructions as the processor you simulated for Project 1, **plus some other instructions** as described later in Section 2.
- The only flag of concern is the zero flag, **Z.**

- The processor uses **register renaming** with an IQ, ROB and LSQ. A **unified register file** is used, along with a front-end rename table and a back-end RAT. The issue queue entry holds literal operands and all register operand values in addition to all other relevant information that is needed.
- Memory addresses are calculated on the IntFU and written to the LSQ entry for LOADs and STOREs. Memory accesses take place from the head of the LSQ using the stage MEM, which is not pipelined and has a 3-cycle delay. Each memory access thus takes 3 cycles. The **LSQ does not support load bypassing or store-to-load forwarding** based on memory address matching.
- **Three dedicated forwarding/broadcast buses exist** to the IQ for simultaneous use by the IntFU, MUL FU and MEM in a single cycle if needed. The unified register file, IQ, LSQ and ROB each have enough ports to support as many concurrent reads and writes that are needed in a single cycle.
- All decoding, renaming and dispatch steps are performed in a single D/RF stage.
- The processor supports **back-to-back execution**, with the tag (which is a URF register address) broadcasted **one cycle before** the result is available.
- As in Project 1, the target address of the branches, JUMP and JAL (see later) are computed in the IntFU.
- A field is added to every register in the unified register file to hold the flag value produced by any associated arithmetic instruction. The other field in the register holds the register value. For Project 2, the ONLY flag that you have to deal with is the Z flag, so you will need a 1-bit extension for each URF register. As an example, when you have a SUB instruction that uses register U5 as a destination, store the result of the SUB in U5 and hold the Z flag produced by the SUB (which would be a zero or a one) in the 1-bit extension of U5. The rename table and back-end register alias table will need an entry for the Z flag.

The configuration details on the number/capacities of the key datapath components are as:

Architectural registers: 16 (Numbered R0 to R15), unified registers: 40 (Numbered U0 to U39)

IQ capacity: 16 entries, ROB capacity: 32 entries, LSQ capacity: 20 entries

The "display" command needs to print the contents of all major components that are simulated (rename table, LSQ, IQ, ROB, PRF, Fetch PC, rename table and back-end register alias table, elapsed cycles, CPI up to the last simulated cycle etc.

2. NEW INSTRUCTIONS TO BE ADDED

In addition to the instructions that were implemented for Project 1, some new instructions are to be supported. These are:

- ***A jump-and-link instruction, JAL to implement a function call***. The format and semantics of the BAL instruction is as follows:
  - Format: JAL <dest>, <src>, <#literal>, as in: BAL R3, R2, #-24
  - Semantics: Saves the address of the instruction following JAL (PC-value of BAL + 4) in the register named in <dest> and transfers control to the address of the first instruction in the called function ("target address"), which is aligned at 4-Byte boundary. The target address is obtained by adding the contents of the register named in <src> with the literal. All instructions that have followed JAL into the pipeline are flushed. As in Project 1, the target address of JAL has to be converted to the address of a line in the text file that contains the code of the program whose execution is being simulated.

  A return from this function call is simply implemented as a JUMP <dest>, <#literal>. where the literal value is zero, as in JUMP R3, #0 used to return from the function called using JAL R3, R2, #-24.

- ***Add and subtract instructions, ADDL and SUBL, which specify a literal operand instead of a second source register.*** An example of such an instruction is ADDL R1, R15, #4, which adds the contents of R15 with the literal value of 4 and writes the result into R1 and updates the Z flag. SUBL is similar. The literal value can have a sign, as in SUBL R2, R14, # -8.

3. DATAPATH AND OTHER DETAILS/SUGGESTIONS

3.1. Registers, connections, commitments, breaking ties for the selection logic

- In addition to holding a result, each URF registers has an **extension to hold the Z flag** that was generated if the content was produced by an arithmetic instruction. All URF registers also have an additional **one-bit status flag** that indicates if their contents are valid.
- The **front end rename table** (or, simply, rename table) points to the most recent instances created for architectural registers and the Z flag in the URF. For each architectural register and the Z flag, the **back-end register alias table** points to the committed register (or flag) value in a URF register.
- The D/RF stage can read source register values and check their status flag.
- **At most two instructions can be committed per cycle starting with the one at the head of the ROB**. Do this whenever possible.
- **Each FU has its own selection logic**. Ties to decide which instruction to issue to a specific FU are broken by issuing the earliest dispatched instruction that is eligible for issue. To implement this, add a field to each IQ entry that holds the cycle count at the time of dispatch. Do not use the instruction address as the dispatch order, as branch instructions can transfer control to an earlier instruction.
- **Forwarding connections exist to forward concurrently** from the last stages of the FUs and the last memory access stage – all in the same cycle if need be.
- An instruction being dispatched in a cycle **cannot** issue in the same cycle even if all issuing conditions are valid. An instruction spends at least one cycle in the IQ.
- An issue queue (IQ) entry for a LOAD or STORE is established to compute the memory address targeted by a LOAD or STORE. Once the entry is issued from the IQ, it needs to be removed from the IQ. The issued operation (an addition) carries with it the index of the LSQ entry to let the computed address to be written to the LSQ entry **using a dedicated connection (and LSQ port).**
- A free list of the URF registers is separately maintained – **for allocation, this is always scanned from the lowest register address (0) to the highest (39)** to identify a free register. You can implement the same function by adding **a free/allocated bit to each URF register** and scanning these bit in the order stated.
- Deallocate registers in the URF when a **renamer instruction** is committed (as described in the class notes for Variation 3 of out-of-order processors using renaming).
- Update the front-end rename table and back-end register alias table as required.

3.2. Handling control flow instructions (BZ, BNZ, JUMP) with an issue queue

With an issue queue, several instructions following the control flow instructions (including other control flow instructions) can enter the pipeline - these following instructions can be dispatched and (for the BZ, BNZ, JUMP) can issue and execute before the control flow instruction. When the control flow instruction issues, the instructions that followed it in program order will have to be located **wherever they are in the pipeline and flushed** depending on the control flow path. Here's how you can simulate this requirement correctly. The

mechanism described is very similar to what has been used to support speculative execution (as in the class notes):

3.2.1 Labeling and Recording the Program Order of Control Flow Instructions

Every control flow instruction gets a unique id (label) that comes from a pool of free ids - you can assume that 8 such ids (called **CFID**s) are available for this particular pipeline. Set up a FIFO queue to hold the CFIDs for all dispatched instruction - this queue is called **CF_instrn_order**.

3.2.2. Dispatching a Control Flow Instruction

At the time of dispatching a control flow instruction, set up a ROB entry and an issue queue entry.

At the same time a free CFID is assigned to the branch and recorded in its ROB entry. If a CFID is not available, stall dispatch till one is available. The IQ entry for a control flow instruction records the location of its ROB entry.

On dispatch, the CFID is also copied to a global variable, **Last_control_flow_instrn**. Add the label of the dispatched branch to the tail of CF_instrn_order.

Finally, the URF status **and values** (status of each URF register **and their contents**) and rename table contents *just prior to this dispatch* are saved and a pointer to the structure containing the saved information is stored in the ROB entry of the dispatched control flow instruction, along with the index of the entry created for the control flow instruction at the tail of CF_instrn_order.

3.2.3. Subsequent Dispatches

All subsequent instructions that are dispatched, including and up to the next control flow instruction are marked with the CFOD in Last_control_flow_instrn. This mark moves with the instruction in the pipeline.

3.2.4. Resolving Control Flow Direction

When a control flow instruction is issued and in IntFU, if no instructions need to be flushed, simply add the CFID of this instruction back to the free list of CFIDs. If all instructions following it need to be flushed, do the following:

1. Stall dispatch. (This will also stall the stages that precede the decode/rename/dispatch stage.)
2. Flush all ROB entries that follow this control flow instruction's entry in the ROB and use the pointer to the saved information (as stored in the ROB entry) to access the saved information on the URF and rename table. Use the saved information to restore the status **and contents** of the registers in the URF and the rename table. (If you have used a separate structure to list URF registers, you have to restore that using the saved info.)
3. Locate the index of the corresponding entry in the CF_instrn_order queue. Starting with the entry so located in the CF_instrn_order queue to all the way to the most recent entry created at the tail of this queue, locate all instructions marked with the CFID stored in the queue entry- wherever they are - and convert them to NOPs and add these CFIDs back to the pool of free CFIDs.
4. Resume dispatch if the control flow instruction was not a HALT.

3.3. Handling the HALT instruction

The simulation of a HALT is easy with a ROB in place: when D/RF encounters a HALT, it stalls the D/RF stage and adds ONLY a ROB entry for the HALT at the end of the ROB. An IQ entry is not needed. When the ROB entry for the HALT reaches the head of the ROB, simply return control to the user prompt!

3.4. LOAD/STORE INSTRUCTIONS AND THE LSQ

Memory operations are started when the LSQ entry for the LOAD or STORE is at the head of the LSQ and when all pertinent conditions are valid. LSQ entries need to be removed once the memory operation has started – this is detailed for a STORE instruction below. A LOAD is handled somewhat differently - you need to figure this out!

A STORE can consider starting its memory operation when all of the following four conditions are valid:

1. The targeted memory address is computed
2. The value of the register to be stored is available
3. The STORE's ROB entry is at the head of the ROB
4. The STORE's LSQ entry is at the head of the LSQ

The need for conditions (1) and (2) are obvious. Condition 3 is needed **as memory writes are irreversible** and one needs to ensure that all prior instructions have committed before an irreversible change is made to memory. Condition 4 is needed to comply with the sequential execution model. For this Project 2, a fifth condition needs to be valid before the memory write for a STORE can be started - the non-pipelined Mem unit has to be free.

When Conditions 1, 3 and 4 are valid, you can wait for the register valid bit of the register to be stored to become a one (that is the register value is valid (= available)) and check that the non-pipelined memory unit is free before starting the memory write for the STORE.

Once the STORE's memory operation starts, you can remove the ROB entry and let ROB operations continue as usual from the next cycle. The reason is that we assume memory writes to not fail in the middle. (In a real machine the write goes thru the cache hierarchy and the WB buffer.) If the memory write eventually fails, we have and the program exec an irrecoverable error and program execution is suspended.

4. TIMING REQUIREMENTS TO BE IMPLEMENTENTED

The write to a URF register and the commitment of the corresponding instruction **cannot** take place in the same cycle. Specifically, consider the entry at the head of the ROB in cycle T. If this entry (and the register it points to) is updated in cycle T, the commitment of this instruction takes place in the next cycle, T+1.

The timing for the LOAD for dispatch, selection, issue and execution is as follows; many of the steps are similar for other instructions.

Cycle T:

LOAD is in D/RF - entries created in IQ, ROB, and LSQ. Source operand availability is checked. LOAD moves to IQ at end of cycle T:

> **Case 1:** If the source register value is available, it is copied into the source operand field of the IQ entry that is created in this cycle.
>
> **Case 2:** If the value needed by LOAD is being broadcasted in this cycle, that value is also picked up by the LOAD ***as it is dispatched***.
>
> **Case 3:** If only the tag that matches the source URF register of LOAD is broadcasted, a tag match is noted to allow LOAD's IQ entry to pick up the value when it is broadcasted in the next cycle. (Remember that tags are broadcast one cycle before the corresponding data is available to support back-to-back execution.) ***However, LOAD will not be selected for issue at the end of Cycle T, as it's IQ entry is completed only at the end of Cycle T, giving the selection logic no time to go through the selection process to begin execution of the LOAD in Cycle (T + 1).***

Case 4: The source register value is not ready and neither a tag nor the value is broadcasted in this cycle: in this case, the IQ entry simply notes that src 1 is not available and the src1 tag field is set appropriately.

Cycle (T+1):

LOAD is already in IQ:

Cases 1 and 2 (matches Cases 1 and 2 above): selection logic examines IQ entry of LOAD and may select LOAD to move into the IntFU ***at the end of this Cycle.*** Thus, LOAD spends a minimum of one cycle in the IQ.

Case 3 (matches Case 3 above): LOAD's IQ entry picks up the forwarded value it needs. Here also, the selection logic may select the LOAD for issue to begin execution in **Cycle (T+2) - see Case 3 above**.

Case 4 (matches Case 4 above): LOAD's IQ entry checks for the tag broadcasted ***at the beginning of this cycle.*** Here again, the selection logic may select the LOAD for issue to begin execution in **Cycle (T+2).** If the LOAD's IQ entry was selected for issue to begin execution in Cycle (T+2), the forwarded value is picked up by the LOAD ***at the end of Cycle (T+1), as it is moved to the IntFU***. Note the implication on FU timing - in the last cycle of execution, the result itself is broadcasted - this is what you can assume for simulation. In a real machine, this happens immediately after the last execution cycle ends and just as the dependent begins execution. You can, of course simulate the timing of the real machine.

Cycle (T+2):

This step applies only if the selection logic chooses the LOAD's IQ entry to begin execution in Cycle (T+2): LOAD computes memory address and writes this data into the LSQ entry at the end of Cycle (T+2).

Memory operations are started when the LSQ entry for a LOAD or STORE is at the head of the LSQ.

5. SUBMISSION REQUIREMENTS

- Since this is a team project, team members MUST document their contributions – what portions they have contributed to actively. Team members must contribute equally on all aspects of the project. This document is part of the submission requirement.
- The "simulate" command have to be modified if needed to support simulation for N cycles ("simulate N") and then enable the display of the simulated processor/memory status using "Display". A subsequent "simulate M", say, should be able to **continue** simulation for the **next** M cycles from the state where it left off at the end of the N-th cycle and then have the ability to display the state etc.
- Other submission requirements will be announced later.

In consideration of the schedule of the TAs, please sign up for the demos as soon as the signup arrangement is announced.