CS 520 - Fall 2023 - Ghose

Programming Project 3: Simulator for Out-of-Order Processor supporting speculative execution using register renaming

DUE: Friday, December 8, 2023 by midnight via Brightspace
Demos scheduled with TAs will require appointments: Details will be Posted Soon

This is a project that has to be done by a TEAM of 2 to 3 Students

START WORKING ON THIS AS SOON AS POSSIBLE

Soft copies of all documents and code to be submitted via Brightspace by midnight on Friday, Dec. 8. You also need to demonstrate your simulator to one of the TAs.

This document *may* be revised with clarifications and any other details that are needed in the next few days.

PROJECT DESCRIPTION

This project requires you to implement a simulator for an out-of-order processor implementing an APEX-like ISA with an issue queue, a load-store queue and a reorder buffer that uses register renaming. The instructions are similar to the ones in Project 2, but **BN and BNN are excluded for this project**. The datapath of the processor to be simulated is shown on the following page.

Processor Details and Datapath

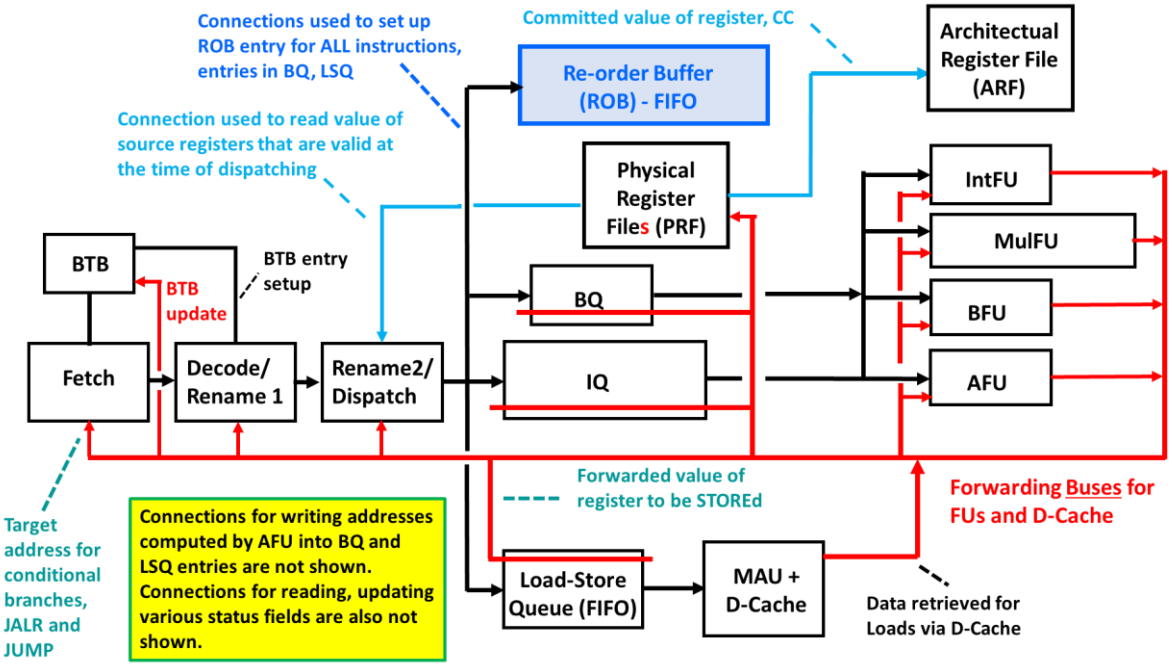
The specific details of the processor are as follows:

- The ISA of the processor simulated is as follows, with semantics as defined in Project 1:

Function of instruction(s)	Symbolic Name
Writing a literal value to a register	**MOVC**
Memory accesses	**LOAD, LOADP, STORE, STOREP**
Arithmetic operations	**ADD, ADDL, SUB, SUBL, MUL**
Bitwise logical operations	**AND, OR, EX-OR**
Comparing register value(s)	**CML, CMP**
Conditional branching	**BZ, BNZ, BP, BNP**
Unconditional control transfer	**JUMP**
Function call	**JALR**
Miscellaneous	**NOP, HALT**

- **16 architectural registers** (R0 through R15) and **a condition code register CC**, are supported in the ISA. Each of the architectural registers, R0 through R15, holds an integer value. Only the **Z**(ero) and **P**(ositive) flags are supported in the current implementation.

- **Register renaming is used. There are 25 physical registers** (P0 through P24) to hold values. **A separate set of 16 physical registers**, CP0 through CP15 implement instances of CC. Each of physical registers P0 through P24 has a valid bit, a data field (to hold an integer value). Similarly, the physical registers CP0 through CP15 for the flags have a single valid bit and a field to hold the flag values. Physical registers are freed up ***when the renamer instruction commits***. The renamer for an architectural register is defined as in the lectures; the renamer for the CC is defined analogously – as the next instruction in program order that updates CC. The renaming table has a separate entry for CC.



The Datapath of the Processor to be Simulated for Project 3. **The PRF shown actually consists of two different sets of physical registers (P0 through P24 and CP0 through CP7, respectively, as described)**

- The processor incorporates a BTB identical to that used in Project 2. The branch prediction policies and the default history bits are also as defined for Project 2. The roles of the Fetch, Decode/Rename 1 and Rename 2/Dispatch stages are as presented in the lectures, with appropriate changes needed for accommodating the BQ (and the AFU) – see below. Each of these stages have a one-cycle delay.
- The processor supports **speculative execution** and **limits the maximum speculation depth to 4**.
- There are two separate issue queues – one dedicated to conditional branch, JALR and JUMP instructions, called **BQ** (**B**ranch instruction **Q**ueue) and another dedicated to all other instructions, the **IQ**.
- **Source (physical) register operands are read at dispatch time** if they are valid. A source register whose value becomes valid after the dispatch of the consuming instruction uses forwarding to deliver the value to the consumer instruction as it waits in its respective issue queue (BQ or IQ). An instruction issues out of the BQ or IQ when all of its operands are valid. For both queues, the earliest dispatched instruction is selected for issue should multiple instructions become ready for issue in the same cycle. ***Details of the entries in the queues and relevant timing information is described later.***
- To support speculative execution based on the prediction made by the BTB for the *conditional branches* BZ, BNZ, BP and BNP, the free list of both types of physical registers and the rename table are **checkpointed** when these instructions are dispatched.

- The processor supports a load-store queue, **LSQ** to maintain the dispatch order of load and store instruction. No form of bypassing is supported by the LSQ for speeding up memory operations. The LSQ supports a total of 16 entries.
- A 32-entry reorder buffer (ROB) is used. ALL dispatched instructions have an entry in the ROB. An **18-entry architectural register file, ARF** is supported. **The first 16 of these are for the committed values of R0 through R15. The 17-th and 18-th entries are, respectively, for the CC Flags and the address of the last committed instruction.**
- The processor has the following function units:

1. A dedicated FU, **AFU** that computes the memory addresses for the load and store instructions, as well as the target addresses for BZ, BNZ, BP, BNP, JUMP and JALR. **The AFU delivers the computed address to an entry in the LSQ or BQ using the index of the entry in the LSQ or BQ**. Note that the BQ does not have to be a FIFO queue (like the LSQ) to allow this. The AFU takes one cycle to compute an address and writes this result into the appropriate LSQ or BQ entry ***in the same cycle***. **Assume that dedicated connections exist from the AFU to the IQ or BQ** for these deliveries. **For LOADP and STOREP, the increment to the register used for memory address calculation is also performed in parallel with the memory address calculation in the AFU.** A dedicated incrementor is part of the AFU.
2. A branch function unit, **BFU**, that implements the actual control flow transfers for BZ, BNZ, BP, BNP, JUMP and JALR, detects and corrects for misspeculation. Assume that branch resolution, discovery of a misprediction and flushing for misspeculation and state restoration (free list and rename table) is all implemented by the BFU **in a single cycle**.
3. A Multiplier FU, **MulFU,** that computes the products of two registers for the MUL instruction. The product is assumed to fit into a single register, as in Project 1. **The MulFU is *not pipelined* and has a 3-cycle latency.**
4. A memory access unit, **MAU**, that implements the memory operation following program order using the LSQ and ROB. **The MAU has two pipeline stages, each with a one cycle latency and integrates the D-Cache**. Loads take the full 2 cycles and the two cycles include the time it takes to write the data retrieved from memory into the destination physical register. Stores also use both stages of the MAU pipeline to update memory.
5. An integer FU, **IntFU**, that implements all other operations. **IntFU has a single cycle latency**.

- **A sufficient number of forwarding buses exist in the system to ANY avoid contention**. The forwarding bus has a latency of one cycle for tag or data.
- **Early tag broadcasting** is used, allowing for back-to-back execution of a producer and consumer instruction pair under suitable conditions. The tag broadcasts take place on the tag portion of a forwarding bus one cycle *before* the actual result is broadcasted on the data portion of the bus. **Instructions that are being issued and require this forwarded data, pick the data up as they are moving into their designated function unit.** The forwarding bus is used to deliver the results of register-to-register instructions or loads (via the DCache), all destined to a physical register and/or the physical register implementing an instance of CC. ***The forwarding buses are also used to forward the target address of a BZ, BNZ, BP, BNP, JALR or JUMP to the fetch stage*** (see datapath diagram).
- **Enough connections exist to permit one issue per cycle to each of the function units.**
- The sizes of various structures (as the maximum number of respective entries entries) are: **IQ – 24** entries; **BQ – 16** entries; **LSQ – 16** entries; **ROB – 32** entries and **BTB – 8 entries**. The **BIS supports 4 entries**, as speculation depth is limited to 4.

Other relevant details required for the simulation are as described in the following.

Priority for Using a Common FU

When two or more instructions that require the same function unit become ready to issue in the same cycle, the earliest dispatched instruction has the (highest) priority for using the FU. To implement this, have an elapsed cycle counter that is incremented at the end of a cycle. When an instruction is dispatched, the current value of this counter is added to the IQ entry for the instruction (as a field called "dispatch time") to indicate when it was dispatched. So, to identify the earliest dispatched instruction, you have to look at the "dispatch time" field of the IQ entries.

BTB, BQ, Branch Considerations and Related Information

On a BTB miss, a BTB entry is established from the Decode/Dispatch 1 stage (after decoding the instruction). A BTB hit implies that ALL fields in the BTB entry are valid.

Because of the OoO pipeline, several things are worth noting:

- As soon as a target address is calculated, the default pred of a branch can be used, as the flag values needed may not be available when the target address is calculated. This is very different from an in-order pipeline.
- BTB updates require a lookup from the AFU (for inserting the target address into a newly-established BTB entry) and from the BFU (for updating branch behavior). A miss can occur when either of these updates is attempted, as by the time a conditional branch's target address is calculated or when it is resolved, its BTB entry may have been replaced. On such a miss, there is no need to re-establish an entry and the update is dropped.

The BQ entry for conditional branches includes: (a) a valid bit; (b) A field for the instruction type; (c) a field for the prediction based on the default history when a BTB miss was encountered or the prediction derived from the BTB history if a BTB hit occurred; (d) a field for the target address computed by the AFU (on a BTB miss) or the target address obtained from the BTB (on a BTB hit); (e) a field for holding the tag and contents for the register providing the condition codes; (f) a field for the elapsed clock at the time of dispatching the conditional branch (used for breaking ties on contention for issuing the branch to the BFU) and any other field(s) that you may need.

The format of entries for JUMP and JALR are similar and JALR requires separate fields for the destination physical register address, the saved return address. You can come up with a common struct for a BQ entry that accommodates all instructions or have separate structs for each type of control flow instruction and have the BQ entry point to the struct.

You will need to implement a branch target stack (**BIS**), as described in the notes, for handling misspeculation.

BQ and BFU Timing

- The **BQ requires one cycle to select and issue an instruction** to the BFU. Early tag broadcasting is used to hide this delay.
- As stated earlier, **the BFU takes one cycle to resolve a branch resolution, discover and correct for a misprediction, flush instructions along the misspeculated path, state restoration (free list and rename table) and updating the PC in the Fetch stage**.

Hardware Initializations, the Two Free Lists of Physical Registers

The same memory model that was used for Projects 1 and 2 are used for this project. The initializations done in the hardware, that your code should mimic, are as follows:

a. A FIFO list-based register allocation is used. For the physical registers used to hold a computed result or an updated register pointer value or data retrieved from memory, the free list of physical registers contains

the addresses of physical registers P0, P1, P2, .., P24 initially, with P0 at the head of the free list. Allocations are made from the head of the list, starting with P0. A free register is added back to the end of this list when a renamer instruction frees it up, as described in the notes. The list of free physical registers for the flags is, similarly a FIFO, initialized to CP0, CP1,.., CP15, with the CP0 at the head of this list. Allocations are made from the head, starting with CP0; deallocated registers are added back to the tail.

b. At startup, all physical registers are considered invalid.

c. At startup, the BTB, IQ, BQ, LSQ and ROB empty. The head and tail pointers in the LSQ and ROB both point to the entry at index 0.

d. Allocations from the IQ and BQ can use ANY free entry in these queues; no specific allocation order is dictated.

LSQ Timing

If the value of the register to be stored is not available at the time of dispatch, that fact is noted and the value is delivered to the LSQ entry via forwarding (as shown in the diagram of the datapath). If the value is available at the time of dispatching the STORE, the value is read out from the PRF and inserted into the LSQ entry that is being set up for the STORE – this happens in the Rename2/Dispatch stage. Values of the register to be stored can be forwarded using tag matching to a LSQ entry for the stores. Memory operations start on the MAU under conditions as described in the notes. MAU timings are as noted earlier. If the entry at the head of the LSQ has a corresponding entry at the head of the ROB, it can start on the MAU only after all information it needs are valid. If any such information was delivered to the entry from the AFU (the computed memory address) or via forwarding (for stores), the MAU operation can start in the immediately following cycle after delivery at the earliest, assuming that all other conditions are valid.

Other Timing Details

- Forwarding takes one full clock cycle for the tag broadcast and the following cycle for the data broadcast/address delivery. As the tag is broadcasted, the IQ logic selects instructions and readies them for dispatch.
- Instruction issue (and the reading of valid physical registers) takes place at the start of the cycle following the selection and execution starts immediately. Any other register operand, who value is needed via forwarding, is picked up as the instruction moves into its specific FU to begin execution.
- The BTB lookup is assumed to be instantaneous.
- The Decode/Rename 1 and the Rename2/Dispatch stages are assumed to have a latency of one cycle each. The Fetch stage also has a latency of one cycle.
- Instructions spend ***at least*** one cycle in the IQ or BQ or LSQ.
- **Instruction commitment is assumed to be instantaneous**, that is at the end of the cycle when the entry at the head of the ROB becomes eligible for commitment, the architectural register file is updated.

OTHER REQUIREMENTS

For this project, two separate requirements exist for enabling the automated runs for grading and for displaying simulation states during demos (and to assist debugging).

For the automated runs, use the format:

<executable file name> <argument file name> <optionally, number of cycles to simulate>

For displaying the simulation state and to assist debugging, when you just run your executable, a command prompt is displayed and you have various commands that can be typed in, such as **initialize**, **run** for a specified number of

cycles, an **exit** command, a command to **simulate for one cycle** (on a return), **display** command with options for displaying:

- The contents of the entries at the head and tail of the LSQ, ROB, contents of the IQ and BQ.
- Displaying what in one the forwarding buses. Remember that we have as many buses as are needed to avoid contention delays, so just display a list of what is being forwarded, written etc. in the last cycle that was simulated.
- Contents of the BTB.
- The rename table.
- Display contents of the free lists for physical registers.
- Contents of physical registers and architectural registers.
- The last set of memory locations that were updated (most recent 5 locations and cycles when the updates were completed.
- Fetch PC value, last committed PC value.
- Elapsed cycle counter.
- Anything else that assists debugging.

TEST CASES

Please design some test cases yourself; this is part of the software design phase in ANY real world scenario. Here are some suggestions: Start with simple ones without any conditional branches, loads or stores to see if renaming and the IQ, ROB is working properly. Single step to verify correctly implemented activities. Then begin by having a single loop to check the BTB, the BQ implementation, the prediction and any misspeculation handling. Following this add loads and stores and check your implementation. You can then add multiple branches, JUMP and JALR to check out the entire code.

We will supply additional test cases later.

ADDITIONAL REQUIREMENTS

This is a team effort and to track what each member contributed towards this project. Each team will need to submit a one to 3-page long documentation along with your code. This documentation should also describe what each team member did. **THIS IS A REQUIREMENT** that has some points allocated to it. Please make sure that all team members contribute equally.