

Portland State University
ECE 587/687

Precise Interrupts and Issue Logic

Precise Interrupts

- Process state consists of program counter, registers and memory
- An interrupt or exception is precise if the saved process state is consistent with the sequential architectural model
 - ◆ All instructions preceding interrupted instruction have been executed and modified state correctly
 - ◆ All instructions following interrupted instruction are unexecuted and haven't modified state
 - ◆ If interrupt is caused by exception due to an instruction in the program, the saved PC points to the interrupted instruction
- Providing precise state can be difficult on pipelined processors that allow out of order execution

Why is Precise State Needed?

- With speculative execution, precise state must be maintained to recover from mispredictions and exceptions
 - ◆ Exceptions, e.g., page faults, occur without warning, but infrequently
 - ◆ Branch mispredictions occur in predictable locations, but happen frequently
- Precise state is necessary or desirable
 - ◆ I/O and timer interrupts: makes restarting possible
 - ◆ Allows restart after page fault (virtual memory systems)
 - ◆ Software debugging: isolate instruction causing bug
 - ◆ Graceful recovery from arithmetic exceptions by software
 - ◆ Implementing unimplemented op-codes in software
 - ◆ Precise interrupts from privileged instructions are necessary to implement virtual machines

Solution: In-Order Execution

- We could avoid the problem completely by disallowing out of order completion at the expense of performance
- Result shift register (RSR) can be used for instructions to reserve write back cycles to the register file
- Instructions that conflict with earlier instructions marked in the RSR stall at the issue stage
- Example code in paper
- RSR Figure 2

Buffering Methods

- The general approach:
 - ◆ Maintain copies of the speculative state and the precise state
 - ◆ Implementing this buffering efficiently is key
- Register buffering methods
 - ◆ Checkpoint repair
 - ◆ Reorder buffer
 - ◆ History Buffer
 - ◆ Future file

Checkpoint Repair

- Multiple copies of the register file provide multiple logical spaces, organized as a stack
- A single logical space is active at any given time
- Periodically, the active logical space's architecture state is pushed onto the stack as a backup or checkpoint

Checkpoint Repair (Cont.)

- An exception or mispredicted branch causes the following:
 - ◆ The backup copy is made safe by allowing all instructions before the branch or exception to complete
 - ◆ The state is recovered from the backup copy
 - ◆ The backup copy used depends on the location of the exception or branch and on the time the backups were taken
- Disadvantage
 - ◆ requires a lot of storage and overhead to make and store backups
 - ◆ This is especially bad for branch recovery, since a backup at every branch is needed

Reorder Buffer (ROB)

- The reorder buffer FIFO is used to hold the speculative state while the register file holds the in-order state
 - ◆ The architectural state is obtained by taking the most recent entry for a register from either the register file or the reorder buffer
- When an instruction is decoded, an entry is allocated on the top of the reorder buffer to hold the result of the instruction
- Example in Fig. 3

Reorder Buffer Operation

- When a value reaches the head of the ROB
 - ◆ If the associated instruction is not complete, the slot remains there until it completes
 - Instructions can continue to be decoded until the reorder buffer is full
 - ROB size can limit performance
 - ◆ If there is a fault associated with the value, the ROB is discarded, and the in-order state of the register file is used
 - ◆ If there is no fault, the value is written to the register file and the entry removed from the ROB

Reorder Buffer Discussion

- The in-order state is always available in the register file, thus it is restored immediately
- Some instructions (e.g., branches and stores) do not produce a register result, so a ROB entry does not need to be allocated
 - ◆ However, not allocating a ROB entry makes it difficult to recover from exceptions
 - For branch misprediction, the ROB needs to know of the position of the instructions preceding the branch
 - For exceptions, it is possible that the faulting instruction is not marked in the reorder buffer

Reorder Buffer Discussion (Cont.)

- A simple solution is to allocate an entry in the reorder buffer for all instructions
 - ◆ Branches and faulting instructions are always marked
 - ◆ ROB becomes the single site that completely records the program order of all instructions
 - Remember that register writes and memory accesses are required to complete in program order

Reorder Buffer Discussion (Cont.)

- We may need to add a few bits to each ROB entry
 - ◆ We must mark instructions that cause faults
 - ◆ We can use the information to find the program counter for a faulting instruction
- ROB Disadvantage: Limits instruction-level parallelism unless bypassing is implemented
 - ◆ Paper Fig. 4
 - ◆ Discuss overhead

History Buffer

- The register file stores the architectural state, while a history buffer holds old values of registers (to be restored if needed)
- When an instruction is decoded:
 - ◆ The value in its destination register is pushed into the history buffer
 - ◆ The oldest value in the history buffer is discarded
 - If the oldest value's corresponding instruction has not completed, the decode process stalls until it does
- Paper Fig. 5

History Buffer (Cont.)

- When an exception or misprediction occurs:
 - ◆ Decoding is suspended
 - ◆ All pending instructions are allowed to complete
 - ◆ Register values are popped off the history buffer stack until the in-order state prior to the problem instruction has been restored
- Disadvantage
 - ◆ Many cycles are required to restore in-order state in the event of a fault
 - ◆ This is especially bad for branch mispredictions since they occur frequently

Future File

- The future file acts like a normal register file
 - ◆ The regular “architectural” register file holds in-order state
 - ◆ The future file holds the latest register state
 - During decode, all operands are read from either the register file or the future file, whichever is current
 - The value read from the future file could be a tag if the instruction producing the value has not yet completed
 - ◆ The reorder buffer manages lookahead state for eventual retirement to the register file
- Paper Fig. 6

Future File (Cont.)

- When non-faulting instructions reach the head of the ROB, their results are written to the register file
- When a faulting instruction reaches the head of the ROB, the future file and the ROB are cleared
- When an instruction completes:
 - ◆ It writes its result to the ROB
 - ◆ If the instruction is the last decoded instruction to write the register, it writes its result to the future file
- Advantages:
 - ◆ Reading operands from the ROB is not required
 - ◆ Recovering to precise state is fast (unlike history buffer)

Instruction Issue Logic (Sohi & Vajapeyam, 1987)

- After instructions are fetched, decoded and renamed, they are placed in instruction buffers where they wait until issue
- An instruction can be issued when its input operands are ready, and there is a functional unit available

Issue Logic Operation

- All out-of-order issue methods must handle the same basic steps
 - ◆ Identify all instructions that are ready to issue
 - ◆ Select among ready instructions to issue as many as possible
 - ◆ Issue the selected instructions, e.g., pass operands and other information to the functional units
 - ◆ Reclaim instruction window storage used by the now issued instructions

Methods of Organizing Instruction Issue Buffers

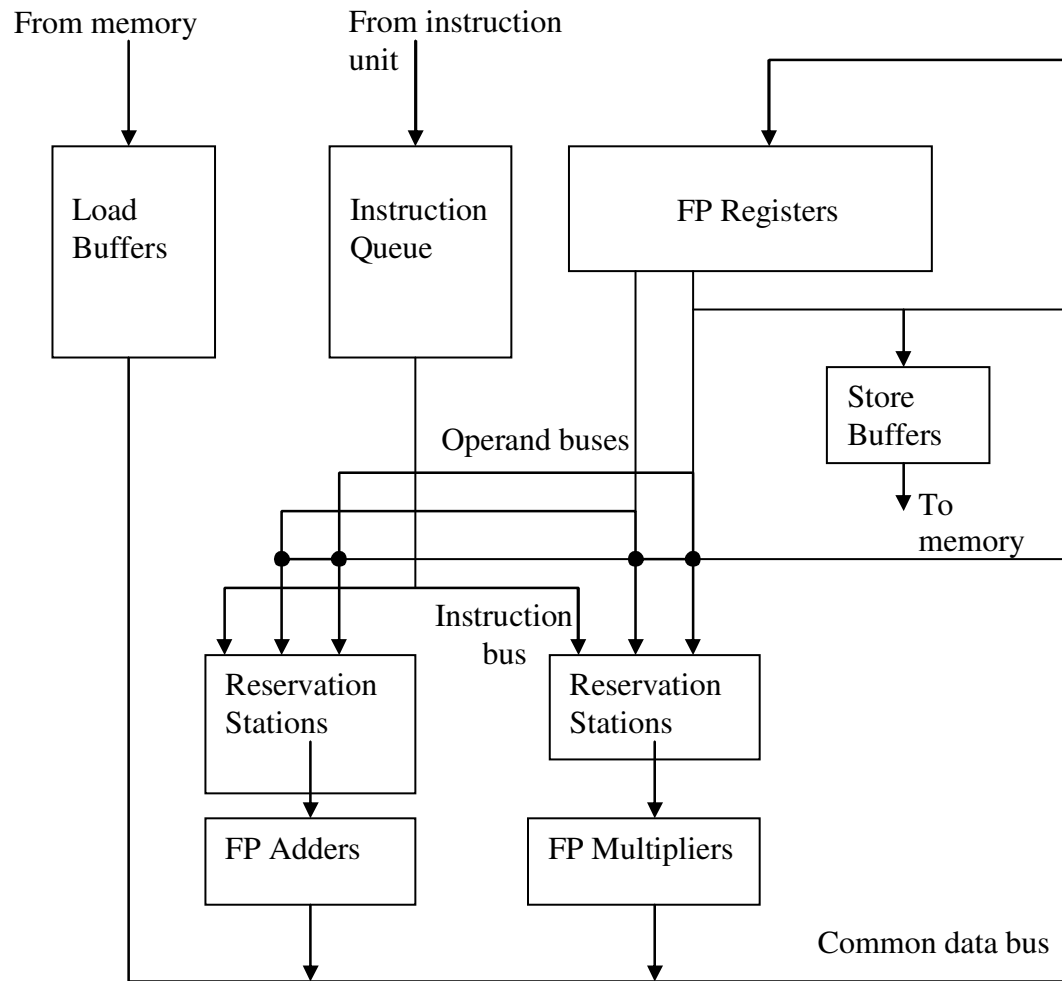
- Single shared queue
 - ◆ Only for in-order issue
- Multiple queues, one per instruction type
- Multiple reservation stations, one per instruction type
- Single central reservation stations buffer
- Reference: Smith & Sohi, Proc. IEEE 1995, “The Microarchitecture of Superscalar Processors”

Tomasulo's Algorithm

- Based on a technique used in the IBM 360/91 floating point execution unit
- Dispatch:
 - ◆ an instruction is read from the instruction queue and sent into an empty reservation station
 - ◆ The operands or tags (if the operands are not available) are read from the register file and written into the operand fields of the reservation station
 - Tags identify functional unit that will write a register

Tomasulo's Algorithm (Cont.)

- Execute:
 - ◆ If both operands are available, execute the instruction
 - ◆ If one or more of the operands is not available, monitor the result writeback bus until the operand is computed and broadcast
- Result writeback:
 - ◆ When a result is computed, broadcast it with its tag on the writeback bus
 - ◆ If the tag in its destination register matches, it is written into the register file, and all reservation stations that need it also grab it



Structure of FP unit using Tomasulo's Algorithm

Central Window Example

- The Register Update Unit (RUU) combines the reservation stations and reorder buffer units
- Allocation and removal is in order
- Instructions may stay in reservation stations longer than necessary
- See Figures 2, 3 and 4 in the Sohi & Vajapeyam paper

Central Window (Cont.)

- Note the continuing importance of the reorder buffer's ability to maintain program order
 - ◆ Exception recovery
 - ◆ Branch misprediction recovery
 - ◆ Memory accesses
 - ◆ Central window simplification

Reading Assignment

- G.Z. Chrysos and J. S. Emer, “Memory Dependence Prediction Using Store Sets”, ISCA 1998 (Review)
- HW2 due Wednesday before class
 - ◆ In paper form (preferred)
 - ◆ If you can't make it to Wednesday's class, send me a pdf file by email