Review of Last 3 Lectures
Phases of Instruction Execution

**Fetch:** Instruction bits retrieved from cache.

**Decode:** Instructions decoded, registers renamed, placed in appropriate issue buffer.

**Execute:** Instructions and operands sent to execution units. When execution completes, all results and exception flags are available.

**Commit:** Instruction irrevocably updates architectural state.

---

In-Order Pipeline

Instructions pass through issue stage and enter execution in-order. May complete out-of-order, but must commit in-order.
Exception Handling
(In-Order Five-Stage Pipeline)

- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

In-Order Superscalar Pipeline

- Fetch two instructions per cycle; issue both simultaneously if one is integer/memory and other is floating point
- Inexpensive way of increasing throughput, examples include Alpha 21064 (1992) & MIPS R5000 series (1996)
- Same idea can be extended to wider issue by duplicating functional units (e.g. 4-issue UltraSPARC) but regfile ports and bypassing costs grow quickly
Out-of-Order Issue

- Issue stage buffer holds multiple instructions waiting to issue.
- Decode adds next instruction to buffer if there is space and the instruction does not cause a WAR or WAW hazard.
  - Note: WAR possible again because issue is out-of-order (WAR not possible with in-order issue and latching of input operands at functional unit)
- Any instruction in buffer whose RAW hazards are satisfied can be issued (for now at most one dispatch per cycle). On a write back (WB), new instructions may get enabled.

Register Renaming

- Decode does register renaming and adds instructions to the issue stage reorder buffer (ROB)
  ⇒ renaming makes WAR or WAW hazards impossible
- Any instruction in ROB whose RAW hazards have been satisfied can be dispatched.
  ⇒ Out-of-order or dataflow execution
Out-of-Order Execution Pipeline

- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- Commit (write-back to architectural state, i.e., regfile & memory), is in-order

Temporary storage needed in ROB to hold results before commit

“Data in ROB” Design
(HP PA8000, Pentium Pro, Core2Duo)

- On dispatch into ROB, ready sources can be in regfile or in ROB dest (copied into src1/src2 if ready before dispatch)
- On completion, write to dest field and broadcast to src fields.
- On issue, read from ROB src fields
Unified Physical Register File  
(MIPS R10K, Alpha 21264, Pentium 4)

• One regfile for both committed and speculative values (no data in ROB)
• During decode, instruction result allocated new physical register, source regs translated to physical regs through rename table
• Instruction reads data from regfile at start of execute (not in decode)
• Write-back updates reg. busy bits on instructions in ROB (assoc. search)
• Snapshots of rename table taken at every branch to recover mispredicts
• On exception, renaming undone in reverse order of issue (MIPS R10000)

Pipeline Design with Physical Regfile
**CS152 Administrivia**

- Quiz 4, Tuesday April 7, Complex Pipelining

- Quiz 5 and 6 moved back one class:
  - Quiz 5, Thursday April 23
  - Quiz 6, Thursday May 7

- Also, PS/Lab 5/6 moved back one class

---

**Memory Dependencies**

\[\text{st } r1, (r2)\]
\[\text{ld } r3, (r4)\]

When can we execute the load?
In-Order Memory Queue

• Execute all loads and stores in program order

=> Load and store cannot leave ROB for execution until all previous loads and stores have completed execution

• Can still execute loads and stores speculatively, and out-of-order with respect to other instructions

• Need a structure to handle memory ordering…

Conservative O-o-O Load Execution

\[
\begin{align*}
st & \ r1, \ (r2) \\
ld & \ r3, \ (r4)
\end{align*}
\]

• Split execution of store instruction into two phases: address calculation and data write

• Can execute load before store, if addresses known and r4 \neq r2

• Each load address compared with addresses of all previous uncommitted stores (can use partial conservative check i.e., bottom 12 bits of address)

• Don’t execute load if any previous store address not known

\[(MIPS \ R10K, \ 16 \ entry \ address \ queue)\]
Address Speculation

\[
\begin{align*}
\text{st} & \ r1, (r2) \\
\text{ld} & \ r3, (r4)
\end{align*}
\]

- Guess that \( r4 \neq r2 \)
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find \( r4 = r2 \), squash load and all following instructions

=> Large penalty for inaccurate address speculation

---

Memory Dependence Prediction

*(Alpha 21264)*

\[
\begin{align*}
\text{st} & \ r1, (r2) \\
\text{ld} & \ r3, (r4)
\end{align*}
\]

- Guess that \( r4 \neq r2 \) and execute load before store
- If later find \( r4 = r2 \), squash load and all following instructions, but mark load instruction as *store-wait*
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear *store-wait* bits
Speculative Loads / Stores

Just like register updates, stores should not modify the memory until after the instruction is committed.

- A speculative store buffer is a structure introduced to hold speculative store data.

Speculative Store Buffer

- On store execute:
  - mark entry valid and speculative, and save data and tag of instruction.
- On store commit:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit
Speculative Store Buffer

- If data in both store buffer and cache, which should we use?
  Speculative store buffer
- If same address in store buffer twice, which should we use?
  Youngest store older than load

Datapath: Branch Prediction and Speculative Execution
Instruction Flow in Unified Physical Register File Pipeline

- **Fetch**
  - Get instruction bits from current guess at PC, place in fetch buffer
  - Update PC using sequential address or branch predictor (BTB)
- **Decode**
  - Take instruction from fetch buffer
  - Allocate resources to execute instruction:
    » Destination physical register, if instruction writes a register
    » Entry in reorder buffer to provide in-order commit
    » Entry in issue window to wait for execution
    » Entry in memory buffer, if load or store
  - Decode will stall if resources not available
  - Rename source and destination registers
  - Check source registers for readiness
  - Insert instruction into issue window+reorder buffer+memory buffer

Memory Instructions

- Split store instruction into two pieces during decode:
  - Address calculation, store-address
  - Data movement, store-data
- Allocate space in program order in memory buffers during decode
- **Store instructions**:
  - Store-address calculates address and places in store buffer
  - Store-data copies store value into store buffer
  - Store-address and store-data execute independently out of issue window
  - Stores only commit to data cache at commit point
- **Load instructions**:
  - Load address calculation executes from window
  - Load with completed effective address searches memory buffer
  - Load instruction may have to wait in memory buffer for earlier store ops to resolve
Issue Stage

• Writebacks from completion phase “wakeup” some instructions by causing their source operands to become ready in issue window
  – In more speculative machines, might wake up waiting loads in memory buffer

• Need to “select” some instructions for issue
  – Arbiter picks a subset of ready instructions for execution
  – Example policies: random, lower-first, oldest-first, critical-first

• Instructions read out from issue window and sent to execution

Execute Stage

• Read operands from physical register file and/or bypass network from other functional units
• Execute on functional unit
• Write result value to physical register file (or store buffer if store)
• Produce exception status, write to reorder buffer
• Free slot in instruction window
Commit Stage

• Read completed instructions in-order from reorder buffer
  – (may need to wait for next oldest instruction to complete)

• If exception raised
  – flush pipeline, jump to exception handler

• Otherwise, release resources:
  – Free physical register used by last writer to same architectural register
  – Free reorder buffer slot
  – Free memory reorder buffer slot

Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)

• MIT material derived from course 6.823
• UCB material derived from course CS252