Superscalar* Club Meeting #4

*we really mean: superscalar speculative out-of-order

James C. Hoe
Department of ECE
Carnegie Mellon University
Goal set when we started

• Achieve an RTL-precise understanding of superscalar speculative out-of-order register dataflow—as with 5-stage pipeline in 18-447
  – know precise you could make a working design and know what it does (how good is it?)
  – know shortcomings and limits in the simplifications you chose to make
  – know what you have tried to but not figured out
  – have a gut-feel for the boundary between known-unknown and unknown-unknown
Our Plan of Attack

• Focus
  – mainly on register dataflow
  – lightly on memory dataflow
  – not at all on i-fetch (a well decoupled subject both conceptually and physically)

• Path
  – further develop concepts in L19 we didn’t have time for
  – study Metaflow DRIS to flesh out conceptual-level understanding
  – study how things were really done in R10K
  – play with an RTL-precise executable model (in C++)
Memory Dataflow
Scheduling Memory Operations

• Memory data dependence (RAW, WAR, WAW)

  Address calculation use registers as input ⇒ OOO

• Storing has side-effect that cannot be undone ⇒ wait until commit

• When to start LW (on uniprocessor)?
  – no more older pending SW OR
  – no older SW with conflicting address (requires knowing all older SW addresses) OR
  – just go if no known conflict; reload if new RAW hazard later

What about MP memory consistency?
IBM 360/91 FP Module [1967]

inorder?  

out-of-order issue window

[Internal Use Draft] Superscalar Club Meeting #4, Slide 6, James C. Hoe, CMU/ECE/CALCM, ©2021 [Do not redistribute.]
# Tomasulo’s Algorithm

<table>
<thead>
<tr>
<th>Instruction state</th>
<th>Wait until</th>
<th>Action or bookkeeping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
<td></td>
<td>if (RegisterStat[rs].Qi ≠ 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{RS[r].Qi ← RegisterStat[rs].Qi}</td>
</tr>
<tr>
<td>FP operation</td>
<td></td>
<td>else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (RegisterStat[rt].Qi ≠ 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{RS[r].Qk ← RegisterStat[rt].Qi}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS[r].Busy ← yes; RegisterStat[rd].Q ← r;</td>
</tr>
<tr>
<td>Load or store</td>
<td></td>
<td>if (RegisterStat[rs].Qi ≠ 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{RS[r].Qi ← RegisterStat[rs].Qi}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS[r].A ← imm; RS[r].Busy ← yes;</td>
</tr>
<tr>
<td>Load only</td>
<td></td>
<td>RegisterStat[rt].Qi ← r;</td>
</tr>
<tr>
<td>Store only</td>
<td></td>
<td>if (RegisterStat[rt].Qi ≠ 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{RS[r].Qk ← RegisterStat[rs].Qi}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute result: operands are in Vj and Vk</td>
</tr>
<tr>
<td>Execute</td>
<td>(RS[r].Qj = 0) and (RS[r].Qk = 0)</td>
<td>RS[r].A ← RS[r].Vj + RS[r].A;</td>
</tr>
<tr>
<td>FP operation</td>
<td></td>
<td>Write Result</td>
</tr>
<tr>
<td>Load-store step 1</td>
<td></td>
<td>Execution complete at r &amp; CDB available</td>
</tr>
<tr>
<td></td>
<td></td>
<td>∀x(if (RegisterStat[x].Qi = r) {Regs[x] ← result; RegisterStat[x].Qi ← 0});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>∀x(if (RS[x].Qj = r) {RS[x].Vj ← result; RS[x].Qj ← 0});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>∀x(if (RS[x].Qk = r) {RS[x].Vk ← result; RS[x].Qk ← 0});</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS[r].Busy ← no;</td>
</tr>
<tr>
<td>Load step 2</td>
<td></td>
<td>Read from Mem[RS[r].A]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load step 1 complete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Case (RegisterStat)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: RF val current</td>
</tr>
<tr>
<td>Store</td>
<td></td>
<td>RS-tag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to be produce by corresponding instruction</td>
</tr>
</tbody>
</table>

- **RS entry**
  - **Busy**: in use
  - **Op**: operand
  - **Vj**: op1 value
  - **Vk**: op2 value
  - **Qj**: op1 RS-tag
  - **Qk**: op2 RS-tag

Reference: [Hennessy & Patterson, CAAQA]
Tomasulo + Speculative Execution
### Tomasulo’s Algorithm + ROB

<table>
<thead>
<tr>
<th>Status</th>
<th>Wait until</th>
<th>Action or bookkeeping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue all instructions</td>
<td></td>
<td>If (RegisterStat[rs].Busy) /* in-flight instr. writes rs */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ h ← RegisterStat[rs].Reorder;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If (ROB[h].Ready) /* Instr completed already */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Else {RS[r].Qj ← h;} /* wait for instruction */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>} Else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS[r].Busy ← yes; RS[r].Dest ← b;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROB[b].Instruction ← opcode; ROB[h].Dest ← rd; ROB[b].Ready ← no;</td>
</tr>
<tr>
<td>Reservation station (r) and ROB (b) both available</td>
<td>If (RegisterStat[rt].Busy) /* in-flight instr writes rt */</td>
<td>{ h ← RegisterStat[rt].Reorder;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If (ROB[h].Ready) /* Instr completed already */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Else {RS[r].Qk ← h;} /* wait for instruction */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>} Else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;}</td>
</tr>
<tr>
<td>FP operations and stores</td>
<td></td>
<td>RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROB[b].Dest ← rd;</td>
</tr>
<tr>
<td>FP operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loads</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS[r].A ← imm; RegisterStat[rt].Reorder ← b;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;</td>
</tr>
<tr>
<td>Stores</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS[r].A ← imm;</td>
</tr>
</tbody>
</table>

**ROB Entry:**

- **is Busy / Instruction / logical Dest reg / dest Value / value is Ready**
- **RegisterStat: reg is Busy / renamed to Reorder buffer entry #**

[Internal Use Draft] Superscalar Club Meeting #4, Slide 9, James C. Hoe, CMU/ECE/CALCM, ©2021 [Do not redistribute.]
Tomasulo’s Algorithm + ROB

Execute

\[(RS[r].Qj == 0) \text{ and } (RS[r].Qk == 0)\]

Compute results—operands are in \(V_j\) and \(V_k\)

Load step 1

\[(RS[r].Qj == 0) \text{ and there are no stores earlier in the queue}\]

\[RS[r].A \leftarrow RS[r].V_j + RS[r].A;\]

Load step 2

Load step 1 done and all stores earlier in ROB have different address

Read from Mem[RS[r].A]

Store

\[(RS[r].Qj == 0) \text{ and store at queue head}\]

\[ROB[h].Address \leftarrow RS[r].V_j + RS[r].A;\]

Write result

Execution done at \(r\) and CDB available

\[b \leftarrow RS[r].Dest; RS[r].Busy \leftarrow \text{no};\]
\[
\forall x(\text{if } (RS[x].Qj=b) \{RS[x].V_j \leftarrow \text{result}; RS[x].Qj \leftarrow 0\})\]
\[
\forall x(\text{if } (RS[x].Qk=b) \{RS[x].V_k \leftarrow \text{result}; RS[x].Qk \leftarrow 0\})\]
\[ROB[b].Value \leftarrow \text{result}; ROB[b].Ready \leftarrow \text{yes};\]

Store

Execution done at \(r\) and \(RS[r].Qk == 0\)

\[ROB[h].Value \leftarrow RS[r].V_k;\]

Commit

Instruction is at the head of the ROB (entry \(h\) and \(ROB[h].ready == yes\)

\[d \leftarrow ROB[h].Dest; /* register dest, if exists */\]
\[
\text{if } (ROB[h].Instruction==\text{Branch}) \text{ then}\]
\[
\{\text{if branch is mispredicted} \}
\[
\{\text{clear ROB[h], RegisterStat; fetch branch dest;};\}\}
\[
\text{else if } (ROB[h].Instruction==\text{Store}) \}
\[
\{\text{Mem[ROB[h].Destination] \leftarrow ROB[h].Value;\}
\[
\text{else } /* put the result in the register destination */\]
\[
\{\text{Regs[d] \leftarrow ROB[h].Value;\}
\[
\text{ROB[h].Busy \leftarrow \text{no}; */ free up ROB entry */\}
\[
/* free up dest register if no one else writing it */\]
\[
\text{if } (\text{RegisterStat[d].Reorder==\text{h}}) \{\text{RegisterStat[d].Busy \leftarrow \text{no};\}
\]

[Internal Use Draft] Superscalar Club Meeting #4, Slide 10, James C. Hoe, CMU/ECE/CALCM, ©2021 [Do not redistribute.]
What is actually hard

- Memory addresses are much wider than register names
- Memory dependencies are not static
  LW and SW addresses need to be calculated and translated first
- Memory instructions take longer to execute relative to other instructions types
- Special ordering rules for correctness
- More special ordering rules if shared memory MP
DRIS is also Address Queue

- LW/SW issued 1st-time as “ADD”
  - compute base+offset
  - result written to Data; complete in dataflow order
- SW issue 2\textsuperscript{nd}-time on retire
- LW issue 2\textsuperscript{nd}-time when RAW-free
  1. wait until all older SW addresses available
  2. check (by CAM) LW addr against older SW addr’s
  3. If conflict, forward youngest matching SW to LW
- OOO for shared-memory MP must obey additional ordering rules

\textit{Again, read R10K for black-belt magic}
Memory Disambiguation

FIGURE 6.8: Schematics of the MIPS R10000 pipeline to implement the partial ordering memory disambiguation policy. from [Gonzalez, et al., 2010]
Address Queue

- Track LW/SW and addr in program order
- Addr calculation issued in dataflow order
- N-comparators compares new/external addr to each recorded addr to detect
  - LW/SW RAW hazards
  - cache line return after load miss
  - external invalidations (see S17)
- Pair-wise RAW hazards matrix ($N^2$ bits) updated incrementally after each LW/SW addr becomes known
- LW do not issue against known/potential RAW hazard or pending cache miss
Dependence Matrix

indetermination matrix (i.e., \( \neg \text{valid bit for dependence matrix} \))
at the time of the latter of the 2 addresses ready for compare

dependence matrix

from [Gonzalez, et al., 2010]
Weak Consistency (WC)

- WC only impose uniprocessor memory dependence: \( R_i(x) < W_j(x) \); \( W_i(x) < R_j(x) \); \( W_i(x) < W_j(x) \)
- Program insert explicit **memory fence** instructions to force serialization when it matters

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y is initially 0</td>
<td>...... do {</td>
</tr>
<tr>
<td>......</td>
<td>ready = load Y</td>
</tr>
<tr>
<td>compute v</td>
<td>fence</td>
</tr>
<tr>
<td>store ( (X, v) )</td>
<td>} while (!ready)</td>
</tr>
<tr>
<td>fence</td>
<td>fence</td>
</tr>
<tr>
<td>store ( (Y, 1) )</td>
<td>data = load X</td>
</tr>
</tbody>
</table>

- Implementation wise, heavy-weight fence stalls subsequent LW/SW progress until fence retire
R10K Speculative SC

- Stores commit in-order when retired
- Loads attempted speculatively and out-of-order
  - address of invalidated cache blocks checked against speculative loads in load/store queue
  - if a loaded cache-block invalidated before load retires, “soft exception” restart from oldest affected load
  - load retires only if the fetched value is still “current”
    (some very subtle strangeness is possible though)

⇒ Loads and stores appears executed in program order at the commit point
⇒ No penalty for sequential programs!!!