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 $\Rightarrow$ OOO

- Storing has side-effect that cannot be undone $\Rightarrow$
must 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
### Tomasulo’s Algorithm

<table>
<thead>
<tr>
<th>Instruction state</th>
<th>Wait until</th>
<th>Action or bookkeeping</th>
</tr>
</thead>
</table>
| Issue FP operation| Station r empty | if (RegisterStat[rs].Qi≠0)  
{RS[r].Qj ← RegisterStat[rs].Qi}  
else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};  
if (RegisterStat[rt].Qi≠0)  
{RS[r].Qk ← RegisterStat[rt].Qi}  
else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};  
RS[r].Busy ← yes; RegisterStat[rd].Qj ← r; |
| Load or store     | Buffer r empty | if (RegisterStat[rs].Qi≠0)  
{RS[r].Qj ← RegisterStat[rs].Qi}  
else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};  
RS[r].A ← imm; RS[r].Busy ← yes;  
RegisterStat[rt].Qi ← r; |
| Load only         |             | if (RegisterStat[rs].Qi≠0)  
{RS[r].Qk ← RegisterStat[rs].Qi}  
else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; |
| Store only        |             | |
| Execute FP operation| (RS[r].Qi = 0) and (RS[r].Qk = 0) | Compute result: operands are in Vj and Vk |
| Load-store step 1 | RS[r].Qj = 0 & r is head of load-store queue | RS[r].A ← RS[r].Vj + RS[r].A; |
| Load step 2       |             | Read from Mem[RS[r].A]; |
| Write Result FP operation | Execution complete at r & CDB available | ∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result;  
RegisterStat[x].Qi ← 0});  
∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0});  
∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0});  
RS[r].Busy ← no; |
| Store             | Execution complete at r & RS[r].Qk = 0 | Mem[RS[r].A] ← RS[r].Vk;  
RS[r].Busy ← no; |

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

Case (RegisterStat)

- **0**: RF val current
- **RS-tag**: to be produce by corresponding instruction
Tomasulo + Speculative Execution
### Tomasulo’s Algorithm + ROB

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

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, ©2022 [Do not redistribute.]
### Tomasulo’s Algorithm + ROB

**Execute**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS[r].Qj == 0 and RS[r].Qk == 0</td>
<td>Compute results—operands are in Vj and Vk</td>
</tr>
</tbody>
</table>

**Load Step 1**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS[r].Qj == 0 and there are no stores earlier in the queue</td>
<td>RS[r].A ← RS[r].Vj + RS[r].A;</td>
</tr>
</tbody>
</table>

**Load Step 2**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load step 1 done and all stores earlier in ROB have different address</td>
<td>Read from Mem[RS[r].A]</td>
</tr>
</tbody>
</table>

**Store**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS[r].Qj == 0 and store at queue head</td>
<td>ROB[h].Address ← RS[r].Vj + RS[r].A;</td>
</tr>
</tbody>
</table>

**Write Result**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution done at r and CDB available</td>
<td>b ← RS[r].Dest; RS[r].Busy ← no; ∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0}); ∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0}); ROB[b].Value ← result; ROB[b].Ready ← yes;</td>
</tr>
</tbody>
</table>

**Commit**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes</td>
<td>d ← ROB[h].Dest; /* register dest, if exists <em>/ if (ROB[h].Instruction==Branch) {if (branch is mispredicted) {clear ROB[h], RegisterStat; fetch branch dest;};} else if (ROB[h].Instruction==Store) {Mem[ROB[h].Destination] ← ROB[h].Value;} else /</em> put the result in the register destination <em>/ {Regs[d] ← ROB[h].Value;} ROB[h].Busy ← no; /</em> free up ROB entry <em>/ /</em> free up dest register if no one else writing it */ if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;}</td>
</tr>
</tbody>
</table>

*? subtle*
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
- **Want to load as soon as possible!!!**
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 2nd-time on retire
- LW issue 2nd-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

See 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

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

```
T1:
  Y is initially 0
  ......
  compute v
  store (X, v)
  fence
  store (Y, 1)

T2:
  ......
  do {
    ready = load Y
  } while (!ready)
  fence
  data = load X
```

- Implementation wise, heavy-weight fence stalls subsequent LW/SW progress until fence retires
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!!!