Speculative Execution using Reorder Buffer

CprE 581 Computer Systems Architecture
Reading: Textbook 2.6
Control Dependencies

- Every instruction is control dependent on some set of branches
  
  ```
  if p1
    S1;
  if p2
    S2;
  ```

- $S_1$ is control dependent on $p_1$, and $S_2$ is control dependent on $p_2$ but not on $p_1$.

control dependencies must be preserved to preserve program order
Speculation is to run instructions on prediction – predictions could be wrong.

Branch prediction: crucial to performance, could be very accurate.

Mis-prediction is less frequent event – but cannot be ignored.

Example:

```
for (i=0; i<1000; i++)
    C[i] = A[i]+B[i];
```

Branch prediction: predict the execution as accurately as possible (frequent cases).

Speculative execution recovery: if prediction is wrong, roll the execution back.
Exception Behavior

Preserving exception behavior -- exceptions must be raised exactly as in sequential execution

- Same sequences
- No “extra” exceptions

Example:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DADDU</td>
<td>R2, R3, R4</td>
</tr>
<tr>
<td>BEQZ</td>
<td>R2, L1</td>
</tr>
<tr>
<td>LW</td>
<td>R1, 0(R2)</td>
</tr>
</tbody>
</table>

L1:

Problem with moving \texttt{LW} before \texttt{BEQZ}?

Again, a dynamic execution must produce the same register/memory contents as a sequential execution, even if an exception happens.
Precise Interrupts

Tomasulo had:

In-order issue, out-of-order execution, and out-of-order completion

Need to “fix” the out-of-order completion aspect so that we can find precise breakpoint in instruction stream.
Branch Prediction vs. Precise Interrupt

Mis-prediction is “exception” on the branch inst

Execution “branches out” on exceptions
- Every instruction is “predicted” not to take the “branch” to interrupt handler

Same technique for handling both issues: in-order completion or commit: change register/memory only in program order

How does it ensure correctness?
Correctness of Speculative Execution

Hazards that could be raised by speculative execution

1. Would junk instructions write bad data to register or memory?

2. Would junk instructions break the data flows between good instructions?
Correctness of Speculative Execution

1. The same set of instructions write to user-visible register and memory;  
   No junk instruction should write to user-visible register or memory

2. Each instruction receives the same operands as in the sequential execution;  
   No junk instruction should affect the inputs of any good instructions

3. Any register or memory word receives the value of the last write as in the sequential execution
Tomasulo and Speculative Execution

Tomasulo’s Property: Because of renaming, the direct data flow between two instructions cannot be affected by any instructions after them.
Tomasulo and Speculative Execution

Example:

```
ADD.D   F0, F2, F4
DIV.D   F6, F6, F0
SUB.D   F0, F12, F14
DIV.D   F16, F16, F0
```

How does Tomasulo help maintain the data flow between good instructions? Assume SUB.D is executed before the first DIV.D.
Speculative Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   - **Condition:** a free RS at the required FU
   - **Actions:** (1) decode the instruction; (2) allocate a RS and ROB entry; (3) do source register renaming; (4) do dest register renaming; (5) read register file; (6) issue the decoded and renamed instruction to the RS and ROB

2. Execution—operate on operands (EX)
   - **Condition:** At a given FU, At least one instruction is ready
   - **Action:** select a ready instruction and send it to the FU

3. Write result—finish execution (WB)
   - **Condition:** At a given FU, some instruction finishes FU execution
   - **Actions:** (1) FU writes to CDB, broadcast to all RSs and to the ROB; (2) FU broadcast tag (ROB index) to all RS; (3) de-allocate the RS. Note: no register status update at this time
Speculative Tomasulo Algorithm

4. Commit—update register with reorder result

- **Condition:** ROB is not empty and the ROB head inst has finished execution
- **Actions if no mis-prediction/exception:** (1) write result to register/memory, (2) update register status, (3) de-allocate the ROB entry
- **Actions if with mis-prediction/exception:** flush the pipeline, e.g. (1) flush IFQ; (2) clear register status; (3) flush all RS and reset FU; (4) reset ROB
The Hardware: Reorder Buffer

- If inst write results in program order, reg/memory always get the correct values

- Reorder buffer (ROB) - reorder out-of-order inst to program order at the time of writing reg/memory (commit)

- If some inst goes wrong, handle it at the time of commit - just flush inst afterwards

- Inst cannot write reg/memory immediately after execution, so ROB also buffer the results

  *No such place in Tomasulo original*
Reorder Buffer Details

- Holds branch valid and exception bits
  - Flush pipeline when any bit is set
  - How do the architectural states look like after the flushing?

- Holds dest, result and PC
  - Write results to dest at the time of commit
  - Which PC to hold?
  - A ready bit (not shown)

- Supplies operands between execution complete and commit

![Reorder Buffer Diagram]

- Branch or L/W?
- Dest reg
- Result
- Exceptions?
- Program Counter
- Ready?
ROB: Circular Buffer

- Head
- Tail

... allocated

... free

...
<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  
FP operations and stores | if (RegisterStat[rt].Busy)/*in-flight instr writes rt*/  
   h ← RegisterStat[rt].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[rt]; RS[r].Qj ← 0;);  
| FP operations | RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd; |
| Loads        | RS[r].A ← imm; RegisterStat[rt].Reorder ← b;  
   RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt; |
| Stores       | RS[r].A ← imm; |
| Execute FP op | (RS[r].Qj == 0) and (RS[r].Qk == 0) | Compute results—operands are in Vj and Vk |
| Load step 1  | (RS[r].Qj == 0) and (RS[r].Qk == 0) | RS[r].A ← RS[r].Vj + 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) and store at queue head | ROB[h].Address ← RS[r].Vj + RS[r].A; |
| Write result all but store | Execution done at r and CDB available | b ← RS[r].Dest; RS[r].Busy ← no;  
   VX (if (RS[x].Qj==b) RS[x].Vj ← result; RS[x].Qj ← 0);  
   VX (if (RS[x].Qk==b) RS[x].Vj ← result; RS[x].Qk ← 0);  
   ROB[b].Value ← result; ROB[b].Ready ← yes; |
| Store        | Execution done at r and (RS[r].Qk == 0) | ROB[h].Value ← RS[r].Vj; |
| Commit       | Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes | d ← ROB[h].Dest; /* register dest, if exists */  
   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 /* put the result in the register destination */  
   (Reg[d] ← ROB[h].Value;);  
   ROB[h].Busy ← no; /* free up ROB entry */  
   /* free up dest register if no one else writing it */  
   if (RegisterStat[d].Reorder==h) (RegisterStat[d].Busy ← no); |
Tag: ROB Index

Use ROB index as tag

- Why not RS index any more?
- Why is ROB index a valid choice?

- Register result status rename a register index to ROB index if the register is renamed
- Reservation stations now use ROB index for tracking dependence and for wakeup
- Again tag (now ROB index) and data are broadcast on CDB at writeback

- Inst may receive register values from (1) register, (2) CDB broadcasting, or (3) ROB
Data Flow in Tomasulo with ROB

Assume inst i has a source Rx produced by inst j.

Three possibilities at i.rename:

1. **Rx is not renamed**
   \[ \Rightarrow i \text{ receives } j\text{'}s output from the register file \]

2. **Rx is renamed and j.wb has finished**
   \[ \Rightarrow i \text{ receives } j\text{'}s output from ROB \]

3. **Rx is renamed and j.wb is happening at the same cycle**
   \[ \Rightarrow i \text{ receives } j\text{'}s output from CDB broadcasting \]

4. **Rx is renamed, and j.exe has not finished**
   \[ \Rightarrow i \text{ will receive } j\text{'}s output from CDB broadcasting \]

And i will never receive the output of a following instruction.
**Code Example**

; X[i] = X[i] + 1 for 0 <= i < N

Loop:
    LW R2, 0(R1)
    DADDIU R2, R2, #1
    SW R2, 0(R1)
    DADDIU R1, R1, #4
    BNE R2, R3, Loop
    LW R3, 0(R4)
    ...

How would this code be executed? What if the BNE is incorrectly predicted?
## Scheduling Results

<table>
<thead>
<tr>
<th>Inst</th>
<th>Fetch</th>
<th>Disp</th>
<th>Schd</th>
<th>EXE</th>
<th>MEM</th>
<th>CDB</th>
<th>CMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW(1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>DADDIU</td>
<td>2</td>
<td>3</td>
<td>4-6</td>
<td>7</td>
<td>-</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>SW</td>
<td>3</td>
<td>4</td>
<td>5-8</td>
<td>9</td>
<td>10</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>DADDIU</td>
<td>4</td>
<td>5</td>
<td>6-7</td>
<td>8</td>
<td>-</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>BNE</td>
<td>5</td>
<td>6</td>
<td>7-8</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>13</td>
</tr>
<tr>
<td>LW(2)</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW(1)</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>DADDIU</td>
<td>15</td>
<td>16</td>
<td>17-19</td>
<td>20</td>
<td>-</td>
<td>21</td>
<td>22</td>
</tr>
</tbody>
</table>

Assume the first branch is predicted incorrectly.
Tomasulo Summary

- Reservations stations: *implicit register renaming* to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards of Scoreboard
- Not limited to basic blocks when compared to static scheduling (integer units gets ahead, beyond branches)
- Today, helps cache misses as well
  - Don’t stall for L1 Data cache miss (insufficient ILP for L2 miss?)
  - Can support memory-level parallelism
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation (discuss later)
- 360/91 descendants are Pentium III; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264