Register Renaming

- Register renaming is a method to address storage conflicts. Duplication of resources can address it.

- Output dependency and anti-dependency are examples of storage conflicts. So, they can be address using register renaming.

- In essence, registers are allocated dynamically by the processor hardware.
  - When a new register value is created, a new register is allocated for that value.
  - Subsequent instructions that access that value must be revised to refer to the register containing the value. (Renaming)
  - The same original register reference in several different instructions may refer to different actual registers, if different values are intended.

Let's use the above example again:

I0: R3 op R5 -> R3
I1: R3 + 1 -> R4
I2: R5 + 1 -> R3
I3: R3 op R4 -> R7

- After applying register renaming, the revised instructions look like:
  I0: R3 op R5 -> R3(a)
  I1: R3(a) + 1 -> R4(a)
  I2: R5 + 1 -> R3(b)
  I3: R3(b) op R4(a) -> R7(a)

- The register reference with a pair parentheses refers to a hardware register allocated to hold a new value.
- Note: R3(a), R3(b), and R4(a) avoids the WAR and WAW.

Exercise

- Given the instructions below, list all the dependencies it has, and apply register renaming to address output dependency (WAW) and anti-dependency (WAR).

  I0: R3 + 1 -> R3
  I1: R3 + R2 -> R4
  I2: R3 op R4 -> R7
  I3: Store R0 -> R4

- True data dependency:
  - I1 depends on the result of I0 (R3)
  - I2 depends on the result of I0 and I1 (R3, R4)
• WAW: \textbf{I3} writes after \textbf{I1} write to R4
• WAR: \textbf{I3} writes after \textbf{I2} reads R4
• Register renaming:
  \begin{align*}
  \text{I0: } & R3 + 1 \rightarrow R3(a) \\
  \text{I1: } & R3(a) + R2 \rightarrow R4(a) \\
  \text{I2: } & R3(a) \text{ op } R4(a) \rightarrow R7(a) \\
  \text{I3: } & \text{Store } R0 \rightarrow R4(b)
  \end{align*}

\textbf{Instruction Issue Policy}

- In essence, the processor is trying to \textbf{look ahead of current point} of execution to \textbf{locate instructions} that can be brought \textbf{into the pipeline}.

- Three types of ordering are important in this regards:
  • Order in which instructions are \textbf{fetched}
  • Order in which instructions are \textbf{executed} (constrained by data dependencies)
  • Order in which instructions \textbf{update} registers and memory values (order of completion)

- One \textbf{constraint: results must be correct}. So, the processor must \textbf{accommodate} the various \textbf{dependencies and conflicts} discussed earlier.

- Four categories:
  • In-order issue (order to execute), in-order completion (order to write the result)
  • In-order issue, out-of-order completion
  • Out-of-order issue, out-of-order completion
  • Out-of-order issue, in-order completion

- We use this example to illustrate out-of-order issue, out-of-order completion:
  • Assume a superscalar pipeline is capable of fetching and decoding 2 instructions at a time
    • Instructions are fetched and decoded in pair. The next two instructions must wait to be decoded until the pair of decode pipeline stages has cleared.
  • having 3 separate ALUs (e.g., two for integer arithmetic and one for floating-point arithmetic)
  • 2 instances of the write-back pipeline stage
  • 6 instruction code fragment with the following constraints:
    • I1 requires two cycles to execute
    • I3 and I4 conflict for the same functional unit (e.g., both need floating-point arithmetic)
    • I5 depends on the value produced by I4
- I5 and I6 conflict for a functional unit
- To fetch, decode, and write back an instruction, each stage need 1 clock cycle.
- When there is a conflict for a functional unit, or when a functional unit requires more than one cycle to generate a result, instructions temporarily stall.

---

**Out-of-order issue, out-of-order completion**

- With **in-order issue**, the processor will only **decode** instructions **up to the point of true data dependency or resource conflict**.
- The processor **cannot look ahead beyond the point of conflict** to subsequent instructions that may be independent of those already in the pipeline that may be usefully introduced into the pipeline.
- It is necessary to **decouple the decode and execute stages**.
- This is done with a buffer, **instruction window**.
  - After a processor has finished decoding an instruction, the decoded instruction is placed in the instruction window. **As long as this buffer is not full, the processor can continue to fetch and decode new instructions.**
- **When a functional unit becomes available** in the execute stage, an instruction from the instruction window can be issued to the execute stage. Any instruction may be issued, provided that:
  - it needs the particular **functional unit** that is **available**
  - **no conflicts or dependencies** block this instruction

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Window</th>
<th>ALU1</th>
<th>ALU2</th>
<th>ALU3</th>
<th>WriteBack1</th>
<th>WriteBack2</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>I2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>I3</td>
<td>I4</td>
<td>I1</td>
<td>I2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>I5</td>
<td>I6</td>
<td>I3</td>
<td>I4</td>
<td>11, 12</td>
<td>I1</td>
<td>I2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td>I6</td>
<td>I3, 4</td>
<td>I1</td>
<td>I3</td>
<td>I2</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>I4, I5, I6</td>
<td>I6</td>
<td>I4</td>
<td>I1</td>
<td>I3</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>I5</td>
<td>I5</td>
<td>I5</td>
<td>I4</td>
<td>I6</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>I5</td>
<td>I5</td>
<td>I6</td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

- Note: in this example, I3 can use ALU2 or ALU3 at cycle 4. This example lets I3 use ALU3. I6 can use ALU1 or ALU2 at cycle 5. This example lets I6 use ALU2. I5 and I6 have functional unit conflict with I4, it's unnecessary to let them use ALU3 here, as it slows down the execution for one more cycle. This example assumes that the window is large enough to buffer all decoded instructions.