Computer Architecture

R. Poss
Instruction policies & related hazards

Instruction issue vs completion, new data hazards
Instruction issue basics

Just widening of the processor’s pipeline does not necessarily improve its performance.

The processor’s **policy in fetching, decoding and executing instructions** also has a significant effect on its performance.

The instruction issue policy is determined by its **look-ahead capability** in the instruction stream.

For example, with no look-ahead, if a resource conflict halts instruction fetching the processor is not able to find any further instructions until the conflict is resolved.

If the processor is able to continue fetching instructions it may find an independent instruction that can be executed on a free resource out of programmed order.

Policies characterized by **issue order** and **completion order**.
In-order issue, in-order completion

- Simplest, unusual with superscalar designs
- Instructions issued in exact program order with results written in the same order
- This is shown here for comparison purposes only, as very few pipelines use in-order completion
In-order issue, in-order completion

Assume a 3 stage execution in a pipeline that can issue two instructions, execute three instructions and write back two results every cycle... assume:

- I1 requires 2 cycles to execute
- I3 and I4 are in conflict for a functional unit
- I5 depends on the value produced by I4
- I5 and I6 are in conflict for a functional unit

<table>
<thead>
<tr>
<th>Time</th>
<th>Decode</th>
<th>Execute</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I1 I2</td>
<td>I1 I2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I3 I4</td>
<td>I1 I4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I3 I4</td>
<td>I1 I2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I4 I4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I4 I4</td>
<td>I3 I4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I5 I6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I5 I6</td>
<td>I3 I4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I5 I6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I5 I6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6 instructions require 8 cycles

IPC = $\frac{6}{8} = 0.75$
In-order issue, out-of-order completion

Out-of-order completion, improves performance of instructions with long latency operations, such as loads and floating point

The modifications made to execution are:

- any number of instructions allowed in the execution stage up to the total number of pipeline slots (stages × functional units)
- instruction issue is not stalled when an instruction takes more than one cycle to complete
In-order issue, out-of-order completion

Again assume a processor issues two instructions, executes three instructions and writes back two results every cycle

- I1 requires 2 cycles to execute
- I3 and I4 are in conflict for a functional unit
- I5 depends on the value produced by I4
- I5 and I6 are in conflict for a functional unit

<table>
<thead>
<tr>
<th>Time</th>
<th>Decode</th>
<th>Execute</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I1</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I2</td>
<td>I2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I3</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I4</td>
<td>I3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I3</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I4</td>
<td>I4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I5</td>
<td>I5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I6</td>
<td>I6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I6</td>
<td>I6</td>
<td></td>
</tr>
</tbody>
</table>

6 instructions require 7 cycles

IPC = 6/7 = 0.86
In-order issue, out-of-order completion

In a processor with out-of-order completion, instruction issue is stalled when:

- There is a conflict for a functional unit
- An instruction depends on a result that is not yet computed - a data dependency
- can use register specifiers to detect dependencies between instructions and logic to ensure synchronisation between producer and consumer instructions - e.g. scoreboard logic, cf CDC 6600
- Also: a new type of dependency caused by out-of-order completion: the output dependency
Output dependencies

Consider the code to the right:

- the 1st instruction must be completed before the 3rd, otherwise the 4th instruction may receive the wrong result!
- this is a new type of dependency caused by allowing out-of-order completion
- the result of the 3rd instruction has an output dependency on the 1st instruction
- the 3rd instruction must be stalled if its result may be overwritten by a previous instruction which takes longer to complete
Out-of-order issue, out-of-order completion

In-order issue stalls when the decoded instruction has:

- a resource conflict, a true data dependency or an output dependency on an uncompleted instruction
- this is true even if instructions after the stalled one can execute
- to avoid stalling, decode must be decoupled from execution

Conceptually out-of-order issue decouples the decode/issue stage from instruction execution

- it requires an instruction window between the decode and execute stages to buffer decoded or part pre-decoded instructions
- this buffer serves as a pool of instructions giving the processor a look-ahead facility
- instructions are issued from the buffer in any order, provided there are no resource conflicts or dependencies with executing instructions
Out-of-order issue, out-of-order completion

Again assume a processor issues two instructions, executes three instructions and writes back two results every cycle but now has a issue window of at least three instructions.

- I1 requires 2 cycles to execute
- I3 and I4 are in conflict for a functional unit
- I5 depends on the value produced by I4
- I5 and I6 are in conflict for a functional unit

<table>
<thead>
<tr>
<th>Time</th>
<th>Decode</th>
<th>I. Window</th>
<th>Execute</th>
<th>Writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I1, I2</td>
<td>I1, I2, I3, I4, I4, I5, I6</td>
<td>I1, I2, I3, I4, I5, I6</td>
<td>I2</td>
</tr>
<tr>
<td></td>
<td>I3, I4</td>
<td>I3, I4, I5</td>
<td>I6, I4, I5</td>
<td>I1, I3, I4, I5, I6</td>
</tr>
<tr>
<td></td>
<td>I5, I6</td>
<td>I5, I5</td>
<td>I5</td>
<td>I5</td>
</tr>
</tbody>
</table>

6 instructions require 6 cycles
IPC = 1
Anti-dependencies

Out-of-order issue introduces yet another dependency - called an anti-dependency

the 3rd instruction can not be completed until the second instruction has read its operands

otherwise the 3rd instruction may overwrite the operand of the 2nd instruction

we say that the result of the 3rd instruction has an anti-dependency on the 1st operand of the 2nd instruction

this is like a true dependency but reversed
Summary of data hazards

We have now seen three kinds of dependencies

- **True (data) dependencies** ... read after write (RAW)
- **Output dependencies** ... write after write (WAW) - out of order completion
- **Anti dependencies** ... write after read (WAR) - out of order issue

Only true dependencies reflect the flow of data in a program and should require the pipeline to stall

- when instructions are issued and completed out of order, the one-to-one relationship between registers and values at any given time is lost
- new dependencies arise because registers hold different values from independent computations at different times – they are **resource dependencies**

**Resource dependencies are really just storage conflicts** and can be eliminated by introducing new registers to re-establish the one-to-one relationship between registers and values at a given time
Register renaming

How resource dependencies are managed in out-of-order issue or completion
Renaming – example

Renaming dynamically rewrites the machine code using a larger register set

A renamed register is allocated somehow and remains in force until commit

Subsequent use of a register name as an operand uses the latest rename of it

may need to keep several renamed registers if using branch prediction

\[
\begin{align*}
R_3 & \rightarrow R_{3b} \rightarrow R_{3c} \\
R_{3b} & := R_3 \text{ op } R_5 \\
R_4 & := R_{3b} + 1 \\
R_{3c} & := R_5 + 1 \\
R_7 & := R_{3c} \text{ op } R_4 
\end{align*}
\]
Register renaming

Storage conflicts can be removed in out-of-order issue microprocessors by renaming registers

- this requires additional registers e.g. a rename buffer or an extended register file, not visible to the program
- mapping between logical name and physical location is maintained in hardware while the instructions are executing

Instructions are executed out of sequence from the instruction window using the renamed registers

- new physical register allocated on multiple use of same target register name
- mapping from instruction register to architectural register stored in hardware
- instructions executing after a rename use the renamed register rather than instruction-specified register as an operand

A commit stage is used to preserve sequential machine state by storing values to architectural registers in program order
Strategies renaming

Can either rename at **instruction issue**

- explicit renaming maps architectural register to physical register used in conjunction with a **scoreboard** to track dependencies

Can remap implicitly using **reservation stations** at the execute stage and a **reorder buffer** on instruction completion

- reservation stations use dataflow to manage true dependencies and implicitly rename registers

- the reorder buffer holds the data until all previous instructions have completed then write it to the architectural register specified