Out-of-order (OoO) Processing

Suvinay Subramanian

(adapted from prior 6.823 offerings, ack: Nathan Beckmann)
Since Last Time...

1. Complex Pipelines
   - Superscalar execution
   - Out-of-order (OoO) processing
     - Scoreboarding
     - OoO: Issue, completion, retiring
     - Register renaming

2. Branch Prediction
In-Order Pipeline

- Fetch phase
- Decode & Reg-fetch phase
- Execute phase
- Memory phase
- Write-back phase

Diagram: A flowchart illustrating the in-order pipeline stages, including:
- PC
- Memory
- IR
- ALU
- Memory
- Backwards

Key components in each phase:
- Addr
- Rdata
- Inst. Memory
- We
- Rs1
- Rs2
- Rd1
- Ws
- Rd2
- GPRs
- We
- Addr
- Rdata
- Data
- Memory
- Wdata
In-Order Pipeline Limitations

Observation: True data dependency stalls dispatch of younger instructions into functional (execution) units.

```
MUL  R3 <- R1, R2
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
MUL  R5 <- R6, R8
ADD  R7 <- R3, R5

LD  R3 <- R1 (0)
ADD  R3 <- R3, R1
ADD  R1 <- R6, R7
MUL  R5 <- R6, R8
ADD  R7 <- R3, R5
```
Let’s take a step back: What limits performance?

1. Von Neumann Model
   • Sequential stream of instructions

2. Implementation Issues
   • Multi-cycle operations
   • Variable latency operations
Computation Structure

Every algorithm is conceptually a number of tasks with dependencies between them.
Compilation

Compilation serializes this graph in some way

In-order semantics—"false" dependencies

“True” data dependencies
Out-of-order Processing

Essentially, OOO tries to *dynamically* recover the true computation graph.
How to do this correctly?

1. Must recognize dependencies between instructions

2. Must cause correct sequencing of the dependent instructions

3. Allow independent sequences of instructions to proceed concurrently

{Correctness

{Performance
Dependencies

Data-dependence

\[ r_3 \leftarrow (r_1) \text{ op } (r_2) \quad \text{Read-after-Write} \]
\[ r_5 \leftarrow (r_3) \text{ op } (r_4) \quad \text{(RAW) hazard} \]

Anti-dependence

\[ r_3 \leftarrow (r_1) \text{ op } (r_2) \quad \text{Write-after-Read} \]
\[ r_1 \leftarrow (r_4) \text{ op } (r_5) \quad \text{(WAR) hazard} \]

Output-dependence

\[ r_3 \leftarrow (r_1) \text{ op } (r_2) \quad \text{Write-after-Write} \]
\[ r_3 \leftarrow (r_6) \text{ op } (r_7) \quad \text{(WAW) hazard} \]
OoO Issue / Dispatch

Stall until sure that issuing will cause no dependences

• Is the required functional unit available?
• Is the input data available?
• Is it safe to write the destination

Dependencies due to registers can be determined at decode stage. Data hazards due to memory operands can be determined only after computing effective address.
OoO Implementation

OoO implemented via a *re-order buffer* (ROB):

- ROB remembers original program order for in-order commit.
- ROB stores computation graph by its edges—the dependencies.

<table>
<thead>
<tr>
<th>Task</th>
<th>State</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Committed</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Committed</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Completed</td>
<td>A</td>
</tr>
<tr>
<td>F</td>
<td>Ready</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>Completed</td>
<td>B</td>
</tr>
<tr>
<td>E</td>
<td>Ready</td>
<td>B, C</td>
</tr>
<tr>
<td>H</td>
<td>Waiting</td>
<td>E, F</td>
</tr>
</tbody>
</table>

(Not decoded)
Is this sufficient?

Number of registers limits maximum number of instructions in the pipeline.

WAW, WAR hazards are false dependencies introduced by limited number of architectural registers
False dependencies

4 now overwrites 1 since both use R1, so we can’t execute F before B even though there is no data dependency.

Register renaming helps remove false data dependencies.

3. Register conflicts arise—"false" dependencies

2. Given three architectural registers: R1, R2, R3

In-order semantics—"false" dependencies

1. Give data dependencies names...
Register Renaming

- *Register renaming* eliminates false dependencies by allocating a new register on every write.

- Requires many more “physical registers” than architectural registers and a layer of indirection.
  - Can think of architectural registers as “virtual registers” with the renaming table acting as a “register page table”.

- As before, the idea is to recover the *computation’s true structure* from the *over constrained* compiled code.
Register Renaming

Hooray we recovered the original register names!
⇒ No false dependencies!

<table>
<thead>
<tr>
<th>Architectural Register</th>
<th>Physical Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

“True” data dependencies

In-order semantics—“false” dependencies
OoO Implementation w/ renaming

Express dependencies in terms of the physical registers that pass the data between instructions.

<table>
<thead>
<tr>
<th>Architectural Register</th>
<th>Physical Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>P1, P4</td>
</tr>
<tr>
<td>R2</td>
<td>P2, P5</td>
</tr>
<tr>
<td>R3</td>
<td>P3, P6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>State</th>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Committed</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Committed</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>C</td>
<td>Completed</td>
<td>P1</td>
<td>P3</td>
</tr>
<tr>
<td>F</td>
<td>Ready</td>
<td>P3</td>
<td>P4</td>
</tr>
<tr>
<td>D</td>
<td>Completed</td>
<td>P2</td>
<td>P5</td>
</tr>
<tr>
<td>E</td>
<td>Ready</td>
<td>P2,P3</td>
<td>P6</td>
</tr>
<tr>
<td>H</td>
<td>Waiting</td>
<td>P4,P6</td>
<td></td>
</tr>
</tbody>
</table>
OoO: Summary

• OoO Processor: Restricted “data-flow” machine
  • Dynamically builds the data-flow graph

• The dynamically constructed data-flow graph is limited to the instruction window

• Tolerates long latency operations by executing independent instructions concurrently
Branch Prediction

Control Flow Dependences. How to handle them?

• Stall: Delay until we know the next PC
• Speculate: Guess next value
• Do something else: Multi-threading
Branch Predictors

• 1-bit predictor

• 2-bit predictor
Branch Predictors

Two empirical observations
1. A branch’s outcome can be correlated with other branches’ outcomes
   • Global branch correlation

2. A branch’s outcome can be correlated with past outcomes of the same branch
   • Local branch correlation
History-based Prediction

Index → History → Concat → Prediction → +/- → Taken
Two-level Predictor
Tournament Predictors

LHist → Prediction

GHist → Prediction

Chooser → Prediction
TAGE Predictor

- Entry tagging:
  Helps avoid aliasing between different branch scenarios
- Entry selection:
  Use branch address + history to accurately identify different branch scenario for same branch
- Longer branch histories as required:
  Use long histories for branches that actually benefit