Out of Order Execution

Pooja Saraff
Manoj Mardithaya
How the notion was conceived

Sequential Execution

Pipelining

Superscalar Execution

Out-Of-Order Execution
What is OOO?

- OOO execution is a type of processing where the instructions can begin execution as soon as operands are ready.
- Instructions are issued in order however execution proceeds out of order.

**Evolution**

<table>
<thead>
<tr>
<th>Year</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1964</td>
<td>CDC 6600</td>
</tr>
<tr>
<td>1966</td>
<td>IBM 360/91 <strong>Tomasulo's algorithm</strong></td>
</tr>
<tr>
<td>1993</td>
<td>IBM/Motorola PowerPC 601</td>
</tr>
<tr>
<td>1995</td>
<td>Fujitsu/HAL SPARC64, Intel Pentium Pro</td>
</tr>
<tr>
<td>1996</td>
<td>MIPS R10000, AMD K5</td>
</tr>
<tr>
<td>1998</td>
<td>DEC Alpha 21264</td>
</tr>
<tr>
<td>2011</td>
<td>Sandy Bridge</td>
</tr>
</tbody>
</table>
Architecture without Common Data Bus

Storage-to-register instruction

Stores field in register-to-register instruction

Store data instruction
Operation

Sets Control Bit

Decodes

As the buffer gets filled

Results
Results

• It doesn’t take care of data dependency
• Thus busy bit added – however FLOS hold-up because of busy sink register
• Solution to it – **Reservation Station**
  (control,sink,source)
• Execution now depends on appropriate reservation station
3 Types of Data Dependencies

- RAW (Read After Write)
  - R2 <- R1 + R3
  - R4 <- R2 + R3

- WAR (Write After Read)
  - R4 <- R1 + R3
  - R3 <- R1 + R2

- WAW (Write After Write)
  - R2 <- R4 + R7
  - R2 <- R1 + R2

Register Renaming uses in-order decoding to properly identify dependences.
Register Renaming

A: DIVF   F3,  F1,  F0  r1, -, -
B: SUBF   F2,  F1,  F0  r2, -, -
C: MULF   F0,  F2,  F4  r3, r2, -
D: SUBF   F6,  F2,  F3  r4, r2, r1
E: ADDF   F2,  F5,  F4  r5, -, -
F: ADDF   F0,  F0,  F2  r6, r3, r5

Register Rename Table

<table>
<thead>
<tr>
<th></th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>...</th>
<th>F30</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>R2</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>R3</td>
<td></td>
<td></td>
<td>R2</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>R3</td>
<td></td>
<td></td>
<td>R2</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R4</td>
</tr>
<tr>
<td>E</td>
<td>R3</td>
<td></td>
<td></td>
<td>R5</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R4</td>
</tr>
<tr>
<td>F</td>
<td>R6</td>
<td></td>
<td></td>
<td>R5</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R4</td>
</tr>
</tbody>
</table>

Need more physical registers than architectural
Ignores control flow for the time being.
Architecture

- 3 Adders
- 2 Multipliers
- Load buffers (6)
- Store buffers (3)
- FP Queue
- FP registers
- CDB: Common Data Bus
Tomasulo’s Algorithm Steps

• Issue
  - Issue if empty reservation station is found, fetch operands if they are in registers, otherwise assign a tag
  - If no empty reservation is found, stall and wait for one to get free
  - Renaming is performed here and WAW and WAR are resolved

• Execute
  – If operands are not ready, monitor the CDB for them
  – RAWs are resolved
  – When they are ready, execute the op in the FU

• Write Back
  – Send the results to CDB and update registers and the Store buffers
  – Store Buffers will write to memory during this step
AD F0, FLB1
AD F0, FLB1
AD F0,.....
AD F0, FLB1 - execution
AD F0,.....

However reservation tag for A1 has initial F0 tag i.e. 1010
Tag miss match
Drawbacks

• CDB is a bottleneck
  • Limits the execution time of any instruction to 2 cycles, minimum

• Complex implementation
HPS: a new microarchitecture
High Performance Substrate

• ~20 years after Tomasulo’s seminal paper, slightly different game.

• Three Tiers to Optimize
  – Global Parallelism
  – Sequential Flow
  – Local Parallelism

• Exploit local parallelism with a very small ‘window’ of instructions at the microarchitecture level
Add 1000, A, B
Add 1000, A, B

Decoder

Merger

Node Tables

Result Buffer

Fn. Unit
Fn. Unit
Fn. Unit
Fn. Unit

Read from Addr. A
Add 1000, A
Write to Addr. B
Add 1000, A, B
Decoding

Read from Addr. A
Add 1000, A
Write to Addr. B

Decoder

Merger

Result Buffer

Node Tables

Fn. Unit

Fn. Unit

Fn. Unit

Fn. Unit
Add 1000, A, B
Add 100, B, C
Add 100, B, C

Add 1000, A, B
Add 1000, A, B

Decoder

Read from Addr. A
Add 1000, A
Write to Addr. B

Merger

Result Buffer

Node Tables

Fn. Unit
Fn. Unit
Fn. Unit
Fn. Unit
Still Unknown:
• Format of the Node Table
• Finding Dependencies
• Scheduling Nodes
Still Unknown:

- Format of the Node Table
- Finding Dependencies
- Scheduling Nodes
What about Memory Access?
Still Unknown:

- Format of the Node Table
- Finding Dependencies
- Scheduling Nodes
Scheduling

- Node is ‘ready to fire’ when Ready Bits of all operands are set.
- Oldest ‘Fires’ when a Functional Unit is ready.

* Can the scheduling make smarter choices?
Still Unknown:

- Format of the Node Table
- Finding Dependencies
- Scheduling Nodes
Advantages over Tomasulo’s Algorithm

• No ‘renaming’ involved, register alias table.
  – Eliminates anti and output dependencies *without* messy renaming schemes.

• Don’t need to queue instructions to ‘reservation stations’ before both source and sink are ready.

• Node tables allow an ‘active window’ worth of possible parallelism.
HPSm[minimal]

• Implementation of the HPS model.
• Minimal, because of practical issues HPS did not address:
  – Branch Prediction.

  – Memory dependencies.

  – Number of nodes per instruction.
HPSm[minimal]

• Implementation of the HPS model.

• Minimal, because of practical issues HPS did not address:
  – Branch Prediction.
    • Fixed to 1 unresolved prediction at a time.
  – Memory dependencies.
    • Fire oldest writes, then oldest reads.
  – Number of nodes per instruction.
    • At most two.
Wrong Predictions and Exceptions

• We are executing out of order.
  – What happens when the executed instructions shouldn’t have been?
  – What happens if an exception is thrown?

• Solution: Register Alias Table has backups

<table>
<thead>
<tr>
<th>Current</th>
<th>Backup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 B 6 7 8</td>
</tr>
<tr>
<td>Current</td>
<td></td>
</tr>
</tbody>
</table>
Wrong Predictions and Exceptions

When the instructions are decoded:
Wrong Predictions and Exceptions

When they are executed (out of order):

```
1 2 3 4 5 6 7 8
```

<table>
<thead>
<tr>
<th>Current</th>
<th>Backup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current</td>
<td>Backup</td>
</tr>
<tr>
<td>Current</td>
<td>Backup</td>
</tr>
<tr>
<td>Current</td>
<td>Backup</td>
</tr>
<tr>
<td>Current</td>
<td>Backup</td>
</tr>
</tbody>
</table>
Wrong Predictions and Exceptions

When they are executed (out of order):

```
Current
Backup
Current
Backup
Current
Backup
```
Wrong Predictions and Exceptions

When they are executed (out of order):
Wrong Predictions and Exceptions

When they are executed (out of order):

```
1 2 3 4 5  B  X  X  X
```
Memory Dependencies

• Algorithm:
  – Fire the oldest fire-able memory write. If none:
    – Fire the oldest fire-able memory read.
• All access addresses translated, sit in the Write Buffer.
• *Reads* check the Write Buffer before going to memory.
• Write to memory only when the instruction RETIRES.
HPSm Results

- Comparison against the RISC II, with both non-optimizing and optimizing compilers
Out of Order: *Today*

- Sandy Bridge, POWER, Bulldozer and Bobcat all have **Awesome** out of order execution capabilities
  - Use Physical Register Files for renaming.
- Cortex A9, over a year ago – 1\textsuperscript{st} mobile CPU with an OOO Execution Engine.
- Superscalar, OOO, Register Renaming – We’re hitting an ‘ILP Wall’
Thank you

Any Questions?