Vorlesung / Course IN2075: Mikroprozessoren / Microprocessors

Superscalarity

8 Jan 2018

Carsten Trinitis

Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR)
Technische Universität München
Superscalarity
Parallel Execution & ILP

**Parallelism at Data Level**
- fetch
- decode
- execute
- writeback

- instruction stream

  e.g. MMX or SSE instructions

**Parallelism at Instruction Level**
- parallel fetch
- decode & issue
- execute
- writeback

- instruction stream

  add
  mul
  jmp
Instruction Level Parallelism

• Superscalarity:
  - identification of independent instructions by *hardware*.
Superscalarity

• Pipeline-Stages:
  - Parallel fetch
  - Decode
  - Fetch Data
    - Issue (assign instructions to functional units)
      - In-order issue: e.g. Pentium I
      - Out-of-Order issue (since Pentium II)
  - Process Data
  - Write Data
Superscalar Execution

- **Pure pipelining:**
  - Every unit exists once in a processor
  - In every cycle the first stage is filled
  - One instruction per clock cycle started
Superscalar Execution

- Replicate functional units
  - Create several pipelines in the microprocessors
  - Start several instructions concurrently
  - Potential speed-up: number of replicas
Implementation Issues

- Normally not implemented as N pipelines
  - Flexible assignment of free units at each stage
  - Possible:
    - Different number of units at some stages
    - Specialised Units (load/store, arithmetic)

- Important property: „Issue Width“
  - How many instructions can at most be issued at one clock cycle

- Name: Super\textbf{SCALAR} in contrast to vectors
Dealing with Conflicts

- Similar problems as with standard pipelining
  - Need to observe dependencies
  - Need to observe control flow changes

- Additional problem
  - Resource conflicts when accessing functional units
  - Assignments and Reservation of Units
  - Need for complex hardware

- Possible optimisation
  - Out of order issue / execution
  - Instructions with open dependencies wait
From a Software Point of View

- Again similar as for pipelining
  - Arrange code to avoid dependencies
  - Try to group instructions correctly
  - Reduce the task of the dynamic scheduler
  - Task of the compiler backend

- Utilise software tools
  - Play with compiler options
  - For some architectures, additional tools exist
    - E.g. Parallel Studio XE from Intel
    - Useful for performance debugging of kernels
The Intel Pentium II
Intel Sandy Bridge
AMD Bulldozer
AMD Bulldozer
AMD Bulldozer
AMD Bulldozer
Superscalar Pipelines (Overview)

- **Conflicts:**
  - **Structural conflicts**
    - E.g. 2 integer units, one FPU, and one load / store unit
  - **Control flow conflicts**
  - **Data conflicts**
    - Read after write (RAW)
    - Write after read (WAR, commonly due to reordering)
    - Write after write (WAW)
    - Techniques to handle data conflicts:
      - Scoreboarding, 1964, Control Data Corporation (CDC)
      - Tomasulo Algorithm, Register Renaming, 1967, IBM 360/91
Simplest Case: In-Order Execution

• Examples:
  - Intel Pentium I, Atom, simpler ARM architectures

• Principle:
  - $N$ (usually 2) instructions are fetched at the same time.
  - If a RAW conflict is detected, the execution of one of the instructions is delayed (NOP inserted).
  - WAW conflicts usually only occur in conjunction with a RAW:
    - (1) $\text{MUL} \ r1 \leftarrow r2,r3$
    - (2) $\text{ADD} \ r4 \leftarrow r1,r5$  // RAW with line (1)
      // $r1$ no longer needed and could be reused:
    - (3) $\text{SUB} \ r1 \leftarrow r10,1$  // WAW with line (1)
  - WAR conflicts usually do not occur
Out-of-Order Execution: Motivation

(1) MUL r1, r2, r3
(2) SUB r4, r1, r5 // RAW data conflict with (1)
(3) MUL r6, r2, r3 // resource conflict with (1)
(4) SUB r1, r10, 1 // WAW data conflict with (1)
// WAR data conflict with (2)

• Problem:
  - Line (2) has to wait for the result of line (1) (e.g. 6 cycles)
  - Line (3) has to wait for only 1 cycle!
  - If WAR and WAW conflicts could be resolved, line (4) could also start after 1 cycle
  - Without reordering of instructions, lines (3) and (4) have to wait unnecessarily.
Out of Order Execution: Overview

• Two implementations:

- Scoreboarding
  - Simple, centralised approach
  - Register file is extended by status bits

- Register-renaming & Tomasulo Algorithm
  - More powerful, decentralised approach
  - Each functional unit (FU) is extended by so-called reservation stations
Scoreboarding

- Register file is extended by status bits:
  - Valid: cleared if register will be written to
  - Read-tag: stores number of FU waiting for a result which will be written to this register.
  - Pref-tag: tells order of read / write, important to discriminate between RAW and WAR.

1. $A = Y / Z$
2. $B = C - A$
3. $C = W \times X$
4. $A = U + V$

<table>
<thead>
<tr>
<th></th>
<th>valid</th>
<th>read</th>
<th>pref</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>true</td>
<td>none</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>true</td>
<td>none</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>none</td>
<td>X</td>
</tr>
</tbody>
</table>

© Carsten Trinitis, Microprocessors Lecture @TUM, WS 2017/2018
Scoreboarding

- Register file is extended by status bits:
  - Valid: cleared if register will be written to
  - Read-tag: stores number of FU waiting for a result which will be written to this register.
  - Pref-tag: tells order of read / write, important to discriminate between RAW and WAR.

<table>
<thead>
<tr>
<th></th>
<th>valid</th>
<th>read</th>
<th>pref</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>false</td>
<td>none</td>
<td>X</td>
</tr>
<tr>
<td>B</td>
<td>true</td>
<td>none</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>none</td>
<td>X</td>
</tr>
</tbody>
</table>

Reg A no longer valid!

1. $A = Y / Z$
2. $B = C - A$
3. $C = W * X$
4. $A = U + V$
Scoreboarding

- Register file is extended by status bits:
  - Valid: cleared if register will be written to
  - Read-tag: stores number of FU waiting for a result which will be written to this register.
  - Pref-tag: tells order of read / write, important to discriminate between RAW and WAR.

1. $A = Y / Z$
2. $B = C - A$
3. $C = W \times X$
4. $A = U + V$

<table>
<thead>
<tr>
<th></th>
<th>Valid</th>
<th>Read</th>
<th>Pref</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>false</td>
<td>SUB</td>
<td>W</td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>none</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>true</td>
<td>SUB</td>
<td>X</td>
</tr>
</tbody>
</table>

A and C required for SUB

SUB unit waits until register A is written to
Scoreboarding

- Register file is extended by status bits:
  - Valid: cleared if register will be written to
  - Read-tag: stores number of FU waiting for a result which will be written to this register.
  - Pref-tag: tells order of read / write, important to discriminate between RAW and WAR.

1. \( A = Y / Z \)
2. \( B = C - A \)
3. \( C = W \times X \)
4. \( A = U + V \)

<table>
<thead>
<tr>
<th></th>
<th>valid</th>
<th>read</th>
<th>pref</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>false</td>
<td>SUB</td>
<td>W</td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>none</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>false</td>
<td>SUB</td>
<td>R</td>
</tr>
</tbody>
</table>

Write back to C must wait for read by SUB
Scoreboarding

- Register file is extended by status bits:
  - Valid: cleared if register will be written to
  - Read-tag: stores number of FU waiting for a result which will be written to this register.
  - Pref-tag: tells order of read / write, important to discriminate between RAW and WAR.

<table>
<thead>
<tr>
<th></th>
<th>valid</th>
<th>read</th>
<th>pref</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>false</td>
<td>SUB</td>
<td>W</td>
</tr>
<tr>
<td>B</td>
<td>false</td>
<td>none</td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td>false</td>
<td>SUB</td>
<td>R</td>
</tr>
</tbody>
</table>

1. \( A = Y / Z \)
2. \( B = C - A \)
3. \( C = W \times X \)
4. \( A = U + V \)

Cannot issue (4)!

© Carsten Trinitis, Microprocessors Lecture @TUM, WS 2017/2018
Scoreboarding - Properties

- Handles RAW and WAR dependencies.
- Execution / writeback of respective instructions is delayed.
- For each register, only one dependency can be stored.
- Further dependencies block execution.
- WAW dependencies block execution.
Tomasulo Algorithm - Goals

• Get rid of WAR and WAW by register renaming.

• RAW still requires delay, though.

• RAW does not wait until writeback, but results are directly forwarded from FU to FU.

• Execution can also be blocked due to resource conflicts.
Reservation Stations

1. $A = Y / Z$
2. $B = C - A$
3. $C = W \times X$
4. $A = U + V$
Reservation Stations

Values from registers X and Y are copied into the reservation station

Reference to DIV-unit

1. A = Y / Z
2. B = C - A
3. C = W * X
4. A = U + V
Reservation Stations

1. A = Y / Z
2. B = C - A
3. C = W * X
4. A = U + V
Reservation Stations

1. \( A = Y / Z \)
2. \( B = C - A \)
3. \( C = W \times X \)
4. \( A = U + V \)

---

© Carsten Trinitis, Microprocessors Lecture @TUM, WS 2017/2018
Reservation Stations

1. \( A = Y / Z \)
2. \( B = C - A \)
3. \( C = W \times X \)
4. \( A = U + V \)

Following instructions must wait because all reservation stations are occupied.
Superscalar processors can issue more than one instruction per cycle.

**Two varieties:**
- In order: one conflict blocks overall stream.
- Out-of-order: a conflict only blocks one instruction.

Tomasulo Algorithm used in out-of-order superscalar processors.
- Detects and avoids WAW and WAR conflicts
- Detects RAW and resource conflicts

Limitation in practice:
- # of independent instructions found: \( \approx 4-6 \)