Superscalar Processors
- 7.1 Introduction
- 7.2 Parallel decoding
- 7.3 Superscalar instruction issue
- 7.4 Shelving
- 7.5 Register renaming
- 7.6 Parallel execution
- 7.7 Preserving the sequential consistency of instruction execution
- 7.8 Preserving the sequential consistency of exception processing
- 7.9 Implementation of superscalar CISC processors using a superscalar RISC core
- 7.10 Case studies of superscalar processors

Superscalar Processors vs. VLIW
- Parallel Issue
- Parallel Execution
- {Hardware} Dynamic Instruction Scheduling
- Currently the predominant class of processors
  - Pentium
  - PowerPC
  - UltraSparc
  - AMD K5
  - HP PA7100
  - DEC α

Superscalar Processor: Intro
- Parallel Issue
- Parallel Execution
- {Hardware} Dynamic Instruction Scheduling
- Currently the predominant class of processors

Emergence and spread of superscalar processors

Evolution of superscalar processor

Specific tasks of superscalar processing
Parallel decoding (and Dependencies check)

- What need to be done

Decoding and Pre-decoding

- Superscalar processors tend to use 2 and sometimes even 3 or more pipeline cycles for decoding and issuing instructions
- >> Pre-decoding:
  - shifts a part of the decode task up into loading phase
  - resulting of pre-decoding
    - the instruction class
    - the type of resources required for the execution
    - in some processor (e.g. UltraSparc), branch target addresses calculation as well
  - the results are stored by attaching 4-7 bits
- + shortens the overall cycle time or reduces the number of cycles needed

The principle of perdecoding

Number of perdecode bits used

<table>
<thead>
<tr>
<th>Type of first volume shipment</th>
<th>Number of perdecode bits appended to each instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA 7200 (1995)</td>
<td>5</td>
</tr>
<tr>
<td>PA 8000 (1996)</td>
<td>5</td>
</tr>
<tr>
<td>PowerPC 620 (1996)</td>
<td>7</td>
</tr>
<tr>
<td>UltraSparc (1995)</td>
<td>4</td>
</tr>
<tr>
<td>HAL PMI (1995)</td>
<td>4</td>
</tr>
<tr>
<td>AMD K5 (1995)</td>
<td>5</td>
</tr>
<tr>
<td>R10000 (1996)</td>
<td>4</td>
</tr>
</tbody>
</table>

1 In the K5, 5 perdecode bits are added to each byte

Specific tasks of superscalar processing: Issue

7.3 Superscalar instruction issue

- How and when to send the instruction(s) to EU(s)
Issue policies

Instruction issue policies of superscalar processors:

Instruction rate (How many instructions/cycle)

- CISC about 2
- RISC:

Issue stopped by True dependency

- True dependency → (Blocked: need to wait)

Issue order of instructions

Instructions are issued strictly in program order

In-Order Issue

Out-of-order Issue
**Aligned vs. unaligned issue**

![Diagram showing the alignment of instruction issues]

**Issue policies: Use of Shelving**

![Diagram showing the use of shelving policies]

**Direct Issue**

![Diagram showing the direct issue process]

**The principle of shelving: Indirect Issue**

![Diagram showing the indirect issue with shelving]

**Design space of shelving**

![Diagram showing the design space of shelving]

**Scope of shelving**

![Diagram showing the scope of shelving]

**Issue performance, impact**
**Layout of shelving buffers**

- Type of the shelving buffers
- Number of shelving buffer entries
- Number of read and write ports

**Implementation of shelving buffer**

- Standalone shelving buffers
- Combined buffers for shelving, renaming, and reordering
  - Reservaton stations
  - DRS (Deferred scheduling, Register renaming, Instruction Shelve)

**Basic variants of shelving buffers**

- Using a combined buffer for shelving, renaming, and reordering
  - Combined buffer for shelving, renaming, and reordering
    - From decode/issue
      - Recorder buffer which is also used for renaming and shelving
        - Also designed as DRS (Deferred scheduling, Register renaming, Instruction Shelve)

**Number of shelving buffer entries**

<table>
<thead>
<tr>
<th>Processor</th>
<th>Total number of shelves</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC 603 (1993)</td>
<td>3</td>
</tr>
<tr>
<td>PowerPC 604 (1994)</td>
<td>12</td>
</tr>
<tr>
<td>PowerPC 620 (1995)</td>
<td>15</td>
</tr>
<tr>
<td>M386 (1994)</td>
<td>42</td>
</tr>
<tr>
<td>K5 (1995)</td>
<td>14</td>
</tr>
<tr>
<td>PM1 (Sparc5i) (1995)</td>
<td>36</td>
</tr>
<tr>
<td>PentiumPro (1995)</td>
<td>20</td>
</tr>
<tr>
<td>R10000 (1996)</td>
<td>48</td>
</tr>
</tbody>
</table>

**Number of read and write ports**

- how many instructions may be written into (input ports) or
- read out from (output parts) a particular shelving buffer in a cycle
- depend on individual, group, or central reservation stations
- Dispatch policy

• Selection Rule
  ➔ Specifies when instructions are considered executable
  ➔ e.g. Dataflow principle of operation
    ➔ Those instructions whose operands are available are executable.

• Arbitration Rule
  ➔ Needed when more instructions are eligible for execution than can be disseminated.
  ➔ e.g. choose the 'oldest' instruction.

• Dispatch order
  ➔ Determines whether a non-executable instruction prevents all subsequent instructions from being dispatched.

---

Dispatch policy: Dispatch order

- Trend of Dispatch order

- Dispatch rate (instructions/cycle)

---

- Scheme for checking the availability of operands:
  The principle of scoreboard

---
Schemes for checking the availability of operand

- Direct check of the scoreboard bits
  - The availability of source operands is not explicitly indicated in the RS.
  - Thus, the scoreboard bits are tested for availability.
  - Usually employed if operands are fetched during instruction dispatch, as assumed below.

- Check of the explicit status bits
  - The availability of source operands is explicitly indicated in the RS.
  - These explicit status bits are tested for availability.
  - Usually employed if operands are fetched during instruction issue, as assumed below.

Operands fetched during dispatch or during issue

- Use of multiple buses for updating multiple individual reservation stations

- Internal data paths of the PowerPC 604

7.4.6 Detail Example of Shelving

- Issuing the following instruction
  - Cycle 1: `mul r1, r2, r3`
  - Cycle i+1: `add r2, r3, r5`
  - `add r3, r4, r6`

- Format: Rs1, Rs2, Rd
Example overview

Cycle i: Issue of the 'mul' instruction into the reservation station and fetching of the corresponding operands.

Cycle i+1: Checking for executable instructions and dispatching of the 'mul' instruction.

Cycle i+1 (2nd phase): Issue of the subsequent two 'ad' instructions into the reservation station.

Cycle i+2: Checking for executable instructions (mul not yet completed).

Cycle i+3: Updating the FX register file with the result of the 'mul' instruction.
Cycle i+3 (2nd phase): Checking for executable instructions and dispatching the 'older' 'ad' instruction

Instruction Issue policies: Register Renaming

Register Remaining and dependency
- three-operand instruction format
- e.g. Rd, Rs1, Rs2
- False dependency (WAW)
  - *mul* r2, ..., ...
  - *add* r2, ..., ...
  - two different rename buffer have to allocated
- True data dependency (RAW)
  - *mul* r2, ..., ...
  - *add* ..., r2, ...
  - rename to e.g.
    - *mul* p12, ..., ...
    - *add* ..., p12, ...

Static or Dynamic Renaming

Design space of register renaming

Chronology of introduction of renaming (high complexity, Sparc64 used 371K transistors that is more than i386)
-Scope of register renaming

Partial renaming
Renaming is restricted to particular instruction types.

Full renaming
Renaming comprises all eligible instruction types.

PowerPC 603 (1993)
Am29000 (1995)

-Layout of rename buffers

Type of the rename buffers
Number of rename buffers
Basic mechanism used for accessing rename buffers

-Number of rename buffers

<table>
<thead>
<tr>
<th>Processor type</th>
<th>Implementation of renaming</th>
<th>Number of rename buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merged rename and arch. register file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 603 (1993)</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>PowerPC 604 (1995)</td>
<td>(32 arch. + 8 rename)</td>
<td></td>
</tr>
<tr>
<td>PowerPC 620 (1996)</td>
<td>(32 arch. + 32 rename)</td>
<td></td>
</tr>
<tr>
<td>Separate rename register file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PowerPC 603 (1993)</td>
<td>n.a.</td>
<td>4</td>
</tr>
<tr>
<td>PowerPC 604 (1995)</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>PowerPC 620 (1996)</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

-Type of rename buffers

- Basic mechanisms used for accessing rename buffers

- Rename buffers with associative access (latter e.g.)
- Rename buffers with indexed access

- Rename buffers hold intermediate results

- Each time a Destination register is referred to, a new rename register is allocated to it.
- Final results are stored in the Architectural Register file.
- Access both rename buffer and architectural register file. If found in both, the data content in rename buffer (the intermediate result) is chosen.
- When an instruction completed (retired), ROB (retire only in strict program sequence), the correspond rename buffer entry is writing into the architectural register file (as a result modifying the actual program state)
- The correspond rename buffer entry can be de-allocated.
-Operand fetch policies and Rename Rate

- rename bound: fetch operands during renaming (during instruction issue)
- dispatch bound: fetch operand during dispatching

- Rename Rate

- the maximum number of renames per cycle
- equals the issue rate: to avoid bottlenecks.

7.5.8 Detailed example of renaming

- renaming:
  - `mul r2, r0, r1`
  - `ad r3, r1, r2`
  - `sub r2, r0, r1`

- format:
  - `op Rd, Rs1, Rs2`

- Assume:
  - separate rename register file,
  - associative access, and
  - operand fetching during renaming

Structure of the rename buffers and their supposed initial contents

- Latest bit: the most recent rename 1, previous 0

<table>
<thead>
<tr>
<th>Entry</th>
<th>Dest reg.no.</th>
<th>Value</th>
<th>Value valid</th>
<th>Latest bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>40</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Head</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tail</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rename registers

Renaming steps

- Allocation of a free rename register to a destination register
- Accessing valid source register value or a register value that is not yet available
- Re-allocation of destination register
- Updating a particular rename buffer with a computed result
- De-allocation of a rename buffer that is no longer needed.

Allocation of a new rename buffer to destination register (circular buffer: Head and Tail) (before allocation)

(After allocation) of a destination register
Accessing available register values

Accessing a register value that is not yet available

Re-allocate of r2 (a destination register)

Updating the rename buffers with computed result of [mul r2, r0, r1] (register 2 with the result 0)

Deallocation of the rename buffer no. 0 (ROB retires instructions) (update tail pointer)

7.6 Parallel Execution

- Executing several instructions in parallel
  - instructions will generally be finished in out-of-program order
- to finish
  - operation of the instruction is accomplished,
  - except for writing back the result into
    - the architectural register or
    - memory location specified, and/or
    - updating the status bits
- to complete
  - writing back the results
- to retire (ROB)
  - write back the results, and
  - delete the completed instruction from the last ROB entry
7.7 Preserving Sequential Consistency

of instruction execution //

- Multiple EUs operating in parallel, the overall instruction execution should
- mimic sequential execution
  - the order in which instruction are completed
  - the order in which memory is accessed

Sequential consistency models

Consistency relate to
instruction completions or memory access

- Consistency of the sequence of instruction completions
- Consistency of the sequence of memory accesses

- Weak processor consistency
  - Instructions may complete out of order, programs that do not depend on this order may
  - proceed as if they completed in order

- Strong processor consistency
  - Instructions complete strictly in program order

- Weak memory consistency
  - Memory access due to load and store instructions may be out of order, provided that no
  - dependencies are affected

- Strong memory consistency
  - Memory is accessed due to load and store instructions strictly in program order

Trend and performance

<table>
<thead>
<tr>
<th>Processor</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency</td>
<td>Consistency</td>
</tr>
<tr>
<td>Weak</td>
<td>Weak</td>
</tr>
<tr>
<td>Strong</td>
<td>Strong</td>
</tr>
<tr>
<td>Consistency</td>
<td>Consistency</td>
</tr>
</tbody>
</table>

- Allows the reordering of memory access
  - It permits load/store reordering
    - Either loads can be performed before pending stores, or vice versa
    - A load can be performed before pending stores only if
      - None of the preceding stores has the same target address as the load
  - It makes speculative loads or stores feasible
    - When addresses of pending stores are not yet available,
      - Speculative loads avoid delaying memory accesses, perform the load anywhere.
    - When store addresses have been computed, they are compared against the addresses of all younger loads.
    - Re-load is needed if any hit is found.
  - It allows cache misses to be hidden
    - If a cache miss, it allows loads to be performed before the missed load, or it allows stores to be performed before the missed store.

Using Re-Order Buffer (ROB) for Preserving:
The order in which instruction are completed

- 1. Instruction are written into the ROB in strict program order:
  - One new entry is allocated for each active instruction
- 2. Each entry indicates the status of the corresponding instruction
  - Issued (i), in execution (s), already finished (f)
- 3. An instruction is allowed to retire only if it has finished and all previous instruction are already retired.
  - Retiring in strict program order
  - Only retiring instructions are permitted to complete, that is, to update the program state:
    - By writing their result into the referenced architectural register or memory
Principle of the ROB {Circular Buffer}

- Introduction of ROBs in commercial superscalar processors
  - Use ROB for speculative execution
    - Guess the outcome of a branch and execute the path before the condition is ready
    - Each entry is extended to include a speculative status field indicating whether the corresponding instruction has been executed speculatively
    - Speculatively executed instructions are not allowed to retire before the related condition is resolved
    - After the related condition is resolved,
      - If the guess turns out to be right, the instruction can retire in order.
      - If the guess is wrong, the speculative instructions are marked to be cancelled. Then, instruction execution continues with the correct instructions.

Design space of ROBs

Basic layout of ROBs

ROB implementation details

<table>
<thead>
<tr>
<th>ROB size</th>
<th>Issue rate</th>
<th>Retire rate</th>
<th>Intermediate results stored</th>
<th>Designation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES/9000</td>
<td>32</td>
<td>2</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>PowerPC 603 (1993)</td>
<td>4</td>
<td>2</td>
<td>n.a.</td>
<td>Completion unit</td>
</tr>
<tr>
<td>PowerPC 603 (1993)</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>PowerPC 604 (1995)</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td>PowerPC 620 (1998)</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>PentiumPro (1995)</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>Am29000sup (1995)</td>
<td>40</td>
<td>3</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>K5 (1995)</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>Pentium Pro (1995)</td>
<td>56</td>
<td>4</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td>R10K000 (1998)</td>
<td>32</td>
<td>4</td>
<td>4</td>
<td>No</td>
</tr>
</tbody>
</table>
7.8 Preserving the Sequential consistency of exception processing

- When instructions are executed in parallel, interrupt request, which are caused by exceptions arising in instruction execution, are also generated out of order.
- If the requests are acted upon immediately, the requests are handled in different order than in a sequential operation processor called imprecise interrupts.
- Precise interrupts: handling the interrupts in consistent with the state of a sequential processor.

Use ROB for preserving sequential order of interrupt requests

- Interrupts generated in connection with instruction execution can be handled at the correct point in the execution, by accepting interrupt requests only when the related instruction becomes the next to retire.

7.9 Implementation of superscalar CISC processors using superscalar RISC core

- CISC instructions are first converted into RISC-like instructions during decoding.
  - Simple CISC register-to-register instructions are converted to single RISC operation (1-to-1).
  - CISC ALU instructions referring to memory are converted to two or more RISC operations (1-to-(2-4)).
    - SUB EAX, [EDI]
    - MOV EBX, [EDI]
    - SUB EAX, EBX
  - More complex CISC instructions are converted to long sequences of RISC operations (1-to-(more than 4)).
- On average one CISC instruction is converted to 1.5-2 RISC operations.

The principle of superscalar CISC execution using a superscalar RISC core

- PentiumPro: Decoding/converting CISC instructions to RISC operations (are done in program order).
Case Studies: R10000
Core part of the micro-architecture of the R10000

Case Studies: PowerPC 620

Case Studies: PentiumPro
Core part of the micro-architecture

PentiumPro Long pipeline:
Layout of the FX and load pipelines

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{array}
\]