UNIT- 5

Chapter 14
Instruction Level Parallelism and Superscalar Processors
What is Superscalar?

- Use multiple independent instruction pipeline.
- Instruction level parallelism
- Identify dependencies in program
- Eliminate Unnecessary dependencies
- Use additional registers & renaming of register references in original code.
- Branch prediction improves efficiency
What is Superscalar?

- RISC-like instructions, one per word
- Multiple parallel execution units
- Reorders and optimizes instruction stream at run time
- Branch prediction with speculative execution of one path
- Loads data from memory only when needed, and tries to find the data in the caches first
Why Superscalar?

- In 1987, scalar instruction
- Most operations are on scalar quantities
- Improving these operations to get an overall improvement
- High performance microprocessors.
General Superscalar Organization
Superscalar Operational Block Diagram

A Superscalar Processor with 3 Functional Units
Instruction Flow in Superscalar Architecture
Instruction Flow

A Pipeline architecture in more detail

A Superscalar microarchitecture
Superscalar V/S Superpipelined

- **Ordinary**: -
  - One instruction/clock cycle & can perform one pipeline stage per clock cycle.

- **Superpipelined**: -
  - 2 pipeline stage per cycle
  - *Execution time*: - half a clock cycle
  - Double internal clock speed gets two tasks per external clock cycle

- **Superscalar**: -
  - allows parallel fetch execute
  - Start of the program & at branch target it performs better than above said.
Superscalar v Superpipeline
Limitations

• Instruction level parallelism
• Compiler based optimisation
• Hardware techniques
• Limited by
  — True data dependency
  — Procedural dependency
  — Resource conflicts
  — Output dependency
  — Antidependency
True Data Dependency

- ADD r1, r2 (r1 := r1+r2;)
- MOVE r3,r1 (r3 := r1;)
- Can fetch and decode second instruction in parallel with first
- Can NOT execute second instruction until first is finished
Procedural Dependency

• Can not execute instructions after a branch in parallel with instructions before a branch

• Also, if instruction length is not fixed, instructions have to be decoded to find out how many fetches are needed

• This prevents simultaneous fetches
Resource Conflict

- Two or more instructions requiring access to the same resource at the same time
  - e.g. two arithmetic instructions
- Can duplicate resources
  - e.g. have two arithmetic units
Effect of Dependencies

Key:
- Ifetch
- Decode
- Execute
- Write

- No Dependency
- Data Dependency (i1 uses data computed by i0)
- Procedural Dependency
- Resource Conflict (i0 and i1 use the same functional unit)
Design Issues

- Instruction level parallelism
  - Instructions in a sequence are independent
  - Execution can be overlapped
  - Governed by data and procedural dependency

- Machine Parallelism
  - Ability to take advantage of instruction level parallelism
  - Governed by number of parallel pipelines
Instruction Issue Policy

- Order in which instructions are fetched
- Order in which instructions are executed
- Order in which instructions change registers and memory
In-Order Issue
In-Order Completion

• Issue instructions in the order they occur
• Not very efficient
• May fetch >1 instruction
• Instructions must stall if necessary
In-Order Issue In-Order Completion (Diagram)

<table>
<thead>
<tr>
<th>Decode</th>
<th>Execute</th>
<th>Write</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1 I2</td>
<td>I1 I2</td>
<td>I1 I2</td>
<td>1</td>
</tr>
<tr>
<td>I3 I4</td>
<td>I1 I1</td>
<td>I3 I3</td>
<td>2</td>
</tr>
<tr>
<td>I3 I4</td>
<td>I1 I1</td>
<td>I4 I4</td>
<td>3</td>
</tr>
<tr>
<td>I5 I6</td>
<td>I5 I5</td>
<td>I5 I5</td>
<td>4</td>
</tr>
<tr>
<td>I6 I6</td>
<td>I6 I6</td>
<td>I5 I6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>
In-Order Issue
Out-of-Order Completion

- **Output dependency**
  - R3 := R3 + R5; (I1)
  - R4 := R3 + 1;  (I2)
  - R3 := R5 + 1;  (I3)
  - I2 depends on result of I1 - data dependency
  - If I3 completes before I1, the result from I1 will be wrong - output (read-write) dependency
In-Order Issue Out-of-Order Completion (Diagram)
Out-of-Order Issue
Out-of-Order Completion

- Decouple decode pipeline from execution pipeline
- Can continue to fetch and decode until this pipeline is full
- When a functional unit becomes available an instruction can be executed
- Since instructions have been decoded, processor can look ahead
# Out-of-Order Issue Out-of-Order Completion (Diagram)

<table>
<thead>
<tr>
<th>Decode</th>
<th>Window</th>
<th>Execute</th>
<th>Write</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>I2</td>
<td>I1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>I3</td>
<td>I4</td>
<td>I1</td>
<td>I2</td>
<td>2</td>
</tr>
<tr>
<td>I5</td>
<td>I6</td>
<td>I1</td>
<td>I3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>I1, I2</td>
<td>I1, I3</td>
<td>I4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>I3, I4</td>
<td></td>
<td>I5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>I4, I5, I6</td>
<td>I6, I4</td>
<td>I5</td>
<td>6</td>
</tr>
</tbody>
</table>
Antidependency

- Write-write dependency
  - \( R3 := R3 + R5; \) (I1)
  - \( R4 := R3 + 1; \) (I2)
  - \( R3 := R5 + 1; \) (I3)
  - \( R7 := R3 + R4; \) (I4)
  - I3 can not complete before I2 starts as I2 needs a value in R3 and I3 changes R3
Reorder Buffer

- Temporary storage for results
- Commit to register file in program order
Register Renaming

• Output and antidependencies occur because register contents may not reflect the correct ordering from the program
• May result in a pipeline stall
• Registers allocated dynamically
  —i.e. registers are not specifically named
Register Renaming example

- $R3b := R3a + R5a$  \( (I1) \)
- $R4b := R3b + 1$  \( (I2) \)
- $R3c := R5a + 1$  \( (I3) \)
- $R7b := R3c + R4b$  \( (I4) \)
- Without subscript refers to logical register in instruction
- With subscript is hardware register allocated
- Note $R3a$ $R3b$ $R3c$
- Alternative: Scoreboarding
  - Bookkeeping technique
  - Allow instruction execution whenever not dependent on previous instructions and no structural hazards
Machine Parallelism

- Duplication of Resources
- Out of order issue
- Renaming
- Not worth duplication functions without register renaming
- Need instruction window large enough (more than 8)
Speedups of Machine Organizations Without Procedural Dependencies

Without renaming

With renaming

Window size (construction): 8, 16, 32

Speedup
Branch Prediction

- 80486 fetches both next sequential instruction after branch and branch target instruction
- Gives two cycle delay if branch taken
RISC - Delayed Branch

- Calculate result of branch before unusable instructions pre-fetched
- Always execute single instruction immediately following branch
- Keeps pipeline full while fetching new instruction stream
- Not as good for superscalar
  - Multiple instructions need to execute in delay slot
  - Instruction dependence problems
- Revert to branch prediction
Superscalar Execution
Superscalar Implementation

- Simultaneously fetch multiple instructions
- Logic to determine true dependencies involving register values
- Mechanisms to communicate these values
- Mechanisms to initiate multiple instructions in parallel
- Resources for parallel execution of multiple instructions
- Mechanisms for committing process state in correct order
Pentium 4

- 80486 - CISC
- Pentium – some superscalar components
  - Two separate integer execution units
- Pentium Pro – Full blown superscalar
- Subsequent models refine & enhance superscalar design
Pentium 4 Block Diagram

AGU = address generation unit
BTB = branch target buffer
D-TLB = data translation lookaside buffer
I-TLB = instruction translation lookaside buffer
Pentium 4 Operation

- Fetch instructions from memory in order of static program
- Translate instruction into one or more fixed length RISC instructions (micro-operations)
- Execute micro-ops on superscalar pipeline
  - micro-ops may be executed out of order
- Commit results of micro-ops to register set in original program flow order
- Outer CISC shell with inner RISC core
- Inner RISC core pipeline at least 20 stages
  - Some micro-ops require multiple execution stages
  - Longer pipeline
  - c.f. five stage pipeline on x86 up to Pentium
Pentium 4 Pipeline

TC Nxt IP = trace cache next instruction pointer
TC Fetch = trace cache fetch
Alloc = allocate

Rename = register renaming
Que = micro-op queuing
Sch = micro-op scheduling
Disp = Dispatch

RF = register file
Ex = execute
Flgs = flags
Br Ck = branch check
(a) Generation of micro-ops

(b) Trace cache next instruction pointer
Pentium 4 Pipeline Operation (2)

(c) Trace cache fetch

(d) Drive
Pentium 4 Pipeline Operation (3)
Pentium 4 Pipeline Operation (4)
Pentium 4 Pipeline Operation (5)
Pentium 4 Pipeline Operation (6)
Reference