Lecture 5: Superscalar Processors

- Definition and motivation
- Superpipeline
- Dependency issues
- Parallel instruction execution

Superscalar Architecture

- Superscalar is a computer designed to improve the performance of the execution of scalar instructions.
  - A scalar is a variable that can hold only one atomic value at a time, e.g., an integer or a real.
  - A scalar architecture processes one data item at a time — the computers we discussed up till now.

- Examples of non-scalar variables:
  - Arrays
  - Matrices
  - Records
Superscalar Architecture (Cont’d)

- In a **superscalar architecture** (SSA), several scalar instructions can be initiated simultaneously and executed independently.
- Pipelining allows also several instructions to be executed at the same time, but they have to be in different pipeline stages at a given moment.
- SSA includes all features of pipelining but, in addition, there can be several instructions executing simultaneously in the same pipeline stage.
- SSA introduces therefore a new level of parallelism, called **instruction-level parallelism**.

Motivation

- Most operations are on scalar quantities (about 80%).
- Speedup these operations will lead to large overall performance improvement.

**How to implement the idea?**

- A SSA processor fetches multiple instructions at a time, and attempts to find nearby instructions that are independent of each other and therefore can be executed in parallel.
- Based on the dependency analysis, the processor may issue and execute instructions in an order that differs from that of the original machine code.
- The processor may eliminate some unnecessary dependencies by the use of additional registers and renaming of register references.
Superpipelining

- Superpipelining is based on dividing the stages of a pipeline into several sub-stages, and thus increasing the number of instructions which are handled by the pipeline at the same time.
- For example, by dividing each stage into two sub-stages, a pipeline can perform at twice the speed in the ideal situation.
  - Many pipeline stages may perform tasks that require less than half a clock cycle.
  - No duplication of hardware is needed for these stages.

Superpipelining (Cont’d)

- For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages/sub-stages.
- Increasing the number of stages/sub-stages over this limit reduces the overall performance.
  - Overhead of data buffering between the stages.
  - Not all stages can be divided into (equal-length) sub-stages.
  - The hazards will be more difficult to resolved.
  - The clock skew problem.
  - More complex hardware.
Superscalar vs. Superpipeline

- Base machine: 4-stage pipeline
  - Instruction fetch
  - Operation decode
  - Operation execution
  - Result write back

- Superpipeline of degree 2
  - A sub-stage often takes half a clock cycle to finish.

- Superscalar of degree 2
  - Two instructions are executed concurrently in each pipeline stage.
  - Duplication of hardware is required by definition.

Superpipelined Superscalar Design

- This is a new trend of architecture design:
  - Pentium Pro (P6): 3-degree superscalar, 12-stage “superpipeline”.
  - PowerPC 620: 4-degree superscalar, 4/6-stage pipeline.
Basic Superscalar Concepts

- SSA allows several instructions to be issued and completed per clock cycle.
- It consists of a number of pipelines that are working in parallel.
- Depending on the number and kind of parallel units available, a certain number of instructions can be executed in parallel.
- In the following example two floating point and two integer operations can be issued and executed simultaneously.
  - Each unit is also pipelined and can execute several operations in different pipeline stages.

An SSA Example
Lecture 5: Superscalar Processors

- Definition and motivation
- Superpipeline
- Dependency issues
- Parallel instruction execution

Parallel Execution Limitation

- The situations which prevent instructions to be executed in parallel by SSA are very similar to those which prevent efficient execution on a pipelined architecture (pipeline hazards):
  - Resource conflicts.
  - Control (procedural) dependency.
  - Data dependencies.

- Their consequences on SSA are more severe than those on simple pipelines, because the potential of parallelism in SSA is greater and, thus, a larger amount of performance will be lost.

- Instruction-level parallelism = the degree in which, on average, the instructions of a program can be executed in parallel.
Resource Conflicts

- Several instructions compete for the same hardware resource at the same time.
  - e.g., two arithmetic instructions need the same floating-point unit for execution.
  - similar to structural hazards in pipeline.
- They can be solved partly by introducing several hardware units for the same functions.
  - e.g., have two floating-point units.
  - the hardware units can also be pipelined to support several operations at the same time.

Procedural Dependency

- The presence of branches creates major problems in assuring the optimal parallelism.
  - cannot execute instructions after a branch in parallel with instructions before a branch.
  - similar to control hazards in pipeline.
- If instructions are of variable length, they cannot be fetched and issued in parallel, since an instruction has to be decoded in order to identify the following one.
  - therefore, superscalar techniques are more efficiently applicable to RISCs, with fixed instruction length and format.
Data Conflicts

- Caused by data dependencies between instructions in the program.
  - similar to date hazards in pipeline.

- To address the problem and to increase the degree of parallel execution, SSA provides a great liberty in the order in which instructions are issued and executed.

- Therefore, data dependencies have to be considered and dealt with much more carefully.

Window of Execution

- Due to data dependencies, only some part of the instructions are potential subjects for parallel execution.

- In order to find instructions to be issued in parallel, the processor has to select from a sufficiently large instruction sequence.
  - There are usually a lot of data dependencies in a short instruction sequence.

- **Window of execution** is defined as the set of instructions that is considered for execution at a certain moment.

- The number of instructions in the window should be as large as possible. However, this is limited by:
  - Capacity to fetch instructions at a high rate.
  - The problem of branches.
  - The cost of hardware needed to analyze data dependencies.
for (i=0; i<last; i++) {
    if (a[i] > a[i+1]) {
        temp = a[i];
        a[i] = a[i+1];
        a[i+1] = temp;
        change++;
    }
}

L2 move r3,r7
load r8,(r3) r8 := a[i]
add r3,r3,#4 r3 := r3+4
load r9,(r3) r9 := a[i+1]
ble r8,r9,L3 jump if r8≤r9

move r3,r7
store r9,(r3) a[i] := r9
add r3,r3,#4 r3 := r3+4
store r8,(r3) a[i+1] := r8
add r5,r5,#1 change++

L3 add r6,r6,#1 i++
add r7,r7,#4 r7 := r7+4
blt r6,r4,L2 jump if r6<r4

Basic Blocks
Window of Execution (cont’d)

- The window of execution can be extended over basic block borders by **branch prediction**.
  - Speculative execution.

- With speculative execution, instructions of the predicted path are entered into the window of execution.
  - Instructions from the predicted path are executed tentatively.
  - If the prediction turns out to be correct the state change produced by these instructions will become permanent and visible (the instructions commit);
  - Otherwise, all effects are removed.

Data Dependencies

- All instructions in the window of execution may begin execution, subject to data dependence and resource constraints.

- Three types of data dependencies can be identified:
  - True data dependency
  - Output dependency
  - Anti-dependency

\[\text{Artificial dependencies}\]
True Data Dependency

- True data dependencies exist when the output of one instruction is required as an input to a subsequent instruction:
  
  MUL R4, R3, R1 \ (R4 := R3 \times R1)
  
  ADD R2, R4, R5 \ (R2 := R4 + R5)

  - can fetch and decode second instruction in parallel with first.
  - can NOT execute second instruction until first is finished.

- They are intrinsic features of the user’s program, and cannot be eliminated by compiler or hardware techniques.

- They have to be detected and handled by hardware.
  - The addition above cannot be executed before the result of the multiplication is available.
  - The simplest solution is to stall the adder until the multiplier has finished.
  - In order to avoid the adder to be idle, the hardware can find other instructions which can be executed by the adder.

True Data Dependency Example

- There are often a lot of true data dependencies in a small region of a program.
  - Increasing the window size can reduce the impact of these dependencies.

- A compiler cannot help to eliminate them!
Output Dependency

- An output dependency exists if two instructions are writing into the same location.
  - If the second instruction writes before the first one, an error occurs:
    
    ```
    MUL R4,R3,R1  (R4 := R3 * R1)
    ... 
    ADD R4,R2,R5  (R4 := R2 + R5)
    ```

    ```
    L2 move r3,r7
    load r8,(r3)
    add r3,r3,#4
    load r9,(r3)
    ble r8,r9,L3
    ```

Anti-dependency

- An anti-dependency exists if an instruction uses a location as an operand while a following one is writing into that location.
  - If the first one is still using the location when the second one writes into it, an error occurs:
    
    ```
    MUL R4,R3,R1  (R4 := R3 * R1)
    ... 
    ADD R3,R2,R5  (R3 := R2 + R5)
    ```

    ```
    L2 move r3,r7
    load r8,(r3)
    add r3,r3,#4
    load r9,(r3)
    ble r8,r9,L3
    ```
Output and Anti-Dependencies

- Output dependencies and anti-dependencies are not intrinsic features of the executed program.
  - They are not real data dependencies but storage conflicts.
  - They are due to the competition of several instructions for the same register.
- They are only the consequence of the manner in which the programmer or the compiler are using registers (or memory locations).
- In the previous examples the conflicts are produced only because:
  - The output dependency: R4 is used by both instructions to store the result (due to, for example, optimization of register usage);
  - The anti-dependency: R3 is used by the second instruction to store the result.

Output and Anti-Dependencies (Cont’d)

- Output dependencies and anti-dependencies can usually be eliminated by using additional registers.
- This technique is called register renaming.

```
MUL R4, R3, R1          (R4 := R3 * R1)
...                       
ADD R9, R2, R5           (R9 := R2 + R5)
```

```
MUL R4, R3, R1          (R4 := R3 * R1)
...                       
ADD R9, R2, R5           (R9 := R2 + R5)
```
Lecture 5: Superscalar Processors

- Definition and motivation
- Superpipeline
- Dependency issues
- Parallel instruction execution
Instruction vs Machine Parallelism

- **Instruction-level parallelism (ILP)** — the average number of instructions in a program that a processor might be able to execute at the same time.
  - Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions.
- **Machine parallelism** of a processor — the ability of the processor to take advantage of the ILP of the program.
  - Determined by the number of instructions that can be fetched and executed at the same time, i.e., the capacity of the hardware.
- To achieve high performance, we need both ILP and machine parallelism.
  - The ideal situation is that we have the same ILP and machine parallelism.

Division and Decoupling

To increase ILP, we should divide the instruction execution into smaller tasks and decouple them. In particular, we have three important activities:

- **Instruction issue** — an instruction is initiated and starts execution.
- **Instruction completion** — an instruction has competed its specified operations.
- **Instruction commit** — the operation results are written back to the register files or cache.
  - The machine state is changed.
SSA Instruction Execution Policies

- Instructions can be executed in an order different from the strictly sequential one, with the requirement that the results must be the same.

- **Execution policies** usually used:
  - In-order issue with in-order completion.
  - In-order issue with out-of-order completion.
  - Out-of-order issue with out-of-order completion.
  - Out-of-order issue with in-order completion.

In-Order Issue with In-Order Completion

- Instructions are issued in exact program order, and completed in the same order (with parallel issue and completion, of course!).
  - An instruction cannot be issued before the previous one has been issued;
  - An instruction cannot be completed before the previous one has been completed.

- To guarantee in-order completion, an instruction will stall when there is a conflict and when a unit requires more than one cycle to execute.

Example:
- Assume a processor that can issue and decode two instructions per cycle, that has three functional units (two single-cycle integer units, and a two-cycle floating-point unit), and that can complete and write back two results per cycle.
- And an instruction sequence with the characteristics given in the next slide.
IOI with IOC Example

- I1 - needs two execute cycles (floating-point)
- I2 -
- I3 -
- I4 - needs the same function unit as I3
- I5 - needs data value produced by I4
- I6 - needs the same function unit as I5

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

In-Order Completion

IOI with IOC Discussion

- The processor detects and handles (by stalling) true data dependencies and resource conflicts.
- The basic idea of SSA is not to rely on compiler-based technique (compatibility consideration).
- SSA allows the hardware alone to detect instructions which can be executed in parallel and to do that accordingly.
- IOI with IOC is not very efficient, but it simplifies the hardware.
In-Order Issue w. Out-of-Order Completion

- With out-of-order completion, a later instruction may complete before a previous one.
  - Address mainly the issue of long-latency operations such as division.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Issue/Decode</th>
<th>Execute</th>
<th>Write/Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13 14</td>
<td>11 12</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15 14</td>
<td>11 13</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

Out-of-Order Completion

Out-of-Order Issue w. Out-of-Order Completion

- With in-order issue, no new instruction can be issued when the processor has detected a conflict, and is stalled until after the conflict has been resolved.
  - The processor is not allowed to look ahead for further instructions, which could be executed in parallel with the current ones.

- Out-of-order issue takes a set of decoded instructions, issues any instruction, in any order, as long as the program execution is correct.
  - Decouple decode pipeline from execution pipeline, by introducing an instruction window.
  - When a functional unit becomes available an instruction can be executed.
  - Since instructions have been decoded, processor can look ahead.
**OOI with OOC Example**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Decode</th>
<th>Ins. Window</th>
<th>Execute</th>
<th>Write/Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>11, 12</td>
<td>11, 12</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13, 14</td>
<td>13, 14</td>
<td>13, 14</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>15, 16</td>
<td>14, 15, 16</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>15, 16</td>
<td>15</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

I1 - needs two cycles
I2 -
I3 -
I4 - conflict with I3
I5 - depending on I4
I6 - conflict with I5

**Speedup Over Simple Scalar**

![Graph showing speedup over simple scalar](image)

- **All dependencies**
  - Without renaming
- **Only true dependencies**
  - With renaming
Summary

- The following techniques are main features for superscalar processors:
  - Several pipelined units which are working in parallel;
  - Out-of-order issue and out-of-order completion;
  - Register renaming.

- All of the above techniques are aimed to enhance performance.

- Experiments have shown:
  - Only adding additional functional units is not very efficient;
  - Out-of-order issue and completion are extremely important, which allows to look ahead for independent instructions;
  - Register renaming can improve performance with more than 30%; in this case performance is limited only by true dependencies.
  - It is important to provide a fetching/decoding capacity so that the window of execution is sufficiently large.