Going Out of Order: Data dependence refresher.

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or  $t5,$t1,$t2
4: add $t3,$t1,$t2
Going Out of Order: Data dependence refresher.

1: add $t1, $s2, $s3
2: sub $t2, $s3, $s4
3: or $t5, $t1, $t2
4: add $t3, $t1, $t2

There is parallelism!!
We can execute 1 & 2 at once and 3 & 4 at once
Going Out of Order: Data dependence refresher.

There is parallelism!!

We can execute 1 & 2 at once and 3 & 4 at once

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t5,$t1,$t2
4: add $t3,$t1,$t2

We can parallelize instructions that do not have a “read-after-write” dependence (RAW)
Data dependences

• In general, if there is no dependence between two instructions, we can execute them in either order or simultaneously.

• But beware:
  • Is there a dependence here?
    1: add $t1,$s2,$s3
    2: sub $t1,$s3,$s4
  • Can we reorder the instructions?
    2: sub $t1,$s3,$s4
    1: add $t1,$s2,$s3
  • Is the result the same?
Data dependences

• In general, if there is no dependence between two instructions, we can execute them in either order or simultaneously.
• But beware:
  • Is there a dependence here?
    1: add $t1,$s2,$s3
    2: sub $t1,$s3,$s4
  • Can we reorder the instructions?
    2: sub $t1,$s3,$s4
    1: add $t1,$s2,$s3
  • Is the result the same?

No! The final value of $t1$ is different
False Dependence #1

• Also called “Write-after-Write” dependences (WAW) occur when two instructions write to the same value
• The dependence is “false” because no data flows between the instructions -- They just produce an output with the same name.
Beware again!

• Is there a dependence here?
  1: add $t1,$s2,$s3
  2: sub $s2,$s3,$s4

• Can we reorder the instructions?
  22: sub $s2,$s3,$s4
  11: add $t1,$s2,$s3

• Is the result the same?
Beware again!

- Is there a dependence here?
  1: add $t1,$s2,$s3
  2: sub $s2,$s3,$s4

- Can we reorder the instructions?
  2: sub $s2,$s3,$s4
  1: add $t1,$s2,$s3

No! The value in $s2$ that I needs will be destroyed

- Is the result the same?
False Dependence #2

- This is a Write-after-Read (WAR) dependence
- Again, it is “false” because no data flows between the instructions
Out-of-Order Execution

• Any sequence of instructions has set of RAW, WAW, and WAR hazards that constrain its execution.

• Can we design a processor that extracts as much parallelism as possible, while still respecting these dependences?
The Central OOO Idea

1. Fetch a bunch of instructions
2. Build the dependence graph
3. Find all instructions with no unmet dependences
4. Execute them.
5. Repeat
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or  $t3,$t1,$t2
4: add $t5,$t1,$t2
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t3,$t1,$t2
4: add $t5,$t1,$t2
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or  $t3,$t1,$t2
4: add $t5,$t1,$t2
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t3,$t1,$t2
4: add $t5,$t1,$t2
5: or $t4,$s1,$s3
6: mul $t2,$t3,$s5
7: sl $t3,$t4,$t2
8: add $t3,$t5,$t1
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or  $t3,$t1,$t2
4: add $t5,$t1,$t2
5: or $t4,$s1,$s3
6: mul $t2,$t3,$s5
7: sl  $t3,$t4,$t2
8: add $t3,$t5,$t1
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t3,$t1,$t2
4: add $t5,$t1,$t2
5: or $t4,$s1,$s3
6: mul $t2,$t3,$s5
7: sl $t3,$t4,$t2
8: add $t3,$t5,$t1
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t3,$t1,$t2
4: add $t5,$t1,$t2
5: or $t4,$s1,$s3
6: mul $t2,$t3,$s5
7: sl $t3,$t4,$t2
8: add $t3,$t5,$t1
Example

1: add $t1,$s2,$s3
2: sub $t2,$s3,$s4
3: or $t3,$t1,$t2
4: add $t5,$t1,$t2
5: or $t4,$s1,$s3
6: mul $t2,$t3,$s5
7: sl $t3,$t4,$t2
8: add $t3,$t5,$t1

8 Instructions in 5 cycles
Simplified OOO Pipeline

- A new “schedule” stage manages the “Instruction Window”
- The window holds the set of instruction the processor examines
  - The fetch and decode fill the window
  - Execute stage drains it
- Typically, OOO pipelines are also “wide” but it is not necessary.
- Impacts
  - More forwarding, More stalls, longer branch resolution
  - Fundamentally more work per instruction.
The Instruction Window

- The “Instruction Window” is the set of instruction the processor examines
  - The fetch and decode fill the window
  - Execute stage drains it
- The larger the window, the more parallelism the processor can find, but...
- Keeping the window filled is a challenge
Keeping the Window Filled

- Keeping the instruction window filled is key!
- Instruction windows are about 32 instructions
  - (size is limited by their complexity, which is considerable)
- Branches are every 4-5 instructions.
- This means that the processor predict 6-8 consecutive branches correctly to keep the window full.
- On a mispredict, you flush the pipeline, which includes the emptying the window.
How Much Parallelism is There?

- Not much, in the presence of WAW and WAR dependences.
- These arise because we must reuse registers, and there are a limited number we can freely reuse.
- How can we get rid of them?
Removing False Dependences

• If WAW and WAR dependences arise because we have too few registers, add more!
• But! We can’t! The architecture only gives us 32.
• Solution:
  • Define a set of internal “physical” register that is as large as the number of instructions that can be “in flight” -- 128 in the latest intel chip.
  • Every instruction in the pipeline gets a registers
  • Maintaining a register mapping table that determines which physical register currently holds the value for the required “architectural” registers.
• This is called “Register Renaming”
Alpha 21264: Renaming

1: Add r3, r2, r3
2: Sub r2, r1, r3
3: Mult r1, r3, r1
4: Add r2, r3, r1
5: Add r2, r1, r3

Register map table

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RAW  →  WAW  →  WAR
Alpha 21264: Renaming

1: Add r3, r2, r3
2: Sub r2, r1, r3
3: Mult r1, r3, r1
4: Add r2, r3, r1
5: Add r2, r1, r3

Register map table

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
</tbody>
</table>

p1 currently holds the value of architectural registers r1
Alpha 21264: Renaming

1: Add r3, r2, r3
2: Sub r2, r1, r3
3: Mult r1, r3, r1
4: Add r2, r3, r1
5: Add r2, r1, r3

p4, p2, p3

<table>
<thead>
<tr>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>1:</td>
<td>p1</td>
<td>p2</td>
</tr>
<tr>
<td>2:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alpha 21264: Renaming

1: Add r3, r2, r3
2: Sub r2, r1, r3
3: Mult r1, r3, r1
4: Add r2, r3, r1
5: Add r2, r1, r3

p4, p2, p3
p5, p1, p4

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>1:</td>
<td>p1</td>
<td>p2</td>
<td>p4</td>
</tr>
<tr>
<td>2:</td>
<td>p1</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>3:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alpha 21264: Renaming

1: Add r3, r2, r3
2: Sub r2, r1, r3
3: Mult r1, r3, r1
4: Add r2, r3, r1
5: Add r2, r1, r3

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>1</td>
<td>p1</td>
<td>p2</td>
<td>p4</td>
</tr>
<tr>
<td>2</td>
<td>p1</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>3</td>
<td>p6</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alpha 21264: Renaming

1: Add r3, r2, r3
2: Sub r2, r1, r3
3: Mult r1, r3, r1
4: Add r2, r3, r1
5: Add r2, r1, r3

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>1</td>
<td>p1</td>
<td>p2</td>
<td>p4</td>
</tr>
<tr>
<td>2</td>
<td>p1</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>3</td>
<td>p6</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>4</td>
<td>p6</td>
<td>p7</td>
<td>p4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alpha 21264: Renaming

1: Add r3, r2, r3  
2: Sub r2, r1, r3  
3: Mult r1, r3, r1  
4: Add r2, r3, r1  
5: Add r2, r1, r3

\[
\begin{array}{ccc}
\text{r1} & \text{r2} & \text{r3} \\
0: & p1 & p2 & p3 \\
1: & p1 & p2 & p4 \\
2: & p1 & p5 & p4 \\
3: & p6 & p5 & p4 \\
4: & p6 & p7 & p4 \\
5: & p6 & p8 & p4 \\
\end{array}
\]
Alpha 21264: Renaming

1: Add r3, r2, r3
2: Sub r2, r1, r3
3: Mult r1, r3, r1
4: Add r2, r3, r1
5: Add r2, r1, r3

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>1</td>
<td>p1</td>
<td>p2</td>
<td>p4</td>
</tr>
<tr>
<td>2</td>
<td>p1</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>3</td>
<td>p6</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>4</td>
<td>p6</td>
<td>p7</td>
<td>p4</td>
</tr>
<tr>
<td>5</td>
<td>p6</td>
<td>p8</td>
<td>p4</td>
</tr>
</tbody>
</table>

RAW  WAW  WAR
New OOO Pipeline

- The register file is larger (to hold the physical registers)
- The pipeline is longer
  - more forwarding
  - Long branch delay
- The payoff had better be significant (and it is)
Modern OOO Processors

- The fastest machines in the world are OOO superscalars
- AMD Barcelona
  - 6-wide issue
  - 106 instructions in flight at once.
- Intel Nehalem
  - 5-way issue to 12 ALUs
  - > 128 instructions in flight
- OOO provides the most benefit for memory operations.
  - Non-dependent instructions can keep executing during cache misses.
  - This is so-called “memory-level parallelism.”
  - It is enormously important. CPU performance is (almost) all about memory performance nowadays (remember the memory wall graphs!)