This Unit: Scheduling (Static + Dynamic)

- Previously:
  - Pipelining
    - Multiple stages
    - Different instructions each stage
  - Superscalar
    - Multiple instructions in each stage
    - “N-wide”
- Now:
  - Compiler (static) scheduling
  - Hardware (dynamic) scheduling

Readings

- H+P
  - None (not happy with explanation of this topic)
- Papers
  - Alpha 21164
    - Due today
  - Discussion
  - Alpha 21264
    - Due next week

Review Example

```
loop:
ld r1 -> r2
add r2 + r3 -> r2
st r2 -> [r1]
addi r1 + 4 -> r1
sub r1, r5 -> r6
inz r6, loop
```

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F D X M W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F D X MW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F D X MW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F D X MW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F D X MW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IPC = 6/7 = 0.86

On a single-issue, 5-stage pipeline,
How many cycles does each loop iteration take?
Assume all cache hits and perfect branch prediction
Review Example

Un-optimized code

What if the pipeline is 2-wide?

What if the pipeline is 2-wide?

Would we get any more performance by going 4-wide?

Would we get any more performance by going 4-wide?

Scheduling Code

Dataflow graph

- Compiler can re-order instructions
  - Eliminate RAW stalls
  - Place independent instructions near each other
- Called static scheduling
- Must be careful to preserve program behavior in all cases!
Optimization

A problem with that

Fixed

Optimized code

Now pieces of each statement are interleaved.

(Aside: why debugging optimized code is confusing)
How fast now?

**loop:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [r1] -&gt; r2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi r1 + 4 -&gt; r1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2 + r3 -&gt; r2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub r1, r5 -&gt; r6</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st r2 -&gt; -4[r1]</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inz r6, loop</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IPC = 6/6 = 1.0

as fast as 2-wide unoptimized

How fast is this on a 1-wide machine?

How fast now?

**loop:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [r1] -&gt; r2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi r1 + 4 -&gt; r1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2 + r3 -&gt; r2</td>
<td>F</td>
<td>D</td>
<td>$d^*$</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub r1, r5 -&gt; r6</td>
<td>F</td>
<td>D</td>
<td>$p^*$</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st r2 -&gt; -4[r1]</td>
<td>F</td>
<td>$p^*$</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inz r6, loop</td>
<td>F</td>
<td>$p^*$</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IPC = 6/4 = 1.5

50% speedup

How fast is this on a 2-wide machine?

What about 4-wide? 8? 16?

Performance vs width

![Graph showing performance vs width](image)

Room to improve?

- Code is much better
  - 2-wide performance greatly improved
  - 4-wide now useful
- Can we do better?
  - With the **scheduling scope** shown: no
  - Larger scheduling scope: yes
**Scheduling Scope**

- Window of instructions we can re-order in
  - Larger => better schedules
  - Compiler: theoretically whole program
    - Not practical for many reasons...
- How?
  - One way: Loop un-rolling
  - Others exist: not the scope of this class

**Loop un-rolling**

- Take 2 (or more) iterations
- Remove extra loop control
  - Getting rid of extra instructions saves time!
  - Note: must compare cycles, not IPC now
- Re-schedule both together
  - Larger scope to schedule from
  - Register names may need changing

---

**Loop unrolling**

- ld [r1] -> r2
- add r2 + r3 -> r2
- st r2 -> [r1]
- addi r1 + 4 -> r1
- sub r1, r5 -> r6
- jnz r6, loop

- ld [r1] -> r2
- add r2 + r3 -> r2
- st r2 -> [r1]
- addi r1 + 4 -> r1
- sub r1, r5 -> r6
- jnz r6, loop

- ld [r1] -> r2
  - addi r1 + 8 -> r1
  - ld -4[r1] -> r7
  - sub r1, r5 -> r6
  - add r2 + r3 -> r2
  - add r7 + r3 -> r7
  - st r2 -> -8[r1]
  - st r7 -> -4[r1]
  - jnz r6, loop

---

**Performance vs width**

- Unoptimized
- Scheduled
- Unrolled 2x
- Unrolled 4x
- Unrolled 8x
Why not unroll 1K times?
- More unrolling => more performance
  - Fewer dynamic instructions
  - Better scheduling
- Downsides / limiting factors?
  - Number of registers
  - More static instructions => $I$ pressure

Limitations of static scheduling
- Assumes cache hits
  - Common case
  - Miss? Different schedule maybe better
- Compiler must be conservative
  - Needs to guarantee correctness
  - Sometimes tough to tell if re-ordering is legal

Re-ordering barrier: branches
```
loop:
jz r1, not_found
ld [r1] -> r2
sub r1, r2 -> r2
jz r2, found
ld 4[r1] -> r1
jmp loop
```

Can these be switched?
```
ld [r1] -> r2
st r3 -> [r4]
```

No: if r1 is null, will cause a fault

Technical term: alias
- Two names for same memory location

Re-ordering barrier: ld/st
```
An example

```c
void f(int * a, int * b, int * c, int N) {
    for (int i = 0; i < N; i++) {
        a[i] = b[i] + c[i];
    }
}
```

Loop unrolled 2x

```c
ld [r1] -> r5
ld [r2] -> r6
add r5 + r6 -> r7
st r7 -> [r3]
```

What can we re-order here?

```c
ld 4[r1] -> r5
ld 4[r2] -> r6
add r5 + r6 -> r7
st r7 -> 4[r3]
// loop control here
```

Aliasing problems

- Must be conservative
  - f(ptr+4, ptr, ptr) not common case
  - but is possible
- If only we could speculate....
  - Allow re-ordering in the common case
  - Get correctness in the rare case
- Anything software can do, hardware can do better..
Out-of-order execution

- Hardware can speculate
  - Load/store ordering
  - Branches
- D$ misses?
  - Compiler: no idea
  - Hardware: knows when they happen
- Out-of-order execution
  - Aka dynamic scheduling

Out-of-order execution

- Execute out of program order
  - Execute oldest ready instruction
    - Ready: all input values available
    - Reduce RAW stalls
- Retain appearance of in-order
  - Maintain correctness

Out-of-order pipeline

Register renaming

- Recall static scheduling:
  xor r1 ^ r2 -> r3
  add r3 + r4 -> r4
  sub r5 - r2 -> r3
  addi r3 + 1 -> r1
- sub/add can be re-ordered
- Must change register of sub
  xor r1 ^ r2 -> r3
  sub r5 - r2 -> r7
  add r3 + r4 -> r4
  addi r7 + 1 -> r1
Register renaming

- Same principle applies to hardware
  - Might re-order anything
  - Create unique names
- Logical registers => physical registers
  - Map table: holds translation
    - Indexed by logical register
    - Holds physical register numbers

Register renaming steps

- Read input numbers from map table
- Allocate new physical register
  - None available? => stall
- Update map table with destination reg

Renaming example

xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p3</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

Map table

<table>
<thead>
<tr>
<th>p6</th>
</tr>
</thead>
<tbody>
<tr>
<td>p7</td>
</tr>
<tr>
<td>p8</td>
</tr>
<tr>
<td>p9</td>
</tr>
<tr>
<td>p10</td>
</tr>
</tbody>
</table>

Free-list

Renaming example

xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

xor p1 ^ p2 ->
add p3 + p4 -> p4
sub p5 - p2 -> p3
addi p3 + 1 -> p1

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p3</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

Map table

<table>
<thead>
<tr>
<th>p6</th>
</tr>
</thead>
<tbody>
<tr>
<td>p7</td>
</tr>
<tr>
<td>p8</td>
</tr>
<tr>
<td>p9</td>
</tr>
<tr>
<td>p10</td>
</tr>
</tbody>
</table>

Free-list
Renaming example

xor r1 ^ r2 → r3  xor p1 ^ p2 → p6
add r3 + r4 → r4  add p6 + p4 → p7
sub r5 - r2 → r3  sub p5 - p2 → p8
addi r3 + 1 → r1

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td></td>
<td>p2</td>
<td>p8</td>
<td>p9</td>
<td>p10</td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td>p6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td>p7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td>p5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Map table     Free-list

Renaming example

xor r1 ^ r2 → r3  xor p1 ^ p2 → p6
add r3 + r4 → r4  add p6 + p4 → p7
sub r5 - r2 → r3  sub p5 - p2 → p8
addi r3 + 1 → r1

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td></td>
<td>p2</td>
<td>p8</td>
<td>p9</td>
<td>p10</td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td>p6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td>p7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td>p5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Map table     Free-list

Renaming example

xor r1 ^ r2 → r3  xor p1 ^ p2 → p6
add r3 + r4 → r4  add p6 + p4 → p7
sub r5 - r2 → r3  sub p5 - p2 → p8
addi r3 + 1 → r1

<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td></td>
<td>p2</td>
<td>p8</td>
<td>p9</td>
<td>p10</td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td>p6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td>p7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td>p5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Map table     Free-list
Renaming example

xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

r1 p1
r2 p2
r3 p8
r4 p7
r5 p5

Map table
Free-list

xor p1 ^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 ->

Copy table
Free-list

Renaming example

xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

r1 p1
r2 p2
r3 p8
r4 p7
r5 p5

Map table
Free-list

xor p1 ^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9

Copy table
Free-list

Out-of-order pipeline

Buffer of instructions

Fetch Decode Rename Dispatch Issue Reg-read Execute Writeback Commit

Have unique register names
Now put into ooo execution structures
Dispatch

- Renamed instructions into ooo structures
  - Re-order buffer (ROB)
  - All instruction until commit
- Issue Queue
  - Un-executed instructions
  - Central piece of scheduling logic
  - Content Addressable Memory (CAM)

RAM vs CAM

- Random Access Memory
  - Read/write specific index
  - Get/set value there
- Content Addressable Memory
  - Search for a value
  - Find matching indices
- One structure can have ports of both types

RAM vs CAM: RAM

Read index 4

<table>
<thead>
<tr>
<th></th>
<th>17</th>
<th>22</th>
<th>47</th>
<th>17</th>
<th>19</th>
<th>12</th>
<th>13</th>
<th>42</th>
</tr>
</thead>
</table>

RAM: read/write specific index

RAM vs CAM: CAM

Find 17

<table>
<thead>
<tr>
<th></th>
<th>17</th>
<th>22</th>
<th>47</th>
<th>17</th>
<th>19</th>
<th>12</th>
<th>13</th>
<th>42</th>
</tr>
</thead>
</table>

CAM: search for value
Issue Queue

- Holds un-executed instructions
- Tracks ready inputs
  - Physical register names + ready bit
  - AND to tell if ready

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ready?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Dispatch Steps

- Allocate IQ slot
  - Full? Stall
- Read ready bits of inputs
  - Table 1-bit per preg
- Clear ready bit of output in table
  - Instruction has not produced value yet
- Write data in IQ slot

Dispatch Example

xor p1 ^ p2 -> p6  
add p6 + p4 -> p7  
sub p5 - p2 -> p8  
addi p8 + 1 -> p9

<table>
<thead>
<tr>
<th>Ready bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1 y</td>
</tr>
<tr>
<td>p2 y</td>
</tr>
<tr>
<td>p3 y</td>
</tr>
<tr>
<td>p4 y</td>
</tr>
<tr>
<td>p5 y</td>
</tr>
<tr>
<td>p6 y</td>
</tr>
<tr>
<td>p7 y</td>
</tr>
<tr>
<td>p8 y</td>
</tr>
<tr>
<td>p9 y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Issue Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insn Inp1 R Inp2 R Dst Age</td>
</tr>
<tr>
<td>-------------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Issue Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insn Inp1 R Inp2 R Dst Age</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>xor p1 y p2 y p6 0</td>
</tr>
<tr>
<td>p5 y</td>
</tr>
<tr>
<td>p6 n</td>
</tr>
<tr>
<td>p7 y</td>
</tr>
<tr>
<td>p8 y</td>
</tr>
<tr>
<td>p9 y</td>
</tr>
</tbody>
</table>
Dispatch Example

xor p1 ^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor</td>
<td>p1</td>
<td>y</td>
<td>p2</td>
<td>y</td>
<td>p6</td>
<td>0</td>
</tr>
<tr>
<td>add</td>
<td>p6</td>
<td>n</td>
<td>p4</td>
<td>y</td>
<td>p7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>p7</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p8</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p9</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ready bits

p1 y
p2 y
p3 y
p4 y
p5 y
p6 y
p7 y
p8 y
p9 y

Issue Queue

Out-of-order pipeline

- Execution (ooo) stages
  - Select ready instructions
    - Send for execution
  - Wakeup dependents

- Issue
- Reg-read
- Execute
- Writeback
**Issue = Select + Wakeup**

- **Select** N oldest, ready instructions

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor</td>
<td>p1</td>
<td>y</td>
<td>p2</td>
<td>y</td>
<td>p6</td>
<td>0</td>
</tr>
<tr>
<td>add</td>
<td>p6</td>
<td>n</td>
<td>p4</td>
<td>y</td>
<td>p7</td>
<td>1</td>
</tr>
<tr>
<td>sub</td>
<td>p5</td>
<td>y</td>
<td>p2</td>
<td>y</td>
<td>p8</td>
<td>2</td>
</tr>
<tr>
<td>addi</td>
<td>p8</td>
<td>n</td>
<td>---</td>
<td>y</td>
<td>p9</td>
<td>3</td>
</tr>
</tbody>
</table>

- N == 1? xor
- N >= 2? xor and sub
- Note: may have resource constraints: i.e. ld/st/fp

**Issue = Select + Wakeup**

- **Wakeup** dependent instructions
  - CAM search for Dst in inputs
  - Set ready
  - Also update ready-bit table for future instructions

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor</td>
<td>p1</td>
<td>y</td>
<td>p2</td>
<td>y</td>
<td>p6</td>
<td>0</td>
</tr>
<tr>
<td>add</td>
<td>p6</td>
<td>y</td>
<td>p4</td>
<td>y</td>
<td>p7</td>
<td>1</td>
</tr>
<tr>
<td>sub</td>
<td>p5</td>
<td>y</td>
<td>p2</td>
<td>y</td>
<td>p8</td>
<td>2</td>
</tr>
<tr>
<td>addi</td>
<td>p8</td>
<td>y</td>
<td>---</td>
<td>y</td>
<td>p9</td>
<td>3</td>
</tr>
</tbody>
</table>

**Register Read**

- Not done at decode
  - Must read physical register (renamed)
  - Must be done when value ready
    - Or gone thru when expecting bypass
- Physical register file may be large
  - Multi-cycle read

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>p6</td>
<td>y</td>
<td>p4</td>
<td>y</td>
<td>p7</td>
<td>1</td>
</tr>
<tr>
<td>addi</td>
<td>p8</td>
<td>y</td>
<td>---</td>
<td>y</td>
<td>p9</td>
<td>3</td>
</tr>
</tbody>
</table>
Renaming review

Everyone rename this instruction:

\[ \text{mul } r4 * r5 \rightarrow r1 \]

<table>
<thead>
<tr>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
<td>p3</td>
<td>p4</td>
<td>p5</td>
</tr>
</tbody>
</table>

Map table

Free-list

Dispatch Review

Everyone dispatch this instruction:

\[ \text{div } p7 \div p6 \rightarrow p1 \]

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>p3</td>
<td>y</td>
<td>p1</td>
<td>y</td>
<td>p2</td>
<td>0</td>
</tr>
<tr>
<td>mul</td>
<td>p2</td>
<td>n</td>
<td>p4</td>
<td>y</td>
<td>p5</td>
<td>1</td>
</tr>
<tr>
<td>div</td>
<td>p1</td>
<td>y</td>
<td>p5</td>
<td>n</td>
<td>p6</td>
<td>2</td>
</tr>
<tr>
<td>xor</td>
<td>p4</td>
<td>y</td>
<td>p1</td>
<td>y</td>
<td>p9</td>
<td>3</td>
</tr>
</tbody>
</table>

Select Review

Select Review

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>p3</td>
<td>y</td>
<td>p1</td>
<td>y</td>
<td>p2</td>
<td>0</td>
</tr>
<tr>
<td>mul</td>
<td>p2</td>
<td>n</td>
<td>p4</td>
<td>y</td>
<td>p5</td>
<td>1</td>
</tr>
<tr>
<td>div</td>
<td>p1</td>
<td>y</td>
<td>p5</td>
<td>n</td>
<td>p6</td>
<td>2</td>
</tr>
<tr>
<td>xor</td>
<td>p4</td>
<td>y</td>
<td>p1</td>
<td>y</td>
<td>p9</td>
<td>3</td>
</tr>
</tbody>
</table>

Determine which instructions are ready.
Which will be issued on a 1-wide machine?
Which will be issued on a 2-wide machine?

Wakeup Review

Wakeup Review

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Inp2</th>
<th>R</th>
<th>Dst</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>p3</td>
<td>y</td>
<td>p1</td>
<td>y</td>
<td>p2</td>
<td>0</td>
</tr>
<tr>
<td>mul</td>
<td>p2</td>
<td>n</td>
<td>p4</td>
<td>y</td>
<td>p5</td>
<td>1</td>
</tr>
<tr>
<td>div</td>
<td>p1</td>
<td>y</td>
<td>p5</td>
<td>n</td>
<td>p6</td>
<td>2</td>
</tr>
<tr>
<td>xor</td>
<td>p4</td>
<td>y</td>
<td>p1</td>
<td>y</td>
<td>p9</td>
<td>3</td>
</tr>
</tbody>
</table>

What information will change if we issue the add?
OOO execution (2-wide)

```
xor  RDY
add   RDY
sub   RDY
addi
```

```
xor p1^ p2 -> p6
sub p5 - p2 -> p8
```

```
add p6 + p4 -> p7
addi p8 + 1 -> p9
```

```
xor 7^ 3 -> p6
sub 6 - 3 -> p8
```

```
add _ + 9 -> p7
```

```
add _ + 1 -> p9
```

```
add _ + 9 -> p7
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```

```
add _ + 1 -> p9
```
OOO execution (2-wide)

Multi-cycle operations

- Multi-cycle ops (ld, fp, mul, etc)
  - Wakeup deferred a few cycles
    - Structural hazard?
  - Cache misses?
    - Speculative wake-up (assume hit)
    - Cancel exec of dependents
    - Re-issue later
- Details: complex, not important

Note similarity to in-order
Re-order Buffer (ROB)

- All instructions in order
- 2 Purposes
  - Misprediction recovery
  - In-order commit
    - Maintain appearance of in-order execution
    - Freeing of physical registers

Renaming revisited

- Overwritten register
  - Freed at commit
  - Restore in map table on recovery
    - Also must be read at rename

Renaming example

```
xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1
```

- **Map table**
  - r1: p1
  - r2: p2
  - r3: p3
  - r4: p4
  - r5: p5

- **Free-list**
  - p6
  - p7
  - p8
  - p9
  - p10

Renaming example

```
xor r1 ^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1
```

- **Map table**
  - r1: p1
  - r2: p2
  - r3: p3
  - r4: p4
  - r5: p5

- **Free-list**
  - p6
  - p7
  - p8
  - p9
  - p10

- p3
Renaming example

xor r1 ^ r2 -> r3  →  xor p1 ^ p2 -> p6  [ p3 ]
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

Map table
\[ \begin{array}{l}
  r1 \quad p1 \\
r2 \quad p2 \\
r3 \quad p6 \\
r4 \quad p4 \\
r5 \quad p5 \\
\end{array} \]

Free-list
\[ \begin{array}{l}
p7 \\
p8 \\
p9 \\
p10 \\
\end{array} \]

Renaming example

xor r1 ^ r2 -> r3  →  xor p1 ^ p2 -> p6  [ p3 ]
add r3 + r4 -> r4  →  add p6 + p4 ->  [ p4 ]
sub r5 - r2 -> r3
addi r3 + 1 -> r1

Map table
\[ \begin{array}{l}
r1 \quad p1 \\
r2 \quad p2 \\
r3 \quad p6 \\
r4 \quad p4 \\
r5 \quad p5 \\
\end{array} \]

Free-list
\[ \begin{array}{l}
p7 \\
p8 \\
p9 \\
p10 \\
\end{array} \]
### Renaming example

- **xor r1 ^ r2 -> r3**
- **add r3 + r4 -> r4**
- **sub r5 - r2 -> r3**
- **addi r3 + 1 -> r1**

#### Map table
- r1
- r2
- r3
- r4
- r5

#### Free-list
- p1
- p2
- p8
- p7
- p5

### Renaming example

- **xor p1 ^ p2 -> p6**
- **add p6 + p4 -> p7**
- **sub p5 - p2 -> p8**
- **addi p8 + 1 -> p9**

#### Map table
- r1
- r2
- r3
- r4
- r5

#### Free-list
- p9
- p10

### ROB

- ROB entry holds all info for recover/commit
  - Logical register names
  - Physical register names
  - Instruction types
- Dispatch: insert at tail
  - Full? Stall
- Commit: remove from head
  - Not completed? Stall
Recovery

- Completely remove wrong path instructions
  - Flush from IQ
  - Remove from ROB
  - Restore map table to before misprediction
  - Free destination registers

Recovery example

```
bnz r1 loop
xor r1 \^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

bnz p1, loop
xor p1 \^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9
```

Map table

```
<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p8</td>
</tr>
<tr>
<td>r4</td>
<td>p7</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>
```

Free-list

```
[ p9 ]
```

Recovery example

```
bnz r1 loop
xor r1 \^ r2 -> r3
add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

bnz p1, loop
xor p1 \^ p2 -> p6
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9
```

Map table

```
<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p8</td>
</tr>
<tr>
<td>r4</td>
<td>p7</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>
```

Free-list

```
[ p9 ]
```
## Recovery example

```
bnz r1, loop
xor r1 ^ r2 -> r3
add r3 + r4 -> r4
```

### Map table

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p6</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p7</th>
</tr>
</thead>
</table>

### Free-list

<table>
<thead>
<tr>
<th>p8</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>p9</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>p10</th>
</tr>
</thead>
</table>

## Recovery example

```
bnz p1, loop
xor p1 ^ p2 -> p6
add p6 + p4 -> p7
```

### Map table

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p6</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p7</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>p8</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>p9</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>p10</th>
</tr>
</thead>
</table>

## Recovery example

```
bnz p1, loop
xor p1 ^ p2 -> p6
```

### Map table

<table>
<thead>
<tr>
<th>r1</th>
<th>p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>p2</td>
</tr>
<tr>
<td>r3</td>
<td>p6</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p6</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>p7</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>p8</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>p9</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>p10</th>
</tr>
</thead>
</table>

## What about stores

- Stores: Write D$, not registers
- Can we rename memory?
- Recover in the cache?
What about stores

• Stores: Write D$, not registers
  • Can we rename memory?
  • Recover in the cache?
  ➢ No (at least not easily)
  • Cache writes unrecoverable
  • Stores: only when certain
    • Commit

Commit

xor r1 ^ r2 -> r3 xor p1 ^ p2 -> p6 [ p3 ]
add r3 + r4 -> r4 add p6 + p4 -> p7 [ p4 ]
sub r5 - r2 -> r3 sub p5 - p2 -> p8 [ p6 ]
addi r3 + 1 -> r1 addi p8 + 1 -> p9 [ p1 ]

• Commit: instruction becomes **architected state**
  • In-order, only when instructions are finished
  • Free overwritten register (why?)

Freeing over-written register

xor r1 ^ r2 -> r3 xor p1 ^ p2 -> p6 [ p3 ]
add r3 + r4 -> r4 add p6 + p4 -> p7 [ p4 ]
sub r5 - r2 -> r3 sub p5 - p2 -> p8 [ p6 ]
addi r3 + 1 -> r1 addi p8 + 1 -> p9 [ p1 ]

• P3 was r3 before xor
• P6 is r3 after xor
  • Anything older than xor should read p3
  • Anything younger than xor should read p6 (until next r3 writing instruction)
• At commit of xor, no older instructions exist
Commit Example

xor r1 ^ r2 -> r3
deti r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1

xor p1 ^ p2 -> p6
addi p + 4 ^ p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9

Map table

Free-list

[ p3 ]
[ p4 ]
[ p6 ]
[ p1 ]
Out of order pipeline diagrams

- Standard style: large and cumbersome
- Change layout slightly
  - Columns = stages (dispatch, issue, etc)
  - Rows = instructions
  - Content of boxes = cycles
- For our purposes: issue/execute = 1 cycle
  - Ignore register read latency, etc
  - Load-use, multiply, divide, and floating-point longer

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld [p1] -&gt; p2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add p2 + p3 -&gt; p4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor p4 ^ p5 -&gt; p6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ld [p7] -&gt; p8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycle 1:
- Dispatch Ld and add

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld [p1] -&gt; p2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>add p2 + p3 -&gt; p4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor p4 ^ p5 -&gt; p6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ld [p7] -&gt; p8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycle 1:
- Dispatch xor and Ld
- 1st Ld issues -- also note WB cycle while you do this
  (Note: don’t issue if WB ports full)
Out of order pipeline diagrams

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld [p1] -&gt; p2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>add p2 + p3 -&gt; p4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor p4 ^ p5 -&gt; p6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld [p7] -&gt; p8</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Cycle 3:
- add and xor are not ready
- 2nd load is- issue it

Cycle 4:
- Nothing

Cycle 5:
- Add can issue

Out of order pipeline diagrams

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld [p1] -&gt; p2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>add p2 + p3 -&gt; p4</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>xor p4 ^ p5 -&gt; p6</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>ld [p7] -&gt; p8</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Cycle 6:
- 1st load can commit (oldest instruction and finished)
- xor can issue

Cycle 7:
- Add can commit
Out of order pipeline diagrams

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld [p1] -&gt; p2</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>add p2 + p3 -&gt; p4</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>xor p4 ^ p5 -&gt; p6</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>ld [p7] -&gt; p8</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Cycle 8:
- Commit xor and ld (2-wide: can do both at once)

Loads and stores

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdiv p1 / p2 -&gt; p3</td>
<td>1</td>
<td>2</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>st p4 -&gt; [p5 ]</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>st p3 -&gt; [p6 ]</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld [p7] -&gt; p8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycle 3:
- Can ld [p7] -> p8 execute?
- Why or why not?

Loads and stores

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdiv p1 / p2 -&gt; p3</td>
<td>1</td>
<td>2</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>st p4 -&gt; [p5 ]</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>st p3 -&gt; [p6 ]</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld [p7] -&gt; p8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Aliasing (again)
- p5 == p7?
- p6 == p7?

Loads and stores

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdiv p1 / p2 -&gt; p3</td>
<td>1</td>
<td>2</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>st p4 -&gt; [p5 ]</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>st p3 -&gt; [p6 ]</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld [p7] -&gt; p8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Suppose p5 == p7 and p6 != 7
Can ld execute now?
Forwarding

- Stores write cache at commit
  - Commit is in-order, delayed by all instructions
- Loads read cache
  - But execution is critical
- Forwarding
  - Allow store -> load communication before store commit
  - Conceptually like bypassing, but very different implementation

Load scheduling

- Store->Load Forwarding:
  - Get value from executed (but not committed) store to load
- Load Scheduling:
  - Determine when load can execute with regard to older stores
- Conservative load scheduling:
  - All older stores have executed
  - Some architectures: split store address / store data
  - Only require known address
  - Advantage: always safe
  - Disadvantage: performance (limits out-of-orderness)

Our example from before

\[
\begin{align*}
\text{ld } [r1] & \rightarrow r5 \\
\text{ld } [r2] & \rightarrow r6 \\
\text{add } r5 + r6 & \rightarrow r7 \\
\text{st } r7 & \rightarrow [r3] \\
\text{ld } 4[r1] & \rightarrow r5 \\
\text{ld } 4[r2] & \rightarrow r6 \\
\text{add } r5 + r6 & \rightarrow r7 \\
\text{st } r7 & \rightarrow 4[r3]
\end{align*}
\]

With conservative load scheduling, what can go out of order?

\[
\begin{align*}
\text{ld } [r1] & \rightarrow r5 \\
\text{ld } [r2] & \rightarrow r6 \\
\text{add } r5 + r6 & \rightarrow r7 \\
\text{st } r7 & \rightarrow [r3] \\
\text{ld } 4[r1] & \rightarrow r5 \\
\text{ld } 4[r2] & \rightarrow r6 \\
\text{add } r5 + r6 & \rightarrow r7 \\
\text{st } r7 & \rightarrow 4[r3]
\end{align*}
\]

// loop control here
Our example from before

<table>
<thead>
<tr>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [p1] -&gt; p5</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld [p2] -&gt; p6</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add p5 + p6 -&gt; p7</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st p7 -&gt; [p3]</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld 4[p1] -&gt; p8</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld 4[p2] -&gt; p9</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add p8 + p9 -&gt; p4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st p4 -&gt; 4[p3]</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Suppose 2 wide, conservative scheduling. May issue 1 load per cycle. Loads take 3 cycles to complete.

Conservative load scheduling: can’t issue ld4[p1]->p8
### Our example from before

<table>
<thead>
<tr>
<th></th>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [p1] -&gt; p5</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>ld [p2] -&gt; p6</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>add p5 + p6 -&gt; p7</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>st p7 -&gt; [p3]</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld 4[p1] -&gt; p8</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld 4[p2] -&gt; p9</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add p8 + p9 -&gt; p4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st p4 -&gt; 4[p3]</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Our example from before

<table>
<thead>
<tr>
<th></th>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [p1] -&gt; p5</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>ld [p2] -&gt; p6</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>add p5 + p6 -&gt; p7</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>st p7 -&gt; [p3]</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld 4[p1] -&gt; p8</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld 4[p2] -&gt; p9</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add p8 + p9 -&gt; p4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>st p4 -&gt; 4[p3]</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Our example from before

<table>
<thead>
<tr>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [p1] -&gt; p5</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>ld [p2] -&gt; p6</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>add p5 + p6 -&gt; p7</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>st p7 -&gt; [p3]</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>ld 4[p1] -&gt; p8</td>
<td>3</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>ld 4[p2] -&gt; p9</td>
<td>3</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>add p8 + p9 -&gt; p4</td>
<td>4</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>st p4 -&gt; 4[p3]</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Our 2-wide ooo processor may as well be 1-wide in-order!
Our example from before

<table>
<thead>
<tr>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld p1 -&gt; p5</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>ld p2 -&gt; p6</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>add p5 + p6 -&gt; p7</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st p7 -&gt; [p3]</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld 4[p1] -&gt; p8</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>ld 4[p2] -&gt; p9</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add p8 + p9 -&gt; p4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st p4 -&gt; 4[p3]</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• It would be nice if we could issue ld 4[p1] -> p8 in c4.
  • Can we speculate and issue it then?

Load Speculation

• Speculation requires two things.....
  • Detection of mis-speculations
    • How can we do this?
  • Recovery from mis-speculations
    • Squash from offending load
    • Saw how to squash from branches: same method

Load Queue

• Detects load ordering violations
• Load execution: Write address into LQ
  • Also note any store forwarded from
• Store execution: Search LQ
  • Younger load with same addr?
  • Didn’t forward from younger store?

Store Queue + Load Queue

• Store Queue: handles forwarding
  • Written by stores
  • Searched by loads
• Load Queue: detects ordering violations
  • Written by loads
  • Searched by stores
• Both together
  • Allows aggressive load scheduling
  • Stores don’t constrain load execution
Our example from before

<table>
<thead>
<tr>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [p1] -&gt; p5</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>ld [p2] -&gt; p6</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>add p5 + p6 -&gt; p7</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st p7 -&gt; [p3]</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld 4[p1] -&gt; p8</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>ld 4[p2] -&gt; p9</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add p8 + p9 -&gt; p4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st p4 -&gt; 4[p3]</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Aggressive load scheduling?
  - Issue ld 4[p1] -> p8 in cycle 4

Saves 4 cycles over conservative
Actually uses ooo-ness

Our example from before

<table>
<thead>
<tr>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [p1] -&gt; p5</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>ld [p2] -&gt; p6</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>add p5 + p6 -&gt; p7</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>st p7 -&gt; [p3]</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>ld 4[p1] -&gt; p8</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>ld 4[p2] -&gt; p9</td>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>add p8 + p9 -&gt; p4</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>st p4 -&gt; 4[p3]</td>
<td>4</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Our example from before

<table>
<thead>
<tr>
<th>Disp</th>
<th>Issue</th>
<th>WB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [p1] -&gt; p5</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>ld [p2] -&gt; p6</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>add p5 + p6 -&gt; p7</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>st p7 -&gt; [p3]</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>ld 4[p1] -&gt; p8</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>ld 4[p2] -&gt; p9</td>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>add p8 + p9 -&gt; p4</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>st p4 -&gt; 4[p3]</td>
<td>4</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Aggressive Load scheduling

- Allows loads to issue before older stores
  - Increases out-of-orderness
    + When no conflict, increases performance
    - Conflict => squash => worse performance than waiting
  - Some loads might forward from stores
    - Always aggressive will squash a lot

- Can we have our cake AND eat it too?
Predictive Load scheduling

- Predict which loads must wait for stores
- Fool me once, shame on you-- fool me twice?
  - Loads default to aggressive
  - Keep table of load PCs that have been caused squashes
    - Schedule these conservatively
  - Simple predictor
    - Makes “bad” loads wait for all older stores is not so great
- More complex predictors used in practice
  - Predict which stores loads should wait for

Out of Order: Window Size

- Scheduling scope = ooo window size
  - Larger = better
  - Constrained by physical registers
    - ROB roughly limited by $\#preg = ROB$ size + $\#logical$ registers
    - Big register file = hard/slow
  - Constrained by issue queue
    - Limits number of un-executed instructions
    - CAM = can’t make big (power + area)
  - Constrained by load + store queues
    - Limit number of loads/stores
    - CAMs

OOO scalability research

- Checkpoint Processing and Recovery [Akkary ‘03]
  - Attacks scaling of register file
  - Take checkpoints at rename
  - Only recover to those
  - Free Pregs aggressively
- Continual Flow Pipelines [Srinivasan ‘04]
  - Attacks scaling of Issue Queue
  - Put L2 misses and dependents out of IQ
  - Place back in when L2 miss returns
- Store Vulnerability Window [Roth ‘05] + Store Queue Index Prediction [Sha ‘05]
  - Scalable (non-associative) load/store queues
  - Predict store queue index for forwarding
  - Filtered load re-execution prior to commit

Out of Order: Benefits

- Allows speculative re-ordering
  - Loads / stores
  - Branch prediction
- Schedule can change due to cache misses
  - Different schedule optimal from on cache hit
- Done by hardware
  - Compiler may want different schedule for different hw configs
  - Hardware has only its own configuration to deal with
Dependence types

- RAW (Read After Write) = “true dependence”
  
  ```
  ld [r1] -> r2
  add r2 + r3 -> r4
  ```

- WAW (Write After Write) = “output dependence”
  
  ```
  ld [r1] -> r2
  add r1 + r3 -> r2
  ```

- WAR (Write After Read) = “anti-dependence”
  
  ```
  ld [r1] -> r2
  add r3 + r4 -> r1
  ```

Memory dependences

- RAW (Read After Write)
  
  ```
  st r1 -> [r2]
  ld [r2] -> r4
  ```

- WAW (Write After Write)
  
  ```
  st r1 -> [r2]
  st r3 -> [r2]
  ```

- WAR (Write After Read)
  
  ```
  ld [r1] -> r2
  st r3 -> [r1]
  ```

More on dependences

- RAW
  
  - When more than one applies, RAW dominates:
    
    ```
    add r1 + r2 -> r3
    addi r3 + 1 -> r3
    ```
  
  - Must be respected: no trick to avoid

- WAR/WAW on registers
  
  - Two things happen to use same name
  
  - Can be eliminated by renaming

- WAR/WAW on memory
  
  - Can’t rename memory
  
  - Need to use other tricks (later this lecture)

Out of Order: Top 5 things to know

- Register renaming
  
  - How to perform is and how to recover it

- Commit
  
  - Precise state (ROB)
  
  - How/when registers are freed

- Issue/Select
  
  - Wakeup: CAM
  
  - Choose N oldest ready instructions

- Stores
  
  - Write at commit
  
  - Forward to loads via LQ

- Loads
  
  - Conservative/aggressive/predictive scheduling
  
  - Violation detection