This Unit: Static & Dynamic Scheduling

- Pipelining and superscalar review
  - Code scheduling
    - To reduce pipeline stalls
    - To increase ILP (insn level parallelism)
- Two approaches
  - Static scheduling by the compiler
  - Dynamic scheduling by the hardware

Readings
- H+P
- TBD
- Papers
  - Alpha 21164
    - Due today
    - Discussion
  - Alpha 21264
    - Due next week

Pipelining Review
- Increases clock frequency by staging instruction execution
- “Scalar” pipelines have a best-case CPI of 1
- Challenges:
  - Data and control dependencies further worsen CPI
  - Data: With full bypassing, load-to-use stalls
  - Control: use branch prediction to mitigate penalty
- Big win, done by all processors today
- How many stages (depth)?
  - Five stages is pretty good minimum
  - Intel Pentium II/III: 12 stages
  - Intel Pentium 4: 22+ stages
  - Intel Core 2: 14 stages
Pipeline Diagram

<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>
</tr>
</thead>
<tbody>
<tr>
<td>add $3, $2, $1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $4, 4($3)</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $6, $4, 1</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>
</tr>
<tr>
<td>sub $8, $3, $1</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>
</tr>
</tbody>
</table>

- Use compiler scheduling to reduce load-use stall frequency
  - Like software interlocks, but for performance not correctness

Superscalar Pipeline Review

- Execute two or more instruction per cycle
- Challenges:
  - wide fetch (branch prediction harder, misprediction more costly)
  - wide decode (stall logic)
  - wide execute (more ALUs)
  - wide bypassing (more possibly bypassing paths)
  - Finding enough independent instructions (and fill delay slots)
- How many instructions per cycle max (width)?
  - Really simple, low-power cores are still single-issue (most ARMs)
  - Even low-power cores a dual-issue (ARM A8, Intel Atom)
  - Most desktop/laptop chips three-issue or four-issue (Core i7)
  - A few 5 or 6-issue chips have been built (IBM Power4, Itanium II)

Superscalar Pipeline Diagrams - Ideal

Superscalar Pipeline Diagrams - Realistic

2-way superscalar

Scalar
Code Scheduling

- Scheduling: act of finding independent instructions
  - "Static" done at compile time by the compiler (software)
  - "Dynamic" done at runtime by the processor (hardware)

- Why schedule code?
  - Scalar pipelines: fill in load-to-use delay slots to improve CPI
  - Superscalar: place independent instructions together
    - As above, load-to-use delay slots
    - Allow multiple-issue decode logic to let them execute at the same time

Compiler Scheduling

- Compiler can schedule (move) instructions to reduce stalls
  - Basic pipeline scheduling: eliminate back-to-back load-use pairs
  - Example code sequence: \( a = b + c; \quad d = f - e; \)
    - \( sp \) stack pointer, \( sp + 0 \) is "a", \( sp + 4 \) is "b", etc...

    | Before                          | After                          |
    |--------------------------------|--------------------------------|
    | 1d r2,4(sp)                    | 1d r2,4(sp)                    |
    | 1d r3,8(sp)                    | 1d r3,8(sp)                    |
    | add r3, r2, r1 //stall         | add r3, r2, r1 //stall         |
    | st r1, 0(sp)                   | st r1, 0(sp)                   |

- One way to create larger scheduling scopes?
  - Loop unrolling

Compiler Scheduling Requires

- Large scheduling scope
  - Independent instruction to put between load-use pairs
    + Original example: large scope, two independent computations
    - This example: small scope, one computation

    | Before                          | After                          |
    |--------------------------------|--------------------------------|
    | ld r2,4(sp)                    | ld r2,4(sp)                    |
    | ld r3,8(sp)                    | ld r3,8(sp)                    |
    | add r3, r2, r1 //stall         | add r3, r2, r1 //stall         |
    | st r1, 0(sp)                   | st r1, 0(sp)                   |

- Enough registers
  - To hold additional "live" values
    - Example code contains 7 different values (including \( sp \))
  - Before: max 3 values live at any time \( \rightarrow \) 3 registers enough
  - After: max 4 values live \( \rightarrow \) 3 registers not enough

    | Original                          | Wrong!                          |
    |-----------------------------------|---------------------------------|
    | ld r2,4(sp)                       | ld r2,4(sp)                     |
    | ld r1,8(sp)                       | ld r1,8(sp)                     |
    | add r1, r2, r1 //stall            | add r1, r2, r1 //wrong r2       |
    | st r1, 0(sp)                      | st r1, 0(sp)                    |
    | ld r2,16(sp)                      | ld r2,16(sp)                    |
    | ld r1,20(sp)                      | ld r1,20(sp)                    |
    | sub r2, r1, r1 //stall            | sub r2, r1, r1 //wrong r1       |
    | st r1, 12(sp)                     | st r1, 12(sp)                   |
Compiler Scheduling Requires

- **Alias analysis**
  - Ability to tell whether load/store reference same memory locations
  - Effectively, whether load/store can be rearranged
  - Example code: easy, all loads/stores use same base register (sp)
  - New example: can compiler tell that r8 != sp?
  - Must be **conservative**

Before | Wrong(?)
--- | ---
ld r2,4(sp) | ld r2,4(sp)
ld r3,8(sp) | ld r3,8(sp)
add r3,r2,r1 //stall | ld r5,0(r8) //does r8==sp?
st r1,0(sp) | add r3,r2,r1
ld r5,0(r8) | ld r6,4(r8) //does r8+4==sp?
ld r6,4(r8) | st r1,0(sp)
sub r5,r6,r4 //stall | sub r5,r6,r4
st r4,8(r8) | st r4,8(r8)

New Metric: Utilization

- **Utilization**: actual performance / peak performance
  - Important metric for performance/cost
  - No point to paying for hardware you will rarely use
  - Adding hardware usually improves performance & reduces utilization
  - Additional hardware can only be exploited some of the time
  - Diminishing marginal returns
  - Compiler can help make better use of existing hardware
  - Important for superscalar

**SAXPY** (Single-precision A X Plus Y)
- Linear algebra routine (used in solving systems of equations)
- Part of early "Livermore Loops" benchmark suite
- Uses floating point values in "F" registers
- Uses floating point version of instructions (ldf, addf, mulf, stf, etc.)

```plaintext
for (i=0; i<N; i++)
  Z[i]=A*X[i]+Y[i];
```

for loop:
0: ldf X(r1)⇒f1 // loop
1: mulf f0,f1⇒f2 // A in f0
2: ldf Y(r1)⇒f3 // X,Y,Z are constant addresses
3: addf f2,f3⇒f4
4: stf f4⇒Z(r1)
5: addi r1,4⇒r1 // i in r1
6: blt r1,r2,0 // N*4 in r2

**SAXPY Performance and Utilization**

<table>
<thead>
<tr>
<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>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

- **Scalar pipeline**
  - Full bypassing, 5-cycle E*, 2-cycle E+, branches predicted taken
  - Single iteration (7 insns) latency: 16–5 = 11 cycles
  - **Performance**: 7 insns / 11 cycles = 0.64 IPC
  - **Utilization**: 0.64 actual IPC / 1 peak IPC = 64%
SAXPY Performance and Utilization

- 2-way superscalar pipeline
  - Any two insns per cycle + split integer and floating point pipelines
  - Performance: 7 insns / 10 cycles = 0.70 IPC
  - Utilization: 0.70 actual IPC / 2 peak IPC = 35%
  - More hazards → more stalls
  - Each stall is more expensive

Static (Compiler) Instruction Scheduling

- Idea: place independent insns between slow ops and uses
  - Otherwise, pipeline stalls while waiting for RAW hazards to resolve
  - Have already seen pipeline scheduling

- To schedule well you need ... independent insns
- Scheduling scope: code region we are scheduling
  - The bigger the better (more independent insns to choose from)
  - Once scope is defined, schedule is pretty obvious
  - Trick is creating a large scope (must schedule across branches)

- Scope enlarging techniques
  - Loop unrolling
  - Others: superblocks, hyperblocks, trace scheduling, etc.

Loop Unrolling SAXPY

- Goal: separate dependent insns from one another
- SAXPY problem: not enough flexibility within one iteration
  - Longest chain of insns is 9 cycles
    - Load (1)
    - Forward to multiply (5)
    - Forward to add (2)
    - Forward to store (1)
      - Can't hide a 9-cycle chain using only 7 insns
    - But how about two 9-cycle chains using 14 insns?
- Loop unrolling: schedule two or more iterations together
  - Fuse iterations
  - Schedule to reduce stalls
  - Schedule introduces ordering problems, rename registers to fix

Unrolling SAXPY I: Fuse Iterations

- Combine two (in general K) iterations of loop
  - Fuse loop control: induction variable (i) increment + branch
  - Adjust (implicit) induction uses: constants → constants + 4

  ldf X(r1), f1
  mulf f0, f1, f2
  ldf Y(r1), f3
  addf f2, f3, f4
  stf f4, Z(r1)
  addi r1, 4, r1
  blt r1, r2, 0

  ldf X(r1), f1
  mulf f0, f1, f2
  ldf Y(r1), f3
  addf f2, f3, f4
  stf f4, Z(r1)
  addi r1, 4, r1
  blt r1, r2, 0

  ldf X(r1), f1
  mulf f0, f1, f2
  ldf Y(r1), f3
  addf f2, f3, f4
  stf f4, Z(r1)
  addi r1, 4, r1
  blt r1, r2, 0
### Unrolling SAXPY II: Pipeline Schedule

- Pipeline schedule to reduce stalls
  - Have already seen this: pipeline scheduling

```plaintext
ldf X(r1),f1
mul f0,f1,f2
ld Y(r1),f3
add f2,f3,f4
st f4,Z(r1)
ld X+4(r1),f1
mul f0,f1,f2
ld Y+4(r1),f3
add f2,f3,f4
st f4,Z+4(r1)
addi r1,8,r1
blt r1,r2,0
```

### Unrolling SAXPY III: “Rename” Registers

- Pipeline scheduling causes reordering violations
  - Rename registers to correct

```plaintext
ldf X(r1),f1
mul f0,f1,f2
ld Y(r1),f3
mul f0,f1,f2
ld Y+4(r1),f3
add f2,f3,f4
add f2,f3,f4
st f4,Z(r1)
addi r1,8,r1
blt r1,r2,0
```

### Unrolled SAXPY Performance/Utilization

\[
\begin{array}{cccccccccccccccc}
\hline
\text{liff X(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 & \text{liff X+4(r1)},f1 \end{array}
\]

+ Performance: 12 insn / 13 cycles = 0.92 IPC
+ Utilization: 0.92 actual IPC / 1 peak IPC = 92%
+ Speedup: \(2 * 11 \text{ cycles}) / 13 \text{ cycles} = 1.69

### Loop Unrolling Shortcomings

- Static code growth → more I$ misses (limits degree of unrolling)
- Needs more registers to hold values (ISA limits this)
- Doesn’t handle non-loops
- Doesn’t handle inter-iteration dependences

```plaintext
for (i0;i<N;i++)
X[i]=A*X[i-1];
```

- Two mulf’s are not parallel
- Other (more advanced) techniques help
Another Limitation: Branches

```
loop:
  jz r1, not_found
  ld [r1] -> r2
  sub r1, r2 -> r2
  jz r2, found
  ld [r1+4] -> r1
  jmp loop
```

Aside: what does this code do?

Legal to move load up past branch?

Summary: Static Scheduling Limitations

- Limited number of registers (set by ISA)
- Scheduling scope
  - Example: can’t generally move memory operations past branches
- Inexact memory aliasing information
  - Often prevents reordering of loads above stores
- Caches misses (or any runtime event) confound scheduling
  - How can the compiler know which loads will miss vs hit?
  - Can impact the compiler’s scheduling decisions

Can Hardware Overcome These Limits?

- **Dynamically-scheduled processors**
  - Also called “out-of-order” processors
  - Hardware re-schedules insns...
  - ...within a sliding window of VonNeumann insns
  - As with pipelining and superscalar, ISA unchanged
    - Same hardware/software interface, appearance of in-order
- Increases scheduling scope
  - Does loop unrolling transparently
  - Uses branch prediction to “unroll” branches
- Examples:
  - Pentium Pro/II/III (3-wide), Core 2 (4-wide),
    Alpha 21264 (4-wide), MIPS R10000 (4-wide), Power5 (5-wide)
- Basic overview of approach

The Problem With In-Order Pipelines

```
addf f0, f1 | f2  
mulf f2, f3 | f2  
subf f0, f1 | f4  
```

![Pipeline Schedule]

- **What’s happening in cycle 4?**
  - **mulf** stalls due to **data dependence**
    - OK, this is a fundamental problem
  - **subf** stalls due to **pipeline hazard**
    - Why? **subf** can’t proceed into D because **addf** is there
      - That is the only reason, and it isn’t a fundamental one
    - Maintaining in-order writes to register file
- **Why can’t **subf** go into D in cycle 4 and E+ in cycle 5?**
Out-of-order Pipeline

**Step #1: Register Renaming**
- To eliminate register conflicts/hazards
- “Architected” vs “Physical” registers – level of indirection
  - Names: r1, r2, r3
  - Locations: p1, p2, p3, p4, p5, p6, p7
  - Original mapping: r1 → p1, r2 → p2, r3 → p3, p4→p7 are “available”

<table>
<thead>
<tr>
<th>MapTable</th>
<th>FreeList</th>
<th>Original insns</th>
<th>Renamed insns</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>r2</td>
<td>r3</td>
<td>p1, p2, p3</td>
</tr>
<tr>
<td>p4</td>
<td>p2</td>
<td>p5</td>
<td>p2, p3</td>
</tr>
<tr>
<td>p4</td>
<td>p2</td>
<td>p5</td>
<td>p4, p5</td>
</tr>
<tr>
<td>p4</td>
<td>p2</td>
<td>p6</td>
<td>p4, p6</td>
</tr>
</tbody>
</table>

- Renaming – conceptually write each register once
  + Removes false dependences
  + Leaves true dependences intact!
- When to reuse a physical register? After overwriting insn done

**Step #2: Dynamic Scheduling**
- Instructions fetch/decoded/renamed into Instruction Buffer
  - Also called "instruction window" or "instruction scheduler"
- Instructions (conceptually) check ready bits every cycle
  - Execute when ready

Code Example
- Code:
  ```plaintext
  add r2, r3, r1
  sub r2, r1, r3
  mul r2, r3, r3
  div r1, 4, r1
  ```
- “True” (real) & “False” (artificial) dependencies
- Divide insn independent of subtract and multiply insns
  - Can execute in parallel with subtract
- Many registers re-used
  - Just as in static scheduling, the register names get in the way
  - How does the hardware get around this?
- Approach: (step #1) rename registers, (step #2) schedule
Register Renaming Algorithm

- **Data structures:**
  - `maptable[architectural_reg] → physical_reg`
  - Free list: get/put free register

- **Algorithm:** at decode for each instruction:
  ```plaintext
  insn.phys_input1 = maptable[insn.arch_input1]
  insn.phys_input2 = maptable[insn.arch_input2]
  insn.phys_to_free = maptable[arch_output]
  new_reg = get_free_phys_reg()
  insn.phys_output = new_reg
  maptable[arch_output] = new_reg
  ```

- **At “commit”**
  - Once all older instructions have committed, free register
    ```plaintext
    put_free_phys_reg(insn.phys_to_free)
    ```

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>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

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

Renaming example

oxor p1 ^ p2 →
add r3 + r4 → r4
sub r5 - r2 → r3
addi r3 + 1 → 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

| p6 | p7 | p8 | p9 | p10 |
Free-list
Renaming example

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

xor p1 ^ p2 -> p6
add p6 + p4 -> p7

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

xor p1 ^ p2 -> p6
add p6 + p4 -> p7

r1 p1
r2 p2
r3 p6
r4 p7
r5 p5
p8 p9
p10

Map table Free-list

r1 p1
r2 p2
r3 p6
r4 p7
r5 p5
p8 p9
p10

Map table Free-list

CIS 501 (Martin/Hilton/Roth): Scheduling

41
Renaming example

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

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

Map table
r1 p1
r2 p2
r3 p8
r4 p7
r5 p5

Free-list
p9
p10

Renaming example

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

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

Map table
r1 p1
r2 p2
r3 p8
r4 p7
r5 p5

Free-list
p9
p10

Out-of-order Pipeline

Fetch Decode Rename Dispatch
Buffer of instructions

Issue Reg-read Execute Writeback Commit

Have unique register names
Now put into ooo execution structures
DYNAMIC SCHEDULING

RAM vs CAM

- Random Access Memory
  - Read/write specific index
  - Get/set value there
- Content Addressable Memory
  - Search for a value (send value to all entries)
  - Find matching indices (use comparator at each entry)
  - Output: one bit per entry (multiple match)
- One structure can have ports of both types

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: RAM

Read index 4

17
22
47
17
19
12
13
42

RAM: read/write specific index

19
RAM vs CAM: CAM

Find value “17”

<table>
<thead>
<tr>
<th>Index 0</th>
<th>Index 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

CAM: search for value

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>
</table>

Ready?

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

Issue Queue
### Dispatch Example

**Insn**

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

**Ready bits**

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

**Issue Queue**

<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>
</tbody>
</table>

### Dispatch Example

**Insn**

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

**Ready bits**

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

**Issue Queue**

<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>
</tbody>
</table>

### Dispatch Example

**Insn**

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

**Ready bits**

- p1 y
- p2 y
- p3 y
- p4 y
- p5 y
- p6 n
- p7 n
- p8 n

**Issue Queue**

<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>
</tbody>
</table>

<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Insn</th>
<th>Inp1</th>
<th>R</th>
<th>Dst</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>p8</td>
<td>n</td>
<td>y</td>
<td>p9</td>
</tr>
</tbody>
</table>
Out-of-order pipeline

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

Dynamic Scheduling/Issue Algorithm

- Data structures:
  - Ready table[phys_reg] → yes/no (part of issue queue)

- Algorithm at “schedule” stage (prior to read registers):
  - foreach instruction:
    - if table[insn.phys_input1] == ready &&
      table[insn.phys_input2] == ready then
      insn as “ready”
    - select the oldest “ready” instruction
      table[insn.phys_output] = ready

Issue = Select + Wakeup

- **Select** N oldest, ready instructions
  - “xor” is the oldest ready instruction below
  - “xor” and “sub” are the two oldest ready instructions below
- Note: may have resource constraints: i.e. load/store/fp

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

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

- **Select/Wakeup** one cycle
- Dependents go back to back
  - Next cycle: add/addi are 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>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>

**Register Read**

- When do instructions read the register file?
- Option #1: after select, right before execute
  - (Not done at decode)
  - Read physical register (renamed)
  - Or get value via bypassing (based on physical register name)
  - This is Pentium 4, MIPS R10k, Alpha 21264 style
- Physical register file may be large
  - Multi-cycle read
- Option #2: as part of issue, keep values in Issue Queue
  - Pentium Pro, Core 2, Core i7

**Renaming review**

Everyone rename this instruction:

```
mul r4 * r5 -> 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p7</td>
<td></td>
</tr>
<tr>
<td>p8</td>
<td></td>
</tr>
<tr>
<td>p9</td>
<td></td>
</tr>
<tr>
<td>p10</td>
<td></td>
</tr>
</tbody>
</table>

Free-list

**Dispatch Review**

Everyone dispatch this instruction:

```
div p7 / p6 -> 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ready bits

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

<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)

CIS 501 (Martin/Hilton/Roth): Scheduling 69

CIS 501 (Martin/Hilton/Roth): Scheduling 70

OOO execution (2-wide)
### OOO execution (2-wide)

<table>
<thead>
<tr>
<th>p1</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td>3</td>
</tr>
<tr>
<td>p3</td>
<td>4</td>
</tr>
<tr>
<td>p4</td>
<td>9</td>
</tr>
<tr>
<td>p5</td>
<td>6</td>
</tr>
<tr>
<td>p6</td>
<td>4</td>
</tr>
<tr>
<td>p7</td>
<td>13</td>
</tr>
<tr>
<td>p8</td>
<td>3</td>
</tr>
<tr>
<td>p9</td>
<td>4</td>
</tr>
</tbody>
</table>

Note similarity to in-order

### Multi-cycle operations

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

### Re-order Buffer (ROB)

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

### RENAMING REVISITED
Renaming revisited

- Overwritten register
  - Freed at commit
  - Restore in map table on recovery
  - Branch mis-prediction 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
```

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

```
xor p1 ^ p2 -> p6
```

```
xor r1 ^ 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>p3</td>
</tr>
<tr>
<td>r4</td>
<td>p4</td>
</tr>
<tr>
<td>r5</td>
<td>p5</td>
</tr>
</tbody>
</table>

Free-list

<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>
<tr>
<td>p6</td>
<td></td>
</tr>
<tr>
<td>p7</td>
<td></td>
</tr>
<tr>
<td>p8</td>
<td></td>
</tr>
<tr>
<td>p9</td>
<td></td>
</tr>
<tr>
<td>p10</td>
<td></td>
</tr>
</tbody>
</table>
Renaming example

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

- **xor** p1 ^ p2 -> p6
- **add** p6 + p4 ->

---

**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>

**Free-list**

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

**CIS 501 (Martin/Hilton/Roth): Scheduling** 85
Renaming example

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

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

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

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

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

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

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

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

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

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

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
Recovery example

bnz r1 loop
xor r1 ^ r2 -> r3
bnz p1, loop
xor p1 ^ p2 -> p6
r1 p1
r2 p2
r3 p3
r4 p4
r5 p5
p6
p7
p8
p9
p10

Map table
Free-list

What about stores

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

- 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

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

Freeing over-written register

- P3 was \( r_3 \) before xor
- P6 is \( r_3 \) after xor
  - Anything older than xor should read p3
  - Anything younger than xor should p6 (until next \( r_3 \) writing instruction)
  - At commit of xor, no older instructions exist

Commit Example

Freeing over-written register

- P3 was \( r_3 \) before xor
- P6 is \( r_3 \) after xor
  - Anything older than xor should read p3
  - Anything younger than xor should p6 (until next \( r_3 \) writing instruction)
  - At commit of xor, no older instructions exist

Commit Example
Commit Example

add r3 + r4 -> r4
sub r5 - r2 -> r3
addi r3 + 1 -> r1
add p6 + p4 -> p7
sub p5 - p2 -> p8
addi p8 + 1 -> p9

Map table
Free-list

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/exec = 1 cycle
  • Ignore preg read latency, etc
  • Load-use, mul, div, and FP longer
### 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add p2 + p3 -&gt; p4</td>
<td>1</td>
<td>2</td>
<td>5</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>

2-wide
Infinite ROB, IQ, Pregs
Loads: 3 cycles

### 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></td>
</tr>
<tr>
<td>add p2 + p3 -&gt; p4</td>
<td></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 ld and add

Cycle 3:
- add and xor are not ready
- 2nd load is- issue it
### 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></td>
</tr>
<tr>
<td>ld [p7] -&gt; p8</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**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>7</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

### 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></td>
</tr>
<tr>
<td>ld [p7] -&gt; p8</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**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></td>
</tr>
<tr>
<td>ld [p7] -&gt; p8</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Cycle 8:**
- Commit xor and ld (2-wide: can do both at once)
**Dynamically Scheduling Memory Ops**

- Compilers must schedule memory ops conservatively
- Options for hardware:
  - Don’t execute any load until all prior stores execute (conservative)
  - Execute loads as soon as possible, detect violations (aggressive)
    - When a store executes, it checks if any later loads executed too early (to same address). If so, flush pipeline
  - Learn violations over time, selectively reorder (predictive)

```plaintext
Before
ld r2,4(sp)
ld r3,8(sp)
add r3,r2,r1  //stall
st r1,0(sp)
ld r5,0(r8)
ld r6,4(r8)
sub r5,r6,r4  //stall
st r4,8(r8)
```

Wrong(?)

```plaintext
ld r2,4(sp)
ld r3,8(sp)
ld r5,0(r8)  //does r8==sp?
add r3,r2,r1
ld r5,0(r8)  //does r8+4==sp?
st r1,0(sp)
ld r6,4(r8)
sub r5,r6,r4
st r4,8(r8)
```

**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?

**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>fdv 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></td>
<td>3</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 != p7
Can ld execute now?

Memory Forwarding

- Stores write cache at commit
  - Commit is in-order, delayed by all instructions
  - Allows stores to be “undone” on branch mis-predictions, etc.

- Loads read cache
  - Early execution of loads is critical

- Forwarding
  - Allow store -> load communication before store commit
  - Conceptually like reg. bypassing, but different implementation
    - Why? Addresses unknown until execute

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

ld [r1] -> r5
ld [r2] -> r6
add r5 + r6 -> r7    With conservative load scheduling, what can go out of order?
st r7 -> [r3]
ld 4[r1] -> r5
ld 4[r2] -> r6
add r5 + r6 -> r7
st r7 -> 4[r3]
// loop control here

Disp   Issue    WB    Commit
ld [p1] -> p5  1    2   5  Commit
ld [p2] -> p6  1
add p5 + p6 -> p7
st p7 -> [p3]
ld 4[p1] -> p8
ld 4[p2] -> p9
add p8 + p9 -> p4
st p4 -> 4[p3]

Suppose 2 wide, conservative scheduling. May issue 1 load per cycle. Loads take 3 cycles to complete.
### 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></td>
</tr>
<tr>
<td>ld 4[p2] -&gt; p9</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>add p8 + p9 -&gt; p4</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>st p4 -&gt; 4[p3]</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Conservative load scheduling: can’t issue ld4[p1] -> p8**

### 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></td>
</tr>
<tr>
<td>ld 4[p2] -&gt; p9</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>add p8 + p9 -&gt; p4</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>st p4 -&gt; 4[p3]</td>
<td>4</td>
<td>6</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 ([4p1]) -&gt; p8</td>
<td>3</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>ld ([4p2]) -&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; ([4p3])</td>
<td>4</td>
<td>13</td>
<td>14</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>13</td>
<td>14</td>
</tr>
</tbody>
</table>

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

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 (@ execute)
  - Searched by loads (@ execute)
  - Read from to write data cache (@ commit)

- Load Queue: detects ordering violations
  - Written by loads (@ execute)
  - Searched by stores (@ execute)

- 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>

Saves 4 cycles over conservative
Actually uses ooo-ness
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 (#preg)
    - 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
    - Active area of research: scaling window sizes

- Usefulness of large window: limited by branch prediction
  - 95% branch mis-prediction rate: 1 in 20 branches, or 1 in 100 insn.

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
Summary: Dynamic Scheduling

- Dynamic scheduling
  - Totally in the hardware
  - Also called “out-of-order execution” (OoO)
- Fetch many instructions into instruction window
  - Use branch prediction to speculate past (multiple) branches
  - Flush pipeline on branch misprediction
- Rename to avoid false dependencies
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies more tricky
- “Commit” instructions in order
  - Anything strange happens before commit, just flush the pipeline
- Current machines: 100+ instruction scheduling window

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

Static vs Dynamic Scheduling

- If we can do this in software...
- ...why build complex (slow-clock, high-power) hardware?
  + Performance portability
    - Don’t want to recompile for new machines
  + More information available
    - Memory addresses, branch directions, cache misses
  + More registers available
    - Compiler may not have enough to schedule well
  + Speculative memory operation re-ordering
    - Compiler must be conservative, hardware can speculate
    - But compiler has a larger scope
      - Compiler does as much as it can (not much)
      - Hardware does the rest

Summary: Scheduling

- Pipelining and superscalar review
- Code scheduling
  - To reduce pipeline stalls
  - To increase ILP (insn level parallelism)
- Two approaches
  - Static scheduling by the compiler
  - Dynamic scheduling by the hardware
- Up next: multiprocessing