This Unit: Static & Dynamic Scheduling

- 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

- P&H
  - Chapter 4.10 – 4.11

Code Scheduling & Limitations
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...

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

Scheduling Scope Limited by Branches

Before

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2,4(sp)</td>
<td>sp+4</td>
</tr>
<tr>
<td>ld r3,8(sp)</td>
<td>sp+8</td>
</tr>
<tr>
<td>add r3, r2, r1</td>
<td>//stall</td>
</tr>
<tr>
<td>st r1,0(sp)</td>
<td>sp+0</td>
</tr>
<tr>
<td>ld r5,16(sp)</td>
<td>sp+16</td>
</tr>
<tr>
<td>ld r6,20(sp)</td>
<td>sp+20</td>
</tr>
<tr>
<td>sub r5, r6, r4</td>
<td>//stall</td>
</tr>
<tr>
<td>st r4,12(sp)</td>
<td>sp+12</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2,4(sp)</td>
<td>sp+4</td>
</tr>
<tr>
<td>ld r3,8(sp)</td>
<td>sp+8</td>
</tr>
<tr>
<td>add r3, r2, r1</td>
<td>//stall</td>
</tr>
<tr>
<td>st r1,0(sp)</td>
<td>sp+0</td>
</tr>
<tr>
<td>ld r6,20(sp)</td>
<td>sp+20</td>
</tr>
<tr>
<td>sub r5, r6, r4</td>
<td>//stall</td>
</tr>
<tr>
<td>st r4,12(sp)</td>
<td>sp+12</td>
</tr>
</tbody>
</table>

Aside: what does this code do?
Legal to move load up past branch?

Before

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2,4(sp)</td>
<td>sp+4</td>
</tr>
<tr>
<td>ld r3,8(sp)</td>
<td>sp+8</td>
</tr>
<tr>
<td>add r3, r2, r1</td>
<td>//stall</td>
</tr>
<tr>
<td>st r1,0(sp)</td>
<td>sp+0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2,4(sp)</td>
<td>sp+4</td>
</tr>
<tr>
<td>ld r3,8(sp)</td>
<td>sp+8</td>
</tr>
<tr>
<td>add r3, r2, r1</td>
<td>//stall</td>
</tr>
<tr>
<td>st r1,0(sp)</td>
<td>sp+0</td>
</tr>
<tr>
<td>add r3, r2, r1</td>
<td>//stall</td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2,4(sp)</td>
<td>sp+4</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2,4(sp)</td>
<td>sp+4</td>
</tr>
</tbody>
</table>

Aside: what does this code do?
Legal to move load up past branch?
Compiler Scheduling Requires

- **Enough registers**
  - To hold additional “live” values
  - Example code contains 7 different values (including `sp`)
  - Before: max 3 values live at any time → 3 registers enough
  - After: max 4 values live → 3 registers not enough

<table>
<thead>
<tr>
<th>Original</th>
<th>Wrong!</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2,4(sp)</td>
<td>ld r2,4(sp)</td>
</tr>
<tr>
<td>ld r1,8(sp)</td>
<td>ld r1,8(sp)</td>
</tr>
<tr>
<td>add r1,r2,r1</td>
<td>ld r2,16(sp)</td>
</tr>
<tr>
<td>st r1,0(sp)</td>
<td>add r1,r2,r1</td>
</tr>
<tr>
<td>sub r2,r1,r1</td>
<td>sub r2,r1,r1</td>
</tr>
<tr>
<td>st r1,12(sp)</td>
<td>st r1,12(sp)</td>
</tr>
</tbody>
</table>

Code Scheduling Example

**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.)

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

| 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: bht r1,r2,0    | // N*4 in r2 |
SAXPY Performance and Utilization

- Scalar pipeline
  - Full bypassing, 5-cycle E*, 2-cycle E+, branches predicted taken
  - **Performance**: 7 insns / 11 cycles = 0.64 IPC
  - **Utilization**: 0.64 actual IPC / 1 peak IPC = 64%

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

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
```

```
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
```

CIS 371 (Martin): Scheduling 13

CIS 371 (Martin): Scheduling 14

CIS 371 (Martin): Scheduling 15

CIS 371 (Martin): Scheduling 16
Unrolling SAXPY II: Pipeline Schedule

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

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

Unrolled SAXPY Performance/Utilization

- Performance: 12 insn / 13 cycles = 0.92 IPC
- Utilization: 0.92 actual IPC / 1 peak IPC = 92%
- Speedup: (2 * 11 cycles) / 13 cycles = 1.69

Unrolling SAXPY III: “Rename” Registers

- Pipeline scheduling causes reordering violations
  - Use different register names to fix problem

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

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 recurrences (inter-iteration dependences)

```
for (i=0; i<N; i++)
X[i]=A*X[i-1];
```

```
ldf X-4(r1),f1
mul f0,f1,f2
st f2,X(r1)
addi r1,4,r1
blt r1,r2,0
```

- Two mulf’s are not parallel
- Other (more advanced) techniques help
**Recap: 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

**Dynamic Scheduling**

**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 (more information in CIS501)

**Out-of-order Pipeline**
Limitations of In-Order Pipelines

- In-order pipeline, two-cycle load-use penalty
- 2-wide
- Why not?

CIS 371 (Martin): Scheduling

Out-of-Order to the Rescue

- "Dynamic scheduling" done by the hardware
- Still 2-wide superscalar, but now out-of-order, too
  - Allows instructions to issues when dependences are ready
- Longer pipeline
  - Front end: Fetch, "Dispatch"
  - Execution core: "Issue", "Reg. Read", Execute, Memory, Writeback
  - Retirement: "Commit"

Code Example

- Code:
  - Raw insns
  - "Renamed" insns
  - Difficult to reorder above code, names get in the way
  - Divide insn independent of subtract and multiply insns
    - Should be able to 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
**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</td>
</tr>
<tr>
<td>p1</td>
<td>p2</td>
<td>p3</td>
<td>p4</td>
</tr>
<tr>
<td>p4</td>
<td>p2</td>
<td>p5</td>
<td>p6</td>
</tr>
<tr>
<td>p4</td>
<td>p2</td>
<td>p5</td>
<td>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

**Register Renaming Algorithm**

- Data structures:
  - maptable[architectural_reg] → physical_reg
  - Free list: get/put free register (implemented as a queue)
- Algorithm: at decode for each instruction:
  
  ```
  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()
  maptable[arch_output] = new_reg
  insn.phys_output = new_reg
  ```

  - At "commit"
    - Once all older instructions have committed, free register
      put_free_phys_reg(insn.phys_to_free)

**Out-of-order Pipeline**

- Have unique register names
- Now put into out-of-order execution structures

- Freeing over-written register

```
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
```

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

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 is "ready"
      select the oldest "ready" instruction
      table[insn.phys_output] = ready
  ```

Dynamic Scheduling Example

- The following slides are a detailed but concrete example
- Yet, it contains enough detail to be overwhelming
  - Try not to worry about the details
- Focus on the big picture take-away:
  
  **Hardware can reorder instructions to extract instruction-level parallelism**
Recall: Motivating Example

How would this execution occur cycle-by-cycle?

Out-of-Order Pipeline – Cycle 0

Out-of-Order Pipeline – Cycle 1a

Out-of-Order Pipeline – Cycle 1b
### Out-of-Order Pipeline – Cycle 6

<table>
<thead>
<tr>
<th>Cycle 6</th>
<th>0</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>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld ([r1] \rightarrow r2)</td>
<td>F</td>
<td>Di</td>
<td>1</td>
<td>RR</td>
<td>X</td>
<td>M_1</td>
<td>M_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add (r2 + r3 \rightarrow r4)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td>RR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor (r4 \rightarrow r5 \rightarrow r6)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld ([r7] \rightarrow r4)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td>RR</td>
<td>X</td>
<td>M_1</td>
<td>M_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Map Table
- \(r1\): p8
- \(r2\): p9
- \(r3\): p6
- \(r4\): p12
- \(r5\): p4
- \(r6\): p11
- \(r7\): p2
- \(r8\): p1

#### Ready Table
- \(p1\): yes
- \(p2\): yes
- \(p3\): yes
- \(p4\): yes
- \(p6\): yes
- \(p9\): yes
- \(p10\): yes
- \(p11\): yes

#### Issue Queue
- Insns: \(ld\), \(add\), \(xor\)
- Src1: p8, p3, p10
- R7: yes, yes, yes
- Src2: p9, p4, p11
- R8: yes, yes, yes
- RDest: p7, p10, p12
- Age: 0, 2, 3

#### Reorder Buffer
- Insns: \(ld\), \(add\), \(xor\)
- To Free: yes, no, yes
- Done?: yes, no, yes

### Out-of-Order Pipeline – Cycle 7

<table>
<thead>
<tr>
<th>Cycle 7</th>
<th>0</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>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld ([r1] \rightarrow r2)</td>
<td>F</td>
<td>Di</td>
<td>1</td>
<td>RR</td>
<td>X</td>
<td>M_1</td>
<td>M_2</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add (r2 + r3 \rightarrow r4)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td>RR</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor (r4 \rightarrow r5 \rightarrow r6)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td>RR</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld ([r7] \rightarrow r4)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td>RR</td>
<td>X</td>
<td>M_1</td>
<td>M_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Map Table
- \(r1\): p8
- \(r2\): p9
- \(r3\): p6
- \(r4\): p12
- \(r5\): p4
- \(r6\): p11
- \(r7\): p2
- \(r8\): p1

#### Ready Table
- \(p1\): yes
- \(p2\): yes
- \(p3\): yes
- \(p4\): yes
- \(p6\): yes
- \(p9\): yes
- \(p10\): yes
- \(p11\): yes

#### Issue Queue
- Insns: \(ld\), \(add\), \(xor\)
- Src1: p8, p3, p10
- R7: yes, yes, yes
- Src2: p9, p4, p11
- R8: yes, yes, yes
- RDest: p7, p10, p12
- Age: 0, 2, 3

#### Reorder Buffer
- Insns: \(ld\), \(add\), \(xor\)
- To Free: yes, no, yes
- Done?: yes, no, yes

### Out-of-Order Pipeline – Cycle 8a

<table>
<thead>
<tr>
<th>Cycle 8a</th>
<th>0</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>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld ([r1] \rightarrow r2)</td>
<td>F</td>
<td>Di</td>
<td>1</td>
<td>RR</td>
<td>X</td>
<td>M_1</td>
<td>M_2</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add (r2 + r3 \rightarrow r4)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td>RR</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor (r4 \rightarrow r5 \rightarrow r6)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td>RR</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld ([r7] \rightarrow r4)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td>RR</td>
<td>X</td>
<td>M_1</td>
<td>M_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Map Table
- \(r1\): p8
- \(r2\): p9
- \(r3\): p6
- \(r4\): p12
- \(r5\): p4
- \(r6\): p11
- \(r7\): p2
- \(r8\): p1

#### Ready Table
- \(p1\): yes
- \(p2\): yes
- \(p3\): yes
- \(p4\): yes
- \(p6\): yes
- \(p9\): yes
- \(p10\): yes
- \(p11\): yes

#### Issue Queue
- Insns: \(ld\), \(add\), \(xor\)
- Src1: p8, p3, p10
- R7: yes, yes, yes
- Src2: p9, p4, p11
- R8: yes, yes, yes
- RDest: p7, p10, p12
- Age: 0, 2, 3

#### Reorder Buffer
- Insns: \(ld\), \(add\), \(xor\)
- To Free: yes, no, yes
- Done?: yes, no, yes

### Out-of-Order Pipeline – Cycle 8b

<table>
<thead>
<tr>
<th>Cycle 8b</th>
<th>0</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>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld ([r1] \rightarrow r2)</td>
<td>F</td>
<td>Di</td>
<td>1</td>
<td>RR</td>
<td>X</td>
<td>M_1</td>
<td>M_2</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add (r2 + r3 \rightarrow r4)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td>RR</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor (r4 \rightarrow r5 \rightarrow r6)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td>RR</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld ([r7] \rightarrow r4)</td>
<td>F</td>
<td>Di</td>
<td>I</td>
<td>RR</td>
<td>X</td>
<td>M_1</td>
<td>M_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Map Table
- \(r1\): p8
- \(r2\): p9
- \(r3\): p6
- \(r4\): p12
- \(r5\): p4
- \(r6\): p11
- \(r7\): p2
- \(r8\): p1

#### Ready Table
- \(p1\): yes
- \(p2\): yes
- \(p3\): yes
- \(p4\): yes
- \(p6\): yes
- \(p9\): yes
- \(p10\): yes
- \(p11\): yes

#### Issue Queue
- Insns: \(ld\), \(add\), \(xor\)
- Src1: p8, p3, p10
- R7: yes, yes, yes
- Src2: p9, p4, p11
- R8: yes, yes, yes
- RDest: p7, p10, p12
- Age: 0, 2, 3

#### Reorder Buffer
- Insns: \(ld\), \(add\), \(xor\)
- To Free: yes, no, yes
- Done?: yes, no, yes
### Out-of-Order Pipeline – Cycle 9a

<table>
<thead>
<tr>
<th>Cycle 9a</th>
<th>0</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>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Table</td>
<td></td>
<td></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>p1</td>
<td>p8</td>
<td>yes</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>p2</td>
<td>p9</td>
<td>yes</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>p3</td>
<td>p6</td>
<td>yes</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>p4</td>
<td>p12</td>
<td>yes</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>p5</td>
<td>p7</td>
<td>yes</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>p6</td>
<td>p11</td>
<td>yes</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>p7</td>
<td>p2</td>
<td>yes</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>p8</td>
<td>p1</td>
<td>yes</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>

### Out-of-Order Pipeline – Cycle 9b

<table>
<thead>
<tr>
<th>Cycle 9b</th>
<th>0</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>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Table</td>
<td></td>
<td></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>p1</td>
<td>p8</td>
<td>yes</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>p2</td>
<td>p9</td>
<td>yes</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>p3</td>
<td>p6</td>
<td>yes</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>p4</td>
<td>p12</td>
<td>yes</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>p5</td>
<td>p7</td>
<td>yes</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>p6</td>
<td>p11</td>
<td>yes</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>p7</td>
<td>p2</td>
<td>yes</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>p8</td>
<td>p1</td>
<td>yes</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>

### Out-of-Order Pipeline – Cycle 10

<table>
<thead>
<tr>
<th>Cycle 10</th>
<th>0</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>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Table</td>
<td></td>
<td></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>p1</td>
<td>p8</td>
<td>yes</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>p2</td>
<td>p9</td>
<td>yes</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>p3</td>
<td>p6</td>
<td>yes</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>p4</td>
<td>p12</td>
<td>yes</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>p5</td>
<td>p7</td>
<td>yes</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>p6</td>
<td>p11</td>
<td>yes</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>p7</td>
<td>p2</td>
<td>yes</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>p8</td>
<td>p1</td>
<td>yes</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>

### Out-of-Order Pipeline – Done!

<table>
<thead>
<tr>
<th>Cycle Done</th>
<th>0</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>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Table</td>
<td></td>
<td></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>p1</td>
<td>p8</td>
<td>yes</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>p2</td>
<td>p9</td>
<td>yes</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>p3</td>
<td>p6</td>
<td>yes</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>p4</td>
<td>p12</td>
<td>yes</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>p5</td>
<td>p7</td>
<td>yes</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>p6</td>
<td>p11</td>
<td>yes</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>p7</td>
<td>p2</td>
<td>yes</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>p8</td>
<td>p1</td>
<td>yes</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>
More Dynamic Scheduling Mechanisms

- How are physical registers reclaimed?
  - Need to recycle them eventually
- How are branch mispredictions handled?
  - Need to selectively flush instructions
- How are stores handled?
  - If they execute early, but then need to be flushed?
  - Avoid writing cache until "commit"
  - Forward to dependent loads with "load/store queue"
- What about out-of-order stores & loads?
  - What if a store executes "too early"
  - Solution: predict when to execute, speculate, detect violations
- How do we avoid hurting clock frequency?
  - And without using too much energy?

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)

Before

\begin{verbatim}
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)
\end{verbatim}

Wrong(?)

\begin{verbatim}
ld r2,4(sp)  
ld r3,8(sp)  
add r3,r2,r1  
ld r5,0(r8) //does r8==sp?  
add r3,r2,r1  
ld r6,4(r8) //does r8+4==sp?  
st r1,0(sp)  
sub r5,r6,r4 //stall  
st r5,r6,r4  
st r4,8(r8)
\end{verbatim}

Scheduling Redux

- Static scheduling
  - Performed by compiler, limited in several ways
- Dynamic scheduling
  - Performed by the hardware, overcomes limitations
- Static limitation -> Dynamic mitigation
  - Number of registers in the ISA -> register renaming
  - Scheduling scope -> branch prediction & speculation
  - Inexact memory aliasing information -> speculative memory ops
  - Unknown latencies of cache misses -> execute when ready
- Which to do? Compiler does what it can, hardware the rest
  - Why? dynamic scheduling needed to sustain more than 2-way issue
  - Helps with hiding memory latency (execute around misses)
  - Intel Core i7 is four-wide execute w/ 128-insn scheduling window
  - Even mobile phones will have dynamic scheduled cores (ARM A9)