Parallelism I: Inside the Core
The final

- Comprehensive
- Same general format as the Midterm.
- Review the homeworks, the slides, and the quizzes.
Key Points

• What is wide issue mean?
• How does it affect performance?
• How does it affect pipeline design?
• What is the basic idea behind out-of-order execution?
• What is the difference between a true and false dependence?
• How do OOO processors remove false dependences?
• What is Simultaneous Multithreading?
Parallelism

- ET = IC * CPI * CT
- IC is more or less fixed
- We have shrunk cycle time as far as we can
- We have achieved a CPI of 1.
- Can we get faster?

We can reduce our CPI to less than 1. The processor must do multiple operations at once. This is called Instruction Level Parallelism (ILP)
Approach 1: Widen the pipeline

- Process two instructions at once instead of 1
- Often 1 “odd” PC instruction and 1 “even” PC
  - This keeps the instruction fetch logic simpler.
- 2-wide, in-order, superscalar processor
- Potential problems?
# Single issue refresher

<table>
<thead>
<tr>
<th></th>
<th>cycle 0</th>
<th>cycle 1</th>
<th>cycle 2</th>
<th>cycle 3</th>
<th>cycle 4</th>
<th>cycle 5</th>
<th>cycle 6</th>
<th>cycle 7</th>
<th>cycle 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1,$s2,$s3</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $s2,$s4,$s5</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld $s3, 0($s2)</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $t1, $s3, $s3</td>
<td>F</td>
<td>D</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Forwarding*
## Dual issue: Ideal Case

<table>
<thead>
<tr>
<th></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>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1,$s2,$s3</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $s2,$s4,$s5</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld $s3, 0($s2)</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $t1, $s3, $s3</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPI == 0.5!
Dual issue: Structural Hazards

• Structural hazards
  • We might not replicate everything
  • Perhaps only one multiplier, one shifter, and one load/store unit
  • What if the instruction is in the wrong place?

If an “upper” instruction needs the “lower” pipeline, squash the “lower” instruction
## Dual issue: dealing with hazards

<table>
<thead>
<tr>
<th></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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PC = 0</strong></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>add</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PC = 8</strong></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>Mul</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PC = 12</strong></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>Shift</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ld</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td><strong>PC = 12</strong></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>Shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Ld</td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Shift moves to lower pipe**
- **Load is squashed**
- **Load uses lower pipe**
- **Shift becomes a noop**
Dual issue: Data Hazards

- The “lower” instruction may need a value produced by the “upper” instruction
- Forwarding cannot help us -- we must stall.
Dual issue: dealing with hazards

- Forwarding is essential!
- Both pipes stall.

<table>
<thead>
<tr>
<th></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>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1, $s3,#4</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $s4, $s1, #4</td>
<td>F</td>
<td>D</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add ...</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub ...</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and ...</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or ...</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dual issue: Control Hazards

- The “upper” instruction might be a branch.
- The “lower” instruction might be on the wrong path.
- Solution 1: Require branches to execute in the lower pipeline -- See “structural hazards”.
- What about consecutive branches? -- Exercise for the reader.
- What about branches to odd addresses? -- Squash the upper pipe.
Beyond Dual Issue

• Wider pipelines are possible.
  • There is often a separate floating point pipeline.
• Wide issue leads to hardware complexity
• Compiling gets harder, too.
• In practice, processors use one of two options if they want more ILP
  • Change the ISA and build a smart compiler: VLIW
  • Keep the same ISA and build a smart processors: Out-of-order
Data dependences

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

• But beware:
  • Is there a dependence here?

    1: add $t1,$s2,$s3

    2: sub $t1,$s3,$s4

  • Can we reorder the instructions?

    2: sub $t1,$s3,$s4

    1: add $t1,$s2,$s3

  • Is the result the same?

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

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

• Is there a dependence here?

1: add $t1,$s2,$s3
2: sub $s2,$s3,$s4

• Can we reorder the instructions?

2: sub $s2,$s3,$s4
1: add $t1,$s2,$s3

• Is the result the same?

No! The value in $s2$ that 1 needs will be destroyed
False Dependence #2

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

- Any sequence of instructions has set of RAW, WAW, and WAR dependences that constrain its execution.
- Can we design a processor that extracts as much parallelism as possible, while still respecting these dependences?
The Central OOO Idea

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

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

8 Instructions in 5 cycles
Simplified OOO Pipeline

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

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

Schedule  execute
Keeping the Window Filled

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

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

• If WAW and WAR dependences arise because we have too few registers
  • Let’s add more!
• But! We can’t! The Architecture only gives us 32 (why or why did we only use 5 bits?)
• Solution:
  • Define a set of internal “physical” register that is as large as the number of instructions that can be “in flight” -- 128 in a recent intel chip.
  • Every instruction in the pipeline gets a registers
  • Maintaining a register mapping table that determines which physical register currently holds the value for the required “architectural” registers.
• This is called “Register Renaming”
Alpha 21264: Renaming

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

Register map table

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

RAW  WAW  WAR
Alpha 21264: Renaming

Register map table

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

p1 currently holds the value of architectural registers r1
1: Add r3, r2, r3  p4, p2, p3
2: Sub r2, r1, r3
3: Mult r1, r3, r1
4: Add r2, r3, r1
5: Add r2, r1, r3

RAW  WAW  WAR
Alpha 21264: Renaming

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

p4, p2, p3
p5, p1, p4

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

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

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

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

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

RAW → WAW → WAR

p4, p2, p3
p5, p1, p4
p6, p4, p1
p7, p4, p6
Alpha 21264: Renaming

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

p4, p2, p3
p5, p1, p4
p6, p4, p1
p7, p4, p6
p8, p6, p4

<table>
<thead>
<tr>
<th></th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p1</td>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>1</td>
<td>p1</td>
<td>p2</td>
<td>p4</td>
</tr>
<tr>
<td>2</td>
<td>p1</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>3</td>
<td>p6</td>
<td>p5</td>
<td>p4</td>
</tr>
<tr>
<td>4</td>
<td>p6</td>
<td>p7</td>
<td>p4</td>
</tr>
<tr>
<td>5</td>
<td>p6</td>
<td>p8</td>
<td>p4</td>
</tr>
</tbody>
</table>
Alpha 21264: Renaming

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

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

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

- The fastest machines in the world are OOO superscalars
- AMD Barcelona
  - 6-wide issue
  - 106 instructions in flight at once.
- Intel Nehalem
  - 5-way issue to 12 ALUs
  - >128 instructions in flight
- OOO provides the most benefit for memory operations.
  - Non-dependent instructions can keep executing during cache misses.
  - This is so-called “memory-level parallelism.”
  - It is enormously important. CPU performance is (almost) all about memory performance nowadays (remember the memory wall graphs!)
### Question 1: Multiple Answer

Which of the following can significantly reduce conflict misses?

<table>
<thead>
<tr>
<th>Correct Answers</th>
<th>Percent Correct</th>
<th>Percent Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reducing associativity.</td>
<td>96.491%</td>
<td>3.509%</td>
</tr>
<tr>
<td>Increasing associativity.</td>
<td>94.737%</td>
<td>5.263%</td>
</tr>
<tr>
<td>Increasing the number of cache lines.</td>
<td>80.702%</td>
<td>19.298%</td>
</tr>
<tr>
<td>Improving the replacement policy.</td>
<td>50.877%</td>
<td>49.123%</td>
</tr>
</tbody>
</table>

Average Score 1.48538 points
Question 2: Multiple Answer

Which of the following can significantly reduce capacity misses?

Correct Answers

- Increasing associativity.
- Increasing the number of cache lines, while holding the cache line size constant.
- Improving the cache replacement policy.
- Making the cache write through, rather than write back.

Average Score 0.5614 points

Percent Correct  Percent Incorrect

40.351%     59.649%
80.702%     19.298%
85.965%     14.035%
87.719%     12.281%
**Question 3: True/False**

Write back is always better than write through.

<table>
<thead>
<tr>
<th>Correct Answers</th>
<th>Percent Answered</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>3.509%</td>
</tr>
<tr>
<td>False</td>
<td>96.491%</td>
</tr>
<tr>
<td>Unanswered</td>
<td>0%</td>
</tr>
</tbody>
</table>

Average Score 0.96491 points
Compute the average CPI for a system with an L1 hit rate of 90% and an L2 hit rate of 95%. Assume that L1 hits take 1 cycle, L2 hits take 20 cycles, and main memory accesses take 100 cycles. Further assume that 20% of instructions are memory accesses, and that the processor has a perfect branch predictor.

- 0.8*1 + // non-memory
- 0.2* // memory
  - (0.9*1 // L1 hits
  - + 0.1* // L1 misses
    - (0.95*20 // L2 hits
    - + 0.05*100) // L2 misses
Question 6: True/False

Increasing the number of sets in an associative cache causes the number of index bits to increase.

<table>
<thead>
<tr>
<th>Correct</th>
<th>Answers</th>
<th>Percent Answered</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>True</td>
<td>57.895%</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>42.105%</td>
</tr>
<tr>
<td></td>
<td>Unanswered</td>
<td>0%</td>
</tr>
</tbody>
</table>
Question 7:  Multiple Choice

Given a 32 KB, 2-way set associative, 64 byte cache lines, give the number of offset, index, and tag bits required to access it.

Correct

Index:32
offset:8
Tag:12

Index:7
offset:5
Tag:19

Index:9
offset:5
Tag:17

Index:8
offset:6
Tag:18

Unanswered

Average Score 2.89474 points

Percent Answered

0%

0%

3.509%

96.491%

0%
The Problem with OOO

- Even the fastest OOO machines only get about 1-2 IPC, even though they are 4-5 wide.
- Problems
  - Insufficient ILP within applications. -- 1-2 per thread, usually
  - Poor branch prediction performance
  - Single threads also have little memory parallelism.
- Observation
  - On many cycles, many ALUs and instruction queue slots sit empty
Simultaneous Multithreading

- AKA HyperThreading in Intel machines
- Run multiple threads at the same time
- Just throw all the instructions into the pipeline
- Keep some separate data for each
  - Renaming table
  - TLB entries
  - PCs
- But the rest of the hardware is shared.
- It is surprisingly simple (but still quite complicated)
SMT Advantages

- Exploit the ILP of multiple threads at once
- Less dependence or branch prediction (fewer correct predictions required per thread)
- Less idle hardware (increased power efficiency)
- Much higher IPC -- up to 4
- Disadvantages: threads can fight over resources and slow each other down.

- Historical footnote: Invented, in part, by our own Dean Tullsen when he was at UW