Simple solution Option 1: Stall on every branch until have new PC value
– Would add 2 bubbles/clock cycles for every Branch (~ 20% of instructions executed)

Control Hazard: Branching
- Optimization #1:
  - Insert special branch comparator in Stage 2
  - As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
- Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
- Side Note: means that branches are idle in Stages 3, 4 and 5

Question: What's an efficient way to implement the equality comparison?
Control Hazards: Branching

- Option 2: Predict outcome of a branch, fix up if guess wrong
  - Must cancel all instructions in pipeline that depended on guess that was wrong
  - This is called "flushing" the pipeline
- Simplest hardware if we predict that all branches are NOT taken
  - Why?

Control Hazards: Branching

- Option #3: Redefine branches
  - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (the branch-delay slot)
- Delayed Branch means we always execute inst after branch
- This optimization is used with MIPS

Example: Nondelayed vs. Delayed Branch

Nondelayed Branch

```
or $8, $9, $10
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
```

Delayed Branch

```
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
or $8, $9, $10
```

Control Hazards: Branching

- Notes on Branch-Delay Slot
  - Worst-Case Scenario: put a no-op in the branch-delay slot
  - Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn’t affect the logic of program
  - Re-ordering instructions is common way to speed up programs
  - Compiler usually finds such an instruction 50% of time
  - Jumps also have a delay slot...

Control Hazards: Branching

- Multiple issue “superscalar”
  - Replicate pipeline stages => multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
  - But dependencies reduce this in practice

Greater Instruction-Level Parallelism (ILP)

- Deeper pipeline (5 => 10 => 15 stages)
  - Less work per stage => shorter clock cycle
- Multiple issue “superscalar”
  - Replicate pipeline stages => multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
  - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
- Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPI examines instruction stream and chooses instructions
  - Issue each cycle
  - Compiler can help by reordering instructions
  - CPI resolves hazards using advanced techniques at runtime

Superscalar Laundry: Parallel per stage

```
6 PM 7 8 9 10 11 12 1 AM
```

Control Hazards: Branching

- More resources, how to match mix of parallel tasks?

Pipeline Depth and Issue Width

- Intel Processors over Time

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipelines</th>
<th>Issue Width</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1985</td>
<td>60 MHz</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>10 W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>233 MHz</td>
<td>14</td>
<td>2</td>
<td>1</td>
<td>65 W</td>
</tr>
<tr>
<td>Athlon</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>3</td>
<td>1</td>
<td>28 W</td>
</tr>
<tr>
<td>Pentium II</td>
<td>2004</td>
<td>2660 MHz</td>
<td>22</td>
<td>2</td>
<td>1</td>
<td>2.8W</td>
</tr>
<tr>
<td>Core 2 Extreme</td>
<td>2006</td>
<td>2900 MHz</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>75 W</td>
</tr>
<tr>
<td>Core 2 Extreme</td>
<td>2006</td>
<td>2930 MHz</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>100 W</td>
</tr>
<tr>
<td>Core 2 Extreme</td>
<td>2007</td>
<td>3400 MHz</td>
<td>16</td>
<td>4</td>
<td>8</td>
<td>150 W</td>
</tr>
</tbody>
</table>

Parallelism and Advanced Instruction Level Parallelism (ILP)
**Pipeline Depth and Issue Width**

![Graph showing pipeline depth and issue width over time]

**Static Multiple Issue**

- Compiler groups instructions into "issue packets"  
  - Group of instructions that can be issued on a single cycle  
  - Determined by pipeline resources required  
- Think of an issue packet as a very long instruction  
  - Specifies multiple concurrent operations

**Scheduling Static Multiple Issue**

- Compiler must remove some/all hazards  
  - Reorder instructions into issue packets  
  - No dependencies within a packet  
  - Possibly some dependencies between packets  
  - Varies between ISAs; compiler must know!  
  - Pad issue packet with nop if necessary

**MIPS with Static Dual Issue**

- Two-issue packets  
  - One ALU/branch instruction  
  - One load/store instruction  
  - 64-bit aligned  
  - ALU/branch, then load/store  
- Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Duration</th>
<th>Latency</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Branch</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Load/Store</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

**Hazards in the Dual-Issue MIPS**

- More instructions executing in parallel  
- EX data hazard  
  - Forwarding avoided stalls with single-issue  
  - Now can’t use ALU result in load/store in same packet  
  - Add $t0, $s0, $t1  
  - Load $s2, $t1[10:0]  
  - Split into two packets, effectively a stall  
- Load-use hazard  
  - Still one cycle use latency, but now two instructions  
- More aggressive scheduling required

**Scheduling Example**

- Schedule this for dual-issue MIPS  

```
Loop: lw $t0, 0($s1)  # Store array element  
add $t0, $t0, $s1  # Add scalar in $s1  
sw $t0, 0($s1)  # Store result  
add $t1, $s1, 4  # Decrement pointer  
bne $t1, $zero, Loop # Branch $s1!=0
```

**Loop Unrolling**

- Replicate loop body to expose more parallelism  
  - Reduces loop-control overhead  
  - Use different registers per replication  
  - Called "register renaming"  
  - Avoid loop-carried "anti-dependencies"  
    - Store followed by a load of the same register  
    - Aka "name dependence"  
      - Race of a register name

```
Loop Unrolling Example
```

```
Pipe = 5/4 = 1.25 (c.f. peak PIPE = 2)
```

**Scheduling Example**

- Schedule this for dual-issue MIPS  

```
Loop: lw $t0, 0($s1)  # Store array element  
add $t0, $t0, $s1  # Add scalar in $s1  
sw $t0, 0($s1)  # Store result  
add $t1, $s1, 4  # Decrement pointer  
bne $t1, $zero, Loop # Branch $s1!=0
```

**Loop Unrolling Example**

```
Pipe = 5/4 = 1.25 (c.f. peak PIPE = 2)
```

**Hazards in the Dual-Issue MIPS**

- More instructions executing in parallel  
- EX data hazard  
  - Forwarding avoided stalls with single-issue  
  - Now can’t use ALU result in load/store in same packet  
  - Add $t0, $s0, $t1  
  - Load $s2, $t1[10:0]  
  - Split into two packets, effectively a stall  
- Load-use hazard  
  - Still one cycle use latency, but now two instructions  
- More aggressive scheduling required

**Scheduling Example**

- Schedule this for dual-issue MIPS  

```
Loop: lw $t0, 0($s1)  # Store array element  
add $t0, $t0, $s1  # Add scalar in $s1  
sw $t0, 0($s1)  # Store result  
add $t1, $s1, 4  # Decrement pointer  
bne $t1, $zero, Loop # Branch $s1!=0
```

**Loop Unrolling**

- Replicate loop body to expose more parallelism  
  - Reduces loop-control overhead  
  - Use different registers per replication  
  - Called "register renaming"  
  - Avoid loop-carried "anti-dependencies"  
    - Store followed by a load of the same register  
    - Aka "name dependence"  
      - Race of a register name

```
Loop Unrolling Example
```

```
Pipe = 5/4 = 1.25 (c.f. peak PIPE = 2)
```

**Scheduling Example**

- Schedule this for dual-issue MIPS  

```
Loop: lw $t0, 0($s1)  # Store array element  
add $t0, $t0, $s1  # Add scalar in $s1  
sw $t0, 0($s1)  # Store result  
add $t1, $s1, 4  # Decrement pointer  
bne $t1, $zero, Loop # Branch $s1!=0
```

**Loop Unrolling Example**

```
Pipe = 5/4 = 1.25 (c.f. peak PIPE = 2)
```

**Scheduling Example**

- Schedule this for dual-issue MIPS  

```
Loop: lw $t0, 0($s1)  # Store array element  
add $t0, $t0, $s1  # Add scalar in $s1  
sw $t0, 0($s1)  # Store result  
add $t1, $s1, 4  # Decrement pointer  
bne $t1, $zero, Loop # Branch $s1!=0
```

**Loop Unrolling**

- Replicate loop body to expose more parallelism  
  - Reduces loop-control overhead  
  - Use different registers per replication  
  - Called "register renaming"  
  - Avoid loop-carried "anti-dependencies"  
    - Store followed by a load of the same register  
    - Aka "name dependence"  
      - Race of a register name

```
Loop Unrolling Example
```

```
Pipe = 5/4 = 1.25 (c.f. peak PIPE = 2)
```
Dynamic Multiple Issue
- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, ... each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling
- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example
  \[ \text{lw} \quad \text{add} \quad \text{slti} \]
  
  30, 20($s2) 31, 10, 20
  
  Can start subu while addu is waiting for lw

Why Do Dynamic Scheduling?
- Why not just let the compiler schedule code?
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Speculation
- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome (Branch Prediction)
  - Roll back if path taken is different
  - Speculate on load
  - Roll back if location is updated

Pipeline Hazard: Matching socks in later load
- A depends on D, stall since folder filled up.

Out of Order Intel
- All use OOO since 2001

Does Multiple Issue Work?
- Yes, but not as much as we’d like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Out-of-Order Laundry: Don’t Wait
- A depends on D: stall since folder filled up.

"And in Conclusion..
- Pipelining is an important form of ILP
- Challenge is (are?) hazards
  - Forwarding helps w/many data hazards
  - Delayed branch helps with control hazard in 5 stage pipeline
  - Load delay slot / interlock necessary
- More aggressive performance:
  - Longer pipelines
  - Superscalar
  - Out-of-order execution
  - Speculation