Pipelined Execution, part II

Review: Pipeline (1/2)
- Optimal Pipeline
  - Each stage is executing part of an instruction each clock cycle.
  - One instruction finishes during each clock cycle.
  - On average, execute far more quickly.
- What makes this work?
  - Similarities between instructions allow us to use same stages for all instructions (generally).
  - Each stage takes about the same amount of time as all others: little wasted time.

Review: Pipeline (2/2)
- Pipelining is a BIG IDEA
  - widely used concept
- What makes it less than perfect?
  - Structural hazards: suppose we had only one cache? => Need more HW resources
  - Control hazards: need to worry about branch instructions?
    => Delayed branch
  - Data hazards: an instruction depends on a previous instruction?

Control Hazard: Branching (1/7)
- We put branch decision-making hardware in ALU stage
  - therefore two more instructions after the branch will always be fetched, whether or not the branch is taken
- Desired functionality of a branch
  - if we do not take the branch, don’t waste any time and continue executing normally
  - if we take the branch, don’t execute any instructions after the branch, just go to the desired label

Games to learn?!
Recent studies show that there may be a place for computer games in traditional K-12 classrooms. There is data that shows dropout rates are lower, SATs, enjoyment, interest up!

Control Hazard: Branching (2/7)
- Initial Solution: Stall until decision is made
  - insert “no-op” instructions: those that accomplish nothing, just take time
  - Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)
Control Hazard: Branching (4/7)

- Optimization #1:
  - move asynchronous comparator up to Stage 2
  - as soon as instruction is decoded (Opcode identifies is as a branch), immediately make a decision and set the value of the PC (if necessary)
  - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  - Side Note: This means that branches are idle in Stages 3, 4 and 5.

Control Hazard: Branching (5/7)

- Insert a single no-op (bubble)

Control Hazard: Branching (6/7)

- Optimization #2: 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 (called the branch-delay slot)
  - The term "Delayed Branch" means we always execute inst after branch

Control Hazard: Branching (7/7)

- Notes on Branch-Delay Slot
  - Worst-Case Scenario: can always put a no-op in the branch-delay slot
  - Better Case: can find an instruction preceding the branch which can be placed in the branch-delay slot without affecting flow of the program
    - re-ordering instructions is a common method of speeding up programs
    - compiler must be very smart in order to find instructions to do this
    - usually can find such an instruction at least 50% of the time
    - Jumps also have a delay slot...

Example: Nondelayed vs. Delayed Branch

<table>
<thead>
<tr>
<th>Nondelayed Branch</th>
<th>Delayed Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>or $8, $9, $10</td>
<td>add $1, $2, $3</td>
</tr>
<tr>
<td>add $1, $2, $3</td>
<td>sub $4, $5, $6</td>
</tr>
<tr>
<td>sub $4, $5, $6</td>
<td>beq $1, $4, Exit</td>
</tr>
<tr>
<td>beq $1, $4, Exit</td>
<td>or $8, $9, $10</td>
</tr>
<tr>
<td>xor $10, $1, $11</td>
<td>xor $10, $1, $11</td>
</tr>
</tbody>
</table>

Data Hazards (1/2)

- Consider the following sequence of instructions
  - add $t0, $t1, $t2
  - sub $t4, $t0, $t3
  - and $t5, $t0, $t6
  - or $t7, $t0, $t8
  - xor $t9, $t0, $t10
Data Hazards (2/2)

Dependencies backwards in time are hazards

<table>
<thead>
<tr>
<th>Instruction Order</th>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t0$, $t1$, $t2$</td>
<td>IF</td>
</tr>
<tr>
<td>sub $t4$, $t0$, $t3$</td>
<td>ALU</td>
</tr>
<tr>
<td>and $t5$, $t0$, $t6$</td>
<td>ALU</td>
</tr>
<tr>
<td>or $t7$, $t0$, $t8$</td>
<td>ALU</td>
</tr>
<tr>
<td>xor $t9$, $t0$, $t10$</td>
<td>ALU</td>
</tr>
</tbody>
</table>

Data Hazard Solution: Forwarding

- Forward result from one stage to another

```
add $t0$, $t1$, $t2$
sub $t4$, $t0$, $t3$
and $t5$, $t0$, $t6$
or $t7$, $t0$, $t8$
xor $t9$, $t0$, $t10$
```

“xor” hazard solved by register hardware

Data Hazard: Loads (1/4)

- Dependencies backwards in time are hazards

```
lw $t0$, 0($t1)
sub $t3$, $t0$, $t2$
and $t5$, $t0$, $t6$
or $t7$, $t0$, $t8$
xor $t9$, $t0$, $t10$
```

Data Hazard: Loads (2/4)

- Hardware must stall pipeline
- Called “interlock”

```
lw $t0$, 0($t1)
sub $t3$, $t0$, $t2$
and $t5$, $t0$, $t6$
or $t7$, $t0$, $t8$
xor $t9$, $t0$, $t10$
```

Data Hazard: Loads (3/4)

- Instruction slot after a load is called “load delay slot”
- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
- If the compiler puts an unrelated instruction in that slot, then no stall
- Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)

Data Hazard: Loads (4/4)

- Stall is equivalent to nop

```
lw $t0$, 0($t1)
nop
sub $t3$, $t0$, $t2$
and $t5$, $t0$, $t6$
or $t7$, $t0$, $t8$
```
Historical Trivia

- First MIPS design did not interlock and stall on load-use data hazard
- Real reason for name behind MIPS: Microprocessor without Interlocked Pipeline Stages
  - Word Play on acronym for Millions of Instructions Per Second, also called MIPS

Review Pipeline Hazard: Stall is dependency

Out-of-Order Laundry: Don’t Wait

Superscalar Laundry: Parallel per stage

Superscalar Laundry: Mismatch Mix

“And in Conclusion...”

- Pipeline challenge is hazards
  - Forwarding helps with many data hazards
  - Delayed branch helps with control hazard in 5 stage pipeline
- More aggressive performance:
  - Superscalar
  - Out-of-order execution