Review: Pipeline (1/2)

Optimal Pipeline
- Each stage is executing part of an instruction each clock cycle.
- One inst. 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: Branch instructions affect which instructions come next.
    ⇒ Delayed branch
  - Data hazards: Data flow between instructions.

Control Hazard: Branching (1/8)

Where do we do the compare for the branch?
Control Hazard: Branching (2/8)

- We had 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

Control Hazard: Branching (3/8)

- 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 (4/8)

**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: This means that branches are idle in Stages 3, 4 and 5.

Control Hazard: Branching (5/8)

Branch comparator moved to Decode stage.
Control Hazard: Branching (6a/8)

° User inserting no-op instruction

° Impact: 2 clock cycles per branch instruction ⇒ slow

Control Hazard: Branching (6b/8)

° Controller inserting a single bubble

° Impact: 2 clock cycles per branch instruction ⇒ slow
Control Hazard: Branching (7/8)

° 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

° This optimization is used on the MIPS

Control Hazard: Branching (8/8)

° 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

Nondelayed Branch

```
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
xor $10, $1, $11
```

Delayed Branch

```
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
xor $10, $1, $11
```

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)

Data-flow backward in time are hazards

Time (clock cycles)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID/RF</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t0,$t1,$t2</td>
<td>IS</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
<tr>
<td>sub $t4,$t0,$t3</td>
<td>IS</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
<tr>
<td>and $t5,$t0,$t6</td>
<td>IS</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
<tr>
<td>or $t7,$t0,$t8</td>
<td>IS</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
<tr>
<td>xor $t9,$t0,$t10</td>
<td>IS</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
</tbody>
</table>

Data Hazard Solution: Forwarding

- Forward result from one stage to another

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID/RF</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t0,$t1,$t2</td>
<td>IS</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
<tr>
<td>sub $t4,$t0,$t3</td>
<td>IS</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
<tr>
<td>and $t5,$t0,$t6</td>
<td>IS</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
<tr>
<td>or $t7,$t0,$t8</td>
<td>IS</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
<tr>
<td>xor $t9,$t0,$t10</td>
<td>IS</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
</tbody>
</table>

“or” hazard solved by register hardware
**Data Hazard: Loads (1/4)**

- Dataflow backwards in time are hazards

\[
\text{lw} \ $t0,0(\$t1) \\
\text{sub} \ $t3,\$t0,\$t2
\]

- Can’t solve all cases with forwarding
- Must stall instruction dependent on load, then forward (more hardware)

**Data Hazard: Loads (2/4)**

- Hardware stalls pipeline
- Called “interlock”

\[
\text{lw} \ $t0,0(\$t1) \\
\text{sub} \ $t3,\$t0,\$t2 \\
\text{and} \ $t5,\$t0,\$t4 \\
\text{or} \ $t7,\$t0,\$t6
\]
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

\[ \text{lw} \ t0, 0(t1) \]

\[ \text{nop} \]

\[ \text{sub} \ t3, t0, t2 \]

\[ \text{and} \ t5, t0, t4 \]

\[ \text{or} \ t7, t0, t6 \]
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

Administrivia

° Any administrivia?

° Advanced Pipelining!
  • “Out-of-order” Execution
  • “Superscalar” Execution
Pipeline Hazard: Matching socks in later load

A depends on D; stall since folder tied up; Note this is much different from processor cases so far. We have not had an earlier instruction depend on a later one.

Out-of-Order Laundry: Don’t Wait

A depends on D; rest continue; need more resources to allow out-of-order
Superscalar Laundry: Parallel per stage

More resources, HW to match mix of parallel tasks?

Superscalar Laundry: Mismatch Mix

Task mix underutilizes extra resources
Quiz
° Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards. $10^3$ iterations, so pipeline full.

Loop:
- lw $t0, 0($s1)
- addu $t0, $t0, $s2
- sw $t0, 0($s1)
- addiu $s1, $s1, -4
- bne $s1, $zero, Loop
- nop

° How many pipeline stages (clock cycles) per loop iteration to execute this code?

1 2 3 4 5 6 7 8 9 10

“And in Conclusion..”
° Pipeline challenge is 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:
  • Superscalar
  • Out-of-order execution