ECE 505 Computer Architecture

Pipelining 2

Berk Sunar and Thomas Eisenbarth
Review

• 5 stages of RISC
  • IF – ID – EX – MEM – WB

• Ideal speedup of pipelining = Pipeline depth (N)
• Practically
  • Implementation problems (slower clock)
  • Hazards (stalls)

\[
\text{Speedup} = \frac{\text{Pipeline depth (N)}}{1 + \text{Average number of stalls per instruction}}
\]
Review

• Type of hazards:
  • Structural Hazards
    Resource conflicts

• Data Hazards
  Result is needed before being written back

• Control Hazards
  Branches
Overview of Data Hazards

• Data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline

• Approaches to resolving data hazards
  • Schedule: Programmer explicitly avoids scheduling instructions that would create data hazards
  • Stall: Hardware includes control logic that freezes earlier stages until preceding instruction has finished producing data value
  • Bypass: Hardware datapath allows values to be sent to an earlier stage before preceding instruction has left the pipeline
  • Speculate: Guess that there is not a problem, if incorrect kill speculative instruction and restart
Review

- Through memory?

Presence of dependency is a property of ……
Generating hazards, and the number of stalls are properties of ……
Control Hazards

• Too many branches
  Basic Block in MIPS is typically 3~6 instructions

Basic Block is a straight-line code sequence with no branches

• Can we improve ILP across branches?
  Can we execute instructions without commit?
  • Preserve execution behavior
  • Preserve data flow

if p1 {
  S1;
  S2;
  S3;
};
Exception Behavior

• Preserving exception behavior: Changes do not raise new exceptions

```
ADD R2, R3, R4
BEQZ R2, L1
LD R1, 0(R2)
L1:
```

Problem with moving \texttt{LD} before \texttt{BEQZ}?
Data Flow

- Data flow:
  The flow of useful data between registers

  \[
  \begin{align*}
  \text{ADD} & \quad R1, R2, R3 \\
  \text{BEQZ} & \quad R4, L \\
  \text{SUB} & \quad R1, R5, R6 \\
  \text{L: ...} & \\
  \text{OR} & \quad R7, R1, R8
  \end{align*}
  \]

OR depends on ADD or SUB?
Dealing with Branches

1. Always stall
   1-stall in each branch

2. Predict Not-Taken
   Wrong guesses? turn instruction into no-op (after IF)

<table>
<thead>
<tr>
<th>Taken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch delay instruction (i + 1)</td>
<td>IF</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
</tr>
<tr>
<td>Instruction i + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i + 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i + 4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

   1 stall only on wrong guesses
Dealing with Branches

3. Predict Taken
   Not useful in 5-stages MIPS, why?
   
<table>
<thead>
<tr>
<th>Branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

   But, may be useful in more complex architectures

4. Delayed Branch
   Find a useful instruction to put in the delay
Dealing with Branches

From before

ADD R1, R2, R3
IF R2 = 0 then
STALL

becomes

IF R2 = 0 then
ADD R1, R2, R3

Perfect
Static or Dynamic?

From target

SUB R4, R5, R6

ADD R1, R2, R3
IF R1 = 0 then
STALL

becomes

SUB R4, R5, R6
ADD $1, $2, $3
IF $1 = 0 then
OR R7, R8, R9

Useful only if taken
Can be arranged by the compiler

Useful only if not-taken

From fall-through

ADD R1, R2, R3
IF R1 = 0 then
STALL

becomes

ADD R1, R2, R3
SUB R4, R5, R6

OR R7, R8, R9

SUB R4, R5, R6
Static Branch Prediction

![Bar chart showing misprediction rates for different benchmarks.](image)
Performance

\[
\text{Speedup} = \frac{\text{Pipeline depth (N)}}{1 + \text{Average number of stalls per instruction}}
\]

Avg # of stalls = Branch frequency \times Branch penalty

- Smaller branch frequency?
- Smaller branch penalty?
  - Resolve branch sooner, AND compute branch address
  - Use Zero test and a dedicated adder in the ID stage.
Simple Implementation of MIPS
Performance

• Example:

In MIPS R4000, the architecture works as follows:

<table>
<thead>
<tr>
<th>Branch scheme</th>
<th>Penalty unconditional</th>
<th>Penalty untaken</th>
<th>Penalty taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush pipeline</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Predicted taken</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Predicted untaken</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Compare always-stall, predict-taken, predict-untaken for a code with:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional branch</td>
<td>4%</td>
</tr>
<tr>
<td>Conditional branch,</td>
<td></td>
</tr>
<tr>
<td>untouched</td>
<td>6%</td>
</tr>
<tr>
<td>Conditional branch,</td>
<td></td>
</tr>
<tr>
<td>taken</td>
<td>10%</td>
</tr>
</tbody>
</table>

Why we didn’t consider delay-branches?
Performance

• Example:
  The branch penalty for MIPS R4000:

<table>
<thead>
<tr>
<th>Branch scheme</th>
<th>Unconditional branches</th>
<th>Untaken conditional branches</th>
<th>Taken conditional branches</th>
<th>All branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency of event</td>
<td>4%</td>
<td>6%</td>
<td>10%</td>
<td>20%</td>
</tr>
<tr>
<td>Stall pipeline</td>
<td>0.08</td>
<td>0.18</td>
<td>0.30</td>
<td>0.56</td>
</tr>
<tr>
<td>Predicted taken</td>
<td>0.08</td>
<td>0.18</td>
<td>0.20</td>
<td>0.46</td>
</tr>
<tr>
<td>Predicted untaken</td>
<td>0.08</td>
<td>0.00</td>
<td>0.30</td>
<td>0.38</td>
</tr>
</tbody>
</table>

Which one is the best?
Dynamic Branch Prediction

• Branch-prediction buffer / Branch history table
  A small cache for branch outcomes
  Managed at run-time (not compile time)

• 1-bit branch prediction

\[
\begin{align*}
\text{for } (i=0;i<3;i++) \\
&\quad s = s + x[i]; \\
\text{for } (i=0;i<3;i++) \\
&\quad y[i] = y[i] + s;
\end{align*}
\]

<table>
<thead>
<tr>
<th>Prediction</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
</tr>
<tr>
<td>Stall?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2-stalls for each miss-guess
Dynamic Branch Prediction

• 2-bit branch prediction

```latex
\begin{align*}
\text{for} \ (i=0; i<3; i++) \\
\ s &= s + x[i]; \\
\text{for} \ (i=0; i<3; i++) \\
\ y[i] &= y[i] + s;
\end{align*}
```

<table>
<thead>
<tr>
<th>Prediction</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
</tr>
<tr>
<td>Stall?</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-stall for each miss-guess
Dynamic Branch Prediction

- Dynamic Branch Prediction

![Bar chart showing the frequency of mispredictions for various SPEC89 benchmarks.](chart)

- nasa7: 1%
- matrix300: 0%
- tomcatv: 1%
- doduc: 5%
- spice: 9%
- fpppp: 9%
- gcc: 12%
- espresso: 5%
- eqntott: 18%
- li: 10%

Frequency of mispredictions
Improving ILP (more to come)

- Loop Unrolling
- Correlating Branch Prediction
- Tournament Prediction
- Dynamic Scheduling
  - Scoreboard
  - Tomasulo’s Algorithm
  - Reservation Stations
- Hardware Based Speculation
- Multiple Issue