Announcements: Spring break next week!
Project 2 due the week after spring break
HW3 due Monday after spring break
(no more homework until week 12)
Handouts: Handout #10 Project 2 (On Blackboard)

Terminology

- Dependencies
  - ordering requirement between instructions

- Pipeline Hazards:
  - (potential) violations of dependencies

- Hazard Resolution:
  - static ⇒ schedule instructions at compile time to avoid hazards
  - dynamic ⇒ detect hazard and adjust pipeline operation
    Stall, Forward/byapss, anything else?

- Pipeline Interlock:
  - hardware mechanisms for dynamic hazard resolution
  - detect and enforce dependences at run time
Forwarding Paths (v1)

Data Hazard Analysis (with Forwarding)

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>use</td>
</tr>
<tr>
<td>EX</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>produce</td>
<td>(use)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Even with data-forwarding, RAW dependence on an immediate preceding LW instruction produces a hazard
Why not very deep pipelines?

- 5-stage pipeline still has plenty of combinational delay between registers
- "Superpipelining" ⇒ increase pipelining such that even intrinsic operations (e.g. ALU, RF access, memory access) require multiple stages
- What's the problem?

\[
\text{Inst}_0: r_1 \leftarrow r_2 + r_3 \\
\text{Inst}_1: r_4 \leftarrow r_1 + 2
\]

Intel P4's Superpipelined Integer ALU

32-bit addition pipelined over 2 stages, BW=1/\text{latency}_{16\text{-bit-add}}

No stall between back-to-back dependencies
What if you really can't superpipeline?

If you can't double the bandwidth by pipelining, doubling the resource also doubles the bandwidth

Instruction Ordering/Dependencies

- **Data Dependence**
  - True dependence or Read after Write (RAW)
    Instruction must wait for all required input operands
  - Anti-Dependence or Write after Read (WAR)
    Later write must not clobber a still-pending earlier read
  - Output dependence or Write after Write (WAW)
    Earlier write must not clobber an already-finished later write

- **Control Dependence** (or Procedural Dependence)
  - all instructions are dependent by control flow
  - every instruction use and set the PC
    Control dependence is data dependence on the PC
PC Data Hazard Analysis

<table>
<thead>
<tr>
<th>R/I-</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</td>
</tr>
<tr>
<td>ID</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
<td>produce</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td>produce</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- PC hazard distance is at least 1
- Does that mean we must stall after every instruction?
  - IF stage can't know which PC to fetch next until the current PC is fetched and decoded

Control Hazard by Stalling

For non-control-flow instructions
Control Speculation for Dummies

- Rather than waiting for true-dependence on PC to resolve, just guess nextPC = PC+4 to keep fetching every cycle. Is this a good guess? What do you lose if you guessed incorrectly?

- Only ~20% of the instruction mix is control flow
  - ~50% of "forward" control flow (i.e., if-then-else) is taken
  - ~90% of "backward" control flow (i.e., loop back) is taken
    Over all, typically ~70% taken and ~30% not taken
    [Lee and Smith, 1984]

- Expect "nextPC = PC+4" ~86% of the time, but what about the remaining 14%?

Control Speculation: PC+4

When a branch resolves
- branch target (Inst_k) is fetched
- all instructions fetched since inst_h (so called "wrong-path" instructions) must be flushed
Pipeline Flush on Misprediction

Inst$_h$ is a branch

Pipeline Flush on Misprediction

<table>
<thead>
<tr>
<th></th>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
<th>$t_7$</th>
<th>$t_8$</th>
<th>$t_9$</th>
<th>$t_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>i</td>
<td>bub</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>h</td>
<td>bub</td>
<td>bub</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>h</td>
<td>bub</td>
<td>bub</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>h</td>
<td>bub</td>
<td>bub</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

branch resolved
Performance Impact

- correct guess ⇒ no penalty ~86% of the time
- incorrect guess ⇒ 2 bubbles

Assume
- no data hazards
- 20% control flow instructions
- 70% of control flow instructions are taken
- \( IPC = \frac{1}{1 + (0.20 \times 0.7) \times 2} = \frac{1}{1 + 0.14 \times 2} = 1 / 1.28 = 0.78 \)

probability of penalty for a wrong guess
Can we reduce either of the two penalty terms?

Reducing Mispredict Penalty

[Diagram of a processor showing control flow and data paths]
MIPS R2000 Control Flow Design

- Simple address calculation based on instruction only
  - Branch PC-offset: 16-bit full-addition + 14-bit half-addition
  - Jump PC-offset: concatenation only
- Simple branch condition based on RF
  - One register relative (> , < , =) to 0
  - Equality between 2 registers
  \text{No addition/subtraction necessary!}
- An explicit ISA design choice to enable branch resolution in ID of a 5-stage pipeline

Branch Resolved in ID

IPC = 1 / [ 1 + (0.2*0.7) * 1 ] = 0.88
Branch Delay Slots

Br \rightarrow L1

PC+4

L1 if taken else PC+8

- PC+4 is already in the pipeline
  - throwing PC+4 away cost 1 bubble
  - letting PC+4 finish to the end won’t hurt performance
- R2000 defined branches to have an architectural latency of 1 instruction
  - the instruction immediately after a branch is always executed
  - branch target takes effect on the 2nd instruction
  - if delay slot can always do useful work, effective IPC=1 without BTB or even a pipeline flush logic
  - ~80% of delay slots can be filled automatically by compilers

Filling Delay Slots by Static Reordering Transformation

reordering data independent (RAW, WAW, WAR) instructions does not change program

[Based on Figures from VLSI CAD, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
## Final Data Hazard Analysis

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td></td>
<td>use</td>
<td></td>
<td>use</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>use</td>
<td>use</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>produce</td>
<td>produce</td>
<td>(use)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- With forwarding, hazard distance is 0 except for RAW dependence on LW where it is 1
- Load delay slot semantics ensures a dependent instruction to be at least distance 2

## Final PC Hazard Analysis

<table>
<thead>
<tr>
<th>R/I-Type</th>
<th>LW</th>
<th>SW</th>
<th>Br</th>
<th>J</th>
<th>Jr</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</td>
<td>use</td>
</tr>
<tr>
<td>ID</td>
<td>(produce)</td>
<td>(produce)</td>
<td>(produce)</td>
<td>produce</td>
<td>produce</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Hazard distance on a taken branch is 1
- Again, branch delay slot semantics ensures a dependent instruction to be at least distance 2
- MIPS R2000 can be interlock-free!!
- Hazard distance is greater in your project, why?
Making a Better Guess

- For ALU instructions
  - can’t do better than guessing nextPC=PC+4
  - still tricky since must guess nextPC before the current instruction is fetched
- For Branch/Jump instructions
  - why not always guess in the taken direction since 70% correct
  - again, must guess nextPC before the branch instruction is fetched (but branch target is encoded in the instruction)
  ⇒ Must make a guess based only on the current fetch PC !!
  ⇒ Fortunately,
    - PC-offset branch/jump target is static
    - We are allowed to be wrong some of the time

The Locality Principle

- One’s recent past is a very good predictor of his/her near future.
- Temporal Locality: If you just did something, it is very likely that you will do the same thing again soon
  - since you are here today, there is a good chance you will be here again and again regularly
  - inverse is also true
- Spatial Locality: If you just did something, it is very likely you will do something similar/related
  - every time I find you in this room, you are probably sitting in the same seat
  - you are probably sitting near the same people
Branch Target Buffer (Oracle)

- BTB (Oracle)
  - a giant table indexed by PC
  - returns the guess for nextPC

- When encountering a PC for the first time, store in BTB
  - $PC + 4$ if ALU/LD/ST
  - $PC + \text{offset}$ if Branch or Jump
  - ?? if JR

- Effectively guessing branches are always taken
  
  
  $IPC = \frac{1}{1 + (0.20 \times 0.3) \times 2}$
  
  $= 0.89$