Control Hazards

- Control hazards
  - Must fetch post branch insns before branch outcome is known
  - Default: assume "not-taken" (at fetch, can’t tell if it’s a branch)

Branch Recovery

- Branch recovery: what to do when branch is taken
  - Flush insns currently in F/D and D/X (they’re wrong)
  - Replace with NOPs
    + Haven’t yet written to permanent state (RegFile, DMem)

Control Hazard Pipeline Diagram

- Control hazards indicated with c* (or not at all)
- Penalty for taken branch is 2 cycles

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>$3,0,1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bnez</td>
<td>$3,tag</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>$6,4($7)</td>
<td>c*</td>
<td>c*</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>W</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch Performance

- Again, measure effect on CPI (clock period is fixed)
- Back of the envelope calculation
  - Branch: 20%, load: 20%, store: 10%, other: 50%
  - 75% of branches are taken (why so many taken?)
- CPI if no branches = 1
- CPI with branches = 1 + 0.20*0.75*2 = 1.3
  - Branches cause 30% slowdown
- How do we reduce this penalty?
One Option: Fast Branches

- **Fast branch**: resolves in Decode stage, not Execute
  - Test must be comparison to zero or equality, no time for ALU
    - New taken branch penalty is only 1
    - Need additional comparison insns (slt) for complex tests
    - Must be able to bypass into decode now, too

Another Option: Delayed Branches

- **Delayed branch**: don’t flush insn immediately following
  - As if branch takes effect one insn later
  - ISA modification compiler accounts for this behavior
  - Insert insns independent of branch into branch delay slot(s)

Improved Branch Performance?

- Same parameters
  - Branch: 20%, load: 20%, store: 10%, other: 50%
  - 75% of branches are taken

- Fast branches
  - 25% of branches have complex tests that require extra insn
    - CPI = \( 1 + 0.20 \times 0.75 \times 1(\text{branch}) + 0.20 \times 0.25 \times 1(\text{extra insn}) = 1.2 \)

- Delayed branches
  - 50% of delay slots can be filled with insns, others need nops
    - CPI = \( 1 + 0.20 \times 0.75 \times 1(\text{branch}) + 0.20 \times 0.50 \times 1(\text{extra insn}) = 1.25 \)
    - **Bad idea: painful for compiler, gains are minimal**
    - E.g., delayed branches in SPARC architecture (Sun computers)

Dynamic Branch Prediction

- **Dynamic branch prediction**: guess outcome
  - Start fetching from guessed address
  - Flush on mis-prediction
Inside A Branch Predictor

- Two parts
  - **Target buffer**: maps PC to taken target
  - **Direction predictor**: maps PC to taken/not-taken
- What does it mean to "map PC"?
  - Use some PC bits as index into an array of data items (like Regfile)

More About "Mapping PCs"

- If array of data has \( N \) entries
  - Need \( \log(N) \) bits to index it
- Which \( \log(N) \) bits to choose?
  - Least significant \( \log(N) \) after the least significant 2, why?
    - LS 2 are always 0 (PCs are aligned on 4 byte boundaries)
    - Least significant change most often \( \rightarrow \) gives best distribution
- What if two PCs have same pattern in that subset of bits?
  - Called **aliasing**
    - We get a nonsense target (intended for another PC)
    - That’s OK, it’s just a guess anyway, we can recover if it’s wrong

Updating A Branch Predictor

- How do targets and directions get into branch predictor?
  - From previous instances of branches
    - Predictor "learns" branch behavior as program is running
      - Branch X was taken last time, probably will be taken next time
  - Branch predictor needs a write port, too (not in my ppt)
    - New prediction written only if old prediction is wrong

Types of Branch Direction Predictors

- Predict same as last time we saw this same branch PC
  - 1 bit of state per predictor entry (take or don’t take)
    - For what code will this work well? When will it do poorly?
  - Use 2-level saturating counter
    - 2 bits of state per predictor entry
      - 11, 10 = take, 01, 00 = don’t take
      - Why is this usually better?
  - And every other possible predictor you could think of!
    - ICQ: Think of other ways to predict branch direction
  - Dynamic branch prediction is one of most important problems in computer architecture
Branch Prediction Performance

- Same parameters
  - Branch: 20%, load: 20%, store: 10%, other: 50%
  - 75% of branches are taken

- Dynamic branch prediction
  - Assume branches predicted with 75% accuracy
  - CPI = 1 + 0.20*0.75*2 = 1.15

- Branch (esp. direction) prediction was a hot research topic
  - Accuracies now 90-95%

Pipelining And Exceptions

- Remember exceptions?
  - Pipelining makes them nasty

  - 5 instructions in pipeline at once
  - Exception happens, how do you know which instruction caused it?
    - Exceptions propagate along pipeline in latches
  - Two exceptions happen, how do you know which one to take first?
    - One belonging to oldest insn
    - When handling exception, you have to flush younger insns
    - Piggy-back on branch mis-prediction machinery to do this

- Just FYI – we’ll solve this problem in ECE 252

Pipeline Performance Summary

- Base CPI is 1, but hazards increase it

- Remember: nothing magical about a 5 stage pipeline
  - Pentium® (first batch) had 20 stage pipeline

- Increasing pipeline depth (#stages)
  - Reduces clock period (that’s why companies do it)
    - But increases CPI
  - Branch mis-prediction penalty becomes longer
  - More stages between fetch and whenever branch computes
  - Non-bypassed data hazard stalls become longer
    - More stages between register read and write
  - At some point, CPI losses offset clock gains, question is when?

Instruction-Level Parallelism (ILP)

- Pipelining: a form of instruction-level parallelism (ILP)
  - Parallel execution of insns from a single sequential program

- There are ways to exploit ILP
  - We’ll discuss this a bit more at end of semester, and then we’ll really cover it in great depth in ECE 252

- We’ll also talk a bit about thread-level parallelism (TLP) and how it’s exploited by multithreaded and multicore processors
Summary

• Principles of pipelining
  • Pipelining a datapath and controller
  • Performance and pipeline diagrams

• Data hazards
  • Software interlocks and code scheduling
  • Hardware interlocks and stalling
  • Bypassing

• Control hazards
  • Branch prediction

Next up: Memory Systems (caches and main memory)