WAR: Write After Read

write-after-read (WAR) = artificial (name) dependence

(add R1, R2, R3)
(sub R2, R4, R1)
(or R1, R6, R3)

- **problem**: add could use wrong value for R2
- can’t happen in vanilla pipeline (reads in ID, writes in WB)
  - can happen if: early writes (e.g., auto-increment) + late reads (??)
  - can happen if: out-of-order reads (e.g., out-of-order execution)
- **artificial**: using different output register for sub would solve
  - The dependence is on the name R2, but not on actual dataflow
WAW: Write After Write

write-after-write (WAW) = artificial (name) dependence

- `add R1, R2, R3`
- `sub R2, R4, R1`
- `or R1, R6, R3`

- **problem**: reordering could leave wrong value in `R1`
  - later instruction that reads `R1` would get wrong value

- can’t happen in vanilla pipeline (register writes are in order)
  - another reason for making ALU ops go through MEM stage
  - can happen: multi-cycle operations (e.g., FP ops, cache misses)

- **artificial**: using different output register for `or` would solve
  - Also a dependence on a name: `R1`
RAR: Read After Read

read-after-read (RAR)

add $R1, R2, R3$
sub $R2, R4, R$
or $R1, R6, R3$

• no problem: $R3$ is correct even with reordering
Memory Data Hazards

have seen register hazards, can also have *memory hazards*

<table>
<thead>
<tr>
<th>RAW</th>
<th>WAR</th>
<th>WAW</th>
</tr>
</thead>
<tbody>
<tr>
<td>store R1, 0(SP)</td>
<td>load R4, 0(SP)</td>
<td>store R1, 0(SP)</td>
</tr>
<tr>
<td>load R4, 0(SP)</td>
<td>store R1, 0(SP)</td>
<td>store R4, 0(SP)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<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>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
</tr>
</tbody>
</table>

- in simple pipeline, memory hazards are easy
  - in-order
  - one at a time
  - read & write in same stage

- in general, though, more difficult than register hazards
Hazards vs. Dependences

dependence: fixed property of instruction stream (i.e., program)

hazard: property of program and processor organization

- implies potential for executing things in wrong order
  - potential only exists if instructions can be simultaneously “in-flight”
  - property of dynamic distance between instructions vs. pipeline depth

For example, can have RAW dependence with or without hazard
- depends on pipeline
Control Hazards

when an instruction affects *which* instruction executes next

\[
\begin{align*}
\text{store } R4, 0(R5) \\
\text{bne } R2, R3, \text{loop} \\
\text{sub } R1, R6, R3
\end{align*}
\]

• naive solution: stall until outcome is available (end of EX)
  + simple
  – low performance (2 cycles here, longer in general)
• e.g. 15% branches * 2 cycle stall ⇒ 30% CPI increase!
Control Hazards: “Fast” Branches

*Fast branches*: can be evaluated in ID (rather than EX)
+ reduce stall from 2 cycles to 1

- requires more hardware
  - dedicated ID adder for (PC + immediate) targets

- requires simple branch instructions
  - no time to compare two registers (would need full ALU)
  - comparisons with 0 are fast (beqz, bnez)
Control Hazards: Delayed Branches

delayed branch: execute next instruction whether taken or not
• instruction after branch said to be in “delay slot”
• old microcode trick stolen by RISC (MIPS)

```
store R4, 0(R5)
bne R2, R3, loop
sub R1, R6, R6
  bned R2, R3, loop
  store R4, 0(R5)
  sub R1, R6, R6
```

```
1 2 3 4 5 6 7 8 9
F D X M W
F D X M W
c* F D X M W
```
What To Put In Delay Slot?

- instruction from before branch
  - when? if branch and instruction are independent
  - helps? always

- instruction from target (taken) path
  - when? if safe to execute, but may have to duplicate code
  - helps? on taken branch, but may increase code size

- instruction from fall-through (not-taken) path
  - when? if safe to execute
  - helps? on not-taken branch

- upshot: short-sighted ISA feature
  - not a big win for today’s machines (why? consider pipeline depth)
  - complicates interrupt handling (later)
Control Hazards: Speculative Execution

idea: doing anything is often better than doing nothing

- **speculative execution**
  - guess branch target ⇒ start executing at guessed position
  - execute branch ⇒ verify (check) guess
  + minimize penalty if guess is right (to zero?)
  - wrong guess could be worse than not guessing

- **branch prediction**: guessing the branch
  - one of the “important” problems in computer architecture
  - very heavily researched area in last 15 years
  - static: prediction by compiler
  - dynamic: prediction by hardware
  - hybrid: compiler hints to hardware predictor
The Speculation Game

*speculation*: engagement in risky business transactions on the chance of quick or considerable profit

- **speculative execution (control speculation)**
  - execute before all parameters known with certainty

+ correct speculation
  + avoid stall/get result early, performance improves

– incorrect speculation (*mis-speculation*)
  – must abort/squash incorrect instructions
  – must undo incorrect changes (recover pre-speculation state)

• **the speculation game**: profit > penalty
  • profit = speculation accuracy * correct-speculation gain
  • penalty = (1–speculation accuracy) * mis-speculation penalty
Speculative Execution Scenarios

- **Correct speculation**

  - **Cycle 1:** Fetch branch, predict next (inst8)
  - **Cycle 2, 3:** Fetch inst8, inst9
  - **Cycle 3:** Execute/verify branch $\implies$ correct
  - Nothing needs to be fixed or changed

- **Incorrect speculation: mis-speculation**

  - **Cycle 1:** Fetch branch, predict next (inst1)
  - **Cycle 2, 3:** Fetch inst1, inst2
  - **Cycle 3:** Execute/verify branch $\implies$ wrong
  -  $c_3$: Send correct target to IF (inst8)
  -  $c_3$: Squash (abort) inst1, inst2 (flush F/D)
  -  $c_4$: Fetch inst8
Static (Compiler) Branch Prediction

Some static prediction options

- **predict always not-taken**
  + very simple, since we already know the target (PC+4)
  - majority of branches (~65%) are taken (why?)

- **predict always taken**
  + better performance
  - more difficult, must know target before branch is decoded

- **predict backward taken**
  - most backward branches are taken

- **predict specific opcodes taken**

- **use profiles to predict on per-static branch basis**
  - pretty good
Comparison of Some Static Schemes

CPI-\text{penalty} = \%_{\text{branch}} \times [\%_T \times \text{penalty}_T + \%_{\text{NT}} \times \text{penalty}_{\text{NT}}]

- simple branch statistics
  - 14\% PC-changing instructions ("branches")
  - 65\% of PC-changing instructions are "taken"

<table>
<thead>
<tr>
<th>scheme</th>
<th>penalty_T</th>
<th>penalty_{NT}</th>
<th>CPI penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>stall</td>
<td>2</td>
<td>2</td>
<td>0.28</td>
</tr>
<tr>
<td>fast branch</td>
<td>1</td>
<td>1</td>
<td>0.14</td>
</tr>
<tr>
<td>delayed branch</td>
<td>1.5</td>
<td>1.5</td>
<td>0.21</td>
</tr>
<tr>
<td>not-taken</td>
<td>2</td>
<td>0</td>
<td>0.18</td>
</tr>
<tr>
<td>taken</td>
<td>0</td>
<td>2</td>
<td>0.10</td>
</tr>
</tbody>
</table>
Dynamic Branch Prediction

hardware (BP) guesses whether and where a branch will go

\[
\begin{align*}
0x64 & \quad \text{bnez r1, #10} \\
0x74 & \quad \text{add r3, r2, r1}
\end{align*}
\]

- start with branch PC (0x64) and produce
  - direction (Taken)
  - direction + target PC (0x74)
  - direction + target PC + target instruction (\text{add r3, r2, r1})
Branch History Table (BHT)

branch PC ⇒ prediction (T, NT)
- need decoder/adder to compute target if taken
  - *branch history table (BHT)*
    - read prediction with least significant bits (LSBs) of branch PC
    - change bit on misprediction
      + simple
      - multiple PCs may map to same bit (aliasing)
  - major improvements
    - two-bit counters [Smith]
    - correlating/two-level predictors [Patt]
    - hybrid predictors [McFarling]
Improvement: Two-bit Counters

example: 4-iteration inner loop branch

<table>
<thead>
<tr>
<th>state/prediction</th>
<th>N</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch outcome</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>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>mis-prediction?</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

- problem: two mis-predictions per loop
  - solution: 2-bit saturating counter to implement hysteresis
    - 4 states: strong/weak not-taken (N/n), strong/weak taken (T/t)
    - transitions: N ↔ n ↔ t ↔ T

<table>
<thead>
<tr>
<th>state/prediction</th>
<th>n</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>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch outcome</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>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>mis-prediction?</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

+ only one mis-prediction per iteration