Dependencies

- Problem with starting next instruction before first is finished
  - dependencies that “go backward in time” are data hazards

Time (in clock cycles)

<table>
<thead>
<tr>
<th></th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/–20</td>
<td>–20</td>
<td>–20</td>
<td>–20</td>
<td>–20</td>
</tr>
<tr>
<td>DM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Value of register $2$: 10

Software Solution

- Have compiler guarantee no hazards
- Where do we insert the “nops”?

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

- Problem: this really slows us down!

Forwarding

- Use temporary results, don’t wait for them to be written
  - register file forwarding to handle read/write to same register
  - ALU forwarding

Value of register $2$: 10

Forwarding

- The main idea (some details not shown)
Can't always forward

- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register.

- Thus, we need a hazard detection unit to "stall" the load instruction

Hazard Detection Unit

- Stall by letting an instruction that won't write anything go forward

Stalling

- We can stall the pipeline by keeping an instruction in the same stage

Branch Hazards

- When we decide to branch, other instructions are in the pipeline!

- We are predicting "branch not taken" — need to add hardware for flushing instructions if we are wrong
Flush Instructions

- If the branch is taken, we have a penalty of one cycle
- For our simple design, this is reasonable
- With deeper pipelines, penalty increases and static branch prediction drastically hurts performance
- Solution: dynamic branch prediction

Branch Prediction

- Sophisticated Techniques:
  - A “branch target buffer” to help us look up the destination
  - Correlating predictors that base prediction on global behavior and recently executed branches (e.g., prediction for a specific branch instruction based on what happened in previous branches)
  - Tournament predictors that use different types of prediction strategies and keep track of which one is performing best.
  - A “branch delay slot” which the compiler tries to fill with a useful instruction (make the one cycle delay part of the ISA)
- Branch prediction is especially important because it enables other more advanced pipelining techniques to be effective!
- Modern processors predict correctly 95% of the time!

Improving Performance

- Try and avoid stalls! E.g., reorder these instructions:
  - \texttt{lw $t0, 0($t1)}
  - \texttt{lw $t2, 4($t1)}
  - \texttt{sw $t2, 0($t1)}
  - \texttt{sw $t0, 4($t1)}
- Dynamic Pipeline Scheduling
  - Hardware chooses which instructions to execute next
  - Will execute instructions out of order (e.g., doesn’t wait for a dependency to be resolved, but rather keeps going!)
  - Speculates on branches and keeps the pipeline full (may need to rollback if prediction incorrect)
- Trying to exploit instruction-level parallelism
Advanced Pipelining

- Increase the depth of the pipeline
- Start more than one instruction each cycle (multiple issue)
- Loop unrolling to expose more ILP (better scheduling)
- "Superscalar" processors
  - DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
- All modern processors are superscalar and issue multiple instructions usually with some limitations (e.g., different "pipes")
- VLIW: very long instruction word, static multiple issue (relies more on compiler technology)

- This class has given you the background you need to learn more!

Chapter 6 Summary

- Pipelining does not improve latency, but does improve throughput

Memories: Review

- SRAM:
  - value is stored on a pair of inverting gates
  - very fast but takes up more space than DRAM (4 to 6 transistors)
- DRAM:
  - value is stored as a charge on capacitor (must be refreshed)
  - very small but slower than SRAM (factor of 5 to 10)
Users want large and fast memories!

- SRAM access times are 0.5 – 5ns at cost of $4000 to $10,000 per GB. 
- DRAM access times are 50-70ns at cost of $100 to $200 per GB. 
- Disk access times are 5 to 20 million ns at cost of $.50 to $2 per GB.

Try and give it to them anyway
  - build a memory hierarchy