Pipelining II

Instructor: Dmitri A. Gusev

Fall 2007

CS 502: Computers and Communications Technology

Lecture 9, October 3, 2007
Dependencies

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

<table>
<thead>
<tr>
<th>Time (in clock cycles)</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>Value of register $2$:</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>
</tbody>
</table>

Program execution order (in instructions)
- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)
Software Solution

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

```assembly
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

```
Time (in clock cycles)

<table>
<thead>
<tr>
<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>Value of register $2$:</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>
</tr>
<tr>
<td>Value of EX/MEM:</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Value of MEM/WB:</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
```

Program execution order (in instructions)

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

what if this $2 was $13?
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.
Stalling

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

Program execution order (in instructions)

lw $2, 20($1)
and becomes nop
add $4, $2, $5
or $8, $2, $6
add $9, $4, $2
Hazard Detection Unit

- Stall by letting an instruction that won’t write anything go forward
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
Flushing Instructions

Note: we’ve also moved branch decision to ID stage
Branches

- 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

A 2-bit prediction scheme
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:

```assembly
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
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)
Chapter 6 Summary

- Pipelining does not improve latency, but does improve throughput