Pipelining: Its Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes
Sequential Laundry

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?
Pipelined Laundry: Start work ASAP

- Pipelined laundry takes 3.5 hours for 4 loads
Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
A simple RISC pipeline

RISC subset instruction steps

<table>
<thead>
<tr>
<th></th>
<th>R-type</th>
<th>Store/Load</th>
<th>Branch/Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Instruction fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>Instruction decode and register fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>Execution</td>
<td>Compute addr</td>
<td>Instr completion</td>
</tr>
<tr>
<td>MEM</td>
<td>Instr completion</td>
<td>Mem access</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td>Load completion</td>
<td></td>
</tr>
</tbody>
</table>

5 Stage Pipeline
Illustrating the datapath visually

Can help with answering questions like:

- # cycles to execute this code?
- Resource conflicts?
Keeping instructions from interfering with each other

Additional registers to hold stage values
Amdahl’s Law Strikes Again

• Unpipelined multi-cycle implementation
  - 1 ns clock cycle
  - Branches and ALU ops take 4 cycles
  - Memory ops take 5 cycles
  - Instruction mix: 20% branch, 40% ALU, 40% mem
  - Avg instr exec time = 1 ns + .6 * 4 + .4 * 5 = 4.4 ns

• 5 stage pipeline
  - .2 ns overhead for clock skew and register delay
  - Each stage takes 1 ns clock cycle
  - Avg instr exec time = 1.2 ns

• Speedup: 4.4/1.2=3.7 Overhead limits speedup!
Speedup in Pipelining

\[
Speedup = \frac{\text{Avg instr time unpipeline}}{\text{Avg instr time pipeline}}
\]

\[
Speedup = \frac{\text{CPI}_{\text{unpipeline}}}{\text{CPI}_{\text{pipeline}}} \times \frac{\text{clock}_{\text{unpipeline}}}{\text{clock}_{\text{pipeline}}}
\]

\[
\text{CPI}_{\text{ideal}} = \frac{\text{CPI}_{\text{unpipeline}}}{\text{Pipeline depth}} = 1
\]

\[
\text{CPI}_{\text{pipeline}} = \text{CPI}_{\text{ideal}} + \text{Pipeline stalls per instr}
\]

\[
Speedup = \frac{\text{CPI}_{\text{unpipeline}}}{1 + \text{pipeline stalls}} \approx \frac{\text{pipeline depth}}{1 + \text{pipeline stalls}}
\]
Project 3: Simplescalar

- Assigned: 30 Oct
- Due: 20 Nov, late submission 25 Nov
- Sorry, no further extensions...
- Handouts will be available soon!

5.10 [30/30] <3, 4, 5> Use an out-of-order, superscalar simulator such as SimpleScalar (www.cs.wisc.edu/~mscalar/simplescalar.html) and a simple benchmark for the following:

a. [30] <3, 4, 5> Run the benchmark and find the hit rates for a memory system with a level 1 cache ranging from 64 KB to 256 KB and a level 2 cache ranging from 512 KB to 4 MB assuming that instructions are executed in order and issued at most one per cycle (in other words, the CPU is not really out of order and superscalar).

b. [30] <3, 4, 5> Run the same program on the same range of memory systems on an out-of-order, four-way instruction issue processor with two integer units, one floating-point unit, and one memory unit. How do the hit rates vary from the in-order case?
Pipelining Difficulties

• What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores

• What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction
  - Control
  - Exceptions
  - Multi-cycle operations
Hazards in Pipelining

- **Limits to pipelining:**
  - Hazards prevent next instruction from executing during its designated clock cycle
- **Structural hazards:**
  - HW cannot support this combination of instructions
- **Data hazards:**
  - Instruction depends on result of prior instruction still in the pipeline
- **Control hazards:**
  - Pipelining of branches & other instructions that change the PC
- **Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline**
Structural Hazards

- Problem with starting instruction 3 before load is finished
  - Processor with only single memory port generates conflicts
Structural Hazards

- Insert stall cycles to avoid problem
  - Problem solved at cost in performance

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load instruction</td>
<td>1</td>
</tr>
<tr>
<td>Instruction i + 1</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction i + 2</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction i + 3</td>
<td>IF</td>
</tr>
<tr>
<td></td>
<td>stall</td>
</tr>
<tr>
<td>Instruction i + 4</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction i + 5</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction i + 6</td>
<td>IF</td>
</tr>
</tbody>
</table>

What is CPI?

What role do cache’s play?
Data Hazards

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

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

- When we decide to branch, other instructions are in the pipeline!
- We are predicting “branch not taken”
Four Compile-time Branch Hazard Alternatives

- Stall until branch direction is clear
  - branch cost fixed, not reducible by software (compiler)

- Predict Branch Not Taken
  - Slightly more complex ("backing-out"), higher performance
  - Execute successor instructions in sequence
  - "Squash" instructions in pipeline if branch actually taken
  - PC+4 already calculated, so use it to get next instruction

- Pipeline freeze or flush
Four Compile-time Branch Hazard Alternatives

- **Predict Branch Taken**
  - Usefulness depends on system characteristics
  - In 5-stage MIPS subset, not useful since target address not available until same stage as comparison (1 cycle stall needed)
  - Other machines: branch target known before outcome

- **Delayed Branch**
  - Define branch to take place AFTER a following instruction

<table>
<thead>
<tr>
<th>branch instruction</th>
<th>branch target if taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untaken branch instruction</td>
<td>IF</td>
</tr>
<tr>
<td>Branch delay instruction ((i + 1))</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction (i + 2)</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction (i + 3)</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction (i + 4)</td>
<td>IF</td>
</tr>
<tr>
<td>Taken branch instruction</td>
<td>IF</td>
</tr>
<tr>
<td>Branch delay instruction ((i + 1))</td>
<td>IF</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
</tr>
<tr>
<td>Branch target + 1</td>
<td>IF</td>
</tr>
<tr>
<td>Branch target + 2</td>
<td>IF</td>
</tr>
</tbody>
</table>
Delayed Branch

• Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - Following compile-time branch prediction:
    • From the target address: only valuable when branch taken
    • From fall through: only valuable when branch not taken

Compiler effectiveness for single branch delay slot:
1) Restriction of instructions scheduled in delay slots
2) Compile-time ability to predict branch outcome

*Support for canceling branch instructions allows compilers to fill more delay slots - hw simply squashes mispredicted instructions
Compiler “Static” Prediction of Taken/Untaken Branches

- Improves strategy for placing instructions in delay slot
- Two strategies
  - Backward branch predict taken, forward branch not taken
  - Profile-based prediction: record branch behavior, predict branch based on prior run
Evaluating Static Branch Prediction Strategies

- Misprediction ignores frequency of branch
- “Instructions between mispredicted branches” is a better metric
Example

- 3 stages before branch-target known
  - 2 cycle penalty on all uncond since target unknown
- 4 stages before condition evaluated
  - Penalty depends on prediction scheme
- Branch penalties for 3 schemes:

<table>
<thead>
<tr>
<th>Branch scheme</th>
<th>Penalty unconditional</th>
<th>Penalty untaken</th>
<th>Penalty taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush pipeline</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Predicted taken</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Predicted untaken</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

- Assume 4% uncond, 6% cond untaken, 10% cond taken
- Quantify effect of branches on CPI
- Solution:
## Revised Unpipelined Version of MIPS

<table>
<thead>
<tr>
<th>R-type</th>
<th>Store/Load</th>
<th>Branch/Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Instruction fetch</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>Instruction decode and register fetch</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>Execution</td>
<td>Compute addr</td>
</tr>
<tr>
<td>MEM</td>
<td>Instr completion</td>
<td>Mem access</td>
</tr>
<tr>
<td>WB</td>
<td>Load completion</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R-R or R-I</th>
<th>Store/Load</th>
<th>Branch (BEQZ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IR ← Mem[PC], NPC ← PC+4</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>A ← Regs[rs], B ← Regs[rt], Imm ← sign-ext immed of IR</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>ALUOutput ← A op Imm, ALUOutput ← A func B</td>
<td>ALUOutput ← A+Imm</td>
</tr>
<tr>
<td>MEM</td>
<td>PC ← NPC</td>
<td>PC ← NPC, LMD ← Mem[ALUOutput] OR Mem[ALUOutput] ← B</td>
</tr>
</tbody>
</table>
Revised MIPS Datapath
Pipelined Datapath
## Events by Pipeline Stage

<table>
<thead>
<tr>
<th>Stage</th>
<th>Any instruction</th>
<th>ALU instruction</th>
<th>Load or store instruction</th>
<th>Branch instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) &amp; EX/MEM.cond){EX/MEM.ALUOutput} else {PC+4});</td>
<td>EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm;</td>
<td>EX/MEM.IR ← ID/EX.IR. EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B ← ID/EX.B;</td>
<td>EX/MEM.IALUOutput ← ID/EX.NPC + (ID/EX.Imm &lt;&lt; 2); EX/MEM.cond ← (ID/EX.A == 0);</td>
</tr>
<tr>
<td>ID</td>
<td>ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;</td>
<td>MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;</td>
<td>For load only: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>