Chapter 3 (and Appendix C)

Instruction-Level Parallelism and Its Exploitation
In the beginning…

- Processors fetched and finished execution of one instruction in a single cycle:
  - Simple architectures (RISC, anyway)
  - Relatively low latency
  - Relatively low throughput

- Later, processors were pipelined to improve throughput:
  - Shorter cycle times
  - More complicated architectures
  - Increased throughput (but also latency)
MIPS Pipeline

- 5 stages:
  - Instruction Fetch
  - Instruction Decode
  - Execution
  - Memory
  - Writeback
MIPS Pipeline
MIPS Pipeline
MIPS Pipeline
Pipeline Hazards

- Types of pipeline hazards:
  - Structural
  - Data
  - Control
Structural Hazards

- Two stages can’t use the same hardware at the same time
- Solution: Separate instruction and data caches, multiport registers that can be read and written in one cycle
Data Hazards

- An instruction requires that data has been processed by a previous instruction that has not yet completed
- Solution: “Forward” execution and memory-load results immediately to earlier stages in the pipeline so that subsequent instructions can access the data as soon as it’s ready
- Still can’t handle a load followed immediately by an instruction which uses the loaded data for computation
  - Causes a single-cycle stall
Data Hazards

Program execution order (in instructions)

DADD R1, R2, R3

LD R4, 0(R1)

SD R4, 12(R1)
Data Hazards
Control Hazards

- Processor loads an incorrect instruction because previous branch instruction has not yet completed

- Solution: Allow compiler to schedule an unrelated instruction just after the branch instruction and allow that instruction to complete whether or not the branch is taken
  - Stall for one cycle if no such instruction is provided
  - Can use a similar technique for load instructions
Exploiting Pipelining

- Pipelining became universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”

- When exploiting instruction-level parallelism, goal is to minimize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls per instruction +
    - Data hazard stalls per instruction +
    - Control stalls per instruction
Instruction-Level Parallelism

- There are two main approaches to ILP:
  - Compiler-based static approaches
    - Does not require complex hardware
    - Works well for loop-heavy scientific applications
  - Hardware-based dynamic approaches
    - Requires complex hardware
    - Used in server and desktop processors
    - Not used as extensively in PMD processors

- Modern processors have multiple execution units with varying latencies
  - Integers units faster than floating-point units
Instruction-Level Parallelism
Instruction-Level Parallelism
Instruction-Level Parallelism

- Programs can be thought of as blocks of execution
  - A “basic block” is a sequence of instructions containing no branches

- Parallelism with basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches
Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Use expected instruction latencies to separate execution of instructions

- Example:
  
  ```c
  for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
  ```

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
Pipeline Stalls

Loop:

- L.D F0,0(R1)
- stall
- ADD.D F4,F0,F2
- stall
- stall
- S.D F4,0(R1)
- DADDUI R1,R1,#-8
  - stall (assume integer op latency is 1)
- BNE R1,R2,Loop

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
Pipeline Scheduling

Scheduled code:
Loop: L.D F0,0(R1)
      DADDUI R1,R1,#-8
      ADD.D F4,F0,F2
      stall
      stall
      S.D F4,8(R1)
      BNE R1,R2,Loop

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
Loop Unrolling

- Loop unrolling
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

Loop:
L.D F0,0(R1)
ADD.D F4,F0,F2
S.D F4,0(R1) ;drop DADDUI & BNE
L.D F6,-8(R1)
ADD.D F8,F6,F2
S.D F8,-8(R1) ;drop DADDUI & BNE
L.D F10,-16(R1)
ADD.D F12,F10,F2
S.D F12,-16(R1) ;drop DADDUI & BNE
L.D F14,-24(R1)
ADD.D F16,F14,F2
S.D F16,-24(R1)
DADDUI R1,R1,#-32
BNE R1,R2,Loop

- note: stalls not included
- notice: number of live registers vs. original loop
Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop to remove stalls:

  Loop:  
  
  L.D F0,0(R1)  
  L.D F6,-8(R1)  
  L.D F10,-16(R1)  
  L.D F14,-24(R1)  
  ADD.D F4,F0,F2  
  ADD.D F8,F6,F2  
  ADD.D F12,F10,F2  
  ADD.D F16,F14,F2  
  S.D F4,0(R1)  
  S.D F8,-8(R1)  
  DADDUI R1,R1,#-32  
  S.D F12,16(R1)  
  S.D F16,8(R1)  
  BNE R1,R2,Loop
Strip Mining

- Unknown number of loop iterations?
  - Number of iterations = \( n \)
  - Goal: make \( k \) copies of the loop body
  - Generate pair of loops:
    - First executes \( n \mod k \) times
    - Second executes \( n / k \) times
    - “Strip mining”
Branch Prediction

- Control hazards are more difficult to handle in deeper pipelines
- Predict branch, flush pipeline if incorrect
- Simple techniques:
  - Predict never taken
  - Predict always taken
  - One-bit predictor
    - Predict last outcome
  - Two-bit saturating predictor
    - Flip prediction after two incorrect predictions
2-Bit Saturating Predictor
Branch Prediction

- Local predictor:
  - Allocate array of 2-bit saturating predictors
  - Use low-order bits of PC to select predictor

- Global predictor
  - Allocate array of 2-bit saturating predictors
  - Use last $n$ outcomes to select predictor
  - $(n, m)$ predictor uses $n$ global bits to select an $m$-bit predictor

- Correlating predictor:
  - Allocate array of 2-bit saturating predictors
  - Use local and global outcomes to select predictor
  - $(n, m)$ correlating predictor with $k$ local bits uses $n + k$ bits to select from $2^{(n + k)}$ $m$-bit predictors

- Tournament predictor:
  - Uses a predictor to select between multiple predictor types
Branch Prediction Performance

SPEC89 benchmarks

- nasa7: 1% 0%
- matrix300: 0% 0%
- tomcatv: 1% 0%
- doduc: 5% 5%
- spice: 9% 9%
- fpppp: 9% 9%
- gcc: 12% 11%
- espresso: 5% 5%
- eqntott: 18% 18%
- li: 10% 10%

4096 entries: 2 bits per entry
Unlimited entries: 2 bits per entry

Frequency of mispredictions
Branch Prediction Performance

Branch predictor performance

Conditional branch misprediction rate vs. Total predictor size for Local 2-bit predictors, Correlating predictors, and Tournament predictors.
Core i7 Branch Predictor

- Mostly confidential
- Two-level predictor
  - Small, simple first-level predictor which can make a prediction each cycle
  - Larger second-level tournament predictor which serves as a backup
    - Combines a 2-bit predictor, a global predictor, and a loop-exit predictor which predicts the number of loop iterations
Pitfalls

- Unexpected instructions can cause unexpected hazards
- Extensive pipelining can impact other aspects of design, leading to overall worse cost-performance