Lecture 5: Pipelining & Instruction Level Parallelism

Professor Alvin R. Lebeck
Computer Science 220
Fall 2001

Todo

- Homework #1 Due
- Read Chapter 4
- Homework #2 Due September 25
- Project selection by October 1

Today
- Finish simple pipelining
- Start ILP and Scheduling to expose ILP

Review: Hazards

Data Hazards
- RAW
  - only one that can occur in current DLX pipeline
- WAR, WAW
- Data Forwarding (Register Bypassing)
  - send data from one stage to another bypassing the register file
- Still have load use delay

Structural Hazards
- Replicate Hardware, scheduling

Control Hazards
- Compute condition and target early (delayed branch)

Review: Speed Up Equation for Pipelining

CPI_{pipelined} = \frac{\text{Ideal CPI} + \text{Pipeline stall clock cycles per instr}}{\text{Pipeline depth}}

\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}

Review: Evaluating Branch Alternatives

- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier

Pipeline speedup = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}

<table>
<thead>
<tr>
<th>Scheduling scheme</th>
<th>Branch penalty</th>
<th>CPI speedup v. unpipelined</th>
<th>speedup v. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>3</td>
<td>1.42</td>
<td>1.0</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.14</td>
<td>1.26</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1</td>
<td>1.09</td>
<td>1.29</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.07</td>
<td>1.31</td>
</tr>
</tbody>
</table>

Pipeline Complications

- Complex Addressing Modes and Instructions
- Address modes: Autoincrement causes register change during instruction execution
  - Interrupts? Need to restore register state
  - Adds WAR and WAW hazards since writes no longer last stage
- Memory-Memory Move Instructions
  - Must be able to handle multiple page faults
  - Long-lived instructions: partial state save on interrupt
- Condition Codes
Pipeline Complications: Floating Point

Floating Point RAW Stalls

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RAW Stalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F4, 0(r2)</td>
<td></td>
</tr>
<tr>
<td>MULTD F0, F4, F6</td>
<td></td>
</tr>
<tr>
<td>ADDD F2, F0, F8</td>
<td></td>
</tr>
<tr>
<td>SD F2, 0(R2)</td>
<td></td>
</tr>
</tbody>
</table>

Floating Point Structural Hazard

<table>
<thead>
<tr>
<th>Instruction</th>
<th>WB Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTD F0, F4, F6</td>
<td></td>
</tr>
<tr>
<td>ADDD F2, F0, F8</td>
<td></td>
</tr>
<tr>
<td>LD F8, 0(R2)</td>
<td></td>
</tr>
</tbody>
</table>

Hazard Detection

- Assume all hazard detection in ID stage
- Check for structural hazards.
- Check for RAW data hazard.
- Check for WAW data hazard.

- If any occur stall at ID stage
- This is called an in-order issue/execute machine, if any instruction stalls all later instructions stall.
  - Note that instructions may complete execution out of order.

Pipelining Complications

- Floating Point: long execution time
- Also, may pipeline FP execution unit so we can initiate new instructions without waiting full latency

<table>
<thead>
<tr>
<th>FP Instruction</th>
<th>Latency</th>
<th>Initiation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, Subtract</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Multiply</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Divide</td>
<td>36</td>
<td>35</td>
</tr>
<tr>
<td>Square root</td>
<td>112</td>
<td>111 (interrupts, WAW, WAR)</td>
</tr>
<tr>
<td>Negate</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Absolute value</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>FP compare</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Cycles before use result Cycles before issue instr of same type

Case Study: MIPS R4000

- 8 Stage Pipeline:
  - IF—first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS—second half of access to instruction cache.
  - RF—instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX—execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF—data fetch, first half of access to data cache.
  - DS—second half of access to data cache.
  - TC—tag check, determine whether the data cache access hit.
  - WB—write back for loads and register-register operations.

- 8 Stages: What is impact on Load delay? Branch delay? Why?
Case Study: MIPS R4000

TWO Cycle
Load Latency
IF IS RF EX DF IF IS IS RF EX DF IF IS IS RF EX DF

THREE Cycle
Branch Latency
IF IS RF EX DF DS TC IF IS IS RF EX DF IF IS IS RF EX DF

Branch likely cancels delay slot if not taken

MIPS R4000 Floating Point

• FP Adder, FP Multiplier, FP Divider
• Last step of FP Multiplier/Divider uses FP Adder HW

8 kinds of stages in FP units:
- Stage
- Functional unit
- Description
- A
- FP adder
- Mantissa ADD stage
- D
- FP divider
- Divide pipeline stage
- E
- FP multiplier
- Exception test stage
- M
- FP multiplier
- First stage of multiplier
- N
- FP multiplier
- Second stage of multiplier
- R
- FP adder
- Rounding stage
- S
- FP adder
- Operand shift stage
- U
- Unpack FP numbers

MIPS FP Pipe Stages

FP Instr 1 2 3 4 5 6 7 8 ...
Add, Subtract U S+ A A+ R R+S
Multiply U E+ M M M N N+ A R
Divide U A R D R U E+ M M M N N+ A R
Square root U E (A=R) N N A R
Negate U S
Absolute value U S
FP compare U A R

Stages:
- M First stage of multiplier
- N Second stage of multiplier
- R Rounding stage
- S Operand shift stage
- U Unpack FP numbers

R4000 Performance

- Not ideal CPI of 1:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)

Summary of Pipelining Basics

• Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
• Increasing length of pipe increases impact of hazards; pipelining helps instruction bandwidth, not latency
• Compilers reduce cost of data and control hazards
  - Load delay slots
  - Branch delay slots
  - Branch prediction
• Interrupts, Instruction Set, FP makes pipelining harder
• Handling context switches.

Advanced Pipelining and Instruction Level Parallelism

• Deep Pipelines
  - Hazards increase with pipeline depth
  - Need more independent instructions
  - gcc 17% control transfer
  - 5 instructions + 1 branch
  - Beyond single basic block to get more instruction level parallelism
• Loop level parallelism one opportunity, SW and HW
• Do examples and then explain nomenclature
• DLX Floating Point as example
  - Measurements suggests R4000 performance FP execution has room for improvement
FP Loop: Where are the Hazards?

Loop: LD F0,0(R1) ;F0=vector element
ADD F4,F0,F2 ;add scalar in F2
SD 0(R1),F4 ;store result
BNEZ R1,R1,8 ;branch R1!=zero
NOP ;delayed branch slot

instruction producing result: Instruction producing result
instruction using result: Instruction producing result
Latency in clock cycles: 3
FP ALU op: Another FP ALU op
Load double: FP ALU op
Store double: FP ALU op
Integer op: FP ALU op

FP Loop Showing Stalls

1 Loop: LD F0,0(R1) ;F0=vector element
2 stall
3 ADD F4,F0,F2 ;add scalar in F2
4 stall
5 stall
6 SD 0(R1),F4 ;store result
7 SUBI R1,R1,8 ;decrement pointer 8B (DW)
8 BNEZ R1,R1,8 ;branch R1!=zero
9 stall ;delayed branch slot

Instruction producing result: Instruction producing result
Instruction using result: Instruction producing result
Latency in clock cycles: 3
FP ALU op: Another FP ALU op
Load double: FP ALU op
Store double: FP ALU op
Integer op: FP ALU op

• Where are the stalls?

Revised FP Loop Minimizing Stalls

1 Loop: LD F0,0(R1) ;F0=vector element
2 stall
3 ADD F4,F0,F2
4 SUBI R1,R1,8
5 BNEZ R1,Loop ;branch R1!=zero
6 SD 0(R1),F4 ;delayed branch
7 stall
8 SD 0(R1),F4 ;delayed branch slot
9 stall
10 Instruction producing result: Instruction producing result
11 Instruction using result: Instruction producing result
12 Latency in clock cycles: 3
13 FP ALU op: Another FP ALU op
14 Load double: FP ALU op
15 Store double: FP ALU op
16 Integer op: FP ALU op

How do we make this faster?

Unrolled Loop That Minimizes Stalls

1 Loop: LD F0,0(R1) ;F0=vector element
2 ADD F4,F0,F2
3 ADD F8,F6,F2
4 ADD F12,F10,F2
5 ADD F16,F14,F2
6 ADD F14,F12,F2
7 ADD F12,F10,F2
8 ADD F16,F14,F2
9 SD 0(R1),F4
10 SD 0(R1),F4 ;8-32 -> 24
11 SUBI R1,R1,#32 ;alter to 4*8
12 BNEZ R1,Loop ;branch R1!=zero
13 instruction producing result: Instruction producing result
14 Instruction using result: Instruction producing result
15 Latency in clock cycles: 3
16 FP ALU op: Another FP ALU op
17 Load double: FP ALU op
18 Store double: FP ALU op
19 Integer op: FP ALU op

What assumptions made when moved code?

- OK to move store past SUBI even though changes register
- When is it safe for compiler to do such changes?

Unrolled Loop Four Times

Rewrite loop to minimize stalls?

1 Loop: LD F0,0(R1) ;F0=vector element
2 ADD F4,F0,F2
3 ADD F8,F6,F2
4 ADD F12,F10,F2
5 ADD F16,F14,F2
6 ADD F14,F12,F2
7 ADD F12,F10,F2
8 ADD F16,F14,F2
9 SD 0(R1),F4
10 SD 0(R1),F4
11 SUBI R1,R1,#32
12 BNEZ R1,Loop ;branch R1!=zero
13 BNEZ R1,Loop
14 Instruction producing result: Instruction producing result
15 Instruction using result: Instruction producing result
16 Latency in clock cycles: 3
17 FP ALU op: Another FP ALU op
18 Load double: FP ALU op
19 Store double: FP ALU op
20 Integer op: FP ALU op

What assumptions made when moved code?

- OK to move store past SUBI even though changes register
- When is it safe for compiler to do such changes?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration
Assumes R1 is multiple of 4

14 clock cycles, or 3.5 per iteration
• Definitions: compiler concerned about dependencies in program, whether or not a HW hazard exists depends on a given pipeline
• (True) Data dependencies (RAW if a hazard for HW)
  – Instruction i produces a result used by instruction j, or
  – Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
• Easy to determine for registers (fixed names)
• Hard for memory:
  – Does 100(R4) = 20(R6)?
  – From different loop iterations, does 20(R6) = 20(R6)?

• Again Hard for Memory Accesses
  – Does 100(R4) = 20(R6)?
  – From different loop iterations, does 20(R6) = 20(R6)?
• Our example required compiler to know that if R1 doesn’t change then:
  0(R1) ? -8(R1) ? -16(R1) ? -24(R1)
• There were no dependencies between some loads and stores so they could be moved past each other

• Two (obvious) constraints on control dependences:
  – An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
  – An instruction that is not control dependent on a branch cannot be moved to after the branch so that its execution is controlled by the branch.

• Control dependencies relaxed to get parallelism; get same effect if preserve order of exceptions and data flow

• Example: Where are data dependencies?
  (A,B,C distinct & nonoverlapping)
  for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i]; /* S1 */
    B[i+1] = B[i] + A[i+1]; /* S2 */
  }
  1. S2 uses the value, A[i+1], computed by S1 in the same iteration.
  2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
  This is a “loop-carried dependence”: between iterations
• Implies that iterations are dependent, and can’t be executed in parallel
• Not the case for our example; each iteration was independent
**Summary**

- Need to expose parallelism to exploit HW
- Loop level parallelism is easiest to see
- SW dependencies defined for program, hazards if HW cannot resolve
- SW dependencies/compiler sophistication determine if compiler can unroll loops

**Can we do better?**

- Problem: Stall in ID stage if any data hazard.
- Your task: Teams of two, propose a design to eliminate these stalls.

```
MULD F2, F3, F4  # Long latency...
ADDD F1, F2, F3
ADDD F3, F4, F5
ADDD F1, F4, F5
```

**Next Time**

- Hardware ILP: Scoreboarding & Tomasulo’s Algorithm