Lecture 9: Introduction to Instruction Level Parallelism

Professor Alvin R. Lebeck
Computer Science 220
Fall 1999

Todo

• Homework #1 Due Today
• Read Chapter 4
• Homework #2 Due October 1
• Project selection by October 1
• Today: ILP and Scheduling to expose ILP
Summary of Pipelining Basics

- Increasing length of pipe increases impact of hazards; pipelining helps instruction bandwidth, not latency
- Interrupts, Instruction Set, FP makes pipelining harder
- Compilers reduce cost of data and control hazards
  - Load delay slots
  - Branch delay slots
  - Branch prediction

Case Study: MIPS R4000 (100 MHz to 200 MHz)

- 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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Functional unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IS</td>
<td>RF</td>
</tr>
<tr>
<td>EX</td>
<td>RF</td>
<td>IF</td>
</tr>
<tr>
<td>DS</td>
<td>TC</td>
<td>WB</td>
</tr>
<tr>
<td>TC</td>
<td>WB</td>
<td>IF</td>
</tr>
</tbody>
</table>

#### THREE Cycle Branch Latency

- **Condition**: evaluated during EX phase
- **Delay slot plus two stalls**: IF IS RF EX DF DS TC WB
- **Branch likely cancels delay slot if not taken**: IF IS RF EX DF DS TC WB

### 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**
  - 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  | M | N | N+A| R |   
| Divide            | U | A  | R  | D  | D2 | D  | D  | D  | D  
| Square root       | U | E  | (A+R) | 10 | 8 |   | 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
- **A**: Mantissa ADD stage
- **D**: Divide pipeline stage
- **E**: Exception test stage

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)
Advanced Pipelining and Instruction Level Parallelism

- Deep Pipelines
  - Hazards increase with pipeline depth
  - Need independent instructions

- gcc 17% control transfer
  - 5 instructions + 1 branch
  - Beyond single 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 suggest R4000 performance FP execution has room for improvement

---

**FP Loop: Where are the Hazards?**

```
Loop:  LD  F0,0(R1) ;F0=vector element
       ADDD F4,F0,F2 ;add scalar in F2
       SD   0(R1),F4 ;store result
       SUBI R1,R1,8 ;decrement pointer 8B (DW)
       BNEZ R1,Loop ;branch R1!=zero
       NOP ;delayed branch slot
```

<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>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>0</td>
</tr>
</tbody>
</table>
FP Loop Hazards

Loop:  
```
LD   F0,0(R1) ;F0=vector element
ADDD F4,F0,F2 ;add scalar in F2
SD   0(R1),F4 ;store result
SUBI R1,R1,8 ;decrement pointer 8B (DW)
BNEZ R1,Loop ;branch R1!=zero
NOP  ;delayed branch slot
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction</th>
<th>Latency in producing result using result 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>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>0</td>
</tr>
</tbody>
</table>

- Where are the stalls?

FP Loop Showing Stalls

1. Loop:  
```
LD   F0,0(R1) ;F0=vector element
```
2. stall
3. ADDD 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,Loop ;branch R1!=zero
9. stall ;delayed branch slot

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction</th>
<th>Latency in producing result using result 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>
</tbody>
</table>

- Rewrite code to minimize stalls?
Revised FP Loop Minimizing Stalls

1 Loop: LD F0, 0(R1)
2 stall
3 ADDD F4, F0, F2
4 SUBI R1, R1, 8
5 BNEZ R1, Loop ; delayed branch
6 SD 8(R1), F4 ; altered when move past SUBI

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction producing 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>
</tbody>
</table>

Unroll loop 4 times to make faster?

Unroll Loop Four Times

1 Loop: LD F0, 0(R1)
2 ADDD F4, F0, F2
3 SD 0(R1), F4 ; drop SUBI & BNEZ
4 LD F6, -8(R1)
5 ADDD F8, F6, F2
6 SD -8(R1), F8 ; drop SUBI & BNEZ
7 LD F10, -16(R1)
8 ADDD F12, F10, F2
9 SD -16(R1), F12 ; drop SUBI & BNEZ
10 LD F14, -24(R1)
11 ADDD F16, F14, F2
12 SD -24(R1), F16
13 SUBI R1, R1, #32 ; alter to 4*8
14 BNEZ R1, LOOP
15 NOP

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

1 Loop: LD F0, 0(R1)
2 LD F6, -8(R1)
3 LD F10, -16(R1)
4 LD F14, -24(R1)
5 ADDD F4, F0, F2
6 ADDD F8, F6, F2
7 ADDD F12, F10, F2
8 ADDD F16, F14, F2
9 SD 0(R1), F4
10 SD -8(R1), F8
11 SD -16(R1), F12
12 SUBI R1, R1, #32
13 BNEZ R1, LOOP
14 SD 8(R1), F16 ; 8-32 = -24

14 clock cycles, or 3.5 per iteration

Compiler Perspectives on Code Movement

- 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)?
Name dependence: two instructions use same name but don’t exchange data

- Antidependence (WAR if a hazard for HW)
  - Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first.

- Output dependence (WAW if a hazard for HW)
  - Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.

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:

\[ 0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1) \]

There were no dependencies between some loads and stores so they could be moved past each other.
Final kind of dependence called control dependence

Example

```c
if p1 {S1;};
if p2 {S2;}
```

S1 is control dependent on p1 and S2 is control dependent on p2 but not on p1.

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
When is it Safe to Unroll a Loop?

- Example: Where are data dependencies? (A,B,C distinct & nonoverlapping)

```c
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 distinct

Summary

- Need to expose parallelism to exploit HW

- Can be done in Instruction Level Parallelism in SW or HW

- Loop level parallelism is easiest to see

- SW parallelism dependencies defined for program, hazards if HW cannot resolve

- SW dependencies/compiler sophistication determine if compiler can unroll loops
Next Time

- Hardware ILP: Scoreboarding