CS152
Computer Architecture and Engineering
Lecture 15
Advanced pipelining/Compiler Scheduling

October 24, 2001
John Kubiatowicz (http.cs.berkeley.edu/~kubitron)

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/

Review: Pipelining
- Key to pipelining: smooth flow
  - Making all instructions the same length can increase performance!
- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Data hazards must be handled carefully:
  - RAW (Read-After-Write) data hazards handled by forwarding
  - WAW (Write-After-Write) and WAR (Write-After-Read) hazards don’t exist in 5-stage pipeline
- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
  - Change in programmer semantics to make hardware simpler

Recap: Data Hazards
- Structural Hazard: IFetch DCD OpFetch
- Control Hazard: IFetch DCD
- RAW (read after write) Data Hazard: IF DCD MemOpFetch
- WAW Data Hazard (write after write): IF DCD Mem

Recap: Data Stationary Control
- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later
Review: Resolve RAW by “forwarding” (or bypassing)

• Detect nearest valid write op operand register and forward into op latches, bypassing remainder of the pipe
• Increase muxes to add paths from pipeline registers
• Data Forwarding = Data Bypassing

What about Interrupts, Traps, Faults?

• External Interrupts:
  – Allow pipeline to drain, Fill with NOPs
  – Load PC with interrupt address

• Faults (within instruction, restartable)
  – Force trap instruction into IF
  – disable writes till trap hits WB
  – must save multiple PCs or PC + state

• Recall: Precise Exceptions ⇒ State of the machine is preserved as if program executed up to the offending instruction
  – All previous instructions completed
  – Offending instruction and all following instructions act as if they have not even started
  – Same system code will work on different implementations

Question: Critical Path???

• Bypass path is invariably trouble
  • Options?
    – Make logic really fast
    – Move forwarding after muxes
      » Problem: screws up branches that require forwarding!
      » Use same tricks as “carry-skip” adder to fix this?
      » This option may just push delay around...!
    – Insert an extra cycle for branches that need forwarding?
      » Or: hit common case of forwarding from EX stage and stall for forward from memory?

Exception/Interrupts: Implementation questions

5 instructions, executing in 5 different pipeline stages!

• Who caused the interrupt?
<table>
<thead>
<tr>
<th>Stage</th>
<th>Problem interrupts occurring</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory-protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory-protection violation; memory error</td>
</tr>
</tbody>
</table>

• How do we stop the pipeline? How do we restart it?
• Do we interrupt immediately or wait?
• How do we sort all of this out to maintain preciseness?
### Exception Handling

1. **IAU**
2. **npc**
3. **I mem**

- Detect bad instruction address
- Detect bad instruction
- Detect overflow
- Detect bad data address

**Allow exception to take effect**

Resolution: Freeze above & Bubble Below

- Flush accomplished by setting “invalid” bit in pipeline

### Another look at the exception problem

- **Use pipeline to sort this out!**
  - Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don’t act on exception until it reaches WB stage.
- **Handle interrupts through “faulting noop” in IF stage**
- **When instruction reaches end of MEM stage:**
  - Save PC → EPC, Interrupt vector addr → PC
  - Turn all instructions in earlier stages into noops!

### FYI: MIPS R3000 clocking discipline

- **phi1**
- **phi2**
- 2-phase non-overlapping clocks
- Pipeline stage is two (level sensitive) latches

**Edge-triggered**
**MIPS R3000 Instruction Pipeline**

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Decode</th>
<th>ALU / E.A</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>E.A.</td>
<td>TLB</td>
</tr>
<tr>
<td>Reg. Read</td>
<td>Operation</td>
<td>D-Cache</td>
<td></td>
<td>WB</td>
</tr>
</tbody>
</table>

**Resource Usage**

<table>
<thead>
<tr>
<th>TLB</th>
<th>I-Cache</th>
<th>RF</th>
<th>ALU/ALU</th>
<th>D-Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write in phase 1, read in phase 2 => eliminates bypass from WB

**Recall: Data Hazard on r1**

Time (clock cycles)

```
add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
```

With MIPS R3000 pipeline, no need to forward from WB stage

**MIPS R3000 Multicycle Operations**

Use control word of local stage to step through multicycle operation

Stall all stages above multicycle operation in the pipeline

Drain (bubble) stages below it

Alternatively, launch multiply/divide to autonomous unit, only stall pipe if attempt to get result before ready

- This means stall mflo/mfhi in decode stage if multiply/divide still executing
- Extra credit in Lab 5 does this

**Is CPI = 1 for our pipeline?**

- Remember that CPI is an “Average # cycles/inst

- CPI here is 1, since the average throughput is 1 instruction every cycle.
- What if there are stalls or multi-cycle execution?
- Usually CPI > 1. How close can we get to 1??
**Recall: Compute CPI?**

- Start with Base CPI
- Add stalls

\[
CPI = CPI_{base} + CPI_{stall}
\]

\[
CPI_{stall} = STALL_{type-1} \times freq_{type-1} + STALL_{type-2} \times freq_{type-2}
\]

Suppose:
- \( CPI_{base} = 1 \)
- \( \text{Freq branch} = 20\%, \text{freq base} = 30\% \)
- Suppose branches always cause 1 cycle stall
- Loads cause a 100 cycle stall 1% of time

Then: \( CPI = 1 + (1 \times 0.20) + (100 \times 0.30 \times 0.01) = 1.5 \)

- Multicycle? Could treat as:
  \( CPI_{stall} = (CYCLES - CPI_{base}) \times freq_{inst} \)

**Administrivia**

- Get moving on Lab 5!
  - This lab is even harder than Lab4.
  - Trickier to debug…!
  - Start with unipipelined version?
    » Interesting thought… May or may not help
  - Division of labor due tonight at Midnight!

- Dynamic scheduling techniques discussed in the Other Hennessy & Patterson book:
  - “Computer Architecture: A Quantitative Approach”
  - Chapter 4

**Administrivia: Be careful about clock edges in lab5!**

- Since Register file has edge-triggered write:
  - Must have everything set up at end of memory stage
  - This means that “M” register here is not necessary!

**Case Study: MIPS R4000 (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

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:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Functional unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>FP adder</td>
<td>Mantissa ADD stage</td>
</tr>
<tr>
<td>D</td>
<td>FP divider</td>
<td>Divide pipeline stage</td>
</tr>
<tr>
<td>E</td>
<td>FP multiplier</td>
<td>Exception test stage</td>
</tr>
<tr>
<td>M</td>
<td>FP multiplier</td>
<td>First stage of multiplier</td>
</tr>
<tr>
<td>N</td>
<td>FP multiplier</td>
<td>Second stage of multiplier</td>
</tr>
<tr>
<td>R</td>
<td>FP adder</td>
<td>Rounding stage</td>
</tr>
<tr>
<td>S</td>
<td>FP adder</td>
<td>Operand shift stage</td>
</tr>
<tr>
<td>U</td>
<td>Unpack FP numbers</td>
<td></td>
</tr>
</tbody>
</table>

MIPS FP Pipe Stages

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

- Let's assume full pipelining:
  - If we have a 4-cycle instruction, then we need 3 instructions between a producing instruction and its use:
    ```
    mult $F0,$F2,$F4
    delay-1
    delay-2
    delay-3
    addf $F6,$F10,$F0
    ```
    **Earliest forwarding for 1-cycle instructions**
    **Earliest forwarding for 4-cycle instructions**

FP Loop: Where are the Hazards?

- Loop: 
  ```
  LD F0,0(R1) ; F0=vector element
  ADDD F4,F0,F2 ; add scalar from 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
  ```

- Where are the stalls?

FP Loop Showing Stalls

- 9 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

- 6 clocks: Unroll loop 4 times code to make faster?

Swap BNEZ and SD by changing address of SD

- Instruction producing result | Instruction using result | Latency in clock cycles
  - FP ALU op | Another FP ALU op | 3
  - FP ALU op | Store double | 2
  - Load double | FP ALU op | 1

- 6 clocks: Unroll loop 4 times code to make faster?
Unroll Loop Four Times (straightforward way)

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

1 cycle stall
2 cycles stall

Rewrite loop to minimize stalls?

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 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 BNEZ R1, LOOP
14 SD 8(R1), F16 ; 8-32 = -24

What assumptions made when moved code?
– OK to move store past SUBI even though changes register
– OK to move loads before stores: get right data?
– When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration
When safe to move instructions?

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Two main variations: Superscalar and VLIW
- Superscalar: varying no. instructions/cycle (1 to 6)
  - Parallelism and dependencies determined/resolved by HW
  - IBM PowerPC 604, Sun UltraSparc, DEC Alpha 21164, HP 7100
- Very Long Instruction Words (VLIW): fixed number of instructions (16) parallelism determined by compiler
  - Pipeline is exposed; compiler must schedule delays to get right result
- Explicit Parallel Instruction Computer (EPIC) / Intel
  - 128 bit packets containing 3 instructions (can execute sequentially)
  - Can link 128 bit packets together to allow more parallelism
  - Compiler determines parallelism, HW checks dependencies and forwards/stalls

Type PipeStages
Int. instruction IF ID EX MEM WB
FP instruction IF ID EX MEM WB
Int. instruction IF ID EX MEM WB
FP instruction IF ID EX MEM WB
FP instruction IF ID EX MEM WB

1 cycle load delay expands to 3 instructions in SS
- instruction in right half can’t use it, nor instructions in next slot

Getting CPI < 1: Issuing Multiple Instructions/Cycle

Type PipeStages
Int. instruction IF ID EX MEM WB
FP instruction IF ID EX MEM WB
Int. instruction IF ID EX MEM WB
FP instruction IF ID EX MEM WB
FP instruction IF ID EX MEM WB

1 cycle load delay expands to 3 instructions in SS
- instruction in right half can’t use it, nor instructions in next slot
Loop Unrolling in Superscalar

<table>
<thead>
<tr>
<th>Integer instruction</th>
<th>FP instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F0,0(R1)</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>3</td>
</tr>
<tr>
<td>LD F14,-24(R1)</td>
<td>ADDD F8,F5,F2</td>
<td>4</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>ADDD F12,F10,F2</td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>ADDD F16,F14,F2</td>
<td>6</td>
</tr>
<tr>
<td>SD -8(R1),F5</td>
<td>ADDD F20,F18,F2</td>
<td>7</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>SD -24(R1),F16</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>SUBI R1,R1,#40</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>BNEZ R1,LOOP</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration

Limits of Superscalar

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at the same time, greater difficulty of decode and issue
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
- VLIW: tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word can execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/ branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>LD F6,-8(R1)</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>LD F14,-24(R1)</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>LD F16,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADDD F4,F0,F2</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td>ADDD F12,F10,F2 ADDD F8,F6,F2</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>SD -8(R1),F5</td>
<td>ADDD F20,F18,F2</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>SD -24(R1),F16</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td>SD -40(R1),F24</td>
<td>SUBI R1,R1,#40</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>SD -0(R1),F28</td>
<td>BNEZ R1,LOOP</td>
<td></td>
<td></td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration

Need more registers in VLIW (EPIC => 128int + 128FP)

Software Pipelining

- Observation: If iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (Tomasulo in SW)
### Software Pipelining Example

**Before: Unrolled 3 times**
1. LD F0,0(R1)
2. ADD F4,F0,F2
3. SD 0(R1),F4
4. LD F6,-8(R1)
5. ADD F8,F6,F2
6. SD -8(R1),F8
7. LD F10,-16(R1)
8. ADD F12,F10,F2
9. SD -16(R1),F12
10. SUBI R1,R1,#24
11. BNEZ R1,LOOP

**After: Software Pipelined**
1. SD 0(R1),F4
2. ADD F4,F0,F2
3. LD F0,-16(R1)
4. SUBI R1,R1,#8
5. BNEZ R1,LOOP

- **Symbolic Loop Unrolling**
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling

### Software Pipelining with Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
<th>FP</th>
<th>FP</th>
<th>Int. op/ Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref 1</td>
<td>ref 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F0,-48(R1)</td>
<td>ST 0(R1),F4</td>
<td>ADDD F4,F0,F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F6,-56(R1)</td>
<td>ST -8(R1),F8</td>
<td>ADDD F8,F6,F2</td>
<td>SUBI R1,R1,#24</td>
<td></td>
</tr>
<tr>
<td>LD F10,-40(R1)</td>
<td>ST 8(R1),F12</td>
<td>ADDD F12,F10,F2</td>
<td>BNEZ R1,LOOP</td>
<td></td>
</tr>
</tbody>
</table>

- **Software pipelined across 9 iterations of original loop**
  - In each iteration of above loop, we:
    - Store to m,m-8,m-16 (iterations i-3,i-2,i-1)
    - Compute for m-24,m-32,m-40 (iterations i,i+1,i+2)
    - Load from m-48,m-56,m-64 (iterations i+3,i+4,i+5)
- 9 results in 9 cycles, or 1 clock per iteration
- **Average:** 3.3 ops per clock, 66% efficiency
- **Note:** Need less registers for software pipelining (only using 7 registers here, was using 15)

### Can we use HW to get CPI closer to 1?

- **Why in HW at run time?**
  - Works when can’t know real dependence at compile time
  - Compiler simpler
  - Code for one machine runs well on another

- **Key idea:** Allow instructions behind stall to proceed:
  - DIVD F0,F2,F4
  - ADDD F10,F0,F8
  - SUBD F12,F8,F14

- **Out-of-order execution => out-of-order completion.**

- **Disadvantages?**
  - Complexity
  - Precise interrupts harder! (Talk about this next time)

### Problems?

- **How do we prevent WAR and WAW hazards?**
- **How do we handle with variable latency?**
  - Forwarding for RAW hazards harder.
Summary

- Precise interrupts are easy to implement in a 5-stage pipeline:
  - Mark exception on pipeline state rather than acting on it
  - Handle exceptions at one place in pipeline (memory-stage/beginning of writeback)
- Loop unrolling ⇒ Multiple iterations of loop in software:
  - Amortizes loop overhead over several iterations
  - Gives more opportunity for scheduling around stalls
- Software Pipelining ⇒ take one instruction from each of several iterations of the loop
  - Software overlapping of loop iterations
- Very Long Instruction Word machines (VLIW) ⇒ Multiple operations coded in single, long instruction
  - Requires compiler to decide which ops can be done in parallel
  - Trace scheduling ⇒ find common path and schedule code as if branches didn’t exist (+ add “fixup code”)

10/24/01 ©UCB Fall 2001 CS152 / Kubiatowicz Lec15.41