CS152: Computer Architecture and Engineering
Introduction to Pipelining

October 24, 1997
Dave Patterson (http.cs.berkeley.edu/~patterson)

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/
Review: Summary of Pipelining Basics

- Pipelines pass control information down the pipe just as data moves down pipe
- Forwarding/Stalls handled by local control
- 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
- Interrupts, Instruction Set, FP makes pipelining harder
- Compilers reduce cost of data and control hazards
  - Load delay slots
  - Branch delay slots
  - Branch prediction
# Recap: Pipeline Hazards

<table>
<thead>
<tr>
<th>I-Fetch</th>
<th>DCD</th>
<th>MemOpFetch</th>
<th>OpFetch</th>
<th>Exec</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFetch</td>
<td>DCD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Structural Hazard**

<table>
<thead>
<tr>
<th>I-Fetch</th>
<th>DCD</th>
<th>OpFetch</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFetch</td>
<td>DCD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Control Hazard**

<table>
<thead>
<tr>
<th>IF</th>
<th>DCD</th>
<th>EX</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>DCD</td>
<td>EX</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>

**RAW (read after write) Data Hazard**

**WAW Data Hazard (write after write)**

<table>
<thead>
<tr>
<th>IF</th>
<th>DCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>DCD</td>
</tr>
</tbody>
</table>

**WAR Data Hazard (write after read)**
Recap: Data Hazards

° Avoid some “by design”
  • eliminate WAR by always fetching operands early (DCD) in pipe
  • eliminate WAW by doing all WBs in order (last stage, static)

° Detect and resolve remaining ones
  • stall or forward (if possible)
### Recap: Exception Problem

- **Exceptions/Interrupts**: 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - Restart?
  - 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>

- Load with data page fault, Add with instruction page fault?
- Solution 1: interrupt vector/instruction, check last stage
- Solution 2: interrupt ASAP, restart everything incomplete
The Big Picture: Where are We Now?

The Five Classic Components of a Computer

Today’s Topics:

• Recap last lecture
• Review MIPS R3000 pipeline
• Administrivia
• Advanced Pipelining
• SuperScalar, VLIW/EPIC
FYI: MIPS R3000 clocking discipline

- 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 Reg. Read</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>Operation</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>E.A.</td>
<td>TLB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D-Cache</td>
</tr>
</tbody>
</table>

#### Resource Usage

Write in phase 1, read in phase 2 => eliminates bypass from WB
Recall: Data Hazard on r1

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

Ex: Multiply, Divide, Cache Miss

- Stall all stages above multicycle operation in the pipeline
- Drain (bubble) stages below it
- Use control word of local stage state to step through multicycle operation
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
Getting CPI < 1: Issuing Multiple Instructions/Cycle

° Superscalar DLX: 2 instructions, 1 FP & 1 anything else
  – Fetch 64-bits/clock cycle; Int on left, FP on right
  – Can only issue 2nd instruction if 1st instruction issues
  – More ports for FP registers to do FP load & FP op in a pair

<table>
<thead>
<tr>
<th>Type</th>
<th>PipeStages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int. instruction</td>
<td>IF  ID   EX  MEM  WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF  ID   EX  MEM  WB</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>IF  ID   EX  MEM  WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF  ID   EX  MEM  WB</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>IF  ID   EX  MEM  WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF  ID   EX  MEM  WB</td>
</tr>
</tbody>
</table>

° 1 cycle load delay expands to 3 instructions in SS
  • instruction in right half can’t use it, nor instructions in next slot
Unrolled Loop that Minimizes Stalls for Scalar

1 Loop: 

1. 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

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles
Loop Unrolling in Superscalar

<table>
<thead>
<tr>
<th>Integer instruction</th>
<th>FP instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F0,0(R1)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F6,-8(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,F6,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),F8</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
Software Pipelining

- Observation: if iterations from loops are independent, then can get 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. ADDD F4,F0,F2
3. SD  0(R1),F4
4. LD  F6,-8(R1)
5. ADDD F8,F6,F2
6. SD  -8(R1),F8
7. LD  F10,-16(R1)
8. ADDD 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 ; Stores M[i]
2. ADDD F4,F0,F2 ; Adds to M[i-1]
3. LD  F0,-16(R1); Loads M[i-2]
4. SUBI R1,R1,#8
5. BNEZ R1,LOOP

- Symbolic Loop Unrolling
  - Less code space
  - Fill & drain pipe only once
    vs. each iteration in loop unrolling
Administrative Issues

° Schedule Ahead

midterm

proj present
last lecture

final report

pipeline (5) → cache (6) → xtra & writeup

Sign up IEEE/computer society: www.computer.org
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 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></td>
<td>1</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>LD F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>ADDD F8,F6,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td></td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F16,F14,F2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDD F20,F18,F2</td>
<td>ADDD F24,F22,F2</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>SD -8(R1),F8</td>
<td>ADDD F28,F26,F2</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>SD -24(R1),F16</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td>SD -40(R1),F24</td>
<td></td>
<td></td>
<td>SUBI R1,R1,#48</td>
<td>8</td>
</tr>
<tr>
<td>SD -0(R1),F28</td>
<td></td>
<td></td>
<td></td>
<td>BNEZ R1,LOOP</td>
<td>9</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)
Trace Scheduling

- **Parallelism across IF branches vs. LOOP branches**

- **Two steps:**
  - *Trace Selection*
    - Find likely sequence of basic blocks (*trace*) of (statically predicted) long sequence of straight-line code
  - *Trace Compaction*
    - Squeeze trace into few VLIW instructions
    - Need bookkeeping code in case prediction is wrong
HW Schemes: Instruction Parallelism

° 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

  • Enables out-of-order execution => out-of-order completion
  • ID stage checked both for structural & data dependencies
HW Schemes: Instruction Parallelism

° Out-of-order execution divides ID stage:
  1. Issue—decode instructions, check for structural hazards
  2. Read operands—wait until no data hazards, then read operands

° Scoreboards allow instruction to execute whenever 1 & 2 hold, not waiting for prior instructions

° CDC 6600: In order issue, out of order execution, out of order commit (also called completion)
Scoreboard Implications

° Out-of-order completion => WAR, WAW hazards?

° Solutions for WAR
  • Queue both the operation and copies of its operands
  • Read registers only during Read Operands stage

° For WAW, must detect hazard: stall until other completes

° Need to have multiple instructions in execution phase => multiple execution units or pipelined execution units

° Scoreboard keeps track of dependencies, state or operations

° Scoreboard replaces ID, EX, WB with 4 stages
## Performance of Dynamic SS

<table>
<thead>
<tr>
<th>Iteration Instructions</th>
<th>Issues</th>
<th>Executes</th>
<th>Writes result</th>
</tr>
</thead>
<tbody>
<tr>
<td>no.</td>
<td>clock-cycle number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LD F0,0(R1)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>ADDD F4,F0,F2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>SD 0(R1),F4</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>SUBI R1,R1,#8</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>BNEZ R1,LOOP</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>LD F0,0(R1)</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>ADDD F4,F0,F2</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>SD 0(R1),F4</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>SUBI R1,R1,#8</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>BNEZ R1,LOOP</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

≈ 4 clocks per iteration

Branches, Decrements still take 1 clock cycle
Dynamic Branch Prediction

Solution: 2-bit scheme where change prediction only if get misprediction *twice*
BHT Accuracy

° **Mispredict because either:**
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table

° 4096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%

° 4096 about as good as infinite table, but 4096 is a lot of HW
Need Address @ Same Time as Prediction

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can’t use wrong branch address

- Return instruction addresses predicted with stack
Dynamic Branch Prediction Summary

- Branch History Table: 2 bits for loop accuracy
- Branch Target Buffer: include branch address & prediction
HW support for More ILP

- Avoid branch prediction by turning branches into conditionally executed instructions:
  
  if (x) \text{then } A = B \text{ op } C \text{ else NOP}
  
  - If false, then neither store result nor cause exception
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr.
  - EPIC: 64 1-bit condition fields selected so conditional execution

- Drawbacks to conditional instructions
  
  - Still takes a clock even if “annulled”
  - Stall if condition evaluated late
  - Complex conditions reduce effectiveness; condition becomes known late in pipeline
HW support for More ILP

° **Speculation**: allow an instruction to issue that is dependent on branch predicted to be taken *without* any consequences (including exceptions) if branch is not actually taken (“HW undo”)

° Often try to combine with dynamic scheduling

° Separate *speculative* bypassing of results from real bypassing of results
  * When instruction no longer speculative, write results (*instruction commit*)
  * execute out-of-order but commit in order
HW support for More ILP

- Need HW buffer for results of uncommitted instructions: \emph{reorder buffer}
  - Reorder buffer can be operand source
  - Once operand commits, result is found in register
  - 3 fields: instr. type, destination, value
  - Use reorder buffer number instead of reservation station
  - Instructions commit in order
  - As a result, it's easy to undo speculated instructions on mispredicted branches or on exceptions
Dynamic Scheduling in PowerPC 604 and Pentium Pro

° Both In-order Issue, Out-of-order execution, In-order Commit

PPro central reservation station for any functional units with one bus shared by a branch and an integer unit
## Dynamic Scheduling in PowerPC 604 and Pentium Pro

<table>
<thead>
<tr>
<th>Parameter</th>
<th>PPC</th>
<th>PPro</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. instructions issued/clock</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Max. instr. complete exec./clock</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Max. instr. commited/clock</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Instructions in reorder buffer</td>
<td>16</td>
<td>40</td>
</tr>
<tr>
<td>Number of rename buffers</td>
<td>12 Int/8 FP</td>
<td>40</td>
</tr>
<tr>
<td>Number of reservations stations</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>No. integer functional units (FUs)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>No. floating point FUs</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>No. branch FUs</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>No. complex integer FUs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>No. memory FUs</td>
<td>1 1 load +1 store</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Scheduling in Pentium Pro

° PPro doesn’t pipeline 80x86 instructions

° PPro decode unit translates the Intel instructions into 72-bit micro-operations (≈ MIPS)

° Sends micro-operations to reorder buffer & reservation stations

° Takes 1 clock cycle to determine length of 80x86 instructions + 2 more to create the micro-operations

° Most instructions translate to 1 to 4 micro-operations

° Complex 80x86 instructions are executed by a conventional microprogram (8K x 72 bits) that issues long sequences of micro-operations
Limits to Multi-Issue Machines

° Inherent limitations of ILP
  • 1 branch in 5: How to keep a 5-way VLIW busy?
  • Latencies of units: many operations must be scheduled
  • Need about Pipeline Depth x No. Functional Units of independent operations to keep machines busy

° Difficulties in building HW
  • Duplicate FUs to get parallel execution
  • Increase ports to Register File
    - VLIW example needs 7 read and 3 write for Int. Reg.
      & 5 read and 3 write for FP reg
  • Increase ports to memory
  • Decoding SS and impact on clock rate, pipeline depth
Limits to Multi-Issue Machines

° Limitations specific to either SS or VLIW implementation
  • Decode issue in SS
  • VLIW code size: unroll loops + wasted fields in VLIW
  • VLIW lock step => 1 hazard & all instructions stall
  • VLIW & binary compatibility is practical weakness
Braniac vs. Speed Demon

- 8-scalar IBM Power-2 @ 71.5 MHz (5 stage pipe)
- vs. 2-scalar Alpha @ 200 MHz (7 stage pipe)

[Graph showing SPEC Marks for various benchmarks]
# Recent Machines

<table>
<thead>
<tr>
<th></th>
<th>Alpha 21164</th>
<th>Pentium II</th>
<th>HP PA-8000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Year</strong></td>
<td>1995</td>
<td>1996</td>
<td>1996</td>
</tr>
<tr>
<td><strong>Clock</strong></td>
<td>600 MHz (‘97)</td>
<td>300 MHz (‘97)</td>
<td>236 MHz (‘97)</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>8K/8K/96K/2M</td>
<td>16K/16K/0.5M</td>
<td>0/0/4M</td>
</tr>
<tr>
<td><strong>Issue rate</strong></td>
<td>2int+2FP</td>
<td>3 instr (x86)</td>
<td>4 instr</td>
</tr>
<tr>
<td><strong>Pipe stages</strong></td>
<td>7-9</td>
<td>12-14</td>
<td>7-9</td>
</tr>
<tr>
<td><strong>Out-of-Order</strong></td>
<td>6 loads</td>
<td>40 instr (μop)</td>
<td>56 instr</td>
</tr>
<tr>
<td><strong>Rename regs</strong></td>
<td>none</td>
<td>40</td>
<td>56</td>
</tr>
</tbody>
</table>
SPECint95base Performance (Oct. 1997)
SPECfp95base Performance (Oct. 1997)
Summary

- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
- More performance from deeper pipelines, parallelism
- Superscalar and VLIW
  - CPI < 1
  - Dynamic issue vs. Static issue
  - More instructions issue at same time, larger the penalty of hazards
- SW Pipelining
  - Symbolic Loop Unrolling to get most from pipeline with little code expansion, little overhead