Lecture 7: Dynamic Branch Prediction, Superscalar, VLIW, and Software Pipelining

Professor Alvin R. Lebeck
Computer Science 220
Fall 2001

Scoreboard Summary
1. Issue: decode instructions & check for structural hazards and WAW hazards (stall all following insts)
2. Read operands: wait until no data hazards, then read operands (RAW)
3. Execution
4. Write Result: check for WAR
   - Limitations of 6600 scoreboard
     - No forwarding
     - Limited to instructions in basic block (small window)
     - Number of functional units (structural hazards)
     - Wait for WAR hazards
     - Prevent WAW hazards

Tomasulo Summary
- Prevents Register as bottleneck
- Avoids WAR, WAW hazards of Scoreboard
- Allows loop unrolling in HW
- Not limited to basic blocks (provided branch prediction)
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- You should know how Scoreboard and Tomasulo’s alg would execute a piece of code...

Preview for CPI < 1
- Issue more than 1 instruction per cycle
- First branches (why?)

Dynamic Branch Prediction
- With CPI < 1 frequency of branches increases
  - Remember Amdahl’s Law...
- Performance = f(accuracy, cost of misprediction)
- Branch History Table is simplest
  - Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
- Question: How many mispredictions in a loop?
- Answer: 2
  - End of loop case, when it exits instead of looping as before
  - First time through loop on next time through code, when it predicts exit instead of looping
Dynamic Branch Prediction

- Solution: 2-bit counter where prediction changes only if mispredict twice:
  - Increment for taken, decrement for not-taken
    - 00, 01, 10, 11
- Helps when target is known before condition

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

Correlating Branches

Idea: taken/not taken of recently executed branches is related to behavior of next branch (as well as the history of that branch behavior)
- Then behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction

Accuracy of Different Schemes

(Figure 4.21, p. 272)

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Two variations
  - Superscalar: varying no. instructions/cycle (1 to 8), scheduled by compiler (statically scheduled) or by HW (Tomasulo; dynamically scheduled)
    - Pentium4, IBM PowerPC, Sun SuperSparc, DEC Alpha, HP PA-8000
  - Very Long Instruction Words (VLIW): fixed number of instructions (16) scheduled by the compiler
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 issued
  - More ports for FP registers to do FP load & FP op in a pair

<table>
<thead>
<tr>
<th>Pipe Stage</th>
<th>Int. instruction</th>
<th>FP instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>ID</td>
<td>EX</td>
<td>EX</td>
</tr>
<tr>
<td>EX</td>
<td>MEM</td>
<td>MEM</td>
</tr>
<tr>
<td>MEM</td>
<td>WB</td>
<td>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: LD F0, 0(R1)
2 LD F4, -8(R1)
3 ADD F14, -24(R1)
4 ADD F18, -32(R1)
5 SD 0(R1), F4
6 SD -8(R1), F8
7 SUBI R1, R1, #32
8 BNEZ R1, LOOP
9 ADD F16, F14, F2
10 SUBI R1, R1, #32
11 BNEZ R1, LOOP
12 ADD F20, F18, F2
13 SD -32(R1), F20

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

Unrolled Loop in Superscalar

<table>
<thead>
<tr>
<th>Loop 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 F6, -8(R1)</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LD F10, -16(R1)</td>
<td>ADD F4, F2</td>
<td>3</td>
</tr>
<tr>
<td>LD F14, -24(R1)</td>
<td>ADD F8, F4</td>
<td>4</td>
</tr>
<tr>
<td>LD F18, -32(R1)</td>
<td>ADD F12, F10, F2</td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1), F4</td>
<td>ADD F16, F14, F2</td>
<td>6</td>
</tr>
<tr>
<td>SD -8(R1), F8</td>
<td>ADD 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

Dynamic Scheduling in Superscalar

- Dependencies stop instruction issue
- Code compiled for scalar version will run poorly on SS
  - May want code to vary depending on how superscalar
- Simple approach: separate Tomasulo Control for separate reservation stations for Integer FU/Reg and for FP FU/Reg

Dynamic Scheduling in Superscalar

- How to do instruction issue with two instructions and keep in-order instruction issue for Tomasulo?
  - Issue 2X Clock Rate, so that issue remains in order
  - Only FP loads might cause dependency between Integer and FP issue:
    - Replace load reservation station with a load queue; operands must be read in the order they are fetched
    - Load checks addresses in Store Queue to avoid RAW violation
    - Store checks addresses in Load Queue to avoid WAR,WAW

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>1 LD F0(R1)</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1 ADD F4, F2</td>
<td>1</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>1 SD 0(R1), F4</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1 SUBI R1, R1, #8</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1 BNEZ R1, LOOP</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2 LD F0(R1)</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>2 ADD F4, F6, F2</td>
<td>5</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>2 SD 0(R1), F4</td>
<td>6</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>2 SUBI R1, R1, #8</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2 BNEZ R1, LOOP</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

- 4 clocks per iteration
- Branches, Decrements still take 1 clock cycle
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 operation 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>ADDD F4,F0,F2</td>
<td>ADDD F4,F0,F2</td>
<td>ADDD F4,F0,F2</td>
<td>1</td>
</tr>
<tr>
<td>LD F18,-16(R1)</td>
<td>ADDD F8,F6,F2</td>
<td>ADDD F8,F6,F2</td>
<td>ADDD F8,F6,F2</td>
<td>ADDD F8,F6,F2</td>
<td>2</td>
</tr>
<tr>
<td>LD F26,-40(R1)</td>
<td>ADDD F10,F12,F2</td>
<td>ADDD F10,F12,F2</td>
<td>ADDD F10,F12,F2</td>
<td>ADDD F10,F12,F2</td>
<td>3</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>SD 0(R1),F4</td>
<td>SD 0(R1),F4</td>
<td>SD 0(R1),F4</td>
<td>SD 0(R1),F4</td>
<td>4</td>
</tr>
<tr>
<td>SD -8(R1),F12</td>
<td>SD -8(R1),F12</td>
<td>SD -8(R1),F12</td>
<td>SD -8(R1),F12</td>
<td>SD -8(R1),F12</td>
<td>5</td>
</tr>
<tr>
<td>LD F10, -16(R1)</td>
<td>SUBI R1,R1,#8</td>
<td>SUBI R1,R1,#8</td>
<td>SUBI R1,R1,#8</td>
<td>SUBI R1,R1,#8</td>
<td>6</td>
</tr>
<tr>
<td>ADDD F12,F10,F2</td>
<td>SD 0(R1),F4</td>
<td>SD 0(R1),F4</td>
<td>SD 0(R1),F4</td>
<td>SD 0(R1),F4</td>
<td>7</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F12,F10,F2</td>
<td>8</td>
</tr>
<tr>
<td>SUBI R1,R1,#4</td>
<td>BNEZ R1,LOOP</td>
<td>BNEZ R1,LOOP</td>
<td>BNEZ R1,LOOP</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

Limits to Multi-Issue Machines

- Inherent limitations of ILP
  - 1 branch in 5 instructions => 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 instructions
- 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

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)

SW Pipelining Example

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

After: Software Pipelined
1. LD F0,0(R1)
2. ADDD F4,F0,F2
3. SD 0(R1),F4
4. ADDD F6,-8(R1)
5. ADDD F0,F6,F2
6. SD -8(R1),F8
7. LD F0,-16(R1)
8. ADDD F12,F10,F2
9. SD -16(R1),F12
10. SUBI R1,R1,#4
11. BNEZ R1,LOOP
SW Pipelining Example

Symbolic Loop Unrolling
- Less code space
- Overhead paid only once vs. each iteration in loop unrolling

<table>
<thead>
<tr>
<th>Software Pipelining</th>
<th>Loop Unrolling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Overlap</td>
<td>Overlap between unrolled items</td>
</tr>
</tbody>
</table>

Number of Overlapped Operations

100 iterations = 25 loops with 4 unrolled iterations each

| Number of Overlapped Operations | Proportional to number of unrolls |

Summary
- Branch Prediction
  - Branch History Table: 2 bits for loop accuracy
  - Correlation: Recently executed branches correlated with next branch
  - Branch Target Buffer: Include branch address & prediction
- 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
  - What about non-loop codes?
    - How do you get > 1 instruction

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

TRACE REORDERING
- Reorder these instructions to improve ILP
- Fix-up instructions in case we were wrong

HW support for More ILP
- Avoid branch prediction by turning branches into conditionally executed instructions:
- If (x) then A = B op C 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, IA-64 predicated 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” squash)
  - Often try to combine with dynamic scheduling
  - Tomasulo: 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
Speculation

- 4-way issue
- All of B2 issued speculatively
- Must be squashed
- Could have execute B1 in speculative mode

HW support for More ILP

- Need HW buffer for results of uncommitted instructions:
  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

Four Steps of Speculative Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination,

2. Execution—operate on operands (EX)
   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute

3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. Commit—update register with reorder result
   When instr at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer.

Limits to ILP

- Conflicting studies of amount of parallelism available in late 1980s and early 1990s. Different assumptions about:
  - Benchmarks (vectorized Fortran FP vs. integer C programs)
  - Hardware sophistication
  - Compiler sophistication

Initial HW Model here; MIPS compilers

1. Register renaming—Infinite virtual registers and all WAW & WAR hazards are avoided
2. Branch prediction—perfect; no mispredictions
3. Jump prediction—all jumps perfectly predicted => machine with perfect speculation & an unbounded buffer of instructions available
4. Memory-address disambiguation—addresses are known & a store can be moved before a load provided addresses not equal
1 cycle latency for all instructions

Upper Limit to ILP
(Figure 4.38, page 319)
More Realistic HW: Branch Impact

Figure 4.40, Page 323

Change from infinite window to examine to 2000 and maximum issue of 64 instructions per clock cycle

More Realistic HW: Register Impact

Figure 4.44, Page 328

Change 2000 instr window, 64 instr issue, 8K 2 level Prediction

More Realistic HW: Alias Impact

Figure 4.46, Page 330

Change 2000 instr window, 64 instr issue, 8K 2 level Prediction, 256 renaming registers

Realistic HW for ‘9X: Window Impact

(Figure 4.48, Page 332)

Perfect disambiguation (HW), 1K Selective Prediction, 16 entry return, 64 registers, issue as many as window

Braniac vs. Speed Demon (Spec Ratio)

- 8-scalar IBM Power-2 @ 71.5 MHz (5 stage pipe)
- 2-scalar Alpha 21064 @ 200 MHz (7 stages)
Next Time

- Read papers and be ready to discuss them
  - Open discussion format
- Microarchitecture of Superscalar processors
- Complexity Effective Processors
- IPC vs. Clock Rate
- HW #2 Due