Lecture 5: VLIW, Software Pipelining, and Limits to ILP

Professor David A. Patterson
Computer Science 252
Fall 1996
Review: Tomasulo

- 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
- 360/91 descendants are PowerPC 604, 620; MIPS R10000; HP-PA 8000; Intel Pentium Pro
Dynamic Branch Prediction

- Performance = $f(\text{accuracy, cost of misprediction})$
- Branch History Table is simplest
  - Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):
  - 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 scheme where change prediction only if get misprediction \textit{twice}: (Figure 4.13, p. 264)

- Red: stop, not taken
- Green: go, taken
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 (in Alpha 211164)
Correlating Branches

• Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch; ie., they are correlated

• Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table

• In general, (m,n) predictor means record last m branches to select between $2^m$ history tables each with n-bit counters
  – Old 2-bit BHT is then a (0,2) predictor
Correlating Branches

(2,2) predictor
- Then behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction
Frequency of Mispredictions

Accuracy of Different Schemes
(Figure 4.21, p. 272)

4096 Entries 2-bit BHT
Unlimited Entries 2-bit BHT
1024 Entries (2,2) BHT
Re-evaluating Correlation

• Several of the SPEC benchmarks have less than a dozen branches responsible for 90% of taken branches:

<table>
<thead>
<tr>
<th>program</th>
<th>% branches</th>
<th>static</th>
<th># = 90% branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>14%</td>
<td>236</td>
<td>13</td>
</tr>
<tr>
<td>eqntott</td>
<td>25%</td>
<td>494</td>
<td>5</td>
</tr>
<tr>
<td>gcc</td>
<td>15%</td>
<td>9531</td>
<td>2020</td>
</tr>
<tr>
<td>mpeg</td>
<td>10%</td>
<td>5598</td>
<td>532</td>
</tr>
<tr>
<td>real gcc</td>
<td>13%</td>
<td>17361</td>
<td>3214</td>
</tr>
</tbody>
</table>

• Real programs + OS more like gcc
• Small benefits beyond benchmarks for correlation? problems with branch aliases
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 (Figure 4.22, p. 273)

- Return instruction addresses predicted with stack
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 (Figure 4.22, p. 273)

- Return instruction addresses predicted with stack
Dynamic Branch Prediction
Summary

• Branch History Table: 2 bits for loop accuracy
• Correlation: Recently executed branches correlated with next branch
• Branch Target Buffer: include branch address & prediction
Getting CPI < 1: Issuing Multiple Instructions/Cycle

• Two variations
• Superscalar: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
  – IBM PowerPC, Sun SuperSparc, DEC Alpha, HP 7100
• Very Long Instruction Words (VLIW): fixed number of instructions (16) scheduled by the compiler
  – Joint HP/Intel agreement in 1998?
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
## 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
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
CS 252 Administrivia

• Reading Assignments for Lectures 3 to 6
  – Chapter 4, Appendix B

• Exercises for Lectures 3 to 6
  – 4.14 parts a - k, 4.25, B.3 parts a - g, B.15
  – also look at
    – Due Monday September 16 at 5PM homework box in 283 Soda (building is locked at 6:45 PM)
    – Done in pairs, but both need to understand whole assignment

• Video in 201 McLaughlin, starting day of lecture
  Mon 9-11AM, 2 - 5 PM; Tue 9 AM - 5 PM;
  Wed 9-11AM, 2 - 10 PM; Thu 9 AM - 6 PM;
  Fri 9 - 5PM, 6 - 10 PM;
# 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>ADDD F4,F0,F2</td>
<td>ADDD F8,F6,F2</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>LD F14,-24(R1)</td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F16,F14,F2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADDD F20,F18,F2</td>
<td>ADDD F24,F22,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td></td>
<td>ADDD F28,F26,F2</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>SD -8(R1),F8</td>
<td>ADDD F28,F26,F2</td>
<td></td>
<td>SUBI R1,R1,#48</td>
<td>5</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>SD -24(R1),F16</td>
<td></td>
<td></td>
<td>BNEZ R1,LOOP</td>
<td>6</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td>SD -40(R1),F24</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>SD -0(R1),F28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</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
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
Dynamic Scheduling in Superscalar

- Dependencies stop instruction issue
- Code compiler 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 no.</th>
<th>Instructions</th>
<th>Issues</th>
<th>Executes</th>
<th>Writes result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD F0,0(R1)</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>ADDD F4,F0,F2</td>
<td>1</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>SD 0(R1),F4</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SUBI R1,R1,#8</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>BNEZ R1,LOOP</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LD F0,0(R1)</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>ADDD F4,F0,F2</td>
<td>5</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>SD 0(R1),F4</td>
<td>6</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SUBI R1,R1,#8</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>BNEZ R1,LOOP</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

\[ \approx 4 \text{ clocks per iteration} \]

Branches, Decrements still take 1 clock cycle
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
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
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.

• 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
- 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
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, its easy to undo speculated instructions on mispredicted branches or on
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
Limits to ILP

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 alias analysis*—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)

Instruction Issues per cycle

<table>
<thead>
<tr>
<th>Programs</th>
<th>gcc</th>
<th>espresso</th>
<th>li</th>
<th>fpppp</th>
<th>doducd</th>
<th>tomcatv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>54.8</td>
<td>62.6</td>
<td>17.9</td>
<td>75.2</td>
<td>118.7</td>
<td>150.1</td>
</tr>
</tbody>
</table>
More Realistic HW: Branch Impact

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Instruction issues per cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>35, 9, 6, 6, 2</td>
</tr>
<tr>
<td>espresso</td>
<td>41, 12, 7, 6, 2</td>
</tr>
<tr>
<td>li</td>
<td>16, 10, 6, 7, 2</td>
</tr>
<tr>
<td>fpppp</td>
<td>61, 48, 16, 6, 2</td>
</tr>
<tr>
<td>doducd</td>
<td>58, 46, 45, 15, 13, 14, 4</td>
</tr>
<tr>
<td>tomcatv</td>
<td>60, 46, 45, 45, 19</td>
</tr>
</tbody>
</table>

Legend:
- Perfect
- Selective predictor
- Standard 2-bit
- Static
- None

Perfect  Pick Cor. or BHT  BHT (512)  Profile
Selective History Predictor

- Branch Addr
  - 2 Global History
  - 8096 x 2 bits
  - 2048 x 4 x 2 bits

- Taken/Not Taken
  - 8K x 2 bit Selector
    - 11 Taken
    - 10
    - 01 Not Taken
    - 00

- Choose Non-correlator
  - 11
  - 10
  - 01
  - 00

- Choose Correlator
More Realistic HW: Register Impact

Figure 4.44, Page 328

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

Program

<table>
<thead>
<tr>
<th>gcc</th>
<th>espresso</th>
<th>li</th>
<th>fpppp</th>
<th>doducd</th>
<th>tomcatv</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Prediction

- Infinite
- 256
- 128
- 64
- 32
- None
More Realistic HW: Alias Impact

Figure 4.46, Page 330

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

Program

- Perfect
- Global/Stack perf
- Inspection
- None

Perfect Global/Stack perf; Inspec.
heap conflicts Assem.

None DAP.F96 38
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
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. committed/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</td>
<td>1 load +1 store</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 ($\approx$ 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
Braniac vs. Speed Demon (1993)

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

<table>
<thead>
<tr>
<th></th>
<th>Alpha 21164</th>
<th>PPro</th>
<th>HP PA-8000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Year</strong></td>
<td>1995</td>
<td>1995</td>
<td>1996</td>
</tr>
<tr>
<td><strong>Clock</strong></td>
<td>400 MHz</td>
<td>200 MHz</td>
<td>180 MHz</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>8K/8K/96K/2M</td>
<td>8K/8K/0.5M</td>
<td>0/0/2M</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

Graph showing the SPECint95base Performance for different benchmarks across three different systems: PA-8000 (green), 21164 (light blue), and PPro (red). The y-axis represents the performance level, and the x-axis lists the benchmarks: go, 88ksim, gcc, compress, li, jpeg, perl, vortex, and SPECint.
SPECfp95base Performance

DAP.F96  46
5 minute Class Break

• Lecture Format:
  – ≈ 1 minute: review last time & motivate this lecture
  – ≈ 20 minute lecture
  – ≈ 3 minutes: discuss class management
  – ≈ 25 minutes: lecture
  – 5 minutes: break
  – ≈ 25 minutes: lecture
  – ≈ 1 minute: summary of today’s important topics
Instruction Level Parallelism

• High speed execution based on *instruction level parallelism* (ilp): potential of short instruction sequences to execute in parallel

• High-speed microprocessors exploit ILP by:
  1) pipelined execution: overlap instructions
  2) superscalar execution: issue and execute multiple instructions per clock cycle
  3) Out-of-order execution (commit in-order)

• Memory accesses for high-speed microprocessor?
  – For cache hits
Problems with conventional approach

- Limits to conventional exploitation of ILP:
  1) *pipelined clock rate*: at some point, each increase in clock rate has corresponding CPI increase
  2) *instruction fetch and decode*: at some point, it's hard to fetch and decode more instructions per clock cycle
  3) *cache hit rate*: some long-running (scientific) programs have very large data sets accessed with poor locality
Vector Processors

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"
  
  e.g., $A = B \times C$, where $A$, $B$, $C$ are 64-element vectors of 64-bit floating point numbers

- Properties of vectors:
  
  - Each result independent of previous result
    => long pipeline, compiler ensures no dependencies
  
  - single vector instruction implies lots of work ($\approx$ loop)
    => fewer instruction fetches
  
  - vector instructions access memory with known pattern
    => highly interleaved memory
    => amortize memory latency of over $\approx 64$ elements
    => no caches required!
  
  - reduces branches and branch problems in pipelines
Styles of Vector Architectures

- **vector-register processors**: all vector operations between vector registers (except load and store)
  - Vector equivalent of load-store architectures
  - Includes all vector machines since late 1980s: Cray, Convex, Fujitsu, Hitachi, NEC

- **memory-memory vector processors**: all vector operations are memory to memory
Components of Vector Processor

- **Vector Register**: fixed length bank holding a single vector
  - has at least 2 read and 1 write ports
  - typically 8-16 vector registers, each holding 64-128 64-bit elements
- **Vector Functional Units (FUs)**: fully pipelined, start new operation every clock
  - typically 4 to 8: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift
- **Vector Load-Store Units (LSUs)**: fully pipelined unit to load or store a vector
- **Scalar registers**: single element for FP scalar or address
- **Cross-bar to connect FUs, LSUs, registers**
## Example Vector Machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>Year</th>
<th>Clock</th>
<th>Regs</th>
<th>Elements</th>
<th>FUs</th>
<th>LSUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray 1</td>
<td>1976</td>
<td>80 MHz</td>
<td>8</td>
<td>64</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Cray XMP</td>
<td>1983</td>
<td>120 MHz</td>
<td>8</td>
<td>64</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Cray YMP</td>
<td>1988</td>
<td>166 MHz</td>
<td>8</td>
<td>64</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Cray C-90</td>
<td>1991</td>
<td>240 MHz</td>
<td>8</td>
<td>128</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Cray T-90</td>
<td>1996</td>
<td>455 MHz</td>
<td>8</td>
<td>128</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Conv. C-1</td>
<td>1984</td>
<td>10 MHz</td>
<td>8</td>
<td>128</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Conv. C-4</td>
<td>1994</td>
<td>133 MHz</td>
<td>16</td>
<td>128</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Fuj. VP200</td>
<td>1982</td>
<td>133 MHz</td>
<td>8-256</td>
<td>32-1024</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Fuj. VP300</td>
<td>1996</td>
<td>100 MHz</td>
<td>8-256</td>
<td>32-1024</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>NEC SX/2</td>
<td>1984</td>
<td>160 MHz</td>
<td>8+8K</td>
<td>256+var</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>NEC SX/3</td>
<td>1995</td>
<td>400 MHz</td>
<td>8+8K</td>
<td>256+var</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>
## Vector Linpack Performance

<table>
<thead>
<tr>
<th>Machine</th>
<th>Year</th>
<th>Clock MHz</th>
<th>100x100</th>
<th>1kx1k Peak (Procs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray 1</td>
<td>1976</td>
<td>80</td>
<td>12</td>
<td>110 160(1)</td>
</tr>
<tr>
<td>Cray XMP</td>
<td>1983</td>
<td>120</td>
<td>121</td>
<td>218 940(4)</td>
</tr>
<tr>
<td>Cray YMP</td>
<td>1988</td>
<td>166</td>
<td>150</td>
<td>307 2,667(8)</td>
</tr>
<tr>
<td>Cray C-90</td>
<td>1991</td>
<td>240</td>
<td>387</td>
<td>902 15,238(16)</td>
</tr>
<tr>
<td>Cray T-90</td>
<td>1996</td>
<td>455</td>
<td>705</td>
<td>1603 57,600(32)</td>
</tr>
<tr>
<td>Conv. C-1</td>
<td>1984</td>
<td>10</td>
<td>3</td>
<td>-- 20(1)</td>
</tr>
<tr>
<td>Conv. C-4</td>
<td>1994</td>
<td>135</td>
<td>160</td>
<td>2531 3240(4)</td>
</tr>
<tr>
<td>Fuj. VP200</td>
<td>1982</td>
<td>133</td>
<td>18</td>
<td>422 533(1)</td>
</tr>
<tr>
<td>NEC SX/2</td>
<td>1984</td>
<td>166</td>
<td>43</td>
<td>885 1300(1)</td>
</tr>
<tr>
<td>NEC SX/3</td>
<td>1995</td>
<td>400</td>
<td>368</td>
<td>2757 25,600(4)</td>
</tr>
</tbody>
</table>
## Vector Instructions

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Operands</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV</td>
<td>V1, V2, V3</td>
<td>V1=V2+V3</td>
<td>vector + vector</td>
</tr>
<tr>
<td>ADDS</td>
<td>V1, F0, V2</td>
<td>V1=F0+V2</td>
<td>scalar + vector</td>
</tr>
<tr>
<td>MULV</td>
<td>V1, V2, V3</td>
<td>V1=V2xV3</td>
<td>vector x vector</td>
</tr>
<tr>
<td>MULS</td>
<td>V1, F0, V2</td>
<td>V1=F0xV2</td>
<td>scalar x vector</td>
</tr>
<tr>
<td>LV</td>
<td>V1, R1</td>
<td>V1=M[R1..R1+63]</td>
<td>load, stride=1</td>
</tr>
<tr>
<td>LVWS</td>
<td>V1, R1, R2</td>
<td>V1=M[R1..R1+63*R2]</td>
<td>load, stride=R2</td>
</tr>
<tr>
<td>LVI</td>
<td>V1, R1, V2</td>
<td>V1=M[R1+V2i,i=0..63]</td>
<td>&quot;gather&quot;</td>
</tr>
<tr>
<td>CeqV</td>
<td>VM, V1, V2</td>
<td>VMASKi = (V1i=V2i)?</td>
<td>comp. setmask</td>
</tr>
<tr>
<td>MOV</td>
<td>VLR, R1</td>
<td>Vec. Len. Reg. = R1</td>
<td>set vector length</td>
</tr>
<tr>
<td>MOV</td>
<td>VM, R1</td>
<td>Vec. Mask = R1</td>
<td>set vector mask</td>
</tr>
</tbody>
</table>
DAXPY \((Y = a \times X + Y)\)

Assuming vectors \(X, Y\) are length 64

Scalar vs. Vector

LD F0,a ;load scalar a
LD V1,Rx ;load vector X
MULS V2,F0,V1 ;vector-scalar mult.
LV V3,Ry ;load vector Y
ADDV V4,V2,V3 ;add
SD Ry,V4 ;store the result

LD F0,a
ADDI R4,Rx,512 ;last address to load
loop: LD F2,0(Rx) ;load X(i)
      MULD F2,F0,F2 ;a*X(i)
      LD F4,0(Ry) ;load Y(i)
      ADDD F4,F2,F4 ;a*X(i) + Y(i)
      SD F4,0(Ry) ;store into Y(i)
      ADDI Rx,Rx,#8 ;increment index to X
      ADDI Ry,Ry,#8 ;increment index to Y
      SUB R20,R4,Rx ;compute bound
      BNZ R20,loop ;check if done

578 \((2+9\times64)\) vs. 6 instructions:
64 operation vectors + no loop overhead
also fewer pipeline hazards
Vector Execution Time

- Time = \( f(\text{vector length, data dependencies, hazards}) \)
- **Initiation rate**: rate that FU consumes vector elements (usually 1, 2 on T-90)
- **Convoy**: set of vector instructions that can begin execution in same clock (no hazards)
- **Chime**: approx. time for a vector operation
- \( m \) convoys take \( m \) chimes; if each vector length is \( n \), then they take approx. \( m \times n \) clock cycles (ignores overhead)

1: \( \text{LV} \quad V_1,Rx \) ;load vector \( X \)
2: \( \text{MULS} \quad V_2,F_0,V_1 \) ;vector-scalar mult.
   \( \text{LV} \quad V_3,Ry \) ;load vector \( Y \)
3: \( \text{ADDV} \quad V_4,V_2,V_3 \) ;add
4: \( \text{SV} \quad Ry,V_4 \) ;store the result

4 conveys \( \implies 4 \times 64 \approx 256 \) clocks
Start-up Time

• **Start-up time**: pipeline latency time (depth of FU pipeline)
• Operation Start-up penalty
• Vector load/store 12
• Vector multiply 7
• Vector add 6
  – Assumes convoys don't overlap; vector length = n

<table>
<thead>
<tr>
<th>Convoy</th>
<th>Start</th>
<th>1st result</th>
<th>last result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. LV</td>
<td>0</td>
<td>12</td>
<td>11+n</td>
</tr>
<tr>
<td>2. MULV, LV</td>
<td>12+n</td>
<td>12+n+12</td>
<td>24+2n</td>
</tr>
<tr>
<td>3. ADDV</td>
<td>25+2n</td>
<td>25+2n+6</td>
<td>31+3n</td>
</tr>
<tr>
<td>4. SV</td>
<td>32+3n</td>
<td>32+3n+12</td>
<td>42+4n</td>
</tr>
</tbody>
</table>
Vector Load/Store Units & Memories

- Start-up overheads usually longer for LSUs
- Memory system must sustain 1 word/clock cycle
- Many Vector Procs. use banks vs. simple interleaving:
  1) support multiple loads/stores per cycle
     => multiple banks & address banks independently
  2) support non-sequential accesses
- Note: No. memory banks > memory latency to avoid stalls
Summary

• 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

• Vector
  – Alternate model accommodates long memory latency
  – Much easier for hardware: more powerful instructions, more predictable memory accesses, fewer branches, ...
  – What % of computation is vectorizable? For new apps?