Getting CPI under 1: Outline

• More ILP
  – VLIW
  – branch target buffer
  – return address predictor
  – superscalar
  – more register renaming
  – value prediction
  – conditional instructions
  – speculative loads
  – superscalar
• Limits to ILP
• Threading
  – fine/coarse
  – simultaneous multithreading
Getting CPI below 1

• CPI ≥ 1 if issue only 1 instruction every clock cycle
• Multiple-issue processors come in 3 flavors:
  – Statically-scheduled *superscalar* processors
  – Dynamically-scheduled superscalar processors
  – VLIW (very long instruction word) processors

Getting CPI below 1

• 2 types of superscalar processors issue varying numbers of instructions per clock
  – Use in-order execution if they are statically scheduled, or
  – Out-of-order execution if they are dynamically scheduled
• VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction (Intel/HP Itanium)
**VLIW: Very Large Instruction Word**

- Each “instruction” has explicit coding for multiple operations
  - In IA-64, grouping called a “packet”
  - In Transmeta, grouping called a “molecule” (with “atoms” as ops)
- 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 are independent ⇒ 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

---

**Recall: Unrolled Loop that Minimizes Stalls for Scalar**

1 Loop: 

<table>
<thead>
<tr>
<th>1</th>
<th>L.D</th>
<th>F0, 0 (R1)</th>
<th>L.D to ADD.D: 1 Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>L.D</td>
<td>F6, -8 (R1)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>L.D</td>
<td>F10, -16 (R1)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>L.D</td>
<td>F14, -24 (R1)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ADD.D</td>
<td>F4, F0, F2</td>
<td>ADD.D to S.D: 2 Cycles</td>
</tr>
<tr>
<td>6</td>
<td>ADD.D</td>
<td>F8, F6, F2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ADD.D</td>
<td>F12, F10, F2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ADD.D</td>
<td>F16, F14, F2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>S.D</td>
<td>0 (R1), F4</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>S.D</td>
<td>-8 (R1), F8</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>DSUBUI</td>
<td>R1, R1, #32</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>S.D</td>
<td>-16 (R1), F12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>BNEZ</td>
<td>R1, LOOP</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>S.D</td>
<td>8 (R1), F16; 8-32 = -24</td>
<td></td>
</tr>
</tbody>
</table>

14 clock cycles, or 3.5 per iteration
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>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>S.D 0(R1),F4</td>
<td>S.D -8(R1),F8</td>
<td>ADD.D F28,F26,F2</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>S.D -16(R1),F12</td>
<td>S.D -24(R1),F16</td>
<td>DSUBUI R1,R1,#48</td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>S.D 16(R1),F20</td>
<td>S.D 8(R1),F24</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>S.D -0(R1),F28</td>
<td>BNEZ R1,LOOP</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency
Note: Need more registers in VLIW (15 vs. 6 in SS)

Problems with 1st Generation VLIW

- Increase in code size
  - Generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - Whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding
Problems with 1st Generation VLIW

- Operated in lock-step; no hazard detection HW
  - Stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Caches hard to predict
- Binary code compatibility
  - Pure VLIW $\Rightarrow$ different numbers of functional units and unit latencies require different versions of the code

Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- IA-64: instruction set architecture
- 128 64-bit integer regs + 128 82-bit floating point regs
  - Not separate register files per functional unit as in old VLIW
- Hardware checks dependencies (interlocks $\Rightarrow$ binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags) $\Rightarrow$ 40% fewer mispredictions?
Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- **Itanium™** was first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 µ process
- **Itanium 2™** is name of 2nd implementation (2005)
  - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 µ process
  - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3

**IF BW: Return Address Predictor**

- Small buffer of return addresses acts as a stack
- Caches most recent return addresses
- Call ⇒ Push a return address on stack
- Return ⇒ Pop an address off stack & predict as new PC

Figure 2.25 – SPEC95
More Instruction Fetch Bandwidth

- Integrated branch prediction
  - branch predictor is part of instruction fetch unit and is constantly predicting branches
- Instruction prefetch
  - Instruction fetch units prefetch to deliver multiple instructions per clock, integrating it with branch prediction
- Instruction memory access and buffering
  - Fetching multiple instructions per cycle:
    » May require accessing multiple cache blocks (prefetch to hide cost of crossing cache blocks)
    » Provides buffering, acting as on-demand unit to provide instructions to issue stage as needed and in quantity needed

Speculation: Register Renaming vs. ROB

- Alternative to ROB is a larger physical set of registers combined with register renaming
  - Extended registers replace function of both ROB and reservation stations
- Instruction issue maps names of architectural registers to physical register numbers in extended register set
  - On issue, allocates a new unused register for the destination (which avoids WAW and WAR hazards)
  - Speculation recovery easy because a physical register holding an instruction destination does not become the architectural register until the instruction commits
- Most Out-of-Order processors today use extended registers with renaming
Value Prediction

- Attempts to predict value produced by instruction
  - E.g., Loads a value that changes infrequently
- Value prediction is useful only if it significantly increases ILP
  - Focus of research has been on loads; so-so results, no processor uses value prediction
- Related topic is address aliasing prediction
  - RAW for load and store or WAW for 2 stores
- Address alias prediction is both more stable and simpler since need not actually predict the address values, only whether such values conflict
  - Has been used by a few processors

Conditional (Predicated) Instructions

- Condition is evaluated as part of the instruction execution
  - if condition true, normal execution
  - if condition false, instruction turned into a no-op
  - IA-64 has a form of these
- Example: conditional move
  - Move a value from one register to another if condition is true
  - Can eliminate a branch in simple code sequences
Example: Conditional Move

- For code:  
  if (A==0) { S=T; }
  - Assume R1, R2, R3 hold values of A, S, T

  With branch:
  BNEZ R1, L
  ADDU R2, R3, R0
  L:

  With conditional move (if 3rd operand equals zero):
  CMOVZ R2, R3, R1

  - Converts the control dependence into a data dependence
    - For a pipeline, moves the dependence from near beginning of pipeline (branch resolution) to end (register write)

Limitations of Conditional Instructions

- Predicated instructions that are squashed still use processor resources
  - Doesn’t matter if resources would have been idle anyway

- Most useful when predicate can be evaluated early
  - Want to avoid data hazards replacing control hazards

- Hard to do for complex control flow
  - For example, moving across multiple branches

- Conditional instructions may have higher cycle count or slower clock rate than unconditional ones
Compiler Speculation w/ Hardware Support

• To move speculated instructions not just before branch, but before condition evaluation
• Compiler can help find instructions that can be speculatively moved and not affect program data flow
• Hard part is preserving exception behavior
  – A speculated instruction that is mispredicted should not cause an exception
  – It can be done, but details are rather complex

Memory Reference Speculation w/ Hardware Support

• To move loads across stores, when compiler can’t be sure it is legal
• Use a speculative load instruction
  – Hardware saves address of memory location
  – If a subsequent store changes that location before the check (to end the speculation), then the speculation failed, otherwise it succeeded
  – On failure, need to redo load and re-execute all speculated instructions after the speculative load
Superscalar Execution

- Predication helps with scheduling
- Example: superscalar that can issue 1 memory reference and 1 ALU op per cycle, or just 1 branch

LWC loads if 3rd operand not 0

<table>
<thead>
<tr>
<th>1st instruction</th>
<th>2nd instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1,40(R2)</td>
<td>ADD R3,R4,R5</td>
</tr>
<tr>
<td></td>
<td>ADD R6,R3,R7</td>
</tr>
<tr>
<td>BEQZ R10,L</td>
<td></td>
</tr>
<tr>
<td>LW R8,0(R10)</td>
<td></td>
</tr>
<tr>
<td>LW R9,0(R8)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st instruction</th>
<th>2nd instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1,40(R2)</td>
<td>ADD R3,R4,R5</td>
</tr>
<tr>
<td>LWC R8,0(R10),R10</td>
<td>ADD R6,R3,R7</td>
</tr>
<tr>
<td>BEQZ R10,L</td>
<td></td>
</tr>
<tr>
<td>LW R9,0(R8)</td>
<td></td>
</tr>
</tbody>
</table>