Overview

- Last 3 lectures: binary compatibility and exploiting ILP in hardware: BTB, ROB, Reservation Stations, ...
- How far can you go in compiler?
- What if you can also change instruction set architecture?
- Will see multi billion dollar gamble by two Bay Area firms for the future of computer architecture: HP and Intel to produce IA-64
  - 7 years in the making?

Review: Dynamic Examples

- P6 (Pentium Pro, II, III) successful micro-architecture, even with imitator (AMD Athlon)
  - Translate most 80x86 instructions to micro-operations
  - Longer pipeline than RISC instructions
  - Dynamically execute micro-operations
- “Netburst” (Pentium 4, ...) success not clear
  - Much longer pipeline, higher clock rate in same technology as P6
  - Trace Cache to capture micro-operations, avoid hardware translation
- Multithreading to increase performance for servers, parallel programs written to use threads
  - Extra copies of PCs, Registers per thread: e.g., IBM AS/400
- Simultaneous Multithreading (SMT) exploit underutilized Dynamic Execution HW to get higher throughput at low extra cost?

Static Branch Prediction

- Simplest: Predict taken
  - average misprediction rate = untaken branch frequency, which for the SPEC programs is 34%.
  - Unfortunately, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%)
- Predict on the basis of branch direction?
  - choosing backward-going branches to be taken (loop)
  - forward-going branches to be not taken (if)
  - SPEC programs, however, most forward-going branches are taken => predict taken is better
- Predict branches on the basis of profile information collected from earlier runs
  - Misprediction varies from 5% to 22%

Running Example

- This code, a scalar to a vector:
  for (i=1000; i>0; i=i–1)
    x[i] = x[i] + s;
- Assume following latency all examples

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Execution in cycles</th>
<th>Latency in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

FP Loop: Where are the Hazards?

- First translate into MIPS code:
  - To simplify, assume 8 is lowest address

```plaintext
Loop: L.D F0,0(R1) ; F0=vector element
ADD D F4,F0,F2 ; add scalar from F2
S.D 0(R1),F4 ; store result
DSUBUI 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

1 Loop: L.D F0,0(R1) ; F0=vector element
2 stall
3 ADD D F4,F0,F2 ; add scalar in F2
4 stall
5 stall
6 S.D 0(R1),F4 ; store result
7 DSUBUI R1,R1,8 ; decrement pointer R1 (DW)
8 BNEZ R1,Loop ; branch R1!=zero
9 stall ; delayed branch slot

Instruction | Instruction | Latency in producing result | Latency in using result | clock cycles
--- | --- | --- | --- | ---
FP ALU op | Another FP ALU op | 3 | 3 | 6
Load double | FP ALU op | 1 | 2 | 3

9 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

1 Loop: L.D F0,0(R1)
2 stall
3 ADD D F4,F0,F2
4 DSUBUI R1,R1,8 ; altered when move past DSUBUI
5 BNEZ R1,Loop ; delayed branch slot
6 S.D 8(R1),F4

Swap BNEZ and S.D by changing address of S.D

Instruction | Instruction | Latency in producing result | Latency in using result | clock cycles
--- | --- | --- | --- | ---
FP ALU op | Another FP ALU op | 3 | 3 | 6
FP ALU op | Store double | 2 | 1 | 3

6 clocks, but just 3 for execution, 3 for loop overhead: How make faster?

Unroll Loop Four Times (straightforward way)

1 Loop: L.D F0,0(R1)
2 ADD D F4,F0,F2
3 S.D 0(R1),F4 ; drop DSUBUI & BNEZ
4 L.D F6,-8(R1)
5 ADD D F8,F6,F2
6 DSUBUI R1,R1,8 ; drop DSUBUI & BNEZ
7 S.D 0(R1),F4
8 ADD D F8,F6,F2
9 DSUBUI R1,R1,32 ; alter to 4*8
10 BNEZ R1,LOOP

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: L.D F0,0(R1)
2 ADD D F4,F0,F2
3 S.D 0(R1),F4 ; drop DSUBUI & BNEZ
4 L.D F6,-8(R1)
5 ADD D F8,F6,F2
6 DSUBUI R1,R1,8 ; drop DSUBUI & BNEZ
7 S.D 0(R1),F4
8 ADD D F8,F6,F2
9 S.D -32(R1),F8
10 L.D F4,-24(R1)
11 ADD D F8,F6,F2
12 S.D -24(R1),F16
13 DSUBUI R1,R1,F32 ; alter to 4*8
14 BNEZ R1,LOOP
15 NOP

Unrolled Loop Detail

• Do not usually know upper bound of loop
• Suppose it is n, and we would like to unroll the loop to make k copies of the body
• Instead of a single unrolled loop, we generate a pair of consecutive loops:
  1st executes (n mod k) times and has a body that is the original loop
  2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
• For large values of n, most of the execution time will be spent in the unrolled loop

14 clock cycles, or 3.5 per iteration

Compiler Perspectives on Code Movement

• Compiler concerned about dependencies in program
• Whether or not a HW hazard depends on pipeline
• Try to schedule to avoid hazards that cause performance losses
• (True) Data dependencies (RAW if a hazard for HW)
  - Instruction j produces a result used by instruction j+1
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
• If dependent, can’t execute in parallel
• Easy to determine for registers (fixed names)
• Hard for memory (“memory disambiguation” problem):
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R5)?
Where are the name dependencies?

1. Loop: L.D F0,0(R1)
2. ADD.D F4,F0,F2
3. S.D 0(R1),F4 ; drop DSUBUI & BNEZ
4. L.D F0,-8(R1)
5. ADD.D F4,F0,F2
6. S.D -8(R1),F4 ; drop DSUBUI & BNEZ
7. L.D F0,-16(R1)
8. ADD.D F4,F0,F2
9. S.D -16(R1),F4 ; drop DSUBUI & BNEZ
10. L.D F0,-24(R1)
11. ADD.D F4,F0,F2
12. S.D -24(R1),F4
13. DSUBUI R1,R1,#32 ; alter to 4*8
14. BNEZ R1,LOOP
15. NOP

How can remove them?

Where are the name dependencies?

1. Loop: L.D F0,0(R1)
2. ADD.D F4,F0,F2
3. S.D 0(R1),F4 ; drop DSUBUI & BNEZ
4. L.D F0,-8(R1)
5. ADD.D F4,F0,F2
6. S.D -8(R1),F4 ; drop DSUBUI & BNEZ
7. L.D F0,-16(R1)
8. ADD.D F4,F0,F2
9. S.D -16(R1),F4 ; drop DSUBUI & BNEZ
10. L.D F0,-24(R1)
11. ADD.D F4,F0,F2
12. S.D -24(R1),F4
13. DSUBUI R1,R1,#32 ; alter to 4*8
14. BNEZ R1,LOOP
15. NOP

The Original "register renaming"

Compiler Perspectives on Code Movement

- Name Dependencies are Hard to discover for Memory Accesses
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- Our example required compiler to know that if R1 doesn’t change then:
  0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)

There were no dependencies between some loads and stores so they could be moved by each other.

Steps Compiler Performed to Unroll

- Check OK to move the S.D after DSUBUI and BNEZ, and find amount to adjust S.D offset
- Determine unrolling the loop would be useful by finding that the loop iterations were independent
- Rename registers to avoid name dependencies
- Eliminate extra test and branch instructions and adjust the loop termination and iteration code
- Determine loads and stores in unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
  - requires analyzing memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield same result as the original code.

Administrativa

- 3rd (last) Homework on Ch 3 due Saturday
- 3rd project meetings 4/11: signup today
- Project Summary due Monday night
- Quiz #2 4/18 310 Soda at 5:30

Another possibility:

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. L.D F0,0(R1)
2. ADD.D F4,F0,F2
3. S.D 0(R1),F4
4. L.D F6,-8(R1)
5. ADD.D F8,F6,F2
6. S.D -8(R1),F8
7. L.D F10,-16(R1)
8. ADD.D F12,F10,F2
9. S.D -16(R1),F12
10. DSUBUI R1,R1,#24
11. BNEZ R1,LOOP

After: Software Pipelined

1. S.D 0(R1),F4 ; Stores M[i]
2. ADD.D F4,F0,F2 ; Adds to M[i-1]
3. L.D F0,-16(R1); Loads M[i-2]
4. DSUBUI 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

5 cycles per iteration

When Safe to Unroll Loop?

• Example: Where are data dependencies?
  (A,B,C distinct & nonoverlapping)

```c
for (i=0; i<100; i=i+1) {
  A[i+1] = A[i] + C[i];    /* S1 */
  B[i+1] = B[i] + A[i+1];  /* S2 */
}
```

1. S2 uses the value, A[i+1], computed by S1 in the same iteration.
2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].

This is a "loop-carried dependence": between iterations

• For our prior example, each iteration was distinct
  - Right????

Does a loop-carried dependence mean there is no parallelism???

• Consider:
  ```c
  for (i=0; i< 8; i=i+1) {
    A = A + C[i];    /* S1 */
  }
  ```

  Could compute:
  ```
  "Cycle 1":
  temp1 = C[0] + C[1];
  temp2 = C[2] + C[3];
  temp3 = C[4] + C[5];
  temp4 = C[6] + C[7];
  "Cycle 2":
  temp5 = temp1 + temp2;
  temp6 = temp3 + temp4;
  "Cycle 3":
  A = temp5 + temp6;
  ```

  Relies on associative nature of "+".
  - See "Parallelizing Complex Scans and Reductions" by Allan Fisher and Anwar Shulam (handed out next week)

Hardware Support for Exposing More Parallelism at Compile-Time

• Conditional or Predicated Instructions
  - Discussed before in context of branch prediction
  - Conditional instruction execution
    ```c
    L1:
    LW    R1,40(R2) ADD R3,R4,R5
    BEQZ  R10,L
    LW    R9,0(R8)
    ```
  - Waste slot since 3rd LW dependent on result of 2nd LW

Exception Behavior Support

• Several mechanisms to ensure that speculation by compiler does not violate exception behavior
  - For example, cannot raise exceptions in predicated code if annulled
  - Prefetch does not cause exceptions
Hardware Support for Memory Reference Speculation

- To compiler to move loads across stores, when it cannot be absolutely certain that such a movement is correct, a special instruction to check for address conflicts can be included in the architecture.
  - The special instruction is left at the original location of the load and the load is moved up across stores.
  - When a speculated load is executed, the hardware saves the address of the accessed memory location.
  - If a subsequent store changes the location before the check instruction, then the speculation has failed.
  - If only load instruction was speculated, then it suffices to redo the load at the point of the check instruction.

What if Can Chance Instruction Set?

- Superscalar processors decide on the fly how many instructions to issue.
  - HW complexity of Number of instructions to issue $O(n^2)$.
- Why not allow compiler to schedule instruction level parallelism explicitly?
  - Format the instructions in a potential issue packet so that HW need not check explicitly for dependences.

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.
  - 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
  - Need compiling technique that schedules across several branches.

Recall: Unrolled Loop that Minimizes Stalls for Scalar

1 Loop: L D   F0, 0(R1) L D to ADD D: 1 Cycle
  2 L D   F6, 0(R1) ADD D to S D: 2 Cycles
  3 L D   F10, 0(R1) ADD D to S D: 2 Cycles
  4 ADD D   F4, F2 ADD D to S D: 2 Cycles
  5 ADD D   F8, F2 ADD D to S D: 2 Cycles
  6 ADD D   F12, F2 ADD D to S D: 2 Cycles
  7 ADD D   F16, F2 ADD D to S D: 2 Cycles
  8 ADD D   F20, F2 ADD D to S D: 2 Cycles
  9 S D   0(R1), F4 S.D to BNEZ: 2 Cycles
 10 S D   0(R1), F6 ADD D to S D: 2 Cycles
 11 S D   0(R1), F8 ADD D to S D: 2 Cycles
 12 S D   0(R1), F10 ADD D to S D: 2 Cycles
 13 S D   0(R1), F12 ADD D to S D: 2 Cycles
 14 DSUBUI R1, R1, #32 ADD D to S D: 2 Cycles
 15 BNEZ R1, LOOP  ADD D to S D: 2 Cycles
 16 S D   0(R1), F16 S.D to BNEZ: 2 Cycles
 17 S D   0(R1), F18 S.D to BNEZ: 2 Cycles
 18 S D   0(R1), F20 S.D to BNEZ: 2 Cycles
 19 S D   0(R1), F22 S.D to BNEZ: 2 Cycles
 20 S D   0(R1), F24 S.D to BNEZ: 2 Cycles
 21 S D   0(R1), F26 S.D to BNEZ: 2 Cycles
 22 S D   0(R1), F28 S.D to BNEZ: 2 Cycles
 23 BNEZ R1, LOOP  S.D to BNEZ: 2 Cycles
 24 14 clock cycles, or 3.5 per iteration

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)

Recall: 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).
Recall: Software Pipelining Example

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

After: Software Pipelined
1  S.D 0(R1),F4 ; Stores M[i]
2  ADD.D F4,F0,F2 ; Adds to M[i-1]
3  L.D F0,-16(R1) ; Loads M[i-2]
4  DSUBUI 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

SW Pipeline
Loop Unrolled

Software Pipelining with Loop Unrolling in VLIW

Memory reference 1 Memory reference 2 FP operation 1 FP op. 2 Int. op/ Clock
L.D F0,-48(R1) ST 0(R1),F4 ADD.D F4,F0,F2 1
L.D F6,-56(R1) ST -8(R1),F8 ADD.D F8,F6,F2 DSUBUI R1,R1,#24 2
L.D F10,-40(R1) ST 8(R1),F12 ADD.D F12,F10,F2 BNEZ R1,LOOP 3

• 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 fewer registers for software pipelining (only using 7 registers here, was using 15)

Trace Scheduling

• Parallelism across IF branches vs. LOOP branches?
• Two steps:
  - Trace Selection
    - Find likely sequence of basic blocks (traces) of (statically predicted or profile predicted) long sequence of straight-line code
  - Trace Compaction
    - Squeeze trace into few VLIW instructions
• This is a form of compiler-generated speculation
  - Compiler must generate “fixup” code to handle cases in which trace is not the taken branch
  - Needs extra registers: undo bad guess by discarding
• Subtle compiler bugs mean wrong answer vs. poorer performance; no hardware interlocks

Advantages of HW (Tomasulo) vs. SW (VLIW) Speculation

• HW advantages:
  - HW better at memory disambiguation since knows actual addresses
  - HW better at branch prediction since lower overhead
  - HW maintains precise exception model
  - HW does not execute bookkeeping instructions
  - Some software works across multiple implementations
  - Smaller code size (not as many nops filling blank instructions)
• SW advantages:
  - Window of instructions that is examined for parallelism much higher
  - Much less hardware involved in VLIW (unless you are Intel...!)
  - More involved types of speculation can be done more easily
  - Speculation can be based on large-scale program behavior, not just local information

Superscalar v. VLIW

• Smaller code size
• Binary compatibility across generations of hardware
• Simplified Hardware for decoding, issuing instructions
• No Interlock Hardware (compiler checks?)
• More registers, but simplified Hardware for Register Ports (multiple independent register files?)

Problems with First Generation VLIW

• Increased 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
• Operated in lock-step; no hazard detection HW
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might prediction function units, but caches hard to predict
• Binary code compatibility
  - Pure VLIW => 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; EPIC is type
- **EPIC** = 2nd generation VLIW?
- **Itanium™** is name of first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 500Mhz on 0.18 µ process
- 128 64-bit integer registers + 128 82-bit floating point registers
- Not separate register files per functional unit as in old VLIW
- Hardware checks dependencies (interlocks => binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags) => 40% fewer mispredictions?

**IA-64 Registers**

- Both the integer and floating point registers support register rotation for registers 32-128.
- Register rotation is designed to ease the task of allocating registers in software pipelined loops
- When combined with predication, possible to avoid the need for unrolling and for separate prologue and epilogue code for a software pipelined loop
- Makes the SW-pipelining usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages

5 Types of Execution in Bundle

<table>
<thead>
<tr>
<th>Execution</th>
<th>Instruction Unit</th>
<th>Slot</th>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-unit</td>
<td>A</td>
<td>Integer ALU</td>
<td>add, subtract, and, or, cmp</td>
<td>Floating point addition</td>
<td></td>
</tr>
<tr>
<td>I-unit</td>
<td>B</td>
<td>Memory access</td>
<td>Loads, stores for int/FP regs</td>
<td>Memory load operation</td>
<td></td>
</tr>
<tr>
<td>F-unit</td>
<td>A</td>
<td>Integer ALU</td>
<td>add, subtract, and, or, cmp</td>
<td>Floating point addition</td>
<td></td>
</tr>
<tr>
<td>M-unit</td>
<td>A</td>
<td>Memory access</td>
<td>Loads, stores for int/FP regs</td>
<td>Memory load operation</td>
<td></td>
</tr>
<tr>
<td>B-unit</td>
<td>B</td>
<td>Branches</td>
<td>Conditional branches, calls</td>
<td>Branch to function call</td>
<td></td>
</tr>
<tr>
<td>L+X</td>
<td>L+X</td>
<td>Extended</td>
<td>Extended immediates, stops</td>
<td>Extended immediates, stops</td>
<td></td>
</tr>
</tbody>
</table>

- 5-bit template field within each bundle describes both the presence of any stops associated with the bundle and the execution unit type required by each instruction within the bundle (see Fig 4.12 page 271).
Itanium™ Machine Characteristics
(Copyright: Intel at Hotchips ’00)

- Frequency: 933 MHz
- Transistor Count: 25 NM CPU, 25NM L3
- Process: 0.18 µm CMOS, 9 metal layer
- Package: Organic Lead Grid Array
- Machine Width: 8 instructions (4 ALU/FP, 2 LDF, 2 FP, 2 Br)
- Registers: 16 primary (2 in L2, 8 in L3, 6 of Predicates)
- Operation: 52 entry ALAT, Exception Deferral
- Branch Prediction: Multilevel 4-stage Prediction Hierarchy
- FP Compute Bandwidth: 2.1 GFlops (DP), 4.2 GFlops (SP)
- L1 Cache: 64K entry, 32/32, 1 level DTLB, VHPT
- CCT Latency: 6.7 cycles
- Cache: 2.1 GB/sec, 4-way s.a., BW of 12.8 GB/sec
- System Bus: Scalable to large (512+ proc) systems

Itanium™ EPIC Design Maximizes SW–HW Synergy
(Copyright: Intel at Hotchips ’00)

- Architecture Features programmed by compiler:
  - Branch Hints
  - Explicit Parallelism
  - Register Stack Predication
  - Data & Control Speculation
  - Memory Hints

Itanium processor 10-stage pipeline

- Front-end (stages IPG, Fetch, and Rotate): prefetches up to 32 bytes per clock (2 bundles) into a prefetch buffer, which can hold up to 8 bundles (24 instructions)
  - Branch prediction is done using a multilevel adaptive predictor like P6 microarchitecture
  - Instruction delivery (stages EXP and REN): distributes up to 6 instructions to the 9 functional units
  - Implements registers renaming for both rotation and register stacking

Comments on Itanium

- Remarkably, the Itanium has many of the features more commonly associated with the dynamically-scheduled pipelines
  - Strong emphasis on branch prediction, register renaming, scoreboardding, a deep pipeline with many stages before execution (to handle instruction alignment, renaming, etc.), and several stages following execution to handle exception detection
  - Surprising that an approach whose goal is to rely on compiler technology and simpler HW seems to be at least as complex as dynamically scheduled processors
Performance of IA-64 Itanium?

- Despite the existence of silicon, no significant standard benchmark results are available for the Itanium.
- Whether this approach will result in significantly higher performance than other recent processors is unclear.
- The clock rate of Itanium (733 MHz) is competitive but slower than the clock rates of several dynamically-scheduled machines, which are already available, including the Pentium III, Pentium 4, and AMD Athlon.

Summary#1: Hardware versus Software Speculation Mechanisms

- To speculate extensively, must be able to disambiguate memory references.
  - Much easier in HW than in SW for code with pointers.
- HW-based speculation works better when control flow is unpredictable, and when HW-based branch prediction is superior to SW-based branch prediction done at compile time.
  - Mispredictions mean wasted speculation.
- HW-based speculation maintains precise exception model even for speculated instructions.
- HW-based speculation does not require compensation or bookkeeping code.

Summary#2: Hardware versus Software Speculation Mechanisms cont'd

- Compiler-based approaches may benefit from the ability to see further in the code sequence, resulting in better code scheduling.
- HW-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementations of an architecture.
  - may be the most important in the long run.

Summary #3: Software Scheduling

- Instruction Level Parallelism (ILP) found either by compiler or hardware.
- Loop level parallelism is easiest to see.
  - SW dependencies/compiler sophistication determine if compiler can unroll loops.
  - Memory dependencies hardest to determine => Memory disambiguation
  - Very sophisticated transformations available.
- Trace Scheduling to Parallelize If statements
- Superscalar and VLIW: CPI < 1 (IPC > 1)
  - Dynamic issue vs. Static issue.
  - More instructions issue at same time => larger hazard penalty.
  - Limitation is often number of instructions that you can successfully fetch and decode per cycle.