Instruction Level Parallelism (Chapter 3)

Further ILP via Speculation

Review from Last Time #1
- Leverage Implicit Parallelism for Performance: Instruction Level Parallelism
- Loop unrolling by compiler to increase ILP
- Branch prediction to increase ILP
- Dynamic HW exploiting ILP
  - Works when can’t know dependence at compile time
  - Can hide L1 cache misses
  - Code for one machine runs well on another

Review from Last Time #2
- Reservations stations: renaming to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards
  - Allows loop unrolling in HW
- Not limited to basic blocks
  (integer units get ahead, beyond branches)
- Helps cache misses as well
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- 360/91 descendants are Pentium 4, Power 5, AMD Athlon/Opteron, …

Outline
- ILP
- Speculation
- Speculative Tomasulo Example
- Memory Aliases
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction
Speculation For Greater ILP

- *In Tomasulo, to preserve exception behavior, no instruction is allowed to initiate execution until all preceding branches have completed.*

- Greater ILP: Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct
  - Speculation $\Rightarrow$ fetch, issue, and execute instructions as if branch predictions were always correct
  - Dynamic scheduling $\Rightarrow$ only fetches and issues instructions
- Essentially, a data flow execution model: Operations execute as soon as their operands are available

Adding Speculation to Tomasulo

- Must separate execution from allowing instruction to finish or "commit"
  - This additional step is called instruction commit; it occurs whenever the branch prediction is confirmed for the branch immediately before a block of speculated instructions.
- When an instruction is no longer speculative, allow it to update the register file or memory.
- Requires an additional set of buffers to hold results of instructions that have finished execution but have not committed.
  - This *reorder buffer (ROB)* is also used to pass results among instructions that may be speculated.

Speculation For Greater ILP

- Three components of HW-based speculation:
  1. Dynamic branch prediction to choose which instructions to execute
  2. Speculation to allow execution of instructions before control dependences are resolved
     - + Ability to *undo* effects of incorrectly speculated sequence
  3. Dynamic scheduling to deal with scheduling of combinations of basic blocks revealed at runtime

Reorder Buffer (ROB)

- In Tomasulo’s algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits
  - (i.e., when we know for sure that the instruction should have executed)
- The ROB supplies operands in the interval between completion of instruction execution and instruction commit
  - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo’s algorithm
  - ROB extends architected registers as the reservation stations did
Reorder Buffer Entry Fields

- Each entry in the ROB contains four fields:

  1. **Instruction type**
     - A branch (has no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, each of which has a register destination for writeback).
  2. **Destination**
     - Register number (for loads and ALU operations) or memory address (for stores) - where the instruction result should be written.
  3. **Value**
     - Value of instruction result being held until the instruction commits.
  4. **Ready**
     - Indicates that instruction has completed execution, and the value is ready once the instruction commits.

Reorder Buffer Operation

- Holds instructions in FIFO order, exactly as issued.
- When instructions complete, results placed into ROB.
- Supplies operands to other instruction between execution complete & commit ⇒ more registers like RSs (reservation stations).
- Tag results with ROB buffer number instead of reservation station number.
- Instructions commit ⇒ values at head of ROB placed in registers.
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions.

4 Steps of Speculative Tomasulo Algorithm (steps added for speculation in blue)

1. **Issue**—get instruction from FP Op Queue
   - If RS and ROB slot free, issue instruction & send the operands to RS if available in registers or ROB, and send ROB no. for destination to RS (this stage sometimes called "dispatch").
2. **Execution**—operate on operands (EX)
   - Checks for RAW hazards; when both operands ready then execute; if not ready, watch CDB for result; when both in RS, execute (sometimes called "issue").
3. **Write result**—finish execution (WB)
   - Write on CDB to all awaiting FUs & reorder buffer; mark RS available.
4. **Commit**—update register with reorder result
   - When instruction at head of reorder buffer & result are present and the branch before it has been confirmed, update register with result (or store to memory) and remove instruction from ROB. Mispredicted branch flushes ROB above (executed after) the branch (sometimes called "graduation").

Tomasulo With Reorder buffer:
Tomasulo With Reorder buffer:

FP adders
FP multipliers
Reservation Stations
FP Op Queue

ROB

Dest. Value Instruction Done?

M[10] ST 0(R3), F4 Y ROB7
F0 ADDD F0, F4, F6 N ROB6
F0 DIVD F2, F10, F6 N ROB8
F0 ADDD F0, F4, F6 N ROB2
F0 ST 0(R3), F4 Y ROB5
F0 M[10] ST 0(R3), F4 Y ROB7
F0 ADDD F0, F4, F6 N ROB6
F0 DIVD F2, F10, F6 N ROB8
F0 ADDD F0, F4, F6 N ROB2
F0 ST 0(R3), F4 Y ROB5
F0 M[10] ST 0(R3), F4 Y ROB7
F0 ADDD F0, F4, F6 N ROB6
F0 DIVD F2, F10, F6 N ROB8
F0 ADDD F0, F4, F6 N ROB2
F0 ST 0(R3), F4 Y ROB5
F0 M[10] ST 0(R3), F4 Y ROB7
F0 ADDD F0, F4, F6 N ROB6
F0 DIVD F2, F10, F6 N ROB8
F0 ADDD F0, F4, F6 N ROB2
F0 ST 0(R3), F4 Y ROB5
F0 M[10] ST 0(R3), F4 Y ROB7

What about memory hazards???
**Avoiding Memory Hazards**

- **WAW and WAR hazards** through memory are eliminated with speculation because actual updating of memory occurs in order, when a store is at head of the ROB, and hence, no earlier loads or stores can still be pending.
- **RAW hazards** through memory are avoided by two restrictions:
  - not allowing a load to initiate the second step (actual memory read) of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the Addr. field of the load, and
  - maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.
- These restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data.

**Exceptions and Interrupts**

- IBM 360/91 invented "imprecise interrupts"
  - "Computer stopped at this PC; error likely near this address"
  - Not so popular with programmers
  - Also, what about Virtual Memory? (Not in IBM 360)
- Technique for both precise interrupts/exceptions and speculation:
  - in-order completion and in-order commit
    - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
    - Exactly the same must be done for precise exceptions
- Exceptions are handled by not recognizing the exception until instruction that caused it is ready to commit in ROB
  - If a speculated instruction raises an exception, the exception is recorded in the ROB
  - There are reorder buffers in all new processors to provide precise interrupts, which help programmers find errors faster.

**Multi-Issue - 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, and
  - VLIW (very long instruction word) processors (static sched.)
- 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 one long instruction word are independent => can execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field for each FU => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling techniques to schedule across several branches (called "trace scheduling")
Recall: Unrolled Loop that Eliminates Stalls for Scalar Pipeline Computers

1 Loop: L.D F0,0(R1) Minimum cycles between pairs of instructions:
2 L.D F6,-8(R1) L.D to ADD.D: 1 Cycle
3 L.D F10,-16(R1) ADD.D to S.D: 2 Cycles
4 ADD.D F4,F0,F2
5 ADD.D F8,F6,F2
6 ADD.D F12,F10,F2 A single branch delay slot follows the BNEZ.
7 S.D 0(R1),F4
8 S.D -6(R1),F8
9 DSUBUI R1,R1,#24
10 BNEZ R1,LOOP
11 S.D 8(R1),F12 ; 8-24 = -16

11 clock cycles, or 3.67 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>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</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 F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td>L.D F30,-56(R1)</td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>S.D 0(R1),F4</td>
<td>S.D -8(R1),F8</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>S.D -16(R1),F12</td>
<td>S.D -24(R1),F16</td>
<td>DSUBUI R1,R1,#56</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>S.D -32(R1),F20</td>
<td>S.D -40(R1),F24</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>S.D -48(R1),F28</td>
<td>S.D -56(R1),F32</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>BNEZ R1,LOOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

- Unrolled 7 times to avoid stall delays from ADD.D to S.D
- 7 results in 9 clocks, or 1.3 clocks per iteration (2.8X: 1.3 vs. 3.67)
- Average: 2.5 ops per clock, 51% efficiency in filling slots (23 ops in 45 slots)

Note: 8, not -48, after DSUBUI R1,R1,#56 - which may be out of place. See next slide.

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
- 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 predict function unit stalls, but cache stalls are hard to predict
- Binary code incompatibility
- 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 – 64 bits per integer
- 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 of some instructions avoids many branches
  (select 1 out of 64 1-bit flags) “if fl2, add op” => 40% fewer mispredictions?
- 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

Outline

- ILP
- Speculation
- Speculative Tomasulo Example
- Memory Aliases
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction

Increasing Instruction Fetch Bandwidth

- Predicts next instruction address, sends it out before decoding instruction
- PC of branch sent to BTB
- When match is found, Predicted PC is returned
- If a branch is predicted taken, instruction fetch continues at Predicted PC

Branch Target Buffer (BTB)

- 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
IF BW: Integrated IF Unit

- **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.

Issues on Implementing 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 architecturally visible register until the instruction commits.
  - Most Out-of-Order processors today use extended registers with renaming.

Value Prediction – just so-so (not used)

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

  With branch:  
  ```c
  BNEZ R1, L
  ADDU R2, R3, R0
  ```
  
  With conditional move (if 3rd operand equals zero):  
  ```c
  CMOVZ R2, R3, R1
  ```

- Convert 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 with 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 with 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

<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 R9,0(R10)</td>
<td></td>
</tr>
</tbody>
</table>

LWC loads if 3rd operand not 0

Putting it All Together: The Intel Pentium 4

(Mis) Speculation on Pentium 4

- % of micro-ops not used (Instr. becomes micro-ops as fetched)

<table>
<thead>
<tr>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>43%</td>
<td>45%</td>
</tr>
<tr>
<td>38%</td>
<td>39%</td>
</tr>
<tr>
<td>24%</td>
<td>24%</td>
</tr>
<tr>
<td>15%</td>
<td>16%</td>
</tr>
<tr>
<td>10%</td>
<td>11%</td>
</tr>
<tr>
<td>5%</td>
<td>6%</td>
</tr>
</tbody>
</table>

Perspective

- Early interest in multiple-issue because wanted to improve performance without affecting uniprocessor programming model
- Taking advantage of ILP is conceptually simple, but design problems are amazingly complex in practice

- Conservative in ideas, just faster clock and bigger chip
- Processors announced in 2005 (Pentium 4, IBM Power 5, AMD Opteron) have the same basic structure and similar sustained issue rates (3 to 4 instructions per clock) as the first dynamically scheduled, multiple-issue processors announced in 1995
  - Clocks 10 to 20X faster, caches 4 to 8X bigger, 2 to 4X as many renaming registers, and 2X as many load-store units => performance 8 to 16X
- Peak vs. delivered performance gap increasing
In Conclusion …

- Interrupts and exceptions either interrupt the current instruction or happen between instructions
  - Possibly large quantities of state must be saved before interrupting
- Machines with precise exceptions provide one single point in the program as PC to restart execution
  - All instructions before that point have completed
  - No instructions after or including that point have completed
- Hardware techniques exist for precise exceptions even in the face of out-of-order execution!
  - Important enabling factor for out-of-order execution