CMSC 411
Computer Systems Architecture
Lecture 11
Instruction Level Parallelism (cont.)

Alan Sussman
als@cs.umd.edu

Administrivia

- Wanli will give lecture on Thursday
- Exam #1 – answers posted
  - Mean: 67  Median: 66  Standard Dev.: 12.5
- Homework #3 posted, from H&P Chapter 2
  - due March 24
- Read Chapter 3 of H&P
  - but not too deeply – there’s way too much detail in the experiments/comparisons

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 RS
  - Tag results with ROB buffer number instead of reservation station
- Instructions commit ⇒ values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions
Recall: 4 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 (this stage sometimes called "dispatch")

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; checks RAW (sometimes called "issue")

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.

Mispredicted branch flushes reorder buffer (sometimes called "graduation")
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 maintained by two restrictions:
  1. not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and
  2. 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; its likely close to 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
  - This is exactly same as need to do with 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
  - This is why reorder buffers in all new processors
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:
  1. statically-scheduled superscalar processors,
  2. dynamically-scheduled superscalar processors, and
  3. VLIW (very long instruction word) processors
- 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: L.D F0,0(R1)                                L.D to ADD.D: 1 Cycle 
2   L.D F6,-8(R1)                                   ADD.D to S.D: 2 Cycles
3   L.D F10,-16(R1)                                 
4   L.D F14,-24(R1)                                 
5   ADD.D F4,F0,F2                                 
6   ADD.D F8,F6,F2                                 
7   ADD.D F12,F10,F2                               
8   ADD.D F16,F14,F2                               
9   S.D 0(R1),F4                                  
10  S.D -8(R1),F8                                 
11  DSUBUI R1,R1,#32                              
12  S.D -16(R1),F12                               
13  BNEZ R1,LOOP                                  
14  S.D 8(R1),F16                                 
    ; 8–32 = -24
```

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>S.D 0(R1),F4</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 -16(R1),F12</td>
<td>S.D -24(R1),F16</td>
<td>DSUBUI R1,R1,#48</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>S.D 16(R1),F20</td>
<td>S.D 8(R1),F24</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>S.D -0(R1),F2B</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

- **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
  - 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
- 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 => binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags) => 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

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

- 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 cond. 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