Multiple Issue Processors

Issuing Multiple Instructions Per Cycle

- Why?
  - All of the schemes described so far can at best achieve 1 instruction/cycle
  - Increasing transistor budgets, parallelism in instruction streams (independent instructions) have pushed for multiple instructions/cycle

Two variations
- **Superscalar**: varying number of instructions/cycle (1 to 8),
  - static (requires compiler support for most benefit), or dynamic
  - with or without speculation (implies hardware scheduling)
  - IBM Power2, Sun UltraSPARC, Intel Pentium III/4, DEC Alpha, HP 8000
- **(Very) Long Instruction Words (V)LIW**: fixed set of instructions (4-16)
  - scheduled by the compiler: put ops into wide templates
  - i860, IA64 (with some hardware support)

- New metric of performance: Instructions Per Clock cycle (IPC) vs. CPI

Statically Scheduled Superscalar MIPS Processor

- **Superscalar MIPS**: 2 instructions; 1 FP op, 1 other
  - Instruction issue
    - Fetch 64-bits/clock cycle
    - Need to handle cache-line complications
    - Hardware determines whether 0, 1, or 2 instructions can be issued
  - Can only issue 2nd instruction if 1st instruction issued

- **Hazard detection**
  - Likelihood of hazards between two instructions in a packet
    - Simple solution: treat this as a structural hazard (issue only 1 of them)
  - 1-instruction load delay can expand to 3-instruction delay in 2-way SS
  - 2nd instruction in the pair can’t use it without stall, our 2 instructions in next slot
  - Branch delay becomes 2 cycles (2nd inst. in branch) or 3 cycles (1st inst. in branch)

- **Execution**
  - Additional (or pipelined) functional units to derive benefit
  - Additional port for FP registers to do FP load or FP store and FP op
  - More bypass paths

2-way Static Superscalar MIPS Pipeline

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Pipe Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Integer</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Integer</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Pipeline Scheduling and Loop Unrolling

- Rare to find the ideal instruction mix of the previous slide
- Modern-day compilers apply several optimizations so as to expose Instruction Level Parallelism (ILP)

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s
```

```
L1: L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
DADDUI R1, R1, #-8
BNE R1, R2, L1
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>L.D F0, 0(R1)</td>
</tr>
<tr>
<td></td>
<td>stall 2</td>
</tr>
<tr>
<td></td>
<td>ADD.D F4, F0, F2</td>
</tr>
<tr>
<td></td>
<td>S.D F4, 0(R1)</td>
</tr>
<tr>
<td></td>
<td>DADDUI R1, R1, #-8</td>
</tr>
<tr>
<td></td>
<td>BNE R1, R2, L1</td>
</tr>
</tbody>
</table>

Loop Unrolling for Superscalar Processors

- Unroll loop 5 times to avoid extra 1-cycle delays in 2-way SS

```
L1: L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
DADDUI R1, R1, #-8
BNE R1, R2, L1
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>FP Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>L.D F0, 0(R1)</td>
</tr>
<tr>
<td></td>
<td>ADD.D F4, F0, F2</td>
</tr>
<tr>
<td></td>
<td>S.D F4, 0(R1)</td>
</tr>
<tr>
<td></td>
<td>DADDUI R1, R1, #-8</td>
</tr>
<tr>
<td></td>
<td>BNE R1, R2, L1</td>
</tr>
</tbody>
</table>

Limits of Superscalar Processors

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- Need more instructions to issue at the same time to get improved performance
  - However, greater difficulty of decode and issue
  - From 2-way superscalar => examine 2 opcodes, 6 register specifiers, and decide if 1 or 2 instructions can issue
- Issue rates of modern processors vary between 2—8 instructions/cycle
- Motivation for VLIW and EPIC processors …

Dynamic Scheduling in Superscalar Processors

- How to extend Tomasulo’s algorithm?
- General solution:
  - Allow issue stage to work faster than rest of architecture
    - Achieved by both pipelining and widening the issue logic
    - Instructions issued, reservation stations allocated in order
  - Rest of the design already supports overlapped execution
  - Need wider CDB to store multiple results/cycle
    - Need to allow multiple instruction commits per clock cycle

VLIW/EPIC Architectures

- Very Long Instruction Word (VLIW)
  - processor can initiate multiple operations per cycle
  - Specified completely by the compiler (unlike superscalar machines)
  - Hardware is simpler; issues the packet given it by the compiler
- Explicitly Parallel Instruction Computing (EPIC)
  - VLIW + new features
    - speculations, rotating registers, speculations, etc.
- More later about compiling for VLIW/EPIC
Studies of the Limitations of ILP

- Is there that much parallelism in programs?
- Start off with a hardware model of an ideal processor
  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
  2 and 3 => no control dependencies == machine with perfect speculation == an unbounded buffer of instructions available for execution
  4. Memory-address alias analysis – addresses are known and a load can be moved before a store provided addresses not equal
  1 and 4 => only true data dependencies
  - 1 cycle latency for all instructions
  - Perfect caches (loads and stores complete in one cycle)

ILP Limit for Six SPEC92 Benchmarks

- Fair bit of instruction-level parallelism, but how much of this stems from the ideal nature of assumptions
  - Infinite registers, perfect jump/branch prediction, perfect alias analysis

More Realistic Hardware: Branch Impact

- The set of instructions that is examined for simultaneous executions is called the window
  - Limiting factor: operand checking
  - Scales as # of inst. completing/cycle * window size * # operands/inst.
  - FP programs have loop level parallelism which benefit from large window

More Realistic HW: Register Impact

- More aggressive optimizations
  - Address value prediction and speculation (for memory accesses)
  - Can help achieve results similar to near-perfect alias analysis
  - Speculating on multiple paths
  - Reduces recovery costs (hopefully some path is useful), and exposes more ILP
- Even perfect model has some limitations
  - WAR and WAW hazards through memory
  - Can arise due to reuse of stack locations
  - Unnecessary dependences (i.e., compilers can do better than assumed)
  - E.g., dependence on loop control variable can be eliminated by loop unrolling
  - Overcoming the data flow limit
  - Recent idea: Value Prediction
  - Speculate that a register will have a certain value, and then recover if this speculation turns out to be false
  - Can speculate both data values and address values (for alias elimination)
A Different Perspective

- So far: Parallelism among instructions in a single thread of control
- What if we interleave instructions from multiple threads of control?
  - These instructions are independent (modulo thread synchronization)
    - Different register sets per thread
  - Overall program finishes earlier
    - Note that behavior of a single thread has not been improved
- Multithreading (later)

Compiling for VLIW/EPIC Processors

Next Week