You Are Here!

Parallel Requests
Assigned to computer
e.g., Search “Katz”

Parallel Threads
Assigned to core
e.g., Lookup, Ads

Parallel Instructions
>1 instruction @ one time
e.g., 5 pipelined instructions

Parallel Data
>1 data item @ one time
e.g., Add of 4 pairs of words

Hardware descriptions
All gates @ one time

• Programming Languages

Review

The BIG Picture

• Pipelining improves performance by increasing instruction throughput: exploits ILP
  – Executes multiple instructions in parallel
  – Each instruction has the same latency
• Subject to hazards
  – Structure, data, control
• Stalls reduce performance
  – But are required to get correct results
• Compiler can arrange code to avoid hazards and stalls
  – Requires knowledge of the pipeline structure

Agenda

• Higher Level ILP
• Administrivia
• Dynamic Scheduling
• And, in Conclusion, ...

Greater Instruction-Level Parallelism (ILP)

1. Deeper pipeline (5 => 10 => 15 stages)
   – Less work per stage ⇒ shorter clock cycle
2. Multiple issue superscalar
   – Replicate pipeline stages ⇒ multiple pipelines
   – Start multiple instructions per clock cycle
• CPI < 1, so can use Instructions Per Cycle (IPC)
  – E.g., 4 GHz 4-way multiple-issue
    • 16 BIPS, peak CPI = 0.25, peak IPC = 4
    – But dependencies reduce this in practice
Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into "issue slots"
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Pipeline Depth and Issue Width

- Intel Processors over Time

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>25W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>3</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td>3</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core 2 Conroe</td>
<td>2006</td>
<td>2930 MHz</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>96W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>

Static Multiple Issue

- Compiler groups instructions into issue packets
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - Called VLIW for Very Long Instruction Word

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
    - Pad with nop if necessary

Superscalar Laundry: Parallel per stage
MIPS with Static Dual Issue

- Dual-issue packets
  - One ALU/branch instruction + One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF ID EX ME WB</td>
</tr>
<tr>
<td>n+1</td>
<td>Load/store</td>
<td>IF ID EX ME WB</td>
</tr>
<tr>
<td>n+2</td>
<td>ALU/branch</td>
<td>IF ID EX ME WB</td>
</tr>
<tr>
<td>n+3</td>
<td>Load/store</td>
<td>IF ID EX ME WB</td>
</tr>
</tbody>
</table>

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result for load/store in same packet
    - add $t0, $s0, $s1
      load $s2, 0($t0)
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
  - More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop:  lw $t0, 0($s1)      # $t0=array element
      addu $t0, $t0, $s2    # add scalar in $s2
      sw $t0, 0($s1)      # store result
      addi $s1, $s1,–4      # decrement pointer
      bne $s1, $zero, Loop # branch $s1!=0
```

ALU/Branch Load/Store Cycle

- IPC = 5/4 = 1.25 (vs. peak IPC = 2)

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called register renaming
  - Avoid loop-carried anti-dependencies
    - Store followed by a load of the same register
    - Aka “name dependence”
    - Reuse of a register name but no real dependency between instructions

Loop Unrolling Example

```
Loop:  add $s1, $s1,–16
        lw $t0, 0($s1)      # $t0-array element
        add $t0, $t0, $s2    # add scalar in $s2
        sw $t0, 12($s1)     # store result
        addi $s1, $s1,–4      # decrement pointer
        bne $s1, $zero, Loop # branch $s1!=0
```

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of more registers and bigger code
**Agenda**

- Higher Level ILP
- Administrivia
- Dynamic Scheduling
- And, in Conclusion, ...

---

**Administrivia**

---

**Agenda**

- Higher Level ILP
- Administrivia
- Dynamic Scheduling
- And, in Conclusion, ...

---

**Dynamic Multiple Issue**

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, ... instructions each cycle
  - Avoiding structural and data hazards
- Avoids need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

---

**Dynamic Pipeline Scheduling**

- Allow the CPU to execute instructions *out of order* to avoid stalls
  - But commit result to registers in order
- Example
  
  ```
  lw  $t0, 20($a2)
  addu  $t1, $t0, $t2
  subu  $s4, $s4, $t3
  slti  $t5, $s4, 20
  ```
  - Can start subu while addu is waiting for lw
- Especially if cache misses, can execute many instructions

---

**Why Do Dynamic Scheduling?**

- Why not just let the compiler schedule code?
- Not all stalls are predictable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards
Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Examples
  - Speculate on branch outcome (Branch Prediction)
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated
- Can be done in hardware or by compiler
- Common to static and dynamic multiple issue

PipeLine Hazard: Matching socks in later load

A depends on D; stall since folder tied up;

Out-of-Order Laundry: Don’t Wait

• A depends on D; rest continue; need more resources to allow out-of-order

Out-of-Order Execution (1/2)

• Basically, unroll loops in hardware
  1. Fetch instructions in program order (≤4/clock)
  2. Predict branches as taken/untaken
  3. To avoid hazards on registers, rename registers using a set of internal registers (~80 registers)
  4. Collection of renamed instructions might execute in a window (~60 instructions)
  5. Execute instructions with ready operands in 1 of multiple functional units (ALUs, FPUs, Ld/St)

Out-of-Order Execution (2/2)

• Basically, unroll loops in hardware
  6. Buffer results of executed instructions until predicted branches are resolved in reorder buffer
  7. If predicted branch correctly, commit results in program order
  8. If predicted branch incorrectly, discard all dependent results and start with correct PC
Dynamically Scheduled CPU

Out Of Order Intel
• All use OOO since 2001

“And in Conclusion, ...”
• Big Ideas of Instruction Level Parallelism
• Pipelining, Hazards, and Stalls
• Forwarding, Speculation to overcome Hazards
• Multiple issue to increase performance
  – IPC instead of CPI
• Dynamic Execution: Superscalar in-order issue, branch prediction, register renaming, out-of-order execution, in-order commit
  – “unroll loops in HW”, hide cache misses