CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Instruction Level Parallelism: Multiple Instruction Issue

Instructors:
Randy H. Katz
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/fa10

Review: Hazards

Situations that prevent starting the next logical instruction in the next clock cycle
1. Structural hazards
   – Required resource is busy (e.g., read/write reg file at same time)
2. Data hazard
   – Need to wait for previous instruction to complete its data read/write (e.g., previous instruction hasn’t written back a result needed by current instruction)
3. Control hazard
   – Deciding on control action depends on previous instruction (e.g., whether or not branch was taken)

Agenda

• Control Hazards
• Administrivia
• Higher Level ILP
• Dynamic Scheduling
• Technology Break
• Example AMD Barcelona
• Big Picture: Types of Parallelism
• Summary

3. Control Hazards

• Branch determines flow of control
  – Fetching next instruction depends on branch outcome
  – Pipeline can’t always fetch correct instruction
    • Still working on ID stage of branch
• BEQ, BNE in MIPS pipeline
• Simple solution Option 1: Stall on every branch until have new PC value
  – Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)
3. Control Hazard: Branching

- Optimization #1:
  - Insert special branch comparator in Stage 2
  - As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  - Side Note: means that branches are idle in Stages 3, 4 and 5

Corrected Datapath for BEQ/BNE?

3. Control Hazards

- Option 2: Predict outcome of a branch, fix up if guess wrong
  - Must cancel all instructions in pipeline that depended on a wrong guess
- Simplest hardware if we predict that all branches are NOT taken
  - Why?

One Clock Cycle Stall

3. Control Hazard: Branching

- Option #3: Redefine branches
  - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (the branch-delay slot)
- Delayed Branch means we always execute inst after branch
- This optimization is used with MIPS
3. Control Hazard: Branching

- Notes on Branch-Delay Slot
  - Worst-Case Scenario: put a no-op in the branch-delay slot
  - Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn’t affect the logic of program
    - Re-ordering instructions is common way to speed up programs
    - Compiler usually finds such an instruction 50% of time
    - Jumps also have a delay slot …

Example: Nondelayed vs. Delayed Branch

<table>
<thead>
<tr>
<th>Nondelayed Branch</th>
<th>Delayed Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>or $8, $9, $10</code></td>
<td><code>add $1, $2,$3</code></td>
</tr>
<tr>
<td><code>add $1, $2, $3</code></td>
<td><code>sub $4, $5, $6</code></td>
</tr>
<tr>
<td><code>sub $4, $5, $6</code></td>
<td><code>beq $1, $4, Exit</code></td>
</tr>
<tr>
<td><code>beq $1, $4, Exit</code></td>
<td><code>or $8, $9, $10</code></td>
</tr>
<tr>
<td><code>xor $10, $1, $11</code></td>
<td><code>xor $10, $1, $11</code></td>
</tr>
</tbody>
</table>

Exit:

Delayed Branch/Jump and MIPS ISA?

- Why does JAL put PC+8 in register 31?
  - JAL executes following instruction (PC+4) so should return to PC+8

Delayed Branch/Jump and MIPS ISA?

- Why does JAL put PC+8 in register 31?
  - JAL executes following instruction (PC+4) so should return to PC+8

Agenda

- Control Hazards
- Administrivia
- Higher Level ILP
- Dynamic Scheduling
- Technology Break
- Example AMD Barcelona
- Big Picture: Types of Parallelism
- Summary

Administrivia

- Project 4: 2 Stage Pipelined Cycle Processor, non-delayed branch, in Logisim
  - Part 1, Datapath, due 4/10, Part 2 due 4/17
  - Face-to-Face grading: Signup for timeslot last week
- Extra Credit: Fastest Version of Project 3
  - Due 4/24 23:59:59
- Final Review: 5/2, 5-8 PM, 2050 Valley LSB
- Final: Mon May 9 11AM-2PM (TBD)
Agenda

- Control Hazards
- Administrivia
- Higher Level ILP
- Dynamic Scheduling
- Technology Break
- Example AMD Barcelona
- Big Picture: Types of Parallelism
- Summary

Greater Instruction-Level Parallelism (ILP)

- Deeper pipeline (5 => 10 => 15 stages)
  - Less work per stage ⇒ shorter clock cycle
- Multiple issue superscalar
  - Replicate pipeline stages ⇒ multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so 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>Microprocessor</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>29W</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>2002</td>
<td>2600 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>2600 MHz</td>
<td>31</td>
<td>3</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2007</td>
<td>2930 MHz</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>75W</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>

Superscalar Laundry: Parallel per stage

<table>
<thead>
<tr>
<th>Time</th>
<th>(light clothing)</th>
<th>(dark clothing)</th>
<th>(very dirty clothing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 PM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 AM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- More resources, HW to match mix of parallel tasks?
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

Scheduling Static Multiple Issue

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

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

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue pipeline
  - Now can’t use ALU result in 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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n ALU/branch</td>
<td>F ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td>n + 4 ALU/branch</td>
<td>F ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td>n + 8 Load/store</td>
<td>F ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td>n + 12 ALU/branch</td>
<td>F ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td>n + 16 Load/store</td>
<td>F ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td>n + 20 ALU/branch</td>
<td>F ID EX MEM WB</td>
<td></td>
</tr>
</tbody>
</table>

  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

  Loop: nop
  lw  $t0, 0($s1)
  addi $s1, $s1,–4
  nop
  addu $t0, $t0, $s2
  nop
  bne  $s1, $zero, Loop
  sw  $s2, 4($s1)

  Loop # branch $s1!=0

  IPC = 5/4 = 1.25 (c.f. 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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>$s1, $s1,–4</td>
<td>2</td>
</tr>
<tr>
<td>addu</td>
<td>$t0, $t0, $s2</td>
<td>1</td>
</tr>
<tr>
<td>bne</td>
<td>$s1, $zero, Loop sw</td>
<td>$s0, 4($s1)</td>
</tr>
</tbody>
</table>
Loop Unrolling Example

<table>
<thead>
<tr>
<th>All</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>addi $s1, $s1, -16</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>lw $s10, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>addu $s10, $s0, $s2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>lw $s11, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>addu $s11, $s1, $s2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>lw $s12, 4($s1)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>addu $s12, $s2, $s2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>lw $s10, 16($s1)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>addu $s10, $s14, $s2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>lw $s11, 12($s1)</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>sw $s12, 8($s1)</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>addu $s11, $s1, $s2</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>sw $s10, 0($s1)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>bne $s1, $zero, Loop</td>
<td>8</td>
</tr>
</tbody>
</table>

• IPC = $14 / 8 = 1.75
  – Closer to 2, but at cost of registers and code size

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
  \[
  lw \hspace{1em} st0, 20(st2) \\
  addu \hspace{1em} st1, st0, st2 \\
  subu \hspace{1em} s4, s4, st3 \\
  slti \hspace{1em} st5, s4, 20
  \]
  – Can start subu while addu is waiting for lw

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
• Common to static and dynamic multiple issue
• Examples
  – Speculate on branch outcome (Branch Prediction)
    • Roll back if path taken is different
  – Speculate on load
    • Roll back if location is updated
Pipeline Hazard: Matching socks in later load

Out-of-Order Laundry: Don’t Wait

Out-of-Order Execution (1/2)

Out-of-Order Execution (2/2)

Dynamically Scheduled CPU

Out Of Order Intel

Microprocessor  | Year  | Clock Rate | Pipeline Stages | Issue width | Out-of-order Speculation | Cores | Power
---             |------ |------------ |-----------------|-------------|------------------------|-------|-------
AMD            | 1989  | 25MHz      | 5               | 1           | No                     | 1     | 5W    
Pentium        | 1993  | 66MHz      | 5               | 2           | No                     | 1     | 10W   
Pentium Pro    | 1997  | 200MHz     | 10              | 3           | Yes                    | 1     | 75W   
P4 Prescott    | 2001  | 300MHz     | 31              | 3           | Yes                    | 1     | 120W  
Core 2          | 2004  | 320MHz     | 31              | 3           | Yes                    | 1     | 175W  
Core 2 Brandon  | 2006  | 2930GHz    | 16              | 4           | Yes                    | 4     | 65W   
Core i7         | 2010  | 3400 MHz   | 16              | 4           | Yes                    | 6     | 150W  

* A depends on D; stall since folder tied up;

• A depends on D; stall since folder Zed up;

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

- Control Hazards
- Administrivia
- Higher Level ILP
- Dynamic Scheduling
  - Technology Break
- Example AMD Barcelona
- Big Picture: Types of Parallelism
- Summary

4/7/11

Spring 2011 - Lecture #21

AMD Opteron X4 Microarchitecture

- 72 physical registers

Does Multiple Issue Work?

The Big Picture

- Yes, but not as much as we’d like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Agenda

- Higher Level ILP
- Administrivia
- Dynamic Scheduling
- Example AMD Barcelona
- Technology Break
- Big Picture: Types of Parallelism
- Summary

4/7/11

Spring 2011 - Lecture #21

AMD Opteron X4 Pipeline Flow

- For integer operations
  - 12 stages (Floating Point is 17 stages)
  - Up to 106 RISC-ops in progress
- Intel Nehalem is 16 stages for integer operations, details not revealed, but likely similar to above+
  - Intel calls RISC operations “Micro operations” or “μops”
New-School Machine Structures

• 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 description
  All gates functioning in parallel at same time

Big Picture on Parallelism

Two types of parallelism in applications
1. Data-Level Parallelism (DLP): arises because there are many data items that can be operated on at the same time
2. Task-Level Parallelism (TLP): arises because tasks of work are created that can operate largely in parallel

Big Picture on Parallelism

Hardware can exploit app Data LP and Task LP in 4 ways:
1. Instruction-Level Parallelism: Hardware exploits application DLP using ideas like pipelining and speculative execution
2. SIMD architectures: exploit app DLP by applying a single instruction to a collection of data in parallel
3. Thread-Level Parallelism: exploits either app DLP or TLP in a tightly-coupled hardware model that allows for interaction among parallel threads
4. Request-Level Parallelism: exploits parallelism among largely decoupled tasks and is specified by the programmer of the operating system

“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