CS 61C:
Great Ideas in Computer Architecture

Instruction Level Parallelism:
Multiple Instruction Issue

Instructor:
Randy H. Katz

http://inst.eecs.Berkeley.edu/~cs61c/fa13

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 Three Kinds of Hazards

Situations that prevent starting the next logical instruction in the next clock cycle

1. Structural hazards
   – Required resource is busy (e.g., single memory)

2. Data hazard
   – Need to wait for previous instruction to complete its data read/write (e.g., read before write)

3. Control hazard
   – Deciding on control action depends on previous instruction (e.g., what to fetch after branch)

Review: Control Hazard/Branching

• 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

• Optimization is used in MIPS
Review: 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 ...

Note: Delayed Branch/Jump

• Why does MIPS JAL put PC+8 in register 31?
• *JAL executes following instruction (PC+4) so should return to PC+8*
Compiler Assist to Minimize Hazards: Code Scheduling

- Reorder code to avoid use of load result in the next instruction
- C code for \( A = B + E; \ C = B + F; \)

\[
\begin{align*}
\text{lw} & \quad $t1, 0($t0) \\
\text{lw} & \quad $t2, 4($t0) \\
\text{add} & \quad $t3, $t1, $t2 \\
\text{sw} & \quad $t3, 12($t0) \\
\text{lw} & \quad $t4, 8($t0) \\
\text{add} & \quad $t5, $t1, $t4 \\
\text{sw} & \quad $t5, 16($t0)
\end{align*}
\]

13 cycles

\[
\begin{align*}
\text{lw} & \quad $t1, 0($t0) \\
\text{lw} & \quad $t2, 4($t0) \\
\text{lw} & \quad $t4, 8($t0) \\
\text{add} & \quad $t3, $t1, $t2 \\
\text{sw} & \quad $t3, 12($t0) \\
\text{add} & \quad $t5, $t1, $t4 \\
\text{sw} & \quad $t5, 16($t0)
\end{align*}
\]

11 cycles

Agenda

- Higher Level ILP
- Static Instruction Scheduling
- Dynamic Instruction Scheduling
- Out of Order Execution
- Parallelism Big Picture
- And, in Conclusion, ...
Agenda

• Higher Level ILP
• Static Instruction Scheduling
• Dynamic Instruction Scheduling
• Out of Order Execution
• Parallelism Big Picture
• 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

Superscalar Laundry: Parallel per Stage

- **More resources, HW to match mix of parallel tasks?**
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>i486</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>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>95W</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>
Agenda

- Higher Level ILP
- Static Instruction Scheduling
- Dynamic Instruction Scheduling
- Out of Order Execution
- Parallelism Big Picture
- And, in Conclusion, ...

Static Multiple Issue

- Compiler groups instructions into *issue packets*
  - Group of instructions that can issue in a single cycle
  - Determined by pipeline resources required
- Think of 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

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 MEM WB</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Datapath with Static Dual Issue

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

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>nop</td>
<td>1</td>
</tr>
<tr>
<td>addi $s1, $s1,–4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t0, 4($s1)</td>
<td>4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $s1, $s1, -16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>lw $t3, 4($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addu $t2, $t2, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>5</td>
</tr>
<tr>
<td>addu $t3, $t3, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>6</td>
</tr>
<tr>
<td>nop</td>
<td>sw $t2, 8($s1)</td>
<td>7</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td>8</td>
</tr>
</tbody>
</table>

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

Administrivia

- As of today, made 1 pass over all Big Ideas in Computer Architecture
- Following lectures go into more depth on topics you’ve already seen while you work on projects
  - 1 Lecture on Memory + Caches
  - 2 Lectures on Dependability/Reliability/Redundancy
  - 1 Lecture on Virtual Memory + Virtual Machines
  - 1 on Programming Contest (Extra Credit Project 3)
  - 1 on Course Wrap-up and Review
**Administrivia**

- **Final Exam**
  - Friday, December 20, 8-11 AM
  - Room TBD
  - Will be assigned by course account login
  - Comprehensive, but concentrated on material since midterm examination
  - Closed book/note, open crib sheet as before
  - Special consideration students, please contact us

**Agenda**

- Higher Level ILP
- Static Instruction Scheduling
- **Dynamic Instruction Scheduling**
- Out of Order Execution
- Parallelism Big Picture
- 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  $t0, 20($s2)
  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 predicable
  — 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
Not a Simple Linear Pipeline

• Three major units operating in parallel
• One Instruction fetch and issue unit
  – Issues instructions in program order
• Many parallel functional (execution) units
  – Each functional unit has input buffers called Reservation Stations
  – Holds operands and records the operation
  – Can execute instructions out-of-program-order (OOO)
• One Commit unit
  – Gets results from functional unit and saves in buffers called Reorder Buffer
  – Stores results once branch resolved so OK to execute
  – Commits results in program order

Agenda

• Higher Level ILP
• Static Instruction Scheduling
• Dynamic Instruction Scheduling
• Out of Order Execution
• Parallelism Big Picture
• And, in Conclusion, ...
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, replace 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

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order/Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>4</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2006</td>
<td>2930MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</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>Yes</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>
Does Multiple Issue Work?

- 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
- Static Instruction Scheduling
- Dynamic Instruction Scheduling
- Out of Order Execution
- Parallelism Big Picture
- And, in Conclusion, ...
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
Thanks to pipelining, I have reduced the time it took me to wash my one shirt.

Longer pipelines are always a win (since less work per stage & a faster clock).

A)(orange) I is True and II is True
B)(green) I is False and II is True
C)(pink) I is True and II is False
D)(yellow) I is False and II is False

Throughput better, not latency!

“...longer pipelines do usually mean faster clock rate, but hazards can cause problems!”
Peer Question
State if following techniques are associated primarily with a software- or hardware-based approach to exploiting ILP (in some cases, the answer may be both): Superscalar, Out-of-Order execution, Speculation, Register Renaming

<table>
<thead>
<tr>
<th></th>
<th>Superscalar</th>
<th>Out of Order</th>
<th>Speculation</th>
<th>Register Renaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>Green</td>
<td>HW</td>
<td>HW</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>Pink</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>Both</td>
</tr>
<tr>
<td>Yellow</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>SW</td>
</tr>
</tbody>
</table>

Peer Instruction
Instr LP, SIMD, Thread LP, Request LP are examples of
- Parallelism *above* ($\land$) the Instruction Set Architecture
- Parallelism explicitly *at* (=) the level of the ISA
- Parallelism *below* ($\lor$) the level of the ISA

<table>
<thead>
<tr>
<th></th>
<th>Inst. LP</th>
<th>SIMD</th>
<th>Thr. LP</th>
<th>Req. LP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange</td>
<td>$\lor$</td>
<td>$=$</td>
<td>$=$</td>
<td>$\land$</td>
</tr>
<tr>
<td>Green</td>
<td>$=$</td>
<td>$=$</td>
<td>$\land$</td>
<td>$\land$</td>
</tr>
<tr>
<td>Pink</td>
<td>$\lor$</td>
<td>$=$</td>
<td>$\land$</td>
<td>$\land$</td>
</tr>
<tr>
<td>Yellow</td>
<td>$=$</td>
<td>$\land$</td>
<td>$\land$</td>
<td>$\land$</td>
</tr>
</tbody>
</table>
Peer Instruction

I. Thanks to pipelining, I have *reduced the time* it took me to wash my one shirt.

II. Longer pipelines are *always a win* (since less work per stage & a faster clock).

A) (orange) I is True and II is True
B) (green) I is False and II is True
C) (pink) I is True and II is False
D) (yellow) I is False and II is False

Peer Question

Not all instructions are active in every stage of the 5-stage pipeline. Ignoring the effects of hazards, which of the following is true?

1. Allowing jumps, branches, and ALU instructions to take fewer stages than the 5 required by the load instruction will increase pipeline performance for most programs.
2. You cannot make ALU instructions take fewer cycles because of the write back of the result, but branches and jumps can take fewer cycles, so there is some opportunity for improvement.
3. Instead of trying to make instructions take fewer cycles, we should explore making the pipeline longer, so that instructions take more cycles, but the cycles are shorter. This could improve performance.
4. The number of pipe stages per instruction affects throughput, not latency.

Orange: 1  
Green: 2  
Pink: 3  
Yellow: 4
“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