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

Instruction Level Parallelism:
Multiple Instruction Issue

Guest Lecturer: Justin Hsia

Agenda

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

Review: Hazards

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

1. Structural hazards
   - A required resource is busy (e.g., needed in multiple stages)

2. Data hazard
   - Data dependency between instructions.
   - Need to wait for previous instruction to complete its data read/write

3. Control hazard
   - Flow of execution depends on previous instruction

Review: Load / Branch Delay Slots

• Stall is equivalent to nop

   lw $t0, 0($t1)
   nop
   sub $t3,$t0,$t2
   and $t5,$t0,$t4
   or $t7,$t0,$t6

Agenda

• Control Hazards
• Higher Level ILP
• Administrivia
• 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 is **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

Superscalar Laundry: Parallel per stage

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>3400 MHz</td>
<td>16</td>
<td>4</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>

Pipe Depth and Issue Width

- 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 (VLIW)
  - Specifies multiple concurrent operations
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>Possible Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF ID EX MEM</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 in load/store in same packet
    - add $t0, $t0, $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
Loop: nop lw $t0, 0($s1) 1
      lw $t0, 0($s1) 2
      lw $t0, 0($s1) 3
      lw $t0, 0($s1) 4
      bne $s1, $zero, Loop 5

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

Loop Unrolling

- Replicate loop body to expose more parallelism
- 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

Loop Unrolling Example

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

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

Administrivia

- Project 2 Part 2 due Sunday.
  - Slides at end of July 12 lecture contain useful info.
- Lab 12 cancelled!
  - Replaced with free study session where you can catch up on labs/work on project 2.
  - The TAs will still be there.
- Project 3 will be posted late Sunday (7/31)
  - Two-stage pipelined CPU in Logisim

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

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 both 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

Out-of-Order Execution (1/3)

- Basic approach: 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)

Out-of-Order Execution (2/3)

- Basic approach: unroll loops in hardware

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)
6. Buffer results of executed instructions until predicted branches are resolved in *reorder buffer*

Out-of-Order Execution (3/3)

- Basic approach: unroll loops in hardware

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

- Branch prediction
- Register renaming
- Results sent to any waiting reservation stations
- Can supply operands for issued instructions
- Out-of-order Intel
  - All use O-O-O since 2001
  - Table of Microprocessors:
    - AMD Opteron X4
      - 72 physical registers
      - 16 architectural registers
      - Supports 1D, 2D, and 3D packing
  - Out-of-order Intel Pipeline Flow:
    - 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”

Out-Of-Order Intel

- All use O-O-O 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>15 W</td>
</tr>
<tr>
<td>Pentium 1</td>
<td>1993</td>
<td>600MHz</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>2000MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>20W</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>3000MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2005</td>
<td>2500MHz</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>2933 MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>4</td>
<td>85W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3465 MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>6</td>
<td>110W</td>
</tr>
</tbody>
</table>

Agenda

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

---

**New-School Machine Structures**

*It’s a bit more complicated!*

- **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 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:** arises because tasks of work are created that can operate largely in parallel

---

**Peer Instruction**

Instr LP, SIMD, Thread LP, Request LP are examples of

- Parallelism above the Instruction Set Architecture
- Parallelism explicitly at the level of the ISA
- Parallelism below the level of the ISA

<table>
<thead>
<tr>
<th>Instruction (LP)</th>
<th>SIMD (SIMD)</th>
<th>Thread (Thr)</th>
<th>Request (Req)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>=</td>
<td>=</td>
<td>△</td>
</tr>
<tr>
<td>White</td>
<td>U</td>
<td>U</td>
<td>△</td>
</tr>
<tr>
<td>Green</td>
<td>=</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Yellow</td>
<td>U</td>
<td>U</td>
<td>△</td>
</tr>
<tr>
<td>Purple</td>
<td>=</td>
<td>△</td>
<td>△</td>
</tr>
<tr>
<td>Blue</td>
<td>△</td>
<td>△</td>
<td>△</td>
</tr>
</tbody>
</table>
Instr LP, SIMD, Thread LP, Request LP are examples of
• Parallelism above the Instruction Set Architecture
• Parallelism explicitly at the level of the ISA
• Parallelism below the level of the ISA

Peer Answer

State if the 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

Peer Question

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

But wait... there’s more???

If we have time...

Review: C Memory Management

C has three pools of data memory:
• Static storage: global variable storage, basically permanent, entire program run
• The Stack: local variable storage, parameters, return address
• The Heap (dynamic storage): malloc() grabs space from here, free() returns it

Common (Dynamic) Memory Problems
• Using uninitialized values
• Accessing memory beyond your allocated region
• Improper use of free/realloc by messing with the pointer handle returned by malloc
• Memory leaks: mismatched malloc/free pairs

OS prevents accesses between stack and heap (via virtual memory)
Simplest Model

- Only one program running on the computer
  - Addresses in the program are exactly the physical memory addresses
- Extensions to the simple model:
  - What if less physical memory than full address space?
  - What if we want to run multiple programs at the same time?

Problem #1: Physical Memory Less Than the Full Address Space

- One architecture, many implementations, with possibly different amounts of memory
- Memory used to very expensive and physically bulky
- Where does the stack grow from then?

Problem #2: Multiple Programs Sharing the Machine’s Address Space

- How can we run multiple programs without accidentally stepping on same addresses?
- How can we protect programs from clobbering each other?

Idea: Level of Indirection to Create Illusion of Large Physical Memory

- Many architectures, many implementations, with possibly different amounts of memory
- Memory used to very expensive and physically bulky
- Where does the stack grow from then?

Idea: Level of Indirection to Create Illusion of Separate Address Spaces

- One architecture, many implementations, with possibly different amounts of memory
- Memory used to very expensive and physically bulky
- Where does the stack grow from then?

Extension to the Simple Model

- Multiple programs sharing the same address space
  - E.g., Operating system uses low end of address range shared with application
  - Multiple programs in shared (virtual) address space
    - Static management: fixed partitioning/allocation of space
    - Dynamic management: programs come and go, take different amounts of time to execute, use different amounts of memory
- How can we protect programs from clobbering each other?
- How can we allocate memory to applications on demand?
Static Division of Shared Address Space

- E.g., how to manage the carving up of the address space among OS and applications?
- Where does the OS end and the application begin?
- Dynamic management, with protection, would be better!

First Idea: Base + Bounds Registers for Location Independence

Location-independent programs
Programming and storage management easier
need for a base register

Protection
Independent programs should not affect each other inadvertently: need for a bound register

Historically, base + bounds registers were a very early idea in computer architecture

Simple Base and Bound Translation

Base and bounds registers are visible/accessible to programmer
Trap to OS if bounds violation detected (“seg fault”/“core dumped”)

Restriction on Base + Bounds Regs

Want only the Operating System to be able to change Base and Bound Registers
Processors need different execution modes
1. User mode: can use Base and Bound Registers, but cannot change them
2. Supervisor mode: can use and change Base and Bound Registers
   - Also need Mode Bit (0=User, 1=Supervisor) to determine processor mode
   - Also need way for program in User Mode to invoke operating system in Supervisor Mode, and vice versa