The Processor: Instruction-Level Parallelism

Computer Organization

Friday 23 October 15

Many slides adapted from:
Computer Organization and Design, Patterson & Hennessy
and from Prof. Mary Jane Irwin, PSU
Summary

• Previous Class
  – Reducing pipeline data and branch hazards
    • Forwarding
    • Delayed Branch & Branch Prediction
  – Exceptions and Interrupts

• Today:
  – Multiple-issue processors
Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel

- To increase ILP
  - Deeper pipeline
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages $\Rightarrow$ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16k MIPS, 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 in each cycle
  – Compiler can help by reordering instructions
  – CPU resolves hazards using advanced techniques at runtime
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
  ⇒ Very Long Instruction Word (VLIW)
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 \texttt{nop} if necessary
MIPS with Static Dual Issue

- **Two-issue packets**
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with \texttt{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>
MIPS 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 in load/store in same packet
    
    \[
    \begin{align*}
    \text{add} & \quad t0, \quad s0, \quad s1 \\
    \text{load} & \quad s2, \quad 0(t0)
    \end{align*}
    \]
  
  • 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></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>lw $t0, 0($s1)</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 (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
Loop Unrolling Example

Loop:  
\[
\begin{align*}
\text{addi } & $s1, $s1, -16 \\
\text{lw } & $t0, 0($s1) \\
\text{addu } & $t0, $t0, $s2 \\
\text{sw } & $t0, 0($s1) \\
\text{addi } & $s1, $s1, -4 \\
\text{bne } & $s1, $zero, \text{Loop} \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>addi $s1, $s1, -16</td>
<td>lw $t0, 0($s1)</td>
</tr>
<tr>
<td></td>
<td>nop</td>
<td>lw $t1, 12($s1)</td>
</tr>
<tr>
<td></td>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
</tr>
<tr>
<td></td>
<td>addu $t1, $t1, $s2</td>
<td>lw $t3, 4($s1)</td>
</tr>
<tr>
<td></td>
<td>addu $t2, $t2, $s2</td>
<td>sw $t0, 16($s1)</td>
</tr>
<tr>
<td></td>
<td>addu $t3, $t3, $s2</td>
<td>sw $t1, 12($s1)</td>
</tr>
<tr>
<td></td>
<td>nop</td>
<td>sw $t2, 8($s1)</td>
</tr>
<tr>
<td></td>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</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, … each cycle
  – Avoiding structural and data hazards

• Avoids the 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

  \[
  \begin{align*}
  \text{lw} & \quad $t_0, 20($s_2) \\
  \text{addu} & \quad $t_1, $t_0, $t_2 \\
  \text{sub} & \quad $s_4, $s_4, $t_3 \\
  \text{slti} & \quad $t_5, $s_4, 20
  \end{align*}
  \]

  - Can start \text{sub} \text{ while} \text{ addu is waiting for lw}
Dynamically Scheduled CPU

- Instruction fetch and decode unit
- Reservation station
- Reservation station
- ... Reservation station
- Reservation station
- Functional units
  - Integer
  - Integer
  - Floating point
  - Load-store
- Commit unit
  - In-order commit
  - Can supply operands for issued instructions
  - Out-of-order execute
  - Results also sent to any waiting reservation stations
  - Hold pending operands
  - In-order issue
  - Preserves dependencies
- Reorders buffer for register writes
Register Renaming

• Reservation stations and reorder buffer effectively provide register renaming

• On instruction issue to reservation station
  – If operand is available in register file or reorder buffer
    • Copied to reservation station
    • No longer required in the register; can be overwritten
  – If operand is not yet available
    • It will be provided to the reservation station by a function unit
    • Register update may not be required
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
    • Roll back if path taken is different
  – Speculate on load
    • Roll back if location is updated
Speculation

• Predict branch and continue issuing
  – Don’t commit until branch outcome determined

• Load speculation
  – Avoid load and cache miss delay
    • Predict the effective address
    • Predict loaded value
    • Load before completing outstanding stores
    • Bypass stored values to load unit
  – Don’t commit load until speculation cleared
Compiler/Hardware Speculation

• Compiler can reorder instructions
  – e.g., move load before branch
  – Can include “fix-up” instructions to recover from incorrect guess

• Hardware can look ahead for instructions to execute
  – Buffer results until it determines they are actually needed
  – Flush buffers on incorrect speculation
Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check

- Static speculation
  - Can add ISA support for deferring exceptions

- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)
Why Do Dynamic Scheduling?

Why not just let the compiler schedule code?

- Not all stalls are predicatable
  - 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
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
### Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

<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/Chip</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5 W</td>
</tr>
<tr>
<td>Intel Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10 W</td>
</tr>
<tr>
<td>Intel Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29 W</td>
</tr>
<tr>
<td>Intel Pentium 4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75 W</td>
</tr>
<tr>
<td>Intel Pentium 4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103 W</td>
</tr>
<tr>
<td>Intel Core</td>
<td>2006</td>
<td>2930 MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75 W</td>
</tr>
<tr>
<td>Intel Core i5 Nehalem</td>
<td>2010</td>
<td>3300 MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>4</td>
<td>87 W</td>
</tr>
<tr>
<td>Intel Core i5 Ivy Bridge</td>
<td>2012</td>
<td>3400 MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>8</td>
<td>77 W</td>
</tr>
</tbody>
</table>
The Opteron X4 Microarchitecture

- Instruction cache
- Instruction prefetch and decode
- RISC-operation queue
- Dispatch and register remaining
- Register file
- 72 physical registers

- Branch prediction
- Integer and floating-point operation queue
  - Integer ALU, Multiplier
  - Integer ALU
  - Integer ALU
  - Floating point Adder /SSE
  - Floating point Multiplier /SSE
  - Floating point Misc
- Load/Store queue
- Data cache
- Commit unit
The Opteron X4 Pipeline Flow

- For integer operations
  - FP is 5 stages long
  - Up to 106 RISC-ops in progress
- Bottlenecks
  - Complex instructions with long dependencies
  - Branch mispredictions
  - Memory access delays
## CISC vs RISC vs SS vs VLIW

<table>
<thead>
<tr>
<th></th>
<th>CISC</th>
<th>RISC</th>
<th>Superscalar</th>
<th>VLIW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instr size</strong></td>
<td>variable size</td>
<td>fixed size</td>
<td>fixed size</td>
<td>fixed size (but large)</td>
</tr>
<tr>
<td><strong>Instr format</strong></td>
<td>variable format</td>
<td>fixed format</td>
<td>fixed format</td>
<td>fixed format</td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td>few, some special</td>
<td>Many GP</td>
<td>GP and rename (RUU)</td>
<td>many, many GP</td>
</tr>
<tr>
<td></td>
<td>Limited # of ports</td>
<td>Limited # of ports</td>
<td>Many ports</td>
<td>Many ports</td>
</tr>
<tr>
<td><strong>Memory reference</strong></td>
<td>embedded in many instr's</td>
<td>load/store</td>
<td>load/store</td>
<td>load/store</td>
</tr>
<tr>
<td><strong>Key Issues</strong></td>
<td>decode complexity</td>
<td>data forwarding, hazards</td>
<td>hardware dependency resolution</td>
<td>(compiler) code scheduling</td>
</tr>
</tbody>
</table>
Fallacies

• Pipelining is easy (!)
  – The basic idea is easy
  – The devil is in the details
    • e.g., detecting data hazards

• Pipelining is independent of technology
  – So why haven’t we always done pipelining?
  – More transistors make more advanced techniques feasible
  – Pipeline-related ISA design needs to take into account technology trends
    • e.g., predicated instructions
Pitfalls

• Poor ISA design can make pipelining harder
  – e.g., complex instruction sets (VAX, IA-32)
    • Significant overhead to make pipelining work
    • IA-32 micro-op approach
  – e.g., complex addressing modes
    • Register update side effects, memory indirection
  – e.g., delayed branches
    • Advanced pipelines have long delay slots
Concluding Remarks

• ISA influences design of datapath and control
• Datapath and control influence design of ISA
• Pipelining improves instruction throughput using parallelism
  – More instructions completed per second
  – Latency for each instruction not reduced
• Hazards: structural, data, control
• Multiple issue and dynamic scheduling (ILP)
  – Dependencies limit achievable parallelism
  – Complexity leads to the power wall
Next Class

• Memory hierarchy
The Processor: Instruction-Level Parallelism

Computer Organization

Friday 23 October 15