CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Lecture 32: Pipeline Parallelism 3

Instructor: Dan Garcia
http://inst.eecs.Berkeley.edu/~cs61c/sp13
"Hadoop's a big deal," said [Berkeley EECS Alum] Cloudera CEO Mike Olson. "It's not just a Web thing. Companies across a wide range of vertical markets are generating big data and need to understand that data in a way they never did before."

JPMorgan Chase makes a case for the big data platform (and career track) of the future.

[JP Morgan] has 150 petabytes (with a "p") of data online, generated by trading operations, banking activities, credit card transactions, and some 3.5 billion logins each year.

“The good news is that Hadoop experts aren't born, they're trained.”
You Are Here!

**Software**

- **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

---

**Hardware**

- Warehouse Scale Computer

---

- Harness Parallelism & Achieve High Performance

---

- Smart Phone

---

- Computer
  - Core
  - Memory
  - Input/Output
  - Instruction Unit(s)
    - \[A_0 + B_0, A_1 + B_1, A_2 + B_2, A_3 + B_3\]
  - Functional Unit(s)
  - Main Memory
  - Logic Gates

---

Today's Lecture
P&H Figure 4.50
P&H 4.51 – Pipelined Control
Hazards

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

1. Structural hazards
   – Required resource is busy (e.g., roommate studying)

2. Data hazard
   – Need to wait for previous instruction to complete its data read/write (e.g., pair of socks in different loads)

3. Control hazard
   – Deciding on control action depends on previous instruction (e.g., how much detergent based on how clean prior load turns out)
Data Hazards

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction

- C code for \( A = B + E; \ C = B + F; \)

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

13 cycles

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

11 cycles
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)
Stall => 2 Bubbles/Clocks

Where do we do the compare for the branch?
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

Question: What’s an efficient way to implement the equality comparison?
One Clock Cycle Stall

Time (clock cycles)

Instr Order

Instr 1  beq
Instr 2  Instr 3  Instr 4

Branch comparator moved to Decode stage.
Control Hazards: Branching

• Option 2: *Predict* outcome of a branch, fix up if guess wrong
  – Must cancel all instructions in pipeline that depended on guess that was wrong
  – This is called “flushing” the pipeline

• Simplest hardware if we predict that all branches are NOT taken
  – Why?
Control Hazards: 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
Example: Nondelayed vs. Delayed Branch

**Nondelayed Branch**
- or $8, $9, $10
- add $1, $2, $3
- sub $4, $5, $6
- beq $1, $4, Exit
- xor $10, $1, $11

**Delayed Branch**
- add $1, $2, $3
- sub $4, $5, $6
- beq $1, $4, Exit
- or $8, $9, $10
- xor $10, $1, $11
Control Hazards: 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 ...
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., 4GHz 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>
Pipeline Depth and Issue Width

- Clock
- Power
- Pipeline Stages
- Issue width
- Cores

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 issue packet with 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 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</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF</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, $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>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Scheduling Example

• Schedule this for dual-issue MIPS

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

<table>
<thead>
<tr>
<th></th>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>nop</td>
<td>lw  $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Scheduling Example

- Schedule this for dual-issue MIPS

Loop: 

<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></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></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Scheduling Example

- Schedule this for dual-issue MIPS

<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></td>
<td>4</td>
</tr>
</tbody>
</table>
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></th>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>nop</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>addi $s1, $s1,-4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td></td>
<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

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loop:</strong></td>
<td></td>
<td></td>
</tr>
<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, $t4, $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 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

  \[
  \begin{align*}
  \text{lw} & \quad \text{$t0$, 20($s2$)} \\
  \text{addu} & \quad \text{$t1$, $t0$, $t2$} \\
  \text{subu} & \quad \text{$s4$, $s4$, $t3$} \\
  \text{slti} & \quad \text{$t5$, $s4$, 20}
  \end{align*}
  \]

  – Can start \text{subu} while \text{addu} is waiting for \text{lw}
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
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
• 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 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>1</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?

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
“And in Conclusion..”

- Pipelining is an important form of ILP
- Challenge is (are?) hazards
  - Forwarding helps w/many data hazards
  - Delayed branch helps with control hazard in 5 stage pipeline
  - Load delay slot / interlock necessary
- More aggressive performance:
  - Longer pipelines
  - Superscalar
  - Out-of-order execution
  - Speculation