Agenda

- Review
- Instruction Set Design and Pipelined Execution
- Control Hazards
- Administrivia
- Branch Prediction
- Higher Level ILP
- Summary

Review

The BIG Picture

- Pipelining improves performance by increasing instruction throughput: exploits ILP
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - x86: 1- to 17-byte instructions
    (x86 HW actually translates to internal RISC instructions!)
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step
- Memory operands only in Loads and Stores
  - Can calculate address 3rd stage, access memory 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle

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: This means that branches are idle in Stages 3, 4 and 5.

Control Hazards

- Option 2: Predict outcome of a branch, fixup if guess wrong
  - Must cancel all instructions in pipeline that depended on guess that was wrong
- Simplest hardware if predict all branches NOT taken

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 (called the branch-delay slot)
  - The term “Delayed Branch” means we always execute inst after branch
  - This optimization is used with MIPS

Control Hazard: Branching

- Notes on Branch-Delay Slot
- Worst Case Scenario: put a no-op in the branch-delay slot
- Better Case: find instruction preceding branch placed in the branch-delay slot without affecting flow 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>or $8, $9, $10</td>
<td>add $1, $2, $3</td>
</tr>
<tr>
<td>add $1, $2, $3</td>
<td>sub $4, $5, $6</td>
</tr>
<tr>
<td>sub $4, $5, $6</td>
<td>beq $1, $4, Exit</td>
</tr>
<tr>
<td>beq $1, $4, Exit</td>
<td>or $8, $9, $10</td>
</tr>
<tr>
<td>xor $10, $1, $11</td>
<td>xor $10, $1, $11</td>
</tr>
<tr>
<td>Exit:</td>
<td>Exit:</td>
</tr>
</tbody>
</table>
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

- Review
- Instruction Set Design and Pipelined Execution
- Control Hazards
- Administrivia
- Branch Prediction
- Higher Level ILP
- Summary

Administrivia

- Project 3: Thread Level Parallelism + Data Level Parallelism + Cache Optimization
  - Due Part 2 due Saturday 11/13
- Project 4: Single Cycle Processor in Logicsim
  - Due Part 2 due Saturday 11/27
  - Face-to-Face grading: Signup for time slot in last week
- Extra Credit: Fastest Version of Project 3
  - Due Monday 11/29 Midnight
- Final Review: TBD (Vote via Survey!)
  - "Please narrow what we need to study on review"
- Final: Mon Dec 13 8AM-11AM (TBD)

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?
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 with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
    - Pad 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

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoids stalls with single-issue
  - Now can’t use ALU result in load/store in same packet
  - Pad with nop if necessary

Scheduling Example

- Schedule this for dual-issue MIPS

Loop: lw $t0, 0($s1)      # $t0 = array element
      addi $s1, $s1, -4       # decrement pointer
      bne $s1, $zero, Loop    # branch $s1 != 0

Loop:

- 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>addi $t1$, $s1$, $-16$</td>
<td>lw $t0$, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>addu $t0$, $t0$, $s2$</td>
<td>lw $t1$, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t1$, $t1$, $s2$</td>
<td>lw $t2$, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>add $t1$, $t1$, $s2$</td>
<td>sw $t0$, 16($s1)</td>
<td>4</td>
</tr>
<tr>
<td>add $t2$, $t2$, $s2$</td>
<td>sw $t1$, 12($s1)</td>
<td>5</td>
</tr>
</tbody>
</table>
|...
| addi $t1$, $s1$, $-16$ | lw $t0$, 0($s1) | 18    |

- 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
  
  lw  $t0$, 20($s2)
  addu $t1$, $s0$, $t2$
  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 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

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

11/4/10
Out-of-Order Laundry: Don’t Wait

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

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

“And in Conclusion..”

- Pipeline challenge is 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