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 functioning in parallel at same time

Smart Phone
Warehouse Scale
Computer

Review: RISC Design Principles
• “A simpler core is a faster core”
• Reduction in the number and complexity of instructions in the ISA → simplifies pipelined implementation
• Common RISC strategies:
  – Fixed/instruction length, generally a single word;
  – Simplifies process of fetching instructions from memory
  – Simplified addressing modes;
  – Simplifies process of fetching operands from memory
  – Fewer and simpler instructions in the instruction set;
  – Simplified process of executing instructions
  – Simplified memory access: only load and store instructions access memory;
  – Let the compiler do it. Use a good compiler to break complex high-level language statements into a number of simple assembly language statements

Review: Single-Cycle Processor
• Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements: re-examine for pipelining
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic
    • Formulate logic equations
    • Design Circuits

Agenda
• Pipelined Execution
• Administrivia
• Pipelined Datapath
• Pipeline Hazards
• Technology Break
• Pipelining and Instruction Set Design
• Summary
Single Cycle Performance

- Assume time for actions are
  - 100ps for register read or write; 200ps for other events
- Clock rate is?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fetch Time</th>
<th>Register Read</th>
<th>ALU Operation</th>
<th>Memory Access</th>
<th>Register Write</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>100ps</td>
<td>100ps</td>
<td>200ps</td>
<td>100ps</td>
<td>600ps</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>100ps</td>
<td>100ps</td>
<td>200ps</td>
<td>100ps</td>
<td>800ps</td>
<td></td>
</tr>
<tr>
<td>R-format</td>
<td>100ps</td>
<td>100ps</td>
<td>200ps</td>
<td>100ps</td>
<td>500ps</td>
<td></td>
</tr>
</tbody>
</table>

- What can we do to improve clock rate?
- Will this improve performance as well?
Want increased clock rate to mean faster programs

Pipeline Analogy: Doing Laundry

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - “Folder” takes 30 minutes
  - “Stasher” takes 30 minutes to put clothes into drawers

Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons (1/2)

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Time to fill pipeline and time to drain it reduces speedup: 2.3X v. 4X in this example

Pipelining Lessons (2/2)

- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
**Agenda**

- Pipelined Execution
- Administrivia
- Pipelined Datapath
- Pipeline Hazards
- Technology Break
- Pipelining and Instruction Set Design
- Summary

**Adminitrivia**

- Project 4: Pipelined Cycle Processor in Logicsim
  - Due Part 1, datapath, due 4/10, Part 2 due 4/17
  - Face-to-Face grading: Signup for timeslot last week
- Extra Credit: Fastest Version of Project 3
  - Due 4/24 23:59:59
- Final Review: TBD
- Final: Mon May 9 11AM-2PM (TBD)

**Steps in Executing MIPS**

1) **IF**: Instruction Fetch, Increment PC
2) **ID**: Instruction Decode, Read Registers
3) **EX**: Mem-ref: Calculate Address Arith-log: Perform Operation
4) **Mem**: Load: Read Data from Memory Store: Write Data to Memory
5) **WB**: Write Data Back to Register

**Redrawn Single-Cycle Datapath**
Pipelined Datapath

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back

- Add registers between stages
  - Hold information produced in previous cycle

More Detailed Pipeline

IF for Load, Store, ...

ID for Load, Store, ...

EX for Load

MEM for Load
Agenda

• Pipelined Execution
• Administrivia
• Pipelined Datapath
  • Pipeline Hazards
  • Technology Break
  • Pipelining and Instruction Set Design
• Summary

Pipelined Execution Representation

Graphical Pipeline Diagrams

Graphical Pipeline Representation (In Reg, right half highlight read, left half write)
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op access</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>100ps</td>
<td>800ps</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100ps</td>
<td>200ps</td>
<td>100ps</td>
<td>600ps</td>
<td></td>
</tr>
</tbody>
</table>

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions_{pipelined} = Time between instructions_{nonpipelined} / Number of stages
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Instruction Level Parallelism (ILP)

- Another parallelism form to go with Request Level Parallelism and Data Level Parallelism
  - RLP – e.g., Warehouse Scale Computing
  - DLP – e.g., SIMD, Map-Reduce
- ILP – e.g., Pipelined Instruction Execution
  - 5 stage pipeline => 5 instructions executing simultaneously, one at each pipeline stage

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)

1. Structural Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/Store requires memory access for data
  - Instruction fetch would have to stall for that cycle
    - Causes a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - In reality, provide separate L1 I$ and L1 D$
1. Structural Hazard #1: Single Memory

Read same memory twice in same clock cycle

1. Structural Hazard #2: Registers (1/2)

Can we read and write to registers simultaneously?

1. Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) RegFile access is VERY fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle
  2) Build RegFile with independent read and write ports
     - Result: can perform Read and Write during same clock cycle

2. Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add $s0, $t0, $t1
  - sub $t2, $s0, $t3

Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath

Corrected Datapath for Forwarding?
Load-Use Data Hazard
• Can’t always avoid stalls by forwarding
  – If value not computed when needed
  – Can’t forward backward in time!

Agenda
• Pipelined Execution
• Administrivia
• Pipelined Datapath
• Pipeline Hazards
• Technology Break
• Pipelining and Instruction Set Design
• Summary

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

Why Isn’t the Destination Register Always in the Same Field in MIPS ISA?
• Need to have 2 part immediate if 2 sources and 1 destination always in same place

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)
3. Control Hazards

- 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
  - Simplest hardware if we predict that all branches are NOT taken
    - Why?

3. 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 (the branch-delay slot)
- Delayed Branch means we always execute inst after branch
- This optimization is used with MIPS
3. 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 ...

---

Example: Nondelayed vs. Delayed Branch

**Nondelayed Branch**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Destination</th>
<th>Source1</th>
<th>Source2</th>
</tr>
</thead>
<tbody>
<tr>
<td>or $8, $9, $10</td>
<td>add 1, 2, 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub 4, 5, 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq $1, $4, Exit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor $10, $1, $11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Delayed Branch**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Destination</th>
<th>Source1</th>
<th>Source2</th>
</tr>
</thead>
<tbody>
<tr>
<td>or $8, $9, $10</td>
<td>add 1, 2, 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub 4, 5, 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq $1, $4, Exit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor $10, $1, $11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Delayed Branch/Jump and MIPS ISA?

- Why does JAL put PC+8 in register 31?

---

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)  # $t1 <- 0
  lw $t2, 4($t0)  # $t2 <- 4
  lw $t4, 8($t0)  # $t4 <- 8
  add $t3, $t1, $t2  # A <- B+E
  sw $t3, 12($t0)  # $t3 <- 12
  add $t5, $t1, $t4  # C <- B+F
  sw $t5, 16($t0)  # $t5 <- 16
  ```

  - 11 cycles
  - 13 cycles

---

And in Conclusion, ...

**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