Initially, when the iPhone 4S went on sale Oct. 14, many users couldn’t get Siri to work because so many people were trying at the same time. Siri needs to go through Apple’s cloud-based servers to work, and Siri’s popularity caused a bit of a traffic jam.

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

Pipeilined Execution

```
<table>
<thead>
<tr>
<th>Time</th>
<th>IFetch</th>
<th>Dst</th>
<th>Exec</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IFetch</td>
<td>Dst</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IFetch</td>
<td>Dst</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IFetch</td>
<td>Dst</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IFetch</td>
<td>Dst</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>
```

- Every instruction must take same number of steps, also called pipeline “stages”, so some will go idle sometimes
Graphical Pipeline Representation
(In Reg, right half highlight read, left half write)

Time (clock cycles)

Instr.
Load
Add
Store
Sub
Or

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</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>500ps</td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

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

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 \textit{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

Data Hazards (1/2)

- Consider the following sequence of instructions:
  - add $t0, $t1, $t2
  - sub $t4, $t0, $t3
  - and $t5, $t0, $t6
  - or $t7, $t0, $t8
  - xor $t9, $t0, $t10

Data Hazards (2/2)

- Data-flow backward in time are hazards

Data Hazard Solution: Forwarding

- Forward result from one stage to another
  - add $t0, $t1, $t2
  - sub $t4, $t0, $t3
  - and $t5, $t0, $t6
  - or $t7, $t0, $t8
  - xor $t9, $t0, $t10

“oz” hazard solved by register hardware
Data Hazard: Load/Use (1/4)

- Dataflow backwards in time are hazards
  - lw $t0, 0($t1)
  - sub $t3, $t0, $t2

- Can’t solve all cases with forwarding
- Must stall instruction dependent on load, then forward (more hardware)

Data Hazard: Load/Use (2/4)

Hardware stalls pipeline (Called “interlock”)

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

Not in MIPS: (MIPS = Microprocessor without Interlocked Pipeline Stages)

Data Hazard: Load/Use (3/4)

- Instruction slot after a load is called “load delay slot”
  - If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
  - Alternative: If the compiler puts an unrelated instruction in that slot, then no stall
  - Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)

Data Hazard: Load/Use (4/4)

- Stall is equivalent to nop
  - lw $t0, 0($t1)
  - nop
  - sub $t3, $t0, $t2
  - and $t5, $t0, $t4
  - or $t7, $t0, $t6

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

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?

Until next time ... 

**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