CS352H: Computer Systems Architecture

Topic 8: MIPS Pipelined Implementation

September 29, 2009
MIPS Pipeline

- Five stages, one step per stage
  - IF: Instruction fetch from memory
  - ID: Instruction decode & register read
  - EX: Execute operation or calculate address
  - MEM: Access memory operand
  - WB: Write result back to register
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- 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></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
<tr>
<td>j</td>
<td>200ps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>200ps</td>
</tr>
</tbody>
</table>
Pipeline Performance

**Single-cycle ($T_c = 800\text{ps}$)**

1. **Iw $1, 100(0)$**
   - Instruction fetch
   - Reg
   - ALU
   - Data access
   - Reg
   - $800 \text{ ps}$

2. **Iw $2, 200(0)$**
   - Instruction fetch
   - Reg
   - ALU
   - Data access
   - Reg

3. **Iw $3, 300(0)$**
   - Instruction fetch
   - Reg
   - ALU
   - Data access
   - Reg
   - $800 \text{ ps}$

**Pipelined ($T_c = 200\text{ps}$)**

1. **Iw $1, 100(0)$**
   - Instruction fetch
   - Reg
   - ALU
   - Data access
   - Reg
   - $200 \text{ ps}$

2. **Iw $2, 200(0)$**
   - Instruction fetch
   - Reg
   - ALU
   - Data access
   - Reg
   - $200 \text{ ps}$

3. **Iw $3, 300(0)$**
   - Instruction fetch
   - Reg
   - ALU
   - Data access
   - Reg
   - $200 \text{ ps}$
Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions_{pipelined} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}

- If not balanced, speedup is less

- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease
MIPS ISA designed for pipelining

- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - c.f. x86: 1- to 17-byte instructions
- Few and regular instruction formats
  - Can decode and read registers in one step
- Load/store addressing
  - Can calculate address in 3\textsuperscript{rd} stage, access memory in 4\textsuperscript{th} stage
- Alignment of memory operands
  - Memory access takes only one cycle
Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction
Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
Data Hazards

- An instruction depends on completion of data access by a previous instruction

  - **add** $s0$, $t0$, $t1$
  - **sub** $t2$, $s0$, $t3$

![Diagram of instruction execution phases and data hazards](image)
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

![Diagram showing program execution order and time](image)
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!
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 $t1, 0($t0)
lw $t2, 4($t0)  lw $t2, 4($t0)
add $t3, $t1, $t2  add $t3, $t1, $t2
lw $t4, 8($t0)    lw $t4, 8($t0)
sw $t3, 12($t0)   sw $t3, 12($t0)
add $t5, $t1, $t4  add $t5, $t1, $t4
sw $t5, 16($t0)   sw $t5, 16($t0)
```

11 cycles 13 cycles
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

- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage
Stall on Branch

- Wait until branch outcome determined before fetching next instruction

Diagram:

- Program execution order (in instructions)
  - add $4, $5, $6
  - beq $1, $2, 40
  - or $7, $8, $9

- Time: 200 400 600 800 1000 1200 1400

- Instruction fetch
- Reg
- ALU
- Data access
- Reg

- Time: 200 ps for beq $1, $2, 40
- Time: 400 ps for add $4, $5, $6 or or $7, $8, $9
Branch Prediction

- Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay
MIPS with Predict Not Taken

Prediction correct

Program execution order (in instructions)
- add $4, $5, $6
- beq $1, $2, 40
- lw $3, 300($0)

Time
- 200 ps
- 200 ps

Prediction incorrect

Program execution order (in instructions)
- add $4, $5, $6
- beq $1, $2, 40
- or $7, $8, $9

Time
- 200 ps
- 400 ps
More-Realistic Branch Prediction

- **Static branch prediction**
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

- **Dynamic branch prediction**
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation
MIPS Pipelined Datapath

Right-to-left flow leads to hazards
Pipeline registers

- Need registers between stages
- To hold information produced in previous cycle
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams for load & store
IF for Load, Store, ...
ID for Load, Store, …
EX for Load
MEM for Load
WB for Load

Wrong register number
Corrected Datapath for Load
EX for Store
MEM for Store
WB for Store
Multi-Cycle Pipeline Diagram

Form showing resource usage

Program execution order (in instructions)

lw $10, 20($1)

sub $11, $2, $3

add $12, $3, $4

lw $13, 24($1)

add $14, $5, $6
Multi-Cycle Pipeline Diagram

- Traditional form

Program execution order (in instructions)

- lw $10, 20($1)
- sub $11, $2, $3
- add $12, $3, $4
- lw $13, 24($1)
- add $14, $5, $6
State of pipeline in a given cycle

- add $14, $5, $6
- lw $13, 24 ($1)
- add $12, $3, $4
- sub $11, $2, $3
- lw $10, 20($1)

Instruction fetch
Instruction decode
Execution
Memory
Write-back
Pipelined Control (Simplified)
Pipelined Control

- Control signals derived from instruction
- As in single-cycle implementation
Pipelined Control
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