ECE/CS 472/572
Computer Architecture: Pipeline Essentials

Prof. Lizhong Chen
Spring 2016
Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance
    - Improve **throughput** rather than latency

- Four loads:
  - Speedup
    \[\text{Speedup} = \frac{8}{3.5} = 2.3\]

- \(n\) loads (non-stop):
  - Speedup
    \[\text{Speedup} = \frac{2n}{(0.5n + 1.5)} \approx 4\]
    \[= \text{number of stages}\]
MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register
MIPS Pipelined Datapath

Right-to-left flow leads to hazards
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>
</tbody>
</table>
Pipeline Performance

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

Program execution order (in instructions)

- $\text{lw } 1, 100(0)$
- $\text{lw } 2, 200(0)$
- $\text{lw } 3, 300(0)$

- Time: 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800

- $800\text{ ps}$

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

Program execution order (in instructions)

- $\text{lw } 1, 100(0)$
- $\text{lw } 2, 200(0)$
- $\text{lw } 3, 300(0)$

- Time: 200, 400, 600, 800, 1000, 1200, 1400

- $200\text{ ps}$

- $800\text{ ps}$
Pipeline Speedup

- If all stages are balanced
  - Speedup is the number of stages
- If not balanced, speedup is less
- Speedup due to increased throughput
- Latency (time for each instruction) does not decrease
Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - x86: 1- to 15-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 stall the starting of 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” (no-op)
- 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
- \texttt{add} $s0$, $t0$, $t1$
- \texttt{sub} $t2$, $s0$, $t3$

```
<table>
<thead>
<tr>
<th>Time</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

“Stall”
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

Program execution order (in instructions)

- add $s0, $t0, $t1
- sub $t2, $s0, $t3
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!
Reorder code to avoid use of load result in the next instruction

C code for $A = B + E; C = B + F;$

13 cycles

11 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
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)`

Prediction incorrect

Program execution order (in instructions)

- `add $4, $5, $6`
- `beq $1, $2, 40`
- `lw $3, 300($0)`
- or `$7, $8, $9`
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 Basics 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

IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back

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
  - “multi-clock-cycle” diagram
    - Graph of operation over time

- We’ll first look at “single-clock-cycle” diagrams for load & store
IF for Load, Store, ...
ID for Load, Store, ...
EX for Load
MEM for Load

[Diagram showing the MEM for Load process with various components such as address, instruction memory, registers, ID/EX, EX/MEM, and MEM/WB stages.]

Iw
Memory
WB for Load

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

- Form showing stage names

![Diagram](image-url)
Multi-Clock-Cycle 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
```

Time (in clock cycles)

```
CC 1  CC 2  CC 3  CC 4  CC 5  CC 6  CC 7  CC 8  CC 9
```

- Instruction fetch
- Instruction decode
- Execution
- Data access
- Write back
Single-Clock-Cycle Diagram

- State of pipeline in a given cycle

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $14, $5, $6</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td>lw $13, 24 ($1)</td>
<td>Instruction decode</td>
</tr>
<tr>
<td>add $12, $3, $4</td>
<td>Execution</td>
</tr>
<tr>
<td>sub $11, $2, $3</td>
<td>Memory</td>
</tr>
<tr>
<td>lw $10, 20($1)</td>
<td>Write-back</td>
</tr>
</tbody>
</table>
## Pipelined Control Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>Reg-Write</th>
<th>Mem-Read</th>
<th>Mem-Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/address calculation stage control lines</th>
<th>Memory access stage control lines</th>
<th>Write-back stage control lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>ALUOp1</td>
<td>ALUOp0</td>
<td>ALUSrc</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>R-format</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>x</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Pipelined Control

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

- Read-After-Write dependence
  - add $s0, $t0, $t1
  - sub $t2, $s0, $t3

- If reversed
  - Read the old value!
WAR Dependence

- Write-After-Read dependence
  - sub $t2, $s0, $t3
  - add $s0, $t0, $t1

- If reversed
  - Read a future value!
RAR Dependence?

- Read-After-Read dependence
  - add $t0, $s0, $t1
  - sub $t2, $s0, $t3

- If reversed
  - No problem at all ^_^
WAW Dependence

- Write-After-Write dependence
  - add $s0, $t0, $t1
  - sub $s0, $t2, $t3

- If reversed
  - The final $s0 value is wrong!