Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

Four loads:
- Speedup = 8/3.5 = 2.3

Non-stop:
- Speedup = 2n/0.5n + 1.5 ≈ 4
  = 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
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)

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

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

Program execution order (in instructions)

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

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions_{pipelined} = \frac{\text{Time between instructions}_{nonpipelined}}{\text{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
    - 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
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
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 $t1, 0($t0)
lw $t2, 4($t0)  lw $t2, 4($t0)  lw $t2, 4($t0)
add $t3, $t1, $t2  add $t3, $t1, $t2  add $t3, $t1, $t2
sw $t3, 12($t0)  sw $t3, 12($t0)  sw $t3, 12($t0)
lw $t4, 8($t0)    lw $t4, 8($t0)    lw $t4, 8($t0)
add $t5, $t1, $t4  add $t5, $t1, $t4  add $t5, $t1, $t4
sw $t5, 16($t0)  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 showing the execution order of instructions with timing details.
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

Chapter 4 — The Processor — 47
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

The BIG Picture

- 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
Multi-Cycle Pipeline Diagram

Traditional form
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

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

- Control signals derived from instruction
- As in single-cycle implementation
Pipelined Control
Data Hazards in ALU Instructions

- Consider this sequence:
  ```
  sub $2, $1,$3
  and $12,$2,$5
  or $13,$6,$2
  add $14,$2,$2
  sw $15,100($2)
  ```

- We can resolve hazards with forwarding
  - How do we detect when to forward?
Dependencies & Forwarding

Program execution order (in instructions)

- `sub $2, $1, $3`
- `and $12, $2, $5`
- `or $13, $6, $2`
- `add $14, $2, $2`
- `sw $15, 100($2)`

Time (in clock cycles)

<table>
<thead>
<tr>
<th>Value of register $2</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
</tbody>
</table>
Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg
Fwd from MEM/WB pipeline reg
Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0
Forwarding Paths
Forwarding Conditions

- **EX hazard**
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    
    ForwardA = 10

  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    
    ForwardB = 10

- **MEM hazard**
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    
    ForwardA = 01

  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    
    ForwardB = 01
Double Data Hazard

- Consider the sequence:
  - add \$1, \$1, \$2
  - add \$1, \$1, \$3
  - add \$1, \$1, \$4

- Both hazards occur
  - Want to use the most recent

- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn’t true
Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 01
Datapath with Forwarding
Load-Use Data Hazard

Need to stall for one cycle

Program execution order (in instructions)

- lw $2, 20($1)
- and $4, $2, $5
- or $8, $2, $6
- add $9, $4, $2
- sli $1, $6, $7

Chapter 4 — The Processor — 77
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    - ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
      - (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
- 1-cycle stall allows MEM to read data for lw
  - Can subsequently forward to EX stage
Stall/Bubble in the Pipeline

Program execution order (in instructions)

lw $2, 20(1)

and becomes nop

and $4, $2, $5

cr $8, $2, $6

add $9, $4, $2

Stall inserted here
Stall/Bubble in the Pipeline

Program execution order
(in instructions)

lw $2, 20($1)

and becomes nop

and $4, $2, $5 stalled in ID

or $8, $2, $6 stalled in IF

add $9, $4, $2

Or, more accurately...
Datapath with Hazard Detection
Stalls and Performance

The BIG Picture

- 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
Branch Hazards

- If branch outcome determined in MEM

Program execution order (in instructions)

- 40 beq $1, $3, 28
- 44 and $12, $2, $6
- 48 or $13, $6, $2
- 52 add $14, $2, $2
- 72 lw $4, 50($7)

Flush these instructions (Set control values to 0)
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator

- Example: branch taken
  
  36:    sub  $10, $4, $8  
  40:    beq  $1, $3, 7  
  44:    and  $12, $2, $5  
  48:    or   $13, $2, $6  
  52:    add  $14, $4, $2  
  56:    slt  $15, $6, $7  
          ...  
  72:    lw   $4, 50($7)
Example: Branch Taken

and $12, $2, $5
beq $1, $3, 7
sub $10, $4, $8
before<1>
before<2>

Chapter 4 — The Processor — 86
Example: Branch Taken
Data Hazards for Branches

- If a comparison register is a destination of 2\textsuperscript{nd} or 3\textsuperscript{rd} preceding ALU instruction

- Can resolve using forwarding
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

lw   $1, addr
add  $4, $5, $6
beq  stalled
beq  $1, $4, target
Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```assembly
lw $1, addr
beq stalled
beq stalled
beq $1, $0, target
```