Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

Four loads:
- Speedup \( = \frac{8}{3.5} = 2.3 \)

Non-stop:
- Speedup \( = \frac{2n}{0.5n + 1.5} \approx 4 \)
  - \( = \) 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)

<table>
<thead>
<tr>
<th>Time (ps)</th>
<th>Reg</th>
<th>ALU</th>
<th>Data access</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Time (ps)</th>
<th>Reg</th>
<th>ALU</th>
<th>Data access</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions_{\text{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
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 an instruction from entering the next stage.
  - Structural 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
Structural 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 work best with 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`

```
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

more on this later....
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding alone
  - If value not computed when needed
  - Can’t forward backward in time!

Program execution order (in instructions)

```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```
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 $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

Stall

```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
add $t5, $t1, $t4
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
  - Nominally, branch condition is resolved in EX stage, which leaves us with two stalls.
  - They added branch delay slots – instruction after branch is always executed. Now we only have one stall to get rid of.
Branch Prediction

- Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Kill instructions after branch if prediction is wrong
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
MEM for Store
WB for Store
Multi-Cycle Pipeline Diagram

- Form showing resource usage

Diagram showing the execution order of 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
Single-Cycle Pipeline Diagram

State of pipeline in a given cycle
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)

<table>
<thead>
<tr>
<th>Time (in clock cycles)</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>Value of register $2:</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
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

b. With forwarding
Forwarding Conditions

- **EX hazard**
  - if \((\text{EX/MEM.RegWrite and (EX/MEM.RegisterRd} \neq 0) \text{ and (EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs}))\)
    \[
    \text{ForwardA} = 10
    \]
  - if \((\text{EX/MEM.RegWrite and (EX/MEM.RegisterRd} \neq 0) \text{ and (EX/MEM.RegisterRd} = \text{ID/EX.RegisterRt}))\)
    \[
    \text{ForwardB} = 10
    \]

- **MEM hazard**
  - if \((\text{MEM/WB.RegWrite and (MEM/WB.RegisterRd} \neq 0) \text{ and (MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs}))\)
    \[
    \text{ForwardA} = 01
    \]
  - if \((\text{MEM/WB.RegWrite and (MEM/WB.RegisterRd} \neq 0) \text{ and (MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt}))\)
    \[
    \text{ForwardB} = 01
    \]
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

Time (in clock cycles)

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

Program execution order (in instructions)

Iw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7

Need to stall for one cycle
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 \texttt{lw}
    - Can subsequently forward to EX stage
Stall/Bubble in the Pipeline

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6  CC 7  CC 8  CC 9  CC 10

Program execution order (in instructions)

Iw $2, 20($1)

and becomes nop

and $4, $2, $5

or $8, $2, $6

add $9, $4, $2

Stall inserted here
Stall/Bubble in the Pipeline

Program execution order (in instructions)

`lw $2, 20($1)` 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
Exceptions and Interrupts

- “Unexpected” events requiring change in flow of control
  - Different ISAs use the terms differently

- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, ...

- Interrupt
  - From an external I/O controller

- Dealing with them without sacrificing performance is hard
Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We’ll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180
An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause

Example:
- Undefined opcode: C000 0000
- Overflow: C000 0020
- ...: C000 0040

- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler
Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
  \[ \text{add } \$1, \$2, \$1 \]
  - Prevent \$1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware
Pipeline with Exceptions
Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust
Exception Example

- Exception on `add` in
  ```assembly
  40  sub  $11, $2, $4
  44  and  $12, $2, $5
  48  or  $13, $2, $6
  4c  add  $1, $2, $1
  50  slt  $15, $6, $7
  54  lw  $16, 50($7)
  ...
  ```

- Handler
  ```assembly
  80000180  sw  $25, 1000($0)
  80000184  sw  $26, 1004($0)
  ...
  ```
Exception Example

sw $25, 1000($0)
bubble (nop)
bubble
bubble
or $13, ...

Clock 7
Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!
Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines
Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards

- Pipelining is independent of technology
  - So why haven’t we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions
Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
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
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall