What do we have so far?
Multi-Cycle Datapath

CPI: R-Type = 4, Load = 5, Store 4, Branch = 3
Only one instruction being processed in datapath

How to lower CPI further?
Instruction Pipelining

• Instruction pipelining is a CPU implementation technique where multiple operations on a number of instructions are overlapped.
  – The next instruction is fetched in the next cycle without waiting for the current instruction to complete.
• An instruction execution pipeline involves a number of steps, where each step completes one part of an instruction.
• Each step is called a pipeline stage or a pipeline segment.
• The stages or steps are connected one to the next to form a pipeline -- instructions enter at one end and progress through the stages and exit at the other end when completed.
• Instruction Pipeline Throughput: The instruction completion rate of the pipeline and is determined by how often an instruction exists the pipeline.
• The time to move an instruction one step down the line is is equal to the machine cycle and is determined by the stage with the longest processing delay.
• Instruction Pipeline Latency: The time required to complete an instruction: Cycle time \( \times \) Number of pipeline stages.
Single Cycle Vs. Pipelining

**Program execution order**

**Single Cycle**

- lw $1, 100($0)  
- lw $2, 200($0)  
- lw $3, 300($0)

**Time for 1000 instructions = 8 x 1000 = 8000 ns**

**Pipelining**

- lw $1, 100($0)  
- lw $2, 200($0)  
- lw $3, 300($0)

**Time for 1000 instructions = time to fill pipeline + cycle time x 1000 = 8 + 2 x 1000 = 2008 ns**

**Pipelining Speedup = 8000/2008 = 3.98**
Pipelining: Design Goals

• The length of the machine clock cycle is determined by the time required for the slowest pipeline stage.

• An important pipeline design consideration is to balance the length of each pipeline stage.

• If all stages are perfectly balanced, then the time per instruction on a pipelined machine (assuming ideal conditions with no stalls):

\[
\text{Time per instruction on unpipelined machine} \quad \frac{\text{Number of pipe stages}}{
\text{Number of pipe stages}}
\]

• Under these ideal conditions:
  – Speedup from pipelining = the number of pipeline stages = k
  – One instruction is completed every cycle: \( \text{CPI} = 1 \).
From MIPS Multi-Cycle Datapath:
Five Stages of Load

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5

Load IF ID EX MEM WB

1- Instruction Fetch (IF) Instruction Fetch
   • Fetch the instruction from the Instruction Memory.

2- Instruction Decode (ID): Registers Fetch and Instruction Decode.

3- Execute (EX): Calculate the memory address.

4- Memory (MEM): Read the data from the Data Memory.

5- Write Back (WB): Write the data back to the register file.
## Pipelined Instruction Processing Representation

<table>
<thead>
<tr>
<th>Instruction Number</th>
<th>Clock cycle Number</th>
<th>Time in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6</td>
<td>7 8 9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction I</th>
<th>IF ID EX MEM WB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction I+1</td>
<td>IF ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td>Instruction I+2</td>
<td>IF ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td>Instruction I+3</td>
<td>IF ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td>Instruction I+4</td>
<td>IF ID EX MEM WB</td>
<td></td>
</tr>
</tbody>
</table>

Pipeline Stages:
- **IF** = Instruction Fetch
- **ID** = Instruction Decode
- **EX** = Execution
- **MEM** = Memory Access
- **WB** = Write Back

Time to fill the pipeline

First instruction, I Completed

Last instruction, I+4 completed
Pipelined Instruction Processing Representation
Single Cycle, Multi-Cycle, Vs. Pipeline

Single Cycle Implementation:

<table>
<thead>
<tr>
<th>Clk</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load</td>
<td>Store</td>
</tr>
<tr>
<td></td>
<td>8 ns</td>
<td>Waste</td>
</tr>
</tbody>
</table>

Multiple Cycle Implementation:

<table>
<thead>
<tr>
<th>Clk</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
<th>Cycle 9</th>
<th>Cycle 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipeline Implementation:

<table>
<thead>
<tr>
<th>Clk</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load</td>
<td>Store</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
</tr>
</tbody>
</table>

EECC550 - Shaaban
Single Cycle, Multi-Cycle, Pipeline: Performance Comparison Example

For 1000 instructions, execution time:

• Single Cycle Machine:
  – $8 \text{ ns/cycle} \times 1 \text{ CPI} \times 1000 \text{ inst} = 8000 \text{ ns}$

• Multicycle Machine:
  – $2 \text{ ns/cycle} \times 4.6 \text{ CPI (due to inst mix)} \times 1000 \text{ inst} = 9200 \text{ ns}$

• Ideal pipelined machine, 5-stages:
  – $2 \text{ ns/cycle} \times (1 \text{ CPI} \times 1000 \text{ inst} + 4 \text{ cycle fill}) = 2008 \text{ ns}$
MIPS Pipeline Stage Identification

What is needed to divide datapath into pipeline stages?
MIPS: An Initial Pipelined Datapath

Can you find a problem even if there are no dependencies?
What instructions can we execute to manifest the problem?
A Corrected Pipelined Datapath

IF
Instruction Fetch

ID
Instruction Decode

EX
Execution

MEM
Memory

WB
Write Back

EECC550 - Shaaban
Representing Pipelines Graphically

Can help with answering questions like:

- How many cycles does it take to execute this code?
- What is the ALU doing during cycle 4?
- Use this representation to help understand datapaths
Adding Pipeline Control Points

IF/ID

ID/EX

EX/MEM

MEM/WB

Address

Instruction memory

IF/ID

ID/EX

EX/MEM

MEM/WB

Address

Instruction memory

Instruction

[15–11]

[16–0]

[20–16]

Instruction

[15–11]

[16–0]

[20–16]

Instruction

[15–11]

[16–0]

[20–16]

Instruction

[15–11]

[16–0]

[20–16]

Instruction

[15–11]

[16–0]

[20–16]
Pipeline Control

- Pass needed control signals along from one stage to the next as the instruction travels through the pipeline just like the data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/Address Calculation</th>
<th>Memory access stage</th>
<th>stage control</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst ALU Op1 ALU Op0 ALU Src</td>
<td>Branch Mem Read Mem Write</td>
<td>Reg write Mem to Reg</td>
</tr>
<tr>
<td>R-format</td>
<td>1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>0 0 0 0 1 0 1 0 0 1 0 1 0 1 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>X 0 0 1 0 0 0 1 0 0 1 0 0 1 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>X 0 1 0 0 0 1 0 0 0 0 0 0 1 0 X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram: IF/ID → ID/EX → EX/MEM → MEM/WB → Control → WB
Pipeline Control

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later

IF/ID Register
ID/Ex Register
Ex/Mem Register
Mem/Reg Register

Main Control

ID
EX
Mem
WB
Pipelined Datapath with Control Added

Target address of branch determined in MEM
Basic Performance Issues In Pipelining

• Pipelining increases the CPU instruction throughput: The number of instructions completed per unit time. Under ideal condition instruction throughput is one instruction per machine cycle, or CPI = 1

• Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency).

• It usually slightly increases the execution time of each instruction over unpipelined implementations due to the increased control overhead of the pipeline and pipeline stage registers delays.
Pipelining Performance Example

- Example: For an unpipelined machine:
  - Clock cycle = 10ns, 4 cycles for ALU operations and branches and 5 cycles for memory operations with instruction frequencies of 40%, 20% and 40%, respectively.
  - If pipelining adds 1ns to the machine clock cycle then the speedup in instruction execution from pipelining is:

Non-pipelined Average instruction execution time = Clock cycle \times Average CPI

\[ = 10 \text{ ns} \times ((40\% + 20\%) \times 4 + 40\% \times 5) = 10 \text{ ns} \times 4.4 = 44 \text{ ns} \]

In the pipelined five implementation five stages are used with an average instruction execution time of: 10 ns + 1 ns = 11 ns

Speedup from pipelining = \frac{\text{Instruction time unpipelined}}{\text{Instruction time pipelined}}

\[ = \frac{44 \text{ ns}}{11 \text{ ns}} = 4 \text{ times} \]
Pipeline Hazards

- Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle resulting in one or more stall cycles.

- Hazards reduce the ideal speedup gained from pipelining and are classified into three classes:
  - *Structural hazards*: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.
  - *Data hazards*: Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
  - *Control hazards*: Arise from the pipelining of conditional branches and other instructions that change the PC.
Structural Hazards

- In pipelined machines, overlapped instruction execution requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.

- If a resource conflict arises due to a hardware resource being required by more than one instruction in a single cycle, and one or more such instructions cannot be accommodated, then a structural hazard has occurred, for example:
  - when a machine has only one register file write port
  - or when a pipelined machine has a shared single-memory pipeline for data and instructions.

→ stall the pipeline for one cycle for register writes or memory data access
Structural hazard Example:
Single Memory For Instructions & Data

Time (clock cycles)

Detection is easy in this case (right half highlight means read, left half write)
Data Hazards Example

- Problem with starting next instruction before first is finished
  - Data dependencies here that “go backward in time” create data hazards.

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```
Data Hazard Resolution: Stall Cycles

Stall the pipeline by a number of cycles. The control unit must detect the need to insert stall cycles. In this case two stall cycles are needed.

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)

Value of register $2:
- CC 1: 10
- CC 2: 10
- CC 3: 10
- CC 4: 10
- CC 5: 10–20
- CC 6: –20
- CC 7: –20
- CC 8: –20
- CC 9: –20
- CC 10: –20
- CC 11: –20

Time (in clock cycles):
- IM Reg
- CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9 CC 10 CC 11

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)
Performance of Pipelines with Stalls

• Hazards in pipelines may make it necessary to stall the pipeline by one or more cycles and thus degrading performance from the ideal CPI of 1.

\[\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}\]

• If pipelining overhead is ignored and we assume that the stages are perfectly balanced then:

\[\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}\]

• When all instructions take the same number of cycles and is equal to the number of pipeline stages then:

\[\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}\]
Data Hazard Resolution: Compiler Instruction Scheduling

• The compiler can guarantee that no data hazards exist by re-ordering instructions and/or adding NOP instructions where needed.

• For the previous example:

```
sub $2, $1, $3
nop
nop
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```
Data Hazard Resolution: Forwarding

• Observation:
  Why not use temporary results produced by memory/ALU and not wait for them to be written back in the register bank.

• Forwarding is a hardware-based technique (also called register bypassing or short-circuiting) used to eliminate or minimize data hazard stalls that makes use of this observation.

• Using forwarding hardware, the result of an instruction is copied directly from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)
Data Hazard Resolution: Forwarding

- Register file forwarding to handle read/write to same register
- ALU forwarding
Pipelined Datapath With Forwarding

PC

Instruction memory

IF/ID

Instruction

Registers

ID/EX

Control

EX

M

WB

EX/MEM

MEM/WB

Data memory

Forwarding unit

IF/ID.RegisterRs

IF/ID.RegisterRt

IF/ID.RegisterRt

IF/ID.RegisterRd

Rt

Rt

Rd

Rs

EX/MEM.RegisterRd

MEM/WB.RegisterRd
Data Hazard Example With Forwarding

<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>
<tr>
<td>Value of EX/MEM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Value of MEM/WB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Program execution order (in instructions)

- sub $2$, $1$, $3$
- and $12$, $2$, $5$
- or $13$, $6$, $2$
- add $14$, $2$, $2$
- sw $15$, 100($2$)
A Data Hazard Requiring A Stall

A load followed by an R-type instruction that uses the loaded value

Even with forwarding in place a stall cycle is needed
This condition must be detected by hardware
A Data Hazard Requiring A Stall

A load followed by an R-type instruction that uses the loaded value

- We can stall the pipeline by keeping an instruction in the same stage
Compiler Scheduling Example

- Reorder the instructions to avoid as many pipeline stalls as possible:
  \[
  \begin{align*}
  \text{lw} & \quad $15, 0 ($2) \\
  \text{lw} & \quad $16, 4 ($2) \\
  \text{sw} & \quad $16, 0 ($2) \\
  \text{sw} & \quad $15, 4 ($2)
  \end{align*}
  \]

- The data hazard occurs on register $16$ between the second \text{lw} and the first \text{sw} resulting in a stall cycle

- With forwarding we need to find only one independent instructions to place between them, swapping the \text{lw} instructions works:
  \[
  \begin{align*}
  \text{lw} & \quad $15, 0 ($2) \\
  \text{lw} & \quad $16, 4 ($2) \\
  \text{nop} & \\
  \text{nop} & \\
  \text{sw} & \quad $15, 0 ($2) \\
  \text{sw} & \quad $16, 4 ($2)
  \end{align*}
  \]

- Without forwarding we need three independent instructions to place between them, so in addition two \text{nops} are added.
  \[
  \begin{align*}
  \text{lw} & \quad $15, 0 ($2) \\
  \text{lw} & \quad $16, 4 ($2) \\
  \text{nop} & \\
  \text{nop} & \\
  \text{sw} & \quad $15, 0 ($2) \\
  \text{sw} & \quad $16, 4 ($2)
  \end{align*}
  \]
Datapath With Hazard Detection Unit

A load followed by an instruction that uses the loaded value is detected and a stall cycle is inserted.
Control Hazards: Example

- Three other instructions are in the pipeline before branch instruction target decision is made when BEQ is in MEM stage.

- In the above diagram, we are predicting “branch not taken”
  - Need to add hardware for flushing the three following instructions if we are wrong losing three cycles.
Reducing Delay of Taken Branches

- Next PC of a branch known in MEM stage: Costs three lost cycles if taken.
- If next PC is known in EX stage, one cycle is saved.
- Branch address calculation can be moved to ID stage using a register comparator, costing only one cycle if branch is taken.
Pipeline Performance Example

• Assume the following MIPS instruction mix:

<table>
<thead>
<tr>
<th>Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>40%</td>
</tr>
<tr>
<td>Load</td>
<td>30%</td>
</tr>
<tr>
<td></td>
<td>of which 25% are followed immediately by an instruction using the loaded value</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
</tr>
<tr>
<td>branch</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>of which 45% are taken</td>
</tr>
</tbody>
</table>

• What is the resulting CPI for the pipelined MIPS with forwarding and branch address calculation in ID stage?

CPI = Ideal CPI + Pipeline stall clock cycles per instruction

\[
\text{CPI} = 1 + \text{stalls by loads} + \text{stalls by branches} \\
= 1 + 0.3 \times 0.25 \times 1 + 0.2 \times 0.45 \times 1 \\
= 1 + 0.075 + 0.09 \\
= 1.165
\]