Pipeline hazards

• What makes pipelining easy
  – all instructions are the same length
  – just a few instruction formats
  – memory operands appear only in loads and stores

• What makes it hard?
  – structural hazards: suppose we had only one memory
  – control hazards: need to worry about branch instructions
  – data hazards: an instruction depends on a previous instruction

• We’ll build a simple pipeline and look at these issues

Soak                 Brush Rinse Dry

Example: pipelined car wash with one hose for both soaking and rinsing.

How do you solve the problem?
(share hardware, replicate hardware, compete for hardware)

Are there similar problems in our MIPS architecture?
### Structural hazards in MIPS

**Potential problem:** Both IF and MEM use memory  
**Solution:** use separate memories (caches)

**Potential problem:** Both REG and WB use the register file  
**Solution:** Read from register file during the first half of a cycle and write to register file during the second half of a cycle.

### Control Hazards

Where are branch conditions and target addresses resolved (when is the PC overwritten with the branch target address)?

When the branch instruction is in MEM stage  
- the 3 instructions following the branch are already in the pipeline

When the branch instruction is in EX stage  
- the 2 instructions following the branch are already in the pipeline  
- How does this affect the cycle time?
Control Hazards

Assume that the decision about “branching” takes place in the EX stage. Hence, when a branch decision is made two instructions are already in the pipe (started execution).

What is wrong and what can be done?

Example: consider the execution of the following code segment:

```
add $4, $5, $6
beq $1, $2, 10
lw $3, 300(0)
```

Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7

What is wrong and what can be done?

Example: consider the execution of the following code segment:

```
add $4, $5, $6
beq $1, $2, 10
lw $3, 300(0)
```

Thus, it appears two instructions are in the pipe: add $4, $5, $6, beq $1, $2, 10. When the branch decision is made, two instructions are already started. This is a problem because the branch instruction is in the EX stage and the add instruction is in the MEM stage.

Introducing bubbles to kill unwanted instructions

A bubble is a no-op introduced by the hardware (we will learn how later).

```
add $4, $5, $6
beq $1, $2, 10
lw $3, 300(0)
```

Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7

What is wrong and what can be done?

Example: consider the execution of the following code segment:

```
add $4, $5, $6
beq $1, $2, 10
lw $3, 300(0)
```

Thus, it appears two instructions are in the pipe: add $4, $5, $6, beq $1, $2, 10. When the branch decision is made, two instructions are already started. This is a problem because the branch instruction is in the EX stage and the add instruction is in the MEM stage.

Introducing bubbles to kill unwanted instructions

A bubble is a no-op introduced by the hardware (we will learn how later).
Adding no-ops (a software solution)

Make the compiler add no-ops after the branch instruction.

<table>
<thead>
<tr>
<th>Cycle 2</th>
<th>IF stage</th>
<th>ID stage</th>
<th>EX stage</th>
<th>MEM stage</th>
<th>WB stage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>beq $1, $2, 10</td>
<td>add $4, $5, $6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>no-op</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 3</td>
<td>beq $1, $2, 10</td>
<td>add $4, $5, $6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>no-op</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 4</td>
<td>lw $3, 300($0)</td>
<td>add $4, $5, $6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>no-op</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 5</td>
<td>no-op</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The branch condition will be resolved in cycle 4 and the correct instruction will enter the pipe in cycle 5

The effect of control hazard on throughput

- Assume that when the branch is resolved, K instructions following the branch are already in the pipeline.
- If control hazards are dynamically resolved, then each taken branch introduces K bubbles in the pipeline.
- Hence, the average number of clock cycles to execute an instruction is
  \[ CPI = 1 + \alpha \times \pi \times K \]
  where
  - 1 is the CPI with no hazard
  - \( \alpha \) is the fraction of branch instructions in the instruction mix
  - \( \pi \) is the probability a branch is actually taken
- For the software solution, where the compiler adds K no-ops after each branch,
  \[ CPI = 1 + \alpha \times K \]

**Example:** if branches are dynamically resolved in the EX stage, 10% of the instructions are branches and the probability that a branch is taken is 40%, then
\[ CPI = 1 + 2 \times 0.1 \times 0.4 = 1 + 0.08 = 1.08 \] cycles per instruction

Hence, the average execution time of an instruction is 1.08 * clock cycle time
Data hazards

Assume that registers 4, 5 and 6 contain the values 100, 200 and 300, respectively

**Expected execution:**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction</th>
<th>Value Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>load $3, 300($4)</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>ID</td>
<td>load $3, 300($4)</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>EX</td>
<td>load $3, 300($4)</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>MEM</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>WB</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction</th>
<th>Value Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>ID</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>EX</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>MEM</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>WB</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
</tr>
</tbody>
</table>

**Pipelined execution:**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>IF stage</th>
<th>ID stage</th>
<th>EX stage</th>
<th>MEM stage</th>
<th>WB stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>sub $1, $2, $4</td>
<td>add $4, $5, $6</td>
<td>sub $1, $2, $4</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>3</td>
<td>lw $3, 300($4)</td>
<td>sub $1, $2, $4</td>
<td>lw $3, 300($4)</td>
<td>sub $1, $2, $4</td>
<td>lw $3, 300($4)</td>
</tr>
<tr>
<td>4</td>
<td>and $7, $8, $4</td>
<td>lw $3, 300($4)</td>
<td>and $7, $8, $4</td>
<td>lw $3, 300($4)</td>
<td>and $7, $8, $4</td>
</tr>
</tbody>
</table>

**What can be done?**

**Forwarding**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>IF stage</th>
<th>ID stage</th>
<th>EX stage</th>
<th>MEM stage</th>
<th>WB stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>lw $3, 300($4)</td>
<td>sub $1, $2, $4</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>4</td>
<td>and $7, $8, $4</td>
<td>lw $3, 300($4)</td>
<td>sub $1, $2, $4</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Solution:**

- at the end of cycle 3, “add” stores the correct value of $4 in the EX/MEM buffer
- at the start of cycle 4, “sub” replaces the incorrect data read in cycle 3 by the correct data stored in the ID/EX buffer.
Forwarding may not be enough

<table>
<thead>
<tr>
<th>IF stage</th>
<th>ID stage</th>
<th>EX stage</th>
<th>MEM stage</th>
<th>WB stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle 3</td>
<td>and $7, $8, $3</td>
<td>$w3, 300($4)</td>
<td>$w3, 300($4)</td>
<td>$w3, 300($4)</td>
</tr>
<tr>
<td>Cycle 4</td>
<td>and $7, $8, $3</td>
<td>$w3, 300($4)</td>
<td>$w3, 300($4)</td>
<td>$w3, 300($4)</td>
</tr>
<tr>
<td>Cycle 5</td>
<td>and $7, $8, $3</td>
<td>$w3, 300($4)</td>
<td>$w3, 300($4)</td>
<td>$w3, 300($4)</td>
</tr>
</tbody>
</table>

Problem: can’t use forwarding since “lw” does not have the correct data at the end of cycle 3

Solution: need to combine forwarding with stalling the pipe.

Software solution to avoid stalling

Should stall the pipeline

lw $t1, 0($t0)
lw $t5, 4($t0)
sw $t5, 0($t0)
sw $t1, 4($t0)

Can the compiler help by rearranging code??
If hardware does not resolve data hazards?

Example:

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

If no hardware for forwarding, then data dependence will cause data hazard.

The compiler can come to the rescue

```
sub $2, $1, $3
no-op
no-op
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

Problem: increases the CPI