Pipeline Hazards
Pipelined Execution Representation

- Every instruction must take the same number of steps, so some stages will idle.
- E.g., the MEM stage for any arithmetic instruction.

<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></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Diagram showing the pipeline stages over time.
Graphical Pipeline Diagrams

- Use datapath figure below to represent pipeline:
Graphical Pipeline Representation

- RegFile: left half is write, right half is read

Time (clock cycles)
Pipelining Performance (1/3)

- Use $T_c$ (“time between completion of instructions”) to measure speedup

$$T_{c,\text{pipelined}} \geq \frac{T_{c,\text{single-cycle}}}{\text{Number of stages}}$$

- Equality only achieved if stages are balanced (i.e. take the same amount of time)
- If not balanced, speedup is reduced
- Speedup due to increased throughput
  - Latency for each instruction does not decrease
  - In fact, latency must increase as the pipeline registers themselves add delay (why Nick's Ph.D. thesis has a "this was a stupid idea" chapter)
Pipelining Performance (2/3)

• Assume time for stages is
  • 100ps for register read or write
  • 200ps for other stages

<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>

• What is pipelined clock rate?
  • Compare pipelined datapath with single-cycle datapath
Pipelining Performance (3/3)

**Single-cycle**
- $T_c = 800$ ps
- $f = 1.25$GHz

**Pipelined**
- $T_c = 200$ ps
- $f = 5$GHz
Administrivia…

• Guerilla Section tonight, 7-9pm, 310 soda
• Start on Project 3-1 now
  • Logisim can be a bit, well, tedious:
    The project isn’t necessarily hard but it will take a fair amount of time
    • Alternative would be to have you learn *yet another programming language* in this class!
• Why is is simplified from last semester…
  • Nick’s solution for part 1 last time:
    it took Nick about an hour of tediously drawing lines for his solution to part 1
    • 5 minutes to know what he wanted to do…
    • And 55 minutes to actually do it. 😟
Pipelining Hazards

- A hazard is a situation that prevents starting the next instruction in the next clock cycle

- **Structural** hazard
  - A required resource is busy
    (e.g. needed in multiple stages)

- **Data** hazard
  - Data dependency between instructions
  - Need to wait for previous instruction to complete its data read/write

- **Control** hazard
  - Flow of execution depends on previous instruction
Structural Hazard #1: Single Memory

**Try reading same memory twice in same clock cycle**
Solving Structural Hazard #1 with Caches
Structural Hazard #2: Registers (1/2)

Can we read and write to registers simultaneously?
Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  - Split RegFile access in two: Write during 1st half and Read during 2nd half of each clock cycle
  - Possible because RegFile access is VERY fast (takes less than half the time of ALU stage)
  - Build RegFile with independent read and write ports (E.g. for your project)

- Conclusion: Read and Write to registers during same clock cycle is okay

- Structural hazards can *(almost)* always be removed by adding hardware resources
Data Hazards (1/2)

- Consider the following sequence of instructions:

```assembly
add $t0, $t1, $t2
sub $t4, $t0, $t3
and $t5, $t0, $t6
or $t7, $t0, $t8
xor $t9, $t0, $t10
```
2. Data Hazards (2/2)

- Data-flow *backwards* in time are hazards

![Diagram showing data flow and hazards in a pipeline]

Time (clock cycles)

- add $t0,t1,t2$
- sub $t4,t0,t3$
- and $t5,t0,t6$
- or $t7,t0,t8$
- xor $t9,t0,t10$
Data Hazard Solution: **Forwarding**

- Forward result as soon as it is available
- OK that it’s not stored in RegFile yet, it just needs to be calculated!

- add $t0,$t1,$t2
- sub $t4,$t0,$t3
- and $t5,$t0,$t6
- or $t7,$t0,$t8
- xor $t9,$t0,$t10
Datapath for Forwarding (1/2)

- What changes need to be made here?
Datapath for Forwarding (2/2)

- Handled by *forwarding unit*
Datapath and Control

- The control signals are pipelined, too
**Data Hazard: Loads (1/3)**

- **Recall:** Dataflow backwards in time are hazards

```
  lw $t0,0($t1)
  sub $t3,$t0,$t2
```

- Can’t solve all cases with forwarding
  - Must *stall* instruction dependent on load, then forward (more hardware)
Data Hazard: Loads (2/3)

- Stalled instruction converted to “bubble”, acts like nop

\[
\text{lw } \$t0, 0(\$t1) \\
\text{sub } \$t3, \$t0, \$t2 \\
\text{sub } \$t3, \$t0, \$t2 \\
\text{and } \$t5, \$t0, \$t4 \\
\text{or } \$t7, \$t0, \$t1
\]

First two pipeline stages stall by repeating stage one cycle later.
Data Hazard: Loads (3/3)

- Slot after a load is called a **load delay slot**
- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
- Letting the hardware stall the instruction in the delay slot is equivalent to putting an explicit `nop` in the slot (except the latter uses more code space).

**Idea:** Let the compiler put an unrelated instruction in that slot $\rightarrow$ no stall!
Clicker Question

How many cycles (pipeline fill+process+drain) does it take to execute the following code?

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)

A. 7
B. 9
C. 11
D. 13
E. 14
Code Scheduling to Avoid Stalls

• Reorder code to avoid use of load result in the next instruction!
• MIPS code for $D=A+B; \ E=A+C;$

<table>
<thead>
<tr>
<th># Method 1:</th>
<th># Method 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t1, 0($t0)</td>
<td>lw $t1, 0($t0)</td>
</tr>
<tr>
<td>lw $t2, 4($t0)</td>
<td>lw $t2, 4($t0)</td>
</tr>
<tr>
<td>add $t3, $t1, $t2</td>
<td>lw $t4, 8($t0)</td>
</tr>
<tr>
<td>sw $t3, 12($t0)</td>
<td>add $t3, $t1, $t2</td>
</tr>
<tr>
<td>lw $t4, 8($t0)</td>
<td>sw $t3, 12($t0)</td>
</tr>
<tr>
<td>add $t5, $t1, $t4</td>
<td>add $t5, $t1, $t4</td>
</tr>
<tr>
<td>sw $t5, 16($t0)</td>
<td>sw $t5, 16($t0)</td>
</tr>
</tbody>
</table>

13 cycles 11 cycles
3. 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
- BEQ, BNE in MIPS pipeline
- Simple solution Option 1: *Stall* on every branch until branch condition resolved
  - Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)
Stall => 2 Bubbles/Clocks

Where do we do the compare for the branch?
Control Hazard: Branching

- **Optimization #1:**
  - Insert special branch comparator in Stage 2
  - As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC

- **Benefit:** since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  - Also takes advantage that EQ/NEQ is just a giant AND gate of the results of an XOR
  - Side Note: means that branches are idle in Stages 3, 4 and 5
One Clock Cycle Stall

Branch comparator moved to Decode stage.

Time (clock cycles)

Instr Order

Instr 1

Instr 2

Instr 3

Instr 4

Instr

beq

Instr

Instr

Instr

Instr

Instr
Control Hazards: Branching

- **Option 2: Predict** outcome of a branch, fix up if guess wrong
  - Must cancel all instructions in pipeline that depended on guess that was wrong
  - This is called “flushing” the pipeline
- Simplest hardware if we predict that all branches are NOT taken
  - Why?
Control Hazards: Branching

• Option #3: Redefine branches
  • Old definition: if we take the branch, none of the instructions after the branch
get executed by accident
  • New definition: whether or not we take the branch, the single instruction
  immediately following the branch gets executed (the branch-delay slot)

• Delayed Branch means we always execute one inst after
  branch

• This optimization is used with MIPS
  • “It seemed like a good idea at the time” school of computer architecture
Example: Nondelayed vs. Delayed Branch

Nondelayed Branch

or $8, $9, $10
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
xor $10, $1, $11

Exit:

Delayed Branch

add $1, $2,$3
sub $4, $5, $6
beq $1, $4, Exit
or $8, $9, $10
xor $10, $1, $11

Exit:
Control Hazards: Branching

• Notes on Branch-Delay Slot
  • Worst-Case Scenario: put a nop in the branch-delay slot
  • Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn’t affect the logic of program
    • Re-ordering instructions is common way to speed up programs
    • Compiler usually finds such an instruction 50% of time
    • Jumps also have a delay slot …
More on the Branch Delay Slot

- MIPS MAL does **not have** the branch delay slot
  - So you don’t write the branch delay slot.

- MIPS TAL **does** have the branch delay slot
  - It is up to the *assembler* to relocate or insert a nop into the branch delay slot

- It also changes how *jal/jalr* work in TAL:
  - Instead of $pc + 4, $ra gets $pc + 8
Greater Instruction-Level Parallelism (ILP): Deeper Pipelines

• Deeper pipeline (5 => 10 => 15 stages)
  • Less work per stage $\Rightarrow$ shorter clock cycle

• But you get diminishing returns:
  • More setup and clk->q times
    • Increases latency to complete a single instruction
  • More hazards that you can’t forward
    • E.G. if the ALU takes 2 cycles
Greater Instruction Level Parallelism: Superscalar

• Don’t just have one execution unit
  • Have multiple…

• So read 4 registers instead of 2…
  • And have two independent ALUs…

• Does up performance…
  • But also ups complexity and space
  • And dependencies will stall things more
Greater ILP: Out of order execution & better branch prediction…

- Have the hardware be a lot “smarter”
  - Reorder instructions to minimize dependencies
  - Keep track of which branches are taken or not taken

- Works, but…
  - This REALLY increases complexity

- Want to learn more about this stuff: Take CS152
In Conclusion

• Pipelining increases throughput by overlapping execution of multiple instructions in different pipe stages
• Pipe stages should be balanced for highest clock rate
• Three types of pipeline hazard limit performance
  • Structural (generally fixable with more hardware)
  • Data (use interlocks or bypassing to resolve)
  • Control (reduce impact with branch prediction or branch delay slots)