Pipeline Hazards
Introduction to Computer Architecture

Contents

• Introduction to hazards: the register file
  – Examples
  – Cheating with double-pumping

• Data hazards
  – Stalling
  – Forwarding

• Control hazards
  – Branches
  – Branch delay slots

• Structural hazards

• Performance impact of hazards
Instruction execution model

How to think about instructions

Processor execution model

- Processor promises that the instruction execution will appear to be sequential and atomic
  - **Sequential**: execute instructions in-order
  - **Atomic**: execute each instruction all at once

- **Sequential**
  - Program says: \( R_2 = R_1 + R_2 \) then \( R_3 = R_1 + R_2 \)
  - Processor may not do: \( R_3 = R_1 + R_2 \) then \( R_2 = R_1 + R_2 \) \( \leftarrow \) wrong result

- **Atomic**
  - Program says: \( R_2 = R_1 + R_2 \) then \( R_3 = R_1 + R_2 \)
  - Processor has to finish \( R_2 = R_1 + R_2 \) before starting \( R_3 = R_1 + R_2 \)

- **Processors don’t do either of these things** (too slow)
  - But it’s important that they make it look like they do
  - We’ll talk a lot about this when we get to pipelines

Why do we care?
If a program is not sequential and atomic we can’t figure out what it should do. Impossible to debug and program.
Hazards

• Hazards exist when we have multiple instructions that want the same resource or result

• They occur because we are breaking the ISA’s promises of serial and atomic execution
  – Remember: programmers assume instructions finish completely in the order written!

• We can fix many hazards by adding logic to the processor to move results

• Some hazards can not be fixed because we don’t have the results yet

Taxonomy of hazards

• Data hazards
  – Instruction depends on data that is not ready yet
    (Data dependencies through the register file)

• Control hazards
  – The choice of next instruction depends on results that aren’t ready yet
    (We’ll see this next)

• Structural hazards
  – Hardware cannot support a combination of instructions
### General data hazards

- **Data is needed by another instruction before it is “ready”**
  - Ready: *where* it is expected (e.g., in the register file)
  - Ready: result *done* being calculated (e.g., after the EX stage)

- **Why are we having these complications?**
  - Remember what the ISA promises:
    - **Sequential execution** (one after another)
    - **Atomic execution** (each one finishes all at once)
  - Well, *pipelining breaks these promises*:
    - **Parallel execution** (instructions are executing at the same time)
    - **Multi-cycle execution** (instructions start before others are finished)

- Dependencies between close instructions now pose problems because we are breaking the promise we made in the ISA!
**Huston we have a problem...**

**What happened?**

- ISA promised `add #1` would execute all at once and before `add #2`
- Pipeline executes `add #1` across 5 cycles and `add #2` at the same time
- Programmer assumed the instructions were independent, but they aren’t in the pipeline!
Example: detecting hazards

<table>
<thead>
<tr>
<th>Operation</th>
<th>Rd, Rs, Rt</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>R2, R1, R3</td>
<td>Rd = R2, Rs = R1, Rt = R3</td>
</tr>
<tr>
<td>and</td>
<td>R12, R2, R5</td>
<td>Rd = R12, Rs = R2, Rt = R5</td>
</tr>
<tr>
<td>or</td>
<td>R13, R6, R2</td>
<td>Rd = R13, Rs = R6, Rt = R2</td>
</tr>
<tr>
<td>add</td>
<td>R14, R2, R2</td>
<td>Rd = R14, Rs = R2, Rt = R2</td>
</tr>
<tr>
<td>sw</td>
<td>R15, 100(R2)</td>
<td>Rd = R15, Rs = R2, Rt = XX</td>
</tr>
</tbody>
</table>

- **sub-and** hazard
  - MEM.Register_Rd == EX.Register_Rs (R2)

- **sub-or** hazard
  - WB.Register_Rd == EX.Register_Rt (R2)

No hazard between sub and add

In the first half of the clock cycle, sub writes to the RF. In the second half of the clock cycle, add reads from the RF.
Compiler inserted stalls (NOPs)

<table>
<thead>
<tr>
<th>Program Execution</th>
<th>Time</th>
<th>Clock Cycle</th>
<th>Clock Cycle</th>
<th>Clock Cycle</th>
<th>Clock Cycle</th>
<th>Clock Cycle</th>
<th>Clock Cycle</th>
<th>Clock Cycle</th>
<th>Clock Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub R2, R1, R3</td>
<td>IM</td>
<td>RF</td>
<td>DM</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
</tr>
<tr>
<td>add R0, R0, R0</td>
<td>IM</td>
<td>RF</td>
<td>DM</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
</tr>
<tr>
<td>add R0, R0, R0</td>
<td>IM</td>
<td>RF</td>
<td>AU</td>
<td>DM</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
</tr>
<tr>
<td>and R12, R2, R5</td>
<td>IM</td>
<td>RF</td>
<td>AU</td>
<td>DM</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
</tr>
<tr>
<td>or R13, R6, R2</td>
<td>IM</td>
<td>RF</td>
<td>AU</td>
<td>DM</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
<td>RF</td>
</tr>
</tbody>
</table>

NOP instructions don’t do anything so they act as delays. Still waste 2/5 of the cycles in this example.

Fixing the pipeline for data hazards

Forwarding data
Two types of data hazards

• Data is ready, but not where we expect it
  – Results are in the EX, MEM, or WB stage
  – But not in the register file

• Data is not yet ready
  – Results are being calculated now or in the future

*We’ll use the predicting the future approach to fix a similar problem in the next lecture.

Forwarding

• Often the data is available, just not where we expect it:
  – In the EX stage, but not in the RF
  – In the MEM stage, but not in the RF

• We can forward the data to where we need it:
  – Grab the result from wherever it is and send (copy) it to where we need it
    (The result will be written to the register file later)
  – Can use it earlier since it’s ready, but need to send it to the right place

• How?
  – Add more MUXes, wires, and logic to send the values around
**Where does forwarding help?**

Program Execution Time

<table>
<thead>
<tr>
<th>Clock Cycle 1</th>
<th>Clock Cycle 2</th>
<th>Clock Cycle 3</th>
<th>Clock Cycle 4</th>
<th>Clock Cycle 5</th>
<th>Clock Cycle 6</th>
<th>Clock Cycle 7</th>
<th>Clock Cycle 8</th>
<th>Clock Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td>RF</td>
<td>R2 Dm</td>
<td>R2</td>
<td>DM</td>
<td>R2 ALU</td>
<td>R2</td>
<td>DM</td>
<td>RF</td>
</tr>
</tbody>
</table>

Forward R2 from the MEM stage for sub to the EX stage for or

Forward R2 from the EX stage for sub to the EX stage for and

From now on R2 is in the register file so no forwarding is needed.

**Forwarding paths**

Regular path from register file

Which sources are needed

Which result data does EX have

Which result data does MEM have

Forward from MEM stage

Forward from EX stage
What did forwarding do?

• If the result is ready but not in the register file:
  – We can forward it to the instruction that needs it
  – Look at the destinations of the data in the MEM and WB stages
  – If the instruction in EX needs that data as a source, we can forward it

• How did we do this?
  – Add more inputs to the ALU
  – Data comes from MEM and WB stages (data generated in EX and MEM)
  – Control logic looks at destination and source registers for each stage

Forwarding for loads and stores

Program Execution Time

Clock Cycle 1: IM
Clock Cycle 2: RF
Clock Cycle 3: PC
Clock Cycle 4: DM
Clock Cycle 5: RF
Clock Cycle 6: IM
Clock Cycle 7: RF
Clock Cycle 8: IM
Clock Cycle 9: RF

lw R2, 0(R4)
add R12, R2, R11

ALU needs R2 at the beginning of the clock cycle, but it isn’t available from the data memory until the end. Can not forward!
Can’t forward: need a delay for load/store

Even with forwarding we need a 1 cycle delay to get through the memory because it is in the next pipeline stage.

add R0, R0, R0 (nop)

add R12, R2, R11

Program Execution

Time

Clock Cycle 1

Clock Cycle 2

Clock Cycle 3

Clock Cycle 4

Clock Cycle 5

Clock Cycle 6

Clock Cycle 7

Clock Cycle 8

Clock Cycle 9

Even with forwarding we need a 1 cycle delay to get through the memory because it is in the next pipeline stage.

add R0, R0, R0 does nothing because we can’t write to R0. It’s a No-Operation, or NOP.

If the compiler can find other work to put here then we can use this slot for something useful.

Control hazards
Control hazards: branches

- Let’s look at the following code:
  - If branch is taken: 40, 44, 72, 76, ...
  - If branch is not taken: 40, 44, 48, 52, 56, 60, ...

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>add R30, R30, R30</td>
</tr>
<tr>
<td>44</td>
<td>beq R1, R3, 24</td>
</tr>
<tr>
<td>48</td>
<td>and R12, R2, R5</td>
</tr>
<tr>
<td>52</td>
<td>or R13, R6, R2</td>
</tr>
<tr>
<td>56</td>
<td>add R14, R2, R2</td>
</tr>
<tr>
<td>60</td>
<td>...</td>
</tr>
<tr>
<td>72</td>
<td>lw R4, S0(R7)</td>
</tr>
<tr>
<td>76</td>
<td>...</td>
</tr>
</tbody>
</table>

Branch decision from MEM used to choose the next instruction.

Branch hazard

Program Execution | Time | Clock Cycle 1 | Clock Cycle 2 | Clock Cycle 3 | Clock Cycle 4 | Determine the branch in cycle 4 | Clock Cycle 8 | Clock Cycle 9
--- | --- | --- | --- | --- | --- | --- | --- | ---
44 beq R1, R3, 24 | IM | RF | DM | RF | RF | | | |
48 and R12, R2, R6 | IM | RF | DM | RF | RF | | | |
52 or R13, R6, R2 | IM | RF | DM | RF | RF | | | |
56 add R14, R2, R2 | IM | RF | DM | RF | RF | | | |
60 or 72 (if R1 == R3) | IM | RF | DM | RF | RF | | | |

This is correct if the branch is not taken. Wrong if it is.

It takes 3 cycles to resolve the branch before we can go to the correct next PC.
Speeding up branches

- The problem is that it takes 3 cycles to resolve branches
- Can we improve on this by changing the pipeline or adding hardware?

1 cycle delay from ID
2 cycle delay from EX
3 cycle delay from MEM

3 cycle delay from MEM
2 cycle delay from EX
1 cycle delay from ID

But now we need some extra hardware...

Moving the branch computation earlier

Need an extra adder to calculate (PC + immediate)

1 cycle delay from ID
2 cycle delay from EX
3 cycle delay from MEM

Need a comparator to check equality
Branches with the earlier branch logic

<table>
<thead>
<tr>
<th>Program Execution</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IM</td>
</tr>
<tr>
<td></td>
<td>RF</td>
</tr>
<tr>
<td></td>
<td>DM</td>
</tr>
<tr>
<td>44</td>
<td>beq R1, R3, 24</td>
</tr>
<tr>
<td>48</td>
<td>and R12, R2, R6</td>
</tr>
<tr>
<td>72</td>
<td>lw R4, 50(R7)</td>
</tr>
</tbody>
</table>

This instruction is always executed. It is called the branch delay slot.

Now takes 1 cycle to resolve the branch before we can go to the correct next PC.

Branch delay slot

- We now have a 1 cycle delay for branches

- MIPS exposes this to the compiler:
  - The instruction after a branch always executes
  - Branch delay slot

- This breaks the ISA’s promise of serial execution

- What this means:
  - The compiler needs to find useful work to put in the branch delay slot
  - This work will be done regardless of whether the branch is taken
Structural hazards

Structural hazards: contention for resources

- Occur when multiple instructions need to use the same resource in the same cycle

- Let’s look at three examples:
  1. Register File read & write → solution: add more hardware
  2. Branch calculation
  3. Memory access
Example 1: Register File Read and Write

Fixing the register file (special case)

• The problem is that we want to **read** and **write** at the same time
• We can build a **double-pumped register file** that:
  – **Writes** in the 1\textsuperscript{st} half of the clock cycle
  – **Reads** in the 2\textsuperscript{nd} half of the clock cycle

---

**Example 1: Register File Read and Write**

<table>
<thead>
<tr>
<th>Program Execution</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clock Cycle 1</td>
</tr>
<tr>
<td>add R10, R11, R12</td>
<td>IM</td>
</tr>
<tr>
<td>add R17, R0, R0</td>
<td>IM</td>
</tr>
<tr>
<td>add R16, R0, R0</td>
<td>IM</td>
</tr>
<tr>
<td>sub R20, R21, R22</td>
<td>IM</td>
</tr>
<tr>
<td>add R30, R17, R18</td>
<td>IM</td>
</tr>
</tbody>
</table>

---

**Fixing the register file (special case)**

- The problem is that we want to **read** and **write** at the same time
- We can build a **double-pumped register file** that:
  - **Writes** in the 1\textsuperscript{st} half of the clock cycle
  - **Reads** in the 2\textsuperscript{nd} half of the clock cycle

---

**Example 1: Register File Read and Write**

<table>
<thead>
<tr>
<th>Program Execution</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clock Cycle 1</td>
</tr>
<tr>
<td>add R10, R11, R12</td>
<td>IM</td>
</tr>
<tr>
<td>add R17, R0, R0</td>
<td>IM</td>
</tr>
<tr>
<td>add R16, R0, R0</td>
<td>IM</td>
</tr>
<tr>
<td>sub R20, R21, R22</td>
<td>IM</td>
</tr>
<tr>
<td>add R30, R17, R18</td>
<td>IM</td>
</tr>
</tbody>
</table>

---

**Fixing the register file (special case)**

- The problem is that we want to **read** and **write** at the same time
- We can build a **double-pumped register file** that:
  - **Writes** in the 1\textsuperscript{st} half of the clock cycle
  - **Reads** in the 2\textsuperscript{nd} half of the clock cycle
Now this works

Example 2: branch calculation hazards

- For branches we need to do:
  - \((R1 - R2)\) to check if they are equal for the branch
  - \((PC + \text{immediate})\) to calculate the branch
- Need two ALUs at the same time!

Solve this structural hazard by adding more hardware: two adders instead of one.
Example 3: memory access hazard

- Two instructions access memory in cycle 4
  - We use separate instruction and data memories to support this
  - (Access instruction memory on every cycle!)

Example 3: memory access with one memory

Without separate memories for instructions and data we need to stall every time a load/store needs to access the memory.

sub and add do not load/store data from the memory so they do not use the memory in the MEM stage.
Performance impact of hazards

Performance and keeping the pipeline full

- We’ve seen how hazards can introduce empty slots in the pipeline
  - Bubbles or NOPs while waiting for results
  - Load/Store or Branch delay slots
  - Bubbles while waiting for resources

- These bubbles reduce the throughput of the pipeline
  - Bubbles are wasted cycles (no work)
  - Speedup of N for an N-stage pipeline assumes useful work on every cycle

- We’ve gone to extremes to avoid bubbles
  - Adding logic: register file, branch calculations, forwarding paths
  - Having compilers understand the details of the pipeline
How significant are these issues?

- **Load/Store delay slot**
  - lui: 3.3%, lw: 18.6%, lbu: 3.7%, sw: 7.6%, sb: 0.6%, **Total: 34% load/store**
  - 1 cycle delay on load stores:
    - 34%*2 + (100%-34%)*1 = 1.34 Cycles-Per-Instruction (CPI)
    - 34% slowdown due to load/store delay slots vs. perfect pipeline
      (If we can’t fill them with useful work)

- **Branch delay slot**
  - beq: 8.6%, bne: 8.4%, **Total: 17%**
  - 1 cycle delay on branches:
    - 17%*2 + (100%-17%)*1 = 1.17 Cycles-Per-Instruction (CPI)
    - 17% slowdown due to branch delay slots
      (If we can’t fill them with useful work)

- **Memory access**
  - **Total: 34% load/store**
  - Without separate Instruction and Data memories, we need another cycle delay:
    - 34% additional slowdown
      (And we can’t avoid it)

(Data from Figure 3.26 in the book.)

Pipeline idiosyncrasies

- **MIPS exposes the load/store and branch delay slot**
  - The compiler must take this into account
  - The architecture does not deal with it

- **Why?**
  - When MIPS was designed logic gates were expensive: easier to do in software
  - MIPS design philosophy: “Microprocessor without Interlocked Pipeline Stages” ⇒ hardware does not detect hazards and stall

- **Was this a good decision?**
  - **No.** Transistors are free. Better to have the hardware adapt than to have to re-compile code for every new pipeline version.
  - The **compiler has to know which version of MIPS** you are targeting and how many delay slots it needs or code will crash.
  - Or, **newer versions of MIPS have to pretend** they have the same pipeline or code will crash.