Pipeline Hazards

Introduction to Computer Architecture
David Black-Schaffer

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

Material that is not in this lecture

Readings from the book
- Detailed hazard logic

The book has excellent descriptions of this topic.
Please read the book before watching this lecture.
The reading assignment is on the website.

[Don't forget: the assigned reading may include details or bits and pieces that I don't cover in
the lecture. You're responsible for that as well on the exam.]

Introduction to hazards: the register file

Where we were with pipelining

- Good news
  - Multiple instructions in parallel in the pipeline
  - Each one is isolated by latches
  - Ideally n-stages → n-times speedup
- The bad news
  - Instructions interfere with each other
  - Conflicts or hazards
- Why?
  - Different instructions need the same resources (structural)
  - Different instructions need results from other instructions (data)
  - Different instructions execute depending on others (control)

Pipelining: the good
Pipelining: interference

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Instruction 3</th>
<th>Instruction 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>add R17, R0, R0</td>
<td>add R16, R0, R0</td>
<td>add R18, R17, R18</td>
<td>add R10, R11, R12</td>
</tr>
<tr>
<td>2</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
</tr>
<tr>
<td>3</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
</tr>
<tr>
<td>4</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
</tr>
<tr>
<td>5</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
</tr>
<tr>
<td>6</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
</tr>
<tr>
<td>7</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
</tr>
<tr>
<td>8</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
<td>IM RF Read ALU DM RF Write</td>
</tr>
</tbody>
</table>

Instructions interfering in the pipeline

- Problem: both instructions want the same resource
  - One wants to write
  - One wants to read
  - But we only have one register file
- How can we fix this?
  - Delay the reading instruction
    - Bad for performance
  - Never write conflicting instructions?
    - Bad for performance
  - Or we could cheat…

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 1st half of the clock cycle
  - Reads in the 2nd half of the clock cycle

Now this works

- We changed the register file to allow:
  - Writes in the 1st half of the clock cycle (clock high)
  - Reads in the 2nd half of the clock cycle (clock low)
- This solved the problem
- But there are other hazards…

What did we do?

- Need to read and write the register file:
  - At the same time
  - From different instructions
    - Due to pipelining and executing multiple instructions at the same time.
- To fix this we could:
  - Avoid bad combinations of instructions (impractical)
  - Put in delays between instructions (slow)
  - Or fix the register file to allow reads and writes together
- We changed the register file to allow:
  - Writes in the 1st half of the clock cycle (clock high)
  - Reads in the 2nd half of the clock cycle (clock low)
- This solved the problem
Data hazards

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 promised:
    - 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!

Handling data hazards: pipeline stalls

- To avoid this problem we will prevent instructions from overlapping in certain patterns

- How? Stall instructions when they need to wait for data
  - Stalled instruction doesn’t do anything
  - Think of this as a “bubble” in the pipeline: moves through with no effect

- How do we make a bubble?
  - Prevent writes to processor state
    - E.g., no writes to register file, memory, and program counter
  - If an instruction doesn’t write, it doesn’t do anything

We still have a problem...

What happened?

No stalls: hazards

- 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!
Detecting hazards

- Need hardware to detect hazards
  - Look at the instruction bits in the pipeline registers
  - Detect when instructions read data that is written in an earlier instruction
- How do we do this? Look at the data in the pipeline registers:

Example: detecting hazards

```
sub R2, R1, R3
and R12, R2, R5
or R13, R6, R2
add R14, R2, R2
sw R15, 100(R2)
```

- **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

```
sub R2, R1, R3
and R12, R2, R5
or R13, R6, R2
add R14, R2, R2
sw R15, 100(R2)
```

- **sub writes to the RF in the first half of the clock cycle**
- **and R12, R2, R5
  - WB/IM: sub-and R2**
- **or R13, R6, R2**
- **add R14, R2, R2**
- **sw R15, 100(R2)**
How do we stall?

- **Key insight:**
  - If an instruction doesn’t write its result, it is as if it never executed
  - A stalled instruction must not write to
    - Register File
    - Memory
    - Program Counter (keeps the same instruction being loaded)

- **Result:**
  - The instructions go through the pipeline, but they don’t make any changes
  - Because the PC is not being written, we keep loading the stalled instruction
  - We have converted the instruction to a bubble by disabling its writes

Detecting hazards early (so we can stall ID)

- Compare the register sources (Rs and Rt) of the instruction in the ID stage with:
  - Destination (Rd) of the instruction in the EX stage
  - Destination (Rd) of the instruction in the MEM stage

Stalling example

Compiler inserted stalls (NOPs)

Fixing the pipeline for data hazards

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

---

**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?**

---

**Forwarding paths**

---

**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**

---
Can’t forward: need a delay for load/store

Control hazards

Control hazards: branches

Branch hazard

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 (We saw this with reading and writing the register file at the same time)
Branch hazard: just stall?

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

Moving the branch computation earlier

- Need an extra adder to calculate (PC + immediate).
- Need a comparator to check equality.
- 1 cycle delay from ID.
- 2 cycle delay from EX.
- 3 cycle delay from MEM.

Branches with the earlier branch logic

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

Branch delay slot

- We now have a 1 cycle delay for branches (yipee!)
- 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

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?
Structural hazards

- Occur when multiple instructions need to use the same resource in the same cycle
  - Example: register read & write  solution: add more hardware

- Let’s look at two other examples:
  - 1. Branch calculation
  - 2. Memory access

Example 1: branch calculation hazards

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

Example 2: memory access hazard

- Two instructions access memory in cycle 4
  - (Access instruction memory on every cycle)

Example 2: memory access with one memory

- We use separate instruction and data memories to support this
Performance impact of hazards

How significant are these issues?

- Load/Store delay slot
  - 34% load/store
  - 3.5% b/c: 3.5% load, 3.5% store

- Branch delay slot
  - 5% b/c: 5% branch delay slots

- Memory access
  - 34%

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

20 years later...

Java-SPARC 2007-2008 Architecture Collaboration

- Sparc Acceleration – Efficient 64-bit constant formation
  - Reduce latency 6 50 cycle, consistently good results

- New-op "Squashing" – Code size reduction

Hazard summary

- 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 (forwarding and/or extra hardware)

- Some hazards cannot be fixed because we don’t have the results yet (memory access delay slots and branch delay slots)

- MIPS exposes the memory and branch delay slots, but this turned out to be a mistake

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

- 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

Remember:

- Programmers assume instructions finish completely in the order written!

- Transistors are free.
  - Beter to have the hardware adapt than to have to re-compile code for newer versions of MIPS

- MIPS design philosophy: "Microprocessor without Interlocked Pipeline Stages" ≡ hardware does not detect hazards and stalls

Was this a good decision?

- No. Transitions are free. Better to have the hardware adapt than to have to re-compile code for newer versions of MIPS

- 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.