Pipeline Hazards

CSE 410
Lecture 07
ISA vs. Implementation

- The ISA defines the operation of the machine as though it were a single cycle implementation
  - All previous instructions have completed before the next one starts

- Pipelining is a mechanism to speed up instruction completion rate
  - The semantics of the ISA have to be respected, though
  - That is, the effect of a pipelined execution of the instructions must be the same as what the ISA expect
5-Stage Pipeline
Hazards – Something goes wrong

add    x3, x2, x1
or     x6, x5, x4
and    x9, x8, x7
addi   x11, x10, 1
sub    x14, x13, x12
Instruction Sequence Processing

\[
\begin{align*}
\text{add} & \quad x3, x2, x1 \\
\text{or} & \quad x6, x5, x4 \\
\text{and} & \quad x9, x8, x7 \\
\text{addi} & \quad x11, x10, 1 \\
\text{sub} & \quad x14, x13, x12 \\
\end{align*}
\]
Data Hazard

- Later register reads a value written by an earlier instruction, but value not yet written to register
  - Instruction N reads register K
  - Instruction N-1 or N-2 or N-3 writes register K
  - According to the ISA, the write should occur before the read
    - Instruction N should get the value that written by the earlier instruction
  - If we’re not careful, in a pipelined implementation it won’t
Data Hazards

In pipelining, this is called a data hazard.

In general, this is called a data dependence, or a true dependence, or a read-after-write dependence.
Data Hazards at Distance 3

Solution
- design the register file so that if one of the read registers will be written this cycle, send the new value to the output pipeline register
- control gets more complicated
Data Hazards at Distance 2

Solution

- If not a lw instruction, “forwarding”
  - The value that will be written has already been produce and is in the ALU pipeline register
  - Feed it back to the rs1 or rs2 pipeline register and use it, rather than the value fetched from the register
  - control gets more complicated
Data Hazards at Distance 2

add x6, x5, x4

lw x5, 12(x3)

Solution
• If a lw instruction
  • the needed data won’t be available until the end of this cycle
  • We don’t want to make the cycle time longer to wait for it to be produced and then feed it back
  • Result: we put a wrong value into the rs1x or rs2x pipeline register
• BUT, the value is available when the add instruction moves to the ALU
Data Hazards at Distance 2

Solution

- ...
- BUT, the value is available when the add instruction moves to the ALU
  - control keeps getting more complicated!
**Data Hazards at Distance 1**

Solution
- If not a lw instruction, “forwarding”
  - Needed value isn’t available during when instructions are at these locations in pipeline
- But...

add \(x_6, x_5, x_4\)  \quad \text{add} \; x_5, x_4, x_3
Data Hazards at Distance 1

Solution

- If not a lw instruction, “forwarding”
  - Needed value is in ALUm pipeline register at start of cycle when it is needed
  - (After this, I’m going to stop repeating that “control gets more complicated”)

add x6, x5, x4  
add x5, x4, x3
Data Hazards at Distance 1

add $x6$, $x5$, $x4$  

$\text{lw } x5, 12(x3)$

Solution

- If a lw instruction, need value isn’t available at this stage of processing
Data Hazards at Distance 1

Solution

• It’s also not available at (the beginning of) this stage
Data Hazards at Distance 1

Solution
- At this stage, it’s too late
  - Instruction has used ALU but with incorrect input values
  - It won’t have another chance to use the ALU

```
add x6, x5, x4  lw x5, 12(x3)
```
Data Hazards at Distance 1

Solution

• If there is “a bubble” between the two instructions, then we can resolve the dependence by forwarding (as it’s now distance two)
• Note: the bubble uses the pipeline but doesn’t do anything useful
  • Bubbles slow down the pipeline
Where Do Bubbles Come From?

- In early RISC architectures, it was the programmer’s responsibility to explicitly code them
  - It was a programmer error to try to use the value produced by a lw instruction in the immediately following instruction
  - You had to explicitly code a nop instruction
    - The hardware didn’t detect it if you didn’t, you just got wrong results
  - Of course, “the programmer” is a compiler, so it’s not such a big deal to have to insert NOPs where needed

- In RISC-V, things have advanced and building complicated control isn’t such a big deal
  - The hardware inserts the NOP whenever needed
Data Hazards at Distance 1

\[
\text{lw \ x5, 12(x3)}
\]

**Bubble Solution**
- When control fetches the lw instruction, it remembers that in the inst_D pipeline register
Data Hazards at Distance 1

Bubble Solution

- When control fetches the next instruction, it detects that it reads a register that is written by the previous instruction, and that the previous instruction was a `lw`
- So, it
  - injects a `nop` into the `inst_D` pipeline register, rather than the add instruction
  - disables updating of the PC
    - (Yes, control keeps getting more complicated)
Data Hazards at Distance 1

Bubble Solution
- Because the PC wasn’t updated last cycle, this next cycle the same add instruction is fetched
- There is no dependence between the add and the nop, so control “dispatches” the add instruction
Control Hazards

- Control hazards result because we don’t know whether or not a branch will be taken for two cycles after they’re fetched
  - What instruction should we fetch in the cycle after we fetch the branch?
    - The next sequential instruction (as if the branch weren’t taken)
    - The instruction at the target address (as if the branch were taken)
  - Note that the branch needs to get to the ALU stage to compute the target address, so maybe we have only one choice?
- Control hazards are data hazards on the PC
Control Hazards

At this stage we can resolve whether the branch is taken or not, and what the target address is. But, what two instructions have been fetched after the branch?

```
beq x4, x5, loop
add x5, x4, x3
add x8, x7, x6
...?
```

```
beq x4, x5, -20
```
Control Hazards – Pessimistic

Control notices branch and inserts two nops.

beq x4, x5, loop
add x5, x4, x3
add x8, x7, x6
...

nop
nop
beq x4, x5, -20
Control Hazards – Speculative

Control fetches instructions consecutively from memory.

```
beq x4, x5, loop
add x5, x4, x3
add x8, x7, x6
...
```

If at this stage branch is taken, turn trailing instructions to nops (in pipeline registers).

If branch isn’t taken, great!
Real Machines – Branch Prediction

- Control maintains a table something like this:

<table>
<thead>
<tr>
<th>PC</th>
<th>Next address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40180</td>
<td>0x442C0</td>
</tr>
<tr>
<td>0x3822C</td>
<td>0x38210</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Each row corresponds to a branch instruction in memory
- The PC column identifies which branch it is
- The “next address” column is a prediction
  - For instance, when a branch is taken, put update table with branch target address; when not taken, update table with next sequential address.

- Real machines use much more sophisticated schemes
Pipelining Summary

- Pipelining is one approach to parallelism
  - Parallelism is the key to “going faster”
- The cycle time in a single cycle implementation has to be the worst case delay through the entire data path
- The cycle time in an N stage pipeline has to be the worst case delay through any one stage
  - So, ideally can get up to an N-fold increase in speed
- So, why not make a 300-stage pipeline?
Limits to Pipelining

- Because of imbalances in the stages and overheads writing pipeline registers, when you double the number of stages you probably don’t have the cycle time.

- Hazards result in bubbles
  - The more stages, the larger the number of bubbles needed
  - The benefits of more stages are limited by the bubbles

- More stages require more pipeline registers and more control, and those take area and power.
Mild Lessons for Software

- **Hardware**
  - Actual performance is much more complicated than just instruction count
  - Actual performance is much more complicated than clock rate

- **Software**
  - Long sequences of consecutive instructions go fastest, so...
  - Try to avoid branching!
    - Some processors have had “conditional instructions” – they were nops (no operation) unless the last comparison instruction evaluated to true
    - Why?