Poll Q: How many stalls?

**type (no enter) into Zoom chat**

- Suppose EX is the longest (in time) pipeline stage
- To reduce CT, we split it in half. Given the following (new) pipeline:

  ```
  IF ID EX1 EX2 M WB
  ```

  Assume the input data must be available at the start of EX1 and the output is available after EX2

- **How many hardware stalls** would be required in the following code (assuming hardware forwarding wherever possible)?

  ```
  add r1, r2, r3
  add r4, r1, r3
  ```
Poll Q: How many stalls?

*type* *(no enter)* into Zoom chat

- Suppose EX is the longest (in time) pipeline stage
- To reduce CT, we split it in half. Given the following (new) pipeline:

  IF ID EX1 EX2 M WB

  Assume the input data must be available at the start of EX1 and the output is available after EX2
- **How many hardware stalls** would be required in the following code (assuming hardware forwarding wherever possible)?

  ```
lw r1, 0(r3)
add r2, r1, r3
```
if (ID/EX.MemRead and
((ID/EX.RegisterRt = IF/ID.RegisterRs) or
(ID/EX.RegisterRt = IF/ID.RegisterRt)))
then stall the pipeline
Hazard Detection

and $4, $2, $5

lw $2, 20($1)
Hazard Detection

\[ \text{and } \$4, \$2, \$5 \quad \text{nop (bubble)} \quad \text{lw } \$2, 20(\$1) \]
What other hazards might we have to watch out for?

• Data hazards are when the result of one computation is used in a later computation
• Is there other re-use?
Control Dependence

- Just as an instruction will be dependent on other instructions to provide its operands (**data dependence**), it will also be dependent on other instructions to determine whether it gets executed or not (**control dependence**, aka, **branch dependence**).
- Control dependences are particularly critical with **conditional branches**.

```
add $5, $3, $2
sub $6, $5, $2
beq $6, $7, somewhere
and $9, $6, $1
... somewhere: or $10, $5, $2
add $12, $11, $9
... bez $6, $7, somewhere
and $9, $6, $1
...```

...
Branch Hazards

- Branch dependences can result in branch hazards (when they are too close to be handled correctly in the pipeline)
  - (sound familiar?)
Given our current pipeline, let’s assume we stall until we know the branch outcome (i.e., until the PC is known to be correct).

How many cycles will we lose per branch?

<table>
<thead>
<tr>
<th></th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
</tr>
</tbody>
</table>
Branch Hazards

beq $2, $1, here

add ...

sub ...

lw ...

here: lw ...
Dealing With Branch Hazards

- Ideas??
Dealing With Branch Hazards

- **Hardware**
  - *stall until you know which direction*
  - reduce hazard through earlier computation of branch direction
  - guess which direction
    - assume not taken (easiest)
    - more educated guess based on history
      - (requires that you know it is a branch before it is even decoded!)
Dealing With Branch Hazards

• Hardware
  – stall until you know which direction
  – reduce hazard through earlier computation of branch direction
  – guess which direction
    • assume not taken (easiest)
    • more educated guess based on history
      – (requires that you know it is a branch before it is even decoded!)

• Hardware/Software
  – nops
  – instructions that get executed either way (delayed branch).
Stalling for Branch Hazards

beq $4, $0, there
and $12, $2, $5
or ...
add ...
sw ...
Stalling for Branch Hazards

• Seems wasteful, particularly when the branch isn’t taken.
• Makes all branches cost 4 cycles.
Assume Branch Not Taken

- works pretty well when you’re right!

\[
\text{beq } $4, $0, \text{there and } $12, $2, $5 \quad \text{or \ ...}
\]

\[
\text{add \ ...}
\]

\[
\text{sw \ ...}
\]
Assume Branch Not Taken

- *same performance as stalling* when you’re wrong

```
beq $4, $0, there
and $12, $2, $5
or ...
add ...
there: sub $12, $4, $2
```
Assume Branch Not Taken

• Performance depends on percentage of time you guess right
• Flushing an instruction means to prevent it from changing any permanent state (registers, memory, PC)
  – sounds a lot like a bubble...
  – But notice that we need to be able to insert those bubbles later in the pipeline
Branch Hazards – What if we predict taken instead?

**Required** information to predict Taken:

1. Whether an instruction is a branch (before decode)
2. The target of the branch
3. The outcome of the branch condition

<table>
<thead>
<tr>
<th>Required knowledge</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2, 3</td>
<td>1, 2, 3</td>
<td>1, 2</td>
<td>2</td>
<td>None of the above</td>
</tr>
</tbody>
</table>
Branch Target Buffer

*aka, how to know it’s a branch before you know it’s a branch*

- Keeps track of the PCs of recently seen branches and their targets.
- Consult during Fetch (in parallel with Instruction Memory read) to determine:
  - Is this a branch?
  - If so, what is the target
Reducing the Branch Delay
Reducing the Branch Delay

• can easily get to 2-cycle stall
Stalling for Branch Hazards

beq $4, $0, there
and $12, $2, $5
or ...
add ...
sw ...

CC1 IM → CC2 Reg → CC3 DM → CC4 Reg

CC5 IM → CC6 Reg → CC7 DM → CC8 Reg

CC1 IM → Bubble → CC3 DM

CC5 IM → Bubble → CC7 DM
Reducing the Branch Delay

• Harder, but possible, to get to 1-cycle stall
Stalling for Branch Hazards

beq $4, $0, there
and $12, $2, $5
or ...
add ...
sw ...

IM  Reg  DM  Reg
CC1  CC2  CC3  CC4  CC5  CC6  CC7  CC8

Bubble
The Pipeline with flushing for taken branches

- Notice the IF/ID flush line added.
Eliminating the Branch Stall

A cute idea, but not one used by any modern core

• There’s no rule that says we have to see the effect of the branch immediately. Why not wait an extra instruction before branching?
• The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches.
• The instruction after a conditional branch is always executed in those machines, regardless of whether the branch is taken or not!