Lecture 12

- HW2 due today at 5pm!
- Today’s lecture:
  - What about load followed by use?
  - What about branches?
  - Crystal ball

Pick up handout!!
Stalls and flushes

- So far, we have discussed data hazards that can occur in pipelined CPUs if some instructions depend upon others that are still executing.
  - Many hazards can be resolved by forwarding data from the pipeline registers, instead of waiting for the writeback stage.
  - The pipeline continues running at full speed, with one instruction beginning on every clock cycle.
- Now, we'll see some real limitations of pipelining.
  - Forwarding may not work for data hazards from load instructions.
  - Branches affect the instruction fetch for the next clock cycle.
- In both of these cases we may need to slow down, or stall, the pipeline.
Data hazard review

- A data hazard arises if one instruction needs data that isn’t ready yet.
  - Below, the AND and OR both need to read register $2$.
  - But $2$ isn’t updated by SUB until the fifth clock cycle.
- Dependency arrows that point backwards indicate hazards.
Forwarding

- The desired value ($1 - $3) has actually already been computed—it just hasn’t been written to the registers yet.
- **Forwarding** allows other instructions to read ALU results directly from the pipeline registers, without going through the register file.
What about loads?

- Imagine if the first instruction in the example was LW instead of SUB.  
  - How does this change the data hazard?
What about loads?

- Imagine if the first instruction in the example was LW instead of SUB.
  - The load data doesn't come from memory until the end of cycle 4.
  - But the AND needs that value at the beginning of the same cycle!
- This is a "true" data hazard—the data is not available when we need it.
Stalling

- The easiest solution is to **stall** the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a **bubble**.

```
<table>
<thead>
<tr>
<th>lw</th>
<th>$2, 20($3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td>Reg</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reg</td>
</tr>
<tr>
<td></td>
<td>DM</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
</tr>
<tr>
<td>and</td>
<td>$12, $2, $5</td>
</tr>
<tr>
<td>IM</td>
<td>Reg</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reg</td>
</tr>
<tr>
<td></td>
<td>DM</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
</tr>
</tbody>
</table>
```

- Do we still need forwarding? **Yes**
Stalling and forwarding

- Without forwarding, we’d have to stall for two cycles to wait for the LW instruction’s writeback stage.

- In general, you can always stall to avoid hazards—but dependencies are very common in real code, and stalling often can reduce performance by a significant amount.
Stalling delays the entire pipeline

- If we delay the second instruction, we’ll have to delay the third one too.
  - Why?

**Static scheduling**

```plaintext
lw $2, 20($3)  
and $12, $2, $5  
or $13, $12, $2
```

Clock cycle

1 2 3 4 5 6 7 8
Stalling delays the entire pipeline

- If we delay the second instruction, we’ll have to delay the third one too.
  - This is necessary to make forwarding work between AND and OR.
  - It also prevents problems such as two instructions trying to write to the same register in the same cycle.
Implementing stalls

- One way to implement a stall is to force the two instructions after LW to pause and remain in their ID and IF stages for one extra cycle.

```
Clock cycle

1  2  3  4  5  6  7  8

lw  $2, 20($3)

and $12, $2, $5

or $13, $12, $2
```

- This is easily accomplished.
  - Don’t update the PC, so the current IF stage is repeated.
  - Don’t update the IF/ID register, so the ID stage is also repeated.
What about EXE, MEM, WB

- But what about the ALU during cycle 4, the data memory in cycle 5, and the register file write in cycle 6?

<table>
<thead>
<tr>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>lw $2, 20($3)</td>
</tr>
</tbody>
</table>

- and $12, $2, $5

- or $13, $12, $2

- Those units aren’t used in those cycles because of the stall, so we can set the EX, MEM and WB control signals to all 0s.
Stall = Nop conversion

- The effect of a load stall is to insert an empty or nop instruction into the pipeline.

lw $2, 20($3) -> nop

and $12, $2, $5

or $13, $12, $2
Detecting stalls

- Detecting stall is much like detecting data hazards.

- Recall the format of hazard detection equations:

  \[
  \text{if } (\text{EX/MEM.RegWrite} = 1 \\
  \text{and } \text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs}) \\
  \text{then Bypass Rs from EX/MEM stage latch}
  \]

  sub $2$, $1$, $3$

  and $12$, $2$, $5$
Detecting Stalls, cont.

- When should stalls be detected?

```
lw $2, 20($3)
```

and $12, $2, $5

- What is the stall condition?

```java
if ( ID/EX.MEMREAD == 1 &&
    ID/EX.AT == IF.ID.AT )
then stall
```
Detecting stalls

- We can detect a load hazard between the current instruction in its ID stage and the previous instruction in the EX stage just like we detected data hazards.
- A hazard occurs if the previous instruction was LW...

\[ \text{ID/EX. MemRead} = 1 \]

...and the LW destination is one of the current source registers.

\[ \text{ID/EX. RegisterRt} = \text{IF/ID. RegisterRs} \]
\[ \text{or} \]
\[ \text{ID/EX. RegisterRt} = \text{IF/ID. RegisterRt} \]

- The complete test for stalling is the conjunction of these two conditions.

\[
\text{if (ID/EX. MemRead} = 1 \text{ and }
\text{(ID/EX. RegisterRt} = \text{IF/ID. RegisterRs} \text{ or }
\text{ID/EX. RegisterRt} = \text{IF/ID. RegisterRt}))
\]
\[
\text{then stall}
\]
Adding hazard detection to the CPU
Adding hazard detection to the CPU
The hazard detection unit

- The hazard detection unit's inputs are as follows.
  - IF/ID.RegisterRs and IF/ID.RegisterRt, the source registers for the current instruction.
  - ID/EX.MemRead and ID/EX.RegisterRt, to determine if the previous instruction is LW and, if so, which register it will write to.
- By inspecting these values, the detection unit generates three outputs.
  - Two new control signals PCWrite and IF/ID Write, which determine whether the pipeline stalls or continues.
  - A mux select for a new multiplexer, which forces control signals for the current EX and future MEM/WB stages to 0 in case of a stall.
Generalizing Forwarding/Stalling

- What if data memory access was so slow, we wanted to pipeline it over 2 cycles?

- How many bypass inputs would the muxes in EXE have?
- Which instructions in the following require stalling and/or bypassing?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>r13, 0(r11)</td>
</tr>
<tr>
<td>add</td>
<td>r7, r8, r9</td>
</tr>
<tr>
<td>add</td>
<td>r15, r7, r13</td>
</tr>
</tbody>
</table>
Branches in the original pipelined datapath

When are they resolved?
Branches

- Most of the work for a branch computation is done in the EX stage.
  - The branch target address is computed.
  - The source registers are compared by the ALU, and the Zero flag is set or cleared accordingly.
- Thus, the branch decision cannot be made until the end of the EX stage.
  - But we need to know which instruction to fetch next, in order to keep the pipeline running!
  - This leads to what's called a control hazard.

```
beq $2, $3, Label
???
```

Clock cycle

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td>Reg</td>
<td></td>
<td>OM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stalling is one solution

- Again, stalling is always one possible solution.

- Here we just stall until cycle 4, after we do make the branch decision.
Branch prediction

- Another approach is to guess whether or not the branch is taken.
  - In terms of hardware, it’s easier to assume the branch is not taken.
  - This way we just increment the PC and continue execution, as for normal instructions.
- If we’re correct, then there is no problem and the pipeline keeps going at full speed.
Branch misprediction

- If our guess is wrong, then we would have already started executing two instructions incorrectly. We'll have to discard, or **flush**, those instructions and begin executing the right ones from the branch target address, Label.

```
beq $2, $3, Label
next instruction 1
next instruction 2
Label: ...
```
Performance gains and losses

- Overall, branch prediction is worth it.
  - Mispredicting a branch means that two clock cycles are wasted.
  - But if our predictions are even just occasionally correct, then this is preferable to stalling and wasting two cycles for every branch.

- All modern CPUs use branch prediction.
  - Accurate predictions are important for optimal performance.
  - Most CPUs predict branches dynamically—statistics are kept at run-time to determine the likelihood of a branch being taken.

- The pipeline structure also has a big impact on branch prediction.
  - A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.
  - We must also be careful that instructions do not modify registers or memory before they get flushed.
Implementing branches

- We can actually decide the branch a little earlier, in ID instead of EX.
  - Our sample instruction set has only a BEQ.
  - We can add a small comparison circuit to the ID stage, after the source registers are read.
- Then we would only need to flush one instruction on a misprediction.
Implementing flushes

- We must flush one instruction (in its IF stage) if the previous instruction is BEQ and its two source registers are equal.
- We can flush an instruction from the IF stage by replacing it in the IF/ID pipeline register with a harmless nop instruction.
  - MIPS uses sll $0, $0, 0 as the nop instruction.
  - This happens to have a binary encoding of all 0s: 0000 .... 0000.
- Flushing introduces a bubble into the pipeline, which represents the one-cycle delay in taking the branch.
- The IF.Flush control signal shown on the next page implements this idea, but no details are shown in the diagram.
Branching without forwarding and load stalls
Timing

- If no prediction:

  IF  ID  EX  MEM  WB
      IF  IF  ID  EX  MEM  WB  --- lost 1 cycle

- If prediction:
  - If Correct
    IF  ID  EX  MEM  WB
    IF  IF  ID  EX  MEM  WB  -- no cycle lost
  - If Misprediction:
    IF  ID  EX  MEM  WB
    IF0  IF1  ID  EX  MEM  WB  --- 1 cycle lost
Summary

- Three kinds of hazards conspire to make pipelining difficult.
- **Structural hazards** result from not having enough hardware available to execute multiple instructions simultaneously.
  - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline stages.
- **Data hazards** can occur when instructions need to access registers that haven’t been updated yet.
  - Hazards from R-type instructions can be avoided with forwarding.
  - Loads can result in a “true” hazard, which must stall the pipeline.
- **Control hazards** arise when the CPU cannot determine which instruction to fetch next.
  - We can minimize delays by doing branch tests earlier in the pipeline.
  - We can also take a chance and predict the branch direction, to make the most of a bad situation.