Lecture 13

- Today's lecture:
  - What about branches?
  - Crystal ball
  - Look at performance again

-Midterm practice questions posted!
HW3 to be posted Fri/Mon
Stall = Nop conversion

Data Hazard

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

and $12, $2, $5

or $13, $12, $2

- The effect of a load stall is to insert an empty or nop instruction into the pipeline
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>Memory Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw r13, 0(r11)</td>
<td>M1, M2, LW</td>
</tr>
<tr>
<td>add r7, r8, r9</td>
<td>M1</td>
</tr>
<tr>
<td>add r15, r7, r13</td>
<td>M1, M2, LW</td>
</tr>
</tbody>
</table>

Clock cycle

1 2 3 4 5 6
IM I D Ex M1 M2 Reg

FORWARD 0C
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.
- Hmm, what is the problem? What do we do to solve?
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**.

![Diagram of branch pipeline stages]

```
beq $2, $3, Label
```
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.

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

Clock cycle 1 2 3 4 5 6 7

next instruction 1

next instruction 2
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.
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.
  - How?
  - 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 ID EX MEM WB --- no cycle lost
  - If Misprediction:
    IF ID EX MEM WB
    IF0 IF1 ID EX MEM WB --- 1 cycle lost

2 cycles
Summary of Pipeline Hazards

- 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.