Lecture 13

- Homework 3 due today.
  - I hope you had fun with recursion and stack smashing.
- Poll: Monday, May 3. Normal lecture or review?

Today's lecture:
- Quick review
- What about load followed by use?
- What about branches?
- Crystal ball

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 $s2$.
  - But $s2$ isn't updated by SUB until the fifth clock cycle.
- Dependency arrows that point backwords indicate hazards.

![Clock cycle diagram]

sub $s2$, $s1$, $s3$
and $s12$, $s2$, $s5$
or $s13$, $s6$, $s2$

Example

![Example diagram]

- Assume again each register initially contains its number plus 100.
  - After the first instruction, $s2$ should contain -2 (101 - 103).
  - The other instructions should all use -2 as one of their operands.
- We'll try to keep the example short.
  - Assume no forwarding is needed except for register $s2$.
  - We'll skip the first two cycles, since they're the same as before.
Clock cycle 5: forwarding $s2$ from MEM/WB

What about stores?

- Two "easy" cases:
  
  \[
  \text{add } s1, s2, s3 \\
  \text{sw } s4, 0(s1)
  \]

Store Bypassing: Version 1

Store Bypassing: Version 2

What about stores?

- A harder case:
  
  \[
  \text{lw } s1, 0(s2) \\
  \text{sw } s1, 0(s4)
  \]

- In what cycle is:
  - The load value available?
  - The store value needed?

- What do we have to add to the datapath?
Load/Store Bypassing: Extend the Datapath

What about loads?
- Imagine if the first instruction in the example was LW instead of SUB.
  - Would the AND instruction be able to be processed in cycle 3?
  - How does this impact the data hazard?

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:
  - If a data hazard occurs, the pipeline flushes all instructions in cycles 3 and 4.
  - To resolve data hazards, all instructions in cycles 3 and 4 flush.

Stalling
- The easiest solution is to stall the pipeline.
- Without forwarding, the AND instruction would have to wait for the LW instruction’s writeback stage.

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

- Notice that we’re still using forwarding in cycle 5, to get data from the WB/WB pipeline register to the ALU.
Stalling delays the entire pipeline

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

```
1 2 3 4 5 6 7 8
lw $2, 20($3)
and $12, $1, $5
or $13, $12, $2
```

```
1 2 3 4 5 6 7 8
lw $2, 20($3)
and $12, $1, $5
or $13, $12, $2
```

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.

```
1 2 3 4 5 6 7 8
lw $1, 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.

```
1 2 3 4 5 6 7 8
lw $1, 20($3)
and $12, $2, $5
or $13, $12, $2
```

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?

```
1 2 3 4 5 6 7 8
lw $2, 20($3)
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.

```
1 2 3 4 5 6 7 8
lw $2, 20($3)
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:
  
  ```
  if (EX/ALU.Register# ≠ 1) 
  and (EX/ALU.Register# ≠ 1) 
  and (EX/Register# ≠ 1) 
  then Bypass Rs from EX/MEM stage latch
  ```

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

```
1 2 3 4 5 6 7 8
lw $1, 20($3)
and $2, $0, $5
or $13, $12, $2
```

```
1 2 3 4 5 6 7 8
lw $1, 20($3)
and $2, $0, $5
or $13, $12, $2
```
Detecting Stalls, cont.

- When should stalls be detected?
  
  ```
  lw $2, 20($3)
  and $12, $2, $3
  ```

- What is the stall condition?
  
  ```
  if (3/D/EX.EmemRead || (3/D/EX.EmemRead || if (I/D.EmemWrite) & (I/D.EmemWrite))
  ```
  then stall

**Detecting stalls**

- We can detect a load hazard between the current instruction in the 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...

  ```
  I/D.EmemWrite || I/D.EmemWrite
  ```

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

  ```
  I/D.EmemWrite || I/D.EmemWrite
  ```

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

  ```
  if (I/D.EmemWrite || I/D.EmemWrite)
  ```

  then stall

Adding hazard detection to the CPU

The hazard detection unit

- The hazard detection unit's inputs are as follows:
  - IF/ID.EmemWrite and I/D.EmemWrite, the source registers for the current instruction.
  - I/D.EmemWrite and I/D.EmemWrite, to determine if the previous instruction is LW and, if so, which register it will write into.
  - By expecting these values, the detection unit generates three outputs.
  - Two new control signals, PCWrite and I/D.EmemWrite, which determine whether the pipeline stalls or continues.
  - A new 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 EIN have?

- Which instructions in the following require stalling and/or bypassing?

```
1: lw $r13, ($15)
add $r15, $r7, $r13
```
Branches in the original pipelined datapath

Stalling is one solution

- Again, stalling is always one possible solution.

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

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

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
```

Performance gains and losses

- Overall, branch prediction is worth it.
  - Underpredicting 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 runtime 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.

![Branching without forwarding and load stalls diagram]

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 ILI 50, 50, 0 as the nop instruction.
- This happens to have a binary encoding of all Os: 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.

Timing

- If no prediction:
  IF ID EX MEB WB
  IF IF ID EX MEB WB --- lost 1 cycle

- If prediction:
  - If Correct
    IF ID EX MEB WB
  - If misprediction:
    IF ID EX MEB WB
  IF IF IP1 ID EX MEB 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 memory) 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.
- 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.