Lecture 12

Today’s topics:

- More pipelining...

Our examples are too simple

- Here is the example instruction sequence used to illustrate pipelining:

```
li $r1, 4($t0)
sub $s2, $t1, $s2
and $s3, $t1, $s2
or $s4, $t1, $s2
add $s5, $t1, $s2
```

- The instructions in this example are independent.
- Each instruction fetch and write fully different registers.
- Our datapath handles this sequence easily, as we saw last time.
- Is this the case for most sequences?

Data hazards in the pipeline diagram

- The SUB instruction does not write to register $s2 until cycle 5. This causes two data hazards in our current pipelined datapath.
  - The AND reads register $s2 in cycle 3. Since SUB hasn’t modified the register yet, this will be the old value of $s2, not the new one.
  - Similarly, the OR instruction uses register $s2 in cycle 4, again before it’s actually updated by SUB.

Pipeline diagram review

- This diagram shows the execution of an ideal code fragment.
  - Each instruction needs a total of five cycles for execution.
  - One instruction begins on every clock cycle for the first five cycles.
  - One instruction completes on each cycle from that time on.

An example with dependencies

```
sub $s2, $s1, $s3
and $s4, $s2, $s5
or $s6, $s2, $s2
add $s7, $s2, $s2
sw $s8, 100($s2)
```

- Is this a problem for the single-cycle and multicyle datapaths? Why?
- How would this code sequence fare in our pipelined datapath?

Things that are okay

- The ADD instruction is okay, because of the register file design.
  - Registers are written at the beginning of a clock cycle.
- The new value will be available by the end of that cycle.
- The SW is no problem at all, since it reads $s2 after the SUB finishes.
Dependency arrows

- Arrows indicate the flow of data between instructions.
  - The tails of the arrows show when register $S_2$ is written.
  - The heads of the arrows show when $S_2$ is read.
- Any arrow that points backward in time represents a data hazard in our basic pipelined datapath. Here, hazards exist between instructions 1 & 2 and 1 & 3.

A more detailed look at the pipeline

- We have to eliminate the hazards, so the AND and OR instructions in our example will use the correct value for register $S_2$.
- When is the data actually produced and consumed?
- What can we do?

Bypassing the register file

- The actual result $S_1 - S_3$ is computed in clock cycle 3, before it’s needed in cycles 4 and 5.
- If we could somehow bypass the writeback and register read stages when needed, then we can eliminate these data hazards.
- Today we’ll focus on hazards involving arithmetic instructions.
- Next, we’ll examine the lw instruction.
- Essentially, we need to pass the ALU output from SUB directly to the AND and OR instructions, without going through the register file.

A fancier pipeline diagram

- We have to eliminate the hazards, so the AND and OR instructions in our example will use the correct value for register $S_2$.
- Let’s look at when the data is actually produced and consumed.
  - The SUB instruction produces its result in its EX stage, during cycle 3 in the diagram below.
  - The AND and OR need the new value of $S_2$ in their EX stages, during clock cycles 4-5 here.

Where to find the ALU result

- The ALU result generated in the EX stage is normally passed through the pipeline registers to the MEM and WB stages, before it is finally written to the register file.
- This is an abridged diagram of our pipelined datapath.
### Forwarding
- Since the pipeline registers already contain the ALU result, we could just forward that value to subsequent instructions, to prevent data hazards.
  - In clock cycle 4, the AND instruction can get the value $s1 - s3$ from the EX/MEM pipeline register used by sub.
  - Then in cycle 5, the OR can get that same result from the ID/EX pipeline register being used by SUB.

### Outline of forwarding hardware
- A forwarding unit selects the correct ALU inputs for the EX stage.
  - If there is no hazard, the ALU’s operands will come from the register file, just like before.
  - If there is a hazard, the operands will come from either the EX/MEM or ID/EX/MEM pipeline registers instead.
- The ALU sources will be selected by two new multiplexers, with control signals named ForwardA and ForwardB.

### Detecting EX/MEM data hazards
- So how can the hardware determine if a hazard exists?
  \[
  (\text{old} \cdot E_D \neq \text{new}, \text{new} = \text{old} \cdot \text{emem})
  \]
- If there is a hazard, the next instruction will write to the register file.

### Simplified datapath with forwarding muxes

### Detecting EX/MEM data hazards
- Data in a pipeline register can be referenced using a class-like syntax. For example, \text{ID/EX.\ RegisterRi} refers to the ri field stored in the ID/EX pipeline.

### EX/MEM data hazard equations
- The first ALU source comes from the pipeline register when necessary.
  \[
  \begin{align*}
  \text{if} & \ (\text{EX/MEM.\ RegisterWr} = 1) \\
  \text{and} & \ (\text{EX/MEM.\ RegisterRd} = \text{ID/EX.\ RegisterRi}) \\
  \text{then} & \ \text{ForwardA} = 1
  \end{align*}
  \]
- The second ALU source is similar.
  \[
  \begin{align*}
  \text{if} & \ (\text{EX/MEM.\ RegisterWr} = 1) \\
  \text{and} & \ (\text{EX/MEM.\ RegisterRd} = \text{ID/EX.\ RegisterRi}) \\
  \text{then} & \ \text{ForwardB} = 2
  \end{align*}
  \]
Detecting MEM/WB data hazards

- A MEM/WB hazard may occur between an instruction in the EX stage and the instruction from two cycles ago.
- One new problem is if a register is updated twice in a row.
  - `add $1, $2, $3`
  - `add $1, $3, $4`
  - `sub $5, $5, $1`
- Register $1$ is written by both of the previous instructions, but only the most recent result (from the second `ADD`) should be forwarded.

MEM/WB hazard equations

- Here is an equation for detecting and handling MEM/WB hazards for the first ALU source.
  - `if (MEM/WB, RegWrite = 1`
  - `and (MEM/WB, RegisterRd = ID/EX, RegisterRt)`
  - `and (EX/ID, RegisterRd = ID/EX, RegisterRt)`
  - `or (EX/ID, MEM/WB, RegWrite = 0)`
  - then ForwardRt = 1
- The second ALU operand is handled similarly.
  - `if (MEM/WB, RegWrite = 1`
  - `and (MEM/WB, RegisterRd = ID/EX, RegisterRt)`
  - `and (EX/ID, RegisterRd = ID/EX, RegisterRt)`
  - `or (EX/ID, MEM/WB, RegWrite = 0)`
  - then ForwardRt = 1

Simplified datapath with forwarding

The forwarding unit

- The forwarding unit has several control signals as inputs:
  - `ID/EX, RegisterRd`  `Ex/ID, RegisterRd`  `MEM/WB, RegisterRd`  `ID/EX, RegisterRt`  `Ex/ID, RegisterRt`  `MEM/WB, RegWrite`
  (The two RegWrite signals are not shown in the diagram, but they come from the control unit.)
- The forwarding unit outputs are selectors for the ForwardRt and ForwardRt multiplexers attached to the ALU. These outputs are generated from the inputs using the equations on the previous page.
- Some new buses route data from pipeline registers to the new muxes.

Example

- `sub $1, $2, $3`
- Assume again each register initially contains its number plus 100.
  - After the first instruction, $3$ should contain $201$.
  - The other instructions should all use $202$ as one of their operands.
- We'll try to keep the example short.
  - Assume no forwarding is needed except for register $3$.
  - We'll skip the first two cycles, since they're the same as before.

Clock cycle 3
Lots of data hazards

- The first data hazard occurs during cycle 4.
  - The forwarding unit notices that the ALU’s first source register for the AND is also the destination of the SUB instruction.
  - The correct value is forwarded from the EX/MEM register, overriding the incorrect old value still in the register file.
- A second hazard occurs during cycle 5.
  - The ALU’s second source (for OR) is the SUB destination again.
  - This time, the value has to be forwarded from the MEM/WB pipeline register instead.
- There are no other hazards involving the SUB instruction.
  - During cycle 5, SUB writes its result back into register $2$.
  - The ADD instruction can read this new value from the register file in the same cycle.

What about stores?

- Two “easy” cases:
  - `add $1, $2, $3`
  - `sw $4, 0($1)`
  - `add $1, $2, $3`
  - `sw $3, 0($4)`

Complete pipelined datapath... so far
Summary

- In real code, most instructions are dependent upon other ones.
- This can lead to data hazards in our original pipelined datapath.
- Instructions can’t write back to the register file soon enough for the next two instructions to read.
- Forwarding eliminates data hazards involving arithmetic instructions.
- The forwarding unit detects hazards by comparing the destination registers of previous instructions to the source registers of the current instruction.
- Hazards are avoided by grabbing results from the pipeline registers before they are written back to the register file.
- Next, we’ll finish up pipelining.
- Forwarding can’t save us in some cases involving `lw`.
- We still haven’t talked about branches for the pipelined datapath.

What about stores?

- A harder case:
  1. `lw $1, 0($2)`
  2. `sw $1, 0($4)`

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

- What do we have to add to the datapath?

Miscellaneous comments

- Each MIPS instruction writes to at most one register.
- This makes the forwarding hardware easier to design, since there is only one destination register that ever needs to be forwarded.
- Forwarding is especially important with deep pipelines like the ones in all current PC processors.
- Section 6.4 of the textbook has some additional material not shown here.
- Their hazard detection equations also ensure that the source register is not $0$, which can never be modified.
- There is a more complex example of forwarding, with several cases covered. Take a look at it!