Today’s topics:
  – More pipelining…
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.
Our examples are too simple

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

\[
\begin{align*}
\text{lw} & \quad \text{\$8, 4\(\text{\$29}\)} \\
\text{sub} & \quad \text{\$2, \$4, \$5} \\
\text{and} & \quad \text{\$9, \$10, \$11} \\
\text{or} & \quad \text{\$16, \$17, \$18} \\
\text{add} & \quad \text{\$13, \$14, \$0}
\end{align*}
\]

- The instructions in this example are independent.
  - Each instruction reads and writes completely different registers.
  - Our datapath handles this sequence easily, as we saw last time.
- But most sequences of instructions are not independent!
An example with dependencies

```plaintext
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

This is not a problem for the single-cycle and multicycle datapaths. Each instruction is executed completely before the next one begins. This ensures that instructions 2 through 5 above use the new value of $2 (the sub result), just as we expect. How would this code sequence fare in our pipelined datapath?
### Data hazards in the pipeline diagram

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The **SUB** instruction does not write to register $2$ until clock cycle 5. This causes two **data hazards** in our current pipelined datapath.
  - The **AND** reads register $2$ in cycle 3. Since **SUB** hasn’t modified the register yet, this will be the *old* value of $2$, not the new one.
  - Similarly, the **OR** instruction uses register $2$ in cycle 4, again before it’s actually updated by **SUB**.
Things that are okay

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)

- 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 $2 after the SUB finishes.
### Dependency arrows

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

- `sub $2, $1, $3`
- `and $12, $2, $5`
- `or $13, $6, $2`
- `add $14, $2, $2`
- `sw $15, 100($2)`
A fancier pipeline diagram

Clock cycle

1 2 3 4 5 6 7 8 9

sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
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 $2$.
- When is the data actually produced and consumed?
- What can we do?

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>$2$, $1$, $3$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>$12$, $2$, $5$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>$13$, $6$, $2$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>
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 $2$.
- 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 $2$ in their EX stages, during clock cycles 4-5 here.

```
<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $2$, $1$, $3$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and $12$, $2$, $5$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or $13$, $6$, $2$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Bypassing the register file

- The actual result $1 - $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 time, 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.

```
Clock cycle

1  2  3  4  5  6  7

sub $2, $1, $3
    IF  ID  EX  MEM  WB

and $12, $2, $5
    IF  ID  EX  MEM  WB

or  $13, $6, $2
    IF  ID  EX  MEM  WB
```
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 $1 - $3 from the EX/MEM pipeline register used by sub.
  - Then in cycle 5, the OR can get that same result from the MEM/WB 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 MEM/WB pipeline registers instead.

- The ALU sources will be selected by two new multiplexers, with control signals named *ForwardA* and *ForwardB*.

```
sub  $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
```
Simplified datapath with forwarding muxes
Detecting EX/MEM data hazards

- So how can the hardware determine if a hazard exists?

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

and $12$, $2$, $5$
Detecting EX/MEM data hazards

- So how can the hardware determine if a hazard exists?
- An EX/MEM hazard occurs between the instruction currently in its EX stage and the previous instruction if:
  1. The previous instruction will write to the register file, and
  2. The destination is one of the ALU source registers in the EX stage.
- There is an EX/MEM hazard between the two instructions below.

\[
\text{sub } \$2, \$1, \$3
\]

\[
\text{and } \$12, \$2, \$5
\]

- Data in a pipeline register can be referenced using a class-like syntax. For example, `ID/EX.RegisterRt` refers to the rt field stored in the ID/EX pipeline.
EX/MEM data hazard equations

- The first ALU source comes from the pipeline register when necessary.
  
  if (EX/MEM.RegWrite = 1  
    and EX/MEM.RegisterRd = ID/EX.RegisterRs)  
  then ForwardA = 2  

- The second ALU source is similar.
  
  if (EX/MEM.RegWrite = 1  
    and EX/MEM.RegisterRd = ID/EX.RegisterRt)  
  then ForwardB = 2

sub $2, $1, $3  
and $12, $2, $5
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, $1, $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.

  \[
  \text{if } (\text{MEM/WB.RegWrite} = 1 \text{ and } \text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs} \text{ and } (\text{EX/MEM.RegisterRd} \neq \text{ID/EX.RegisterRs} \text{ or } \text{EX/MEM.RegWrite} = 0)) \text{ then } \text{ForwardA} = 1
  \]

- The second ALU operand is handled similarly.

  \[
  \text{if } (\text{MEM/WB.RegWrite} = 1 \text{ and } \text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt} \text{ and } (\text{EX/MEM.RegisterRd} \neq \text{ID/EX.RegisterRt} \text{ or } \text{EX/MEM.RegWrite} = 0)) \text{ then } \text{ForwardB} = 1
  \]
Simplified datapath with forwarding
The forwarding unit

- The forwarding unit has several control signals as inputs.

  ID/EX.RegisterRs  EX/MEM.RegisterRd  MEM/WB.RegisterRd
  ID/EX.RegisterRt  EX/MEM.RegWrite   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 ForwardA and ForwardB multiplexers attached to the ALU. These outputs are generated from the inputs using the equations on the previous pages.

- Some new buses route data from pipeline registers to the new muxes.
Example

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)

- Assume again each register initially contains its number plus 100.
  - After the first instruction, $2 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 $2.
  - We’ll skip the first two cycles, since they’re the same as before.
Clock cycle 3

IF: or $13, $6, $2
ID: and $12, $2, $5
EX: sub $2, $1, $3

IF/ID:
- PC
- Instruction memory

ID/EX:
- Registers
- ALU
- Forwarding Unit
- ID/EX. RegisterRt
- ID/EX. 1 RegisterRs
- EX/MEM. RegisterRd

EX/MEM:
- Data memory
- EX/MEM.RegisterRd

MEM/WB:
- MEM/WB.RegisterRd
Clock cycle 4: forwarding $2 from EX/MEM

IF: add $14, $2, $2
ID: or $13, $6, $2
EX: and $12, $2, $5
MEM: sub $2, $1, $3

PC

Instruction memory

Registers

Instruction memory

ALU

Data memory

Forwarding Unit

EX/MEM

MEM/WB

MEM/WB.RegisterRd

EX/MEM.RegisterRd

PC

Instruction memory

Registers
Clock cycle 5: forwarding $2 from MEM/WB

IF: sw $15, 100($2)
ID: add $14, $2, $2
EX: or $13, $6, $2
MEM: and $12, $2, $5
WB: sub $2, $1, $3

PC

Instruction memory

Registers

IF/ID

ID/EX

EX/MEM

MEM/WB

ALU

Data memory

Forwarding Unit

MEM/WB.RegisterRd

EX/MEM.RegisterRd

PC

Instruction memory

Registers

IF/ID

ID/EX

EX/MEM

MEM/WB

ALU

Data memory

Forwarding Unit

MEM/WB.RegisterRd

EX/MEM.RegisterRd
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 clock 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.
Complete pipelined datapath...so far
What about stores?

- Two “easy” cases:

  add $1, $2, $3

  sw $4, 0($1)
Store Bypassing: Version 1

EX: sw $4, 0($1)
MEM: add $1, $2, $3

Instruction memory

PC
Addr  Instr

IF/ID

ID/EX

ID/EX

EX/MEM

MEM/WB

RegDst

ALUSrc

Zero

Result

ALU

EX/MEM.RegisterRd

MEM/WB.RegisterRd

Forwarding Unit

EX/MEM.RegisterRd

MEM/WB.RegisterRd

Data memory

Address

Write data

Read data

EX/MEM.

MEM/.

0

1
Store Bypassing: Version 2

Instruction memory

IF/ID

ID/EX

EX/MEM

MEM/WB

PC

Addr  Instr

Instructions

Read register 1
Read register 2
Write register
Write data

Instr [15 - 0]

Rt

Rd

Rs

ID: ME

EX: sw $1, 0($4)

MEM: add $1, $2, $3

ALUSrc

ALU

Zero

Result

RegDst

Forwarding Unit

EX/MEM.RegisterRd

MEM/WB.RegisterRd

MEM: memory

Write data

Read data

Read data 1

Read data 2

Write register

Write data

Data memory

Address

Write data

Read data

1 0
What about stores?

- A harder case:

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

  - What do we have to add to the datapath?

lw $1, 0($2)

sw $1, 0($4)
Load/Store Bypassing: Extend the Datapath

Sequence:
lw $1, 0($2)
sw $1, 0($4)
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!
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.