Topic 10: Pipelining: Hazards and Forwarding

- Data Hazards
- Stalls
- Forwarding: Solving Data Hazards
- \( l_w \) Data Hazards
- Branch Hazards

https://www.youtube.com/watch?v=zOjYQTkrdl8&t=208s
Data Hazards

• Consider the following MIPS sequence:

  add $t0, $s1, $s2
  beq $t0, $zero, LABEL
  sw $t4, 0($t0)

• For each instruction, determine:
  - When it reads its input
  - When it calculates a value
  - When it stores the result back
Data Hazards

• Consider the following MIPS sequence:

  add $t0, $s1, $s2
  beq $t0, $zero, LABEL
  sw $t4, 0($t0)

Did you draw a picture, or work it by hand?
Draw a picture now!
add $t0, $s1, $s2
beq $t0, $zero, LABEL
sw $t4, 0($t0)
add $t0, $s1, $s2
beq $t0, $zero, LABEL
sw $t4, 0($t0)
Add $t0, $s1, $s2

beq $t0, $zero, LABEL

sw $t4, 0($t0)
add $t0, $s1, $s2
beq $t0, $zero, LABEL
sw $t4, 0($t0)
Data Hazards

- A **data hazard** is a situation where a value, which was recently calculated, is not yet available when we need to read it.
Group Exercise:

Reorder these instructions to eliminate the need for stalls:

\[
\begin{align*}
\text{sw} & \quad \$t0, 0($t1) \\
\text{sw} & \quad \$t2, 0($t3) \\
\text{add} & \quad \$s0, \$s1, \$s2 \\
\text{beq} & \quad \$s0, \$zero, \text{LABEL}
\end{align*}
\]

Draw a pipeline diagram to convince yourself that the new program doesn't need any stalls.
In this new order, the only dependency ($\text{addi} \rightarrow \text{beq}$) has two unrelated instructions between it. This solves the data hazard.

\begin{verbatim}
add $s0, $s1,$s2
sw $t0, 0($t1)
sw $t2, 0($t3)
beq $s0,$zero, LABEL
\end{verbatim}
Software Can't Do Everything

- Sometimes, there's no possible reordering to solve all issues
- Software written for one CPU might not solve all hazards in a different CPU
- Thus, hardware must always look for and solve all hazards
Stalls

- To solve any **hazard** (a data hazard is just one type) we can insert a **NOP** to “stall” the pipeline.
Avoid Software NOPs

• While software can add **NOPs**, that's not optimal
  - **Every** time it runs (on every CPU), we waste a cycle
  - What if fewer NOPs are required on some CPUs?

• Software should:
  - Use reordering to reduce hazards
  - (Almost) never use NOPs
How to implement a NOP?

- **Software solution:**
  \[
  0x0000_0000 \rightarrow sll \ $zero, \ $zero, \ 0
  \]

- **Hardware solution:**
  - Set every control bit to zero
NOP – Set all bits to Zero???

Setting every control bit to zero means that nothing* changes.

* What about the PC?
NOP – Set all bits to Zero???

NOP requires that we add a control bit to decide whether or not to update the PC.
“As If”

• What is the hardware required to do?

• The pipelined design must have the same output as a single-cycle design.

• It must work as if it was really a single-cycle processor.
While software can insert a **NOP** at any point, a **stall** (or “bubble”) is a NOP inserted by the hardware.

- Software doesn't know it exists!
Stalls are Wasteful

- No instruction completes during a NOP
Stalls are Wasteful

Group Exercise:

Imagine a program where every instruction reads the register written by the previous instruction.

How many stalls do we insert between each instruction?

What is the effective throughput, in instructions per clock (IPC)?
Insert 2 NOPs after each instruction.

IPC is 1/3.

This is 3x worse than we hoped for!
Data Forwarding

Insight:

We don't actually need to read the registers in the ID phase. All we need are the right values in the EX phase.

- **Data forwarding** uses values in the pipeline registers (not yet stored in WB), and delivers them directly to the ALU.
Group Exercise:

Simulate the following instructions. Do **NOT** insert stalls. For the EX phase of each instruction, determine:

- Were the values read in ID correct for both inputs?
- If not, what pipeline register contains the not-yet written-back values?

Draw a pipeline diagram, showing the dependencies.

```
sub    $2, $1,$3
and    $12, $2,$5
add    $13, $2,$2
or     $14, $5,$2
sw     $15, 100($2)
```
To represent data forwarding, we will only use vertical arrows.

Why?
The *and* instruction needs $2$, but it was out-of-date in ID.

It is currently in EX/MEM.
The add instruction needs $2 \text{ (twice!)}.\]

It is currently in MEM/WB.
The `or` instruction needs $2$, but it is written (in WB) on the same clock as ID. No forwarding is required.
The `sw` instruction reads $2$ on the clock after it is updated. No data hazard exists.
Data Forwarding

• Either (or both) of the ALU inputs in EX might require a value still in the pipeline.

• Need to detect this condition, and then choose alternate values for the ALU inputs (more MUXes!)
New component: the Forwarding Unit
Needs to know the rs, rt fields (not just the register values).
Needs to know whether the previous two instructions were updating registers (RegWrite).

Why connect at two places?
Needs to know which registers they were updating (output from the RegDst MUX).
Expand the ALU input 2 MUX.

Add a new ALU input 1 MUX.
Data Forwarding

Group Exercise, part 1:

How do the updated ALU input MUXes work?

Start by answering the following questions:

- How many different inputs are possible at each?
- How many different control fields do you need to check?
- In what order should you make the checks?
Data Forwarding

**Inputs to the ALU input 1 MUX:**

```
  id_ex .rsVal
  ex_mem.aluResult
  mem_wb.aluResult
  mem_wb.memResult
```

**Q:** Which of the aluResults to check 1\textsuperscript{st}?

**Control Fields:**

```
  id_ex .rs
  ex_mem.regWrite
  ex_mem.writeRegister
  mem_wb.regWrite
  mem_wb.writeRegister
  mem_wb.memToReg
```

**Group Exercise, part 2:**

Now, write the pseudocode for ALU input 1, using these inputs.
getALUinput1():
    if (ex_mem.regWrite &&
        ex_mem.writeReg == id_ex.rs)
    {
        return ex_mem.aluResult;
    }

    ... now check mem_wb ...
    ... can we forward from memResult?
    ...

Data Forwarding

• What makes ALU input 2 more complex than ALU input 1?
  Immediate Fields!

• Can your code support both sign-extended and zero-extended immediate fields?
Data Forwarding

Group Exercise:

Draw a pipeline diagram to represent the following program. Be sure to mark all data hazards.

```
add    $t0, $s0,$s1
lw     $s2, 100($t0)
sll    $s4, $s2,$s3
```
Data Forwarding

There is a dependency from `add` to `lw`.

This can be handled by forwarding.

```
add $t0, $s0,$s1
lw $s2, 100($t0)
sll $s4, $s2,$s3
```
Data Forwarding

There is a dependency from `lw` to `sll`.

Why doesn't forwarding work?
We have a "time travel" case again here.

We need to stall the \texttt{sll} instruction!
Data Forwarding

```
add $t0, $s0, $s1
lw  $s2, 100($t0)
sll $s4, $s2, $s3
```
Principle: Forwarding is Limited

• Solving data hazards:
  – If each instruction uses the same stage, we can use forwarding to solve dependencies
  – If they use **different** stages, unavoidable stalls can result

• So adding more pipeline stages is sometimes a bad idea...
Implementing Stalls

- Can't know that a stall is required until ID stage
- IF stage is already fetching **next** instruction

```
PC=x+4  PC=x  PC=x-4  PC=x-8  PC=x-12
```
Group Exercise:

Suppose that the instruction in ID needs to stall.

Which instructions should be in which stages, in the next clock cycle? Remember to insert a NOP. And don't discard any of the real instructions!

What new control bits do we need to implement this feature?
Implementing Stalls

- Can't know that a stall is required until ID stage
- IF stage is already fetching next instruction
- Need to stall ID stage and IF stage
- Re-execute same steps next clock
  - Don't advance PC
  - Don't update 'instruction' field in ID/IF
  - Set NOP in ID/EX
Expand the Control to include “Hazard Detection.”

(This is different than the Forwarding Unit.)
(victim)

Must stall if:
- ID/EX is $l_w$
- Cur inst has data hazard
Read \textit{rt} from ID/EX.

Why is this important?
Read $rs$, $rt$ from current instruction.
Read \textit{memRead} from ID/EX.
Hazard Detection

Group Exercise:

Write pseudocode that will determine whether or not a \texttt{lw} hazard exists.

Use the inputs:
- \texttt{rt} from ID/EX
- \texttt{memRead} from ID/EX
- \texttt{rs}, \texttt{rt} from IF/ID

(For simplicity, ignore – for now – the fact that only some instructions actually read \texttt{rsVal, rtVal}.)
hazardExists():
    if (id_ex.memRead == false)
        return false

    if (id_ex.rt == if_id.rs ||
        id_ex.rt == if_id.rt)
    {
        return true;
    }
    else
    {
        return false;
    }

Question:
Will this report a hazard next cycle as well?
If a hazard exists:

Insert a NOP. We use a MUX to choose “normal control, or NOP.”
If a hazard exists:

**Disable** writing to the IF/ID register.
If a hazard exists:

Disable writing to the PC.
One Last Type of Data Forwarding

Group Exercise:

Draw a pipeline diagram to represent the following program. Be sure to mark all data hazards.

```assembly
add  $t0, $s0,$s1
sw   $t0, 0($t1)
```
There is a dependency from add to sw – but it's for the data register!

This can be handled by forwarding, as well.
One Last Type of Data Forwarding

Group Exercise:

But wait...there's another wrinkle...

```
add  $t0, $s0,$s1
addi $t8, $t8,1
sw   $t0, 0($t1)
```
Data Forwarding

There is a dependency from `add` to `sw` – but the result from `add` is already gone when we need it!
Data Forwarding

\[
\begin{align*}
&\text{add} \quad $t0, \quad $s0,\$s1 \\
&\text{addi} \quad $t8, \quad $t8,1 \\
&\text{sw} \quad $t0, \quad 0($t1) \\
&\text{sw} \quad $t0, \quad 0($t1)
\end{align*}
\]
Branch Hazards

• It is (finally) time to get honest about \texttt{beq}
• It's another type of hazard
This was our pipeline from the previous deck.

We calculated the branch destination in EX.
This was our pipeline from the previous deck.

We didn't know whether to branch until MEM – but we pretended that we could know this in IF.
Branch Prediction

**Insight:**
PC+4 is a good default value. Why not use it all the time? Branches are rare.

- **Branch prediction** means making a guess about the result of a branch
  - Simple: PC+4
  - Harder: Guess destination during IF
Branch Prediction

- **Branch prediction** works well if our guess was correct
  - No stalls

- If the guess was wrong, we must **kill** instructions already partially-executed. This is a **pipeline flush**.
  - Turns existing instructions into NOPs
  - Safe so long as no external actions yet (load/store)
Branch Prediction

- Modern processors use branch prediction with pipeline flushing. Our simple MIPS processor does not.

- Instead, how to minimize delay?
Move branch adder into ID phase.
How to compare without the ALU?

(Adding an ALU to ID too expensive, too slow.)
Add a simple “compare for equality” to ID.

Cheaper/faster than an ALU.
Branch Delay Slot

- What about the instruction right after `beq/bne`?
  - It's in IF while branch is happening in ID

- Options:
  - Hardware enforces stall
  - Software inserts NOP
  - **Branch delay slot** (the MIPS choice)
What do do with the “next” instruction?
MIPS says: always execute it!

Branch takes place after that instruction.
Branch Delay Slot

• In this program, the add always runs:
  
  ```
  beq  $s0,$s1,LABEL
  add  $s2,$s3,$s4  \textcolor{red}{\textit{branch delay slot}}
  ori  $s2,$s2,0xff
  ```

• This is used for all branch/jump instructions:
  
  - beq/bne
  - j
  - jal
  - jr
Branch Delay Slot

- **Branch delay slot** is good for hardware performance
- But confusing to programmers

- Most assemblers hide it from the programmer
  - Reorder instructions when possible
  - Insert NOP when no alternative