

Pipelining (III)

Hazards (cont.)

Data Hazards

- Data hazards occur in executing [arithmetic or data transfer operations](#), when the [next instruction needs to wait for the previous instruction](#) to complete its data read/write.

- For example, if we have the following two instructions:

LOAD R1 A # A $\Rightarrow$ R1

SUB R0 R1 R2 # R0 - R1 $\Rightarrow$ R2

- R1 used by SUB won't be ready till after executing LOAD R1 A.
- So, we have to stall one stage for the SUB instruction. The pipelining for these two instructions look like

```

load  F      D      E      W
      \
sub           F      D      E      W

```

- We also call this type of data hazard “[load-use data hazard](#),” which is a specific form of data hazard, in which the [date being loaded](#) by a load instruction has [not yet become available when it is needed by another instruction](#).
- Another way to address the data hazards is to [retrieve](#) the missing data element [from internal buffers](#) (data path) rather than waiting for it to arrive from programmer-visible register or memory. We name this solution “[forwarding](#).”

- For example, if we have the following two instructions:

ADD R1 R2 R0 # R1 + R2 $\Rightarrow$ R0

SUB R0 R3 R4 # R0 - R3 $\Rightarrow$ R4

- The value of R0, which is R1 + R2, used by SUB won't be ready till ALU finish the calculation.

- If using forwarding, the pipelining for these two instructions look like

```

add  F      D      E      W
      \
sub           F      D      E      W

```

- In this way, there is no need to stall one stage for the SUB instruction.

Control Hazards

- Control hazards occur in **branch operation**, where **fetching next instruction depends on branch outcome**.
 - Sol1: **Wait** until branch outcome determined before fetching next instruction.
 - Longer pipelines can't readily determine branch outcome early. In this case, the long waiting caused by stall penalty becomes unacceptable.
 - Sol2: **branch prediction**

Branch Prediction

- A method of resolving a control hazard that **assumes a given outcome for the branch and proceeds from that assumption** rather than waiting to a certain actual outcome.
 - Predict outcome of branch, **only stall if prediction is wrong**.
- Prediction scheme:
 - Assume the branch is not taken:
 - Continue executing the sequential instructions. If the branch is taken, the instruction that are being fetched and decoded must be discarded. Execution continues at the branch target.
 - Discarding instructions means flush these instructions in the pipeline.
- Loop predictor
 - Loops are a common component of programs.
 - A conditional branch is always at the bottom of a loop that will be repeated N times. N-1 times the branch will not be taken, and 1 time it will be taken.
 - Can be implemented by using a counter.
- History-based prediction: correct 90% of time
 - Assume future behavior will continue the trend
 - If wrong, stall while re-fetching, and update history table.