Hazards

Situations that prevent starting the next instruction in the next cycle

Structural hazards
  - A required resource is busy

Data hazard
  - Need to wait for previous instruction to complete its data read/write

Control hazard
  - Deciding on control action depends on previous instruction
Conflict for use of a resource

In MIPS pipeline with a single memory
  – Load/store requires data access
  – Instruction fetch would have to stall for that cycle

Hence, pipelined datapaths require separate instruction/data memories
  – Or separate instruction/data caches
  – Or dual-ported memories
Data Hazards

An instruction depends on completion of writeback by a previous instruction

\[
\begin{align*}
\text{add} &: \$s0, \$t0, \$t1 & \text{// writes $s0$ from WB stage} \\
\text{sub} &: \$t2, \$s0, \$t3 & \text{// needs value in ID stage}
\end{align*}
\]
Forwarding (aka Bypassing)

Use result when it is computed
- Don’t wait for it to be stored in a register
- Requires extra connections in the datapath (& more control logic?)

Program execution order (in instructions)

No stall
Load-Use Data Hazard

Can’t always avoid stalls by forwarding
- If value not computed when needed
- Can’t forward backward in time!

lw $s0, 20($t1)
sub $t2, $s0, $t3

1-stage stall
Code Scheduling to Avoid Stalls

Reorder code to avoid use of load result in the next instruction

C code for \( A = B + E; \ C = B + F; \)

\[
\begin{align*}
\text{lw} & \quad \$t1, 0($t0) \\
\text{lw} & \quad \$t2, 4($t0) \\
\text{add} & \quad \$t3, \$t1, \$t2 \\
\text{sw} & \quad \$t3, 12($t0) \\
\text{lw} & \quad \$t4, 8($t0) \\
\text{add} & \quad \$t5, \$t1, \$t4 \\
\text{sw} & \quad \$t5, 16($t0)
\end{align*}
\]

13 cycles

\[
\begin{align*}
\text{lw} & \quad \$t1, 0($t0) \\
\text{lw} & \quad \$t2, 4($t0) \\
\text{lw} & \quad \$t4, 8($t0) \\
\text{add} & \quad \$t3, \$t1, \$t2 \\
\text{sw} & \quad \$t3, 12($t0) \\
\text{add} & \quad \$t5, \$t1, \$t4 \\
\text{sw} & \quad \$t5, 16($t0)
\end{align*}
\]

11 cycles

Who reorders the code?
Control Hazards: \texttt{beq}

Branch determines flow of control

- Fetching next instruction depends on branch outcome
  - Register comparison done in EX stage
  - Branch target address computed in EX stage
  - MUX selection done in MEM stage

What instruction do we fetch when \texttt{BEQ} is in ID stage?
Control Hazards: beq

How can we deal with BEQ?

Stall until we know whether (& where) to branch?

Make the decision and calculate the address earlier?

Guess whether the branch will be taken?
Stall on Branch

Wait until branch outcome determined before fetching next instruction

How many cycles does this cost?
What new hardware would be needed to decide earlier?

Must compare the registers before the EX stage

Must compute the branch target address before the EX stage

Can we know what to do by the time BEQ enters the ID stage?
Branch Prediction

Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable

Predict outcome of branch
  - Only stall if prediction is wrong

In MIPS pipeline
  - Can predict branches will not be taken
  - Fetch sequential instruction after branch, with no delay
MIPS with Predict Not Taken

**Prediction correct**
- `add $4, $5, $6`
- `beq $1, $2, 40`
- `lw $3, 300($0)`

**Prediction incorrect**
- `add $4, $5, $6`
- `beq $1, $2, 40`
- or `$7, $8, $9`

Program execution order (in instructions)

**Pipeline**

Instruction fetch | Reg | ALU | Data access | Reg
--- | --- | --- | --- | ---
Instruction fetch | Reg | ALU | Data access | Reg
Instruction fetch | Reg | ALU | Data access | Reg
Instruction fetch | Reg | ALU | Data access | Reg
Pipelining improves performance by increasing instruction throughput
- Executes multiple instructions in parallel
- Each instruction has the same latency

Subject to hazards
- Structure, data, control

Instruction set design affects complexity of pipeline implementation