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
Structural Hazards

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
  - Would cause a pipeline “bubble”

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

An instruction depends on completion of writeback by a previous instruction

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

2-stage stall
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?)

![Diagram showing pipeline stages and forwarding](image-url)

- Program execution order (in instructions)
  - add $s0, $t0, $t1
  - sub $t2, $s0, $t3

- Time: 200, 400, 600, 800, 1000
- Pipeline stages: IF, ID, EX, MEM, WB
- 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!

1-stage stall
Code Scheduling to Avoid Stalls

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

\[
A = B + E; \quad C = B + F;
\]

C code for:

\[
lw \quad t1, 0(t0) \\
lw \quad t2, 4(t0) \\
add \quad t3, t1, t2 \\
w \quad t3, 12(t0) \\
lw \quad t4, 8(t0) \\
add \quad t5, t1, t4 \\
w \quad t5, 16(t0)
\]

13 cycles

vs.

\[
lw \quad t1, 0(t0) \\
lw \quad t2, 4(t0) \\
lw \quad t4, 8(t0) \\
add \quad t3, t1, t2 \\
w \quad t3, 12(t0) \\
add \quad t5, t1, t4 \\
w \quad t5, 16(t0)
\]

11 cycles
Control Hazards

Branch determines flow of control
- Fetching next instruction depends on branch outcome
- Pipeline can’t always fetch correct instruction
  - Still working on ID stage of branch

In MIPS pipeline
- Need to compare registers and compute target early in the pipeline
- Add hardware to do it in ID stage?
Stall on Branch

Wait until branch outcome determined before fetching next instruction

Program execution order (in instructions)

Time

<table>
<thead>
<tr>
<th></th>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
<th>1000</th>
<th>1200</th>
<th>1400</th>
</tr>
</thead>
</table>

- add $4, $5, $6
- beq $1, $2, 40
- or $7, $8, $9

Instruction fetch
Reg
ALU
Data access
Reg
Instruction fetch
Reg
ALU
Data access
Reg

200 ps
400 ps
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)

Program execution order (in instructions)
- Time: 200 ps

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

Program execution order (in instructions)
- Time: 400 ps
More-Realistic Branch Prediction

Static branch prediction
- Based on typical branch behavior
- Example: loop and if-statement branches
  - Predict backward branches taken
  - Predict forward branches not taken

Dynamic branch prediction
- Hardware measures actual branch behavior
  - e.g., record recent history of each branch
- Assume future behavior will continue the trend
  - When wrong, stall while re-fetching, and update history
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