Hazards

- **hazard**: previous instruction blocks following instruction
- **structural hazards**
  - required resource is busy
- **data hazard**
  - 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: **stall** for that cycle causing a pipeline “bubble”
- hence pipelined datapaths require
  - separate instruction/data memories
  - or separate instruction/data caches

Data hazards

- an instruction depends on completion of data access by a previous instruction
  - `add $s0, $t0, $t1`
  - `sub $t2, $s0, $t3`

Forwarding or bypassing

- use result when it is computed
  - do not wait for it to be stored in a register
  - requires extra connections in the datapath
Load-use data hazard

- cannot always avoid stalls by forwarding
  - if value not computed when needed
  - cannot forward backward in time!

Code scheduling to avoid stalls

- reorder code to avoid use of load result in the next instruction
- C code for \( A = B + E; \ C = B + F; \)

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

11 cycles

13 cycles

Control hazards

- branch determines flow of control
  - fetching next instruction depends on branch outcome
  - pipeline can’t always fetch correct instruction
  - e.g. still working on ID stage of branch

- In MIPS pipeline
  - compare registers and compute target early in the pipeline
  - have to add hardware to do it in ID stage

Stall on branch

- wait until branch outcome determined before fetching next instruction
Branch prediction

- longer pipelines cannot 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 not taken
  - fetch instruction after branch, with no delay

MIPS with prediction

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
  - only maintain information about small number of branches
Pipelined control: simplified

Pipelined control

- control signals derived from instruction
  - as in single-cycle implementation

Data hazards in ALU instructions

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

- can resolve hazards with forwarding
  - how to detect when to forward?
Dependencies and forwarding

Detecting: need to forward

- pass register numbers along pipeline
  - e.g. ID/EX.RegRS = register number for register RS sitting in ID/EX pipeline register

- ALU operand register numbers in EX stage: given by
  - ID/EX.RegRS, ID/EX.RegRT

- data hazards when
  1a. ID/EX.RegRS = EX/MEM.RegRD
  1b. ID/EX.RegRT = EX/MEM.RegRD
  2a. ID/EX.RegRS = MEM/WB.RegRD
  2b. ID/EX.RegRT = MEM/WB.RegRD

Detecting: need to forward

- but only if forwarding instruction writes to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite

- and only if Rd for that instruction is not $0
  - EX/MEM.RegRD ≠ 0, MEM/WB.RegRD ≠ 0

- When would we use $0 as a destination?
Forwarding paths

![Diagram of forwarding paths]

Forwarding conditions

- **EX hazard**
  - if (EX/MEM.RegWrite and (EX/MEM.RegRD ≠ 0) and (EX/MEM.RegRD = ID/EX.RegRS))
    - then ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegRD ≠ 0) and (EX/MEM.RegRD = ID/EX.RegRT))
    - then ForwardB = 10

- **MEM hazard**
  - if (MEM/WB.RegWrite and (MEM/WB.RegRD ≠ 0) and (MEM/WB.RegRD = ID/EX.RegRS))
    - then ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegRD ≠ 0) and (MEM/WB.RegRD = ID/EX.RegRT))
    - then ForwardB = 01

Double data hazard

- consider the sequence:
  - add $1, $1, $2
  - add $1, $1, $3
  - add $1, $1, $4
- both hazards occur
  - want to use the most recent
- revise MEM hazard condition
  - only forward if EX hazard condition is not true

Revised forwarding condition

- **MEM hazard**
  - if (MEM/WB.RegWrite and (MEM/WB.RegRD ≠ 0)
    and (MEM/WB.RegRD = ID/EX.RegRS)
    and not (EX/MEM.RegWrite and (EX/MEM.RegRD ≠ 0)
    and (EX/MEM.RegRD = ID/EX.RegRS)))
    - then ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegRD ≠ 0)
    and (MEM/WB.RegRD = ID/EX.RegRT)
    and not (EX/MEM.RegWrite and (EX/MEM.RegRD ≠ 0)
    and (EX/MEM.RegRD = ID/EX.RegRT)))
    - then ForwardB = 01
Datapath with forwarding

Load-use data hazard

Load-use hazard detection
- check: when instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegRS, IF/ID.RegRT
- load-use hazard when
  - ID/EX.MemRead and
    \((IF/ID.RegRS = ID/EX.RegRT)\) or \((IF/ID.RegRT = ID/EX.RegRT)\)
- if detected, stall and insert bubble

How to stall the pipeline
- force control values in ID/EX register to 0
  - EX, MEM and WB do **nop** (no-operation)
- prevent update of PC and IF/ID register
  - current instruction is decoded again
  - following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
- will forward to EX stage in next cycle
Stall/bubble in the pipeline

Datapath with hazard detection

Summary: stalls and performance

• stalls
  – reduce performance
  – but are required to get correct results

• compiler
  – arranges code to avoid hazards and stalls
  – requires knowledge of the pipeline structure