Pipeline Hazards

or

_Danger!Danger!Danger!

Data Dependence

- Data hazards are caused by data dependences
- Data dependences, and thus data hazards, come in 3 flavors
  - RAW (read-after-write)
  - WAW (write-after-write)
  - WAR (write-after-read)

RAW Hazard

- later instruction tries to read an operand before earlier instruction writes it
- The dependence
  
  | add R1, R2, R3 | IF | ID | EX | MEM | WB |
  |
  | sub R5, R1, R4 | IF | ID | EX | MEM | WB |

- The hazard
  
  | add R1, R2, R3 | IF | ID | EX | MEM | WB |
  |
  | sub R5, R1, R4 | IF | ID | EX | MEM | WB |

- RAW hazard is extremely common

WAW Hazard

- later instruction tries to write an operand before earlier instruction writes it
- The dependence
  
  | add R1, R2, R3 | IF | ID | EX | MEM | WB |
  |
  | sub R5, R1, R4 | IF | ID | EX | MEM | WB |

- The hazard
  
  | lw R1, R2, R3 | IF | ID | EX | MEM | MEM | MEM | MEM | WB |
  |
  | sub R1, R2, R4 | IF | ID | EX | MEM | WB |

- WAW hazard is possible in a reasonable pipeline
WAR Hazard

- later instruction tries to write an operand before earlier instruction reads it
- The dependence
  - add R1, R2, R3
  - sub R2, R5, R4

- The hazard?
  - add R1, R2, R3
  - sub R2, R5, R4

- WAR hazard is uncommon/impossible in a reasonable (in-order) pipeline

Data Hazards

```plaintext
ADD R1, R2, R3
SUB R4, R5, R1
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
```
Dealing with Data Hazards through Forwarding

Forwarding Options

- ADD -> ADD
- ADD -> LW
- ADD -> SW (2 operands)
- LW -> ADD
- LW -> LW
- LW -> SW (2 operands)

(I'm letting ADD stand in for all ALU operations)
More Forwarding

Forwarding and Stalling

Why a bubble?

Example

Avoiding Pipeline Stalls

lwl R1, 1000(R2)
lw R3, 2000(R2)
add R4, R1, R3
lw R1, 3000(R2)
add R6, R4, R1
sw R6, 1000(R2)

• Can we eliminate the stalls?
• this is a compiler technique called instruction scheduling.
How big a problem are these pipeline stalls?

- 13% of the loads in FP programs
- 25% of the loads in integer programs

Detecting ALU Input Hazards

ID/EX | EX/MEM | MEM/WB
------------------
| ALU |

Inserting Bubbles

- Set all control values in the EX/MEM register to zero (inserts a no-op instruction)
- Keep same values in the ID/EX register and IF/ID register
- Keep PC from incrementing
Adding Datapaths

Control Hazards

- Instructions are not only dependent on instructions that produce their operands, but also on all previous control flow (branch, jump) instructions that lead to that instruction.

Branch Hazards

Problem??
Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- DLX branch tests if register = 0 or ≠ 0
- DLX Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

New Datapath

Branch Hazards

What We Know About Branches

- more conditional branches than unconditional
- more forward than backward
- 67% of branches taken
- backward branches taken 80%
Four Branch Hazard Alternatives

#1: Stall until branch direction is clear
#2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - “Squash” instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 33% DLX branches not taken on average
  - PC+4 already calculated, so use it to get next instruction
#3: Predict Branch Taken
  - 67% DLX branches taken on average
  - But haven't calculated branch target address in DLX
    - DLX still incurs 1 cycle branch penalty
    - Other machines: branch target known before outcome

#4: Delayed Branch
  - Define branch to take place *AFTER* a following instruction
  - 1 slot delay allows proper decision and branch target address in 5 stage pipeline
  - DLX uses this

Delayed Branch

- Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken
  - Cancelling branches allow more slots to be filled

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled