Control Hazards

- Created by branch statements
  
  BEQZ LOC R1, R2, R3
  ADD LOC R1, R2, R3

- PC needs to be computed but it happens too late (at the end of ID, the 2nd stage)

Four Branch Hazard Alternatives

#1 STALL: until branch direction is known
#2 Predict Branch Not Taken: guess that the branch will not be taken and execute successor instruction. Cancel out instructions in the pipeline if wrong guess.
#3 Predict Branch Taken: opposite of above.
#4 Delayed Branch: insert useful instructions into the pipeline until the branch direction is known.
Solution #1: Stall

- PC is computed in ID stage
- Thus only a 1 cycle stall is incurred for branch hazards

Solution #1 (STALL) incurs a penalty of 1 cycle.

#2 Predict Not Taken (Fig. A.12)

Figure A.12: The predicted-not-taken scheme and the pipeline sequence when the branch is not taken (top) and taken (bottom). When the branch is not taken, determined during ID, we have fetched the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.
#2 Predict Not Taken

- Penalty: 0 if not taken, 1 if taken

Assume 30% conditional branches of which 60% are not taken:

\[
\text{CPI} = (.70) ( 1) + (.30)((.60)(1) + (.40)(1+1))
\]

#3 Predict Taken

- Similar situation and analysis as Predict Not Taken
#4 Delayed Branch

Delay Slot = the slots in the pipeline that would be stalls (we are going to fill them with instructions and try to avoid stalls)

(a) Non-cancelling Delayed Branch
   Useful instructions are inserted into the delay slots

(b) Cancelling Delayed Branch
   Some instructions are inserted into the delay slots and cancelled if wrong guess (as in Predict Not Taken and Predict Taken)

#4a: Delayed Branch - non-cancelling

- Fill the slot with a useful instruction
  penalty = 0
- Sometimes no useful instruction can be found by the compiler
  penalty = 1
#4b: Delayed Branch - cancelling with predict taken

- Try to fill with a useful instruction: since we predict taken, it can be chosen from instructions at the taken location
- If the prediction was right
  DLX penalty = 0
- If the prediction was wrong, the instruction must be cancelled
  DLX penalty = 1
Where to get instructions to fill the branch delay slot(s)?

- From before the branch instruction
- From the target address (only OK if branch taken)
- From the fall through (only OK if branch not taken)

Compiler effectiveness for single branch delay slot:
- about 60% of slots are filled
- about 80% of instructions in delay slots are useful (not cancelled)
Predict (un)taken vs. Cancelling branch

- Predict taken/untaken
  - is all done in hardware
  - when branch is in ID stage, PC is modified to taken (or untaken) address
  - special hardware to detect whether the right choice was made (comparator) and hardware to cancel (change to NO-OP)
  - no special branch instructions needed

Predict (un)taken vs. Cancelling branch (cont)

- a combination of hardware and software
- compiler analyzes code and selects either cancelling or non-cancelling branch instructions (the programmer just uses the generic one)
- compiler rearranges the machine instructions before execution
- at execution time, hardware cancels (converts to NO-OP) if the wrong prediction is made
Delayed branch options

- BEST: find a useful instruction that is independent of the branch
- NEXT BEST: find a useful instruction in the target code (predict taken) that has no negative impact if not taken
- OK: find a useful instruction in the target code (predict taken); if guessed wrong, it will be cancelled at runtime
- WORST: empty slot

Performance Analysis

Caution: you must be clear on the assumptions!
Assume: equal pipestages, no change in IC or clock, instructions all require all pipestages in the unpiplined version.

\[
Pipeline \ Speedup = \frac{CPI_{\text{non-pipelined}}}{CPI_{\text{pipelined}}} = \frac{1}{1 + \text{pipeline depth}}
\]

The penalty due to stalls depends on frequency analysis.
Details of the MIPS datapath (Fig. A.17 - does not include pipeline HW)

- Instruction Fetch (IF)
  \[ \text{IR} \leftarrow \text{Mem}[PC] \]
  \[ \text{NPC} \leftarrow \text{PC} + 4 \]

- Instruction Decode/Register Fetch (ID)
  \[ \text{A} \leftarrow \text{Regs}[rs] \]
  \[ \text{B} \leftarrow \text{Regs}[rt] \]
  \[ \text{Imm} \leftarrow \text{sign extended immediate field of IR} \]

- Execution/Effective address (EX)

- Memory access/Branch completion (MEM)

- Write-back (WB)
MIPS pipeline details (p. A27-A28)

• Execution/Effective address (EX)
  If memory reference: \( \text{ALUOutput} \leftarrow A + \text{Imm} \)
  If Reg-Reg ALU: \( \text{ALUOutput} \leftarrow A \text{ func B} \)
  If Reg-Imm ALU: \( \text{ALUOutput} \leftarrow A \text{ op Imm} \)
  If Branch: \( \text{ALUOutput} \leftarrow \text{NPC} + (\text{Imm} \ll 2) \)

• Memory access/Branch completion (MEM)
  If memory reference: \( \text{LMD} \leftarrow \text{Mem[ALUOutput]} \)
  or \( \text{Mem[ALUOutput]} \leftarrow B \)
  If Branch: if (cond) \( \text{PC} \leftarrow \text{ALUOutput} \)

• Write-back (WB)
  If Reg-Reg ALU: \( \text{Regs[rd]} \leftarrow \text{ALUOutput} \)
  If Reg-Imm ALU: \( \text{Regs[rt]} \leftarrow \text{ALUOutput} \)
  If Load: \( \text{Regs[rt]} \leftarrow \text{LMD} \)
Details of the MIPS pipeline HW

MIPS pipeline events compared to MIPS (non-pipelined) datapath (Fig. A.19)

Non-pipelined
- Instruction Fetch (IF)
  IR <-- Mem[PC]
  NPC <-- PC + 4

Pipelined
- Instruction Fetch (IF)
  IF/EX.IR <-- Mem[PC]
  IF/ID.NPC,PC <-- (if ((EX/MEM.opcode == branch) &
                          EX/MEM) {EX/MEM.ALUOutput}
                          else {PC + 4})
**How HW detects data hazards**  
*(See Fig. A.21)*

LW      R1, 0(R2)       in EX stage  
ADD    R3, R1, R4       in ID stage

Check opcode field of ID/EX (bits 0-5): LOAD  
Check opcode field of IF/ID (bits 0-5): ALU

Check operands for RAW:  
ID/EX.IR[rt] == IF/ID.IR[rs]

---

**How HW accomplishes forwarding**  
*(See Fig. A.22 which show all 10 needed comparisons)*

Example of first comparison:

PipelineReg. source: EX/MEM  
Opcode of source: Reg-Reg ALU  
PipelineReg. destination: ID/EX  
Opcode of destination: Reg-Reg ALU, ALU imm, load, store, branch  
Destination of forward: Top ALU input  
Comparison test for forward: EX/MEM.IR[rd] == ID/EX.IR[rs]

etc. etc. etc.
Forwarding hardware