Control Hazards

- Conditional branches
  - If the branch is taken (condition is true) next instruction should be fetched from the target address
  - We have necessary data at the end of ID stage (assuming branch target calculation is done in ID stage!!!)
  - The next instruction needs the data at the beginning of IF stage

Static Techniques to Reduce Branch Penalties

- Freeze or flush the pipeline
- Predict not taken
- Predict taken

Flush Pipeline

- Freeze pipeline until branch outcome is known

<table>
<thead>
<tr>
<th>Branch (not taken)</th>
<th>IF ID EX MEM WB</th>
<th>Branch (taken)</th>
<th>IF ID EX MEM WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i+1</td>
<td>IF ID EX MEM WB</td>
<td>Instruction i+2</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Instruction i+2</td>
<td>IF ID EX MEM WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Predict Not Taken

- Treat every branch as \textit{not taken}
- If \textit{taken}, turn next instruction into \textit{noop}

<table>
<thead>
<tr>
<th>Branch (not taken)</th>
<th>IF ID EX MEM WB</th>
<th>Branch (taken)</th>
<th>IF ID EX MEM WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i+1</td>
<td>IF ID EX MEM WB</td>
<td>Instruction i+2</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Instruction i+2</td>
<td>IF ID EX MEM WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch target</td>
<td>IF ID EX MEM WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Predict Taken

- Treat every branch as \textit{taken}, continue fetching instructions from the target address
- If not taken, turn newly fetched instructions into \textit{noop}
- This would make sense if we knew branch target \textit{before} condition
- In our pipeline it doesn’t make sense as we know branch target at the same time as condition

Delayed Branches

- Compiler reorders instructions so that the next instruction after the branch is:
  - Useful in most cases
  - Harmless both in \textit{taken} and \textit{not taken} case

<table>
<thead>
<tr>
<th>Branch (not taken)</th>
<th>IF ID EX MEM WB</th>
<th>Delay instruction</th>
<th>IF ID EX MEM WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i+1</td>
<td>IF ID EX MEM WB</td>
<td>Instruction i+2</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Instruction i+2</td>
<td>IF ID EX MEM WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch target</td>
<td>IF ID EX MEM WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Delayed Branch

- From before:
  - DADD R1, R2, R3
  - If R2=0 then:
    - DADD R4, R5, R6
- From target:
  - DADD R1, R2, R3
  - If R2=0 then:
    - DSUB R4, R5, R6
- From fall-through:
  - DADD R1, R2, R3
  - If R2=0 then:
    - DADD R4, R5, R6

Nullifying Branches

- Like delayed branches, but they may be harmful if we mispredicted.
- Compiler predicts the outcome of a branch and selects delay instruction.
- Compiler adds the prediction data into branch instruction.
- If branch behaves as predicted the delay instruction is executed, otherwise it is turned into noop.

Performance Degradation Due to Control Hazards

\[
\text{Speedup} = \frac{n}{1 + \text{Branch frequency} \times \text{Branch penalty}}
\]

Example

Assume a deeper pipeline where it takes 3 pipeline stages until branch target is known and 1 more to evaluate branch condition. Find the effective addition to CPI assuming the following frequencies of branches and the following branch penalties:

- Unconditional branch: 4%
- Conditional branch, untaken: 6%
- Conditional branch, taken: 10%

<table>
<thead>
<tr>
<th>Branch Type</th>
<th>Frequency</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional</td>
<td>4%</td>
<td>100%</td>
</tr>
<tr>
<td>Conditional, untaken</td>
<td>6%</td>
<td>100%</td>
</tr>
<tr>
<td>Conditional, taken</td>
<td>10%</td>
<td>100%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Unconditional</th>
<th>Untaken</th>
<th>Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush pipeline</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Predict taken</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Predict untaken</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Pipeline Implementation

- Branch target is calculated in EX stage.
- Consider only BEQZ branches, comparison in EX stage.
- Instructions:
  - opcode, rd, rs, rt
  - opcode, rd, rs, imm

Pipeline Implementation

- IF stage:
  - IR ← Mem[PC]
  - NPC ← PC+4
- ID stage:
  - A ← Reg[rs]
  - B ← Reg[rt]
  - Imm ← sign-extended immediate field of IR
Pipeline Implementation

- **EX stage**
  - Memory reference
    ALUOutput ← A + Imm
  - Register-register ALU
    ALUOutput ← A func B
  - Register-immediate ALU
    ALUOutput ← A op Imm
  - Branch
    ALUOutput ← NPC + (imm <<2)
    Cond ← (A==0)

- **MEM stage**
  - For all instructions
    PC ← NPC
  - Memory reference
    LMD ← Mem[ALUOutput]
    Mem[ALUOutput] ← B
  - Branch
    If (cond) PC ← ALUOutput

Pipeline Implementation

- **WB stage**
  - Register-register ALU/Register-immediate ALU
    Regs[rd] ← ALUOutput
  - Load
    Regs[rt] ← LMD

Unpipelined Implementation

- **IF stage**
  - Instruction Fetch
    PC ← IF

- **ID stage**
  - Instruction Decode
    IF

- **EX stage**
  - Execute
    Branch
    DEX

- **MEM stage**
  - Memory Access
    MEM

- **WB stage**
  - Write Back
    WB

Pipelined Implementation

- **IF/ID**
  - Instruction Fetch
    IF
  - Instruction Decode
    ID

- **EX**
  - Execute
    DEX

- **MEM**
  - Memory Access
    MEM

- **WB**
  - Write Back
    WB
**Pipelined Implementation**

**IF stage**
- IF/ID.IR ← Mem[PC]
- if((EX/MEM.opcode == branch) & EX/MEM.cond) then
  IF/ID.NPC ← EX/MEM.ALUOutput
- else
  IF/ID.NPC ← PC+4

**ID stage**
- ID/EX.A ← Regs[IF/ID.IR[rs]]
- ID/EX.B ← Regs[IF/ID.IR[rt]]
- ID/EX.NPC ← IF/ID.NPC
- ID/EX.IR ← IF/ID.IR
- ID/EX.Imm ← sign-extend(IF/ID.IR[Immediate])

**EX stage**
- EX/MEM.IR ← ID/EX.IR
- Mem reference
  - EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm
  - EX/MEM.B ← ID/EX.B
- Register-register ALU/Register-immediate ALU
  - EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B or ID/EX.A op ID/EX.Imm
- Branch
  - EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm<<2)
  - EX/MEM.Cond ← (ID/EX.A==0)

**MEM stage**
- MEM/WB.IR ← EX/MEM.IR
- ALU instruction
  - MEM/WB.ALUOutput ← EX/MEM.ALUOutput
- Memory reference
  - MEM/WB.LMD ← Mem[EX/MEM.ALUOutput] or Mem[EX/MEM.ALUOutput]

**WB stage**
- Register-register ALU/Register-immediate ALU
  - Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput or
  - Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput
- Load
  - Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD

**Handling Hazards**
- There are no structural hazards in MIPS pipeline
- All data hazards can be detected (and some solved by forwarding) during ID phase
- RAW(Read after Write) – instruction j reads the operand before instruction i writes it
- WAW(Write after Write) – instruction j writes the operand before instruction i writes it
- WAR(Write after Read) – instruction j writes the operand before instruction i reads it
- RAR(Read after Read) is not a hazard
### Data Hazards (Load)

<table>
<thead>
<tr>
<th>Instruction sequence</th>
<th>Hazard detection condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependence</td>
<td>LD R1, 45(R2) DADD R5, R6, R7 OR R9, R6, R7</td>
</tr>
<tr>
<td>Dependence requiring stall</td>
<td>LD R1, 45(R2) DADD R5, R1, R7 DSUB R8, R6, R7 OR R9, R6, R7</td>
</tr>
<tr>
<td>Dependence overcome by forwarding</td>
<td>LD R1, 45(R2) DADD R5, R6, R7 DSUB R8, R1, R7 OR R9, R6, R7</td>
</tr>
</tbody>
</table>

### Data Hazards that Require Stalls

<table>
<thead>
<tr>
<th>Instruction sequence</th>
<th>Hazard detection condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>IF[ID IR][rs] = ID/EX IR[rt]</td>
</tr>
<tr>
<td>Register-register ALU</td>
<td>IF[ID IR][rt] = ID/EX IR[rt]</td>
</tr>
<tr>
<td>Load, store, ALU imm, branch</td>
<td>IF[ID IR][rt] = ID/EX IR[rt]</td>
</tr>
</tbody>
</table>

### Data Hazards with Forwarding

<table>
<thead>
<tr>
<th>Instruction sequence</th>
<th>Hazard detection condition and action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-register ALU</td>
<td>DADD R1, R2, R3 EX/EM IR[rd] = ID/EX IR[rt]</td>
</tr>
<tr>
<td>Register-immediate ALU</td>
<td>DADD R5, R1, R7 Top ALU op= EX/EM ALU output</td>
</tr>
<tr>
<td></td>
<td>DSUB R8, R6, R7 Bottom ALU op= EX/EM ALU output</td>
</tr>
<tr>
<td></td>
<td>OR R9, R6, R7</td>
</tr>
<tr>
<td>Register-register ALU</td>
<td>DADD R1, R2, R3 MEM/WB IR[rd] = ID/EX IR[rt]</td>
</tr>
<tr>
<td>Register-immediate ALU</td>
<td>DADD R5, R3, R7 Top ALU op= MEM/WB ALU output</td>
</tr>
<tr>
<td></td>
<td>DSUB R8, R1, R7 Bottom ALU op= MEM/WB ALU output</td>
</tr>
<tr>
<td></td>
<td>OR R9, R6, R7</td>
</tr>
<tr>
<td>Load</td>
<td>LD R1, 45(R2) MEM/WB IR[rt] = ID/EX IR[rs]</td>
</tr>
<tr>
<td></td>
<td>DADD R5, R6, R7 Top ALU op= MEM/WB LMD</td>
</tr>
<tr>
<td></td>
<td>DSUB R8, R1, R7 Bottom ALU op= MEM/WB LMD</td>
</tr>
<tr>
<td></td>
<td>OR R9, R6, R7</td>
</tr>
</tbody>
</table>

### Control Hazards

- If we only consider BEQZ and BNEZ (BEQ and BNE with R0) we can move comparison to the end of ID stage
- To take advantage of that branch target needs to be computed early
  - We need additional adder
  - Only 1 clock cycle branch penalty
  - ALU followed by a branch on the result will incur stall
Control Hazards

IF stage
IF/ID.IR ← Mem[PC]
If((IF/ID.opcode == branch) & (Regs[IF/ID.IR[6..10] op 0])) then
IF/ID.NPC ← IF/ID.NPC + (IF/ID.IR[16])
else
IF/ID.IR[16..31]#00
else
IF/ID.NPC ← PC+4