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
  - The next instruction needs the data at the beginning of IF stage

Static Techniques to Reduce Branch Penalties

- Treat every branch as not taken
- If taken, turn next instruction into noop

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

<table>
<thead>
<tr>
<th>Branch (taken)</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction +1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction +2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Delayed Branch

From before: DADD R1, R2, R3
If R2=0 then DSUB R4, R5, R6
If R1=0 then OR R7, R8, R9

From target: DADD R1, R2, R3
If R1=0 then OR R7, R8, R9

From fall-through: DADD R1, R2, R3
If R1=0 then OR R7, R8, R9
Nullifying Branches

- 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

\[
\text{Speedup} = \frac{n}{1 + \text{Stall}_\text{cycles per instruction}}
\]

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

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

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

<table>
<thead>
<tr>
<th>Branch Type</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>3</td>
</tr>
<tr>
<td>Predict taken</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Predict untaken</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Pipeline Implementation

- Assume branch target is calculated in EX stage
- Consider only BEQZ branches
- IF stage
  - IR ← Mem[PC]
  - NPC ← PC+4
- ID stage
  - A ← Reg[s]
  - B ← Reg[t]
  - Imm ← sign-extended immediate field of IR

Pipeline Implementation

- 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

- ALU
- PC
- Reg
- ALU Output
- IM
- IR
- M
- U
- X
- M
- U
- X
- B
- A
- Zero?
- NPC
- Sign extend Imm
- DM
- LMD
- M
- U
- X
- Cond
- opcode

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**
  - For all instructions
    - EX/MEM.IR ← ID/EX.IR
  - Memory 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
    - EX/MEM.ALUOutput ← 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**
  - For all instructions
    - 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] ← EX/MEM.B
Pipelined Implementation

- WB stage
  - Register-register ALU/Register-immediate ALU
    \[ \text{Regs[MEM/WB.IR[rd]]} \rightarrow \text{MEM/WB.ALUOutput} \text{ or } \text{Regs[MEM/WB.IR[rt]]} \rightarrow \text{MEM/WB.ALUOutput} \]
  - Load
    \[ \text{Regs[MEM/WB.IR[rt]]} \rightarrow \text{MEM/WB.LMD} \]

Handling Hazards

- There are no structural hazards in our 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

Load Hazards that Require Stalls

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-register ALU</td>
<td>ID/EX.IR[rt] = ID/EX.IR[rt]</td>
</tr>
<tr>
<td>Register-register ALU</td>
<td>ID/EX.IR[rt] = ID/EX.IR[rt]</td>
</tr>
<tr>
<td>Load store, ALU imm, branch</td>
<td>ID/EX.IR[rt] = ID/EX.IR[rt]</td>
</tr>
</tbody>
</table>

Data Hazards with Forwarding

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-register ALU</td>
<td>ID/EX.IR[rt] = ID/EX.IR[rt]</td>
</tr>
<tr>
<td>Register-register ALU</td>
<td>ID/EX.IR[rt] = ID/EX.IR[rt]</td>
</tr>
<tr>
<td>Load</td>
<td>ID/EX.IR[rt] = ID/EX.IR[rt]</td>
</tr>
</tbody>
</table>

Load Hazards

- No dependence
  - \( \text{LD R1, 45(R2)} \)
  - \( \text{DADD R5, R6, R7} \)
  - \( \text{DSUB R8, R6, R7} \)
  - \( \text{OR R9, R6, R7} \)

- Dependence requiring stall
  - \( \text{LD R1, 45(R2)} \)
  - \( \text{DADD R5, R1, R7} \)
  - \( \text{DSUB R8, R6, R7} \)
  - \( \text{OR R9, R6, R7} \)

- Dependence overcome by forwarding
  - \( \text{LD R1, 45(R2)} \)
  - \( \text{DADD R5, R6, R7} \)
  - \( \text{DSUB R8, R6, R7} \)
  - \( \text{OR R9, R1, R7} \)
Data Hazards with Forwarding

PC Reg ALU Output
IM IR B A
4 NPC Sign extend Imm
DM LMD M U X
Cond M U X
IF IF/ID ID ID/EX EX EX/MEM MEM WB
ALU Output LMD ALU M U X
25 Zero?

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
  \[
  \text{IF/ID.IR} \leftarrow \text{Mem}([\text{PC}]) \\
  \text{IF/ID.IR} \leftarrow \text{IF/ID.IR} \text{op} 0
  \]
  if((IF/ID.opcode == branch) & (Regs[IF/ID.IR 6..10] op 0)) then
  \[
  \text{IF/ID.NPC} \leftarrow \text{IF/ID.NPC} + (\text{IF/ID.IR} 16) \\
  \text{IF/ID.IR} 16..31 \leftarrow 00
  \]
  else
  \[
  \text{IF/ID.NPC} \leftarrow \text{PC} + 4
  \]

Dealing with Exceptions
- Problem arises when instruction \(i+k\) raises an exception, while instruction \(i\) is being executed
- Types of Exceptions:
  - I/O request
  - OS system call
  - Tracing instruction execution
  - Arithmetic overflow
  - Page fault
  - Memory protection violation
  - etc.

Requirements
1. Synchronous vs. asynchronous
2. User requested vs. coerced
3. User maskable vs. nonmaskable
4. Within vs. between instructions
5. Resume vs. terminate
Difficult task is implementing exceptions within instructions that must resume after exception
Stopping and Restarting Execution

- Saving the pipeline state:
  - Force a trap instruction in the pipeline on next IF
  - Until trap is taken turn off all writes for the faulting instruction and other instructions that follow in the pipeline
  - When trap becomes active it saves PC of the faulting instruction, it will be used for return. If there are branches in pipeline, we should save PCs for \( \text{branch\_delay} + 1 \) instructions.
  - If the pipeline can be stopped so that instructions before faulting instruction are completed, and the others can be restarted it is said to have precise exceptions.

Exception Handling in MIPS

- Instruction \( j \) can cause exception before instruction \( i \) does
  - However we must handle exceptions the way we would have handled them without pipelining – first \( i \) and then \( j \)
  - Associate a status vector with the instruction
  - Turn off all writes if bit in status vector is set
  - When the instruction is in WB the status vector is checked and handled.

Instruction Set Complications

- Problem arises when an instruction can alter state early in the pipeline:
  - Upon exception this state change must be undone
  - Instructions that update memory are forced to work on registers, thus the state of partially completed instructions is in registers and can be saved and restored

- If instruction set has very long instructions, pipelining is done at microinstruction level

Extending MIPS for FP Pipelining

- FP operations are long and cannot be completed in 5 cycles (EX lasts more than 1 cycle)
  - Simply imagine that EX stage is duplicated for FP
  - There are multiple FP functional units

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF ID EX MEM WB</th>
<th>FP Instruction i+1</th>
<th>IF ID EX EX EX MEM WB</th>
<th>FP Instruction i+2</th>
<th>IF ID EX MEM WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBL2 R2, R3</td>
<td>1 2 3 4 5 6 7 8 9</td>
<td>MOVC2 @(R1)(R2),74(R2),R3</td>
<td>1 2 3 4 5 6 7 8 9</td>
<td>1 2 3 4 5 6 7 8 9</td>
<td></td>
</tr>
</tbody>
</table>
Hazards

- Because DIV unit is not pipelined structural hazards can occur
- Because instructions have varying running times number of register writes in a cycle can be >1
- Instructions don’t reach WB in order, so WAW hazards are possible
- Instructions can raise exceptions out of order
- Stalls for RAW hazards will be longer due to long latency

RAW Hazards

WAW Hazards

- Although it seems useless sequence of instructions (STORE overwrites F2 immediately after ADD writes it) we must detect WAW hazard and make sure the later value appears in register
- One approach (shown) is to delay issue of later instruction
- Another approach is to stamp the result of ADD and don’t write it into memory

Write Port Structural Hazards

Detecting and Handling Hazards

- Since FP and integer operations use different registers we need only consider moves and loads/stores
- Pipeline checks in ID for:
  - Structural hazards – DIV unit and write port
  - RAW hazards – wait until source registers are not listed as pending destinations
  - WAW hazards – determine if any instruction in EX stage has the same destination register, if so stall the current instruction

 RAW Hazards

 WAW Hazards

 Detecting and Handling Hazards
Maintaining Precise Exception

- Out-of-order completion
- \( \text{DIV.D F0, F2, F4} \)
- \( \text{ADD.D F10, F10, F8} \)
- \( \text{SUB.D F12, F12, F14} \)
- Buffer results of the operation until all previous operations have completed
- History file keeps track of original values of registers
- Future file keeps track of new values, registers are updated when all previous instructions have completed
- Proceed only if sure that no previous instructions will cause exceptions

Homework

- Due Tuesday, October 5 by the end of the class
- Submit either in class (paper) or by E-mail (PS or PDF only) or bring the paper copy to my office
- Do exercises A.1 (assume branch target and condition are calculated in ID stage), A.5