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)

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
Unpipelined Implementation

Pipelined Implementation

Pipelined Implementation

Pipelined Implementation

Pipelined Implementation
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)</td>
</tr>
<tr>
<td></td>
<td>DADD R5, R6, R7</td>
</tr>
<tr>
<td></td>
<td>SUBR8, R6, R7</td>
</tr>
<tr>
<td></td>
<td>OR R9, R6, R7</td>
</tr>
<tr>
<td>Dependence requiring stall</td>
<td>LD R1, 45(R2)</td>
</tr>
<tr>
<td></td>
<td>DADD R5, R1, R7</td>
</tr>
<tr>
<td></td>
<td>SUBR8, R6, R7</td>
</tr>
<tr>
<td></td>
<td>OR R9, R6, R7</td>
</tr>
<tr>
<td>Dependence overcome by forwarding</td>
<td>LD R1, 45(R2)</td>
</tr>
<tr>
<td></td>
<td>DADD R5, R6, R7</td>
</tr>
<tr>
<td></td>
<td>SUBR8, R1, R7</td>
</tr>
<tr>
<td></td>
<td>OR R9, R1, R7</td>
</tr>
<tr>
<td>Dependence with accesses in order</td>
<td>LD R1, 45(R2)</td>
</tr>
<tr>
<td></td>
<td>DADD R5, R6, R7</td>
</tr>
<tr>
<td></td>
<td>SUBR8, R6, R7</td>
</tr>
<tr>
<td></td>
<td>OR R9, R1, 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>Load, 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/MEM.IR[rdr] = ID/EX.IR[rs]</td>
</tr>
<tr>
<td></td>
<td>DADD R5, R1, R7 Top ALU op= EX/MEM.ALUOutput</td>
</tr>
<tr>
<td></td>
<td>SUBR8, R6, R7 Bottom ALU op= EX/MEM.ALUOutput</td>
</tr>
<tr>
<td></td>
<td>OR R9, R6, R7</td>
</tr>
<tr>
<td>Register-immediate ALU</td>
<td>DADD R1, R2, R3 MEM.WB.IR[rd] = ID/EX.IR[rs]</td>
</tr>
<tr>
<td></td>
<td>DADD R5, R3, R7 Top ALU op= MEM.WB.ALUOutput</td>
</tr>
<tr>
<td></td>
<td>SUBR8, R1, R7 Bottom ALU op= MEM.WB.ALUOutput</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>SUBR8, 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

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 an 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 BranchDelay+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 \(i\) then \(j\)
  - Associate a status vector with the instruction and set a corresponding bit if instruction has caused an exception
  - Turn off all writes for the instruction if its bit in status vector is set
  - When the instruction is in WB the status vector is checked and exception is 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

Extending MIPS for FP Pipelining
- It would be beneficial to pipeline EX stage for FP
- We define:
  - Latency – number of cycles between the instruction that produces result and instruction that uses the result
  - Initiation interval – number of cycles that must elapse before issuing two operations of a given type

Extending MIPS for FP Pipelining
- 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
- It would be beneficial to pipeline EX stage for FP
- We define:
  - Latency – number of cycles between the instruction that produces result and instruction that uses the result
  - Initiation interval – number of cycles that must elapse before issuing two operations of a given type

<table>
<thead>
<tr>
<th>Functional unit</th>
<th>Latency</th>
<th>Initiation interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ALU</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Data memory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP/int multiply</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>FP/int divide</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>
Hazards In FP MIPS Pipeline

- 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 In FP MIPS Pipeline

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL.D</td>
<td>IF</td>
<td>ID</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>M7</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>ADD.D</td>
<td>IF</td>
<td>ID</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

WAW Hazards In FP MIPS Pipeline

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

Write Port Structural Hazards In FP MIPS Pipeline

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL.D</td>
<td>IF</td>
<td>ID</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>M7</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>ADD.D</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data Hazards In FP MIPS Pipeline

- Since FP and integer operations use different registers we need only consider moves and loads/stores as potential source of hazards between FP and integer instructions
- 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

Maintaining Precise Exceptions In FP MIPS Pipeline

- Out-of-order completion is possible
  - DIV.D F6, F2, F4
  - ADD.D F10, F10, F8
  - SUB.D F12, F12, F14
- Option 1: History file keeps track of original values of registers/memory
- Option 2: Future file keeps track of new values, registers/memory are updated when all previous instructions have completed
- Option 3: Proceed only if sure that no previous instructions will cause exceptions