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>6</td>
<td>1</td>
</tr>
<tr>
<td>Integer ALU</td>
<td>24</td>
<td>2</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

WAW Hazards In FP MIPS Pipeline

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

RAW Hazards In FP MIPS Pipeline

- Detect write port hazard in ID stage, use shift register that indicates when already issued instructions will use write port, shift reservation register one bit at each clock cycle. We could then insert stalls right after ID stage.
- Alternative solution would insert stall before MEM or WB
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 F0, 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

Instruction Level Parallelism
- Amount of parallelism within a basic block is very small
  - We must exploit parallelism across multiple basic blocks
- Pipelining
- Out-of-order execution

Dependencies
- If two instructions are independent then they can be executed in parallel
  - Otherwise they must execute in order, although they may partially overlap
- Types of dependencies:
  - Data (true) dependencies
  - Name dependencies
  - Control dependencies

Data Dependencies
- Instructions \( j \) is data dependent on instruction \( i \) if
  - Instruction \( i \) produces a result that may be used by instruction \( j \)
  - Instruction \( j \) is data dependent on instruction \( k \) and instruction \( k \) is data dependent on instruction \( i \)
- LOOP:
  ```
  L.D F0, 0(R1)
  ADD.D F4, F0, F2
  S.D F4, 0(R1)
  DADDUI R1, R1, #8
  BNE R1, R2, LOOP
  ```
  What effect do we get if we move branch condition test to EX phase?
  Is this RAW, WAW or WAR hazard?

Data Dependencies
- Data dependencies can be overcome by
  - Leaving the dependence but avoiding the hazard
  - Eliminating the dependence by transforming the code
Name Dependencies

- Instructions \( i \) and \( j \) use the same register or memory location
  - Antidependence – instruction \( j \) writes a location that instruction \( i \) reads
  - Output dependence – instruction \( j \) writes a location that instruction \( i \) writes
  - Since there is no data flow between instructions, they can be renamed and executed in parallel

Control Dependencies

- Branches incur some penalty – while the target and condition are evaluated we cannot be sure which instruction is next
  - We have to guess
  - We have to reorder instructions so that we execute useful instructions while waiting for the branch
  - Main goal is not to affect correctness of the program

Control Dependencies

- Preserve data flow and exception behavior
  - Instructions after the branch depend on it and all instructions prior to the branch for correct execution
    
    DADDU R1, R2, R3
    BEQZ R4, L
    DSUBU R1, R5, R6
    L: ...
    OR R7, R1, R8
  - Instruction reordering should not cause exception reordering
    
    DADDU R2, R3, R4
    BEQZ R2, L
    LW R1, 0(R2)
    L:

Dynamic Scheduling

- Techniques we have learned so far are static scheduling techniques – forwarding, delayed branches, flush pipeline, predict taken, predict untaken
  - Compiler detects dependencies and schedules instruction execution to minimize hazards
  - Pipeline executes instructions in order, detects hazards and inserts stalls
  - Dynamic scheduling overcomes data hazards by out-of-order execution

Out-of-Order Execution

- If some instruction is stalled, check the following instructions to see whether they can proceed (they have no hazards with previous instructions)
- Check for structural and data hazards
- Instruction can be issued as soon as its operands are available
  - Out-of-order issue means out-of-order completion and possibility of WAR and WAW hazards, and problems with exception handling