Floating Point/Multicycle Pipelining in MIPS

• Completion of MIPS EX stage floating point arithmetic operations in one or two cycles is impractical since it requires:
  • A much longer CPU clock cycle, and/or
  • An enormous amount of logic.

• Instead, the floating-point pipeline will allow for a longer latency.
• Floating-point operations have the same pipeline stages as the integer instructions with the following differences:
  – The EX cycle may be repeated as many times as needed.
  – There may be multiple floating-point functional units.
  – A stall will occur if the instruction to be issued either causes a structural hazard for the functional unit or cause a data hazard.

• The latency of functional units is defined as the number of intervening cycles between an instruction producing the result and the instruction that uses the result (usually equals stall cycles with forwarding used).

• The initiation or repeat interval is the number of cycles that must elapse between issuing an instruction of a given type.
Extending The MIPS Pipeline to Handle Floating-Point Operations:

Adding Non-Pipelined Floating Point Units

The DLX pipeline with three additional unpipelined, floating-point, functional units.

(In Appendix A)
Extending The MIPS Pipeline: Multiple Outstanding Floating Point Operations

Latency = 6  
Initiation Interval = 1  
Pipelined

Latency = 0  
Initiation Interval = 1

Latency = 3  
Initiation Interval = 1  
Pipelined

Latency = 24  
Initiation Interval = 25  
Non-pipelined

Hazards:  
RAW, WAW possible  
WAR Not Possible  
Structural: Possible  
Control: Possible

A pipeline that supports multiple outstanding FP operations.

(In Appendix A)
## Latencies and Initiation Intervals For Functional Units

<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 (Integer and FP Loads)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP multiply (also integer multiply)</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FP divide (also integer divide)</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>

Latency usually equals stall cycles when full forwarding is used

(In Appendix A)
Pipeline Characteristics With FP

- Instructions are still processed in-order in IF, ID, EX at the rate of instruction per cycle.
- Longer RAW hazard stalls likely due to long FP latencies.
- Structural hazards possible due to varying instruction times and FP latencies:
  - FP unit may not be available; divide in this case.
  - MEM, WB reached by several instructions simultaneously.
- WAW hazards can occur since it is possible for instructions to reach WB out-of-order.
- WAR hazards impossible, since register reads occur in-order in ID.
- Instructions are allowed to complete out-of-order requiring special measures to enforce precise exceptions.

(In Appendix A)
### FP Operations Pipeline Timing Example

<table>
<thead>
<tr>
<th></th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
<th>CC 10</th>
<th>CC 11</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MUL.D</strong></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><strong>ADD.D</strong></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><strong>L.D</strong></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><strong>S.D</strong></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>

All above instructions are assumed independent

(In Appendix A)
FP Code RAW Hazard Stalls Example
(with full data forwarding in place)

6 stall cycles which equals latency of FP add functional unit

Third stall due to structural hazard in MEM stage

(In Appendix A)
FP Code Structural Hazards Example

MULTD F0, F4, F6

ADDD F2, F4, F6

LD F2, 0(R2)

(In Appendix A)
Maintaining Precise Exceptions in Multicycle Pipelining

- In the MIPS code segment:
  
  \[
  \begin{align*}
  &\text{DIV.D } F0, F2, F4 \\
  &\text{ADD.D } F10, F10, F8 \\
  &\text{SUB.D } F12, F12, F14
  \end{align*}
  \]

- The ADD.D, SUB.D instructions can complete before DIV.D is completed causing out-of-order execution completion.

- If SUB.D causes a floating-point arithmetic exception it may prevent DIV.D from completing and draining the floating-point may not be possible causing an imprecise exception.

- Four approaches have been proposed to remedy this type of situation:
  
  1. Ignore the problem and settle for imprecise exception.
  2. Buffer the results of the operation until all the operations issues earlier are done. (large buffers, multiplexers, comparators)
  3. A history file keeps track of the original values of registers (CYBER180/190, VAX)
  4. A Future file keeps the newer value of a register; when all earlier instructions have completed the main register file is updated from the future file. On an exception the main register file has the precise values for the interrupted state.

(In Appendix A)
MIPS FP SPEC92
Floating Point Stalls
Per FP Operation

Stalls per FP operation for each major type of FP operation.

FP SPEC benchmarks
- doduc
- ear
- hydro2d
- mdijdp
- su2cor

Number of stalls

- Add/subtract/convert
- Compares
- Multiply
- Divide
- Divide structural
MIPS FP SPEC89 Floating Point Stalls

The stalls occurring for the MIPS FP pipeline for five of the SPEC89 FP benchmarks
Pipelining and Exploiting Instruction-Level Parallelism (ILP)

- Pipelining increases performance by overlapping the execution of independent instructions.

- The CPI of a real-life pipeline is given by:

\[
\text{Pipeline CPI} = \text{Ideal Pipeline CPI} + \text{Structural Stalls} + \text{RAW Stalls} + \text{WAR Stalls} + \text{WAW Stalls} + \text{Control Stalls}
\]

- A *basic instruction block* is a straight-line code sequence with no branches in, except at the entry point, and no branches out except at the exit point of the sequence.

- The amount of parallelism in a basic block is limited by instruction dependence present and size of the basic block.

- In typical integer code, dynamic branch frequency is about 15% (average basic block size of 7 instructions).

(In Chapter 3.1)
Increasing Instruction-Level Parallelism

- A common way to increase parallelism among instructions is to exploit parallelism among iterations of a loop (i.e., Loop Level Parallelism, LLP).
- This is accomplished by unrolling the loop either statically by the compiler, or dynamically by hardware, which increases the size of the basic block present.
- In this loop every iteration can overlap with any other iteration. Overlap within each iteration is minimal.

```
for (i=1; i<=1000; i=i+1;)
    x[i] = x[i] + y[i];
```

- In vector machines, utilizing vector instructions is an important alternative to exploit loop-level parallelism,
- Vector instructions operate on a number of data items. The above loop would require just four such instructions.

(In Chapter 4.1)
MIPS Loop Unrolling Example

- For the loop:

\[
\text{for (i=1000; i>0; i=i-1) } \\
\quad x[i] = x[i] + s;
\]

The straightforward MIPS assembly code is given by:

```
Loop:  L.D             F0, 0 (R1)           ;F0=array element
       ADD.D        F4, F0, F2           ;add scalar in F2
       S.D               F4, 0(R1)            ;store result
       DADDUI     R1, R1, # -8        ;decrement pointer 8 bytes
       BNE             R1, R2,Loop      ;branch R1!=R2
```

R1 is initially the address of the element with highest address.
8(R2) is the address of the last element to operate on.

(In Chapter 4.1)
DLX FP Latency Assumptions Used In Chapter 4

- All FP units assumed to be pipelined.
- The following FP operations latencies are used:

<table>
<thead>
<tr>
<th>Instruction Producing Result</th>
<th>Instruction Using Result</th>
<th>Latency In Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU Op</td>
<td>Another FP ALU Op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU Op</td>
<td>Store Double</td>
<td>2</td>
</tr>
<tr>
<td>Load Double</td>
<td>FP ALU Op</td>
<td>1</td>
</tr>
<tr>
<td>Load Double</td>
<td>Store Double</td>
<td>0</td>
</tr>
</tbody>
</table>

(In Chapter 4.1)
Loop Unrolling Example (continued)

• This loop code is executed on the MIPS pipeline as follows:

<table>
<thead>
<tr>
<th></th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No scheduling</strong></td>
<td></td>
</tr>
<tr>
<td>Loop: L.D F0, 0(R1)</td>
<td>1</td>
</tr>
<tr>
<td>stall</td>
<td>2</td>
</tr>
<tr>
<td>ADD.D F4, F0, F2</td>
<td>3</td>
</tr>
<tr>
<td>stall</td>
<td>4</td>
</tr>
<tr>
<td>stall</td>
<td>5</td>
</tr>
<tr>
<td>S.D F4, 0 (R1)</td>
<td>6</td>
</tr>
<tr>
<td>DADDUI R1, R1, # -8</td>
<td>7</td>
</tr>
<tr>
<td>stall</td>
<td>8</td>
</tr>
<tr>
<td>BNE R1,R2, Loop</td>
<td>9</td>
</tr>
<tr>
<td>stall</td>
<td>10</td>
</tr>
</tbody>
</table>

|                     |             |
| **With delayed branch scheduling** |             |
| Loop: L.D F0, 0(R1)|             |
| DADDUI R1, R1, # -8|             |
| ADD.D F4, F0, F2   |             |
| stall               |             |
| BNE R1,R2, Loop    |             |
| S.D F4,8(R1)       |             |

6 cycles per iteration

10/6 = 1.7 times faster

(In Chapter 4.1)
Loop Unrolling Example (continued)

- The resulting loop code when four copies of the loop body are unrolled without reuse of registers:

  No scheduling
  Loop:

    L.D      F0, 0(R1)
    ADD.D    F4, F0, F2
    SD       F4,0 (R1) ; drop DADDUI & BNE
    LD       F6, -8(R1)
    ADDD     F8, F6, F2
    SD       F8, -8 (R1), ; drop DADDUI & BNE
    LD       F10, -16(R1)
    ADDD     F12, F10, F2
    SD       F12, -16 (R1) ; drop DADDUI & BNE
    LD       F14, -24 (R1)
    ADDD     F16, F14, F2
    SD       F16, -24(R1)
    DADDUI   R1, R1, # -32
    BNE      R1, R2, Loop

  Three branches and three decrements of R1 are eliminated.
  Load and store addresses are changed to allow DADDUI instructions to be merged.
  The loop runs in 28 assuming each L.D has 1 stall cycle, each ADD.D has 2 stall cycles, the DADDUI 1 stall, the branch 1 stall cycles, or 7 cycles for each of the four elements.

(In Chapter 4.1)
When scheduled for pipeline

Loop:

L.D F0, 0(R1)
L.D F6,-8 (R1)
L.D F10, -16(R1)
L.D F14, -24(R1)
ADD.D F4, F0, F2
ADD.D F8, F6, F2
ADD.D F12, F10, F2
ADD.D F16, F14, F2
S.D F4, 0(R1)
S.D F8, -8(R1)
DADDUI R1, R1,# -32
S.D F12, -16(R1),F12
BNE R1,R2, Loop
S.D F16, 8(R1), F16 ;8-32 = -24

The execution time of the loop has dropped to 14 cycles, or 3.5 clock cycles per element compared to 6.8 before scheduling and 6 when scheduled but unrolled.

Unrolling the loop exposed more computation that can be scheduled to minimize stalls.
Loop Unrolling Requirements

• In the loop unrolling example, the following guidelines were followed:
  – Determine that it was legal to move S.D after DADDUI and BNE; find the S.D offset.
  – Determine that unrolling the loop would be useful by finding that the loop iterations were independent.
  – Use different registers to avoid constraints of using the same registers (WAR, WAW).
  – Eliminate extra tests and branches and adjust loop maintenance code.
  – Determine that loads and stores can be interchanged by observing that they are independent from different loops.
  – Schedule the code, preserving any dependencies needed to give the same result as the original code.

(In Chapter 4.1)
Instruction Dependencies

• Determining instruction dependencies is important for pipeline scheduling and to determine the amount of parallelism in the program to be exploited.

• If two instructions are parallel, they can be executed simultaneously in the pipeline without causing stalls; assuming the pipeline has sufficient resources.

• Instructions that are dependent are not parallel and cannot be reordered.

• Instruction dependencies are classified as:
  – Data dependencies
  – Name dependencies
  – Control dependencies

(In Chapter 3.1)
Instruction Data Dependencies

• An instruction $j$ is data dependent on another instruction $i$ if:

  – Instruction $i$ produces a result used by instruction $j$, resulting in a direct RAW hazard, or
  – Instruction $j$ is data dependent on instruction $k$ and instruction $k$ is data dependent on instruction $i$ which implies a chain of RAW hazard between the two instructions.

Example: The arrows indicate data dependencies and point to the dependent instruction which must follow and remain in the original instruction order to ensure correct execution.

Loop:  

L.D F0, 0 (R1) ; F0=array element
ADD.D F4, F0, F2 ; add scalar in F2
S.D F4, 0 (R1) ; store result

(In Chapter 3.1)
Instruction Name Dependencies

- A name dependence occurs when two instructions use the same register or memory location, called a name.

- No flow of data exist between the instructions involved in the name dependency.

- If instruction $i$ precedes instruction $j$ then two types of name dependencies can occur:
  - An antidependence occurs when $j$ writes to a register or memory location and $i$ reads and instruction $i$ is executed first. This corresponds to a WAR hazard.
  - An output dependence occurs when instruction $i$ and $j$ write to the same register or memory location resulting in a WAW hazard and instruction execution order must be observed.

(In Chapter 3.1)
### Name Dependence Example

In the unrolled loop, using the same registers results in name (green) and data tendencies (red):

<table>
<thead>
<tr>
<th>Loop: L.D</th>
<th>F0, 0 (R1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>S.D</td>
<td>✔️, 0(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F0, -8(R1)</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F4, -8(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F0, -16(R1)</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F4, -16(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F0, -24(R1)</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F4, -24(R1)</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1, # -32</td>
</tr>
<tr>
<td>BNE</td>
<td>R1, R2, Loop</td>
</tr>
</tbody>
</table>

Renaming the registers used for each copy of the loop body are renamed, only true dependencies remain:

<table>
<thead>
<tr>
<th>Loop: L.D</th>
<th>F0, 0(R1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>S.D</td>
<td>✔️, 0(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F0, -8(R1)</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F0, -8(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F0, -16(R1)</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F0, -16(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F0, -24(R1)</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F0, -24(R1)</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1, # -32</td>
</tr>
<tr>
<td>BNE</td>
<td>R1, R2, Loop</td>
</tr>
</tbody>
</table>

(In Chapter 4.1)
Control Dependencies

- Determines the ordering of an instruction with respect to a branch instruction.
- Every instruction except in the first basic block of the program is control dependent on some set of branches.
- An instruction which is control dependent on a branch cannot be moved before the branch.
- An instruction which is not control dependent on the branch cannot be moved so that its execution is controlled by the branch (in the then portion).
- It’s possible in some cases to violate these constraints and still have correct execution.
- Example of control dependence in the then part of an if statement:

```c
if p1 {
    S1; // S1 is control dependent on p1
};
if p2 {
    S2; // S2 is control dependent on p2 but not on p1
}
```

(In Chapter 3.1)
The unrolled loop code with the branches still in place is shown here.

Branch conditions are complemented here to allow the fall-through to execute another loop.

BEQZ instructions prevent the overlapping of iterations for scheduling optimizations.

Moving the instructions requires a change in the control dependencies present.

Removing the branches changes the control dependencies present and makes optimizations possible.