Pipelining and Exploiting Instruction-Level Parallelism (ILP)

- Definition of basic instruction blocks
- Increasing Instruction-Level Parallelism & Size of Basic Blocks
  - Using Loop Unrolling
- MIPS Loop Unrolling Example.
- Loop Unrolling Requirements.
- Classification of Instruction Dependencies
  - Data dependencies
  - Name dependencies
  - Control dependencies

(In Chapter 3.1, 4.1)
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 (assuming ideal memory):

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

• *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. Example: Body of a loop.

• 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% (resulting average basic block size of about 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. This resulting larger basic block provides more instructions that can scheduled or re-ordered to eliminate more stall cycles.

• In this loop every iteration can overlap with any other iteration. Overlap within each iteration is minimal.

```c
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:

```c
for (i=1000; i>0; i=i-1)
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.
```

(Basic block size = 5 instructions)
MIPS FP Latency Assumptions Used In Chapter 4.1

- 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>

Branch resolved in decode stage, Branch penalty = 1 cycle, Full forwarding is used
Loop Unrolling Example (continued)

- This loop code is executed on the MIPS pipeline as follows:
  (Branch resolved in decode stage, Branch penalty = 1 cycle, Full forwarding is used)

No scheduling

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>Scheduled with single delayed branch slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F0, 0(R1)</td>
</tr>
<tr>
<td>stall</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>stall</td>
<td></td>
</tr>
<tr>
<td>stall</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F4, 0 (R1)</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1, # -8</td>
</tr>
<tr>
<td>stall</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>R1,R2, Loop</td>
</tr>
<tr>
<td>stall</td>
<td></td>
</tr>
</tbody>
</table>

6 cycles per iteration

10 cycles per iteration

10/6 = 1.7 times faster
Loop Unrolling Example (continued)

- The resulting loop code when four copies of the loop body are unrolled without reuse of registers.
- The size of the basic block increased from 5 instructions in the original loop to 14 instructions.

Three branches and three decrements of R1 are eliminated.

Load and store addresses are changed to allow DADDUI instructions to be merged.

The unrolled loop runs in 28 cycles assuming each L.D has 1 stall cycle, each ADD.D has 2 stall cycles, the DADDUI 1 stall, the branch 1 stall cycle, or 28/4 = 7 cycles to produce each of the four elements.

(In Chapter 4.1)
Loop Unrolling Example (continued)

When scheduled for pipeline

Loop:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0, 0(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F6, -8 (R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F10, -16(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F14, -24(R1)</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F8, F6, F2</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F12, F10, F2</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F16, F14, F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F4, 0(R1)</td>
</tr>
<tr>
<td>S.D</td>
<td>F8, -8(R1)</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1,# -32</td>
</tr>
<tr>
<td>S.D</td>
<td>F12, 16(R1), F12</td>
</tr>
<tr>
<td>BNE</td>
<td>R1, R2, Loop</td>
</tr>
<tr>
<td>S.D</td>
<td>F16, 8(R1), F16</td>
</tr>
</tbody>
</table>

The execution time of the loop has dropped to 14 cycles, or $14/4 = 3.5$ clock cycles per element compared to 6.8 before scheduling and 6 when scheduled but unrolled.

Unrolling the loop exposed more computations that can be scheduled to minimize stalls by increasing the size of the basic block from 5 instructions in the original loop to 14 instructions in the unrolled loop.

(In Chapter 4.1)
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 are 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 (no data hazards); 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
**Data Dependence**

- Instruction $i$ precedes instruction $j$ in the program sequence or order.
- Instruction $i$ produces a result used by instruction $j$.
  - Then instruction $j$ is said to be data dependent on instruction $i$.
- Changing the relative execution order of $i$, $j$ violates this data dependence and results in RAW hazard and incorrect execution.

**Diagram:**

- **I (Write)**
- **J (Read)**
- **Shared Operand**
- **Dependency Graph Representation**
- **I**
- **J**

J data dependent on I resulting in a Read after Write (RAW) if their relative execution order in changed.
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 anti-dependence 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 Classification: Anti-Dependence

- Instruction $i$ precedes instruction $j$ in the program sequence or order
- Instruction $i$ reads a value from a name (register or memory location)
- Instruction $j$ writes a value to the same a name (same register or memory location read by $i$)
  - Then instruction $j$ is said to be anti-dependent on instruction $i$
- Changing the relative execution order of $i$, $j$ violates this name dependence and results in a WAR hazard and incorrect execution.

J is anti-dependent on I resulting in a Write after Read (WAR) if their relative execution order in changed
Name Dependence Classification: Output (or Write) Dependence

- Instruction $i$ precedes instruction $j$ in the program sequence or order.
- Both instructions $i$, $j$ write to the same name (same register or memory location).
  - Then instruction $j$ is said to be output-dependent on instruction $i$.
- Changing the relative execution order of $i$, $j$ violates this name dependence and results in a WAW hazard and incorrect execution.

Dependency Graph Representation:

$I$ (Write) \[\rightarrow\] Shared Name \[\rightarrow\] $J$ (Write)

$J$ is output-dependent on $I$ resulting a Write after Write (WAW) if their relative execution order in changed.
Instruction Dependence Example

- For the following code identify all data and name dependence between instructions and give the dependency graph

1. L.D  F0, 0 (R1)
2. ADD.D  F4, F0, F2
3. S.D  F4, 0(R1)
4. L.D  F0, -8(R1)
5. ADD.D  F4, F0, F2
6. S.D  F4, -8(R1)

**True Data Dependence:**
Instruction 2  depends on instruction 1  (instruction 1 result in F0 used by instruction 2),  Similarly,  instructions (4,5)
Instruction 3  depends on instruction 2  (instruction 2 result in F4 used by instruction 3)  Similarly,  instructions (5,6)

**Name Dependence**

**Output Name Dependence (WAW):**
Instruction 1  has an output name dependence (WAW) over result register (name)  F0  with instructions  4
Instruction 2  has an output name dependence (WAW) over result register (name)  F4  with instructions  5

**Anti-dependence (WAR):**
Instruction 2  has an anti-dependence with instruction 4  over register (name)  F0  which is an operand of instruction 1 and the result of instruction 4
Instruction 3  has an anti-dependence with instruction 5  over register (name)  F4  which is an operand of instruction 3 and the result of instruction 5
Can instruction 3 (first S.D) be moved just after instruction 4 (second L.D)?
How about moving 3 after 5 (the second ADD.D)?
If not what dependencies are violated?

Can instruction 4 (second L.D) be moved just after instruction 1 (first L.D)?
If not what dependencies are violated?
# Instruction Dependence Example

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

<table>
<thead>
<tr>
<th>Loop:</th>
<th>L.D</th>
<th>F0, 0 (R1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>2</td>
<td>S.D</td>
<td>F4, 0(R1)</td>
</tr>
<tr>
<td>3</td>
<td>L.D</td>
<td>F0, -8(R1)</td>
</tr>
<tr>
<td>4</td>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>5</td>
<td>S.D</td>
<td>F4, -8(R1)</td>
</tr>
<tr>
<td>6</td>
<td>L.D</td>
<td>F0, -16(R1)</td>
</tr>
<tr>
<td>7</td>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>8</td>
<td>S.D</td>
<td>F4, -16(R1)</td>
</tr>
<tr>
<td>9</td>
<td>L.D</td>
<td>F0, -24(R1)</td>
</tr>
<tr>
<td>10</td>
<td>ADD.D</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>11</td>
<td>S.D</td>
<td>F4, -24(R1)</td>
</tr>
<tr>
<td>12</td>
<td>DADDUI</td>
<td>R1, R1, # -32</td>
</tr>
<tr>
<td>13</td>
<td>BNE</td>
<td>R1, R2, Loop</td>
</tr>
</tbody>
</table>

From The Code to the left:

**True Data Dependence (RAW) Examples:**

Instruction 2 \( \text{ADD.D} \) F4, F0, F2 depends on instruction 1 \( \text{L.D} \) F0, 0 (R1) (instruction 1 result in F0 used by instruction 2).

Similarly, instructions (4,5), (7,8), (10,11).

Instruction 3 \( \text{S.D} \) F4, 0(R1) depends on instruction 2 \( \text{ADD.D} \) F4, F0, F2 (instruction 2 result in F4 used by instruction 3).

Similarly, instructions (5,6), (8,9), (11,12).

**Name Dependence (WAR, WAW) Examples:**

Instruction 1 \( \text{L.D} \) F0, 0 (R1) has an output name dependence (WAW) over result register (name) F0 with instructions 4, 7, 10.

**Output Name Dependence (WAW) Examples:**

Instruction 2 \( \text{ADD.D} \) F4, F0, F2 has an anti-dependence (WAR) with instruction 4 \( \text{L.D} \) F0, 0 (R1) over register (name) F0 which is an operand of instruction 1 and the result of instruction 4.

Similarly, an anti-dependence (WAR) over F0 exists between instructions (5, 7), (8, 10).

(In Chapter 4.1)
Name Dependence Removal

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

| Loop: L.D   | F0, 0 (R1) |
| ADD.D      | F4, F0, F2 |
| S.D        | F4, 0 (R1) |
| L.D        | F0, -8 (R1) |
| ADD.D      | F4, F0, F2 |
| S.D        | F4, -8 (R1) |
| L.D        | F0, -16 (R1) |
| ADD.D      | F4, F0, F2 |
| S.D        | F4, -16 (R1) |
| L.D        | F0, -24 (R1) |
| ADD.D      | F4, F0, F2 |
| S.D        | F4, -24 (R1) |
| DADDUI     | R1, R1, # -32 |
| BNE        | R1, R2, Loop |

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

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

(In Chapter 4.1)
Control Dependencies

- Determines the ordering of an instruction with respect to a branch instruction.
- Every instruction in a program except those in the very 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 so that its execution is no longer controlled by 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:

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

If p2 {
    S2;  // What happens if S1 is moved here?
}
```
Control Dependence Example

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.

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