Pipelining and Exploiting Instruction-Level Parallelism (ILP)

- Pipelining and Instruction-Level Parallelism (ILP).
- Definition of basic instruction block
- Increasing Instruction-Level Parallelism (ILP) & Size of Basic Block:
  - Using Loop Unrolling
- MIPS Loop Unrolling Example.
- Loop Unrolling Requirements.
- Classification of Instruction Dependencies
  - Data dependencies
  - Name dependencies
  - Control dependencies

In Fourth Edition: Chapter 2.1, 2.2
(In Third Edition: Chapter 3.1, 4.1)

Dependency Analysis
Dependency Graphs

Pipeline Hazard Condition = Dependency Violation

Static = At compilation time    Dynamic = At run time
Pipelining and Exploiting Instruction-Level Parallelism (ILP)

• Instruction-Level Parallelism (ILP) exists when instructions in a sequence are independent and thus can be executed in parallel by overlapping. (without stalling)
  – Pipelining increases performance by overlapping the execution of independent instructions and thus exploits ILP in the code.

• Preventing instruction dependency violations (hazards) may result in stall cycles in a pipelined CPU increasing its CPI (reducing performance).
  – The CPI of a real-life pipeline is given by (assuming ideal memory):

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

• Programs that have more ILP (fewer dependencies) tend to perform better on pipelined CPUs.
  – More ILP mean fewer instruction dependencies and thus fewer stall cycles needed to prevent instruction dependency violations i.e hazards

\[
T = I \times CPI \times C
\]

In Fourth Edition Chapter 2.1
(In Third Edition Chapter 3.1)
Instruction-Level Parallelism (ILP) Example

Given the following two code sequences with three instructions each:

**Higher ILP**

1. **ADD.D** F2, F4, F6
2. **ADD.D** F10, F6, F8
3. **ADD.D** F12, F12, F14

The instructions in the first code sequence above have no dependencies between the instructions. Thus the three instructions are said to be independent and can be executed in parallel or in any order (re-ordered). This code sequence is said to have a high degree of ILP.

**Lower ILP**

1. **ADD.D** F2, F4, F6
2. **ADD.D** F10, F2, F8
3. **ADD.D** F12, F10, F2

The instructions in the second code sequence above have three data dependencies among them.

- Instruction 2 depends on instruction 1
- Instruction 3 depends on both instructions 1 and 2

Thus the instructions in the sequence are not independent and cannot be executed in parallel. Thus the three instructions are said to be independent and thus can be executed in parallel and their order cannot be changed with causing incorrect execution. This code sequence is said to have a lower degree of ILP. Possibly resulting in more stalls.

More on dependency analysis and dependency graphs later in the lecture.
Basic Instruction Block

• **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 instruction-level parallelism (ILP) 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).

• Any static technique that increases the average size of basic blocks which increases the amount of exposed ILP in the code and provide more instructions for static pipeline scheduling by the compiler possibly eliminating more stall cycles and thus improves pipelined CPU performance.
  – Loop unrolling is one such technique that we examine next.
Basic Blocks/Dynamic Execution Sequence (Trace) Example

Static Program Order

- A-O = Basic Blocks terminating with conditional branches
- The outcomes of branches determine the basic block dynamic execution sequence, path or trace

Trace: Dynamic Sequence of basic blocks executed

Program Control Flow Graph (CFG)

If all three branches are taken the execution trace will be basic blocks: ACGO

Average Basic Block Size = 5-7 instructions

Type of branches in this example: “If-Then-Else” branches (not loops)
Increasing Instruction-Level Parallelism (ILP)

- 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 be scheduled or re-ordered by the compiler to eliminate more stall cycles.
- In this loop every iteration can overlap with any other iteration. Overlap within each iteration is minimal.

Example:

```
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 Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)
MIPS Loop Unrolling Example

For the loop:

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

The straightforward MIPS assembly code is given by:

1. Loop: L.D F0, 0 (R1) ; F0 = array element
2. ADD.D F4, F0, F2 ; add scalar in F2 (constant)
3. S.D F4, 0(R1) ; store result
4. DADDUI R1, R1, # -8 ; decrement pointer 8 bytes
5. 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.

\[X[], \text{array of double-precision floating-point numbers (8-bytes each)}\]

Initial value of R1 = R2 + 8000

Basic block size = 5 instructions
MIPS FP Latency Assumptions Used
In Chapter 2.2

- 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>

Other Assumptions:
- Branch resolved in decode stage, Branch penalty = 1 cycle
- Full forwarding is used
- Single Branch delay Slot
- Some potential structural hazards ignored

i.e. MIPS Pipeline Version #3
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 (as is)  
(Resulting stalls shown)  
Clock cycle

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

6 cycles per iteration

Scheduled with single delayed branch slot:

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)  
S.D in branch delay slot

6 cycles per iteration

10 cycles per iteration

10/6 = 1.7 times faster

- Ignoring Pipeline Fill Cycles
- No Structural Hazards

In Fourth Edition Chapter 2.2
(In Third Edition Chapter 4.1)

CMPE550 - Shaaban
#9 Spring 2017 lec3 2-8-2017
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.

### New Basic Block Size = 14 Instructions

#### Loop: No scheduling

1. **Loop:**
   1. **L.D** F0, 0(R1)
   2. **Stall**
   3. **ADD.D** F4, F0, F2
   4. **Stall**
   5. **Stall**
   6. **SD** F4,0 (R1) ; drop DADDUI & BNE
   7. **LD** F6, -8(R1)
   8. **Stall**
   9. **ADDD** F8, F6, F2
   10. **Stall**
   11. **Stall**
   12. **SD** F8, -8 (R1), ; drop DADDUI & BNE
   13. **LD** F10, -16(R1)
   14. **Stall**
   15. **ADDD** F12, F10, F2
   16. **Stall**
   17. **Stall**
   18. **SD** F12, -16 (R1) ; drop DADDUI & BNE
   19. **LD** F14, -24 (R1)
   20. **Stall**
   21. **ADDD** F16, F14, F2
   22. **Stall**
   23. **Stall**
   24. **SD** F16, -24(R1)
   25. **DADDUI** R1, R1, # -32
   26. **Stall**
   27. **BNE** R1, R2, Loop

#### Loop unrolled 4 times

- Three branches and three decrements of R1 are eliminated.
- Load and store addresses are changed to allow DADDUI instructions to be merged.

#### Performance:

- 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.
- 28/4 = 7 Cycles per original iteration
- i.e. 7 cycles for each original iteration

#### Empty Branch Delay Slot

- In Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)

- Note use of different registers for each iteration (register renaming)
Loop Unrolling Example (continued)

When scheduled for pipeline

Loop:

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

Note: No stalls

Basic Block size = 14 instructions vs. 5 (no unrolling)

The execution time of the loop has dropped to 14 cycles, or 14/4 = 3.5 clock cycles per element compared to 7 before scheduling and 6 when scheduled but unrolled.

Speedup = 6/3.5 = 1.7

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.

Larger Basic Block \(\rightarrow\) More ILP

Offset = 16 - 32 = -16

14/4 = 3.5 Cycles per original iteration

In Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)
Loop Unrolling Benefits & Requirements

• Loop unrolling improves performance in two ways:
  1. Larger basic block size: More instructions to schedule and thus possibly more stall cycles are eliminated. More ILP exposed due to larger basic block
  2. Fewer instructions executed: Fewer branches and loop maintenance instructions executed

• From the loop unrolling example, the following guidelines where followed:
  – Determine that unrolling the loop would be “most” useful by finding that the loop iterations where independent.
  – Determine that it was legal to move S.D after DADDUI and BNE; find the correct S.D offset.
  – Use different registers (rename registers) to avoid constraints of using the same registers (WAR, WAW). More registers are needed.
  – 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 Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)
Instruction Dependencies

• Determining instruction dependencies (dependency analysis) is important for pipeline scheduling and to determine the amount of instruction level parallelism (ILP) in the program to be exploited.

• Instruction Dependency Graph: A directed graph where graph nodes represent instructions and graph edges represent instruction dependencies.

• If two instructions are independent or parallel (no dependencies between them exist), they can be executed simultaneously in the pipeline without causing stalls (no pipeline hazards); assuming the pipeline has sufficient resources (no hardware hazards).

• Instructions that are dependent are not parallel and cannot be reordered by the compiler or hardware. Otherwise incorrect execution results

• Instruction dependencies are classified as:
  • Data dependencies (or Flow)
  • Name dependencies (two types: anti-dependence and write dependence)
  • Control dependencies

Name: Register Name or Named Memory Location

Pipeline Hazard = Dependency Violation

In Fourth Edition Chapter 2.1 (In Third Edition Chapter 3.1)
(True) 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 a RAW hazard and incorrect execution.

**Producer**

$I$ (Write)

- e.g. ADD.D F2, F1, F0
- e.g. ADD.D F8, F2, F9

**Shared Operand**

**Consumer**

$J$ (Read)

- $J$ data dependent on $I$
- resulting in a Read after Write (RAW) hazard if their relative execution order is changed

**Dependency Graph Representation**

Also called: Data Flow Dependence or just Flow Dependence

- e.g. ADD.D F2, F1, F0
- e.g. ADD.D F8, F2, F9

**CMPE550 - Shaaban**

#14 Spring 2017 lec#3 2-8-2017
Instruction Data Dependencies
(or Flow)

Given two instructions $i, j$ where $i$ precedes $j$ in program order:

- Instruction $j$ is data dependent on instruction $i$ if:
  - Instruction $i$ produces a result used by instruction $j$, resulting in a direct RAW hazard if their order is not maintained, or
  - Instruction $j$ is data dependent on instruction $k$ and instruction $k$ is data dependent on instruction $i$ which implies a chain of data dependencies between the 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.

```
1. i: L.D F0, 0 (R1); F0=array element
2. k: ADD.D F4, F0, F2; add scalar in F2
3. j: S.D F4,0 (R1); store result
```

In Fourth Edition Chapter 2.1 (In Third Edition Chapter 3.1)
Instruction Name Dependencies

• A name dependence occurs when two instructions use (share or have a conflict over) the same register or memory location, called a name.

• No flow of data exist between the instructions involved in the name dependency (i.e. no producer/consumer relationship)

• If instruction $i$ precedes instruction $j$ in program order then two types of name dependencies can exist:

  The Two Types of Name Dependence:

  - An **anti-dependence** exists when $j$ writes to the same register or memory location that instruction $i$ reads
    - **Anti-dependence violation:** Relative read/write order is changed
      - This results in a **WAR** hazard and thus the relative instruction read/write and execution order must be preserved.

  - An **output or (write) dependence** exists when instruction $i$ and $j$ write to the same register or memory location (i.e. the same name)
    - **Output-dependence violation:** Relative write order is changed
      - This results in a **WAW** hazard and thus instruction write and execution order must be preserved.

In Fourth Edition Chapter 2.1 (In Third Edition Chapter 3.1)
Name Dependence Classification: **Anti-Dependence**

- Instruction $i$ precedes instruction $j$ in the program sequence or order $I \rightarrow J$
- Instruction $i$ reads a value from a name (register or memory location)
- Instruction $j$ writes a value to the same 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.
- This name dependence can be eliminated by “renaming” the shared name.

**Dependency Graph Representation**

$I$ (Read)  

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

$J$ (Write)  

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Name: Register or Memory Location

- $I$ reads a value from a name (register or memory location)
- $J$ writes a value to the same name (same register or memory location read by $I$)

**i.e. Anti-dependence violation = WAR Hazard**

**Program Order**

- $I$ precedes $J$ in the program sequence or order

- **CMPE550 - Shaaban**
Name Dependence Classification:
**Output (or Write) Dependence**  
\[ I \rightarrow J \]

- 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.
- This name dependence can also be eliminated by “renaming” the shared name.

**J is output-dependent on I**
resulting a Write after Write (WAW) hazard if their relative execution order is changed.

Dependency Graph Representation:

- **I**: \( \text{ADD.D F2, F1, F0} \)
- **J**: \( \text{ADD.D F2, F5, F7} \)

**Output dependence**

- **I**: \( \text{ADD.D F2, F1, F0} \)
- **J**: \( \text{ADD.D F2, F5, F7} \)

- **Program Order**

**Name**: Register or Memory Location

- i.e relative order of write by I and write by J

**CMPE550 - Shaaban**

#18 Spring 2017  lec#3  2-8-2017
Dependency Graph Example

For the given MIPS code identify all data and name dependence between instructions and complete the dependency graph.

MIPS Code

1  ADD.D  F2, F1, F0
2  ADD.D  F4, F2, F3
3  ADD.D  F2, F2, F4
4  ADD.D  F4, F2, F6

True Date Dependence:

Output Dependence:

Anti-dependence:

Start with an Empty Graph
Dependency Graph Example

For the given MIPS code identify all data and name dependence between instructions and complete the dependency graph.

**MIPS Code**

1  ADD.D  F2, F1, F0
2  ADD.D  F4, F2, F3
3  ADD.D  F2, F2, F4
4  ADD.D  F4, F2, F6

**True Date Dependence:**

(1, 2) (1, 3) (2, 3) (3, 4)

i.e. 1 → 2  1 → 3
     2 → 3  3 → 4

**Output Dependence:**

(1, 3) (2, 4)

i.e. 1 ↔ 3  2 ↔ 4

**Anti-dependence:**

(2, 3) (3, 4)

i.e. 2 → 3  3 → 4

What happens if we rename F2 to F8 (destination in instruction 3 and operand in instruction 4) and F4 to F10 in instruction 4?
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)

Program Order

Constant S in F2

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
Instruction Dependence Example

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?

What happens if we rename F0 to F6 and F4 to F8 in instructions 4, 5, 6?

Example Code

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)

Date Dependence:
(1, 2)  (2, 3)  (4, 5)  (5, 6)

Output Dependence:
(1, 4)  (2, 5)

Anti-dependence:
(2, 4)  (3, 5)

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, F0 (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>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From The Code to the left:

True Data Dependence (RAW) Examples:

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

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

Instruction 3 S.D F4, 0(R1) depends on instruction 2 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

Output Name Dependence (WAW) Examples:

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

Anti-dependence (WAR) Examples:

Instruction 2 ADD.D F4, F0, F2 has an anti-dependence (WAR) with instruction 4 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 Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)
**Name Dependence Removal**

*Using Register Renaming*

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

(i.e. no register renaming done)

**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

(Name dependencies are eliminated):

**Using register renaming**

**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

As shown above, name dependencies can be eliminated by “renaming” the shared names (renaming registers in this case, requiring more ISA registers).

As was done in Loop unrolling example.
Control Dependencies

- Control dependence determines the ordering of an instruction with respect to a branch (control) 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.

1. 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.
2. 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).
   → Both scenarios lead a control dependence violation (control hazard).

- 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
    }
```

What happens if   S1  is moved here?

Control Dependence Violation = Control Hazard

In Fourth Edition Chapter 2.1
(In Third Edition Chapter 3.1)
Control Dependence Example

The unrolled loop code with the intermediate branches still in place is shown here.

Branch conditions are complemented here (BEQ instead of BNE, except last one) to allow the fall-through to execute another loop.

BEQ instructions prevent the overlapping of iterations for scheduling optimizations. (4 basic blocks B0-B3 each 5 instructions)

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

Removing the intermediate branches changes (removes) the internal control dependencies present increasing basic block size (to 14) and makes more optimizations (reordering) possible.

As seen previously in the loop unrolling example

B0 – B3: Basic blocks, 5 instructions each