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

(Due to conditional branches)

Or exposing more ILP

A Static Optimization Technique

In terms of number of instructions in Basic Block

→ Dependency Analysis
Dependency Graphs

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

Pipeline Hazard Condition = Dependency Violation

Pipelining and Exploiting Instruction-Level Parallelism (ILP)

+ Such independent instructions can be re-ordered

- Instruction-Level Parallelism (ILP) exists when instructions in a code sequence are independent and thus can be executed in parallel by overlapping. (without stalling)
  - i.e. parallel
  - i.e. instruction throughput
  - 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):

i.e non-ideal

Pipeline CPI = Ideal Pipeline CPI + Structural Stalls + RAW Stalls + WAR Stalls + WAW Stalls + 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

Dependency Violation = Hazard

In Fourth Edition Chapter 2.1
(In Third Edition Chapter 3.1)

T = I x CPI x C

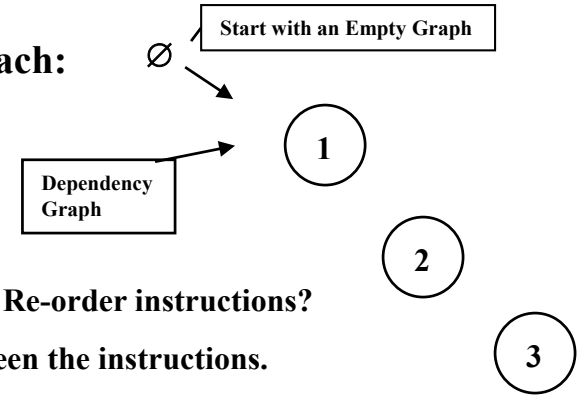
Instruction-Level Parallelism (ILP) Example

Given the following two code sequences with three instructions each:

Higher ILP

Program Order

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



Can Re-order instructions?

The instructions in the first code sequence above have no dependencies between the instructions.
Thus the three instructions are said 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.

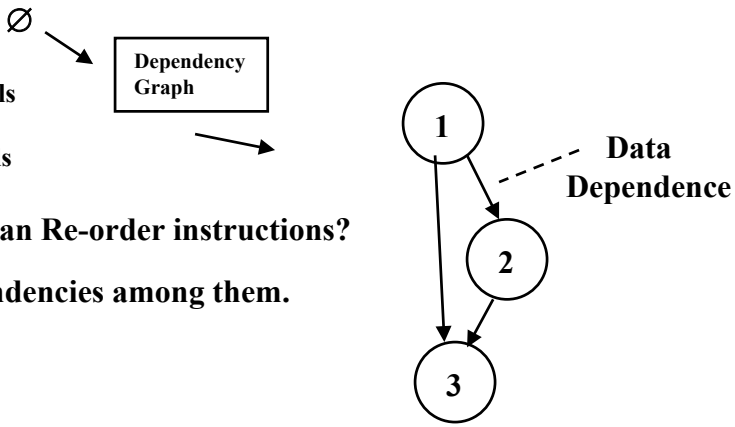
Independent or parallel instructions. (no dependencies exist): High ILP

No Stalls?

Lower ILP

Program Order

```
1   ADD.D   F2, F4, F6
2   ADD.D   F10, F2, F8     ← stalls
3   ADD.D   F12, F10, F2    ← stalls
```



Can Re-order instructions?

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 be independent and thus can be executed in parallel and their order cannot be changed with causing incorrect execution.

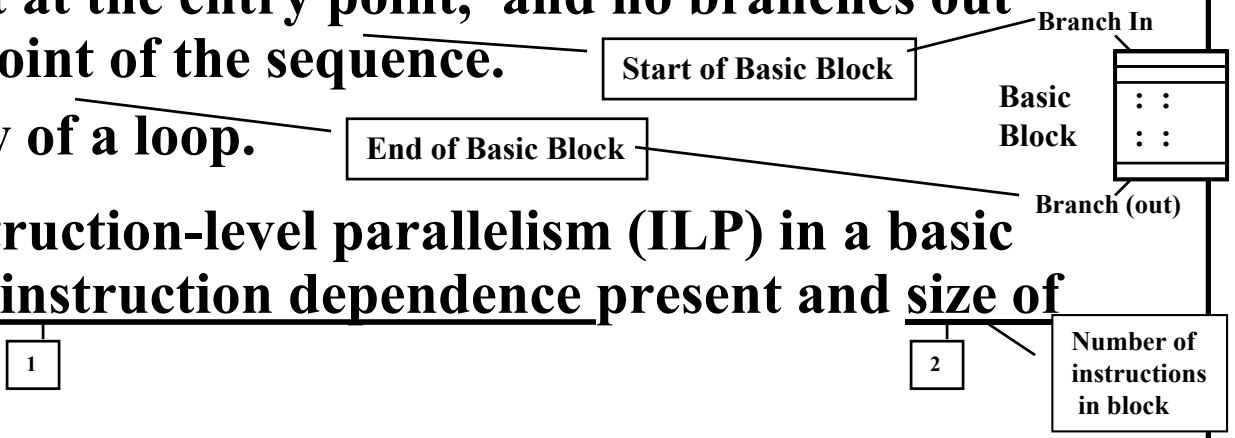
Dependent instructions (three dependencies exist): Lower ILP

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.

1

2

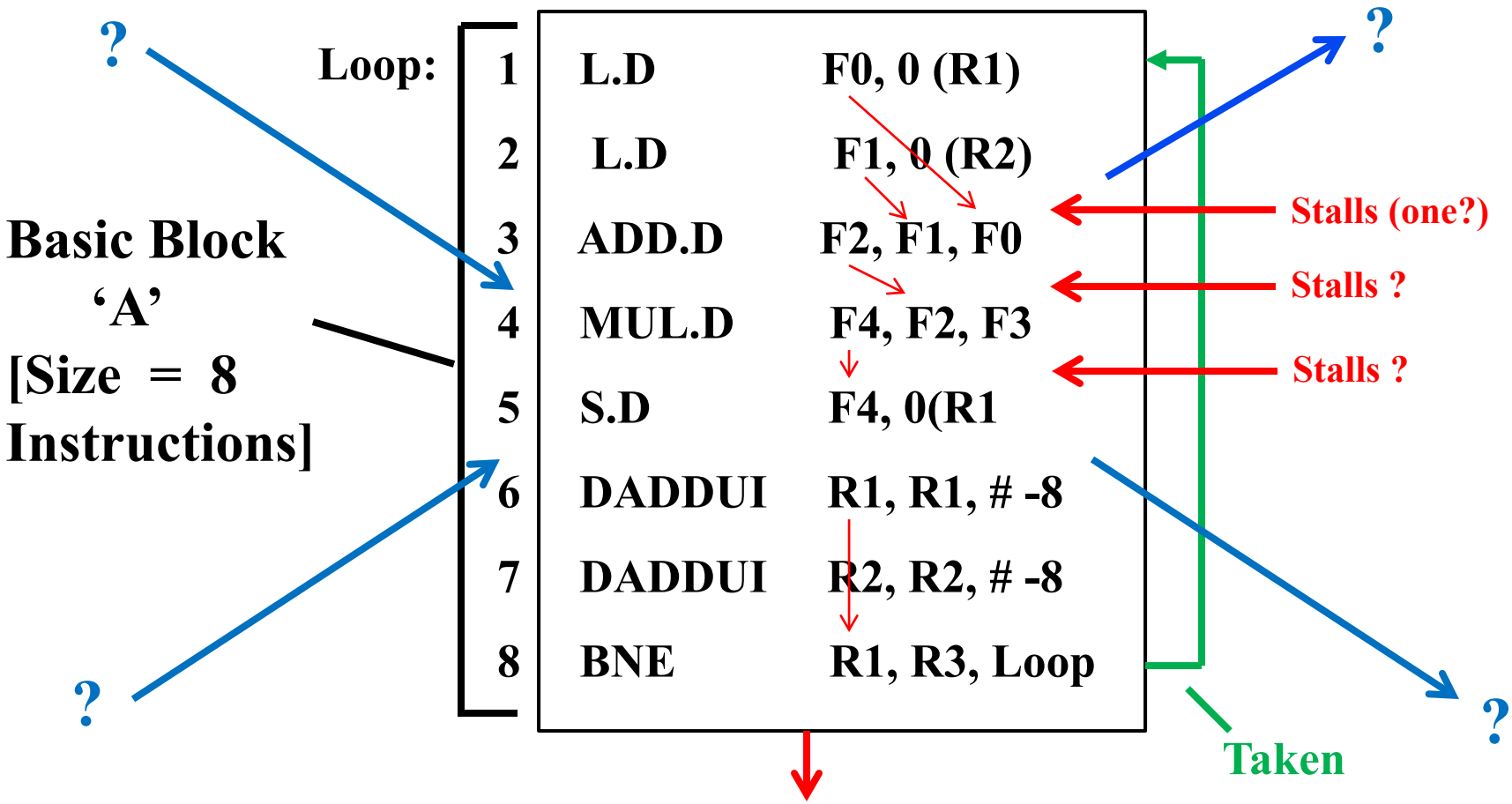
Number of instructions in block

- 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 (reordering) by the compiler possibly eliminating more stall cycles and thus improves pipelined CPU performance.
  - Loop unrolling is one such technique that we examine next

In Fourth Edition Chapter 2.1 (In Third Edition Chapter 3.1)

Basic Block Example: Body of A Loop

Dynamic Execution Trace or Path: AAAAAAAA ….. ABCD……



Basic Blocks/Dynamic Execution Sequence (Trace) Example

Static Program Order

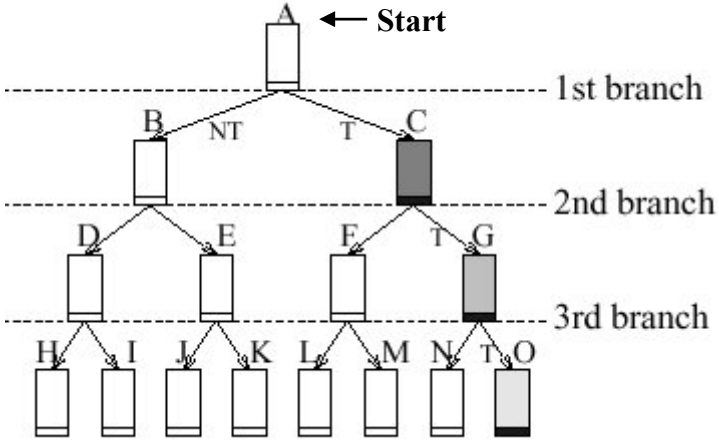
A ← Start
B
D
H
⋮
E
J
⋮
I
⋮
K
⋮
C
F
L
⋮
G
N
⋮
M
⋮
O

- **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

Dynamic Execution Trace (Path), Determined by outcomes of branches

Program Control Flow Graph (CFG)



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

NT = Branch Not Taken
T = Branch Taken

Type of branches in this example: "If-Then-Else" branches (not loops)

Average Basic Block Size = 5-7 instructions

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).

i.e independent or parallel loop iterations

Or Data Parallelism in a loop

- 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];
```

X[i+1] ?

Independent (parallel) loop iterations:
A result of high degree of data parallelism

4 vector instructions:

Load Vector X
Load Vector Y
Add Vector X, X, Y
Store Vector X

- 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.

(potentially)

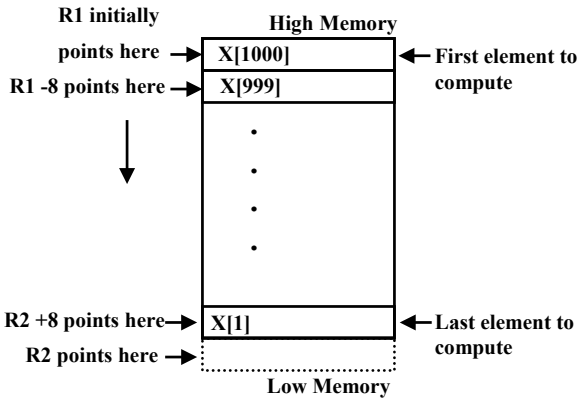
In Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)

MIPS Loop Unrolling Example

- **For the loop:**

Note: Independent (parallel) Loop Iterations

```
for (i=1000; i>0; i=i-1)
x[i] = x[i] + s;
```



The straightforward MIPS assembly code is given by:

Program Order

```
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
```

S

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

X[ ] array of double-precision floating-point numbers (8-bytes each)

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

Initial value of R1 = R2 + 8000

MIPS FP Latency Assumptions Used In Chapter 2.2

For Loop Unrolling Example

3rd Edition in 4.1

FP EX Cycles = 4

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

i.e 4 execution (EX) cycles for FP instructions

(or Number of Stall Cycles)

i.e followed immediately by ..

Instruction Producing Result	Instruction Using Result	Latency In Clock Cycles
FP ALU Op	Another FP ALU Op	3
FP ALU Op	Store Double	2
Load Double	FP ALU Op	1
Load Double	Store Double	0

Taken

Other Assumptions:

- Branch resolved in decode stage, Branch penalty = 1 cycle
- Full forwarding is used
- Single Branch delay Slot
- Number of iterations is multiple of 4
- Some potential structural hazards ignored

i.e. MIPS Pipeline Version #3

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

Loop Unrolling Example (continued)

As is (No Loop Unrolling Yet)

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

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

Program Order

No scheduling (as is)

(Resulting stalls shown)

			Clock cycle
Loop:	L.D	F0, 0(R1)	1
	stall		2
	ADD.D	F4, F0, F2	3
	stall		4
	stall		5
	S.D	F4, 0 (R1)	6
	DADDUI	R1, R1, # -8	7
	stall		8
	BNE	R1,R2, Loop	9
	stall		10

Due to resolving branch in ID

Branch Delay Slot (Empty)

10 cycles per iteration

Scheduled with single delayed branch slot:

(Resulting stalls shown)

			Cycle
Loop:	L.D	F0, 0(R1)	1
	DADDUI	R1, R1, # -8	2
	ADD.D	F4, F0, F2	3
	stall		4
	BNE	R1,R2, Loop	5
	S.D	F4,8(R1)	6

S.D in branch delay slot

Offset adjusted

6 cycles per iteration

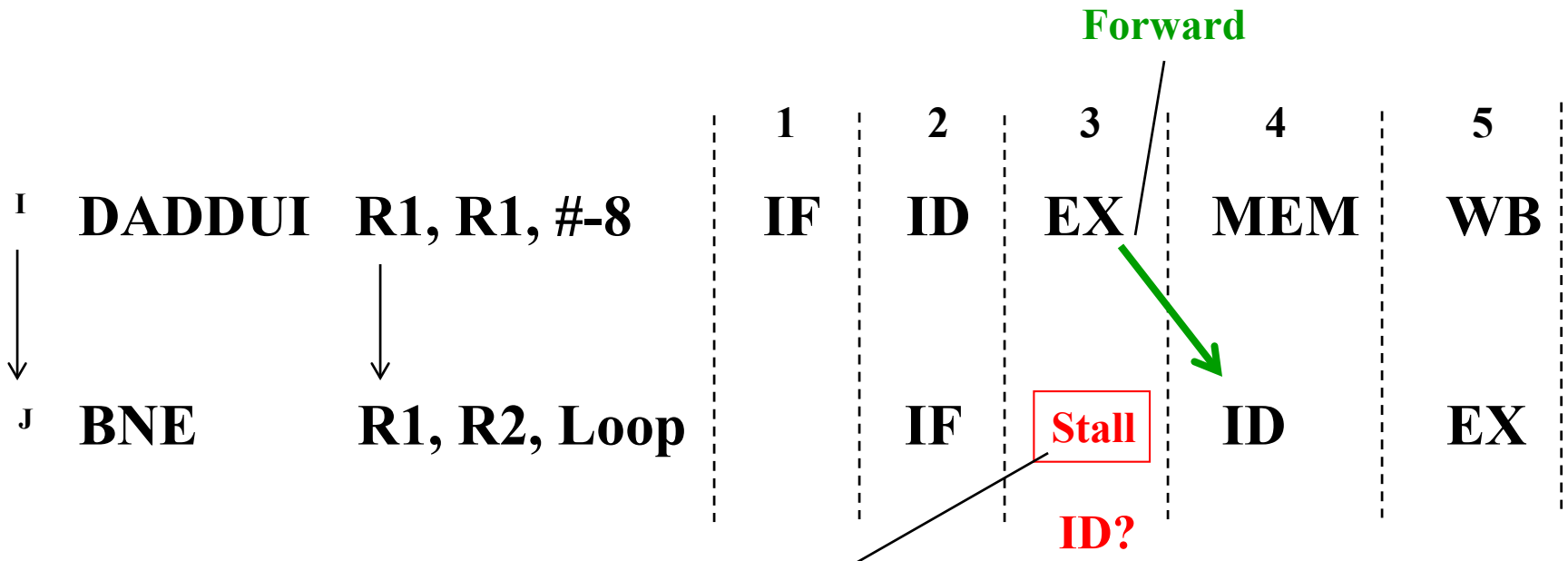
10/6 = 1.7 times faster

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

- Ignoring Pipeline Fill Cycles
- Some Structural Hazards Ignored

Early Resolution of Branches And Potential RAW Hazard

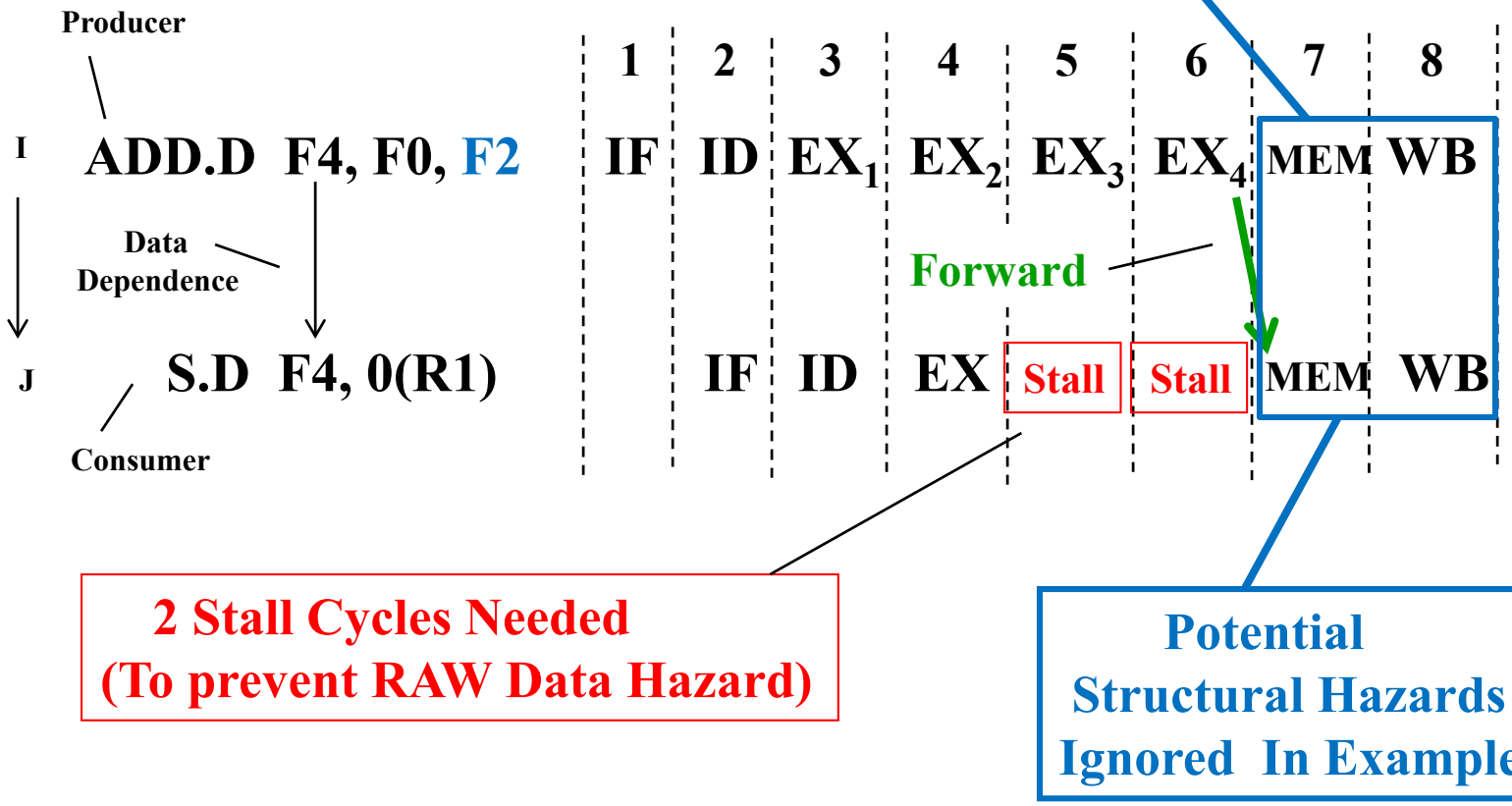
Example [For Pipeline Version #3]



One Stall needed
Then Forward

One stall needed even with forwarding due to resolving branch early in ID stage [Needed to prevent RAW Hazard]

Potential Structural Hazards Ignored In Example



Loop Unrolling Example (continued)

Loop unrolled 4 times

Iteration | Cycle

No scheduling

```
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
     28  Stall
```

1 | 2 | 3 | 4

Empty Branch Delay Slot

(Resulting stalls shown)

New Basic Block Size = 14 Instructions

- 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.

Register Renaming Used

28/4 =7 Cycles per original iteration

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.

i.e 7 cycles for each original iteration

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

i.e. unrolled four times

Note use of different registers for each iteration (register renaming)

Loop Unrolling Example (continued)

Note: No stalls

One possible schedule

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

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
```

Program Order

The execution time of the loop has dropped to 14 cycles, or 14/4 = 3.5 clock cycles per element

i.e 3.5 cycles for each original iteration

compared to 7 before scheduling and 6 when scheduled but unrolled.

Speedup = 6/3.5 = 1.7

i.e more ILP exposed

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 → More ILP Exposed

Offset = 16 - 32 = -16

14/4 = 3.5 Cycles per original iteration

In branch delay slot

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 ISA 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.**

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).

  + Such instructions can be re-ordered
- Instructions that are dependent are not parallel and cannot be reordered by the compiler or hardware.

  Otherwise incorrect execution results

  → Pipeline Hazard = Dependency Violation
- 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

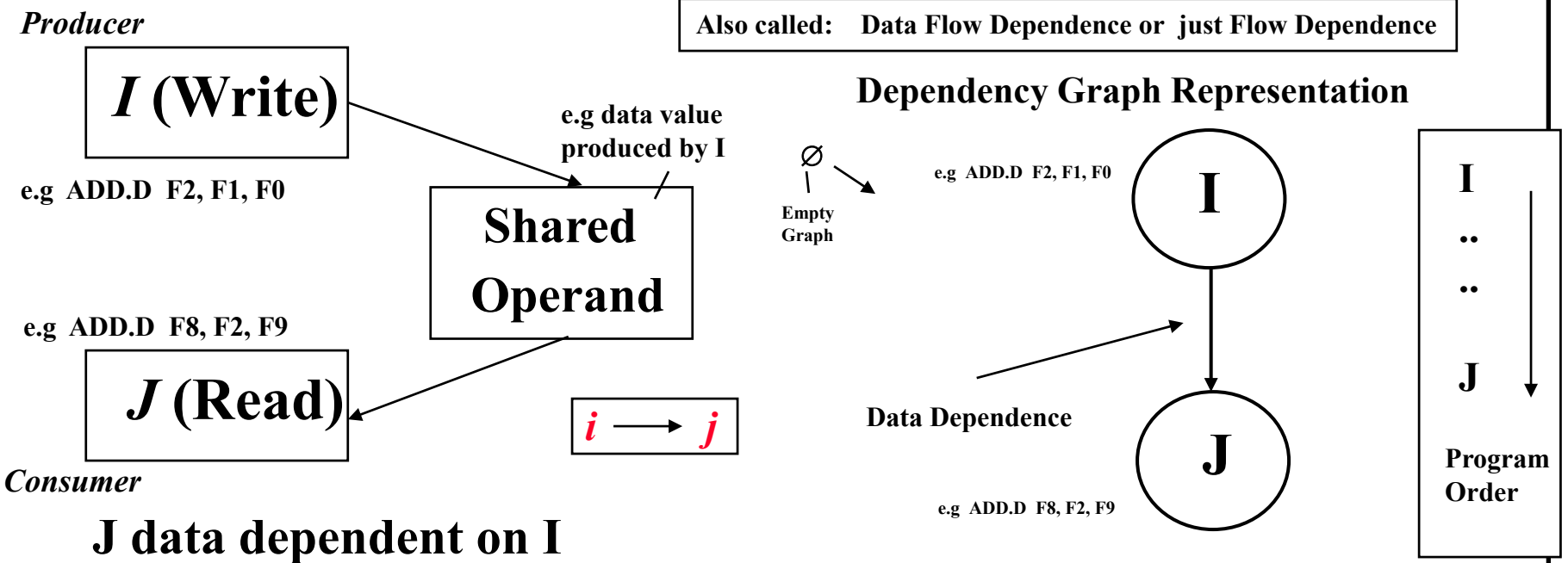
(Due to conditional branches)

(True) Data Dependence

AKA Data Flow Dependence

- Instruction *i* precedes instruction *j* in the program sequence or order
- Instruction *i* produces a result used by instruction *j*, $i \longrightarrow j$
  - 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 in a RAW hazard and incorrect execution.

Also called: Data Flow Dependence or just Flow Dependence



J data dependent on I

resulting in a Read after Write (RAW) hazard if their relative execution order is changed

i.e Data dependence violation = RAW Hazard

i.e. A data dependence is violated

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

Instruction Data Dependencies

(or Flow)

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

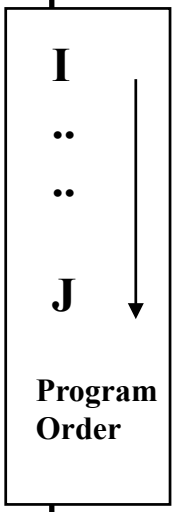
- Instruction *j* is data dependent on instruction *i* if:

Data Dependence Chain

  - 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.

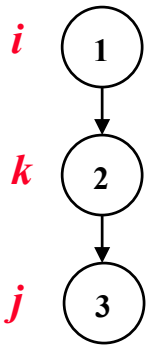
Example

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.




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

Dependency Graph



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:

I
..
..
J

Program Order

– An anti-dependence exists when *j* writes to the same register or memory location that instruction *i* reads (i.e. the same shared name)

- 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 preserved.

– An output or (write) dependence exists when instruction *i* and *j* write to the same register or memory location (i.e. the same shared 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: Register or Memory Location

Name Dependence Classification: Anti-Dependence

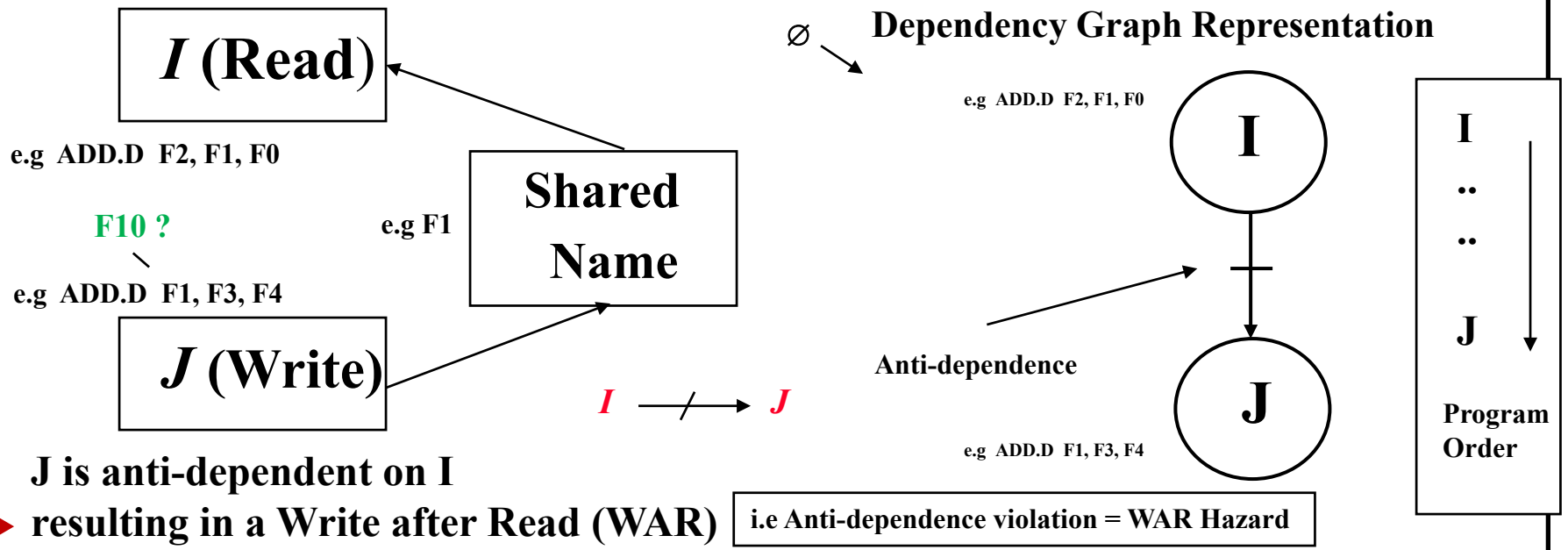
- Instruction *i* precedes instruction *j* in the program sequence or order
- Instruction *i* reads a value from a name (register or memory location)
- Instruction *j* writes a value to the same name (same register or memory location read by i)

$I \not\longrightarrow J$

→ – 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.



J is anti-dependent on I

→ resulting in a Write after Read (WAR) hazard if their relative execution order is changed

i.e Anti-dependence violation = WAR Hazard

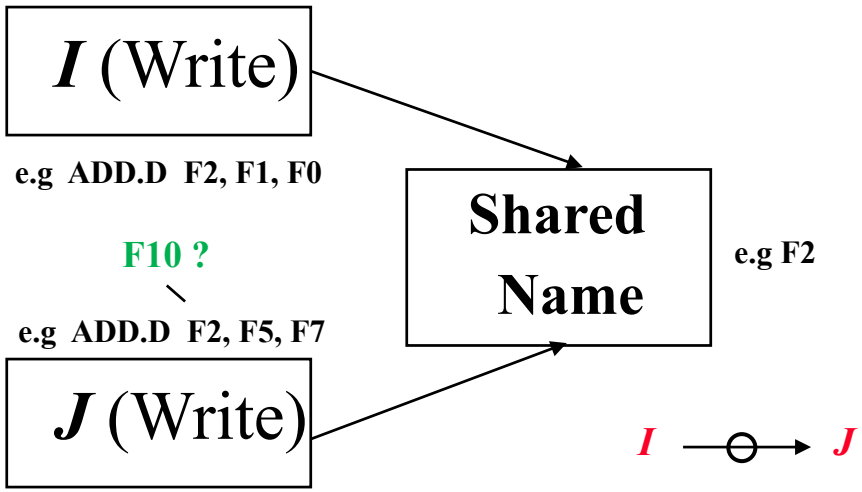
i.e relative order of read by I and write by J

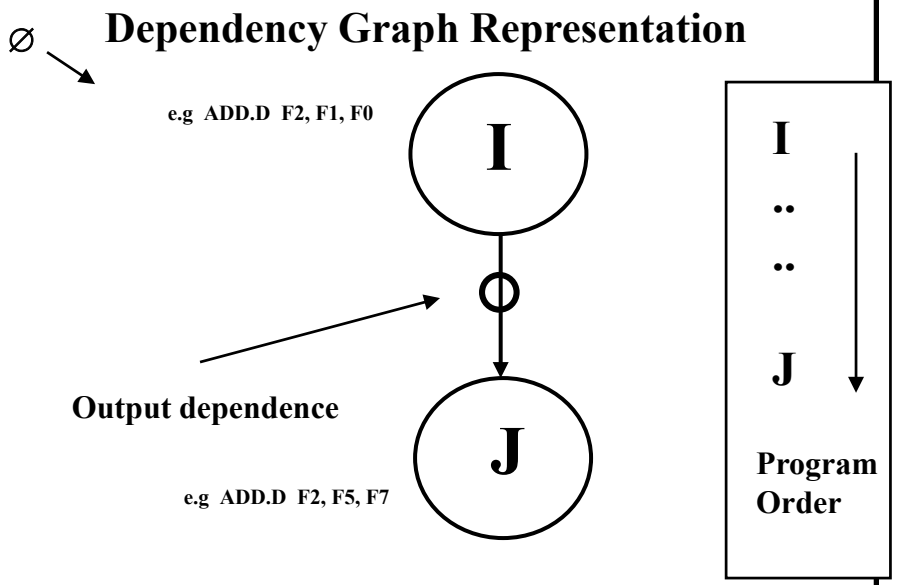
Name: Register or Memory Location

Name Dependence Classification:
Output (or Write) Dependence

$I \xrightarrow{o} J$

- 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

i.e Output-dependence violation = WAW Hazard

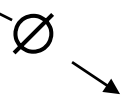
i.e relative order of write by I and write by J

Name: Register or Memory Location

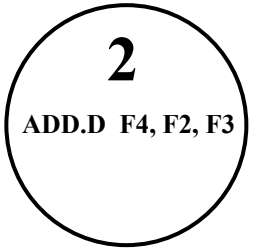
Dependency Graph Example

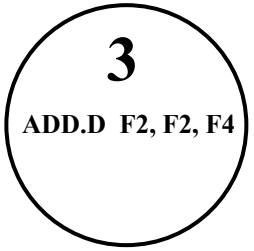
For the given MIPS code identify all data and name dependence between instructions and complete the dependency graph

Start with an Empty Graph



Instruction Dependency graph





MIPS Code

Program Order ↓

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:

Dependency Graph Example

Start with Instruction 1

∅ →

Instruction Dependency graph

MIPS Code

Program Order ↓

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:

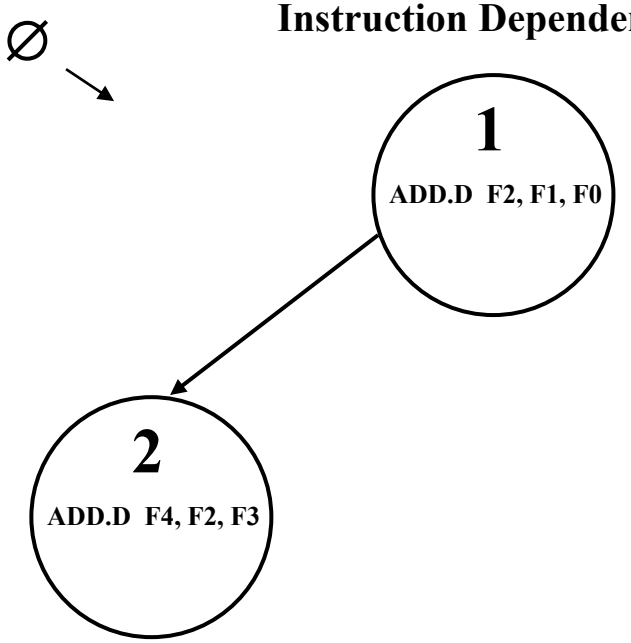
Output Dependence:

Anti-dependence:

Dependency Graph Example

Add Instruction 2 to the graph

Instruction Dependency graph



MIPS Code

Program Order

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)

i.e. 1 ⟶ 2

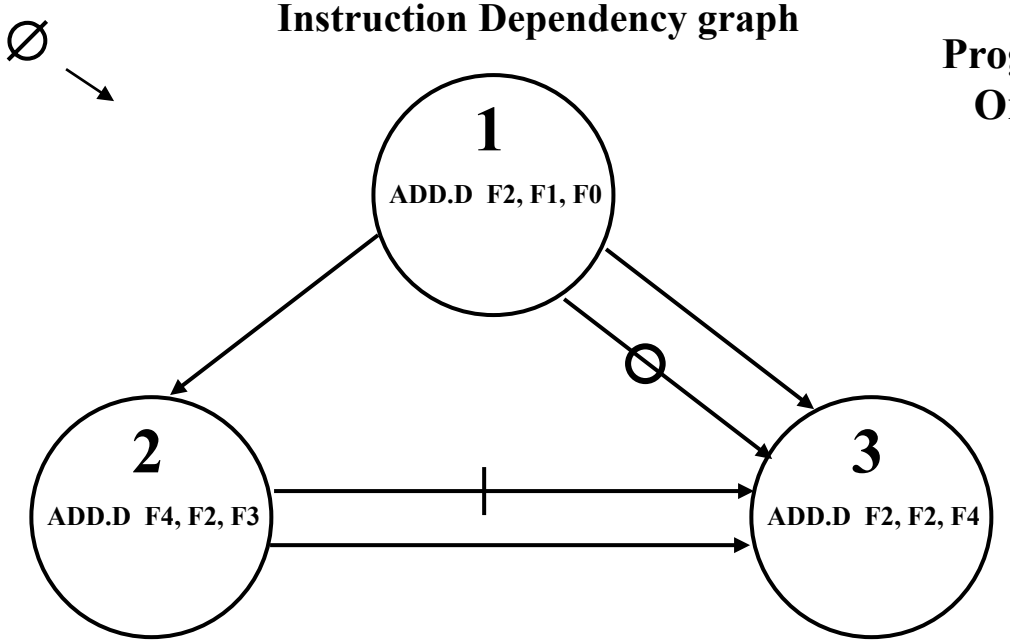
Output Dependence:

Anti-dependence:

Dependency Graph Example

Add Instruction 3 to the graph

Instruction Dependency graph



MIPS Code

Program Order

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)

i.e. 1 → 2 1 → 3
2 → 3

Output Dependence:

(1, 3)

i.e. 1 ⊖→ 3

Anti-dependence:

(2, 3)

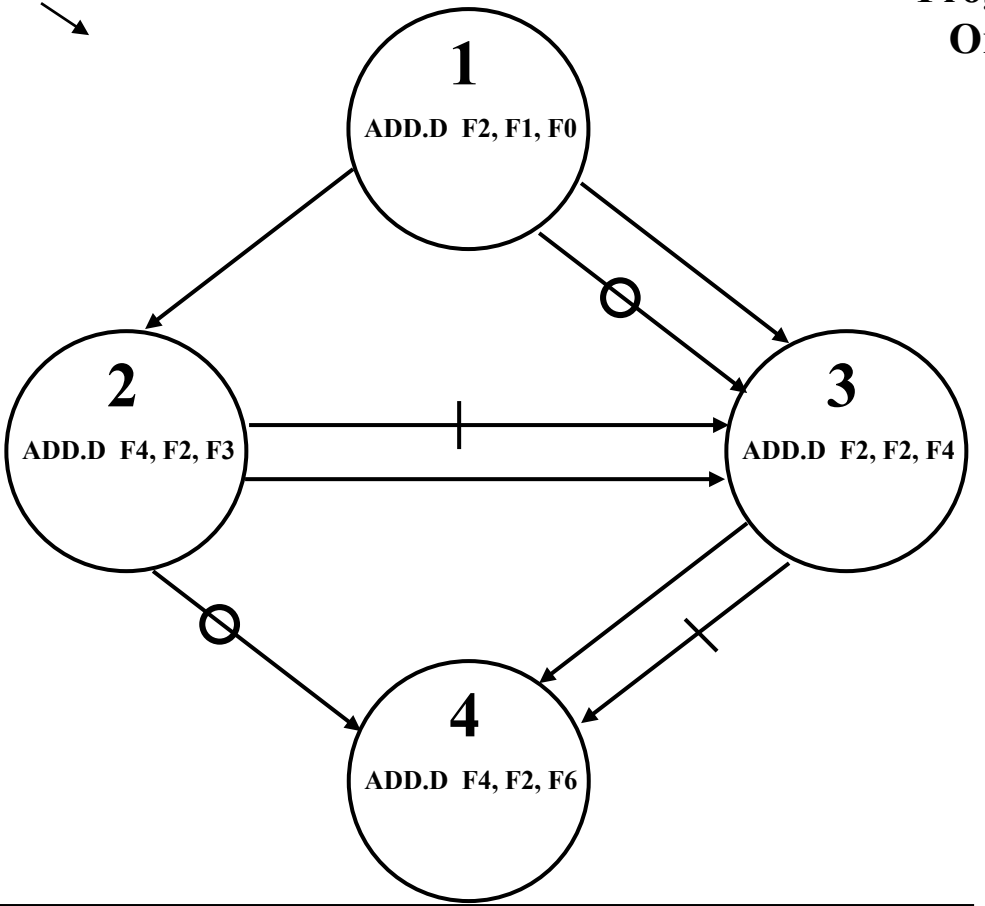
i.e. 2 ⊣→ 3

Dependency Graph Example

Completed Dependency Graph

Add Instruction 4 to the graph

Instruction Dependency graph



Program Order

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)

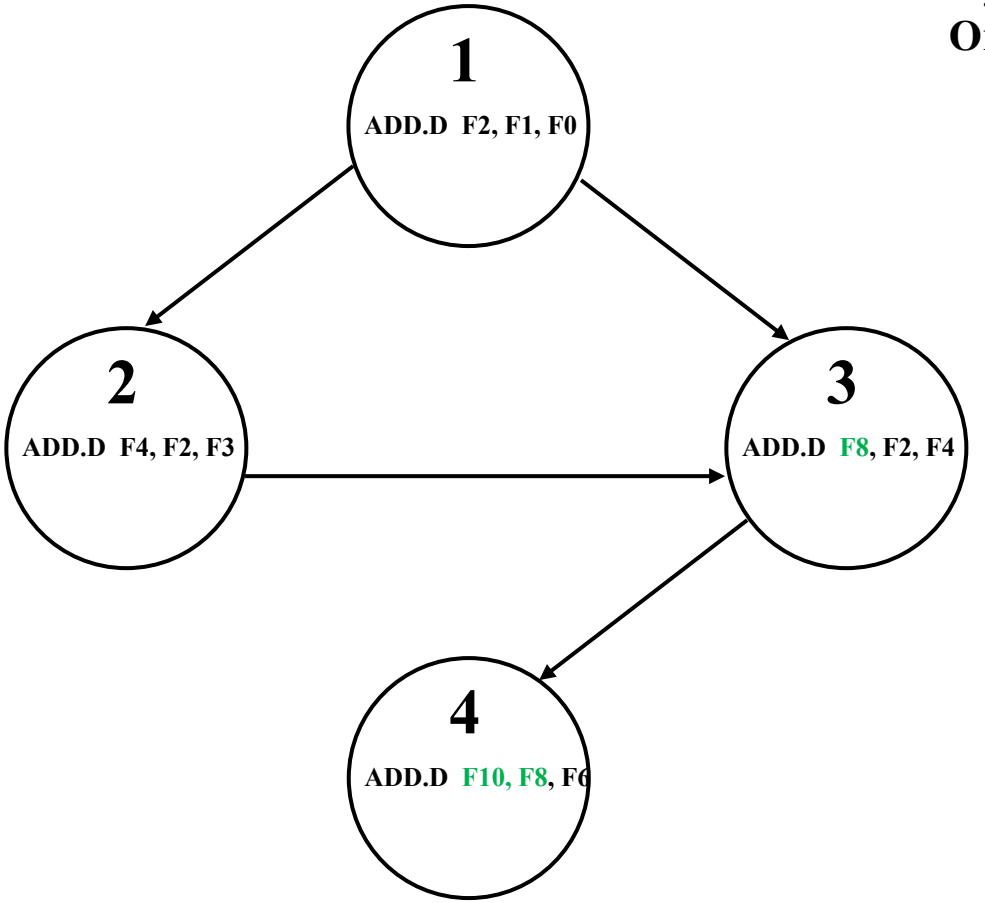
i.e. 2 → 3 3 → 4

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

Dependency Graph Example

Eliminating Name Dependencies By Register Renaming

Instruction Dependency graph



MIPS Code

Program Order

1	ADD.D	F2, F1, F0
2	ADD.D	F4, F2, F3
3	ADD.D	F8, F2, F4
4	ADD.D	F10, F8, F6

True Date Dependence:

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

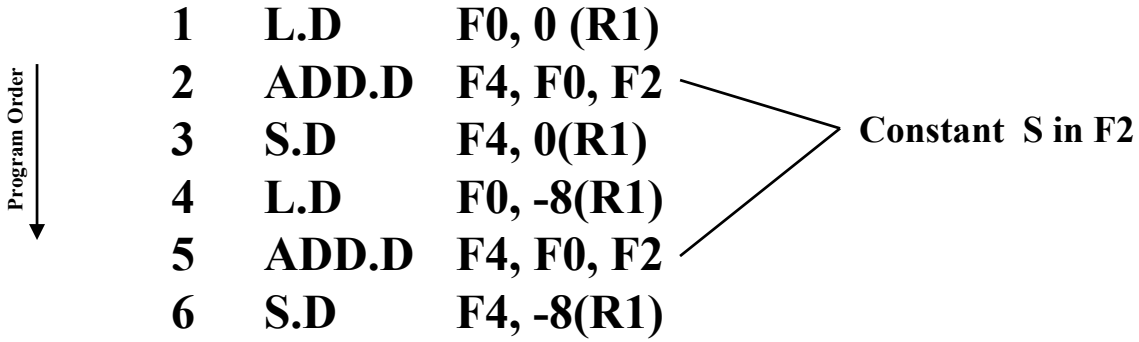
i.e. 1 → 2 1 → 3
2 → 3 3 → 4

Output Dependence:

Anti-dependence:

Instruction Dependence Example

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



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):

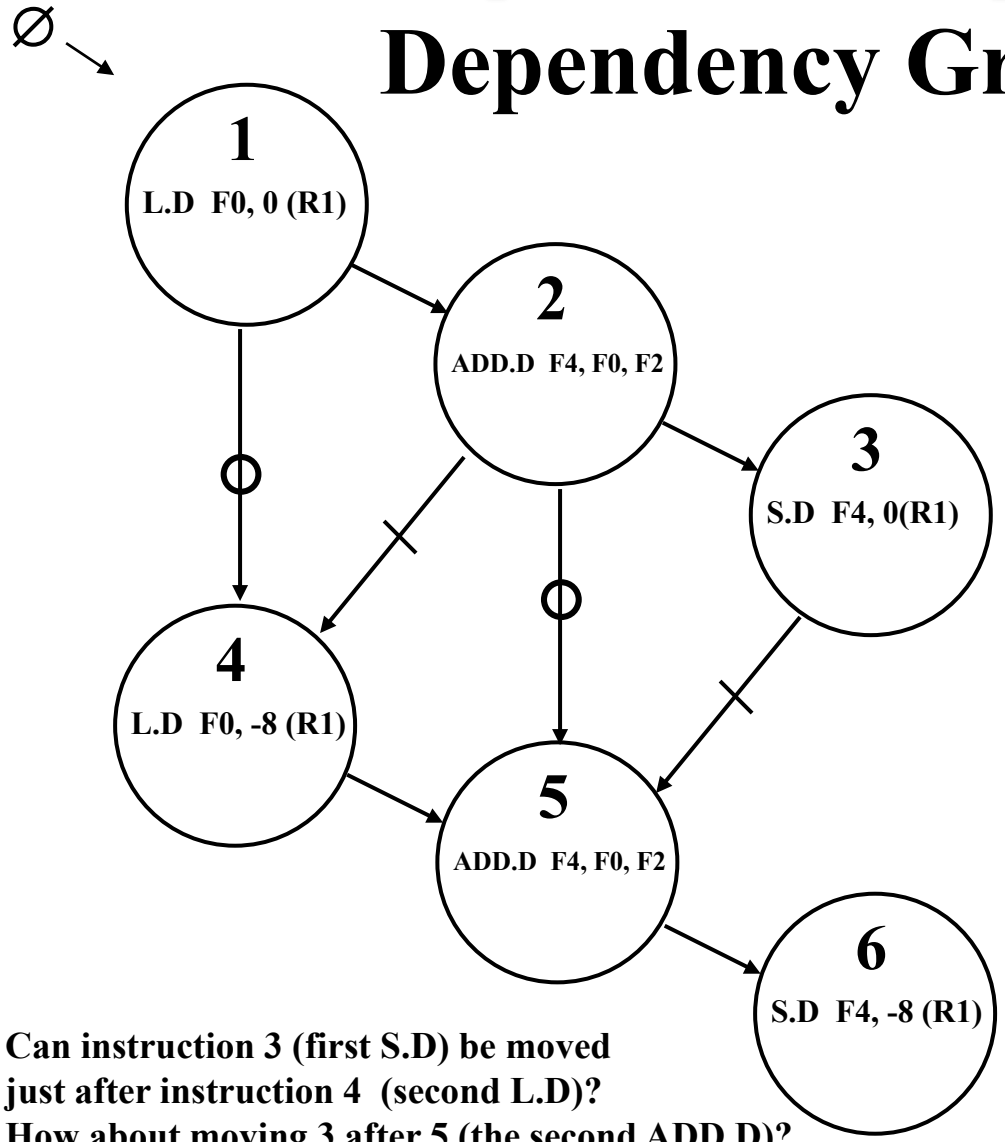
Instruction 1 has an output name dependence (WAW) over result register (name) F0 with instructions 4
Instruction 2 has an output name dependence (WAW) over result register (name) F4 with instructions 5

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

Dependency Graph



Program Order

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?

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?

Instruction Dependence Example

No Register Renaming Done

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

Program Order

```
1   Loop: 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)
7         L.D      F0, -16(R1)
8         ADD.D    F4, F0, F2
9         S.D      F4, -16 (R1)
10        L.D      F0, -24 (R1)
11        ADD.D    F4, F0, F2
12        S.D      F4, -24(R1)
13        DADDUI   R1, R1, # -32
14        BNE      R1, R2, Loop
```

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)

Name Dependence Removal

Using Register Renaming →

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

Program Order ↓

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

(Name dependencies are eliminated):

Using register renaming

As was done in Loop unrolling example

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

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

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:

```
if p1 {          <- Conditional branch
        S1;
};
If p2 {          <- Conditional branch
        S2;      <- What happens if S1 is moved here?
}
```

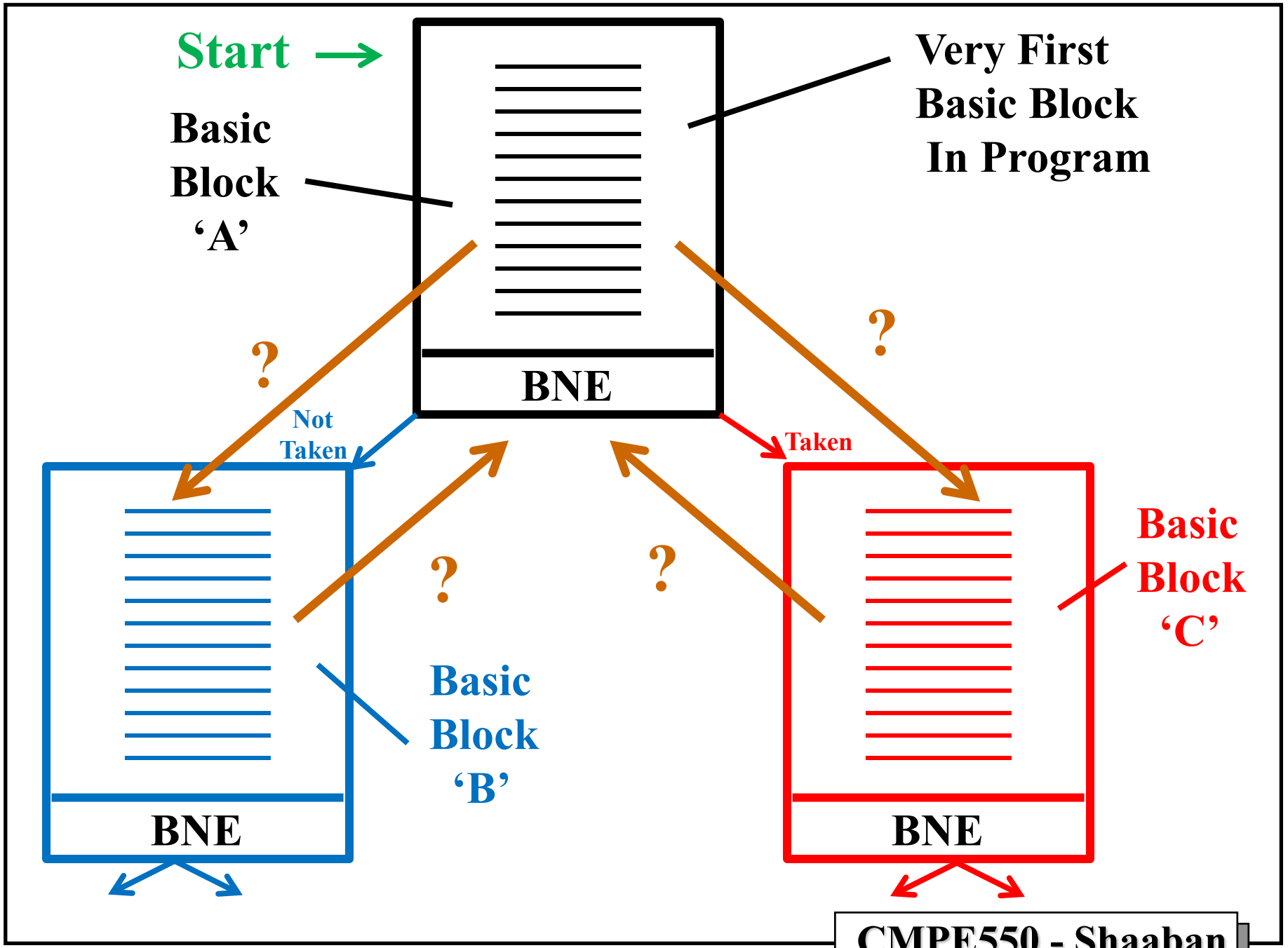
? (between S1 and S2)

S1 is control dependent on p1

S2 is control dependent on p2 but not on p1

In Fourth Edition Chapter 2.1
(In Third Edition Chapter 3.1)

Control Dependence Violation = Control Hazard


Start
Basic Block 'A'
Very First Basic Block In Program
BNE
?
?
Not Taken
Taken
?
?
Basic Block 'B'
Basic Block 'C'
BNE
BNE

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)

Due to control dependencies

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

```
Loop: L.D      F0, 0 (R1)
      ADD.D    F4, F0, F2
B0    S.D      F4,0 (R1)
      DADDUI   R1, R1, # -8
      BEQ      R1, R2, exit
      L.D      F6, 0 (R1)
      ADD.D    F8, F6, F2
B1    S.D      F8, 0 (R1)
      DADDUI   R1, R1, # -8
      BEQ      R1, R2, exit
      L.D      F10, 0 (R1)
      ADD.D    F12, F10, F2
B2    S.D      F12,0 (R1)
      DADDUI   R1, R1, # -8
      BEQ      R1, R2,exit
      L.D      F14, 0 (R1)
      ADD.D    F16, F14, F2
B3    S.D      F16, 0 (R1)
      DADDUI   R1, R1, # -8
      BNE      R1, R2,Loop
exit:
```

B0 – B3: Basic blocks, 5 instructions each

Quiz # 3

On Lecture Notes Set # 3

- **Wednesday, February 15**