CS/ECE 5381/7381 Computer Architecture Spring 2023

Dr. Manikas

Computer Science

Lecture 9: Feb. 21, 2023

Project 1

- Due TODAY Tue., Feb. 21 (11:59 pm)
- First programming project – MARS tool
- Assignment:
  - Run tutorial to get familiar with MARS tool
  - You will use this tool in upcoming projects

Project 2

- Due NEXT Tues, Feb. 28 (11:59 pm)
- MARS tool
- Assignment:
  - Translate high-level language code into MIPS code
  - Run code on MARS tool
  - HINT: feel free to borrow from Fibonacci code from Project 1

Project 3 (7381 only)

- Due NEXT Thur., Mar 2 (11:59 pm)
- For CS 7381 students ONLY
- Additional MIPS programming assignment using MARS tool

Schedule Notes

- NO lecture for Thursday, Feb. 23, since we have Exam 1

Exam 1

- Exam will be administered using Lockdown Browser
- Exam format will be like the quizzes, but longer
  - 25 questions
  - You will be allowed 120 minutes (2 hours) to take the exam
  - The exam will be available from **Thursday, Feb 23 at 12 am**
  - The exam must be completed and submitted by **Saturday, Feb 25 at 11:59 pm**

Exam 1

- **Exam 1 will cover the following materials:**
  - Modules: 1 - 4
  - Quizzes: 1 - 4
  - Text: Ch. 1, App. A, App. C
- **MATERIALS ALLOWED FOR EXAM:**
  - Open book and notes, including MIPS reference data sheet
  - Calculator

Instruction-Level Parallelism (ILP) and Its Exploitation

(Chapter 3, Hennessy and Patterson)

Note: some course slides adopted from publisher-provided material

Outline

- 3.1 ILP Background
- 3.2 Basic Compiler Techniques for ILP
- 3.3 Branch Prediction
- 3.4 Data Hazards and Dynamic Scheduling
- 3.5 Dynamic Scheduling Algorithm
- 3.6 Hardware-Based Speculation

Introduction

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”

- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI (cycles per instruction)
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
- Parallelism with basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches

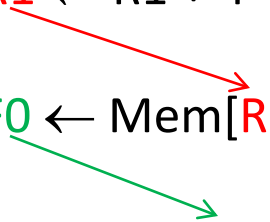
Data Dependencies

- Instruction K is *data dependent* on instruction J if:
  - J produces a result that may be used by K, or
  - K is data dependent on instruction L, and L is data dependent on J

```
DADDUI R1,R1,#4        ;R1 ← R1 + 4

L.D    F0, 0(R1)       ;F0 ← Mem[R1]

ADD.D  F2, F1, F0      ;F2 ← F1 + F0
```



Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, *but is a problem when reordering instructions (to be discussed soon)*
  - *Antidependence*: instruction j **writes** a register or memory location that instruction i **reads**
    - Initial ordering (i before j) must be preserved
  - *Output dependence*: instruction i and instruction j write the same register or memory location
    - Ordering must be preserved
- To resolve, use renaming techniques

Data Hazards

- Recall data hazards due to data dependencies
- Can be addressed using *forwarding* and *stalls*
- Types of data hazards – *more specific*
  - Read After Write (RAW)
  - Write After Write (WAW)
  - Write After Read (WAR)
- NOTE: Read After Read (RAR) is not a hazard

Data Dependences

- Types of data dependences
  - Flow dependence (true data dependence – read after write)
  - Output dependence (write after write)
  - Anti dependence (write after read)

Data Dependences

- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program are correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - They are dependence on a name, not a value
    - We will later see what we can do about them

Flow dependence

$r_3 \leftarrow r_1$ op r_2 Read-after-Write

$r_5 \leftarrow r_3$ op r_4 (RAW)

Anti dependence

$r_3 \leftarrow r_1$ op r_2 Write-after-Read

$r_1 \leftarrow r_4$ op r_5 (WAR)

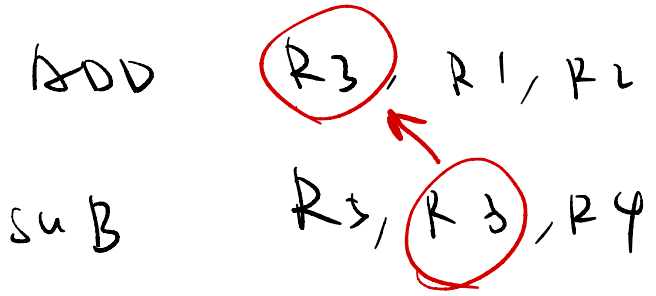
Output-dependence

$r_3 \leftarrow r_1$ op r_2 Write-after-Write

$r_5 \leftarrow r_3$ op r_4 (WAW)

$r_3 \leftarrow r_6$ op r_7

RECALL FROM PIPELINE EXAMPLES



DATA HAZARD R3

R3 MUST BE

(WRITE) UPDATED BY "ADD"

BEFORE READ BY "SUB"

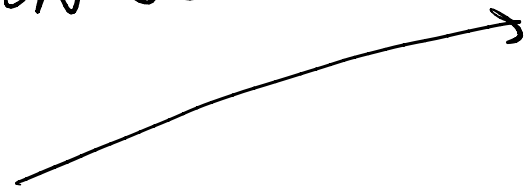
READ AFTER WRITE (RAW)

"SUB" READS R3 AFTER "ADD" WRITES TO R3

RAW is CALLED "FLOW DEPENDENCE"

WRITE-AFTER-READ (WAR)

"ANTI DEPENDENCE"



This is a potential problem if I change the order of these instructions. This is fine but if I swap these will get a problem.

ADD R3, R1, R2

SUB R1, R4, R5

READ → WRITE

WHY MIGHT THIS BE A PROBLEM?

TRY THIS: "RE-ORDER" INSTRUCTIONS

SUB R1, R4, R5

ADD R3, R1, R3

NOW HAVE RAW

⇒ DATA HAZARD

Why would we want to reorder instructions?

Some instructions take longer than others remember we had some multi-cycle instructions we may want to change the order of excution to make the program run a more efficient.

example:

```
ADD  R3, R1, R2
```

→ let's say one of these is a floating point add

```
SUB  R1, R4, R5
```

→ This is integer sub

Add is going to take longer I'm going to move that later, but we have a data Hazard if we do that We can't just randomly randomly reorder instructions, because we may introduce hazards where we didn't have before.

WRITE - AFTER - WRITE (WAW)

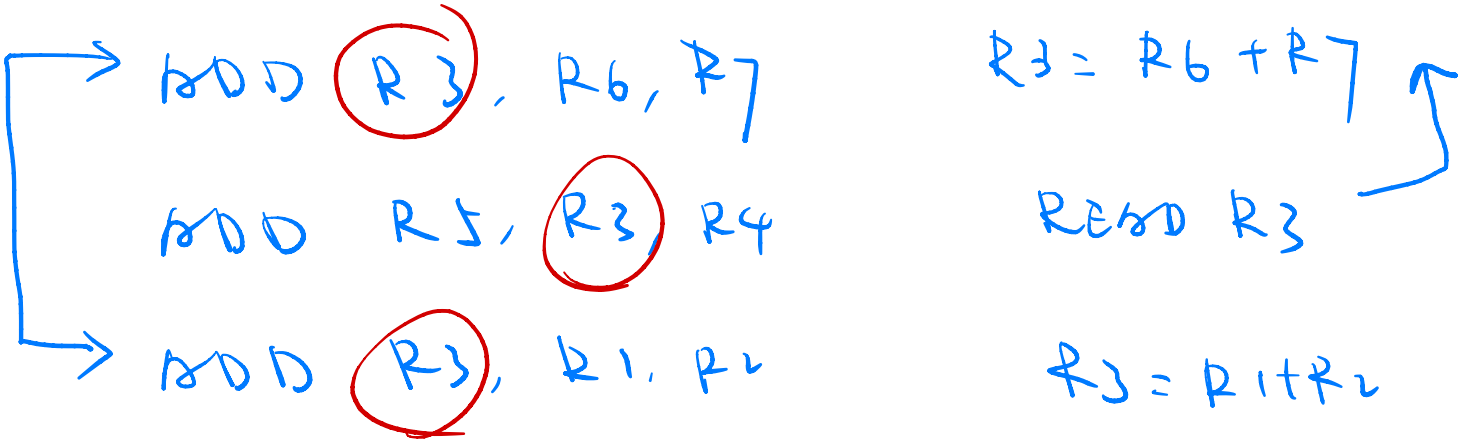
"OUTPUT DEPENDENCE"

ADD R3, R1, R2 — WRITE R3 = R1 + R2

ADD R5, R3, R4 — (RAW HERE) READ R3

ADD R3, R6, R7 — WRITE R3 = R6 + R7

"Re-ORDER" INSTRUCTION



Control Dependencies

ORIGINAL CODE:

```
Instruction J
Instruction K
IF (branch condition) THEN
{
        instruction L
        instruction M
}
```

Reordering not allowed – K is not control dependent

```
Instruction J
IF (branch condition) THEN
{
        instruction K
        instruction L
        instruction M
}
```

Reordering not allowed – L is control dependent

```
Instruction J
Instruction K
Instruction L
IF (branch condition) THEN
{
        instruction M
}
```

Outline

- 3.1 ILP Background
- 3.2 Basic Compiler Techniques for ILP
- 3.3 Branch Prediction
- 3.4 Data Hazards and Dynamic Scheduling
- 3.5 Dynamic Scheduling Algorithm
- 3.6 Hardware-Based Speculation

Basic Compiler Techniques for ILP

- *Static scheduling*: compiler schedules instructions
- Goal: arrange instructions to avoid hazards
  - This is called "instruction reordering"

Instruction Latencies

- For App. C (basic pipelining), we assumed that *all* instructions took the *same* number of clock cycles
- In reality, some instructions require more cycles then others
  - E.g., Multiply, Divide can take many cycles
- Latency = # of **extra** clock cycles required for an instruction

Effect of Instruction Latencies

- As some instructions may take more cycles than others, it can become necessary to add stalls (similar to pipeline stalls) or re-order instructions to improve performance

Baseline Performance

- It is often helpful to determine the *baseline performance* of a set of code with instruction latencies
- The baseline assumes that all instructions execute in order, and that instruction k must complete before instruction k+1 executes (i.e., no pipelining)

Example 3.2-1: Baseline Performance

We are given the following MIPS code with the following instruction latencies:

Label	Opcode	Operands	Latency
Loop:	MULT.D	F2,F6,F2	5
I0:	L.D	F4,0(Ry)	4
I1:	ADD.D	F4,F0,F4	1
I2:	DIV.D	F8,F2,F0	12
I3:	S.D	F4,0(Ry)	1
I4:	BNZ	R20,Loop	1

extra cycles

But now we're ignoring pipelining so we don't care about the data hazards right now.

What is the *baseline performance* (number of cycles) for this code?

LABEL	OP CODE	EXTRA CYCLES LATENCY	CYCLES	TOTAL
Loop:	MULT.D	5	6	6
I0:	L.D	4	5	11 = 6+5
I1:	ADD.D	1	2	13 = 6+5+2
I2:	DIV.D	12	13	>6
I3:	S.D	1	2	28
I4:	BNE	1	2	30

BASELINE PERFORMANCE is 30 CYCLES.

Instruction Latencies and Stalls

- After we have the baseline performance of our code, the next step is to add **stalls** as needed to cover extra clock cycles for data dependencies
  - Similar to adding stalls in pipeline for data hazards

MIPS code example

Original HLL code
(C/C++, Java)

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

MIPS assembly language code
Assume: R1 initially points to x[1000] (R1 = j)
8(R2) points to x[1]
F2 holds value s

```
LOOP:   L.D      F0,0(R1)      ;F0 ← x[j]
        ADD.D    F4,F0,F2      ;x[j] ←x[j] + s
        S.D      F4,0(R1)      ;update x[j] in memory
        DADDUI   R1,R1,#-8     ;j– (8 bytes)
        BNE      R1,R2,LOOP    ;branch if not at x[1]
```

Latencies for Example Code

Opcode	Operands	Latency (Extra Clock Cycles
L.D	F0,0(R1)	1
ADD.D	F4,F0,F2	2
S.D	F4,0(R1)	0
DADDUI	R1,R1,#-8	1
BNE	R1,R2,LOOP	0

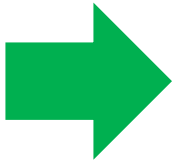
Data Dependencies for Example Code

```
LOOP:   L.D      F0,0(R1)      ;F0 ← x[j]
        ADD.D    F4,F0,F2      ;x[j] ←x[j] + s
        S.D      F4,0(R1)      ;update x[j] in memory
        DADDUI   R1,R1,#-8     ;j– (8 bytes)
        BNE      R1,R2,LOOP    ;branch if not at x[1]
```

1. F0 updated by L.D and used by ADD.D
2. F4 updated by ADD.D and used by S.D
3. R1 updated by DADDUI and used by BNE
    1. (Also used by L.D, S.D., DADDUI in next iteration)

Add Stalls for Example Code

Opcode	Operands	Latency
L.D	F0,0(R1)	1
ADD.D	F4,F0,F2	2
S.D	F4,0(R1)	0
DADDUI	R1,R1,#-8	1
BNE	R1,R2,LOOP	0



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

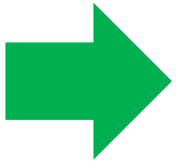
F0 updated by L.D and used by ADD.D

L.D has latency 1 (1 extra clock cycle)

Add 1 stall after L.D

Add Stalls for Example Code

Opcode	Operands	Latency
L.D	F0,0(R1)	1
ADD.D	F4,F0,F2	2
S.D	F4,0(R1)	0
DADDUI	R1,R1,#-8	1
BNE	R1,R2,LOOP	0



Opcode	Operands	Cycle
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
BNE	R1,R2,LOOP	8

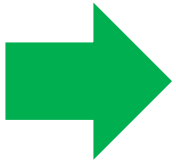
F4 updated by ADD.D and used by S.D

ADD.D has latency 2 (2 extra clock cycles)

Add 2 stalls after ADD.D

Add Stalls for Example Code

Opcode	Operands	Latency
L.D	F0,0(R1)	1
ADD.D	F4,F0,F2	2
S.D	F4,0(R1)	0
DADDUI	R1,R1,#-8	1
BNE	R1,R2,LOOP	0



Opcode	Operands	Cycle
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

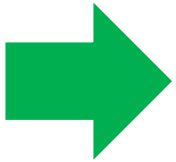
R1 updated by DADDUI and used by BNE

DADDUI has latency 1 (1 extra clock cycle)

Add 1 stall after DADDUI

Add Stalls for Example Code

Opcode	Operands	Latency
L.D	F0,0(R1)	1
ADD.D	F4,F0,F2	2
S.D	F4,0(R1)	0
DADDUI	R1,R1,#-8	1
BNE	R1,R2,LOOP	0



Opcode	Operands	Cycle
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

R1 also referenced by L.D in next iteration of loop.

However, stall after DADDUI covers this issue

How many latency add how many stalls. if they has data dependent. These stalls just for the latency not pipline.

Example 3.2-2: Latencies and Stalls

Label	Opcode	Operands	Latency	Comment
Loop:	L.D	F2,0(Rx)	4	# F2 = Mem[Rx+0]
I0:	MULT.D	F2,F6,F2	5	# F2 = F6*F2
I1:	ADD.D	F4,F8,F2	1	# F4 = F8 + F2
I2:	S.D	F2,0(Ry)	1	# Mem[Ry+0] = F2
I3:	ADDI	R20,R20,-1	0	# R20 = R20 + (-1)
I4:	BNZ	R20,Loop	1	# if R20≠0, go to Loop

We are given MIPS code and instruction latencies as shown.

1. Determine the **data dependencies** between the instructions
2. Using this information, add **stalls** as necessary to the code.

LABEL	OPCODE	OPERANDS	
LOOP:	L.D	F2, 0(Rx)	4 STALLS
I0:	MULD.D	F2, F6, F2	5 STALLS
I1:	ADD.D	F4, F8, F2	(0 STALLS)
I2:	S.D	F2, 0(Ry)	
I3:	ADDI	Rx, Rx, -1	
I4:	BNE	Rx, LOOP	

DATA DEPENDENCIES? F2. Rx

NEXT: ADD STALLS AS NEEDED

F2 DATA DEPENDENCY STALLS?

		LATENCY
L.D	F2, 0(Rx)	4
MULD.D	F2, F6, F2	3

L.D

(4 STALLS)

MULT.D

		LATECY
MULD.D	F2, F6, F2	5
ADD.D	F4, F8, F2	1

MULD.D

(5 STALLS)

ADD.D

ADD.D F4, F8, F2 ← SAME F2 VALUE

S.D F2, 0(R4) ← (DIDN'T CHANGE)

⇒ No STALLS needed.

R2 - STALL?

		(Extra cycles) LATENCY
ADDI	R2, R2, -1	0
BNE	R2, Loop	1

"ADDI" REQUIRE 0 EXTRA CYCLES

SO WE NEED 0 STALLS AFTER "ADDI"

RECALL: WE ARE NOT PIPELINING THESE INSTRUCTIONS.

Scheduling

- Adding stalls is a simple way to address data dependencies and latencies
- However, adding stalls = adding cycles, which affects overall performance
- An alternate approach is to re-arrange the instruction order
  - Also called **instruction re-ordering** or **scheduling**

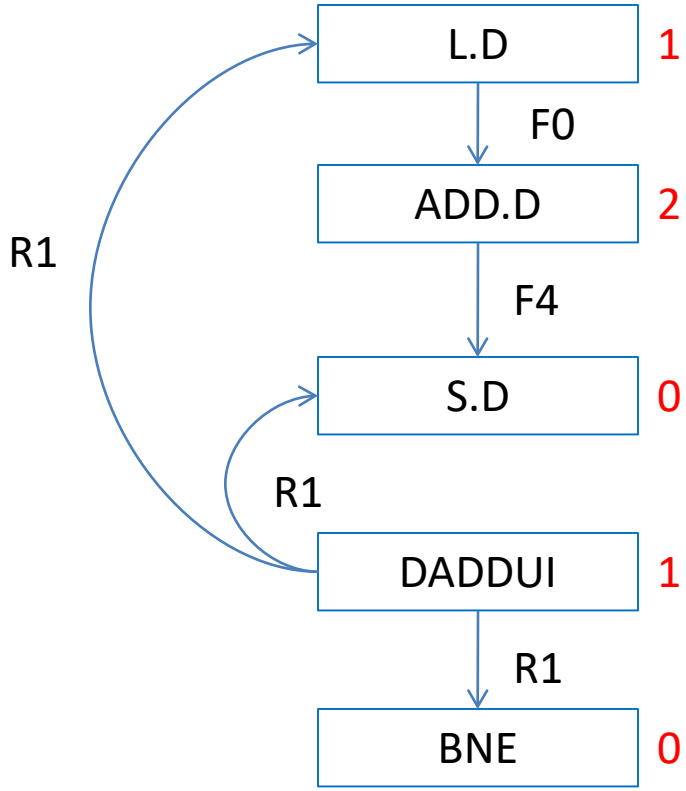
Instruction Dependencies

Opcode	Operands	Latency	Inputs	Outputs
L.D	F0,0(R1)	1	R1	F0
ADD.D	F4,F0,F2	2	F0, F2	F4
S.D	F4,0(R1)	0	F4, R1	(memory)
DADDUI	R1,R1,#-8	1	R1	R1
BNE	R1,R2,LOOP	0	R1,R2	(branch)

Register	Used as Input	Used as Output
R1	L.D, S.D, DADDUI, BNE	DADDUI
R2	BNE	
F0	ADD.D	L.D
F2	ADD.D	
F4	S.D	ADD.D

Graphical Representation of Code

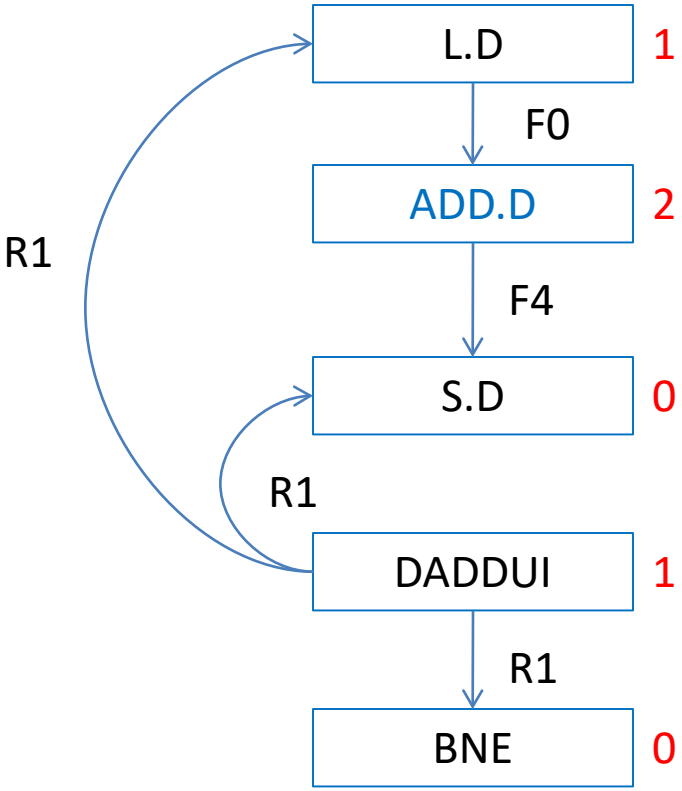
```
LOOP:   L.D      F0,0(R1)
        ADD.D    F4,F0,F2
        S.D      F4, 0(R1)
        DADDUI   R1,R1,#-8
        BNE      R1,R2,LOOP
```



Instruction latencies in RED

x → Input-output dependency of register x

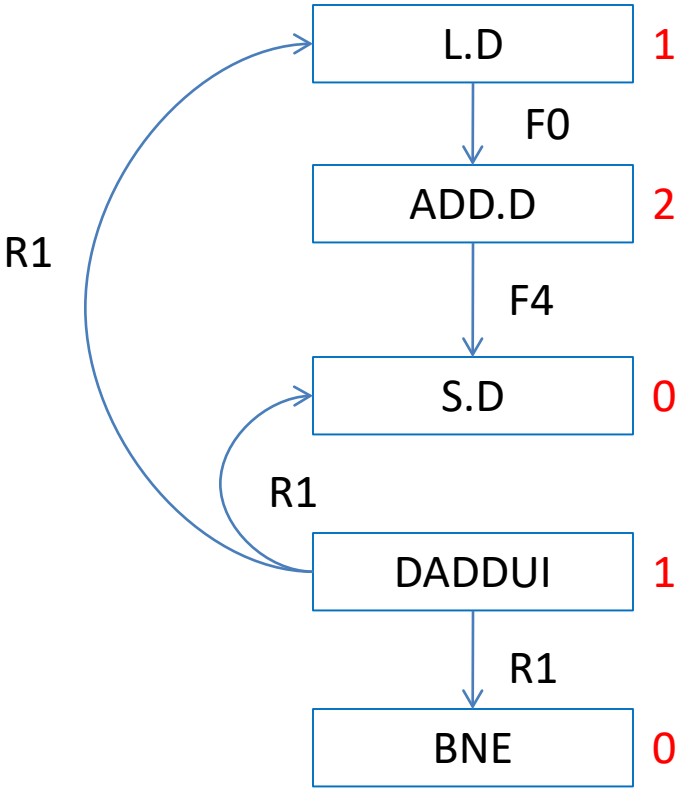
Code Modification



Need to add 2 stalls after ADD.D as before:

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

Using Graph to Revise Code



Note: ADD.D is independent of DADDUI , so both instructions can execute in parallel – **revise code:**

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

Instruction latencies in RED

x → Input-output dependency of register x

Loop Unrolling

- How to handle branch scheduling?
  - "unroll" loops
    - Multiple replications of loop body
  - Eliminates branching (and associated hazards)
  - But: increases code size (more instructions)

Example – Loop Unrolling

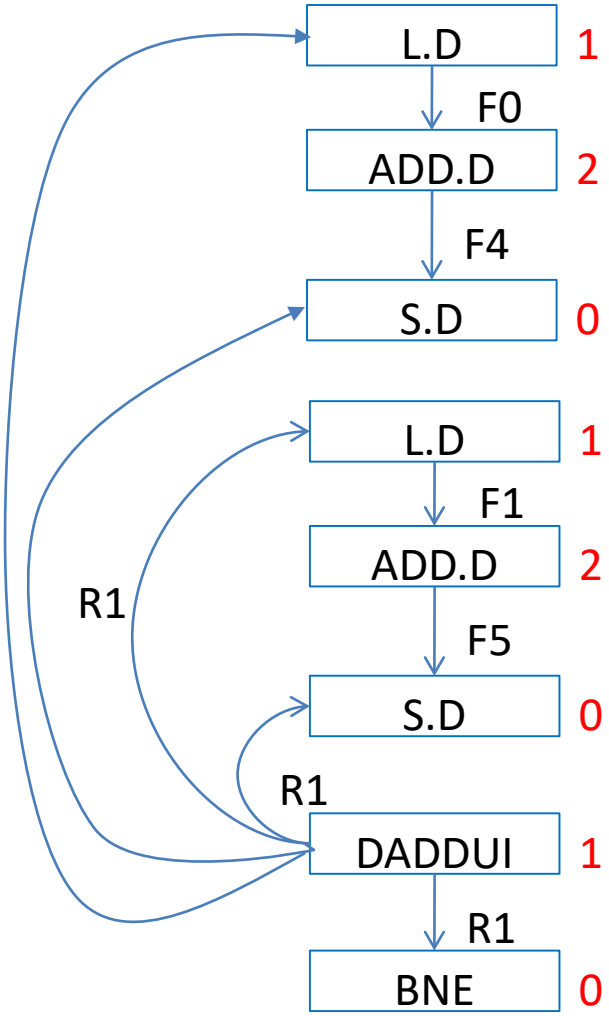
Original Code

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

Now, assume that we know the loop will be executed at least twice – unroll the loop for these two iterations

```
LOOP:   L.D      F0,0(R1)      ;first memory  location
        ADD.D    F4,F0,F2
        S.D      F4,0(R1)
        L.D      F1,-8(R1)     ;second memory location
        ADD.D    F5,F1,F2      ;use new registers (F1, F5)
        S.D      F5,-8(R1)     ;to help with scheduling
        DADDUI   R1,R1,#-16    ;decrement by 2 addresses
        BNE      R1,R2,LOOP
```

Scheduling the Unrolled Loop



Note: can do iterative parts in parallel – just make sure to consider latencies

Due to ADD.D latency

```
LOOP:   L.D      F0,0(R1)
        L.D      F1,-8(R1)
        ADD.D    F4,F0,F2
        ADD.D    F5,F1,F2
        DADDUI   R1,R1,#-16
        S.D      F4,0(R1)
        S.D      F5,-8(R1)
        BNE      R1,R2,LOOP
```

Outline

- 3.1 ILP Background
- 3.2 Basic Compiler Techniques for ILP
- 3.3 Branch Prediction
- 3.4 Data Hazards and Dynamic Scheduling
- 3.5 Dynamic Scheduling Algorithm
- 3.6 Hardware-Based Speculation