

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic look.

# **Computer Architecture**

# Compiler Techniques to Explore ILP

**Dr. Mohammad Reza Selim**

# Lecture Outline

- ▶ Introduction
- ▶ Data and Control Dependence
- ▶ Instruction Scheduling
- ▶ Loop Unrolling

# Introduction

- ❖ Pipelining overlaps execution of instructions
- ❖ Exploits Instruction Level Parallelism (ILP)
- ❖ There are two main approaches:
  - ❖ Compiler-based static approaches
  - ❖ Hardware-based dynamic approaches
- ❖ **Exploiting ILP, goal is to minimize CPI**
  - ❖  $\text{Pipeline CPI} = \text{Ideal (base) CPI} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls}$

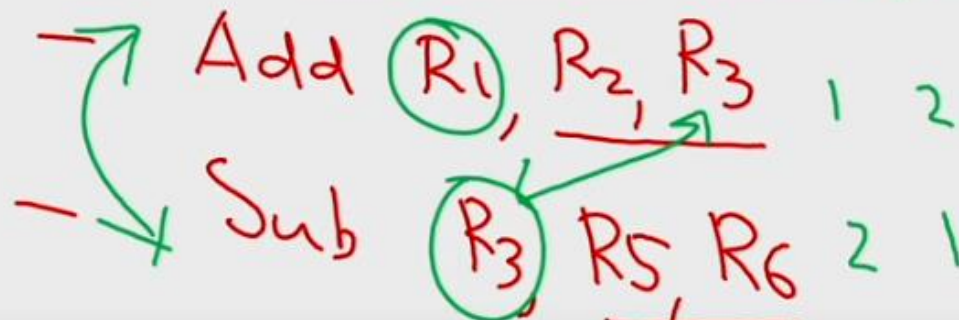
# Data Dependence

- ❖ Loop-Level Parallelism
  - ❖ Unroll loop statically or dynamically
- ❖ Challenges → Data dependency
- ❖ Data dependence conveys possibility of a hazard
- ❖ Dependent instructions cannot be executed simultaneously
- ❖ Pipeline determines if dependence is detected and if it causes a stall or not
- ❖ Data dependence conveys upper bound on exploitable instruction level parallelism



# Name and Output Dependence

- ❖ Two instructions use the same name but no flow of information.
- ❖ Not a true data dependence, but is a problem when reordering instructions



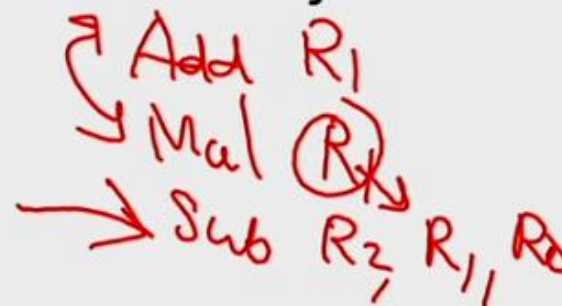
❖ **Antidependence:** instruction j writes a register or memory location that instruction i reads

❖ 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



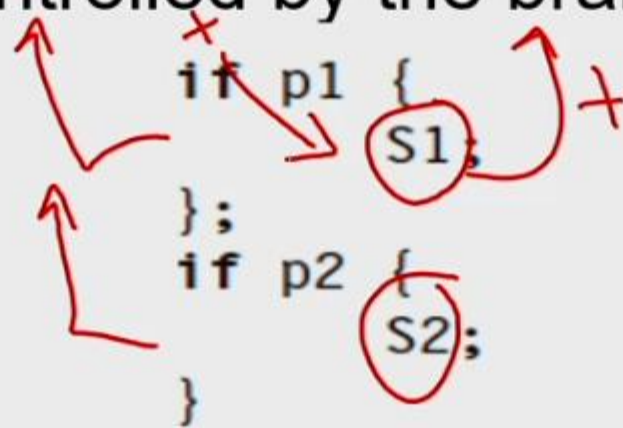
# Control Dependence

- ❖ Ordering of instruction with respect to a branch instruction
  - ❖ Instruction that 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 that is not control dependent on a branch cannot be moved **after** the branch so that its execution is controlled by the branch.

Example :

```
· DADDU R1,R2,R3  
✓ BEQZ R2,L1  
· LW R1, 0(R2)  
L1: ...
```

R15?



$S_1 \rightarrow P_1?$



# Compiler Techniques to Explore ILP

- ❖ Find and overlap sequence of unrelated instruction
- ❖ Pipeline scheduling
  - ❖ Separate dependent instruction from the source instruction by pipeline latency of the source instruction

# Code Example and Pipeline Stalls (1)

## Basic Pipeline Scheduling: Example

```
double x[1000], s;  
for (i=0; i<1000; i++)  
    x[i] = x[i]+s;
```

```
; R1 = x = &x[0]  
; R2 = &x[1000]  
; F2 = s  
for: L.D    F0,0(R1)    ; F0 = x[i]  
      ADD.D F4,F0,F2    ; F4 = x[i]+s  
      S.D    F4,0(R1)   ; x[i] = F4  
      DADDI  R1,R1,8    ; R1 += 8 = &x[i+1]  
      BNE    R1,R2,for  ; if (R1!=&x[1000]) goto for  
      NOP                ; branch delay slot
```



# Code Example and Pipeline Stalls (2)

## Pipeline Assumptions



Producing instr	Consuming instr	Latency in cc
FP ALU	FP ALU	3
FP ALU	Store	2
Load	Any	1
Load	Store	0
Integer	Integer	0
Integer	Branch	1



## Code Example and Pipeline Stalls (3)

## Pipeline Assumptions

Producing instr	Consuming instr	Latency in cc
FP ALU	FP ALU	3
FP ALU	Store	2
Load	Any	1
Load	Store	0
Integer	Integer	0
Integer	Branch	1



# Code Example and Pipeline Stalls (4)

## Pipeline Assumptions

Producing instr	Consuming instr	Latency in cc
FP ALU	FP ALU	3
FP ALU	Store	2
Load	Any	1
Load	Store	0
Integer	Integer	0
Integer	Branch	1




# Code Example and Pipeline Stalls (5)

## How the Loop Will Be Executed

```
for: L.D    F0,0(R1)
      ADD.D  F4,F0,F2
      S.D    F4,0(R1)
      DADDI  R1,R1,8
      BNE    R1,R2,for
      NOP
```

```
for: L.D    F0,0(R1)
      stall
      ADD.D  F4,F0,F2
      stall
      stall
      S.D    F4,0(R1)
      DADDI  R1,R1,8
      stall
      BNE    R1,R2,for
      NOP
```



Producing instr	Consuming instr	Latency
FP ALU	FP ALU	3
FP ALU	Store	2
Load	Any	1
Load	Store	0
Integer	Integer	0
Integer	Branch	1



- 10 cc per iteration
  - 4 stall cycles + branch delay slot



# Pipeline Scheduling

## Schedule Loop to Eliminate Stalls

```
for: L.D    F0,0(R1)
     stall
     ADD.D  F4,F0,F2
     stall
     stall
     S.D    F4,0(R1)
     DADDI  R1,R1,8
     stall
     BNE    R1,R2,for
     NOP
```

```
for: L.D    F0,0(R1)
     DADDI  R1,R1,8
     ADD.D  F4,F0,F2
     stall
     stall
     S.D    F4,-8(R1)
     BNE    R1,R2,for
     NOP
```

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

- 6 cc per iteration but only 3 instrs do actual work
- Employ **loop unrolling** to
  - reduce loop overhead
  - improve potential for instr scheduling

# Loop Unrolling (1)

- Replicate loop body multiple times, adjusting loop termination code

```
for (i=0; i<1000; i++)  
    x[i] = x[i]+s;
```



```
for (i=0; i<1000; i+=4){  
    x[i] = x[i]+s;  
    x[i+1] = x[i+1]+s;  
    x[i+2] = x[i+2]+s;  
    x[i+3] = x[i+3]+s;  
}
```

# Loop Unrolling (2)

## Loop Unrolling — General Case

- Loop count  $n$  can be unknown or no multiple of loop unrolling factor  $k$
- Then need 2 loops:
  - One unrolled that iterates  $n \div k$  times (e.g.  $7 \div 3 = 2$ )
  - Copy of original that iterates  $n \bmod k$  times (e.g.  $7 \bmod 3 = 1$ )

```
for (i=0; i<n; i++)  
    x[i] = x[i]+s;
```



```
for (i=0; i+k-1<n; i+=k){  
    x[i] = x[i]+s;  
    x[i+1] = x[i+1]+s;  
    ...;  
    x[i+k-1] = x[i+k-1]+s;  
}
```

```
for (/* empty */; i<n; i++)  
    x[i] = x[i]+s;
```

# Loop Unrolling (3)

## Unrolled Example Loop

```
for: L.D    F0,0(R1)
      ADD.D  F4,F0,F2
      S.D    F4,0(R1)
      DADDI  R1,R1,8
      BNE    R1,R2,for
      NOP
```

```
for:  L.D    F0, 0(R1)
      ADD.D  F4,F0,F2
      S.D    0(R1),F4
      L.D    F0,8(R1)
      ADD.D  F4,F0,F2
      S.D    8(R1),F4
      L.D    F0,16(R1)
      ADD.D  F4,F0,F2
      S.D    16(R1),F4
      L.D    F0,24(R1)
      ADD.D  F4,F0,F2
      S.D    24(R1),F4
      DADDI  R1,R1,#32
      BNE    R1,R2,for
      NOP
```

Annotations for the unrolled loop:

- 1 cycle stall (between L.D and ADD.D)
- 2 cycles stall (between ADD.D and S.D)
- 1 cycle stall (between S.D and L.D)
- 2 cycles stall (between L.D and ADD.D)
- 2 cycles stall (between ADD.D and S.D)
- 1 cycle stall (between S.D and L.D)
- 2 cycles stall (between L.D and ADD.D)
- 2 cycles stall (between ADD.D and S.D)
- 1 cycle stall (between S.D and DADDI)



# Loop Unrolling (4)

## After Register Renaming

```
for:  L.D    F0, 0(R1)
      ADD.D  F4, F0, F2
      S.D    0(R1), F4
      L.D    F0, 8(R1)
      ADD.D  F4, F0, F2
      S.D    8(R1), F4
      L.D    F0, 16(R1)
      ADD.D  F4, F0, F2
      S.D    16(R1), F4
      L.D    F0, 24(R1)
      ADD.D  F4, F0, F2
      S.D    24(R1), F4
      DADDI  R1, R1, #32
      BNE    R1, R2, for
      NOP
```


```
for:  L.D    F0, 0(R1)
      ADD.D  F4, F0, F2
      S.D    0(R1), F4
      L.D    F6, 8(R1)
      ADD.D  F8, F6, F2
      S.D    8(R1), F8
      L.D    F10, 16(R1)
      ADD.D  F12, F10, F2
      S.D    16(R1), F12
      L.D    F14, 24(R1)
      ADD.D  F16, F14, F2
      S.D    24(R1), F16
      DADDI  R1, R1, #32
      BNE    R1, R2, for
      NOP
```

# Loop Unrolling (5)

## Schedule Unrolled Loop to Avoid Stalls

```
for:  L.D    F0,0(R1)
      ADD.D  F4,F0,F2
      S.D    0(R1),F4
      L.D    F6,8(R1)
      ADD.D  F8,F6,F2
      S.D    8(R1),F8
      L.D    F10,16(R1)
      ADD.D  F12,F10,F2
      S.D    16(R1),F12
      L.D    F14,24(R1)
      ADD.D  F16,F14,F2
      S.D    24(R1),F16
      DADDI  R1,R1,#32
      BNE    R1,R2,for
      NOP
```

```
for:  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    0(R1),F4
      S.D    8(R1),F8
      DADDI  R1,R1,#32
      S.D    -16(R1),F12
      BNE    R1,R2,for
      S.D    -8(R1),F16
```

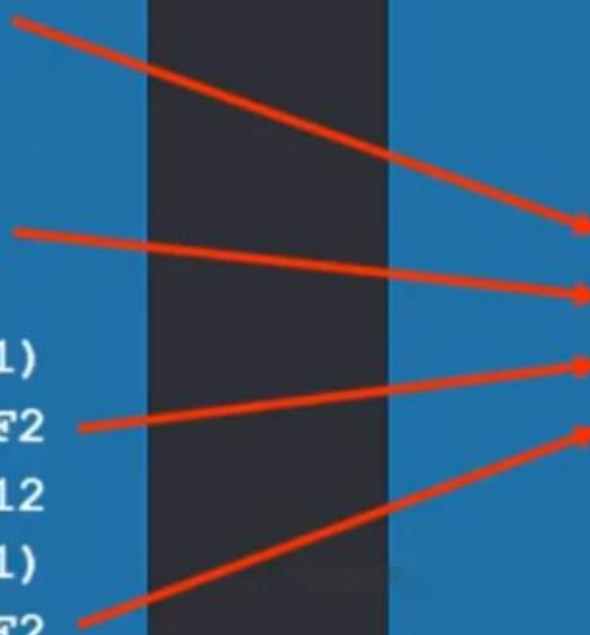


# Loop Unrolling (6)

## Schedule Unrolled Loop to Avoid Stalls

```
for:  L.D    F0,0 (R1)
      ADD.D  F4,F0,F2
      S.D    0 (R1),F4
      L.D    F6,8 (R1)
      ADD.D  F8,F6,F2
      S.D    8 (R1),F8
      L.D    F10,16 (R1)
      ADD.D  F12,F10,F2
      S.D    16 (R1),F12
      L.D    F14,24 (R1)
      ADD.D  F16,F14,F2
      S.D    24 (R1),F16
      DADDI  R1,R1,#32
      BNE    R1,R2,for
      NOP
```

```
for:  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    0 (R1),F4
      S.D    8 (R1),F8
      DADDI  R1,R1,#32
      S.D    -16 (R1),F12
      BNE    R1,R2,for
      S.D    -8 (R1),F16
```






# Loop Unrolling (7)

## Schedule Unrolled Loop to Avoid Stalls

```
for:  L.D    F0,0(R1)
      ADD.D  F4,F0,F2
      S.D    0(R1),F4
      L.D    F6,8(R1)
      ADD.D  F8,F6,F2
      S.D    8(R1),F8
      L.D    F10,16(R1)
      ADD.D  F12,F10,F2
      S.D    16(R1),F12
      L.D    F14,24(R1)
      ADD.D  F16,F14,F2
      S.D    24(R1),F16
      DADDI  R1,R1,#32
      BNE    R1,R2,for
      NOP
```

```
for:  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    0(R1),F4
      S.D    8(R1),F8
      DADDI  R1,R1,#32
      S.D    -16(R1),F12
      BNE    R1,R2,for
      S.D    -8(R1),F16
```






# Loop Unrolling (8)

## Schedule Unrolled Loop to Avoid Stalls

```
for:  L.D    F0,0(R1)
      ADD.D  F4,F0,F2
      S.D    0(R1),F4
      L.D    F6,8(R1)
      ADD.D  F8,F6,F2
      S.D    8(R1),F8
      L.D    F10,16(R1)
      ADD.D  F12,F10,F2
      S.D    16(R1),F12
      L.D    F14,24(R1)
      ADD.D  F16,F14,F2
      S.D    24(R1),F16
      DADDI  R1,R1,#32
      BNE    R1,R2,for
      NOP
```

```
for:  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    0(R1),F4
      S.D    8(R1),F8
      DADDI  R1,R1,#32
      S.D    -16(R1),F12
      BNE    R1,R2,for
      S.D    -8(R1),F16
```




# Loop Unrolling (9)

## Schedule Unrolled Loop to Avoid Stalls

```
for:  L.D    F0,0(R1)
      ADD.D  F4,F0,F2
      S.D    0(R1),F4
      L.D    F6,8(R1)
      ADD.D  F8,F6,F2
      S.D    8(R1),F8
      L.D    F10,16(R1)
      ADD.D  F12,F10,F2
      S.D    16(R1),F12
      L.D    F14,24(R1)
      ADD.D  F16,F14,F2
      S.D    24(R1),F16
      DADDI  R1,R1,#32
      BNE    R1,R2,for
      NOP
```

```
for:  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    0(R1),F4
      S.D    8(R1),F8
      DADDI  R1,R1,#32
      S.D    -16(R1),F12
      BNE    R1,R2,for
      S.D    -8(R1),F16
```



# Loop Unrolling (10)

## Final Scheduled Code for Example Loop

```
for:   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    0(R1),F4
        S.D    8(R1),F8
        DADDI  R1,R1,#32
        S.D    -16(R1),F12
        BNE    R1,R2,for
        S.D    -8(R1),F16
```

- Runs in 14 cc (no stalls) per iteration
  - $14/4 = 3.5$  cc per element
- Made possible by:
  - moving all loads before all ADD.D
  - moving 1 S.D between DADDI and BNE
  - moving 1 S.D in branch delay slot
  - using different registers
- When safe for compiler to do such changes?



# Loop Unrolling (11)

## Steps in Loop Unrolling and Scheduling

- Determine that loop unrolling would be useful by finding that loop iterations were independent
- Use different registers to avoid name dependencies
- Eliminate extra test & branch instrs and adjust loop termination and iteration code
- Determine that it is possible to move S.D after DADDI and BNE, and find amount to adjust S.D offset
- Determine that lds and sts in unrolled loop can be interchanged by observing that lds and sts from different iterations are independent
  - requires analyzing memory addresses to find that they do not refer to the same address
- Schedule the code, preserving any dependences needed to yield same result as original code



# Loop Unrolling (12)

## Pros and Cons of Loop Unrolling

- Advantages:

- reduces loop overhead
- improves potential for instruction scheduling

- Disadvantages:

- increases code size
  - concern in embedded domain
  - may increase instruction cache miss rate
- shortfall in registers
  - register pressure



# Loop Unrolling (9)

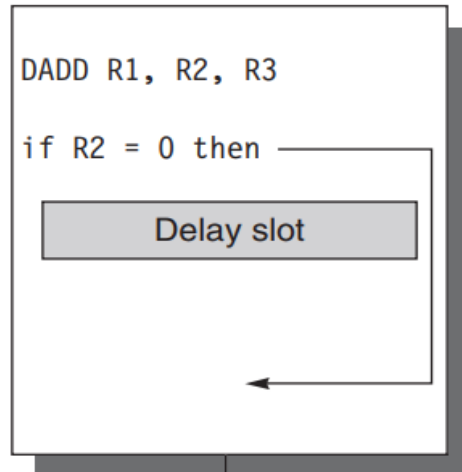
## Schedule Unrolled Loop to Avoid Stalls

```
for:  L.D    F0,0(R1)
      ADD.D  F4,F0,F2
      S.D    0(R1),F4
      L.D    F6,8(R1)
      ADD.D  F8,F6,F2
      S.D    8(R1),F8
      L.D    F10,16(R1)
      ADD.D  F12,F10,F2
      S.D    16(R1),F12
      L.D    F14,24(R1)
      ADD.D  F16,F14,F2
      S.D    24(R1),F16
      DADDI  R1,R1,#32
      BNE    R1,R2,for
      NOP
```

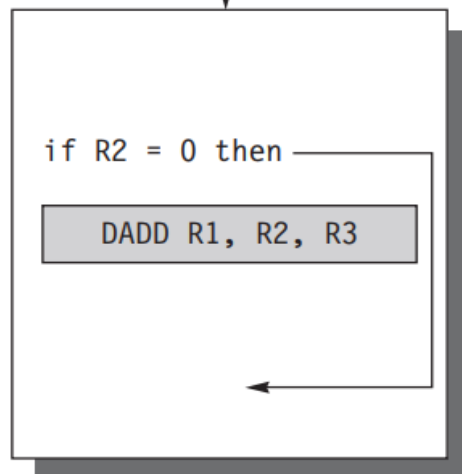
Why is this valid to move an instruction from before branch to after branch?

```
for:  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    0(R1),F4
      S.D    8(R1),F8
      DADDI  R1,R1,#32
      S.D    -16(R1),F12
      BNE    R1,R2,for
      S.D    -8(R1),F16
```

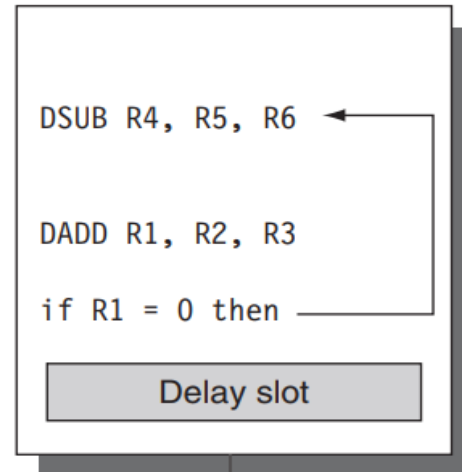
# Filling Branch Delay Slots



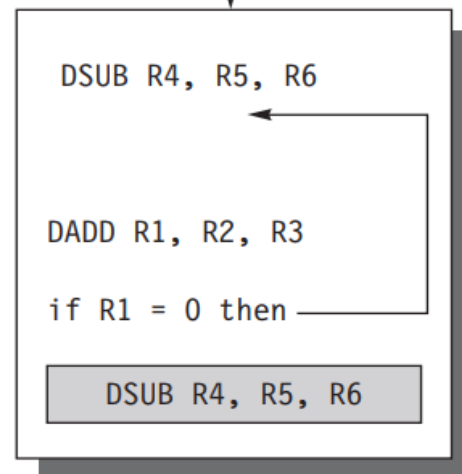
becomes



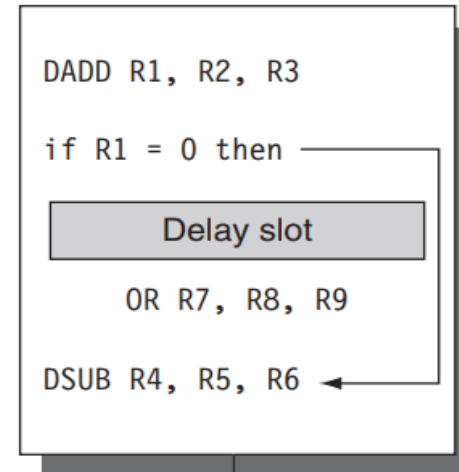
(a) From before



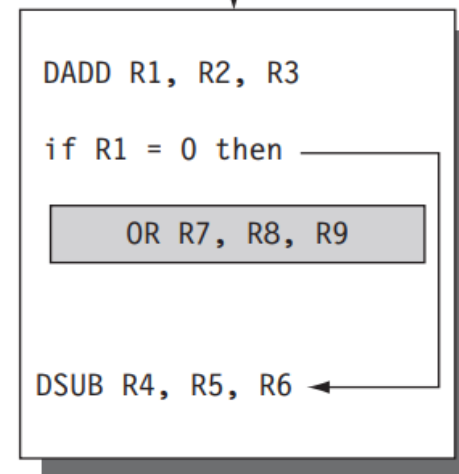
becomes



(b) From target



becomes



(c) From fall-through

# Pipeline Sequence for Taken/Untaken Branch

SHOWS BOTH SITUATIONS.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB
Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

**Figure C.12** The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we fetch the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.



# Loop Unrolling (9)

## Schedule Unrolled Loop to Avoid Stalls

```
for:  L.D    F0,0(R1)
      ADD.D  F4,F0,F2
      S.D    0(R1),F4
      L.D    F6,8(R1)
      ADD.D  F8,F6,F2
      S.D    8(R1),F8
      L.D    F10,16(R1)
      ADD.D  F12,F10,F2
      S.D    16(R1),F12
      L.D    F14,24(R1)
      ADD.D  F16,F14,F2
      S.D    24(R1),F16
      DADDI  R1,R1,#32
      BNE   R1,R2,for
      NOP
```

Valid because S.D.  
instruction will be  
executed whether  
branch is taken or  
not taken.

```
for:  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    0(R1),F4
      S.D    8(R1),F8
      DADDI  R1,R1,#32
      S.D    -16(R1),F12
      BNE   R1,R2,for
      S.D    -8(R1),F16
```