

3.2 Basic Compiler Techniques for Exposing ILP


- Consider the following statements:

$a = b + c; d = e - f;$

- Slow code:

```
lw    $10, ($1) # $1= addr b
lw    $11, ($2) # $2      = addr c
add   $12, $10, $11 # Dependence
sw    $12, ($3) # $3      = addr a
lw    $13, ($4) # $4      = addr e
lw    $14, ($5) # $5      = addr f
sub   $15, $13, $14 # Dependence
sw    $15, ($6) # $6      = addr d
```

```
lw    $10,0($1)
lw    $11,  0($2)
lw    $13,0($4)
lw    $14,0($5)
add$12, $10, $11
sw    $12,0($3)
sub   $15, $13, $14
sw    $14, 0($6)
```



To avoid a pipeline stall, the execution of a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction.

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

- Each iteration is independent; adding a scalar to a vector

Loop: L.D F0,0(R1)	;F0=array element
ADD F4,F0,F2	;add scalar from F2
SD F4,0(R1)	;store result
DADDUI R1,R1,#-8	;decrement ptr 8B
BNE R1,R2,Loop	;branch R1!=R2

- Assume the following FP latencies (averages):

Producer	Consumer	Latency (CCs)
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

Loop Example: Unscheduled Code

```
1. Loop: L.D F0,0(R1)
2. stall                ;load interlock
3. ADD.D F4,F0,F2
4. stall                ;data hazard (F4)
5. stall                ;data hazard (F4)
6. S.D F4,0(R1)
7. DADDUI R1,R1,#-8
8. stall                ;branch interlock
9. BNE R1,R2,Loop
```

Loop Example: Scheduled Code

```
1. Loop: L.D F0,0(R1)
2. DADDUI R1,R1,#-8
3. ADD.D F4,F0,F2
4. stall ;data hazard (F4)
5. stall ;data hazard (F4)
6. S.D F4,8(R1) ;0+8 = 8
7. BNE R1,R2,Loop
```

- Still the actual work of operating on the array element takes just three (the **load**, **add**, and **store**) of those seven clock cycles.
- The remaining four clock cycles consist of loop overhead—the **DADDUI** and **BNE**—and *two stalls*.

Loop Unrolling

- **Unroll the loop**
 - Replicate the body of the loop many times
 - Adjust the loop termination code
- Eliminating the branch allows instructions from different iterations to be scheduled together.
 - In this case we can eliminate the data stall
 - We also increase the ratio of useful work to overhead
 - Doing so requires more registers

First Step: Copy the Loop

```
1 Loop: L.D      F0, 0(R1)
2        ADD.D   F4, F0, F2
3        S.D     F4, 0(R1)      ;drop DADDUI & BNE
4        L.D     F0, -8(R1)
5        ADD.D   F4, F0, F2
6        S.D     F4, -8(R1)     ;drop DADDUI & BNE
7        L.D     F0, -16(R1)
8        ADD.D   F4, F0, F2
9        S.D     F4, -16(R1)    ;drop DADDUI & BNE
10       L.D     F0, -24(R1)
11       ADD.D   F4, F0, F2
12       S.D     F4, -24(R1)
13       DADDUI  R1, R1, #-32    ; -8*4 = 32
14       BNE     R1, R2, LOOP
15       NOP
```

Second Step: Find Name Dependencies

```
1 Loop: L.D      F0, 0(R1)      ;1st iteration
2      ADD.D     F4, F0, F2
3      S.D       F4, 0(R1)
4      L.D       F0, -8(R1)     ;2nd iteration
5      ADD.D     F4, F0, F2
6      S.D       F4, -8(R1)
7      L.D       F0, -16(R1)   ;3rd iteration
8      ADD.D     F4, F0, F2
9      S.D       F4, -16(R1)
10     L.D       F0, -24(R1)
11     ADD.D     F4, F0, F2     ;4th iteration
12     S.D       F4, -24(R1)
13     DADDUI    R1, R1, #-32
14     BNE       R1, R2, LOOP
15     NOP
```


Third Step: Register Renaming

```
1 Loop: L.D      F0, 0(R1)
2        ADD.D   F4, F0, F2
3        S.D     F4, 0(R1)
4        L.D     F6, -8(R1)      ;renamed F0
5        ADD.D   F8, F6, F2      ;renamed F4, F0
6        S.D     F8, -8(R1)      ;renamed F4
7        L.D     F10, -16(R1)    ;renamed F0
8        ADD.D   F12, F10, F2    ;renamed F4, F0
9        S.D     F12, -16(R1)    ;renamed F4
10       L.D     F14, -24(R1)    ;renamed F0
11       ADD.D   F16, F14, F2    ;renamed F4, F0
12       S.D     F16, -24(R1)    ;renamed F4
13       DADDUI  R1, R1, #-32
14       BNE     R1, R2, LOOP
15       NOP
```

Unrolled: Unscheduled

1	Loop:	L.D	F0, 0(R1)	← 1 cycle stall (load interlock)
2		ADD.D	F4, F0, F2	← 2 cycles stall (data hazard)
3		S.D	F4, 0(R1)	
4		L.D	F6, -8(R1)	← 1 cycle stall (load interlock)
5		ADD.D	F8, F6, F2	← 2 cycles stall (data hazard)
6		S.D	F8, -8(R1)	
7		L.D	F10, -16(R1)	← 1 cycle stall (load interlock)
8		ADD.D	F12, F10, F2	← 2 cycles stall (data hazard)
9		S.D	F12, -16(R1)	
10		L.D	F14, -24(R1)	← 1 cycle stall (load interlock)
11		ADD.D	F16, F14, F2	← 2 cycles stall (data hazard)
12		S.D	F16, -24(R1)	
13		DADDUI	R1, R1, #-32	
14		BNE	R1, R2, LOOP	← 1 cycle stall (branch interlock)
15		NOP		

15 + 4(1 + 2) + 1 = 28 cycles, or 7 cycles per iteration!

Unrolled: Scheduled

```

1 Loop: L.D      F0, 0(R1)
2        L.D      F6, -8(R1)
3        L.D      F10, -16(R1)
4        L.D      F14, -24(R1)
5        ADD.D     F4, F0, F2
6        ADD.D     F8, F6, F2
7        ADD.D     F12, F10, F2
8        ADD.D     F16, F14, F2
9        S.D       F4, 0(R1)
10       S.D       F8, -8(R1)
11       DADDUI    R1, R1, #-32
12       S.D       F12, 16(R1)    ; -16+32 = 16
13       BNE      R1, R2, LOOP
14       S.D       F16, 8(R1)     ; -24+32 = 8

```

14 cycles, or 3.5 cycles per iteration!

Loop Unrolling is not Free

- ❑ Benefits decrease with additional unrolling
- ❑ Code length increases with additional unrolling
 - ❑ This is issue for embedded processors
 - ❑ This can increase instruction cache miss rates
- ❑ Uses lots of registers
 - ❑ Renaming requires many registers
 - ❑ When registers become scarce: “register pressure”
 - ❑ Aggressive unrolling and scheduling and cause a compiler to run out of registers to use for renaming.

Compiler's Perspective: Unrolling Loops

- We don't usually know the upper bound of a loop
- Suppose it is n , and we want k copies of the loop
- Don't generate a single unrolled loop!
- Generate a pair of consecutive loops:
 - First executes the original loop $(n \bmod k)$ times
 - Second executes the unrolled loop body (n/k) times
 - For large n , most iterations occur in unrolled loop

Compiler Perspectives: Dependencies

- Compilers must preserve data dependencies
 - Determine if loop iterations are independent
 - Rename registers during unrolling
 - Eliminate extra test and branch instructions
 - Adjust loop maintenance and termination accordingly
- Determine if loads and stores can be interchanged
- Schedule the code, preserving dependencies

Compiler Perspectives: Renaming

- Dependent instructions can't execute in parallel
 - Easy to determine for registers (fixed names)
 - Much harder for memory
 - This is the “memory disambiguation” problem
 - Does $100(R4) = 20(R6)$?
 - In different loop iterations, does $20(R6) = 20(R6)$?
- In our example, compiler must determine that if R1 doesn't change then:
 $0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$
- In this case, loads and stores can be interchanged

Loop Unrolling Summary

- Loop unrolling is a simple but useful method for increasing the size of straight-line code fragments that can be scheduled effectively.
- Looping Unrolling
 - Reduces loop overhead
 - Exposes additional instructions for scheduling
- Compiler unrolls a loop by:
 - Copying the loop
 - Identifying and resolving name dependencies
 - Scheduling the loop
- Compiler must identify and preserve true data dependencies

3.4 Overcoming Data Hazards by Dynamic Scheduling

- With *dynamic scheduling*, the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior.

DIV.D	F0, F2, F4
ADD.D	F10, F0, F8
SUB.D	F12, F8, F14

- In the classic five-stage pipeline, both structural and data hazards could be checked during instruction decode (ID):
 - When an instruction could execute without hazards, it was issued from ID knowing that all data hazards had been resolved.