CS425
Computer Systems Architecture

Fall 2017

Static Instruction Scheduling

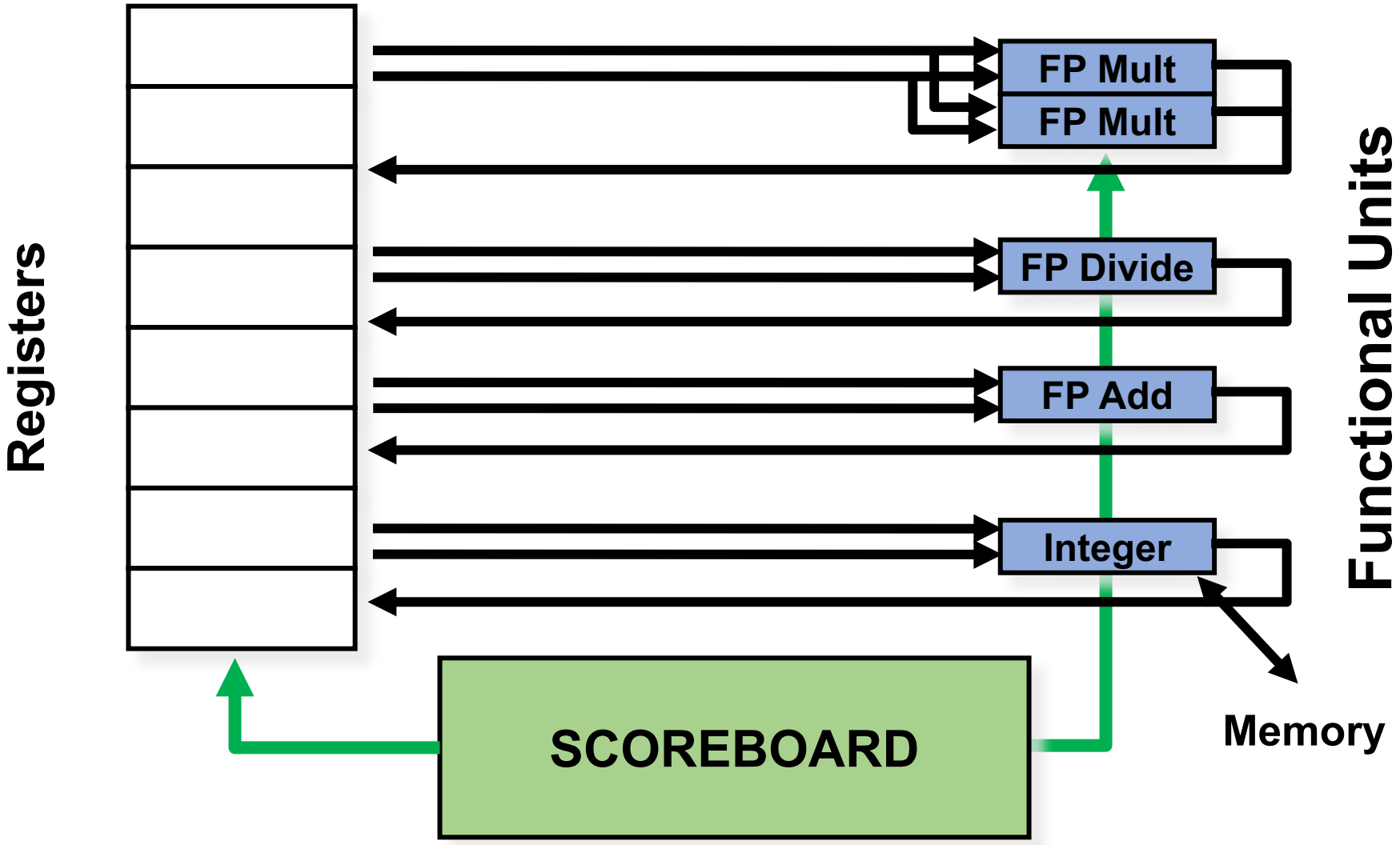
Techniques to reduce stalls

CPI = Ideal CPI + Structural stalls per instruction + RAW stalls per instruction + WAR stalls per instruction + WAW stalls per instruction

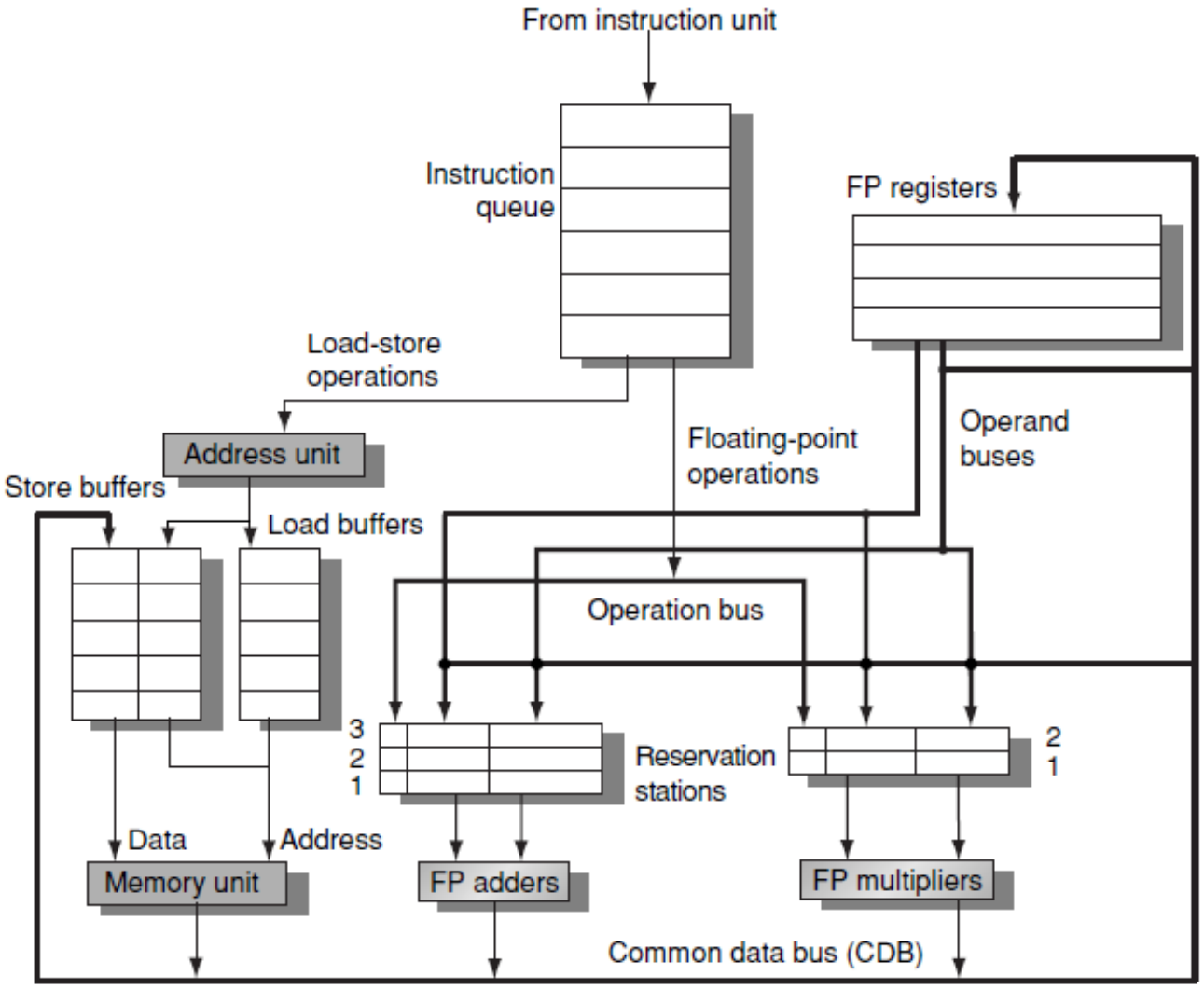
We will study two types of techniques:

Dynamic instruction scheduling	Static instruction scheduling (SW/compiler)
Scoreboard (reduce RAW stalls)	Loop Unrolling
Register Renaming (reduce WAR & WAW stalls) • Tomasulo • Reorder buffer	SW pipelining
Branch Prediction (reduce control stalls)	Trace Scheduling

Scoreboard Architecture (CDC 6600)



Tomasulo Organization



Εξαρτήσεις Μεταξύ Εντολών (Depedences)

- Ποιές είναι οι πηγές των stalls/bubbles;
  - εντολές που χρησιμοποιούν ίδιους registers
- ***Παράλληλες*** εντολές μπορούν να εκτελεστούν διαδοχικά χωρίς να δημιουργούν stalls (αγνοώντας structural hazards)
  - DIV.D F0, F2, F4
  - ADD.D F10, F1, F3
- ***Εξαρτήσεις*** μεταξύ εντολών μπορούν να οδηγήσουν σε stalls
  - DIV.D F0, F2, F4
  - **RAW**
  - ADD.D F10, F0, F3
- Οι εξαρτήσεις μεταξύ εντολών περιορίζουν την σειρά εκτέλεσης των εντολών (in order execution) π.χ. η ADDD πρέπει να εκτελεστεί μετά την DIVD στο 2ο παράδειγμα. Ενώ παράλληλες εντολές μπορούν να εκτελεστούν ανάποδα (out-of-order execution) π.χ. η ADDD μπορεί να εκτελεστεί πριν την DIVD στο 1ο παράδειγμα.

Εξαρτήσεις Μεταξύ Εντολών

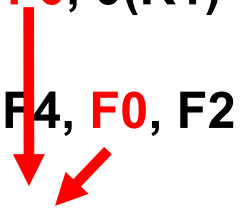
- **Data Dependences** : Δύο εντολές είναι data dependent όταν υπάρχει μία αλυσίδα από RAW hazards μεταξύ τους.

- **Name Dependences** : Δύο εντολές είναι name dependent όταν υπάρχει ένα WAR (antidependence) ή WAW (output dependence) hazard μεταξύ τους.

```
L.D      F0, 0(R1)

ADD.D F4, F0, F2

L.D      F0, 0(R2)
```

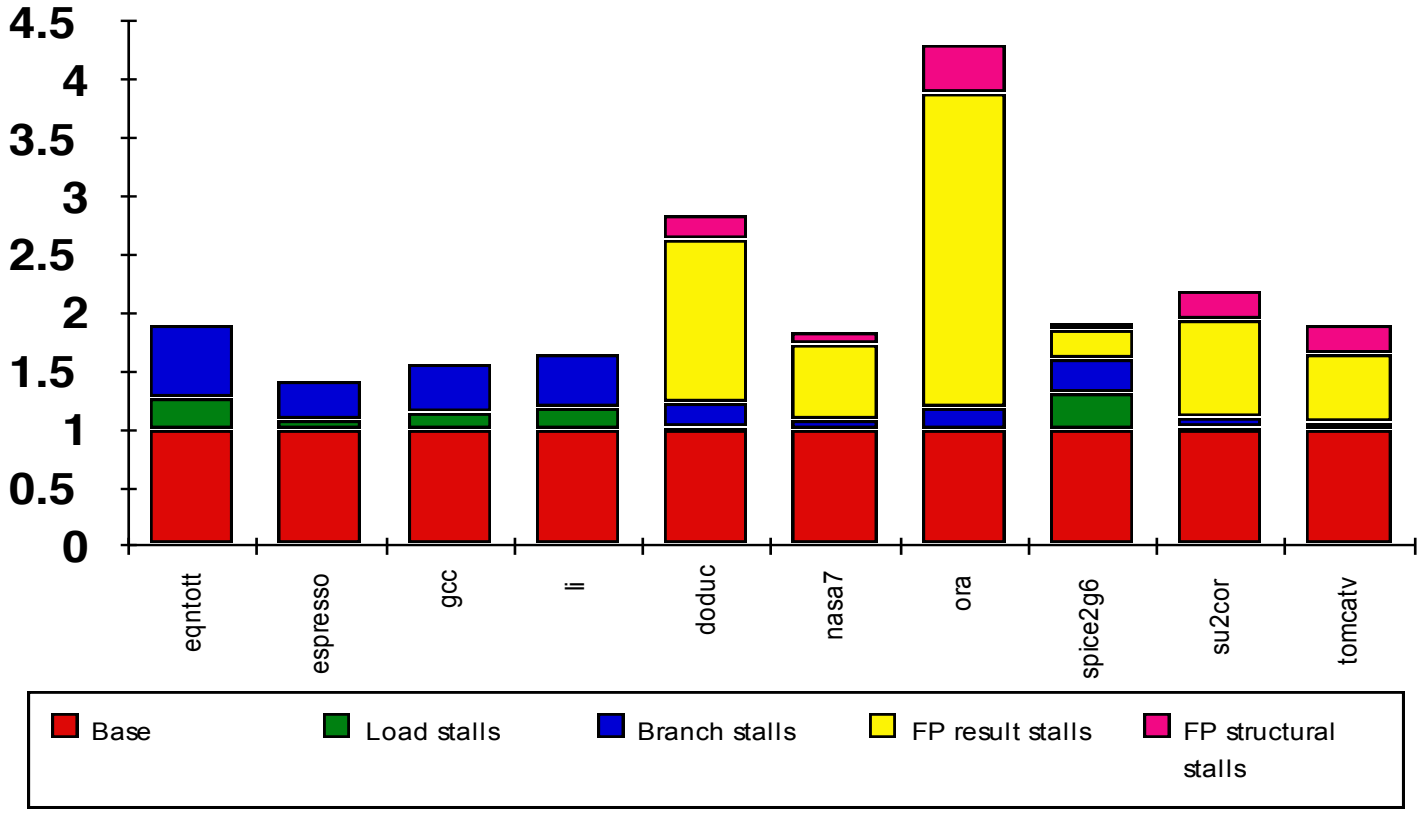


- **Control Dependences** : Εντολές εξαρτώμενες από branch εντολή.

```
if  p1 { S1; }
```

R4000 Performance

- Μη ιδανικό CPI :
  - **Load stalls:**
    1 ή 2 clock cycles
  - **Branch stalls:**
    2 cycles + unfilled slots
  - **FP result stalls:**
    RAW data hazard (latency)
  - **FP structural stalls:**
    Not enough FP hardware (parallelism)



Instruction Level Parallelism (ILP)

- ILP: Παράλληλη εκτέλεση μη σχετιζόμενων (μη εξαρτώμενων) εντολών

- gcc 17% control transfer εντολές
  - 5 εντολές + 1 branch
  - πέρα από ένα block για να έχουμε περισσότερο instruction level parallelism

- Loop level parallelism one opportunity
  - First SW, then HW approaches

FP Loop: Που είναι τα Hazards?

```
while (R1 > 0) { M[R1] = M[R1] + F2; R1 -= 8 }
Loop:  L.D     F0,0(R1) ;F0=vector element
       ADD.D   F4,F0,F2 ;add scalar from F2
       S.D     0(R1),F4 ;store result
       SUBI    R1,R1,8  ;decrement pointer 8B (DW)
       BNEZ    R1,Loop  ;branch R1!=zero
       NOP              ;branch delay slot
```

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	1
Integer op	Integer op	0

Stalls?

Assume 5-stage DLX (in order)

FP Loop Showing Stalls

```
1 Loop: L.D    F0,0(R1)   ;F0=vector element
2       stall
3       ADD.D  F4,F0,F2   ;add scalar in F2
4       stall
5       stall
6       S.D    0(R1), F4  ;store result
7       SUBI   R1,R1,8    ;decrement pointer 8B (DW)
8       BNEZ   R1,Loop    ;branch R1!=zero
9       stall             ;branch delay slot
```

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

9 κύκλοι:

Ξαναγράψτε τον κώδικα για να ελαχιστοποιήσετε τα stalls!

Scheduled κώδικας του FP Loop

```
1 Loop: L.D    F0,0(R1)
2       stall
3       ADD.D  F4,F0,F2
4       SUBI   R1,R1,8
5       BNEZ   R1,Loop    ;delayed branch
6       S.D    8(R1),F4   ;altered when move past SUBI
```

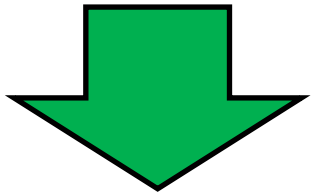
Αλλαγή θέσεων των BNEZ και SD αλλάζοντας την διεύθυνση του SD

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

6 clocks: Unroll loop 4 times to make code faster?

Ιδέα : Loop Unrolling

```
while (R1 > 0) { M[R1] = M[R1] + F2; R1 -= 8 }
```



```
while (R1 >= 4*8) {
  M[R1] = M[R1] + F2;
  M[R1-8] = M[R1-8] + F2;
  M[R1-16] = M[R1-16] + F2;
  M[R1-24] = M[R1-24] + F2;
  R1 -= 4*8
}
while (R1 > 0) { M[R1] = M[R1] + F2; R1 -= 8 }
```

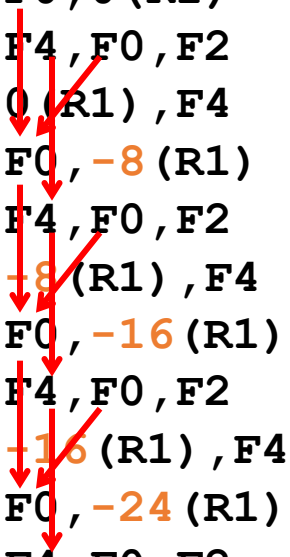
Μη εξαρτώμενες εντολές μέσα στο Loop. Καλές προοπτικές για scheduling.

Unroll Loop 4 times: name dependencies?

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

Unroll Loop 4 times: name dependencies?

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



How to deal with these?

No name dependencies now!

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

“register renaming” removed WAR/WAW stalls

Unroll Loop 4 times

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

1 cycle stall

2 cycles stall

eliminates overhead instructions, but increases code size

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration

Assumes R1 is multiple of 4

Schedule Unrolled Loop

```
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    0(R1),F4
10     S.D    -8(R1),F8
11     S.D    -16(R1),F12
12     SUBI   R1,R1,#32
13     BNEZ   R1,LOOP
14     S.D    8(R1),F16 ; 8-32 = -24
```

14 clock cycles, or 3.5 per iteration

- **Τι υποθέσεις έγιναν κατά την μετακίνηση του κώδικά;**
  - OK to move store past SUBI even though changes register
  - **OK to move loads before stores/add: get right data?**
  - When is it safe for compiler to do such changes?

Compiler Perspectives on Code Movement

- Name Dependencies είναι δύσκολο να διαγνωστούν για Memory Accesses
  - Είναι 100(R4) = 20(R6)?
  - Για διαφορετικές επαναλήψεις του loop, είναι 20(R6) = 20(R6)?
- Στο παράδειγμά μας ο compiler πρέπει να καταλάβει ότι όταν το R1 δεν αλλάζει τότε:

```
0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)
```

- There were no dependencies between some loads and stores so they could be moved by each other

When is it safe to unroll a loop?

- Παράδειγμα: Που είναι οι εξαρτήσεις? (A,B,C distinct & non-overlapping)

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];     /* S1 */
      B[i+1] = B[i] + A[i+1];   /* S2 */
}
```

- `S2` uses the value, `A[i+1]`, computed by `S1` in the same iteration.
- `S1` uses a value computed by `S1` in an earlier iteration, since iteration `i` computes `A[i+1]` which is read in iteration `i+1`. The same is true of `S2` for `B[i]` and `B[i+1]`. Αυτή η εξάρτηση (μεταξύ επαναλήψεων) ονομάζεται loop-carried dependence
- In our prior example, each iteration was distinct. Dependences in the above example force successive iterations of this loop to execute in series.
- Implies that iterations can't be executed in parallel, right ?

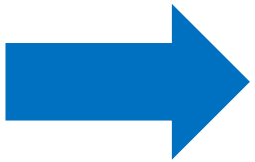
Loop-carried dependence: No parallelism?

- Παράδειγμα:

```
for (i=0; i<100; i=i+1) {
        A[i]   = A[i] + B[i];  /* S1 */
        B[i+1] = C[i] + D[i];  /* S2 */
}
```

- S1 χρησιμοποιεί τιμή του B[i] υπολογισμένη σε προηγούμενη επανάληψη (loop-carried depedence).
- Όμως δεν υπάρχει άλλη εξάρτηση. Άρα η παραπάνω εξάρτηση δεν είναι κυκλική (circular). Επομένως το loop είναι παράλληλο.

```
A[0] = A[0] + B[0]
B[1] = C[0] + D[0]
A[1] = A[1] + B[1]
B[2] = C[1] + D[1]
A[2] = A[2] + B[2]
B[3] = C[2] + D[2]
 . . .
```

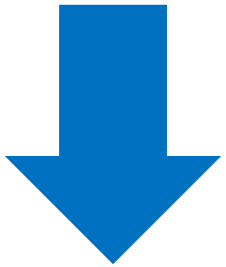


```
A[0] = A[0] + B[0]
B[1] = C[0] + D[0]
A[1] = A[1] + B[1]
B[2] = C[1] + D[1]
A[2] = A[2] + B[2]
B[3] = C[2] + D[2]
 . . .
```

Loop-carried dependence: No parallelism?

- Παράδειγμα:

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i];     /* S1 */
        B[i+1] = C[i] + D[i];   /* S2 */
}
```



```
A[0] = A[0] + B[0];                     /* start-up code */
for (i=0; i<99; i=i+1) {
        B[i+1] = C[i] + D[i];           /* S2 */
        A[i+1] = A[i+1] + B[i+1];       /* S1 */
}
B[100] = C[99] + D[99];                 /* clean-up code */
```

Recurrence – Depedence Distance

- Παράδειγμα:

```
for (i=1; i< 100; i=i+1) {
        Y[i] = Y[i-1] + Y[i];
}
```

  loop-carried εξάρτηση σε μορφή **recurrence.**

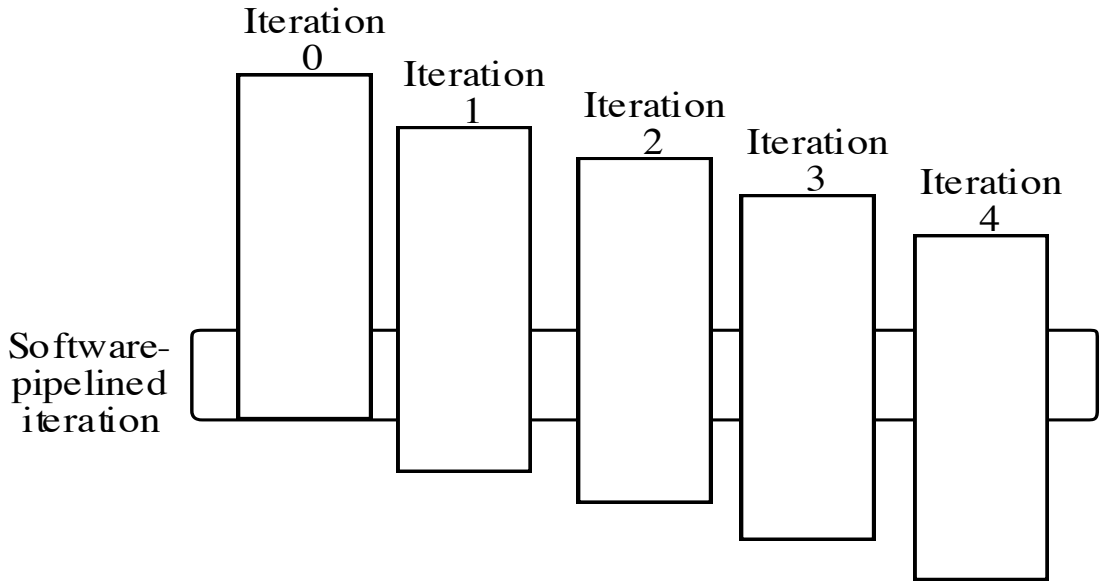
- Παράδειγμα:

```
for (i=5; i< 100; i=i+1) {
        Y[i] = Y[i-5] + Y[i];
}
```

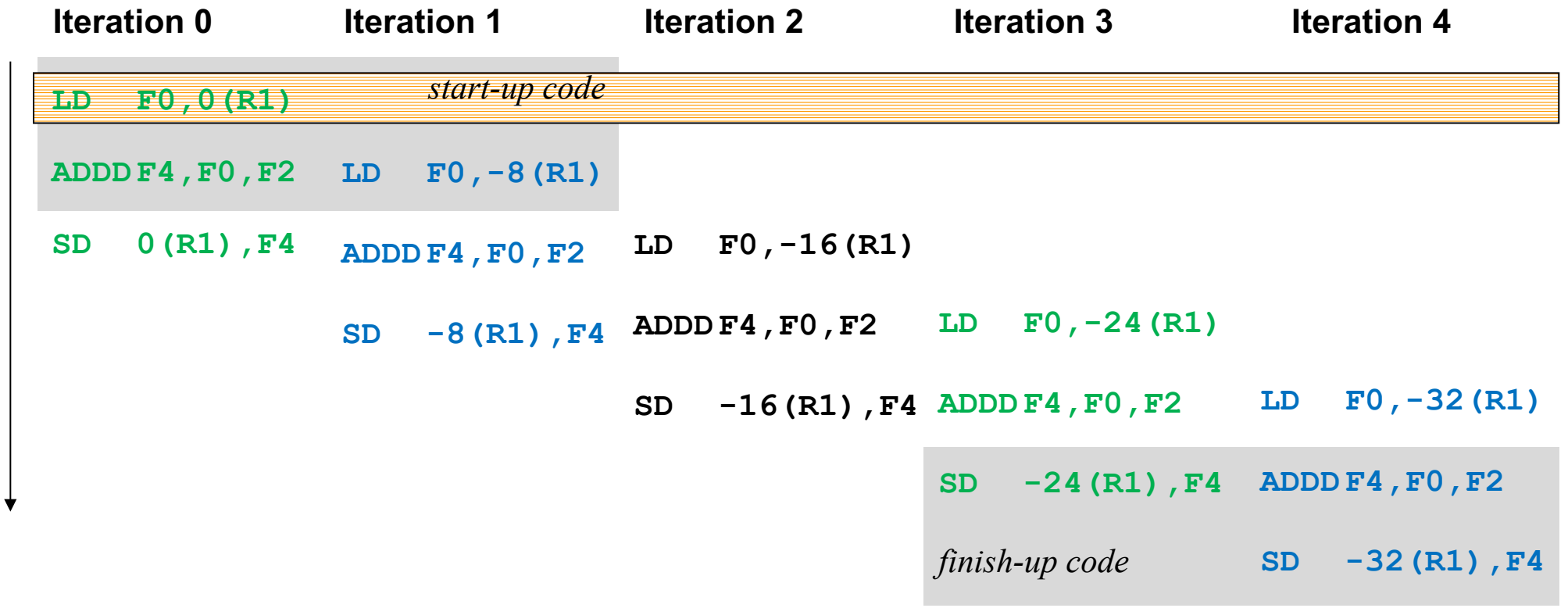
  Η επανάληψη `i` εξαρτάται από την `i-5`, δηλαδή έχει **dependence distance** 5. Όσο μεγαλύτερη η απόσταση τόσο περισσότερο πιθανό παραλληλισμό μπορούμε να πετύχουμε.

Άλλη εκδοχή: Software Pipelining

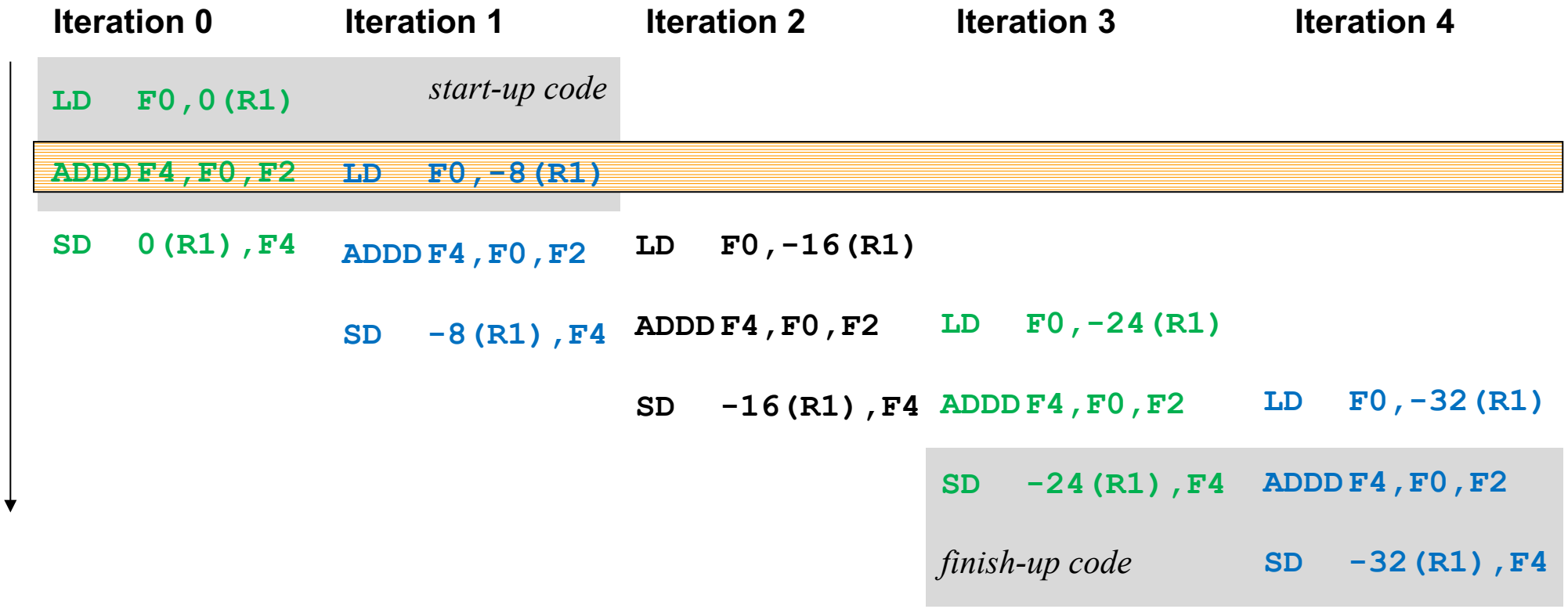
- Παρατήρηση: Αν οι επαναλήψεις του loop είναι ανεξάρτητες, τότε μπορούμε να έχουμε περισσότερο ILP εκτελώντας εντολές από διαφορετικές επαναλήψεις.
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop without loop unrolling (Tomasulo in SW)



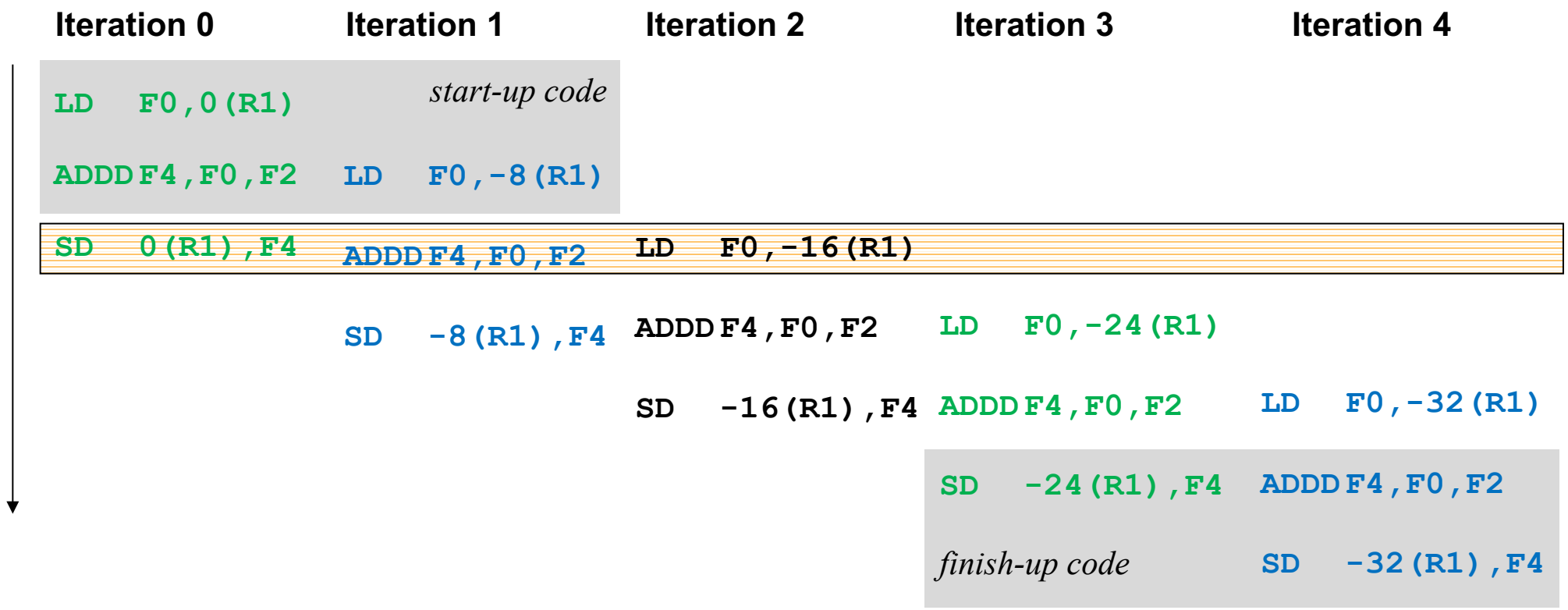
Software Pipelining: Παράδειγμα



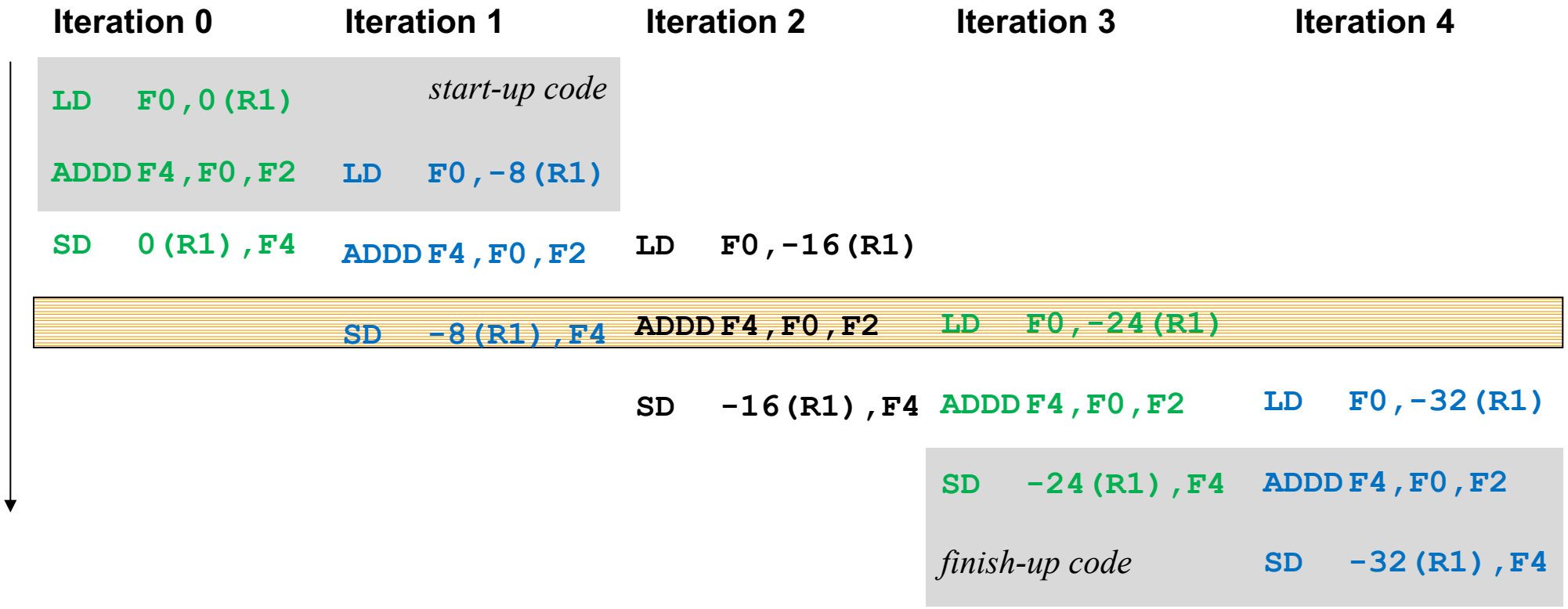
Software Pipelining: Παράδειγμα



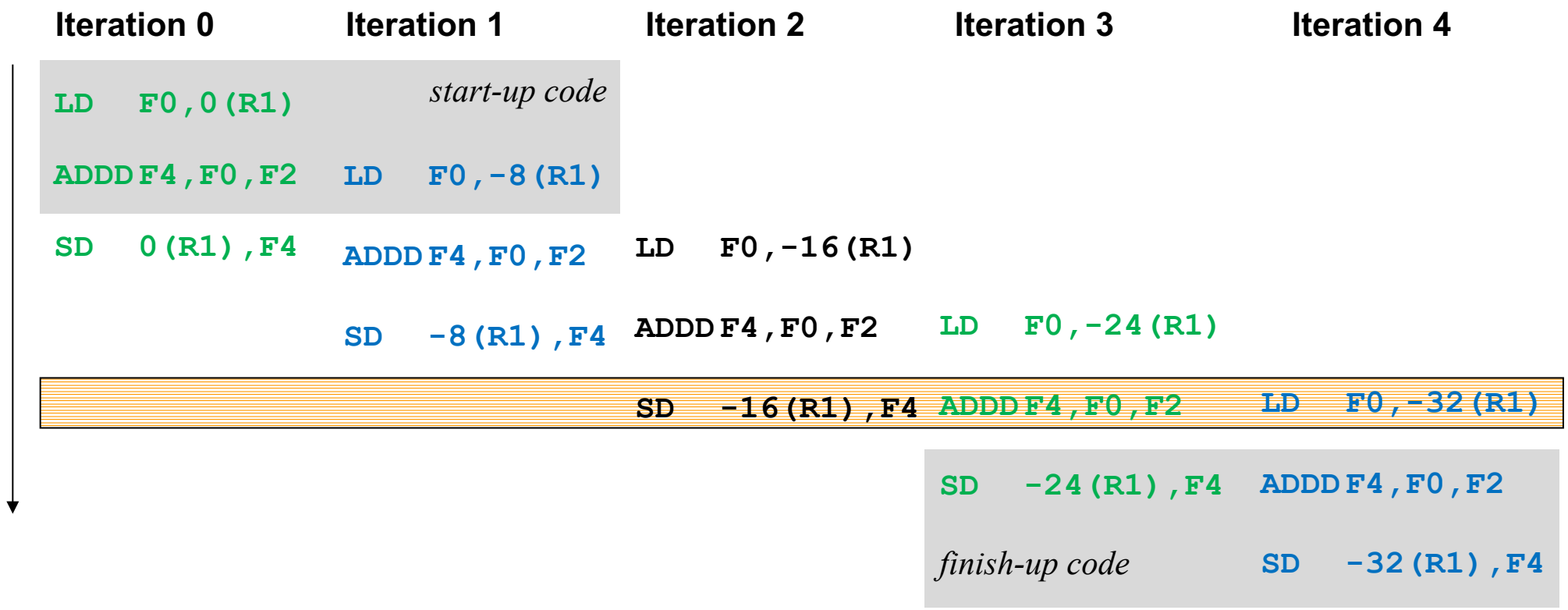
Software Pipelining: Παράδειγμα



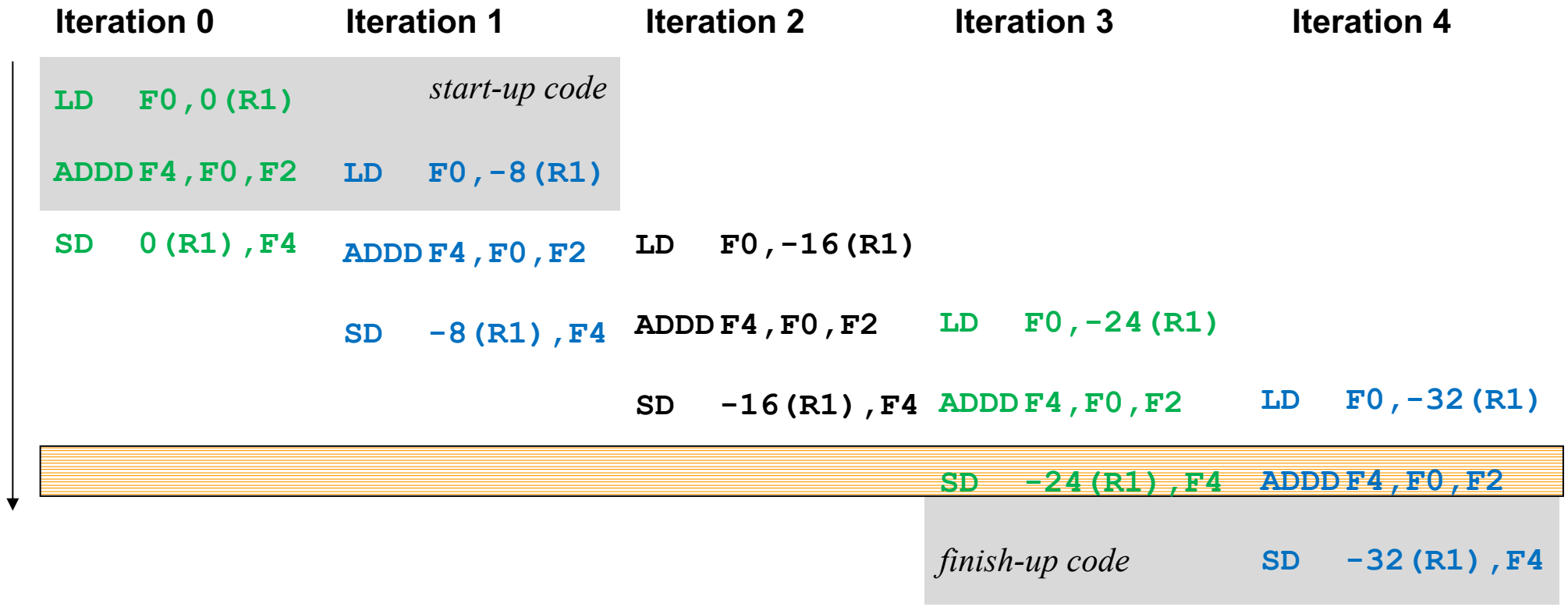
Software Pipelining: Παράδειγμα



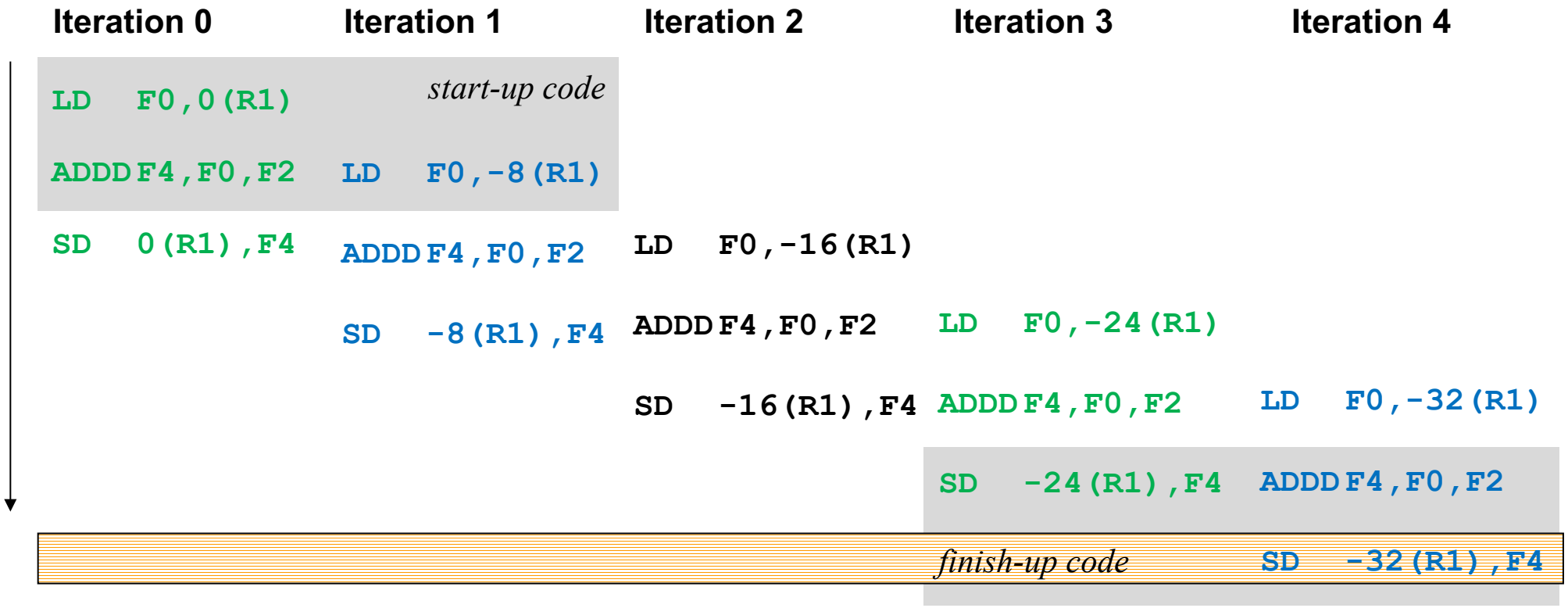
Software Pipelining: Παράδειγμα



Software Pipelining: Παράδειγμα



Software Pipelining: Παράδειγμα



Software Pipelining: Παράδειγμα

Πριν: Unrolled 3 times

```
1   L.D   F0,0(R1)
2   ADD.D F4,F0,F2
3   S.D   0(R1),F4
4   L.D   F6,-8(R1)
5   ADD.D F8,F6,F2
6   S.D   -8(R1),F8
7   L.D   F10,-16(R1)
8   ADD.D F12,F10,F2
9   S.D   -16(R1),F12
10  SUBI  R1,R1,#24
11  BNEZ  R1,LOOP
```

Μετά: Software Pipelined

```
1   S.D   0(R1),F4   ;Stores M[i]
2   ADD.D F4,F0,F2   ;Adds to M[i-1]
3   L.D   F0,-16(R1);Loads M[i-2]
4   SUBI  R1,R1,#8   ; i = i - 1
5   BNEZ  R1,LOOP
```

5 cycles per iteration

Τα RAW hazards μετατρέπονται σε WAR hazards.

Software Pipelining vs Loop Unrolling

Symbolic Loop Unrolling

- Maximize result-use distance
- Less code space than unrolling

But..

- Harder to implement
- Execution of SUB & BNEZ in every iteration