

HY425  
Αρχιτεκτονική Υπολογιστών

Static Scheduling

Ιάκωβος Μαυροειδής

# Τεχνικές ελάττωσης stalls.

$$\text{CPI} = \text{Ideal CPI} + \text{Structural stalls} + \text{RAW stalls} + \text{WAR stalls} + \text{WAW stalls} + \text{Control stalls}$$

Θα μελετήσουμε δύο ειδών τεχνικές

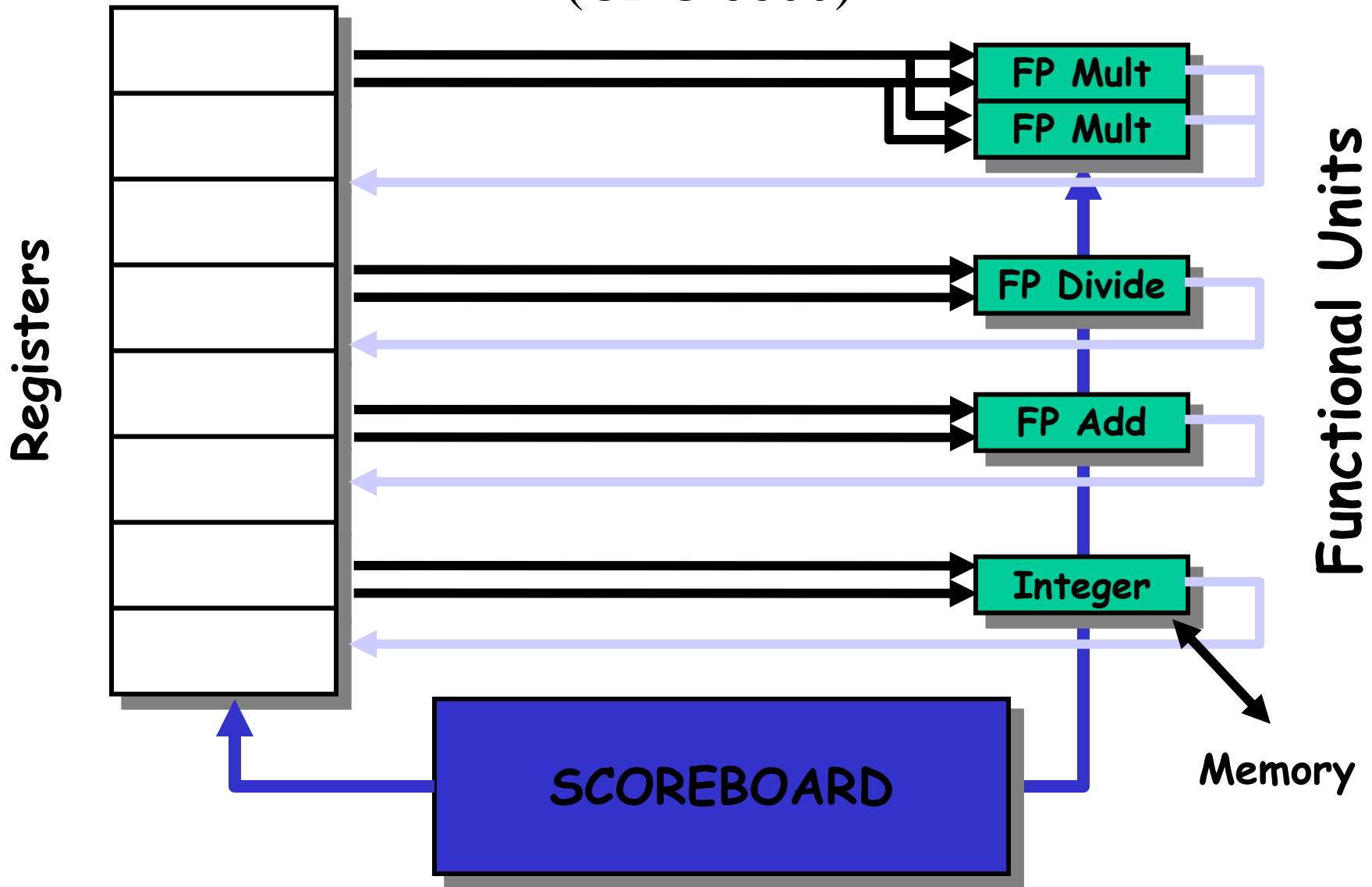
Δυναμικές δρομολόγηση  
εντολών (hardware)

- **Scoreboard** (ελάττωση RAW stalls)
- **Register Renaming**
  - α) **Tomasulo**
  - β) **Reorder Buffer**  
(ελάττωση WAR και WAW stalls)
- **Branch prediction**  
(ελάττωση Control stalls)

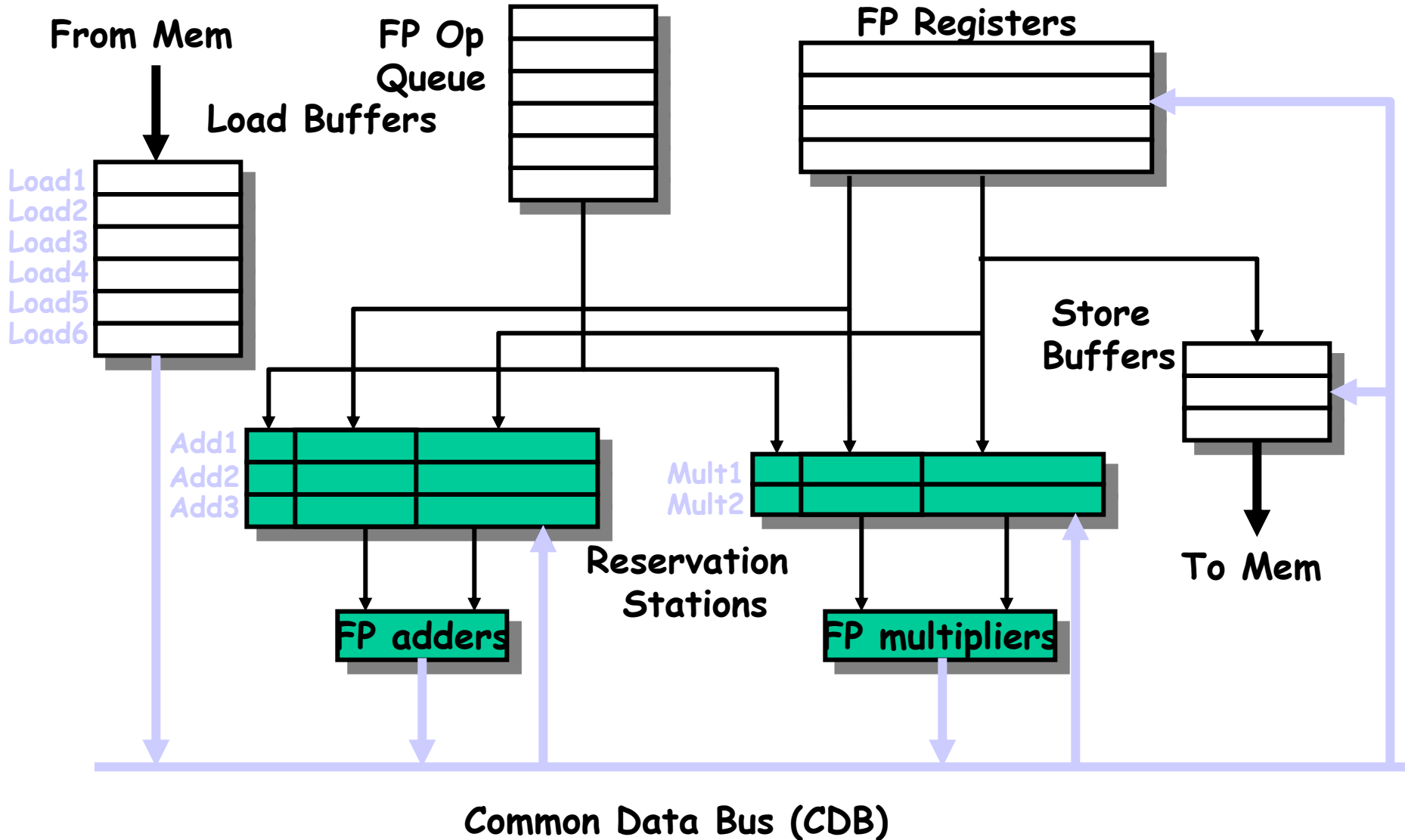
**Στατικές (software/compiler)**

- **Loop Unrolling**
- **Software Pipelining**
- **Trace Scheduling**

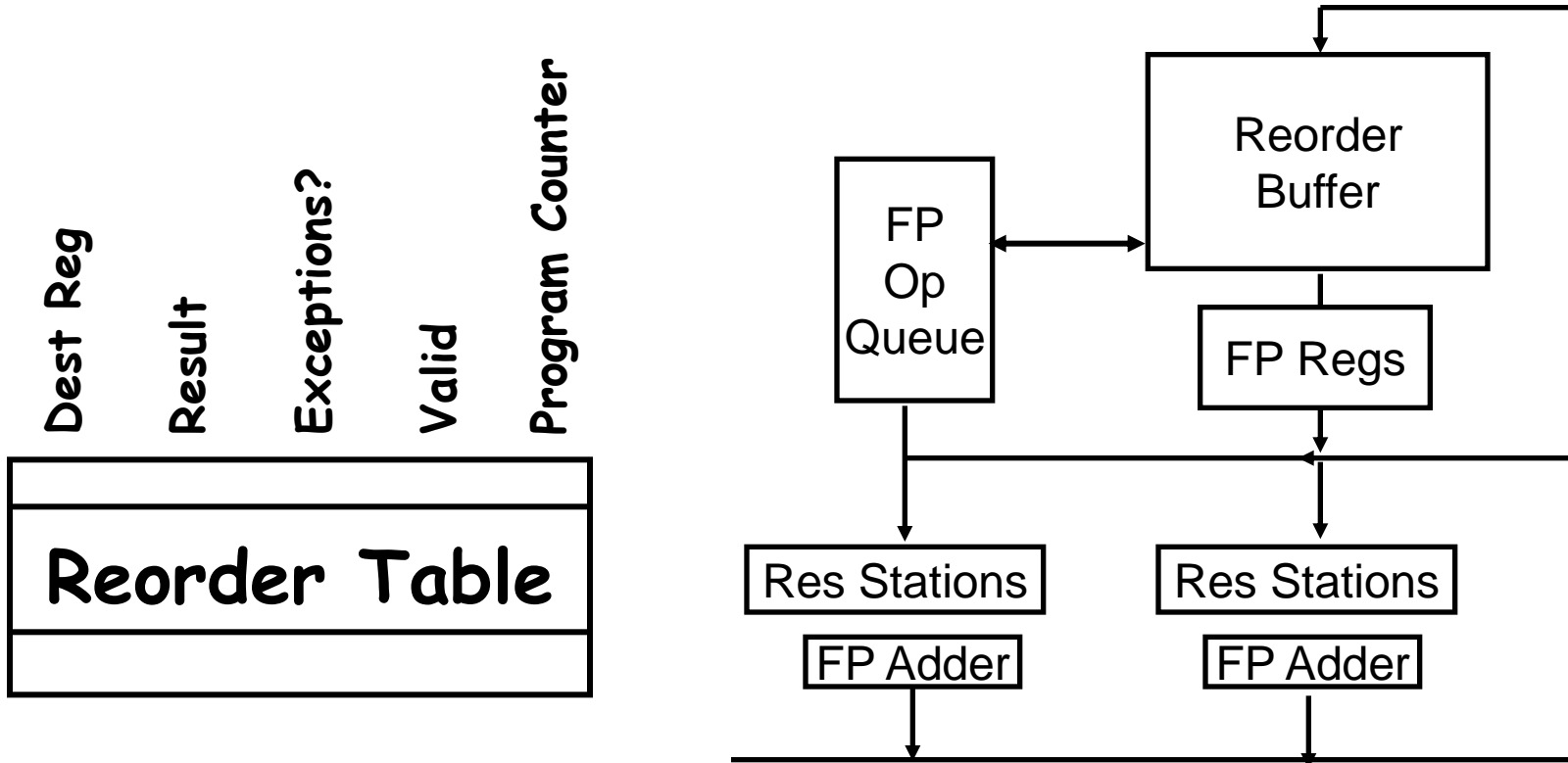
# Αρχιτεκτονική Scoreboard (CDC 6600)



# Tomasulo Organization

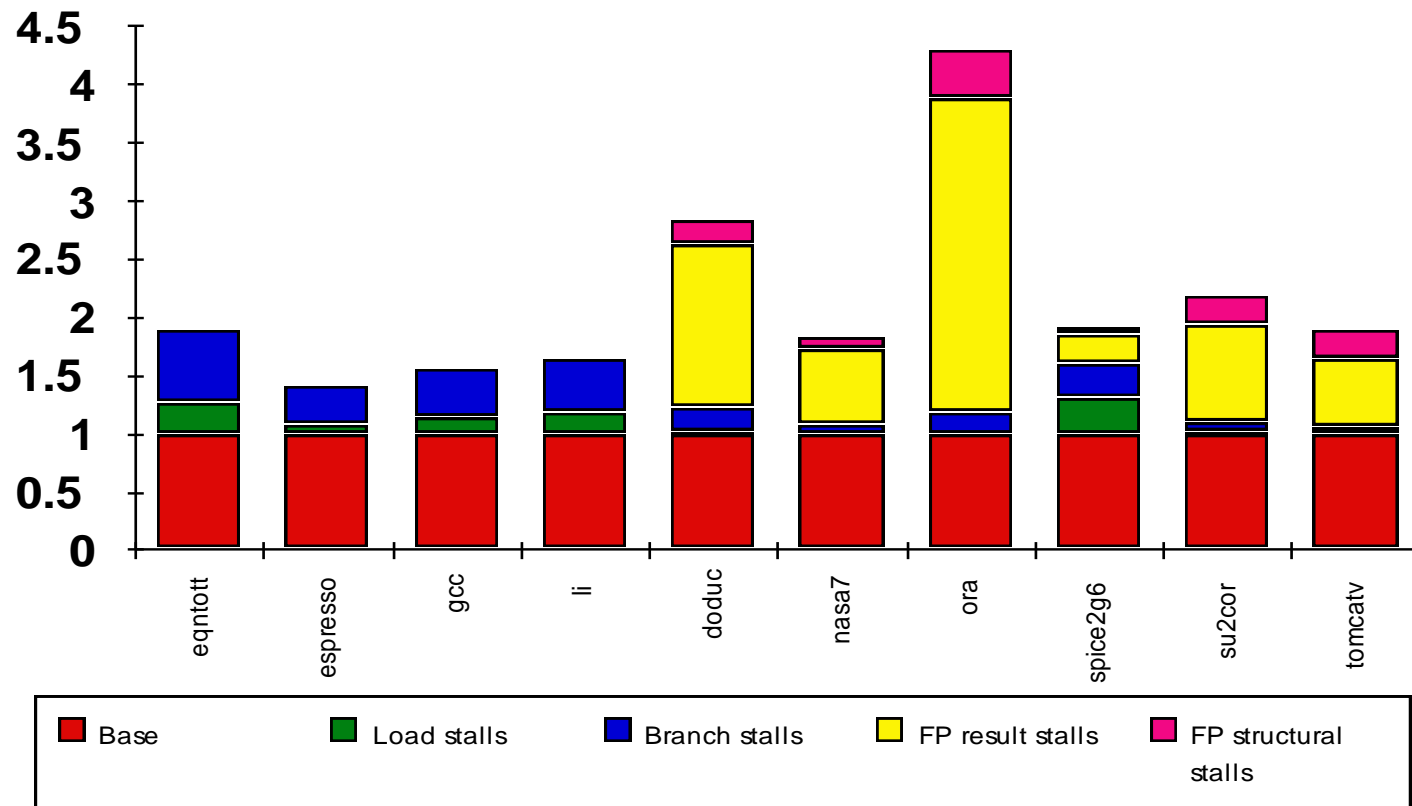


# Reorder buffer (ROB)



# 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 συσχετιζόμενων (μη εξαρτόμενων) εντολών.
- gcc 17% control transfer εντολές
  - 5 εντολές + 1 branch
  - Πέρα από ένα block για να έχουμε περισσότερη instruction level parallelism
- Loop level parallelism one opportunity
  - First SW, then HW approaches

# Εξαρτήσεις Μεταξύ Εντολών (Depedences)

- Ποιές είναι οι πηγές των stall/bubbles; Εντολές που χρησιμοποιούν ίδιους registers.

- *Παράλληλες* εντολές μπορούν να εκτελεστούν διαδοχικά χωρίς να δημιουργούν stalls. Π.χ.

```
DIVD F0, F2, F4  
ADDD F10, F1, F3
```

- *Εξαρτήσεις* μεταξύ εντολών μπορούν να οδηγήσουν σε stalls. Π.χ.

```
DIVD F0, F2, F4  
      RAW  
ADDD 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.

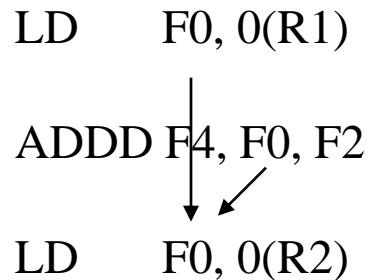


# Εξαρτήσεις Μεταξύ Εντολών

• **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 ή WAW hazard μεταξύ τους.

```
LD    F0, 0(R1)
      |
      v
ADDD  F4, F0, F2
      |
      v
LD    F0, 0(R2)
```



• **Control Dependences** : Εντολές εξαρτούμενες από branch εντολή.  
if p1 { S1; }

# FP Loop: Που είναι τα Hazards?

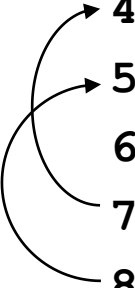
```
while (R1 > 0) { M[R1] = M[R1] + F2; R1 -= 8 }  
Loop:  L.D      F0,0(R1);F0=vector element  
      ADD.D    F4,F0,F2 ;add scalar from F2  
      S.D      0(R1),F4;store result  
      SUBI     R1,R1,8 ;decrement pointer 8B (DW)  
      BNEZ    R1,Loop ;branch R1!=zero  
      NOP               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

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
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

- Που έχουμε stalls?

# 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            ;delayed branch slot
```



<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- 9 κύκλοι: Ξαναγράψε κώδικα για να ελαχιστοποιήσεις 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
```

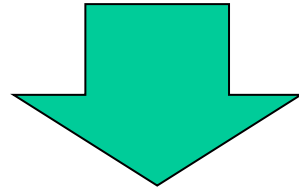
Αλλαγή θέσεων των BNEZ και SD αλλάζοντας την διεύθυνση του SD

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
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  
}
```

**Μη εξαρτόμενες εντολές  
μέσα στο Loop. Καλές  
προοπτικές για  
scheduling.**

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

# Unroll Loop 4 φορές: Που έχουμε 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 φορές: Που έχουμε 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
```

Πώς εξαφανίζονται;

# Που είναι τώρα;

```
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” εξαφάνισε WAR/WAW stalls**



# Unroll Loop 4 φορές

```
1 Loop: LD      F0, 0 (R1)
2      ADDD     F4, F0, F2
3      SD      0 (R1), F4
4      LD      F6, -8 (R1)
5      ADDD     F8, F6, F2
6      SD      -8 (R1), F8
7      LD      F10, -16 (R1)
8      ADDD     F12, F10, F2
9      SD      -16 (R1), F12
10     LD      F14, -24 (R1)
11     ADDD     F16, F14, F2
12     SD      -24 (R1), F16
13     SUBI     R1, R1, #32
14     BNEZ    R1, LOOP
15     NOP
```

1 cycle stall

2 cycles stall

**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: LD      F0, 0(R1)
2      LD      F6, -8(R1)
3      LD      F10, -16(R1)
4      LD      F14, -24(R1)
5      ADDD   F4, F0, F2
6      ADDD   F8, F6, F2
7      ADDD   F12, F10, F2
8      ADDD   F16, F14, F2
9      SD     0(R1), F4
10     SD     -8(R1), F8
11     SD     -16(R1), F12
12     SUBI   R1, R1, #32
13     BNEZ   R1, LOOP
14     SD     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:

$$0 (R1) \neq -8 (R1) \neq -16 (R1) \neq -24 (R1)$$

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

# Πότε είναι ασφαλές να κάνουμε Unroll ένα Loop?

- Παράδειγμα: Που είναι οι εξαρτήσεις?

(A,B,C distinct & nonoverlapping)

```
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 */  
}
```

1. S2 uses the value, A[i+1], computed by S1 in the same iteration.
2. 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**

- For our prior example, each iteration was distinct
- Implies that iterations can't be executed in parallel, Right ??

# loop-carried dependence σημαίνει ότι δεν υπάρχει παραλληλισμός;

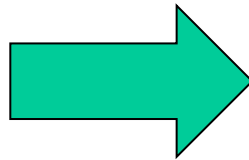
- Παράδειγμα:

```
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 dependence).

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]  
...

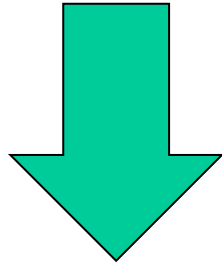


A[0] = A[0] + B[0]  
B[1] = C[0] + D[0]  
A[1] = A[1] + B[1]  
B[2] = C[1] + D[1]  
A[2] = A[2] + B[2]  
B[3] = C[2] + D[2]  
...

# loop-carried dependence σημαίνει ότι δεν υπάρχει παραλληλισμός;

- 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];    /* S1 */  
    A[i+1] = A[i+1] + B[i+1]; /* S2 */  
}  
B[100] = B[99] + D[99]; /* clean-up code*/
```

# Recurrence – Dependence Distance

- Παράδειγμα:

```
for (i=2; i < 100; i=i+1) {  
    Y[i] = Y[i-1] + Y[i];  
}
```

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

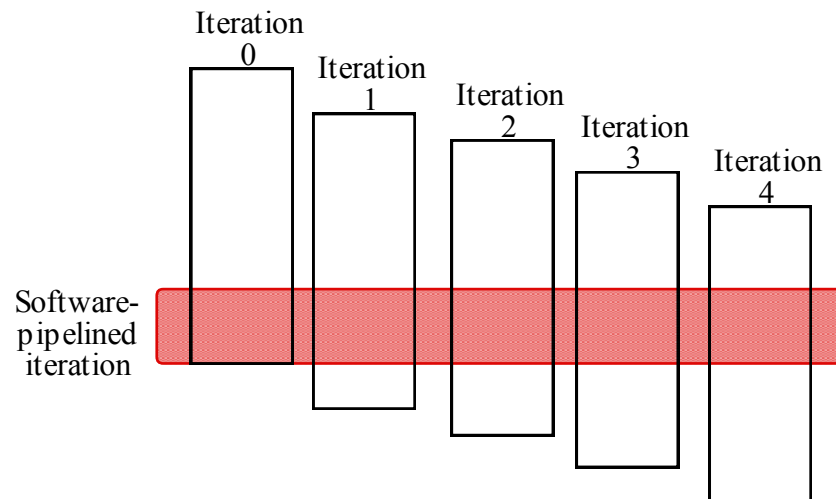
- Παράδειγμα:

```
for (i=2; i < 100; i=i+1) {  
    Y[i] = Y[i-5] + Y[i];  
}
```

Η επανάληψη  $i$  εξαρτιέται 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 (- Tomasulo in SW)





# Software Pipelining: Παράδειγμα

Iteration 0

Iteration 1

Iteration 2

Iteration 3

Iteration 4

LD F0, 0(R1) *start-up code*

ADD F4, F0, F2 LD F0, -8(R1)

SD 0(R1), F4 ADD F4, F0, F2 LD F0, -16(R1)

SD -8(R1), F4 ADD F4, F0, F2 LD F0, -24(R1)

SD -16(R1), F4 ADD F4, F0, F2 LD F0, -32(R1)

SD -24(R1), F4 ADD F4, F0, F2

*finish-up code*

SD -32(R1), F4



# 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
5  BNEZ R1,LOOP
```

*5 cycles per iteration*

**Τα RAW hazards μετατρέπονται σε WAR hazards.**

# Software Pipelining

## Symbolic Loop Unrolling

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

**But..**

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