

# Εισαγωγή

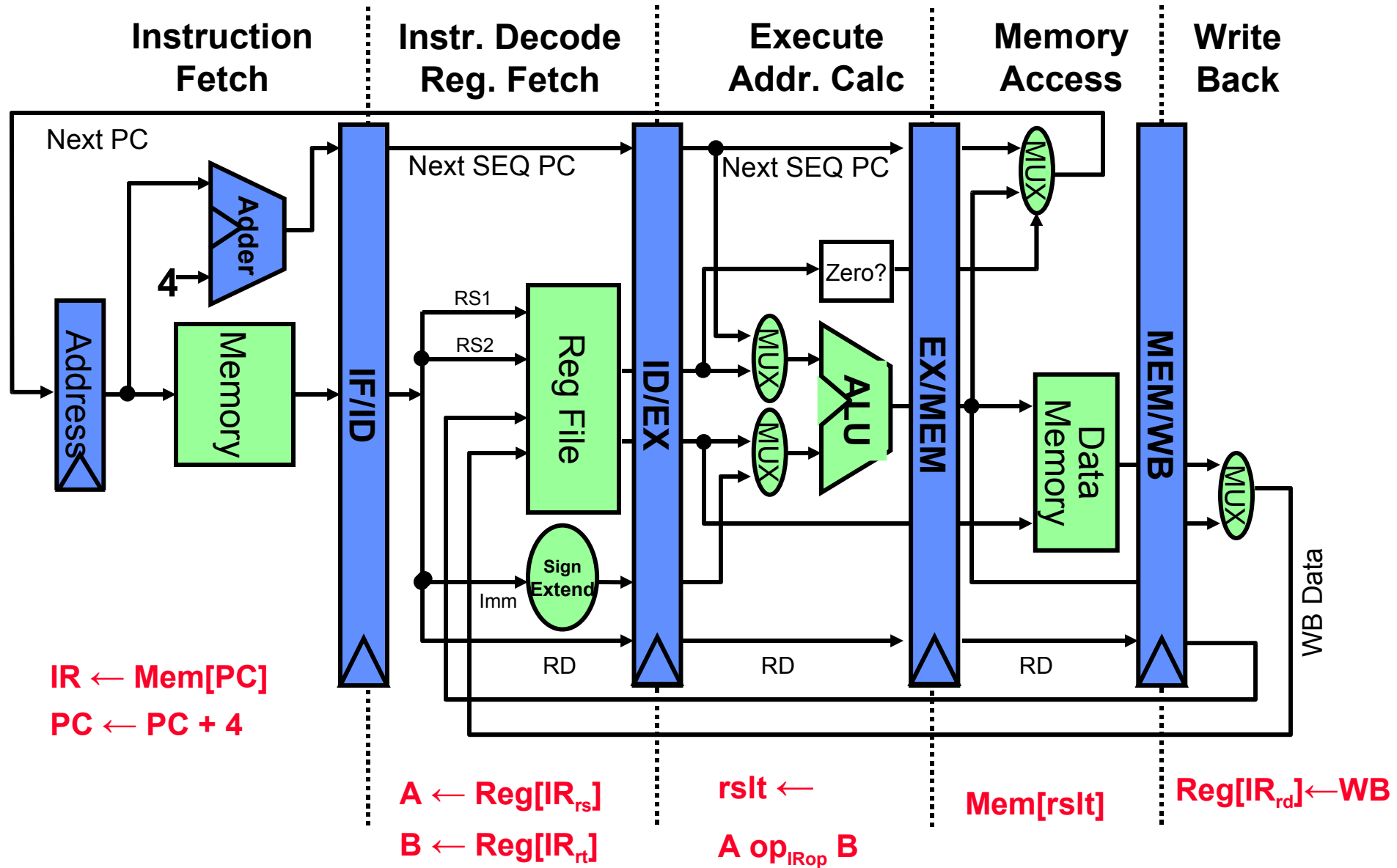
Σύνοψη βασικών εννοιών, 5-stage pipeline,  
επεκτάσεις για λειτουργίες πολλαπλών κύκλων

# Παράγοντες που επηρεάζουν την επίδοση της CPU

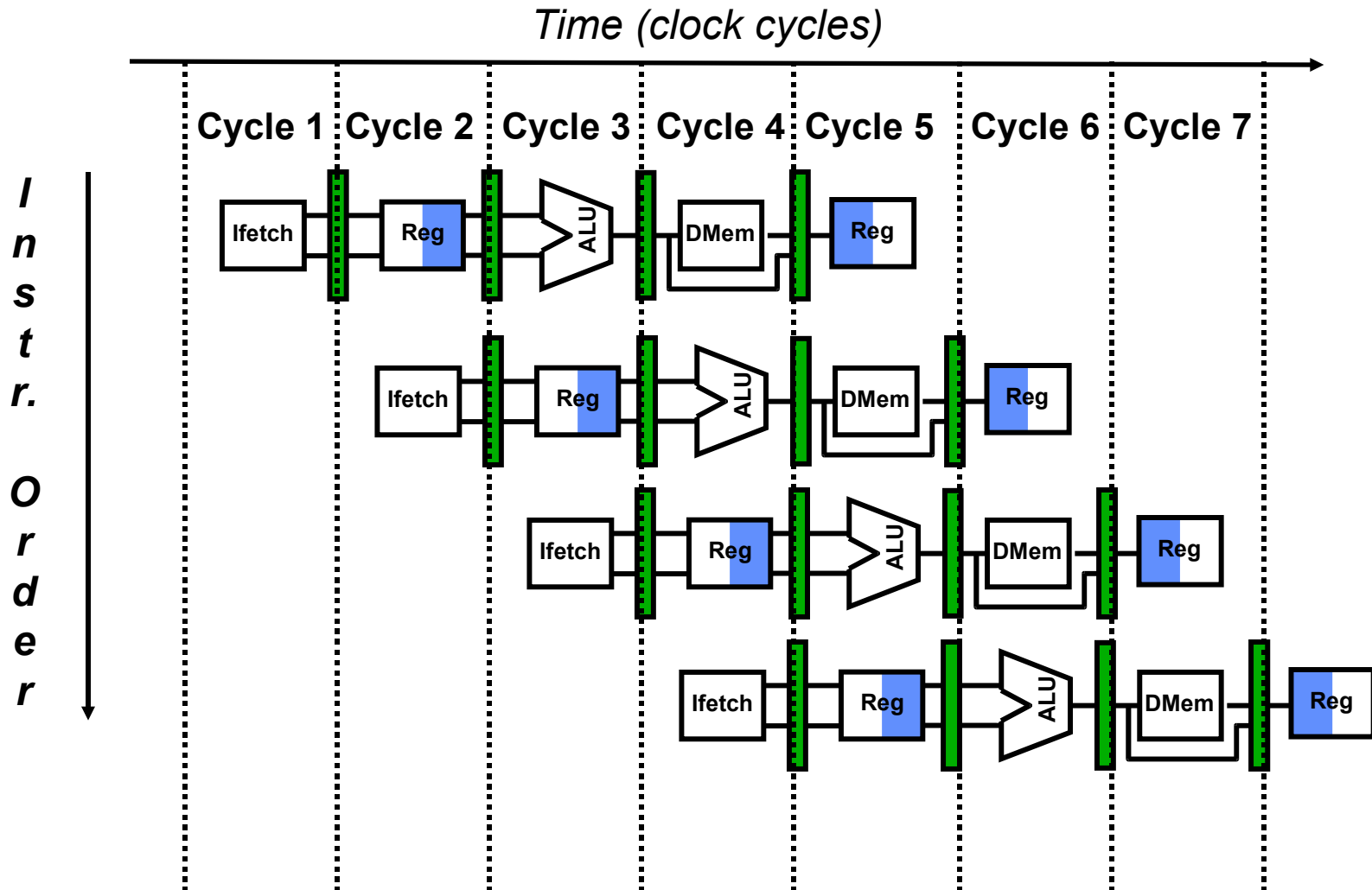
$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Instr. count	CPI	Clock rate
Program	X		
Compiler	X	X	
Instruction Set Architecture (ISA)	X	X	X
Οργάνωση		X	X
Τεχνολογία			X

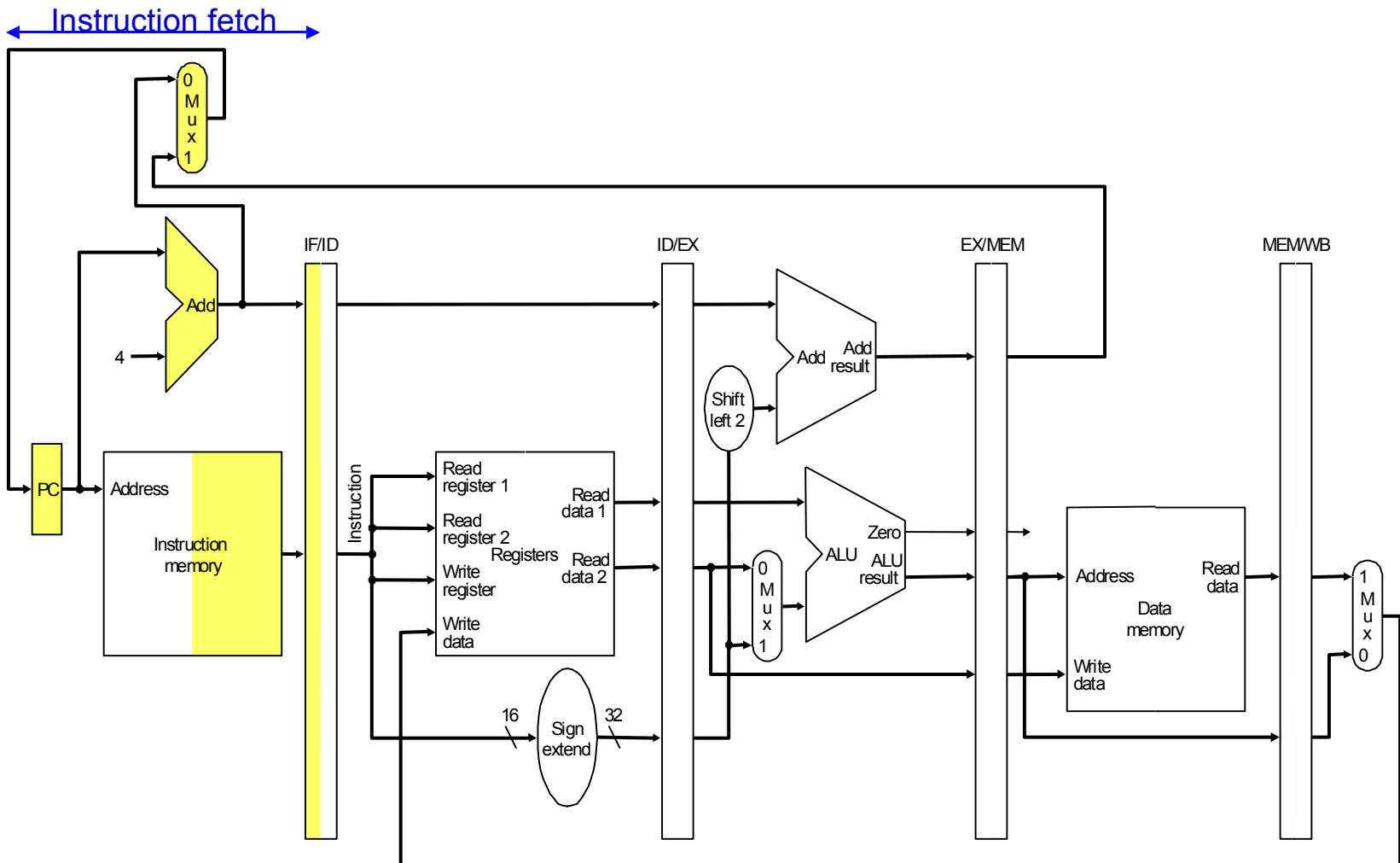
# 5-Stage Pipelined Datapath



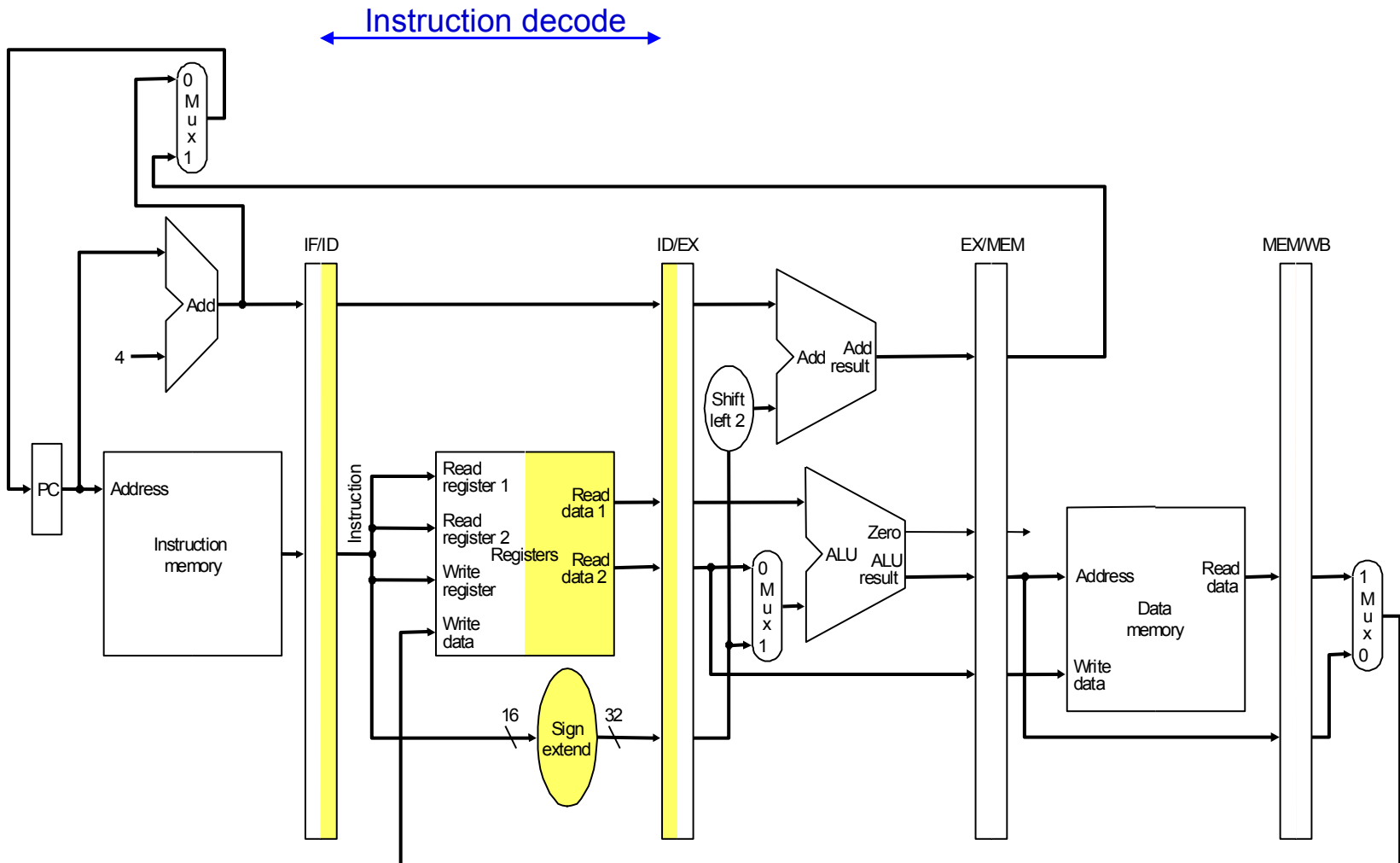
# Διάγραμμα χρονισμού



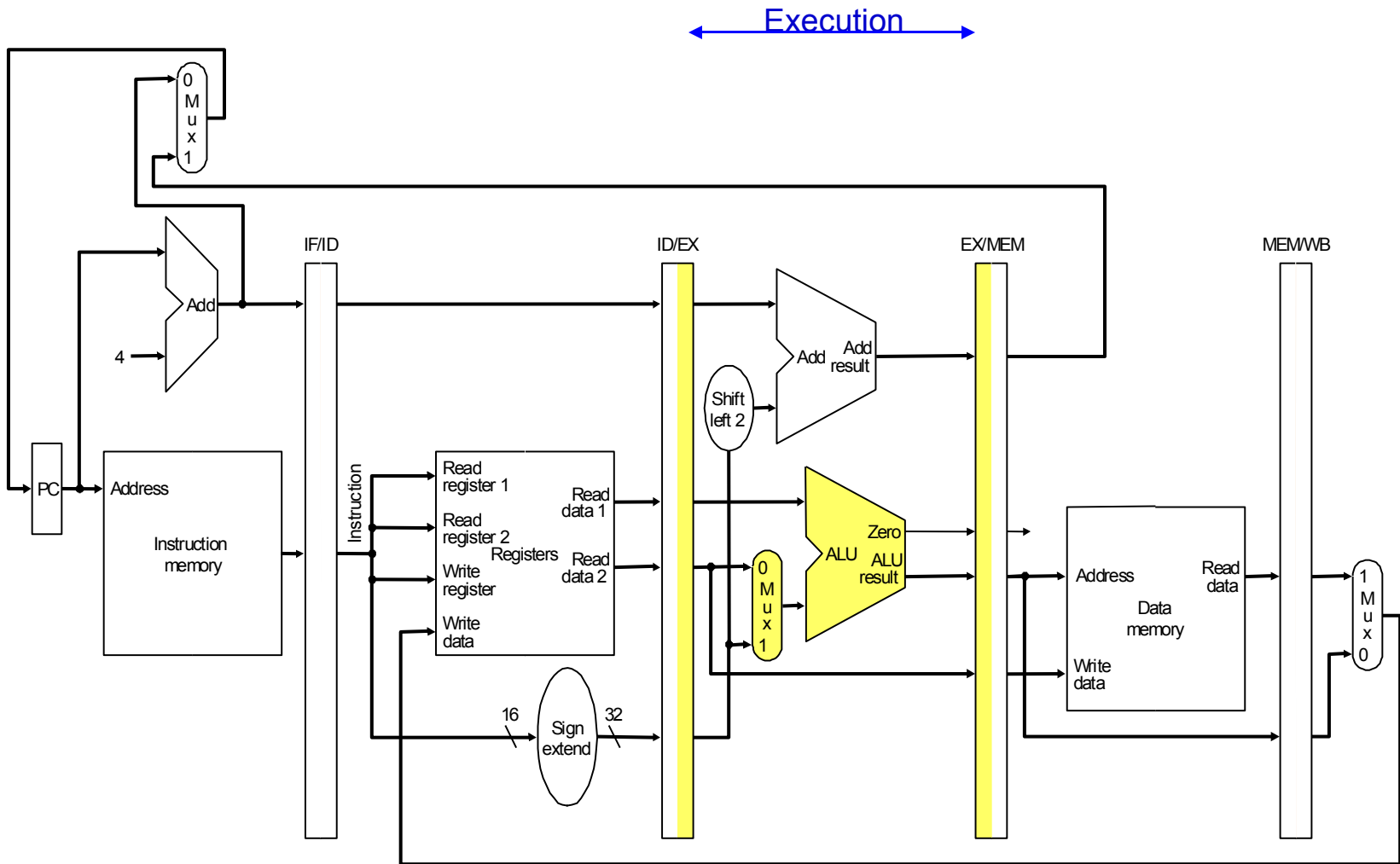
# Παράδειγμα για την εντολή lw: Instruction Fetch (IF)



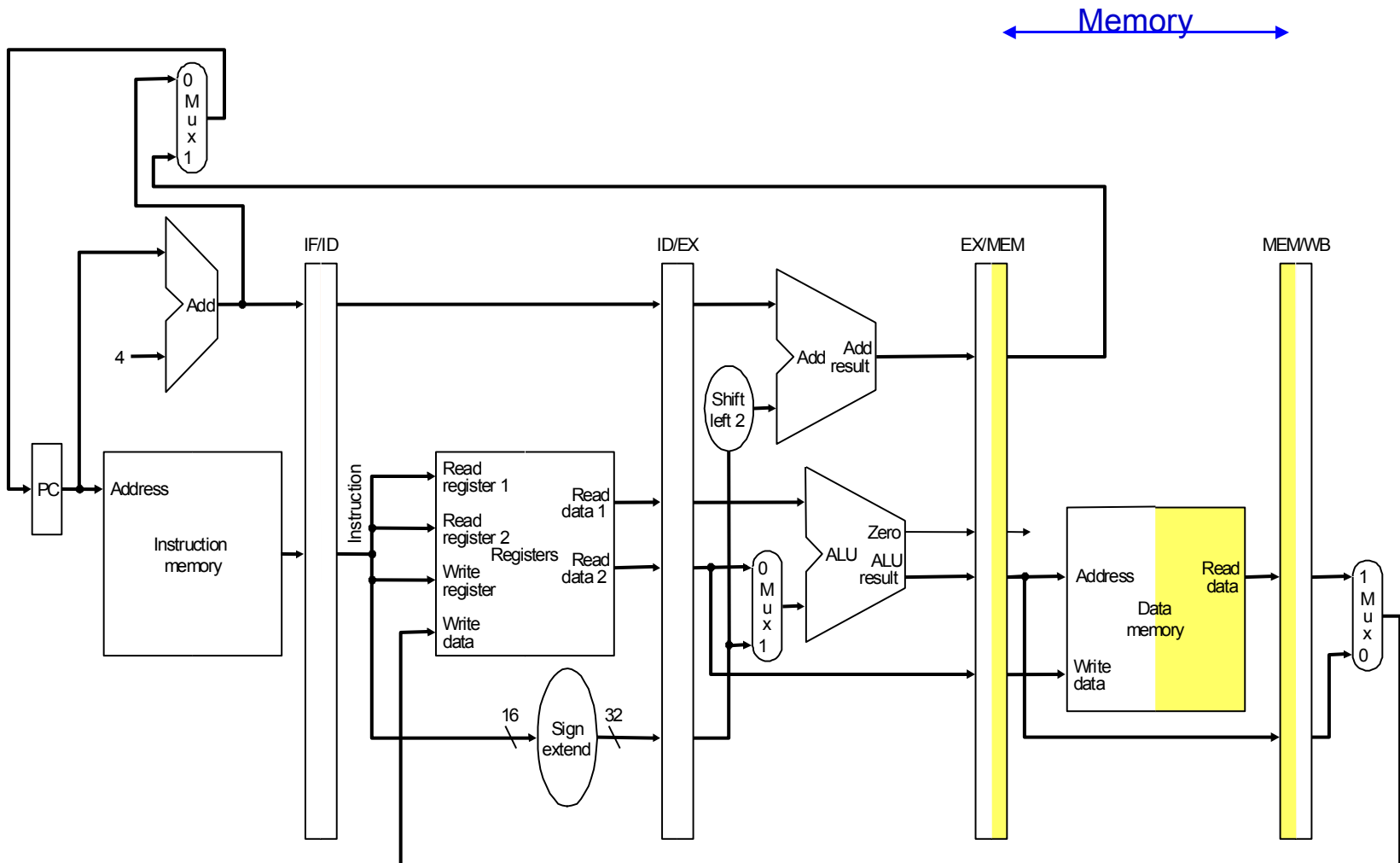
# Παράδειγμα για την εντολή lw: Instruction Decode (ID)



# Παράδειγμα για την εντολή lw: Execution (EX)



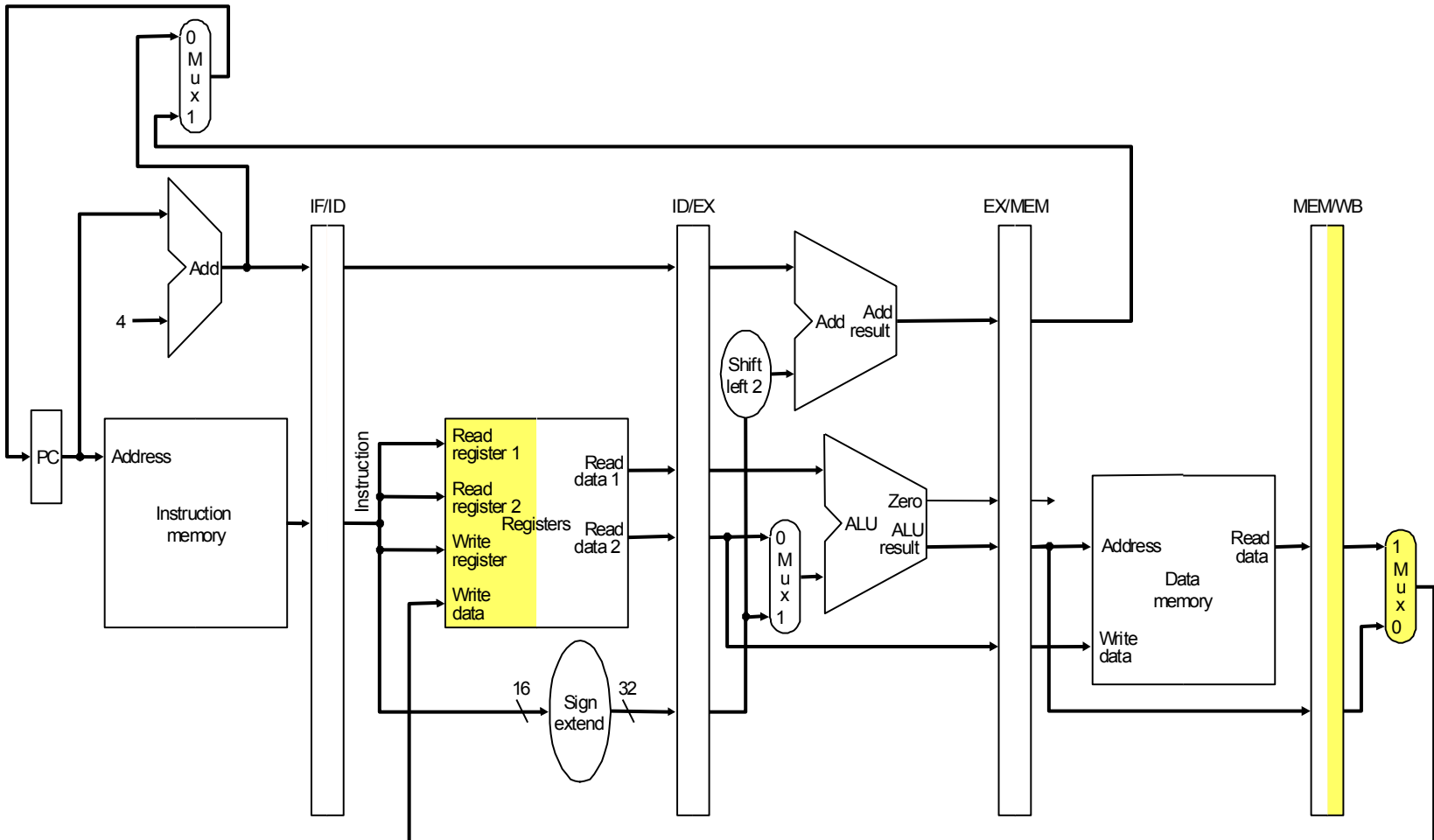
# Παράδειγμα για την εντολή lw: Memory (MEM)





# Παράδειγμα για την εντολή lw: Writeback (WB)

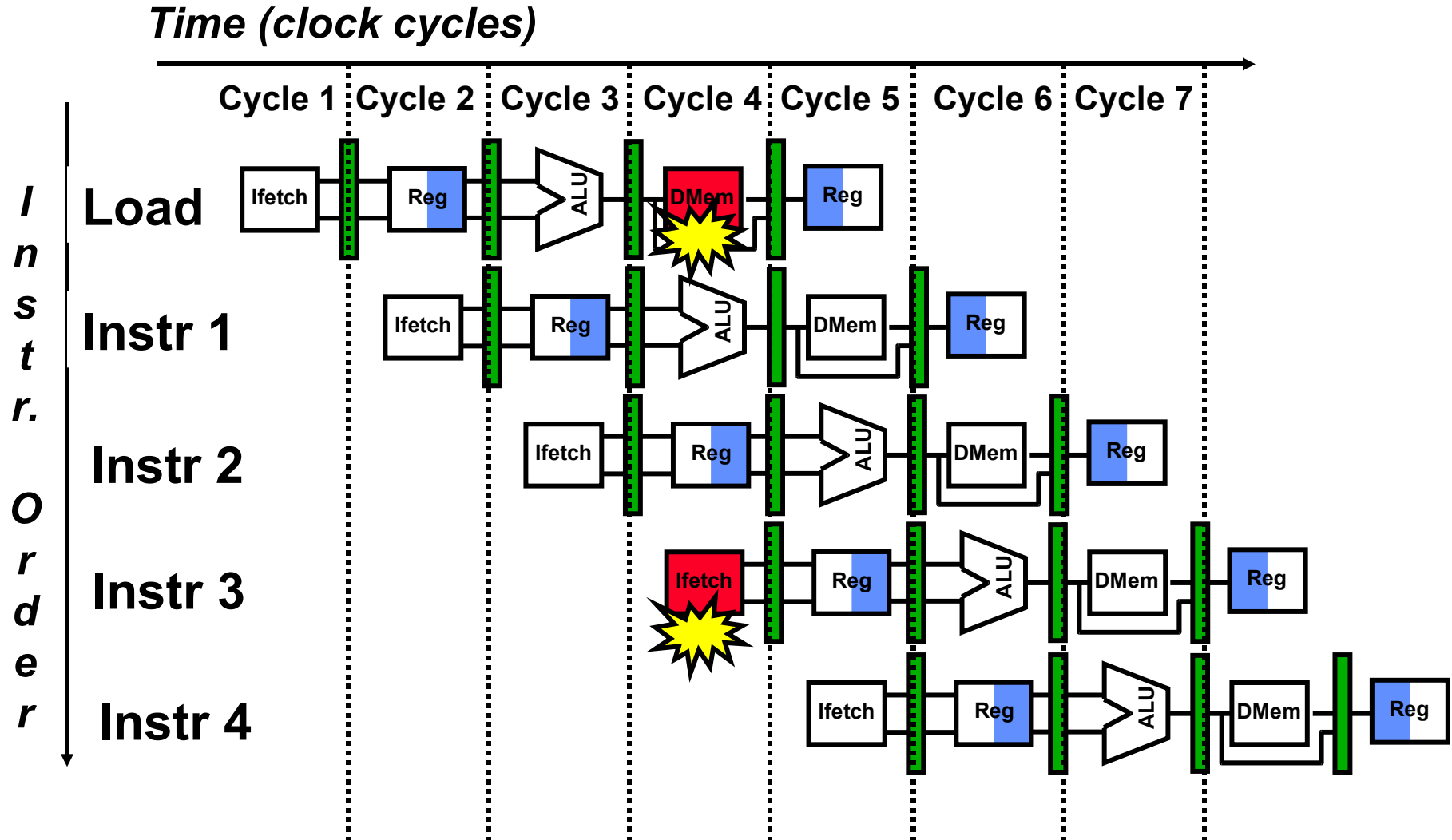
← Writeback →



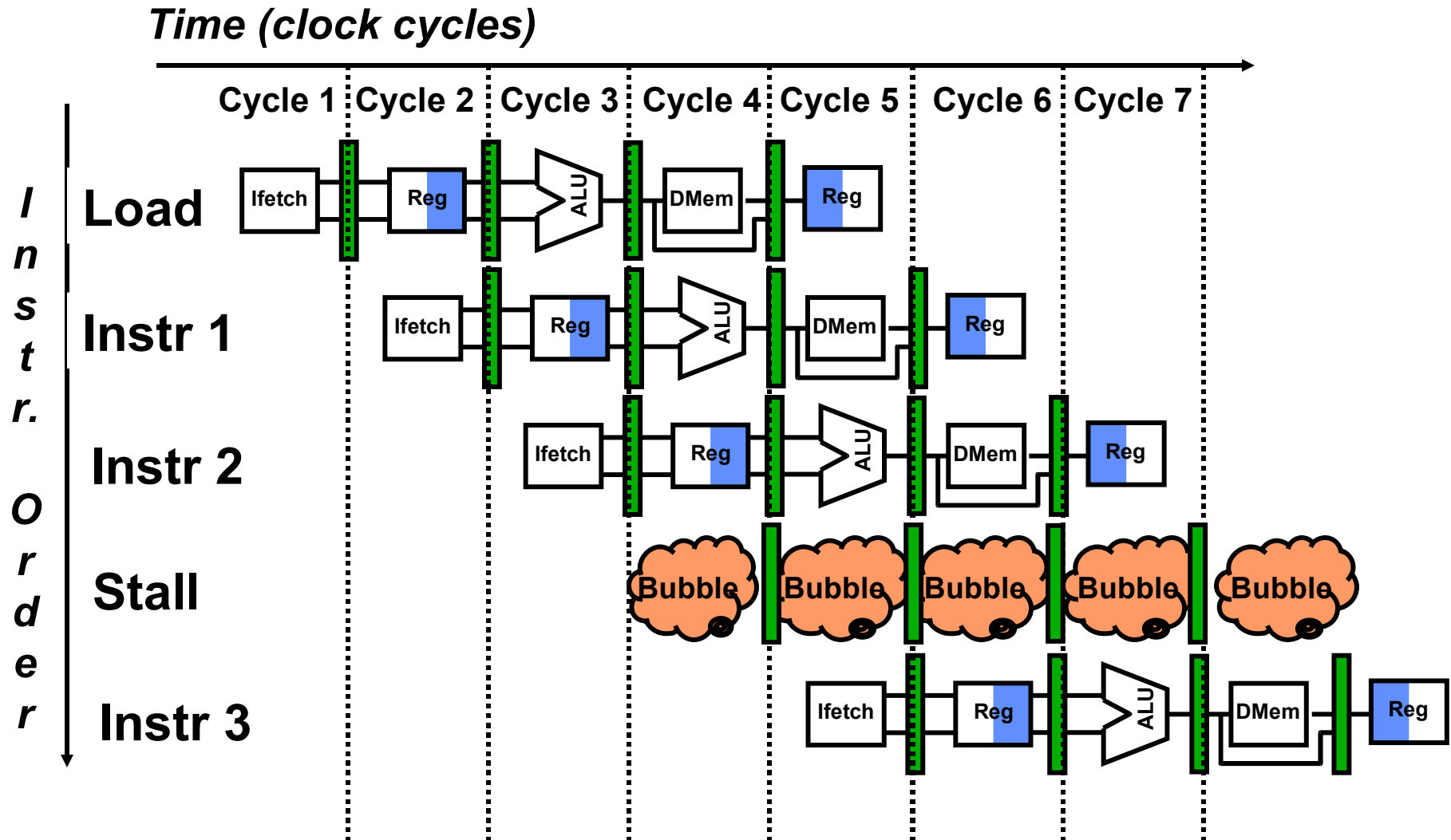
# Περιορισμοί pipeline

- Οι **κίνδυνοι (hazards)** αποτρέπουν την επόμενη εντολή από το να εκτελεστεί στον κύκλο που πρέπει
  - Δομικοί κίνδυνοι (structural): όταν το υλικό δεν μπορεί να υποστηρίξει ταυτόχρονη εκτέλεση συγκεκριμένων εντολών
  - Κίνδυνοι δεδομένων (data): όταν μια εντολή χρειάζεται το αποτέλεσμα μιας προηγούμενης, η οποία βρίσκεται ακόμη στο pipeline
  - Κίνδυνοι ελέγχου (control): όταν εισάγεται καθυστέρηση μεταξύ του φορτώματος εντολών και της λήψης αποφάσεων σχετικά με την αλλαγή της ροής του προγράμματος (branches, jumps)

# Παράδειγμα structural hazard: ένα διαθέσιμο memory port



# Παράδειγμα structural hazard: επίλυση με stall



# Παράδειγμα data hazard στον r1

Time (clock cycles)



*I*  
*n*  
*s*  
*t*  
*r.*  
  
*O*  
*r*  
*d*  
*e*  
*r*

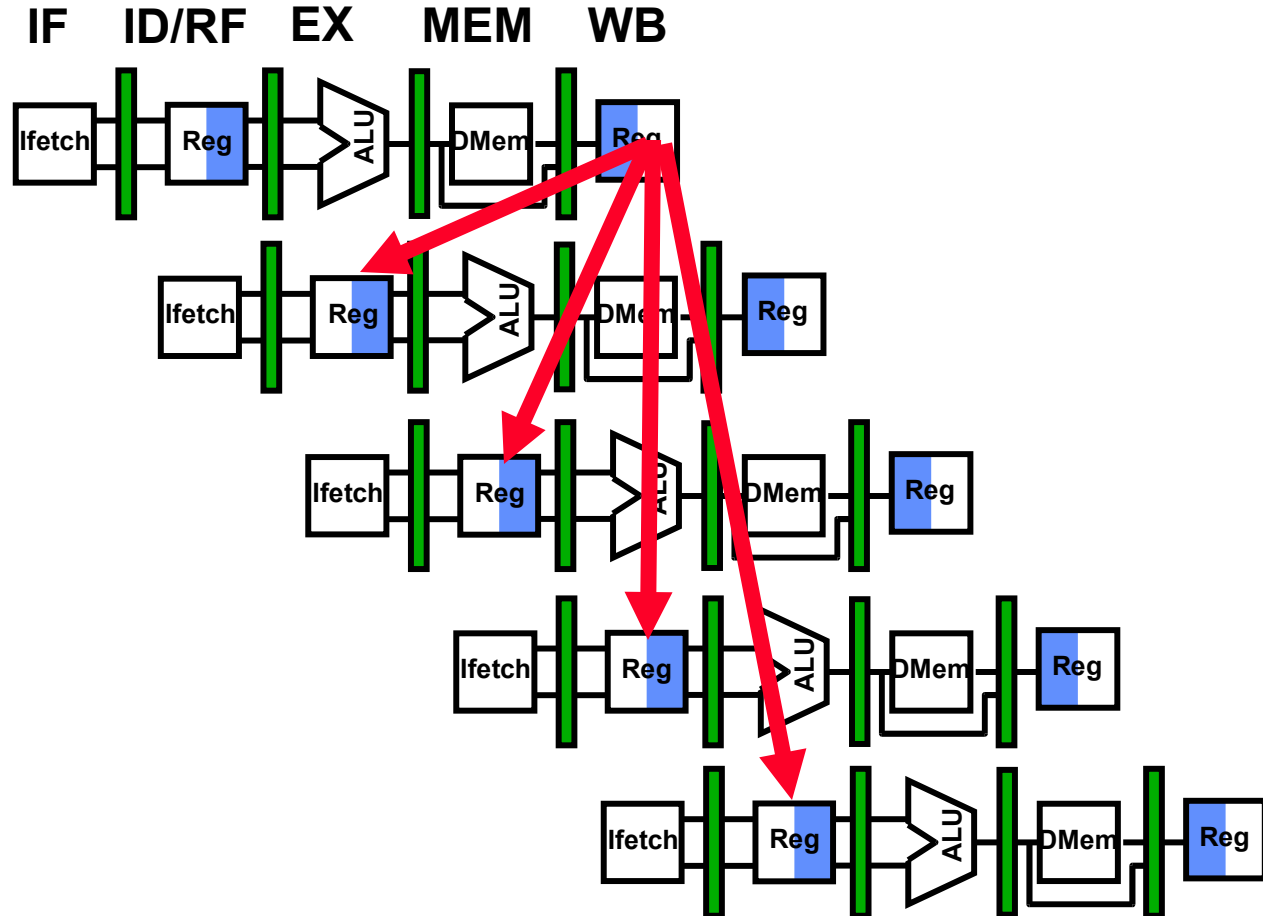
add r1,r2,r3

sub r4,r1,r3

and r6,r1,r7

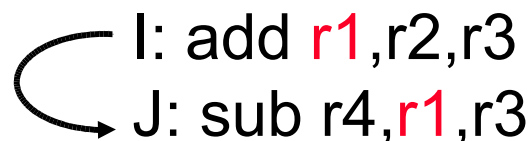
or r8,r1,r9

xor r10,r1,r11



## Εξαρτήσεις και κίνδυνοι δεδομένων

- Η J είναι **data dependent** από την I:  
Η J προσπαθεί να διαβάσει τον source operand πριν τον γράψει η I

  
I: add r1,r2,r3  
J: sub r4,r1,r3

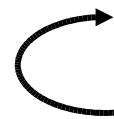
- ή, η J είναι data dependent από την K, η οποία είναι data dependent από την I (αλυσίδα εξαρτήσεων)
- **Πραγματικές εξαρτήσεις (True Dependences)**
- Προκαλούν **Read After Write (RAW) hazards** στο pipeline

# Εξαρτήσεις και κίνδυνοι δεδομένων

- Οι εξαρτήσεις είναι ιδιότητα των **προγραμμάτων**
- Η παρουσία μιας εξάρτησης υποδηλώνει την **πιθανότητα** εμφάνισης hazard, αλλά το αν θα συμβεί πραγματικά το hazard, και το πόση καθυστέρηση θα εισάγει, είναι ιδιότητα του **pipeline**
- Η σημασία των εξαρτήσεων δεδομένων:
  - 1) υποδηλώνουν την πιθανότητα για hazards
  - 2) καθορίζουν τη σειρά σύμφωνα με την οποία πρέπει να υπολογιστούν τα δεδομένα
  - 3) θέτουν ένα άνω όριο στο ποσό του παραλληλισμού που μπορούμε να εκμεταλλευτούμε

# Name Dependences (1): Anti-dependences

- **Name dependences:** όταν 2 εντολές χρησιμοποιούν τον ίδιο καταχωρητή ή θέση μνήμης ("name"), χωρίς όμως να υπάρχει πραγματική ροή δεδομένων μεταξύ τους
- **Anti-dependence:** η J γράφει τον r1 πριν τον διαβάσει η I

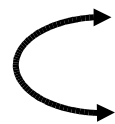
 I: sub r4, r1, r3  
J: add r1, r2, r3  
K: mul r6, r1, r7

- Προκαλούν **Write After Read (WAR) data hazards** στο pipeline
- Δε μπορούν να συμβούν στο κλασικό 5-stage pipeline διότι:
  - όλες οι εντολές χρειάζονται 5 κύκλους για να εκτελεστούν, και
  - οι αναγνώσεις συμβαίνουν πάντα στο στάδιο 2, και
  - οι εγγραφές στο στάδιο 5



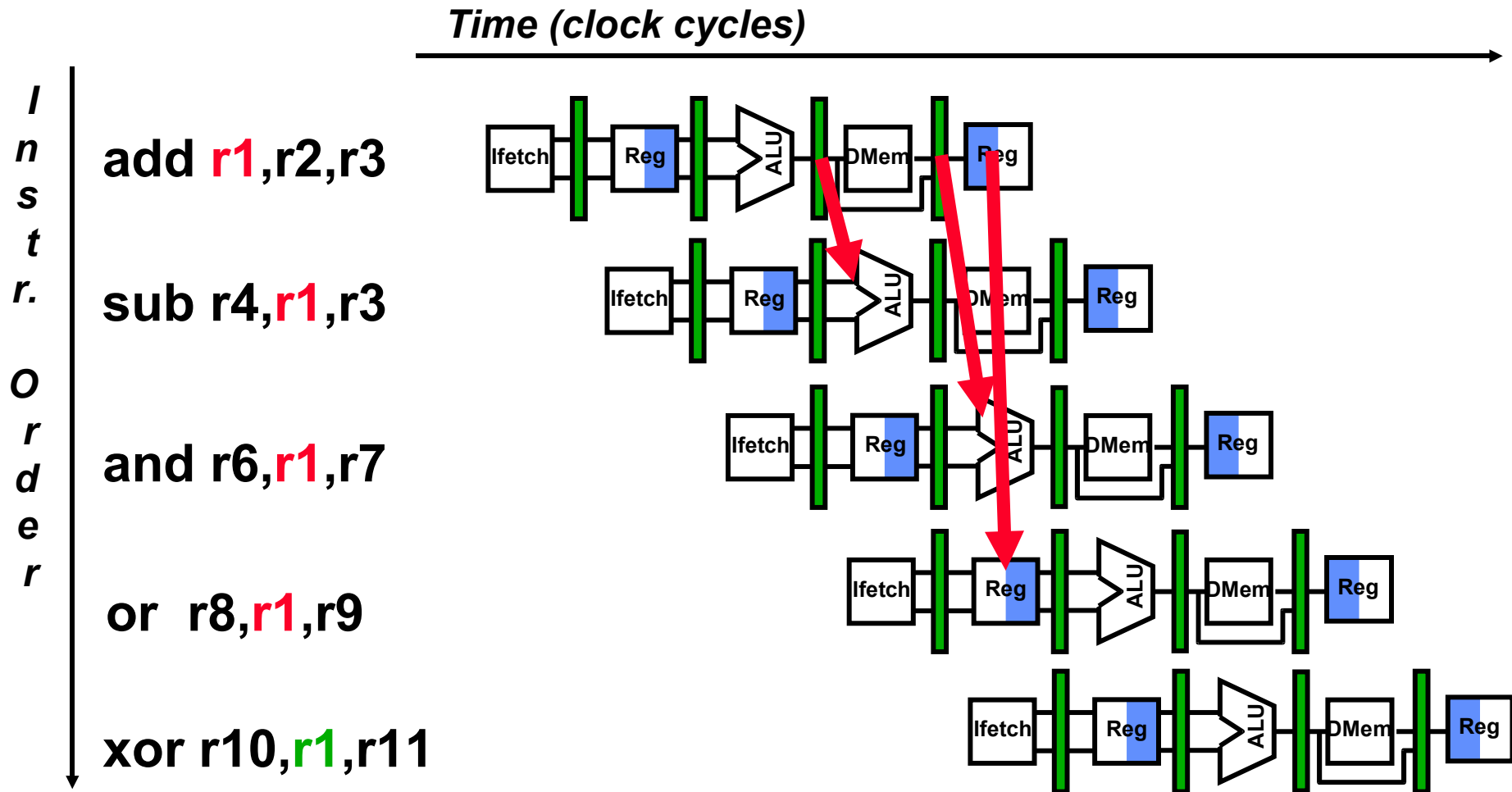
## Name Dependences (2): Output dependences

- **Output dependence:** η J γράφει τον r1 πριν τον γράψει η I

 I: sub r1,r4,r3  
J: add r1,r2,r3  
K: mul r6,r1,r7

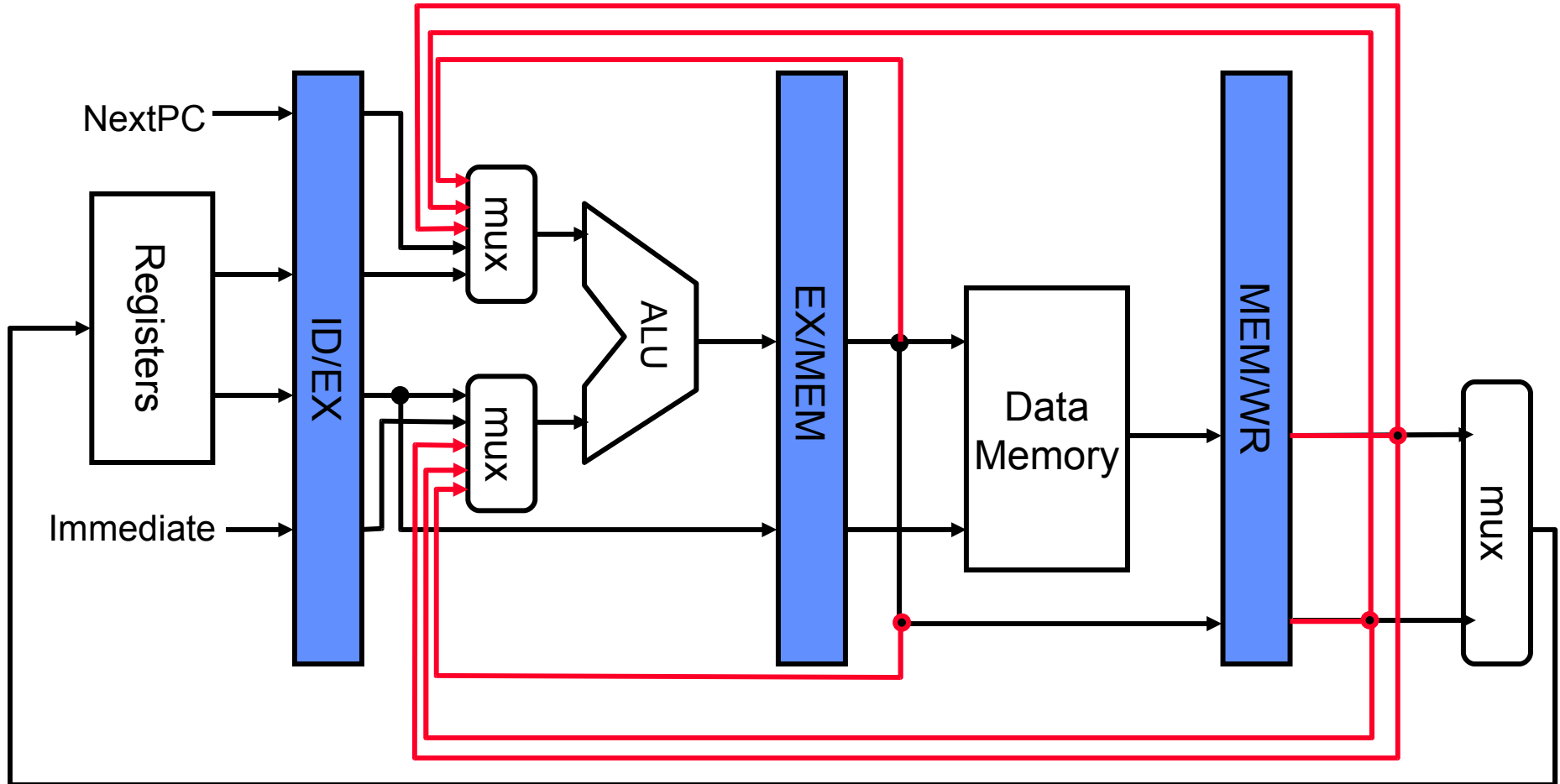
- Προκαλούν **Write After Write (WAW) data hazards** στο pipeline
- Δε μπορούν να συμβούν στο κλασικό 5-stage pipeline διότι:
  - όλες οι εντολές χρειάζονται 5 κύκλους για να εκτελεστούν, και
  - οι εγγραφές συμβαίνουν πάντα στο στάδιο 5
- Τόσο οι WAW όσο και οι WAR κίνδυνοι συναντώνται σε πιο περίπλοκα pipelines (π.χ. multiple cycle, out-of-order execution)

# Πρώθηση



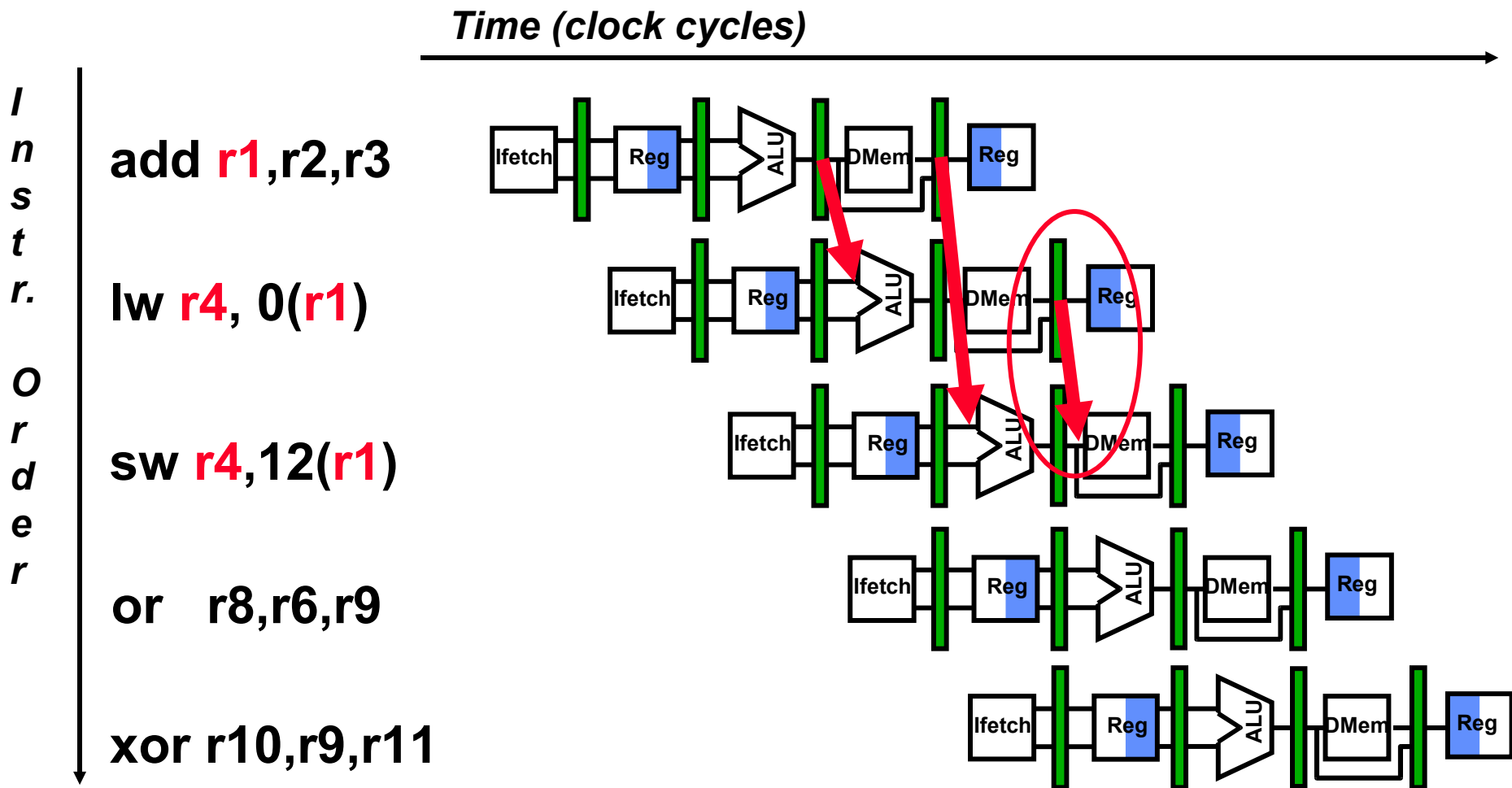
- μειώνονται τα RAW hazards

# Αλλαγές στο hardware για την υποστήριξη προώθησης



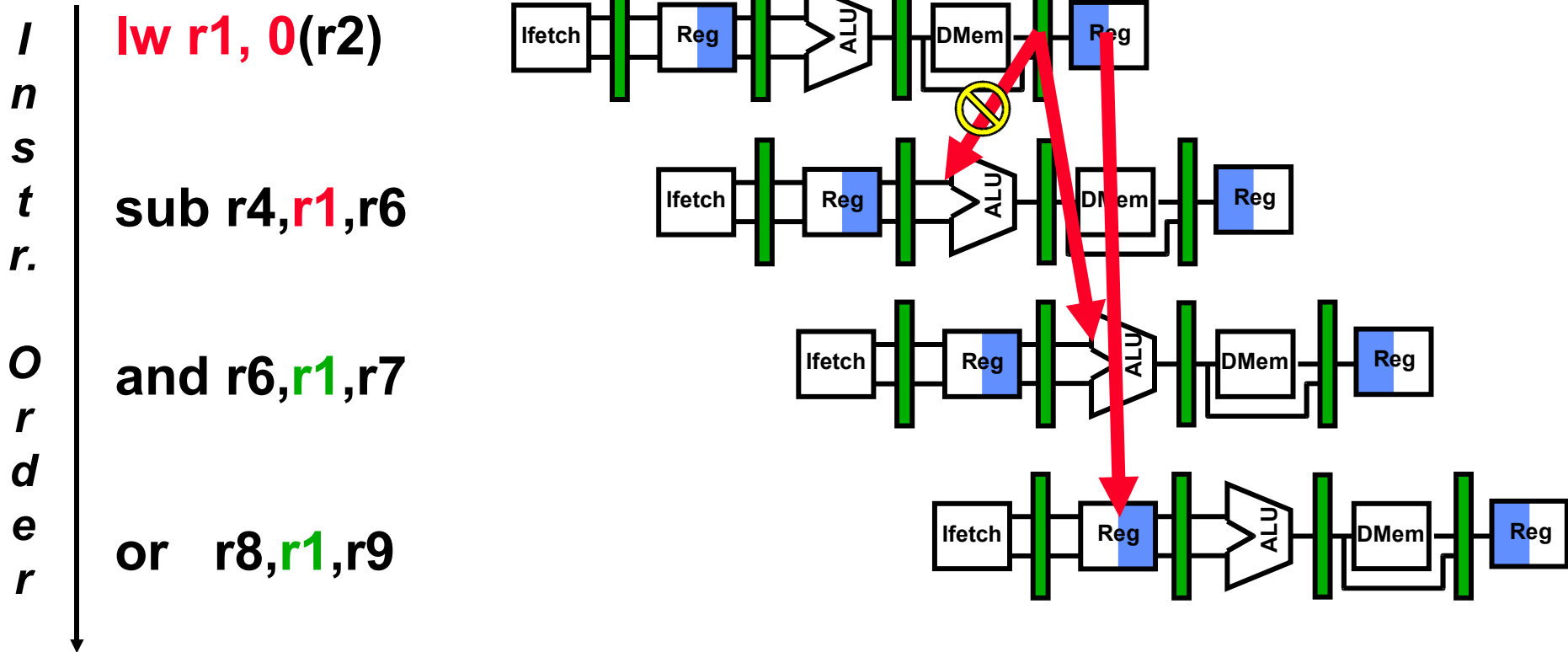
Προώθηση EX->EX, MEM->EX

# Πρώθηση MEM->MEM



# Τα data hazards δεν εξαφανίζονται πλήρως με την προώθηση

Time (clock cycles)



# Τα data hazards δεν εξαφανίζονται πλήρως με την προώθηση

*Time (clock cycles)*

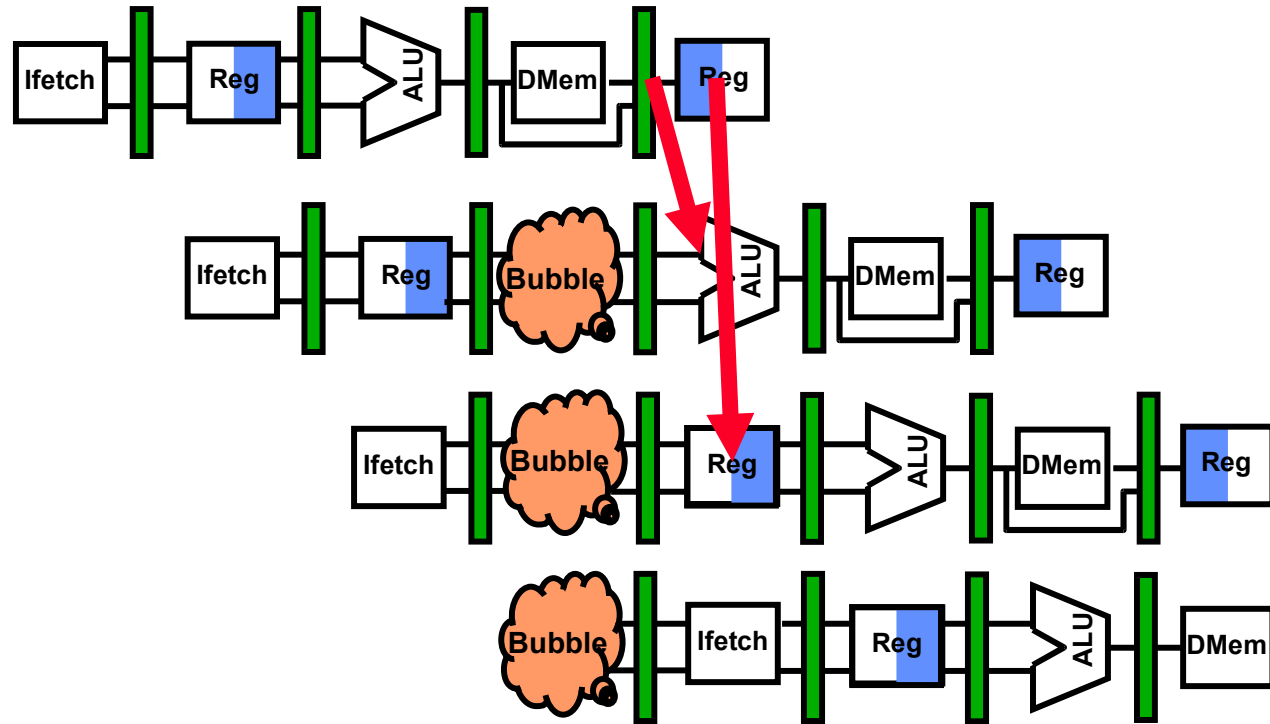
*I  
n  
s  
t  
r.  
O  
r  
d  
e  
r*

**lw r1, 0(r2)**

**sub r4, r1, r6**

**and r6, r1, r7**

**or r8, r1, r9**



# Αναδιάταξη εντολών για την αποφυγή RAW hazards

Πώς μπορούμε να παράξουμε γρηγορότερο κώδικα assembly για τις ακόλουθες πράξεις?

$a = b + c;$

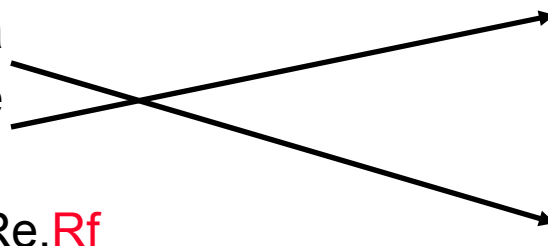
$d = e - f;$

Slow code:

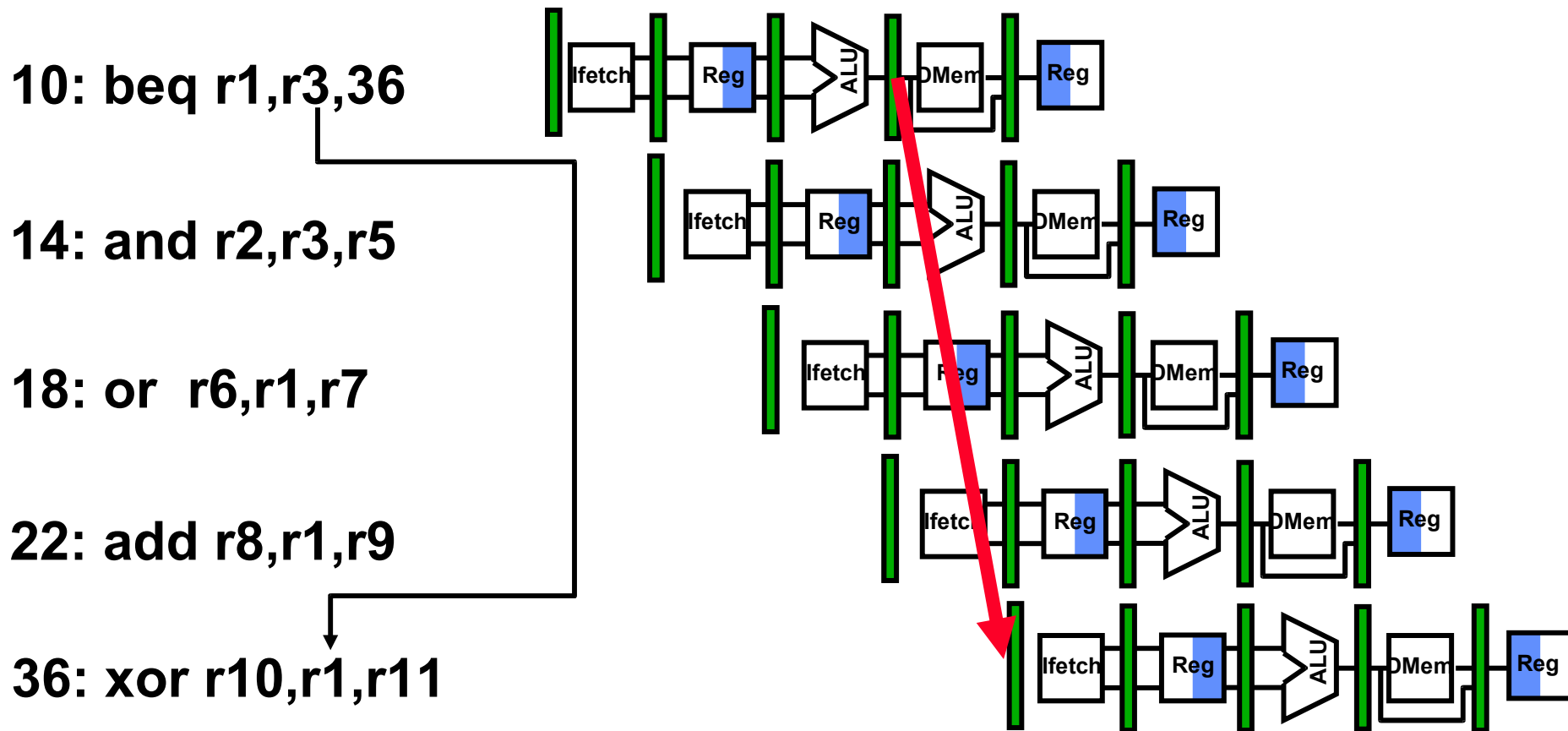
```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

Fast code:

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```



# Κίνδυνοι ελέγχου στις εντολές διακλάδωσης: stalls 3 σταδίων

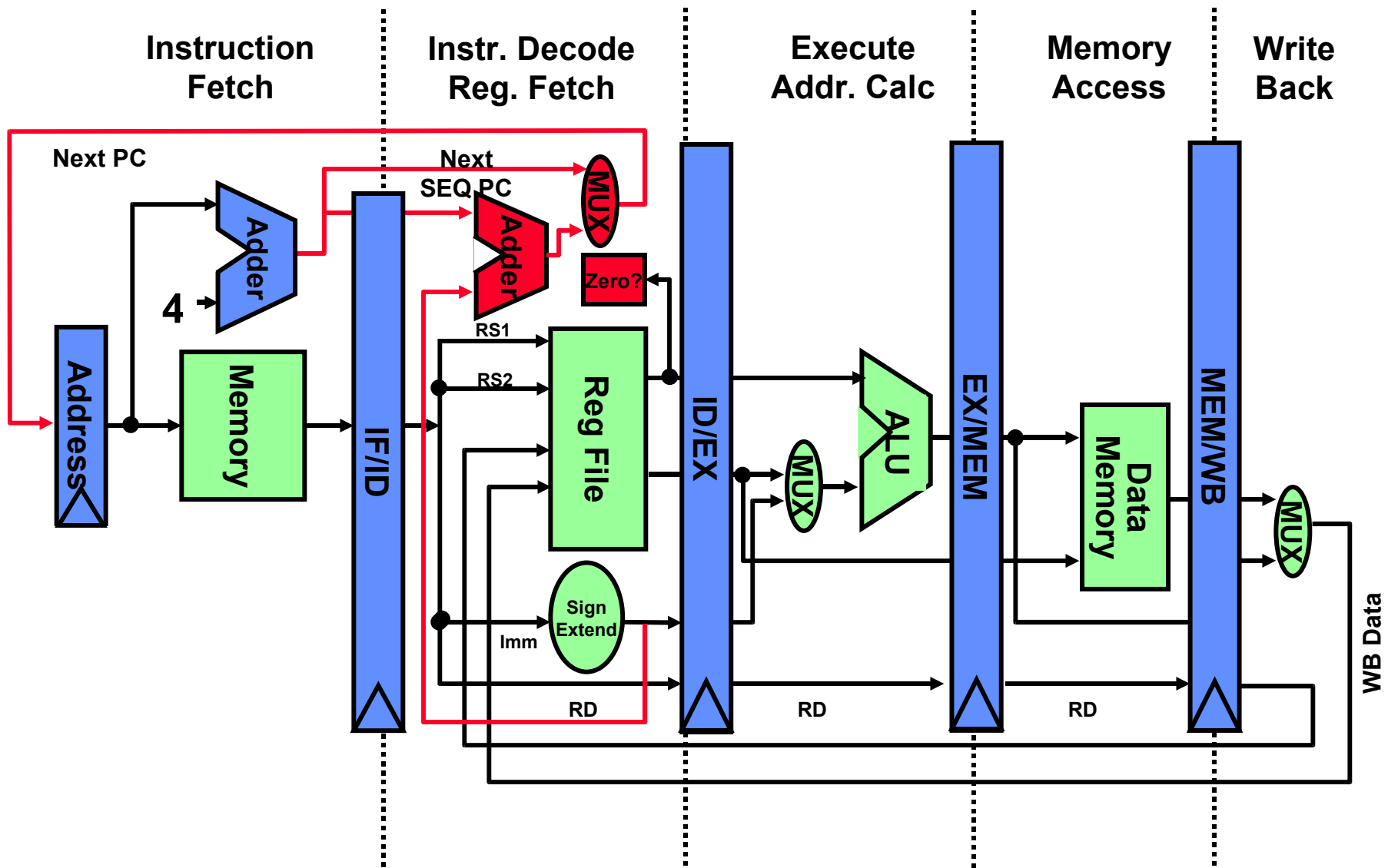




## Επίπτωση των branch stalls

- Αν  $CPI = 1$ , η συχνότητα των branches 30%, και τα branch stalls διαρκούν 3 κύκλους  $\Rightarrow$  νέο  $CPI = 1.9!$
- Μια λύση: καθόρισε το αποτέλεσμα του branch νωρίτερα, ΚΑΙ υπολόγισε τη διεύθυνση-στόχο του branch νωρίτερα
  - μετακίνηση του ελέγχου ισότητας ενός καταχωρητή με το 0 στο στάδιο ID/RF
  - προσθήκη αθροιστή στο στάδιο ID/RF για τον υπολογισμό του PC της διεύθυνσης-στόχου
  - 1 κύκλος branch penalty έναντι 3

# Τροποποιήσεις στο pipeline



# 4 εναλλακτικές προσεγγίσεις για την αντιμετώπιση των control hazards

#1: Πάγωμα του pipeline μέχρι ο στόχος του branch να γίνει γνωστός

#2: Πρόβλεψη “Not Taken” για κάθε branch

- συνεχίζουμε να φορτώνουμε τις επόμενες εντολές, σα να ήταν η εντολή branch μια «κανονική» εντολή
- απορρίπτουμε από το pipeline αυτές τις εντολές, αν τελικά το branch είναι “Taken”
- το PC+4 είναι ήδη υπολογισμένο, το χρησιμοποιούμε για να πάρουμε την επόμενη εντολή

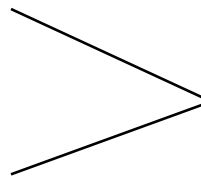
#3: Πρόβλεψη “Taken” για κάθε branch

- φορτώνουμε εντολές αρχίζοντας από τη διεύθυνση-στόχο του branch
- Στον MIPS η διεύθυνση-στόχος δε γίνεται γνωστή πριν το αποτέλεσμα του branch
  - » κανένα επιπλέον πλεονέκτημα στον MIPS (1 cycle branch penalty)
  - » θα είχε νόημα σε άλλα pipelines, όπου η διεύθυνση-στόχος γίνεται γνωστή πριν το αποτέλεσμα του branch

# 4 εναλλακτικές προσεγγίσεις για την αντιμετώπιση των control hazards

## #4: Delayed Branches

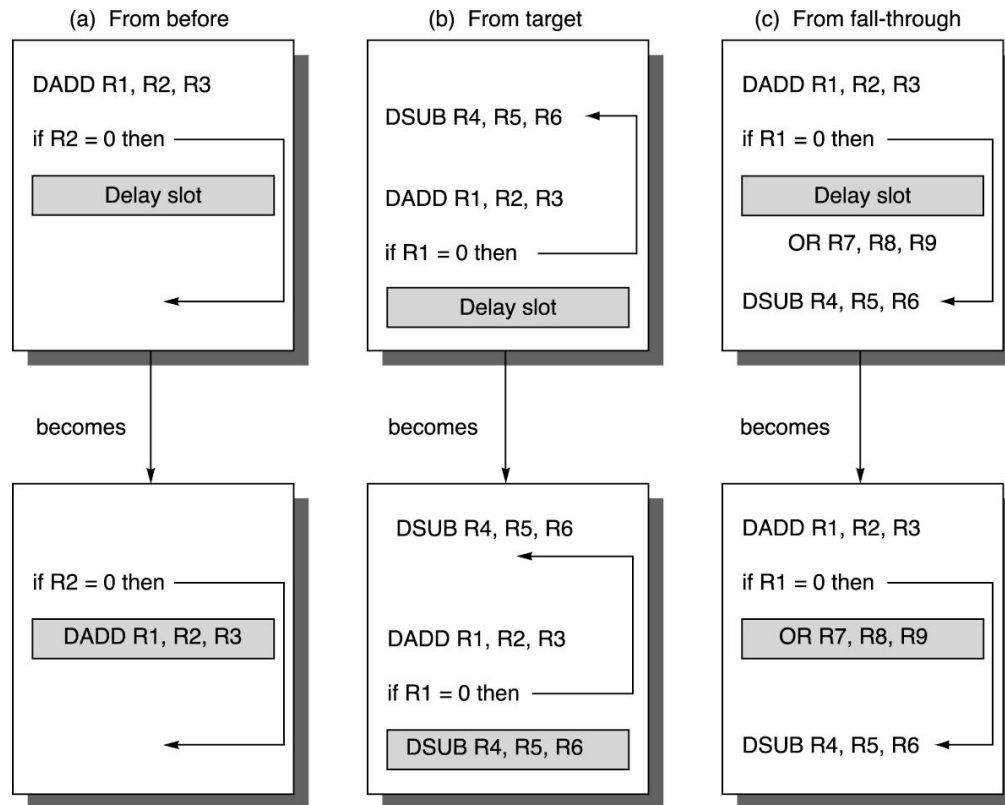
```
branch instruction  
  sequential successor1  
  sequential successor2  
  .....  
  sequential successorn  
branch target if taken
```



Branch delay μήκους n:  
οι εντολές εκτελούνται είτε το  
branch είναι Taken είτε όχι

- delay ενός slot: επιτρέπει απόφαση και υπολογισμό διεύθυνσης-στόχου στο 5-stage pipeline χωρίς stalls

# «Δρομολόγηση» ενός branch delay slot



© 2003 Elsevier Science (USA). All rights reserved.

- το (α) είναι η καλύτερη επιλογή: γεμίζει το delay slot και μειώνει τον αριθμό εντολών

# Περιορισμοί των βαθμωτών αρχιτεκτονικών

- Μέγιστο throughput: 1 εντολή/κύκλο ρολογιού ( $IPC \leq 1$ )
- Υποχρεωτική ροή όλων των (διαφορετικών) τύπων εντολών μέσα από κοινή σωλήνωση
- Εισαγωγή καθυστερήσεων σε ολόκληρη την ακολουθία εκτέλεσης λόγω stalls μίας εντολής (οι απόλυτα βαθμωτές αρχιτεκτονικές πραγματοποιούν εν σειρά (in-order) εκτέλεση των εντολών)

# Πώς μπορούν να ξεπεραστούν οι περιορισμοί;

- Εκτέλεση πολλαπλών εντολών ανά κύκλο μηχανής (παράλληλη εκτέλεση)  
→ υπερβαθμωτές αρχιτεκτονικές
- Ενσωμάτωση διαφορετικών αγωγών ροής δεδομένων, ο καθένας με όμοιες (πολλαπλή εμφάνιση του ίδιου τύπου) ή και ετερογενείς λειτουργικές μονάδες  
→ *multicycle operations*
- Δυνατότητα εκτέλεσης εκτός σειράς (out-of-order) των εντολών  
→ δυναμικές αρχιτεκτονικές

Εναλλακτικά...

**Pipeline CPI =**

**Ideal pipeline CPI +**

**Structural Stalls +**

**Data Hazard Stalls +**

**Control Stalls**

μέτρο της μέγιστης απόδοσης που μπορούμε να έχουμε με την εκάστοτε υλοποίηση του pipeline



Εναλλακτικά...

**Pipeline CPI =**

**Ideal pipeline CPI +**

**Structural Stalls +**

**Data Hazard Stalls +**

**Control Stalls**

υπερβαθμωτή  
εκτέλεση

προώθηση

υποθετική  
εκτέλεση

register  
renaming

δυναμική  
εκτέλεση

loop unrolling

static scheduling,  
software pipelining

πρόβλεψη  
διακλαδώσεων

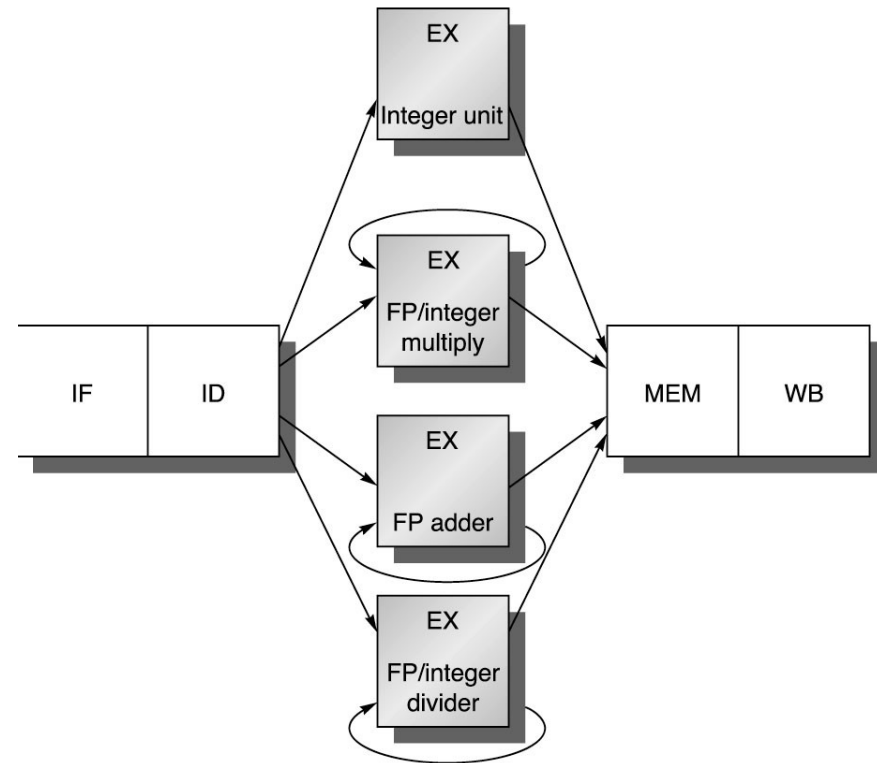
delayed branches,  
branch scheduling

# Λειτουργίες πολλαπλών κύκλων

- Στο κλασικό 5-stage pipeline όλες οι λειτουργίες χρειάζονται 1 κύκλο
- Το να έχουμε όλες τις εντολές να τελειώνουν σε έναν κύκλο σημαίνει:
  - μείωση της συχνότητας για να προσαρμοστεί το pipeline στη διάρκεια της πιο χρονοβόρας λειτουργίας, ή
  - χρησιμοποίηση εξαιρετικά πολύπλοκων κυκλωμάτων για την υλοποίηση της πιο χρονοβόρας λειτουργίας σε 1 κύκλο
- Ρεαλιστική αντιμετώπιση:
  - επέκταση του pipeline για να υποστηρίζει λειτουργίες διαφορετικής διάρκειας
- Παραδείγματα από πραγματικούς επεξεργαστές:
  - FP add, Int/FP mult: ~2-6 κύκλους
  - Int/FP div, sqrt: ~20-50 κύκλους
  - Προσπέλαση στη μνήμη: ~2-200 κύκλους...

# Multi-cycle execution: επέκταση του 5-stage pipeline με non-pipelined FP units

- 4 διαφορετικές ALUs:
  - Integer
  - FP/Integer multiply
  - FP adder
  - FP/Integer divide
- φανταζόμαστε το EX για τις FP εντολές σαν να επαναλαμβάνεται για πολλούς συνεχόμενους κύκλους
- κάθε τέτοια μονάδα είναι **non-pipelined**: δε μπορεί να σταλεί (“issue”) προς εκτέλεση σε μια μονάδα μια εντολή, αν κάποια προηγούμενη χρησιμοποιεί ακόμα τη μονάδα αυτή (**structural hazard**)
- η εντολή που stall-άρει καθυστερεί και όλες τις επόμενες



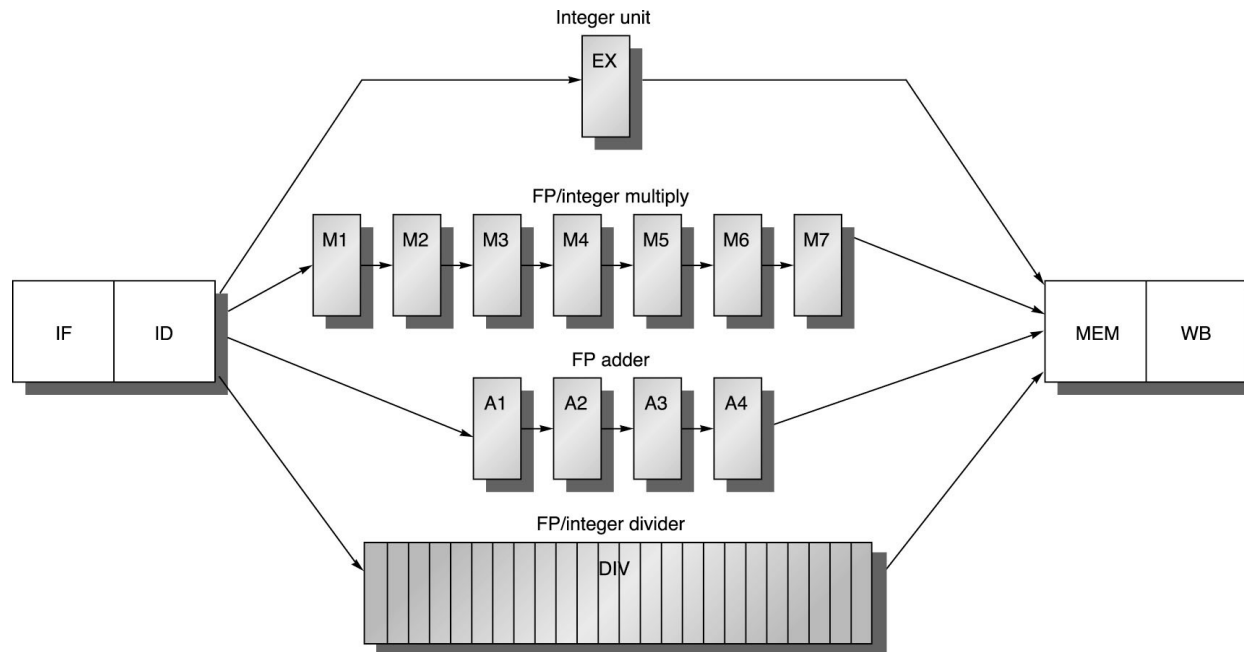
# Μετρικές για την περιγραφή ενός multi-cycle pipeline

- **Initiation interval:** #κύκλων που μεσολαβούν ανάμεσα στην αποστολή στις μονάδες εκτέλεσης δύο εντολών ίδιου τύπου
  - ενδεικτικό του throughput μιας pipelined μονάδας εκτέλεσης
- **Latency:** #κύκλων που μεσολαβούν από τη στιγμή που μια εντολή παράγει ένα αποτέλεσμα, μέχρι τη στιγμή που μια άλλη το καταναλώσει

Unit	Latency	Initiation interval
<b>Integer ALU</b>	<b>0</b>	<b>1</b>
<b>Data memory</b>	<b>1</b>	<b>1</b>
<b>FP add</b>	<b>3</b>	<b>1</b>
<b>FP multiply</b>	<b>6</b>	<b>1</b>
<b>FP divide</b>	<b>24</b>	<b>25</b>

- Οι περισσότερες εντολές καταναλώνουν τους τελεστές τους στην αρχή του EX
  - Latency = #σταδίων μετά το EX όπου μια εντολή παράγει το αποτέλεσμά της

# Multi-cycle execution: επέκταση του 5-stage pipeline με pipelined FP units



© 2003 Elsevier Science (USA). All rights reserved.

- επιτρέπει να βρίσκονται εν εκτελέσει μέχρι 4 FP-adds, 7 FP-muls, 1 FP-divide (non-pipelined)
- τα επιμέρους στάδια είναι ανεξάρτητα και χωρίζονται με ενδιάμεσους καταχωρητές
- διάσπαση μιας λειτουργίας σε πολλά επιμέρους στάδια:
  - ↑ συχνότητα ρολογιού
  - ↑ latency λειτουργιών + ↑ συχνότητα RAW hazards + ↑ stalls



# Hazards

- Νέα ζητήματα που προκύπτουν:
  - η μονάδα divide είναι non-pipelined → μπορεί να προκύψουν **structural hazards**
  - εντολές διαφορετικού latency → #εγγραφών στο register file σε έναν κύκλο μπορεί να είναι  $>1$  (**structural hazards**)
  - οι εντολές μπορεί να φτάσουν στο WB εκτός σειράς προγράμματος → μπορούν να συμβούν **WAW hazards** (**WAR?**)
  - οι εντολές μπορεί να ολοκληρωθούν εκτός σειράς προγράμματος → πρόβλημα με τις εξαιρέσεις
  - μεγαλύτερο latency στις εντολές → συχνότερα τα stalls εξαιτίας **RAW hazards**

## RAW hazards και αύξηση των stalls

Clock Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
L.D F4,0(R2)	IF	ID	EX	M	WB													
MUL.D F0,F4,F6		IF	ID	S	M1	M2	M3	M4	M5	M6	M7	M	WB					
ADD.D F2,F0,F8			IF	S	ID	S	S	S	S	S	S	A1	A2	A3	A4	M	WB	
S.D F2,0(R2)					IF	S	S	S	S	S	S	ID	EX	S	S	S	M	WB

- full bypassing/forwarding
- η S.D πρέπει να καθυστερήσει έναν κύκλο παραπάνω για να αποφύγουμε το conflict στο MEM της ADD.D



## Structural Hazards

Clock Cycles    1   2   3   4   5   6   7   8   9   10   11   12   13   14   15

MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	M	WB				
...		IF	ID	EX	M	WB									
...			IF	ID	EX	M	WB								
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	M	WB				
...					IF	ID	EX	M	WB						
...						IF	ID	EX	M	WB					
L.D F2,0(R2)							IF	ID	EX	M	WB				

Με μόνο ένα write port στο register file → structural hazard

- multiple write ports (...όμως δεν είναι η συνήθης περίπτωση στα προγράμματα)
- interlocks:
  - ελέγχουμε στο ID το ενδεχόμενο η τρέχουσα εντολή να έχει conflict στο WB με μία προηγούμενη εντολή, και αν ισχύει αυτό την stall-άρουμε στο ID (προτού γίνει issue)
  - ελέγχουμε το ενδεχόμενο η εντολή να έχει conflict στο WB όταν αυτή πάει να μπει στο MEM ή στο WB, και αν ισχύει αυτό τη stall-άρουμε εκεί

## WAW Hazards

Clock Cycles    1   2   3   4   5   6   7   8   9   10   11   12   13   14   15

...		IF	ID	EX	M	WB									
...			IF	ID	EX	M	WB								
ADD.D F2,F4,F6 inst				IF	ID	A1	A2	A3	A4	M	WB				
L.D F2,0(R2)					IF	ID	EX	M	WB						

- Μπορεί να προκύψει μόνο αν η “inst” δε χρησιμοποιεί τον F2
  - ποιος ο λόγος να έχουμε δύο producers του ίδιου πράγματος χωρίς ενδιάμεσο consumer???
  - σπάνια περίπτωση, αφού ένας «λογικός» compiler θα έκανε eliminate τον πρώτο producer
  - μπορεί όμως να συμβεί!! π.χ. όταν η σειρά εκτέλεσης των εντολών δεν είναι η αναμενόμενη ( branch delay slots, εντολές σε trap handlers, κ.λπ.)
- Το hazard ανιχνεύεται στο ID, όταν η L.D είναι έτοιμη να γίνει issue
- Αντιμετώπιση:
  - καθυστερούμε το issue της L.D μέχρι η ADD.D να μπει στο MEM
  - απορρίπτουμε την εγγραφή της ADD.D → η L.D μπορεί να γίνει issue άμεσα
- Η δυσκολία δεν έγκειται τόσο στην αντιμετώπιση του hazard, όσο στο να βρούμε ότι η L.D μπορεί να ολοκληρωθεί πιο γρήγορα από την ADD.D

# Συνοψίζοντας

- Τρεις επιπλέον έλεγχοι για hazards στο ID προτού μια εντολή γίνει issue:
  - structural:
    - » εξασφάλισε ότι η (non-pipelined) μονάδα δεν είναι απασχολημένη
    - » εξασφάλισε ότι το write port του register file θα είναι διαθέσιμο όταν θα ζητηθεί
  - RAW:
    - » περίμενε μέχρι οι source registers της issuing εντολής να μην υπάρχουν πλέον σαν destinations στους ενδιάμεσους pipeline registers των pipelined μονάδων εκτέλεσης
    - » π.χ. για τη μονάδα FP-add: αν η εντολή στο ID έχει σαν source τον F2, τότε για να μπορεί να γίνει issue, ο F2 δε θα πρέπει να συγκαταλλέγεται στα destinations των ID/A1, A1/A2, A2/A3.
  - WAW:
    - » έλεγξε αν κάποια εντολή στα στάδια A1,...,A4,D,M1,...,M7 έχει τον ίδιο destination register με αυτή στο ID, και αν ναι, stall-αρε την τελευταία στο ID
- Προώθηση:
  - έλεγξε αν το destination κάποιου από τους EX/MEM, A4/MEM, M7/MEM, D/MEM, MEM/WB ταυτίζεται με τον source register κάποιας FP εντολής, και αν ναι, ενεργοποίησε τον κατάλληλο πολυπλέκτη

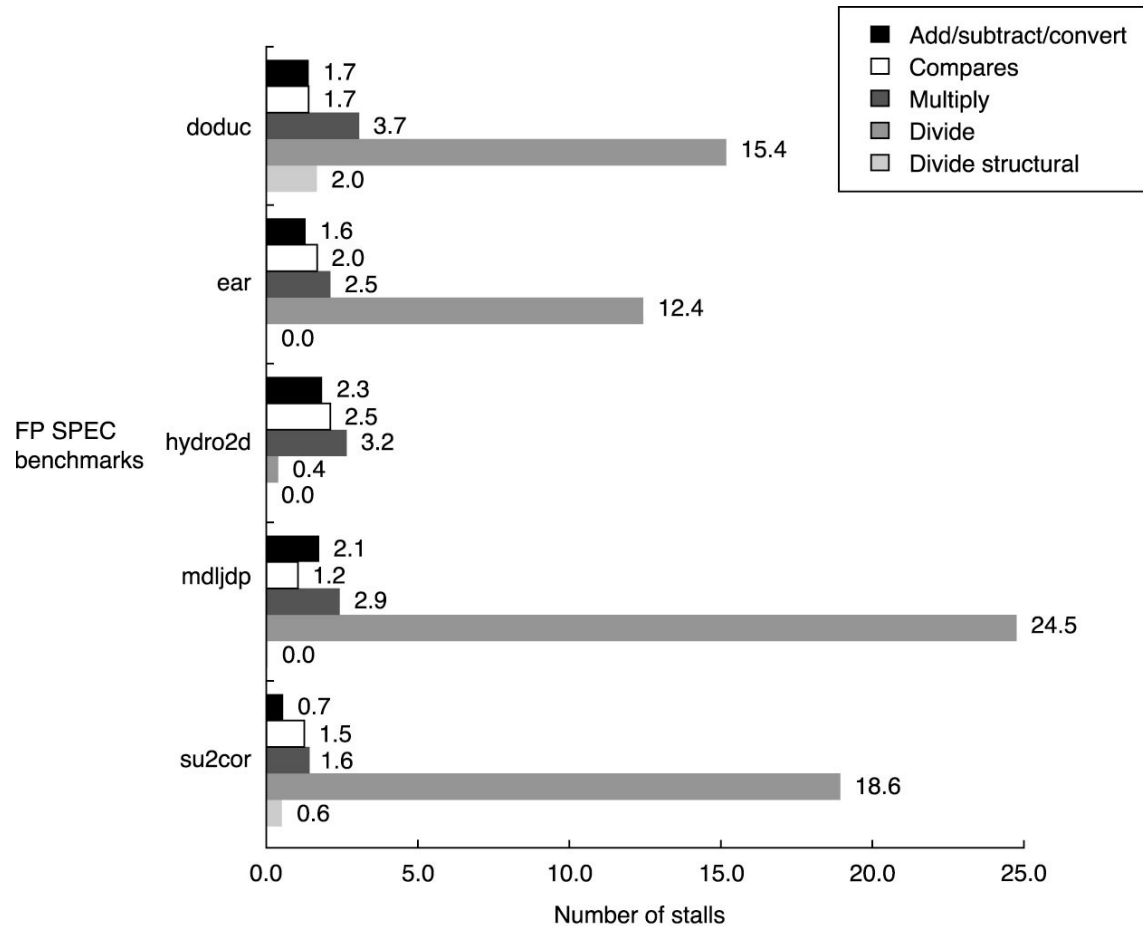
# Απόδοση του multi-cycle FP pipeline

- stall cycles ανά FP-εντολή

- τα RAW hazard stalls «ακολουθούν» το latency της αντίστοιχης μονάδας, π.χ.:

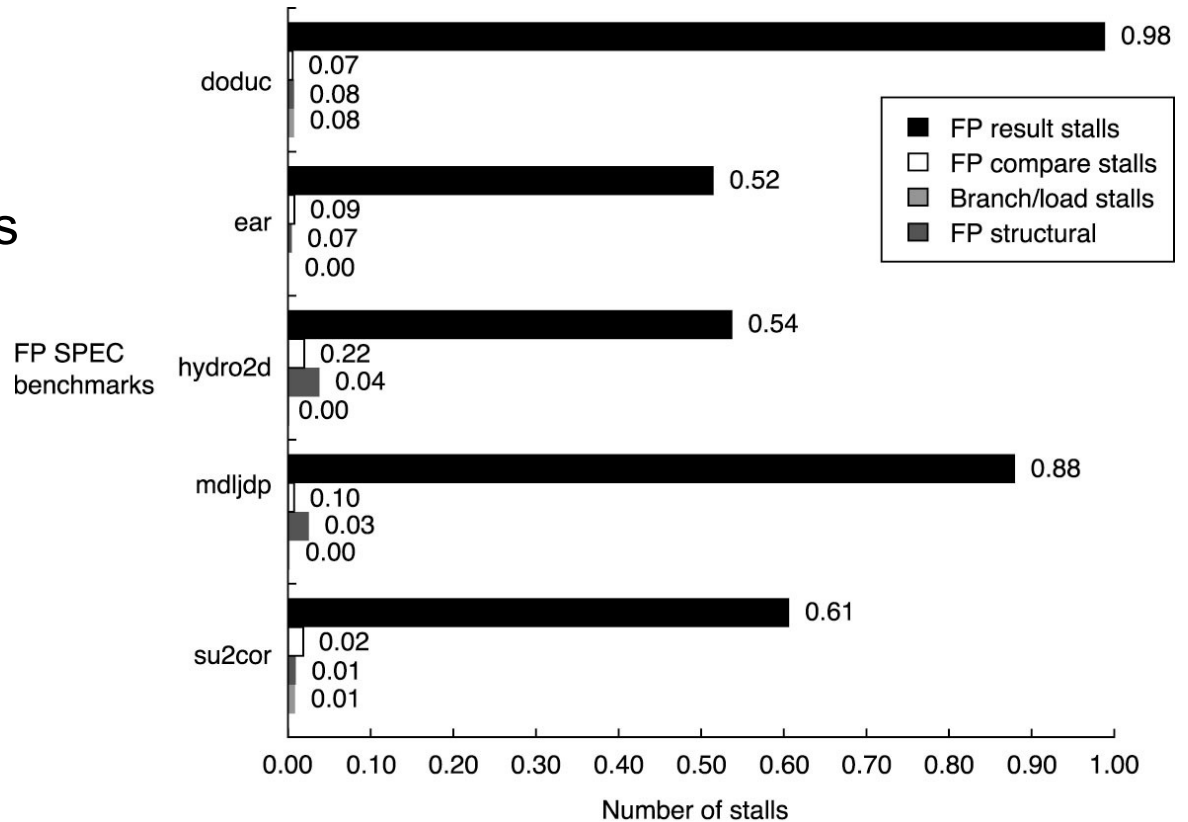
- μέσος #stalls που οφείλεται στην MUL.D=2.8 (46% του latency της FP-Mult)
- μέσος #stalls που οφείλονται στην DIV.D=14.2 (59% του latency της FP-Div)

- τα structural hazards είναι σπάνια, επειδή τα divides δεν είναι συχνά

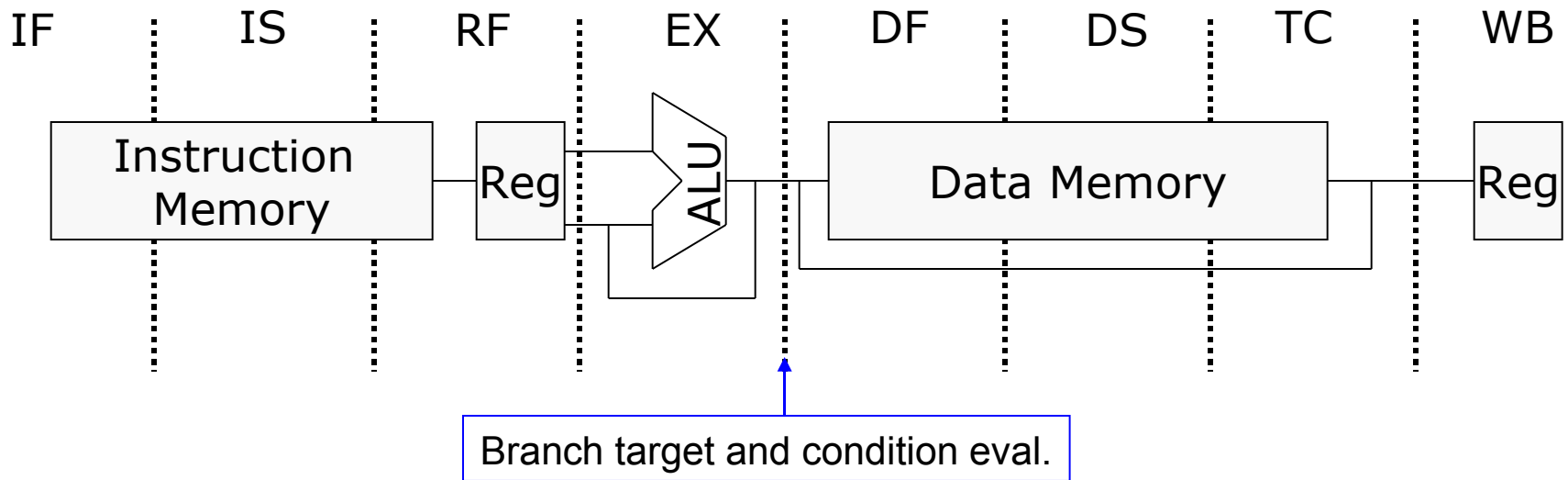


# Απόδοση του multi-cycle FP pipeline

- stalls ανά εντολή + breakdown
- από 0.65 μέχρι 1.21 stalls ανά εντολή
- κυριαρχούν τα RAW hazard stalls («FP result stalls»)

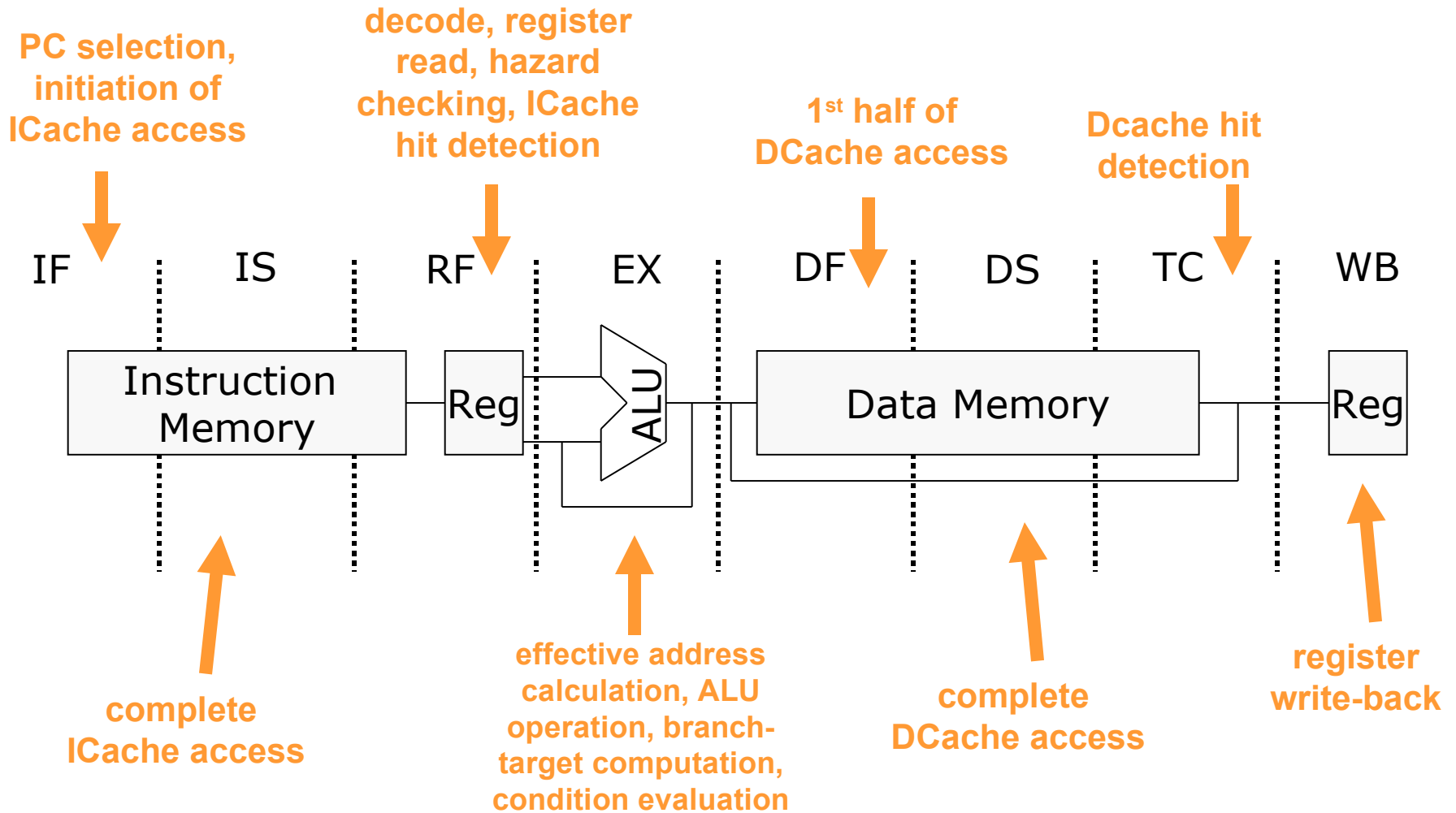


## Case-study: MIPS R4000 Pipeline

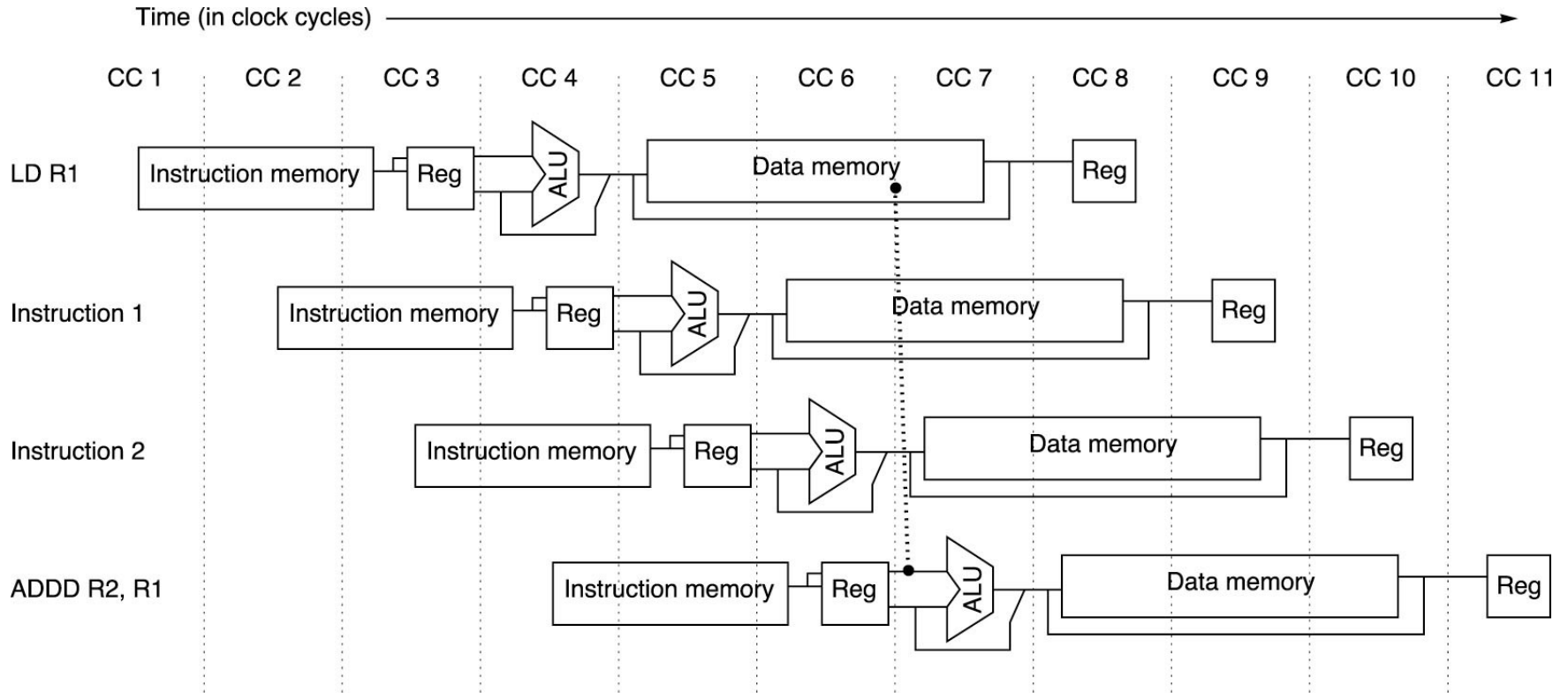


- Deeper Pipeline (superpipelining): επιτρέπει υψηλότερα clock rates
- Fully pipelined memory accesses (2 cycle delays για loads)
- Predicted-Not-Taken πολιτική
  - Not-taken (fall-through) branch : 1 delay slot
  - Taken branch: 1 delay slot + 2 idle cycles

# Case-study: MIPS R4000 Pipeline



# Load delay (2 cycles)



© 2003 Elsevier Science (USA). All rights reserved.

- στην πραγματικότητα, το pipeline μπορεί να προωθήσει τα δεδομένα από την cache πριν διαπιστώσει αν είναι hit ή miss!



# Branch delay (3 cycles)

