ΗΜΥ 314 – Εργαστήριο Αρχιτεκτονικής Υπολογιστών

Εργαστήριο 2

Multi-Cycle MIPS

Μικροεπεξεργαστής MIPS: Υλοποίηση Πολλαπλών Κύκλων

Μαρία Κ. Μιχαήλ
Αναπληρώτρια Καθηγήτρια ΗΜΜΥ

Εργαστηριακή Άσκηση 2

- Η περιγραφή της εργαστηριακής άσκησης γίνεται στο ακόλουθο PDF αρχείο, που είναι αναρτημένο στο Blackboard:

ECE314_LabExercise_2

- Αφού μελετήσετε **ενδελεχώς** το πιο πάνω αρχείο, προχωρήστε στα επόμενα slides…

Εργαστηριακή Άσκηση 2

Supported MIPS32 Instructions

25 supported instructions

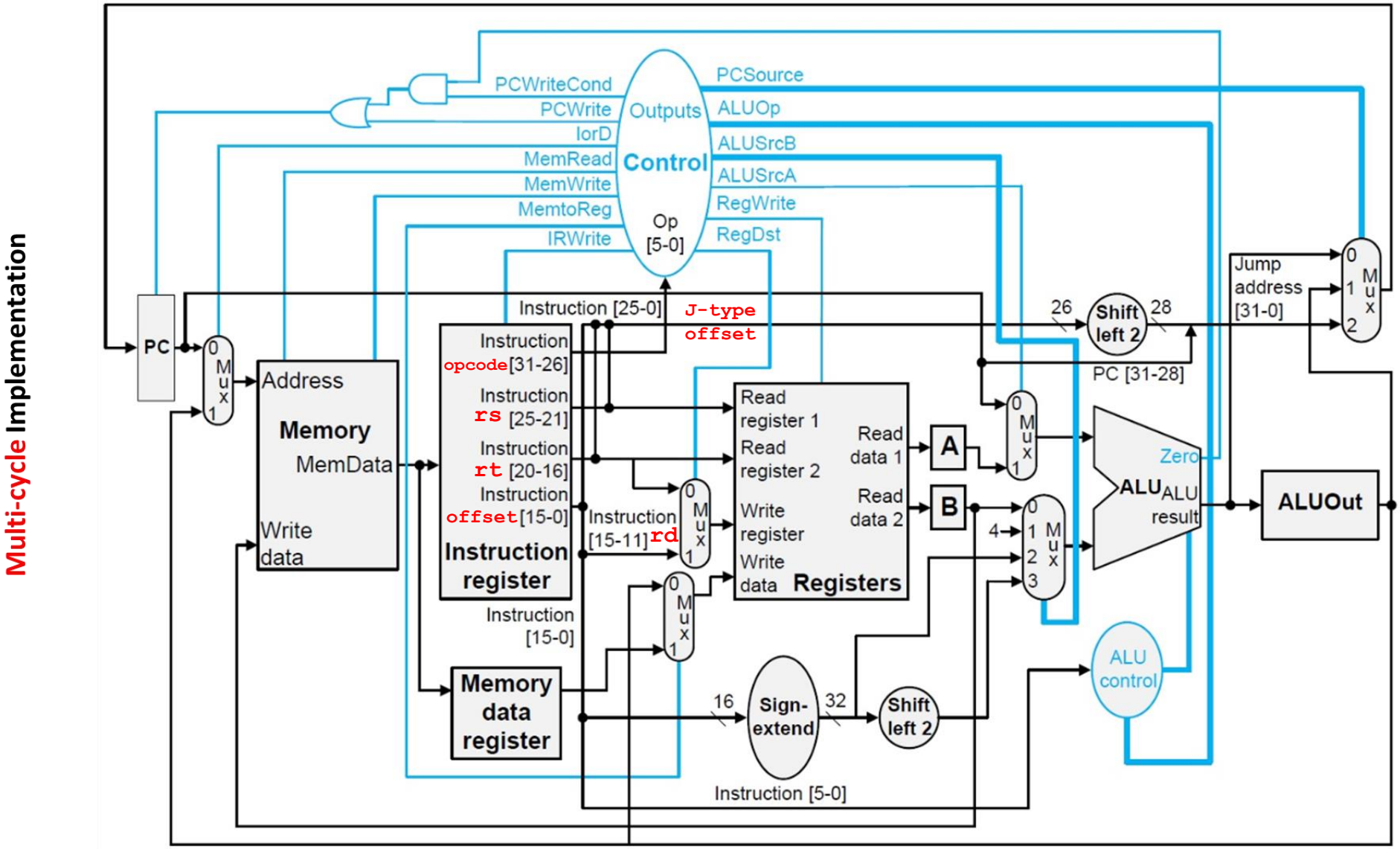
Single-cycle implementation does **NOT** support:
jal jr lui

Pipelined implementation does **NOT** support:
j jal jr lui

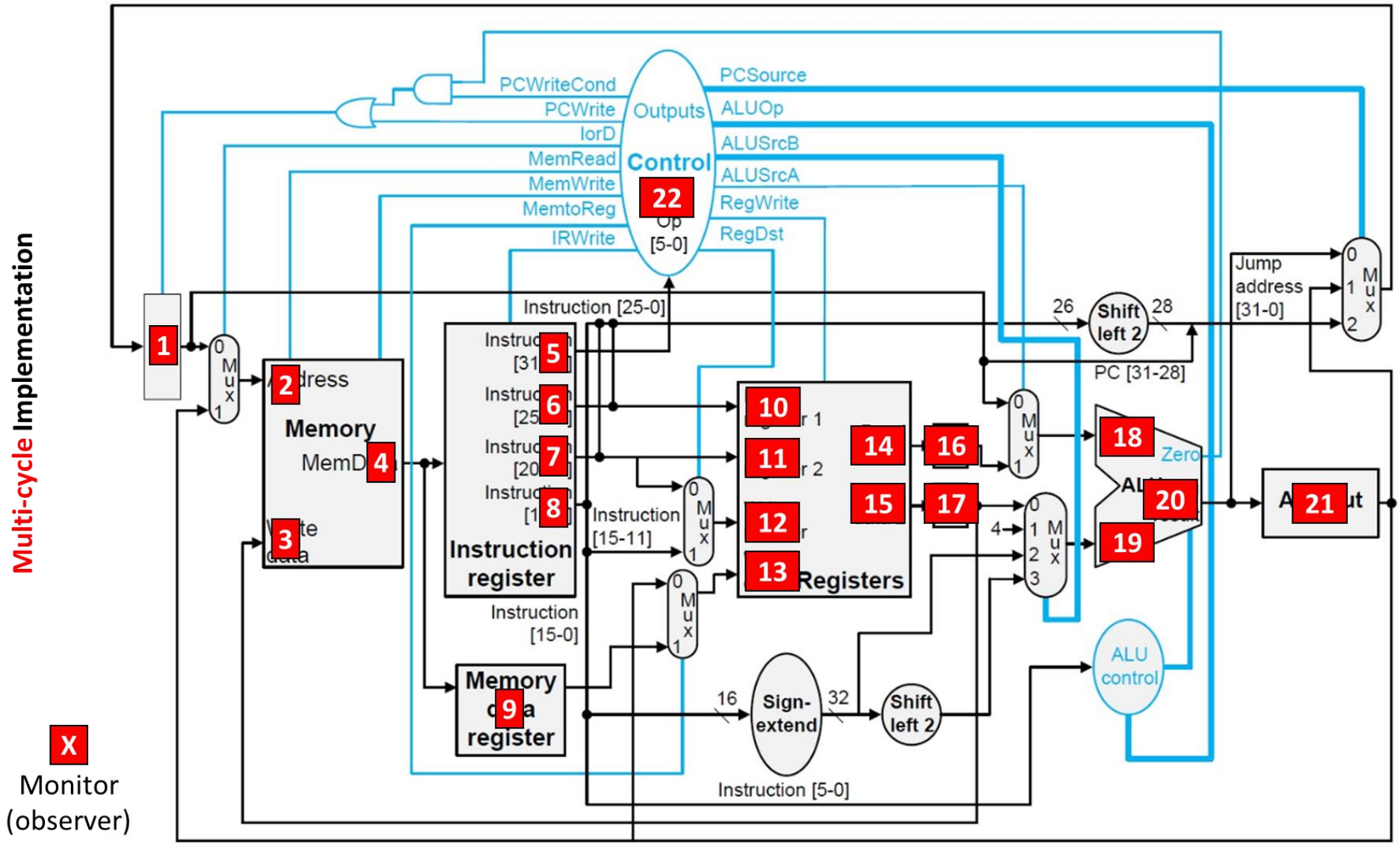
Multi-cycle implementation supports **ALL 25**

Add	add
Add Immediate	addi
Add Imm. Unsigned	addiu
Add Unsigned	addu
And	and
And Immediate	andi
Branch On Equal	beq
Branch On Not Equal	bne
Jump	j
Jump And Link	jal
Jump Register	jr
Load Upper Imm.	lui
Load Word	lw
Nor	nor
Or	or
Or Immediate	ori
Set Less Than	slt
Set Less Than Imm.	slti
Set Less Than Imm. Unsigned	sltiu
Set Less Than Unsig.	sltu
Shift Left Logical	sll
Shift Right Logical	srl
Store Word	sw
Subtract	sub
Subtract Unsigned	subu

Εργαστηριακή Άσκηση 2



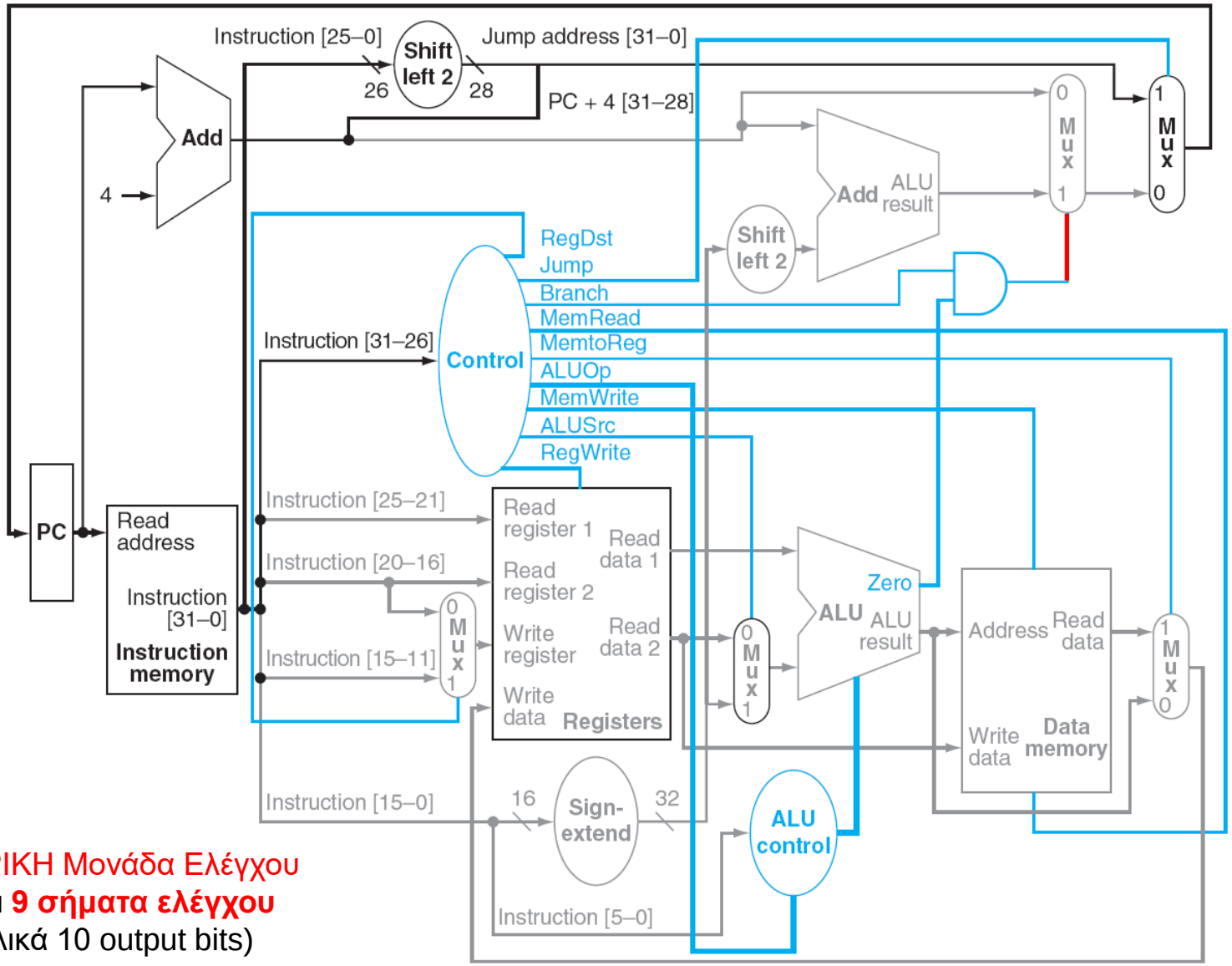
Εργαστηριακή Άσκηση 2



Επανάληψη σχετικής ύλης από ΗΜΥ 212

Σχεδιασμός επεξεργαστή MIPS με υλοποίηση **πολλαπλών κύκλων**

Επανάληψη: Υλοποίηση MIPS ενός κύκλου



Η ΚΕΝΤΡΙΚΗ Μονάδα Ελέγχου
παράγει **9 σήματα ελέγχου**
(συνολικά 10 output bits)

Προβλήματα Σχεδιασμού με ένα κύκλο ρολογιού

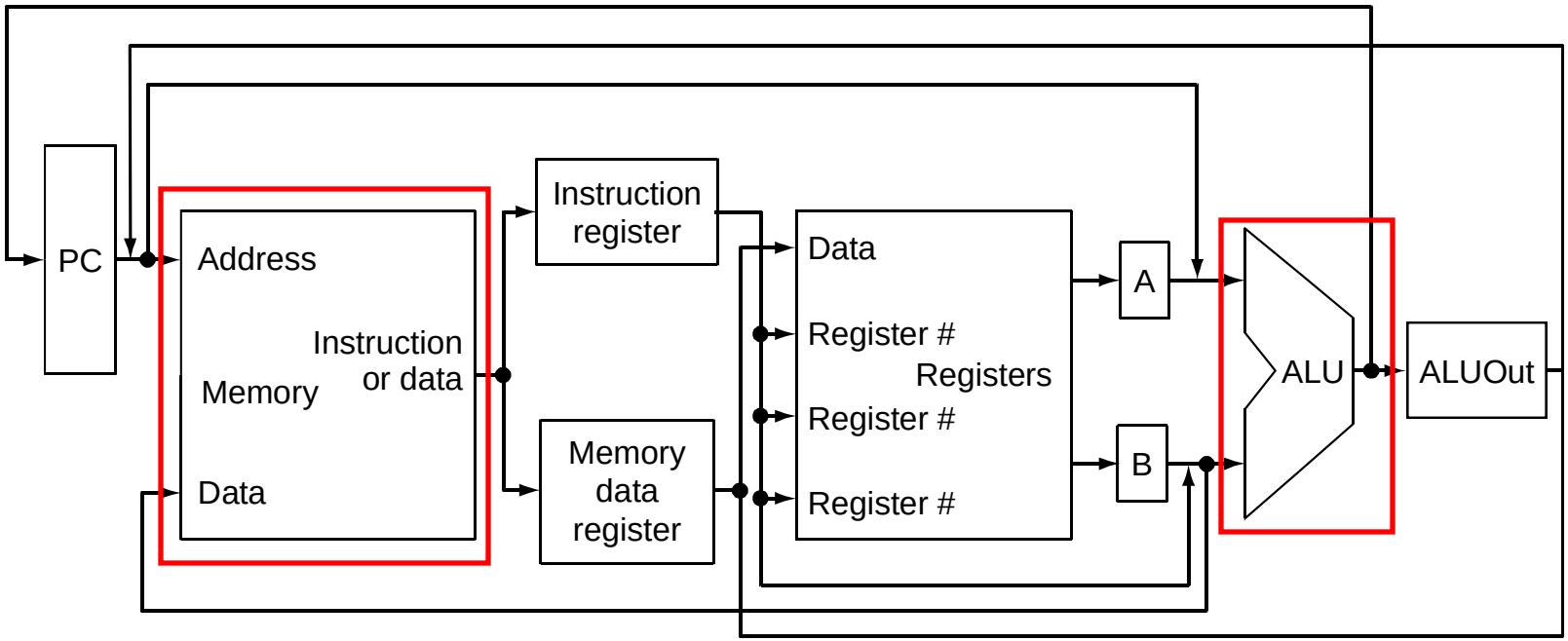
- Πρόβλημα με ένα κύκλο ρολογιού ανά εντολή:
  - Για την οργάνωση που έχουμε δει μέχρι στιγμής, 2/3 της περιόδου του ρολογιού σπαταλούνται άδικα για την εντολή **`jump`**, 1/3 σπαταλείται για εντολές τύπου R, κτλ
  - Τι θα γινόταν αν υλοποιούσαμε μια πιο σύνθετη εντολή, όπως αυτές για αριθμητική κινητής υποδιαστολής;

  → Θα σπαταλούσαμε ακόμη περισσότερο χρόνο

- Μια πιθανή λύση:
  - Χρήση «μικρότερης» περιόδου ρολογιού
  - Υλοποίηση εντολών με διαφορετικό αριθμό κύκλων ρολογιού
  - Επομένως, οργάνωση του διαδρόμου δεδομένων του μικροεπεξεργαστή βάση «πολλαπλών κύκλων»

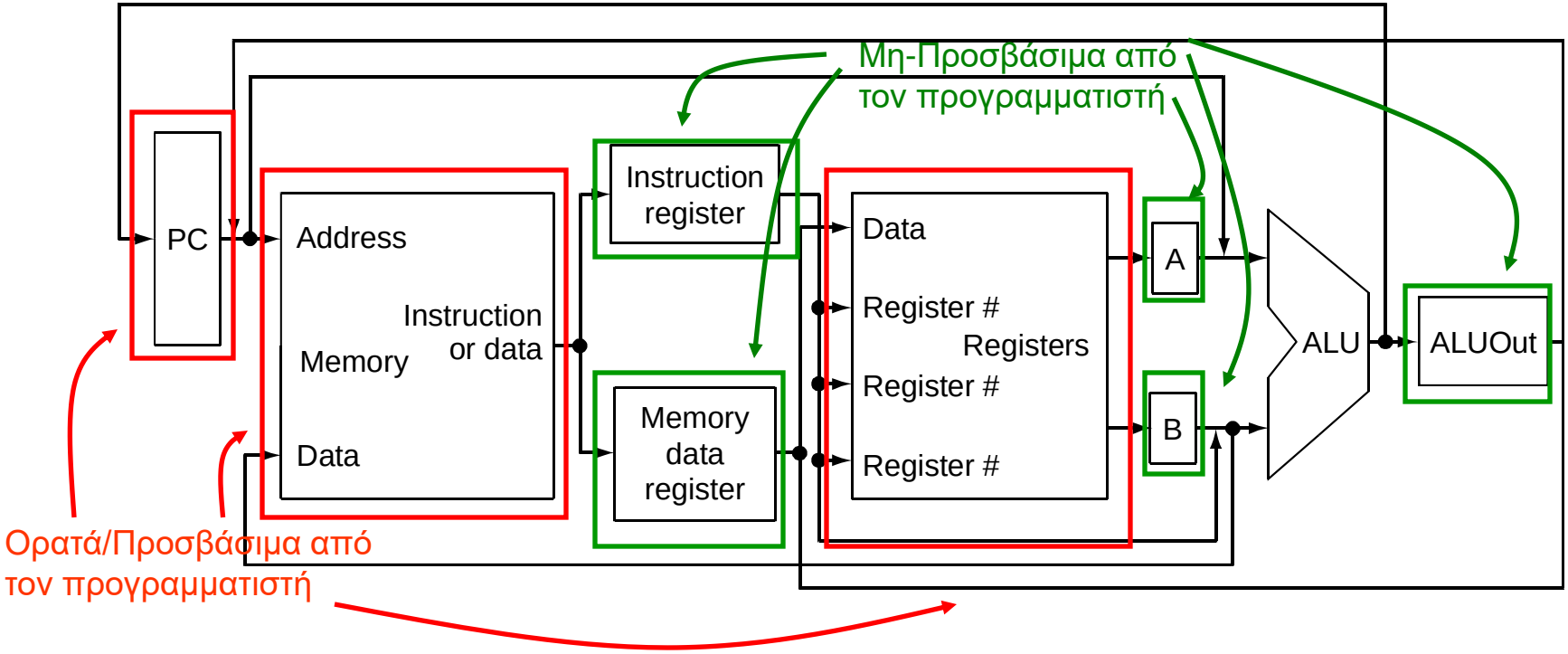
Διάδρομος Δεδομένων Πολλαπλών Κύκλων

- Θυμηθείτε τα βήματα εκτέλεσης στην μέθοδο με ένα κύκλο → ***κάθε βήμα αντιστοιχεί τώρα σε 1 κύκλο ρολογιού***
- Επιτρέπει την επαναχρησιμοποίηση των λειτουργικών μονάδων κατά τη διάρκεια μιας εντολής, εφόσον αυτή γίνεται σε διαφορετικούς κύκλους του ρολογιού
  - Το ίδιο ALU χρησιμοποιείται *και* για αριθμητικές πράξεις *και* για να υπολογίσει την διεύθυνση μνήμης *και* για να αυξήσει τον PC
  - Η μνήμη χρησιμοποιείται για εντολές ΚΑΙ δεδομένα
  - Προσθέτουμε νέους καταχωρητές για να κρατούν το αποτέλεσμα της κάθε λειτουργικής μονάδας



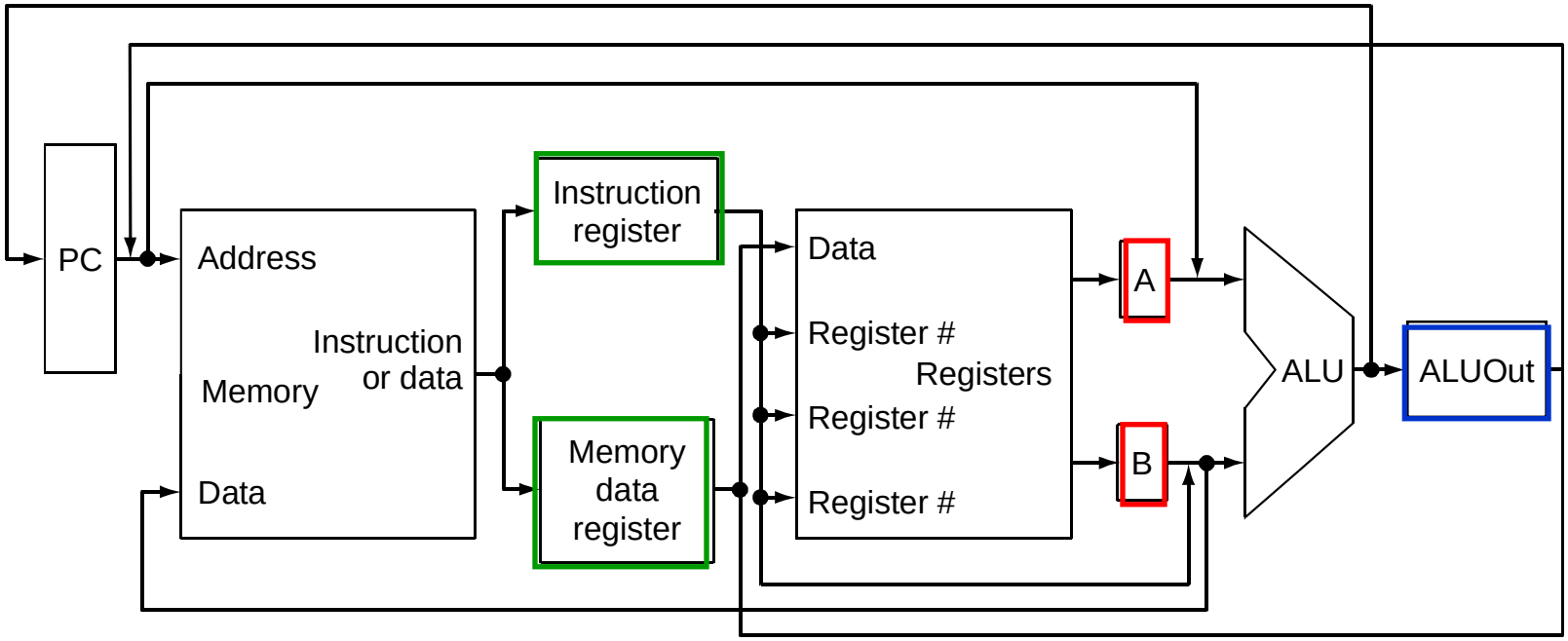
Διάδρομος Δεδομένων Πολλαπλών Κύκλων (συν.)

- Στο τέλος κάθε κύκλου, τα δεδομένα που θα χρησιμοποιηθούν σε επόμενους κύκλους πρέπει να φυλαχτούν σε στοιχεία μνήμης (*state elements*). Αυτά αποτελούνται από:
  - PC/Memory/Register file για δεδομένα που θα χρησιμοποιηθούν από (επόμενες) ***διαφορετικές εντολές***
  - Επιπρόσθετοι καταχωρητές για δεδομένα που θα χρησιμοποιηθούν από την ***ίδια εντολή σε (επόμενους) διαφορετικούς κύκλους*** της εκτέλεσής της



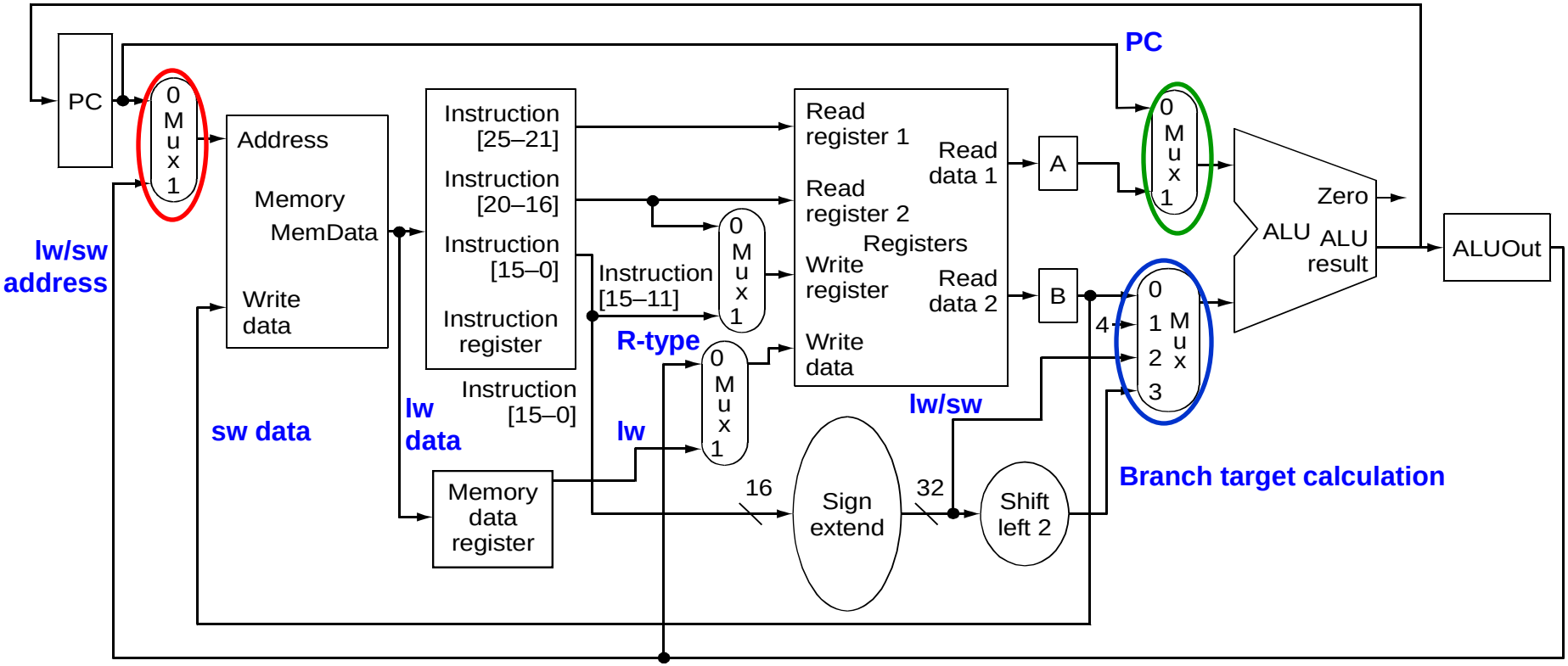
Διάδρομος Δεδομένων Πολλαπλών Κύκλων (συν.)

- Σε αυτό το σχεδιασμό, μπορεί να εκτελεστεί ένα από τα ακόλουθα σε 1 κύκλο:
  - Προσπέλαση Μνήμης
  - Προσπέλαση Αρχείου Καταχωρητών (2 reads ή 1 write)
  - Λειτουργία ALU
- Επομένως, κάθε αποτέλεσμα από τα πιο πάνω πρέπει να αποθηκευτεί σε προσωρινούς καταχωρητές. Instruction register (IR) και Memory data register (MDR) (μνήμη), A και B (αρχείο καταχωρητών) και ALUOut (ALU)



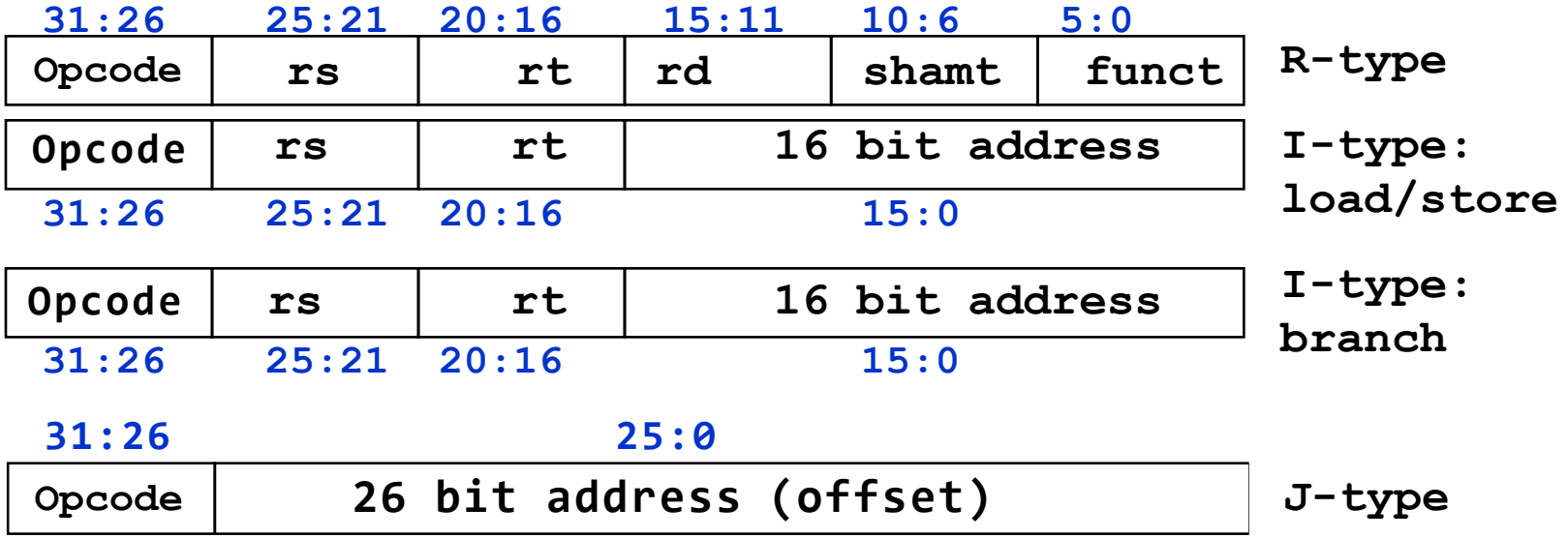
Διάδρομος Δεδομένων Πολλαπλών Κύκλων (συν.)

- Χρειαζόμαστε επιπρόσθετους πολυπλέκτες:
  - Μεταξύ PC και Μνήμης
  - 1ος τελεστής του ALU
  - Επέκταση του πολυπλέκτη για τον 2ο τελεστή του ALU
- Υπολείπεται υλικό για **`branch`** και **`jump`**



Επανάληψη: Διατάξεις Εντολών

- Θυμηθείτε τις διάφορες διατάξεις εντολών:



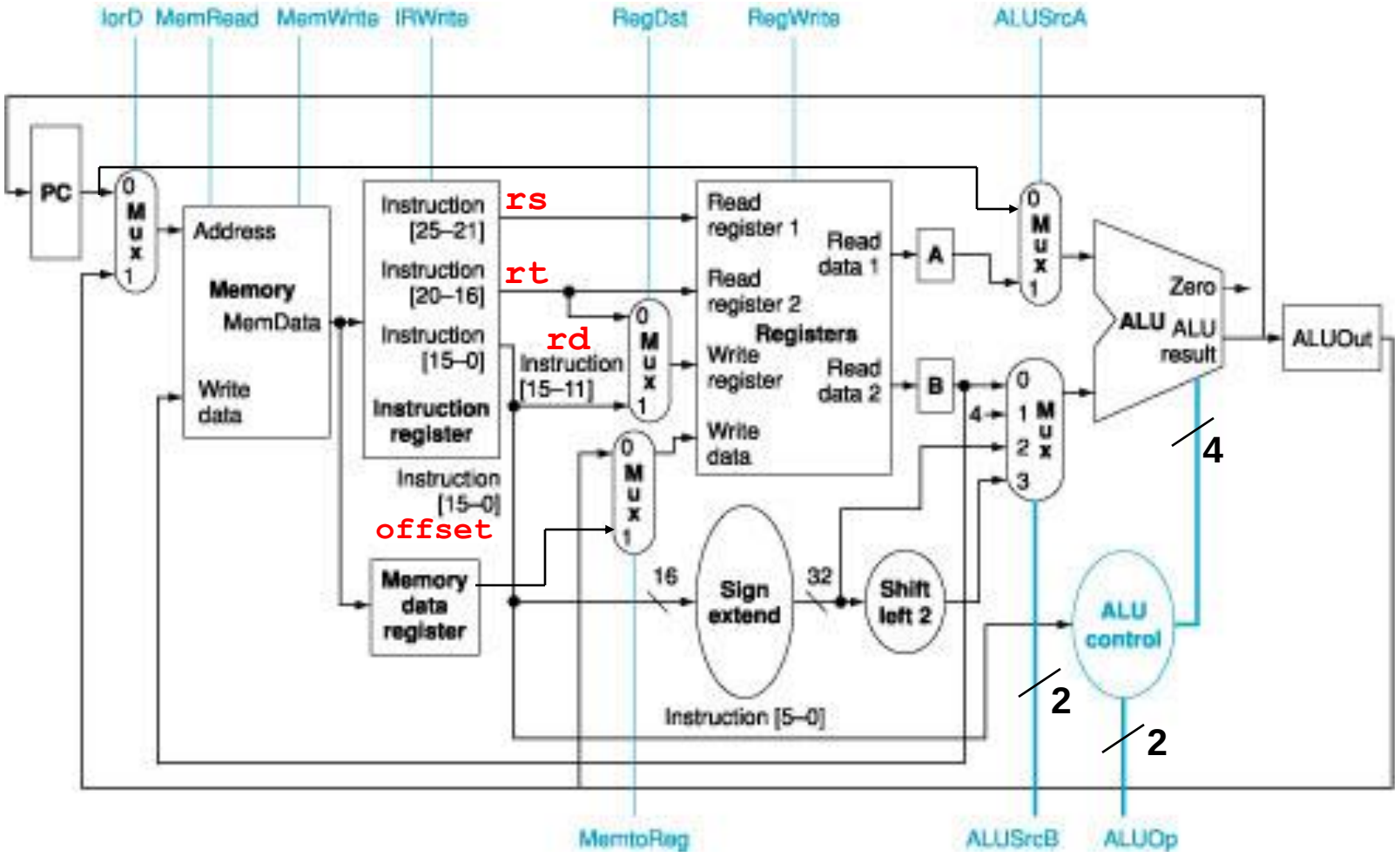
- Το **Opcode** είναι πάντα στη θέση 31:26
- Πάντα διαβάζονται 2 καταχωρητές για R-type/branch/store: **rs** στη θέση 25:21 και **rt** στη θέση 20:16
- Ο καταχωρητής βάσης (*base register*) για load/store είναι στο πεδίο **rs** (25:21):
- Το 16-μπιτο offset για branch/load/store είναι πάντα στη θέση 15:0
- Ο καταχωρητής γραφής ορίζεται στο **rd** στη θέση 15:11 για R-type ή στο **rt** στη 20:16 για load

Σχεδιασμός με πολλαπλούς κύκλους ρολογιού

- Τα σήματα ελέγχου δεν καθορίζονται όλα αμέσως από την εντολή
  - π.χ., η **Μονάδα Ελέγχου ALU** χρειάζεται το πεδίο **FUNCT** της εντολής και το σήμα ελέγχου **ALUOp**
- Θα χρησιμοποιήσουμε **μηχανή πεπερασμένων καταστάσεων** (***finite state machine – FSM*)** για τον σχεδιασμό της μονάδας ελέγχου

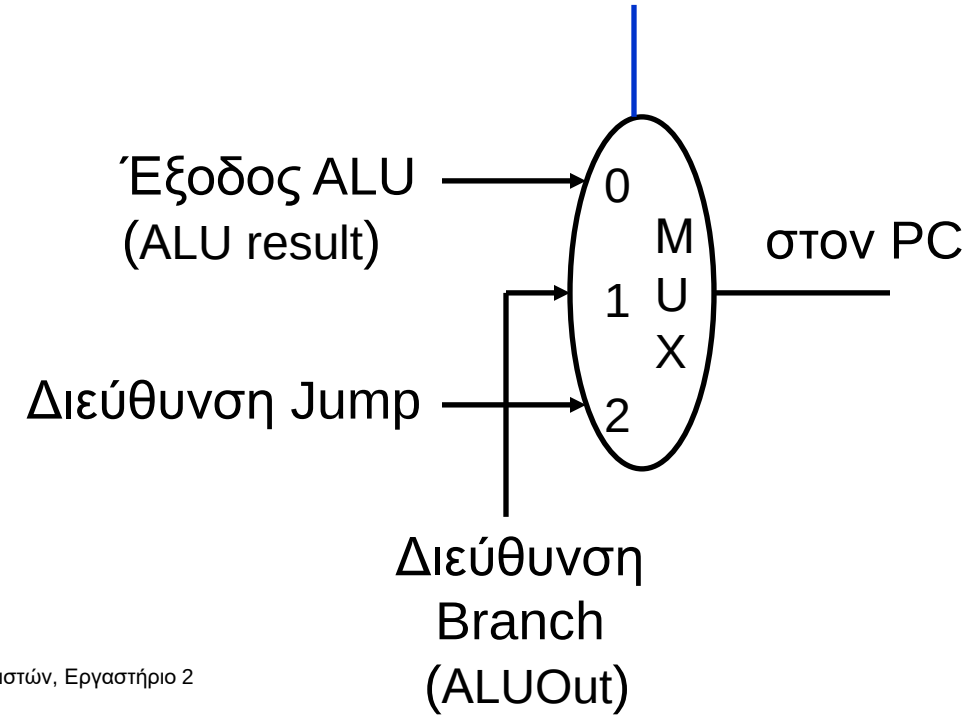
- Μπορούν να χρησιμοποιηθούν τα ίδια σήματα ελέγχου (της κυρίως μονάδας ελέγχου) για τον επεξεργαστή πολλαπλών κύκλων, με αυτό του επεξεργαστή ενός κύκλου; **ΟΧΙ [Χρειάζονται επιπλέον σήματα]**
- Μπορούν τα σήματα ελέγχου του ALU να μείνουν τα ίδια; **ΝΑΙ**

Προσθήκη σημάτων ελέγχου



Εντολές branch και jump

- 3 πιθανές πηγές για την τιμή ανανέωσης του PC
  - Έξοδος από ALU που είναι PC+4, φυλάγεται απευθείας στον PC
  - Έξοδος από ALU που φυλάγεται στον καταχωρητή ALUOut, περιέχει την διεύθυνση διακλάδωσης (**`branch`**)
  - 26 λιγότερο σημαντικά bits του IR, μετατοπισμένα αριστερά κατά 2 και επεκτεινόμενα (:) με τα 4 πιο σημαντικά bits του PC, περιέχει την διεύθυνση άλματος (**`jump`**)



Εντολές branch και jump (συν.)

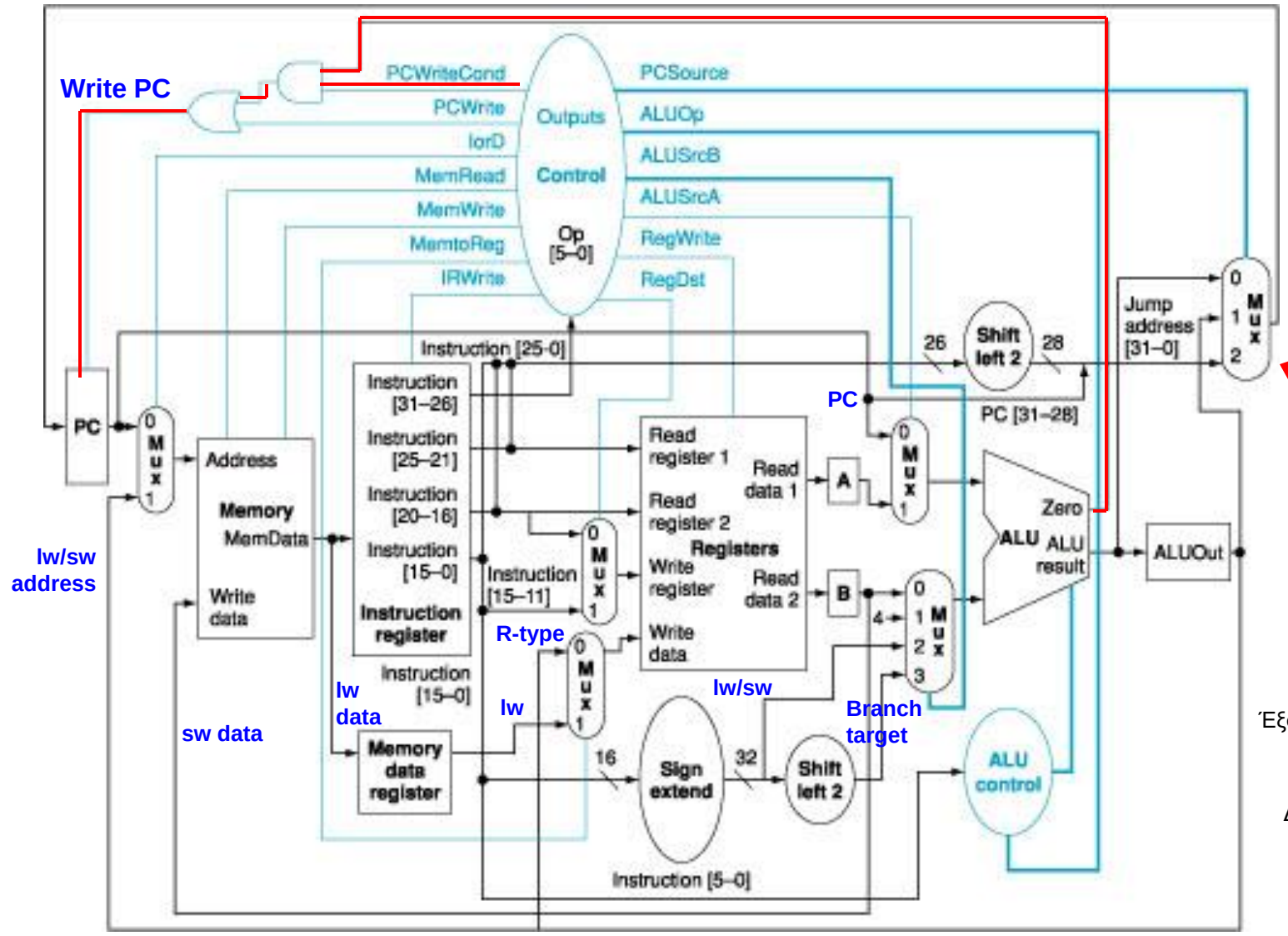
- Πότε γράφουμε στον PC?
- Θυμηθείτε την υλοποίηση ενός κύκλου → Ανεπιφύλακτη (*unconditional*) γραφή για **`jump`** ή αύξηση κατά 4, γραφή υπό συνθήκη (*conditional*) για **`branch`** (Branch=1 *και* ALU Zero=1)

- Θεωρήστε το σήμα ελέγχου PCWrite, που γίνεται 1 όταν έχουμε εντολή **`jump`** ή **`PC+4`**
- Θεωρήστε το σήμα ελέγχου PCWriteCond, που γίνεται 1 όταν έχουμε εντολή **`branch`**. Το PCWriteCond θα προκαλέσει εγγραφή στον PC μόνο όταν **`ALU Zero=1`**
- Τότε,

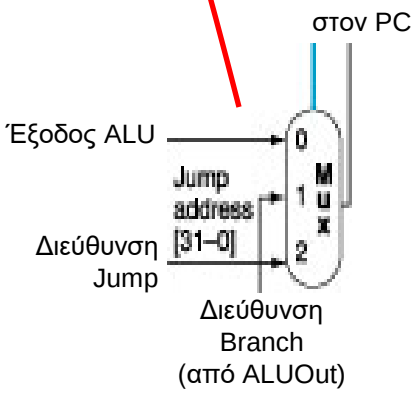
  Write PC = PCWrite ή (PCWriteCond και Zero)

  = PCWrite + PCWriteCond · Zero

Διάδρομος Δεδομένων και απαραίτητα σήματα ελέγχου για πολλαπλούς κύκλους



Πότε (απο) ενεργοποιείται το κάθε σήμα ελέγχου;



Λειτουργία Σημάτων Ελέγχου

Όνομα Σήματος	Αποτέλεσμα όταν Απενεργοποιημένο	Αποτέλεσμα όταν Ενεργοποιημένο
RegDst	Ο καταχωρητής γραφής είναι ο `rt` **Load**	Ο καταχωρητής γραφής είναι ο `rd` **R-type**
RegWrite	Τίποτε	Ο καταχωρητής γραφής ενημερώνεται με τα δεδομένα γραφής
ALUSrcA	1ος τελεστής του ALU είναι ο PC	1ος τελεστής του ALU είναι ο καταχωρητής A
IorD	PC δίνει την διεύθυνση εντολής στη μνήμη	ALUOut δίνει την διεύθυνση δεδομένων στη μνήμη
MemRead	Τίποτε	Ανάγνωση από μνήμη δεδομένων
MemWrite	Τίποτε	Γραφή στη μνήμη δεδομένων
MemtoReg	Δεδομένα καταχωρητή γραφής από τον ALUOut	Δεδομένα καταχωρητή γραφής από τον MDR
IRWrite	Τίποτε	Έξοδος μνήμης γράφεται στον IR
PCWrite	Τίποτε	Γραφή στον PC. Πηγή από PCSource
PCWriteCond	Τίποτε	Γραφή στον PC αν ALU Zero=1

Όνομα Σήματος	Τιμή (Δυαδική)	Αποτέλεσμα
ALUOp	00	Πρόσθεση
	01	Αφαίρεση
	10	*funct* ορίζει την λειτουργία
ALUSrcB	00	2η είσοδος του ALU από τον καταχωρητή B
	01	2η είσοδος του ALU από τον σταθερό 4
	10	2η είσοδος του ALU από τα 16 λιγότερα σημαντικά bits του IR, με επεκτεινόμενο πρόσημο
	11	2η είσοδος του ALU από τα 16 λιγότερα σημαντικά bits του IR, με επεκτεινόμενο πρόσημο και μετατοπισμένα αριστερά κατά 2
PCSource	00	Έξοδος του ALU (PC+4)
	01	ALUOut (διεύθυνση διακλάδωσης)
	10	26 λιγότερο σημαντικά bits του IR, μετατοπισμένα αριστερά κατά 2, επεκτεινόμενα με τα 4 σημαντικότερα bits του PC+4 (διεύθυνση άλματος)

Εντολές βάση του ISA

- Θεωρήστε κάθε εντολή, βάση του ISA.
- Παράδειγμα:
  - Η εντολή **add** αλλάζει τα περιεχόμενα ενός καταχωρητή.
  - Ο καταχωρητής γραφής ορίζεται από τα bits 15:11 της εντολής.
  - Η εντολή καθορίζεται από τον PC.
  - Η νέα τιμή είναι το άθροισμα («op») δύο καταχωρητών.
  - Οι καταχωρητές ανάγνωσης ορίζονται από τα bits 25:21 και 20:16 της εντολής.

```
                          rd
Reg[Instruction[15:11]] <=
Reg[Instruction[25:21]] op Reg[Instruction[20:16]]
                        rs                      rt
```

  - Για να υλοποιήσουμε το πιο πάνω πρέπει να **«σπάσουμε την εντολή»**.
    (παρόμοιο με την εισαγωγή προσωρινών μεταβλητών όταν προγραμματίζουμε)

Τεμαχισμός αριθμητικής εντολής

- Ορισμός ISA για αριθμητική εντολή:

```
                    rd
Reg[Instruction[15:11]] <=
Reg[Instruction[25:21]] op Reg[Instruction[20:16]]
                    rs                          rt
```

- Μπορεί να σπάσει σε:

```
IR <= Memory[PC]
A <= Reg[IR[25:21]]rs
B <= Reg[IR[20:16]]rt
ALUOut <= A op B
Reg[IR[15:11]] <= ALUOut
          rd
```

- Μην ξεχάσετε το ακόλουθο!

```
PC <= PC + 4
```

Η βασική ιδέα της μεθοδολογίας πολλαπλών κύκλων

- Ορίζουμε κάθε εντολή, βάση του ISA
  (επαναλάβετε το προηγούμενο παράδειγμα για άλλες εντολές!)

- Σπάζουμε την εντολή σε βήματα ακολουθώντας τον κανόνα ότι ***τα δεδομένα ρέουν διαμέσου το πολύ μίας λειτουργικής μονάδας ανά κύκλο*** (ισορροπία μεταξύ των διαφόρων βημάτων)

- Εισαγωγή νέων καταχωρητών, όπου χρειάζεται (π.χ., A, B, ALUOut, MDR)

- Προσπαθούμε να «γεμίσουμε» το κάθε βήμα με όσο το δυνατόν περισσότερη εργασία (για να αποφύγουμε αχρείαστους κύκλους) και την ίδια στιγμή, να «μοιράζονται» οι εντολές όσο το δυνατό περισσότερα βήματα (ελαχιστοποιεί τη μονάδα ελέγχου, απλοποιεί την υλοποίηση)

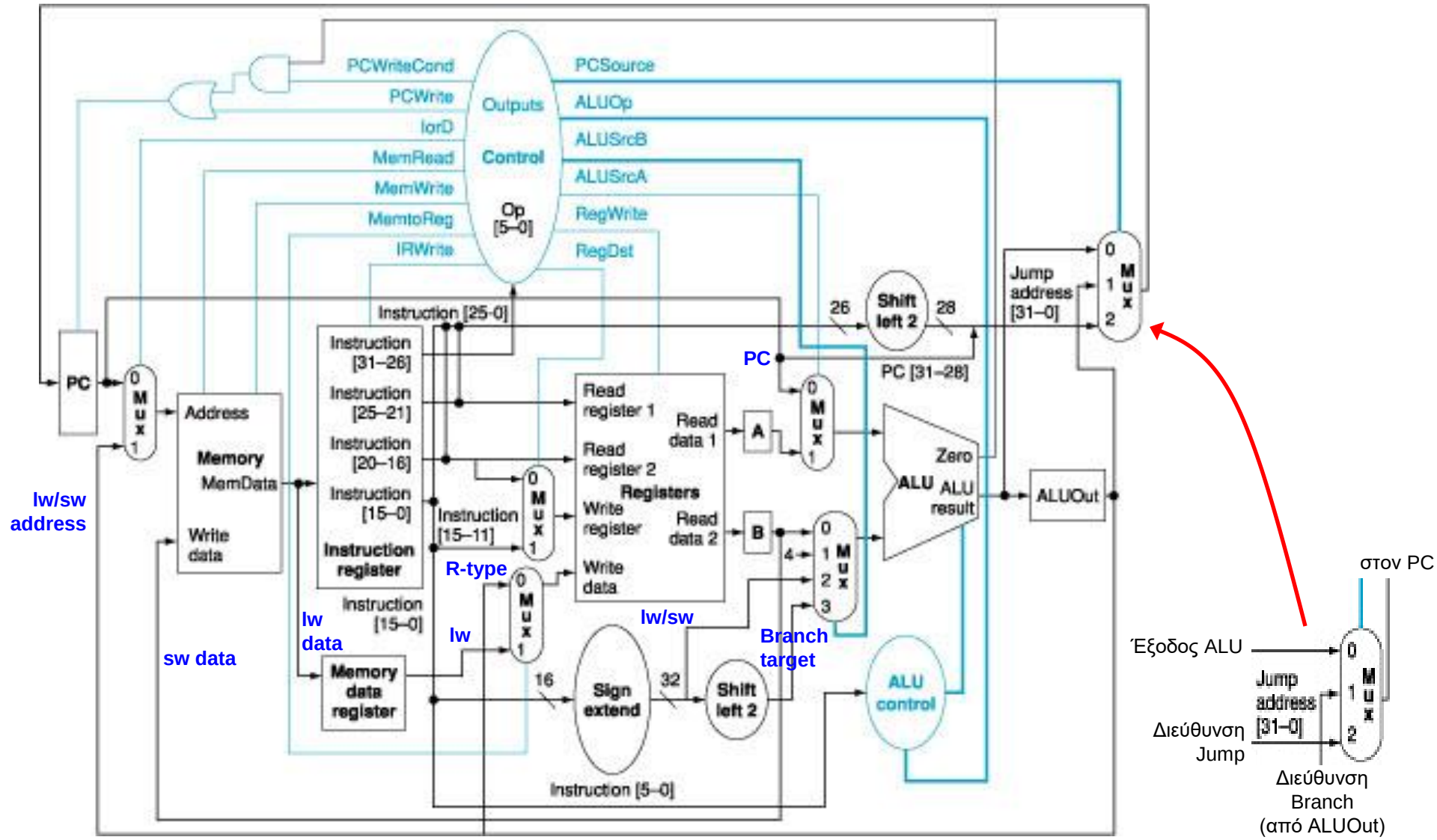
Τεμαχισμός εντολής σε 5 βήματα εκτέλεσης

1. Προσκόμιση Εντολής **[Instruction Fetch, IF]**
2. Αποκωδικοποίηση Εντολής και Προσκόμιση Καταχωρητών **[Instruction Decode, ID]**
3. Εκτέλεση, Υπολογισμός Διεύθυνσης Μνήμης ή Ολοκλήρωση Διακλάδωσης **[Execute, EX]**
4. Προσπέλαση Μνήμης ή Ολοκλήρωση εντολής τύπου R **[Memory Access, MEM]**
5. Αποτέλεσμα σε καταχωρητή (***write-back***) **[Write-back, WB]**

Τα βήματα 1-2 εκτελούνται για ΟΛΕΣ τις εντολές.

ΟΛΕΣ ΟΙ ΕΝΤΟΛΕΣ ΕΚΤΕΛΟΥΝΤΑΙ ΜΕΤΑΞΥ 3ων - 5 ΚΥΚΛΩΝ!

Διάδρομος Δεδομένων και απαραίτητα σήματα ελέγχου για πολλαπλούς κύκλους



Βήμα 1: Προσκόμιση εντολής

- Χρησιμοποιούμε τον PC για να πάρουμε την εντολή από την μνήμη και να την αποθηκεύσουμε στον Instruction Register (IR).
- Αυξάνουμε τον PC κατά 4 και φυλάγουμε το αποτέλεσμα πίσω στον PC.

- Μπορεί να περιγραφεί χρησιμοποιώντας RTL (Register-Transfer Language)

```
IR <= Memory[PC];
PC <= PC + 4;
```

Μπορείτε να υπολογίσετε τις τιμές των σημάτων ελέγχου;

`MemRead=1`, `IRWrite=1`, `IorD=0` (i.e., select PC as source), `ALUSrcA=0` (PC to ALU), `ALUSrcB=01` (4 to ALU), `ALUOp=00` (add), `PCSource=00` (PC+4), `PCWrite=1`

Ποιο είναι το πλεονέκτημα από το να ανανεώνουμε τον PC σε αυτό το στάδιο; [Ready to compute Branch Target Address in next cycle]

Βήμα 2: Αποκωδικοποίηση Εντολής και Προσκόμιση Καταχωρητών

- Ανάγνωση καταχωρητών **rs** και **rt**
- Υπολογισμός της διεύθυνσης διακλάδωσης (σε περίπτωση που η εντολή είναι **branch**)

- RTL:

```
                    rs
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]]; rt
ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
                                  Offset * 4
```

- Σε αυτόν τον κύκλο, η Κεντρική Μονάδα Ελέγχου προσπαθεί να καταλάβει τον τύπο της εντολής (ασχολείται με την «αποκωδικοποίηση» της εντολής).
- **ALUSrcA = 0** (PC to ALU)

  ALUSrcB = 11 (sign-extended and shifted offset to ALU)

  ALUOp = 00 (add)

Βήμα 3: Εξαρτάται από τον τύπο της εντολής

- Το ALU εκτελεί μία από τις πιο κάτω πράξεις, *βάση του τύπου της εντολής*
- Πρόσβαση Μνήμης:

```
ALUOut <= A + sign-extend(IR[15:0]);
                          Offset
```

- R-type:

```
ALUOut <= A op B;
```

- Ολοκλήρωση Branch:

```
if (A==B) PC <= ALUOut;
```

- Ολοκλήρωση Jump:

```
PC <= PC[31:28] : (IR[25:0] << 2);
                      Offset * 4
```

Βήμα 4: R-type ή πρόσβαση στη μνήμη

- Προσπέλαση μνήμης για load ή store

```
MDR <= Memory[ALUOut];    [Load]
       or
Memory[ALUOut] <= B;      [Store]
```

- Ολοκλήρωση εντολής R-type

```
Reg[IR[15:11]] <= ALUOut;
       rd
```

Η γραφή εκτελείται στο τέλος του κύκλου, πάνω στην ακμή

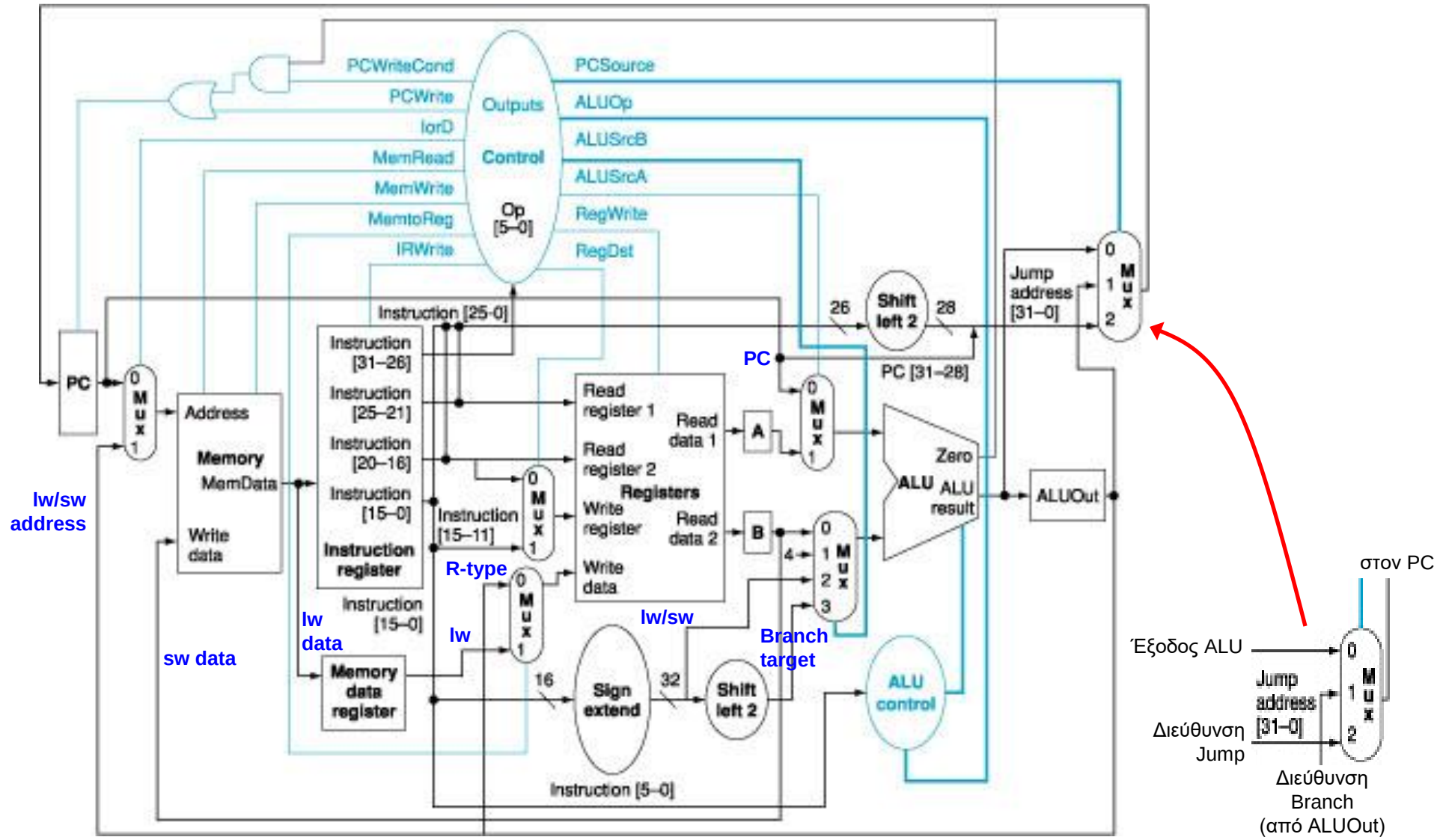
Βήμα 5: Write-back

- `Reg[IR[20:16]] <= MDR;`
  `rt`

Ποια εντολή χρειάζεται το πιο πάνω;

[Ολοκλήρωση της εντολής **Load** – αντιγραφή των δεδομένων από τον προσωρινό καταχωρητή MDR στον κατάλληλο καταχωρητή μέσα στο αρχείο καταχωρητών, Register File]

Διάδρομος Δεδομένων και απαραίτητα σήματα ελέγχου για πολλαπλούς κύκλους



Συνοπτικά

4 κύκλοι | **Load: 5 κύκλοι** / **Store: 4 κύκλοι** | **3 κύκλοι** | **3 κύκλοι**

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]),2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

Ερώτηση:

- Πόσοι κύκλοι απαιτούνται για να εκτελεστεί ο πιο κάτω κώδικας;

```
            lw $t2, 0($t3)
            lw $t3, 4($t3)
            beq $t2, $t3, Label #θεωρήστε ότι δεν ισούνται
            add $t5, $t2, $t3
            sw $t5, 8($t3)
Label:      ...
```

5 (Load) + 5 (Load) + 3 (Branch) + 4 (R-type) + 4 (Store) = 21 κύκλοι

- Τι συμβαίνει κατά τη διάρκεια του 8ου κύκλου; 3ος κύκλος του δεύτερου Load, άρα **`ALUOut <= A + sign-extend(IR[15:0])`**
- Σε πιο κύκλο γίνεται η πρόσθεση των **`$t2`** και **`$t3`**; Στον 3ον κύκλο της εντολής **`add`**, άρα στον 5+5+3+3=16ον κύκλο

Μονάδα Ελέγχου για πολλαπλούς κύκλους ρολογιού

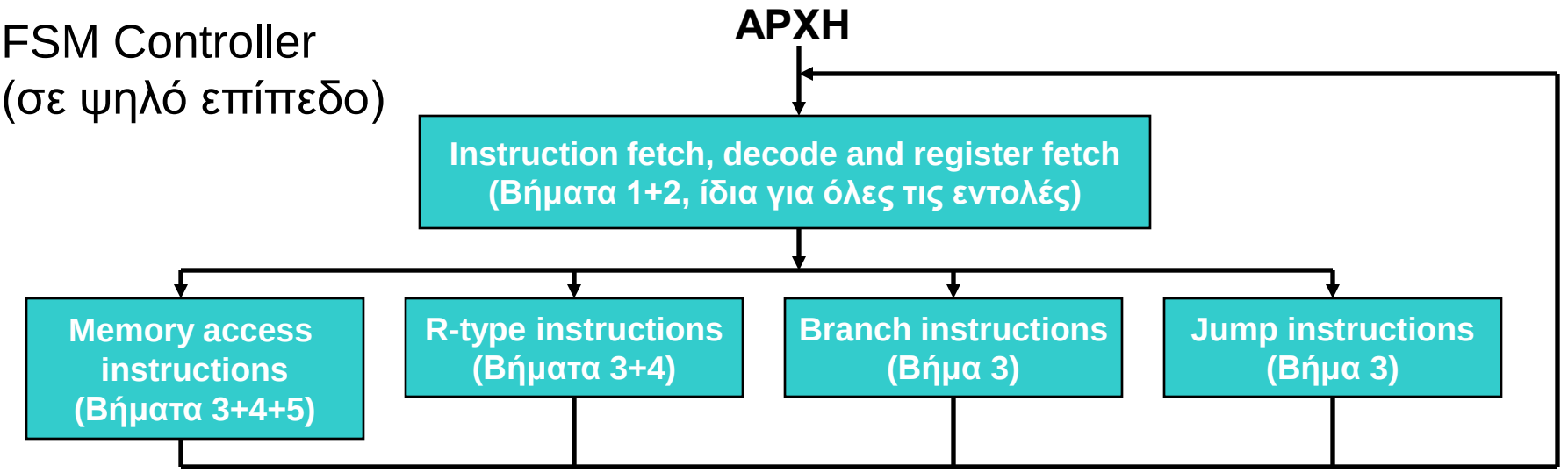
- Η μονάδα ελέγχου για το διάδρομο δεδομένων πολλαπλών κύκλων πρέπει να καθορίζει:
  - Τις τιμές των σημάτων ελέγχου
  - Επόμενο βήμα εκτέλεσης

- 2 τρόποι σχεδιασμού της μονάδας ελέγχου:
  - Ως μηχανή πεπερασμένων καταστάσεων (***FSM***)
    - Γραφική αναπαράσταση που οδηγεί στον πίνακα καταστάσεων → πίνακα αληθείας (βάση των εισόδων/εξόδων) → λογικό διάγραμμα
  - Με χρήση Μικρο-προγραμματισμού (***Microprogramming***)
    - Αναπαράσταση της μονάδας ελέγχου με προγραμματισμό

Υλοποίησης Μονάδας Ελέγχου για πολλαπλούς κύκλους ρολογιού (συν.)

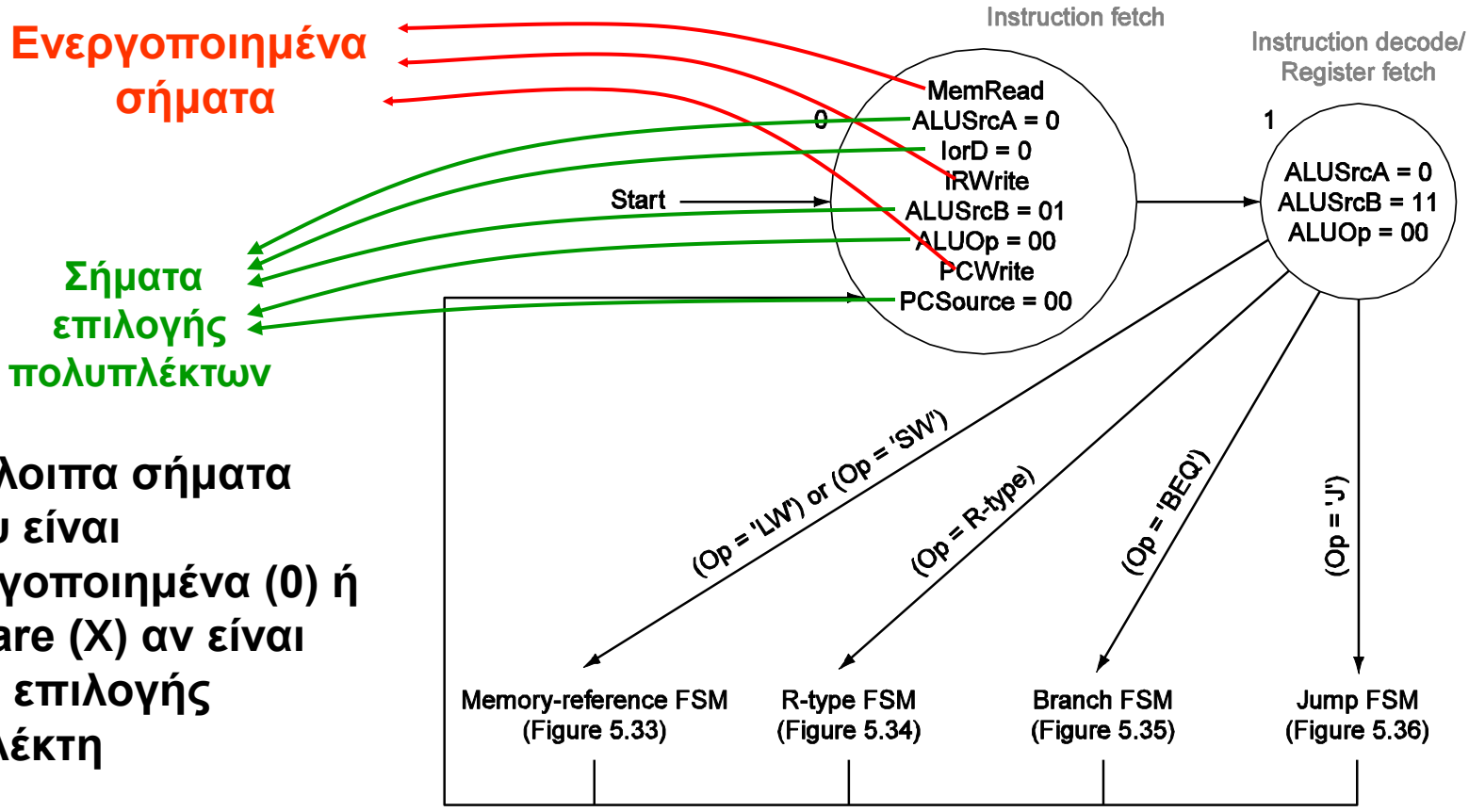
Step name (Αρ. Βήματος)	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch (1)	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch (2)	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion (3)	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion (4)	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion (5)		Load: Reg[IR[20:16]] <= MDR		

FSM Controller
(σε ψηλό επίπεδο)



Παράδειγμα: Βήμα 1+2: Instruction Fetch and Decode

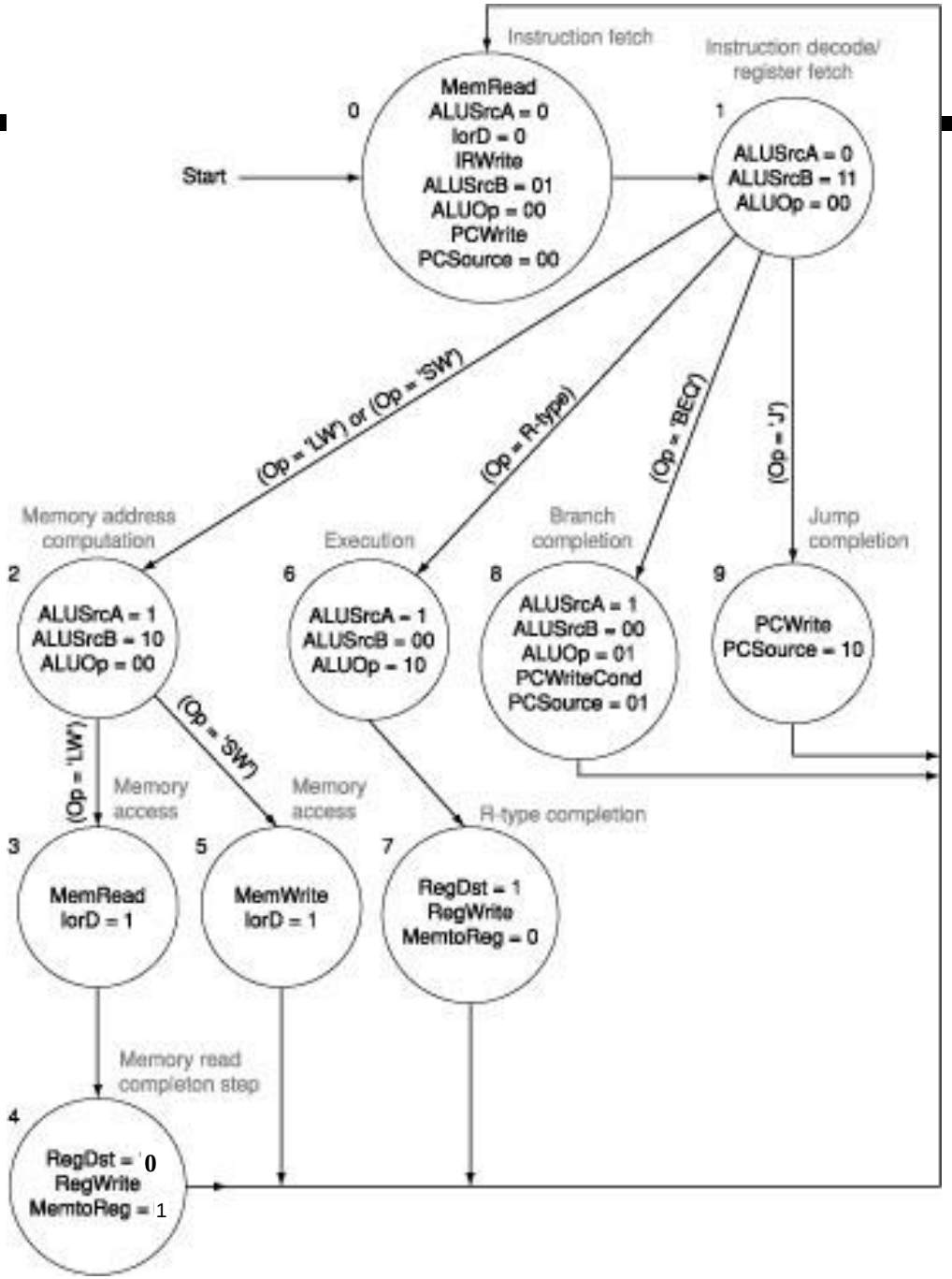
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			



Τα υπόλοιπα σήματα ελέγχου είναι απενεργοποιημένα (0) ή don't care (X) αν είναι σήματα επιλογής πολυπλέκτη

Το τελικό FSM

- Συνολικά:
  10 καταστάσεις
  (**states**)



Το τελικό FSM

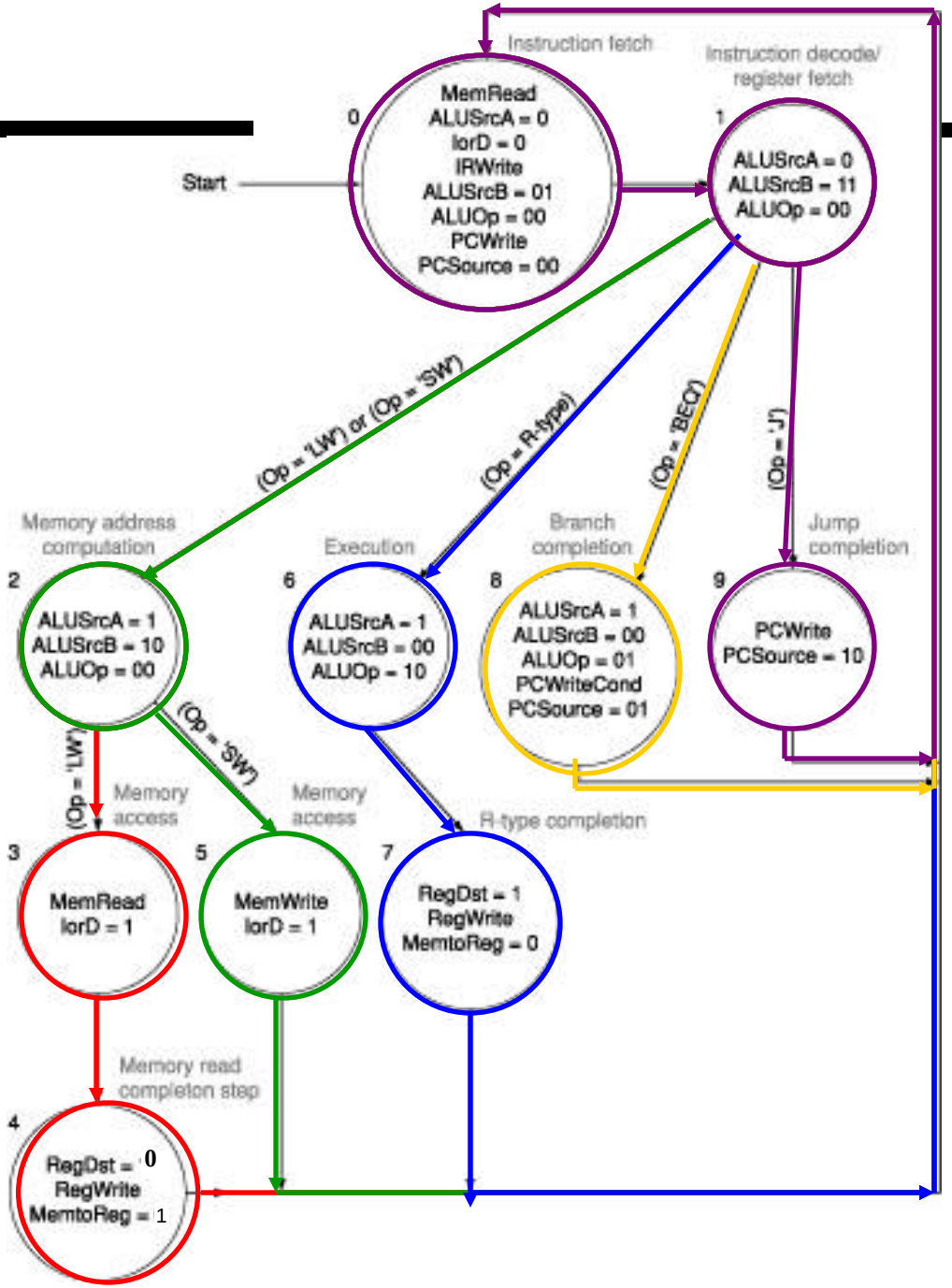
Load

Store

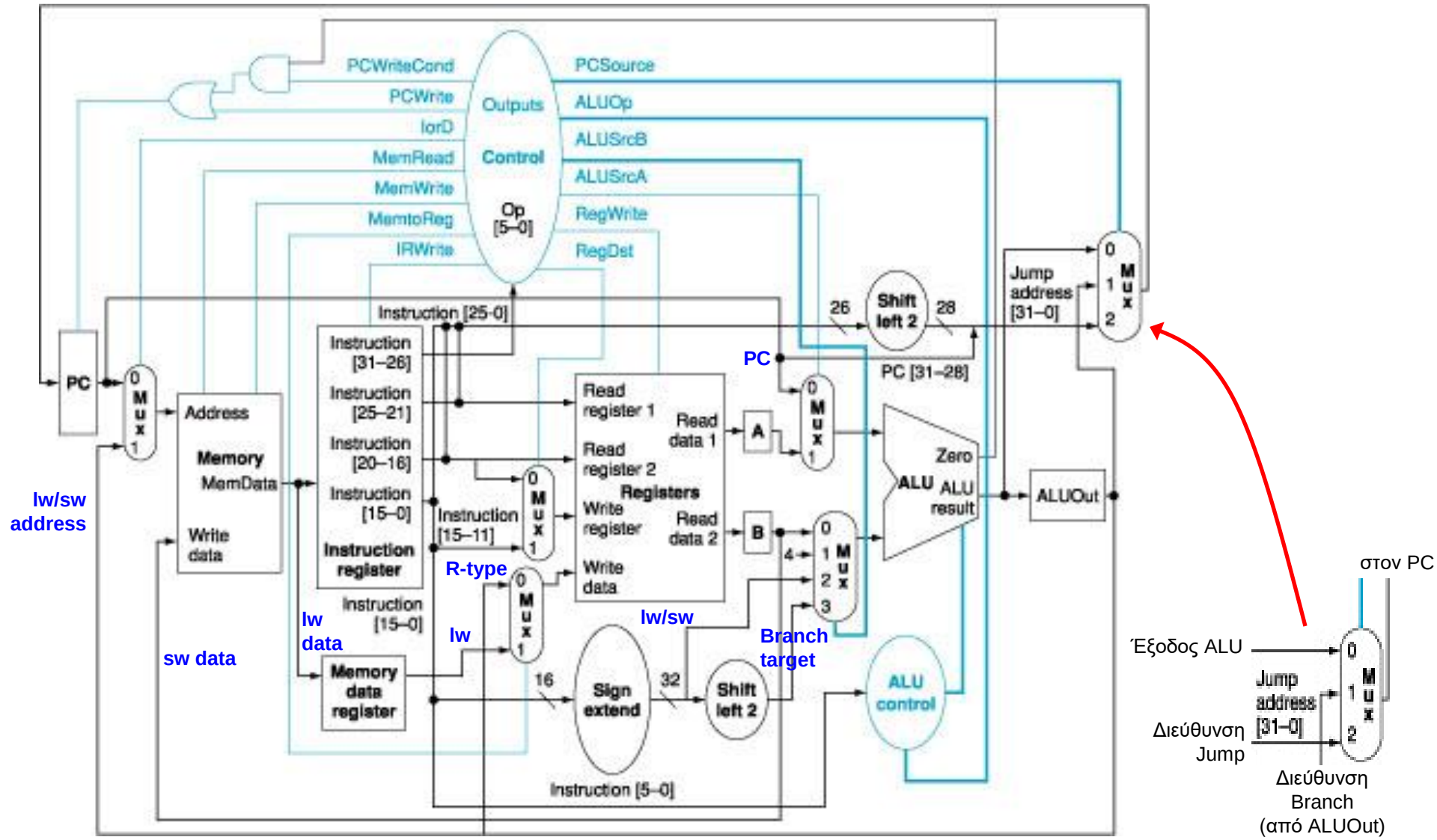
R-type

Branch

Jump



Διάδρομος Δεδομένων και απαραίτητα σήματα ελέγχου για πολλαπλούς κύκλους



Σύνοψη: Εργαστηριακή Άσκηση 2