ΕΡΓΑΣΤΗΡΙΟ ΑΡΧΙΤΕΚΤΟΝΙΚΗ Η/Υ

4Η ΕΡΓΑΣΙΑ ΕΡΓΑΣΤΗΡΙΟΥ

Intel Quartus Prime

Ορφέας- Άγγελος Νικολάου

ΑΜ: 2792

E-MAIL: int02792@uoi.gr

Άρτα, 2024 / 01 / 16

Table of contents

Πληροφορίες

Περίληψη

Σε αυτή την εργασία ζητιέται η υλοποίηση ενός μέρος επεξεργαστή 32bit MIPS στην γλώσσα VHDL στο λογισμικό Quartus.

Από τι αποτελείται ένα MIPS;

Ένας μικροεπεξεργαστής αποτελείται από τα εξής βασικά στοιχεία:

1. ALU (arithmetic and logic unit → αριθμητική και λογική μονάδα):

   Εκτελεί όλες τις απαραίτητες λειτουργίες που απαιτούνται από το set εντολών. Είναι επίσης υπεύθυνη για την υλοποίηση συνθηκών διακλαδώσεων.

2. Αρχείο καταχωρητών:

   Το οποίο περιέχει 32 καταχωρητές με δύο θύρες ανάγνωσης και μία θύρα εγγραφής.

3. IR (instruction register → καταχωρητής εντολών):

   Αποθηκεύει την εντολή που μόλις προσφέρθηκε από την μνήμη.

4. PC (program counter → αριθμητής προγράμματος):

   Περιέχει την διεύθυνση μνήμης της επόμενης εντολής.

5. Control unit (μονάδα ελέγχου):

   ελέγχει το datapath και τη μνήμη.

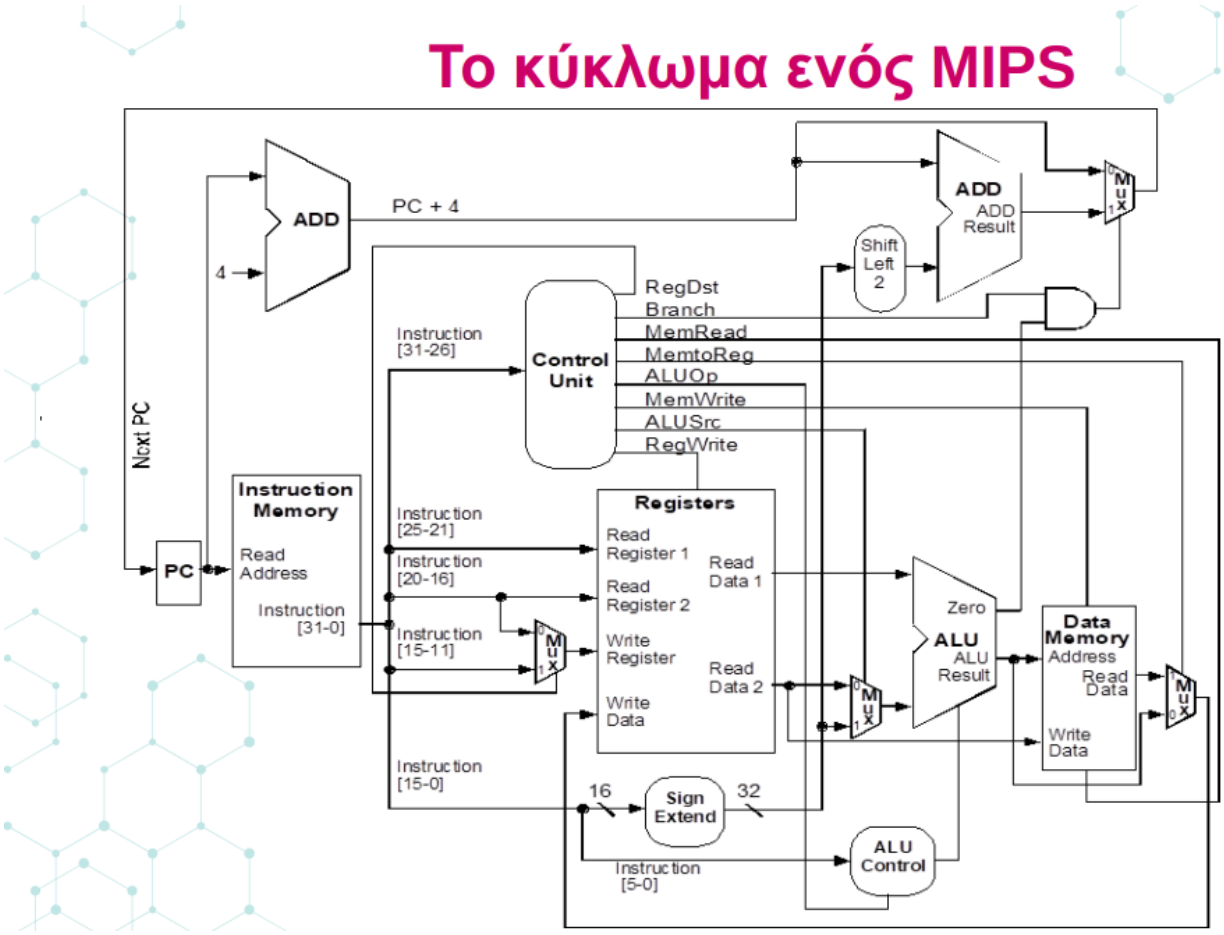
6. ALU control (μονάδα ελέγχου της ALU):

   Ελέγχει όλες τις λειτουργίες της ALU.

7. Sign extend mechanism (μηχανισμός επέκτασης – προσήμανσης):

   16bit → 32bit σήμα.

Κύκλωμα ενός MIPS



Στόχος της εργασίας

Ο στόχος της εργασίας είναι να υλοποιήσουμε (αν και όχι πλήρες) την μονάδα ελέγχου (CONTROL UNIT) και την αριθμητική και λογική μονάδα (ALU).

Μονάδα ελέγχου (CONTROL UNIT)

Για να υλοποιήσουμε την μονάδα ελέγχου χρειαζόμαστε τους εξής πίνακες (δεδομένους από την εκφώνηση):

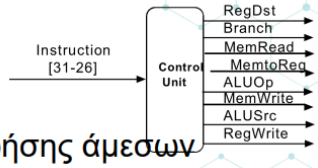
ALUOp[1]	ALUOp[0]	Instruction
0	0	memory operations (`load, store`)
0	1	`beq`
1	0	r-type operations

Name	Op-code					
	Op5	Op4	Op3	Op2	Op1	Op0
R-format	0	0	0	0	0	0
`lw`	1	0	0	0	1	1
`sw`	1	0	1	0	1	1
`beq`	0	0	0	1	0	0
`j`	0	0	0	0	1	0

Όπου Opcode = πρώτα 6bit μιας εντολής.

θα χρειαστεί και η εξής διαφάνεια για να υλοποιήσω όλες τις εξόδους ώστε να βγαίνουν οι σωστές τιμές.

Τα σήματα που απαιτούνται για τον έλεγχο της διαδρομής δεδομένων είναι τα ακόλουθα:



- ✓ **Jump** — ορίστε στο 1 για μια εντολή άλματος
- ✓ **Branch** — ορίστε σε 1 για μια εντολή διακλάδωσης
- ✓ **MemtoReg** — ορίστε σε 1 για μια εντολή φόρτωσης
- ✓ **ALUSrc** — ορίστε σε 0 για εντολές τύπου r και 1 για εντολές χρήσης άμεσων δεδομένων στο ALU (το beq απαιτεί αυτό το σύνολο σε 0)
- ✓ **RegDst** — ορίστε σε 1 για εντολές τύπου r και 0 για άμεση εντολή
- ✓ **MemRead** — ορίστε σε 1 για εντολές φόρτωσης
- ✓ **MemWrite** — ορίστε σε 1 για εντολές αποθήκευσης
- ✓ **RegWrite** - ορίστε σε 1 για οποιαδήποτε εντολή εγγραφής σε έναν καταχωρητή
- ✓ **ALUOp** (k bit) — κωδικοποιεί εντολές ALU εκτός από τον τύπο r εντολές, οι οποίες κωδικοποιούνται από το πεδίο συνάρτησης

Και με τους εξής πίνακες επιβεβαιώνω τις κυματομορφές (υπενθύμιση ότι X στον πίνακα σημαίνει ότι δεν μας νοιάζει τι θα βγει σε αυτή την έξοδο).

Instruction	Op-code	RegDst	ALUSrc	MemtoReg	Reg Write
r-type	0 0 0 0 0 0	1	0	0	1
lw	1 0 0 0 1 1	0	1	1	1
sw	1 0 1 0 1 1	x	1	x	0
beq	0 0 0 1 0 0	x	0	x	0
j	0 0 0 0 1 0	x	x	x	0

Instruction	Op-code	Mem Read	Mem Write	Branch	ALUOp[1:0]	Jump
r-type	0 0 0 0 0 0	0	0	0	1 0	0
lw	1 0 0 0 1 1	1	0	0	0 0	0
sw	1 0 1 0 1 1	0	1	0	0 0	0
beq	0 0 0 1 0 0	0	0	1	0 1	0
j	0 0 0 0 1 0	0	0	0	x x	1

Κώδικας

[GitHub mirror]

[Pastebin mirror] Password: 12341

Φυσικά, υπάρχει και μέσα στα αρχεία του Quartus.

```
1     library ieee;
2     use ieee.std_logic_1164.all;
3     use ieee.std_logic_arith.all;
4     use ieee.std_logic_signed.all;
5     entity control is
6     port ( Opcode : in std_logic_vector (5 downto 0);
7          RegDst : out std_logic;
8          ALUSrc : out std_logic;
9          MemtoReg : out std_logic;
10         RegWrite : out std_logic;
11         MemRead : out std_logic;
12         MemWrite : out std_logic;
13         Branch : out std_logic;
14         ALUop : out std_logic_vector (1 downto 0);
15         clock, reset : in std_logic );
16    end control;
17
18    architecture behavior of control is
19    signal R_format, Lw, Sw, Beq, Ji : std_logic;
20    begin
21    R_format <= '1' when Opcode = "000000" else '0';
22    Lw       <= '1' when Opcode = "100011" else '0';
23    Sw       <= '1' when Opcode = "101011" else '0';
24    Beq    <= '1' when Opcode = "000100" else '0';
25    Ji       <= '1' when Opcode = "000010" else '0';
26    RegDst <= R_format;
27    ALUSrc <= not R_format and not Beq;
28    MemtoReg <= Lw;
29    RegWrite <= R_format or Lw;
30    MemRead <= Lw;
31    MemWrite <= Sw;
32    Branch <= Beq; -- oxi Beq or Ji
33    ALUOp(1) <= R_format;
34    ALUOp(0) <= Beq;
35    end behavior;
36
```

Διάγραμμα

Σημείωση ότι για I/O με χρήση πάνω από ενός bit χρειάζεται το παχύ data bus το οποίο το καλώδιο το τραβάμε από το ίδιο το σύμβολο. Πέρα από αυτό, χρειάζεται και στα ονόματα να έχει στο τέλος [upper_range..lower_range], αλλιώς πετάει λάθος μεταγλώττισης.

Είναι επίσης καλή πρακτική να βάζουμε το ίδιο όνομα με την Ε/Ε επειδή έτσι θα ξέρουμε για τα nodes των κυματομορφών.

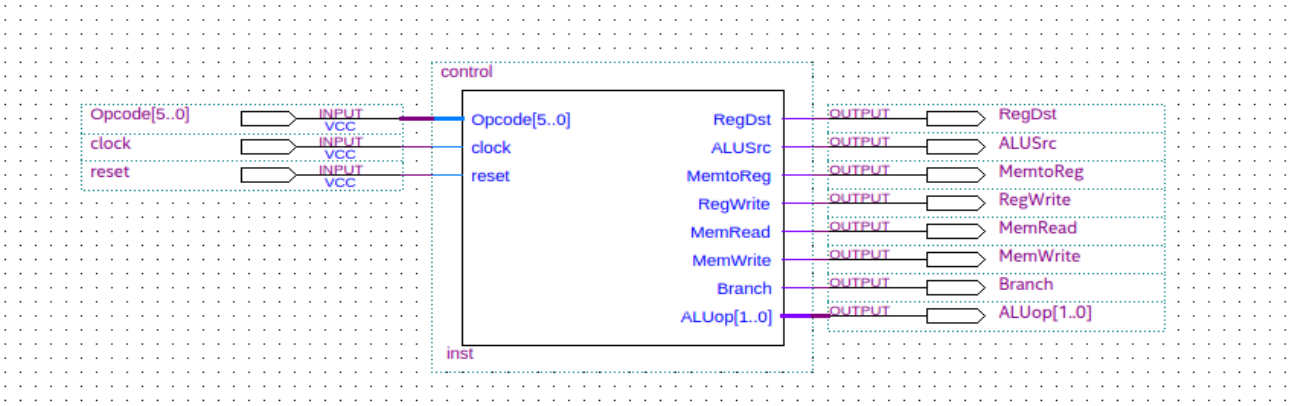


Figure 1: block_diagram_control.png

Κυματομορφές

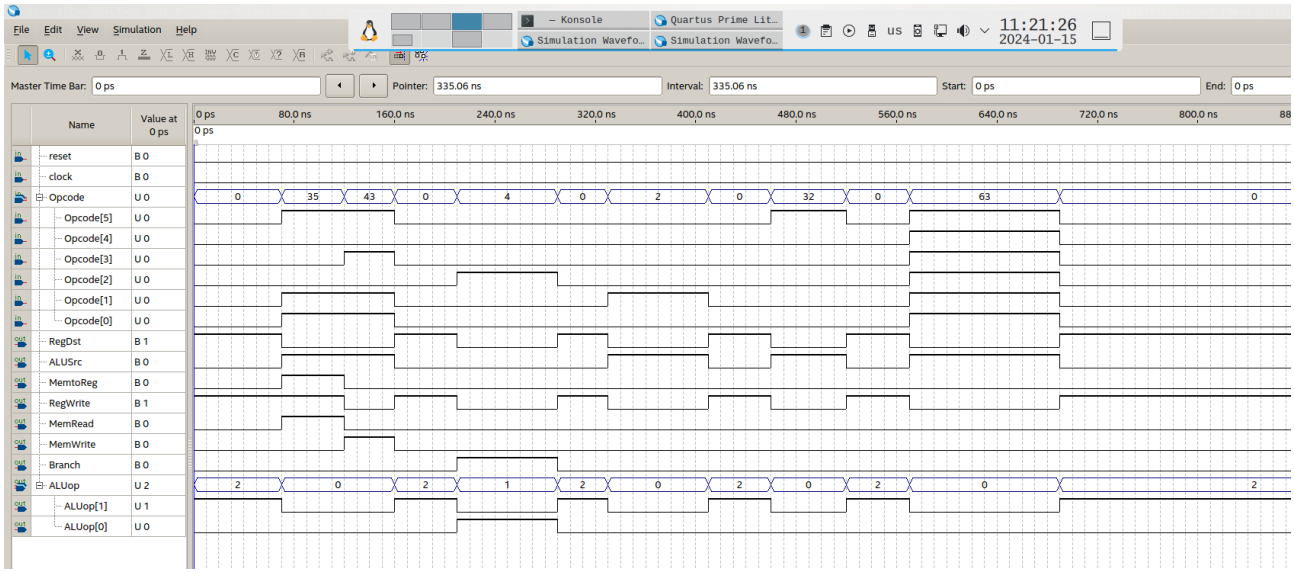


Figure 2: CONTROL.png

Διαπίστωσα ότι οι πίνακες πιο πάνω επαληθεύουν τις κυματομορφές.

Opcodes: R format → lw → sw → R format → beq → R format → j → R format → κάτι άσχετα με την άσκηση.

Για ευκολία οι έξοδοι είναι με την σειρά από τους πίνακες.

Σημείωση ότι από τον κώδικα έλειπε η έξοδος Jump, η οποία θα ήταν 0 σε όλες τις εντολές του πίνακα εκτός από την εντολή j.

Instruction	Op-code	RegDst	ALUSrc	MemtoReg	Reg Write
r-type	0 0 0 0 0 0	1	0	0	1
lw	1 0 0 0 1 1	0	1	1	1
sw	1 0 1 0 1 1	x	1	x	0
beq	0 0 0 1 0 0	x	0	x	0
j	0 0 0 0 1 0	x	x	x	0

Instruction	Op-code	Mem Read	Mem Write	Branch	ALUOp[1:0]	Jump
r-type	0 0 0 0 0 0	0	0	0	1 0	0
lw	1 0 0 0 1 1	1	0	0	0 0	0
sw	1 0 1 0 1 1	0	1	0	0 0	0
beq	0 0 0 1 0 0	0	0	1	0 1	0
j	0 0 0 0 1 0	0	0	0	x x	1

Αριθμητική και λογική μονάδα (ALU)

Οι είσοδοι στο στοιχείο ελέγχου ALU είναι τα σήματα ελέγχου ALUOp και το πεδίο συνάρτησης (funct) 6 bit.

Όταν το πεδίο ALUOp == “10” τότε “ενεργοποιείται” το πεδίο συνάρτησης (αφού έτσι είναι εντολές τύπου R)

Για να υλοποιήσουμε την ALU χρειαζόμαστε τους εξής πίνακες (δεδομένους από την εκφώνηση):

Instruction opcode	ALUOp	Instruction Operation	Funct Field	Desired ALU Action	ALU Control Input
LW	00	Load word	XXXXXX	Add	010
SW	00	Store word	XXXXXX	Add	010
Branch equal	01	Branch equal	XXXXXX	Subtract	110
R type	10	Add (+)	100000	Add	010
R type	10	Subtract (-)	100010	Subtract	110
R type	10	AND	100100	And	000
R type	10	OR	100101	Or	001
R type	10	XOR	101001	Xor	011
R type	10	Set on less than	101010	Set on less than	111

Παρατηρούμε ότι σε εντολές όπως lw και sw η επιθυμητή πράξη είναι εντελώς άσχετη με το τι κάνουν αυτές οι εντολές. Αυτό γίνεται διότι σε άλλα σημεία (που δεν υλοποιούμε σε αυτή την εργασία) των MIPS ακολουθεί η διεργασία που χρειάζεται.

Κώδικας

[GitHub mirror]

[Pastebin mirror] Password: 12341

Φυσικά, υπάρχει και μέσα στα αρχεία του Quartus.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_arith.all;
4   use ieee.std_logic_signed.all;
5
6   entity ALU is
7   port ( Read_data_1 : in std_logic_vector (31 downto 0);
8        Read_data_2 : in std_logic_vector (31 downto 0);
9        Sign_extend : in std_logic_vector (31 downto 0);
10       Function_opcode : in std_logic_vector (5 downto 0);
11       ALUOp : in std_logic_vector (1 downto 0);
12       ALUSrc : in std_logic;
13       Zero : out std_logic;
14       ALU_Result: out std_logic_vector (31 downto 0);
15       Add_Result : out std_logic_vector(7 downto 0);
16       PC_plus_4 : in std_logic_vector (7 downto 0);
17       clock, reset : in std_logic );
18  end ALU;
19  architecture behavior of ALU is
20  signal Ainput, Binput : std_logic_vector (31 downto 0);
21  signal ALU_output_mux : std_logic_vector (31 downto 0);
22  signal Branch_Add : std_logic_vector (7 downto 0);
23  signal ALU_ctl : std_logic_vector (2 downto 0);
24  begin
25  Ainput <= Read_data_1;
26        -- ΑΚΥ input max // δεν καταλαβα αυτο το σχολιο
27  Binput <= Read_data_2 when (ALUSrc = '0') else Sign_extend (31 downto 0);
28
29  -- generate ALU control bits
30  ALU_ctl(0) <= (Function_opcode(0) or Function_opcode(3)) and ALUOp(1);
31  ALU_ctl(1) <= (not Function_opcode(2)) or (not ALUOp(1));
32  ALU_ctl(2) <= (Function_opcode(1) and ALUOp(1)) or ALUOp(0);
33
34  -- generate zero flag
35  Zero <= '1' when ((ALU_output_mux) = X"00000000") else '0';
36  --select ALU output for SLT
37  ALU_result <= X"0000000" & B"000" & ALU_output_mux(31) when ALU_ctl = "111" else ALU_output_mux(31 downto 0);
38  --adder to compute branch adress
39  Branch_Add <= PC_plus_4(7 downto 2) + Sign_extend(7 downto 0);
40  Add_result <= Branch_Add (7 downto 0);
41  process (ALU_ctl, Ainput, Binput)
42  begin
43  case ALU_ctl is
44  when "000" => -- x and y
```

```
45      ALU_output_mux <= Ainput and Binput;
46    when "001" => -- x or y
47      ALU_output_mux <= Ainput or Binput;
48    when "010" =>  --add desired alu action
49        ALU_output_mux <= Ainput + Binput;
50    when "011" =>
51      ALU_output_mux <= Ainput xor Binput;
52    when "100" => -- undef
53        ALU_output_mux <= X"00000000";
54    when "101" => -- undef
55        ALU_output_mux <= X"00000000";
56    when "110" => --sub desired alu action
57        ALU_output_mux <= Ainput - Binput;
58    when "111" => -- set on less than
59        ALU_output_mux <= Ainput - Binput;
60    end case;
61    end process;
62    end behavior;
```

Διάγραμμα

Εννοείται πως το σύμβολο βγήκε αφού κάνω compile των κώδικα και κάνω create/update symbol κτλ κτλ.

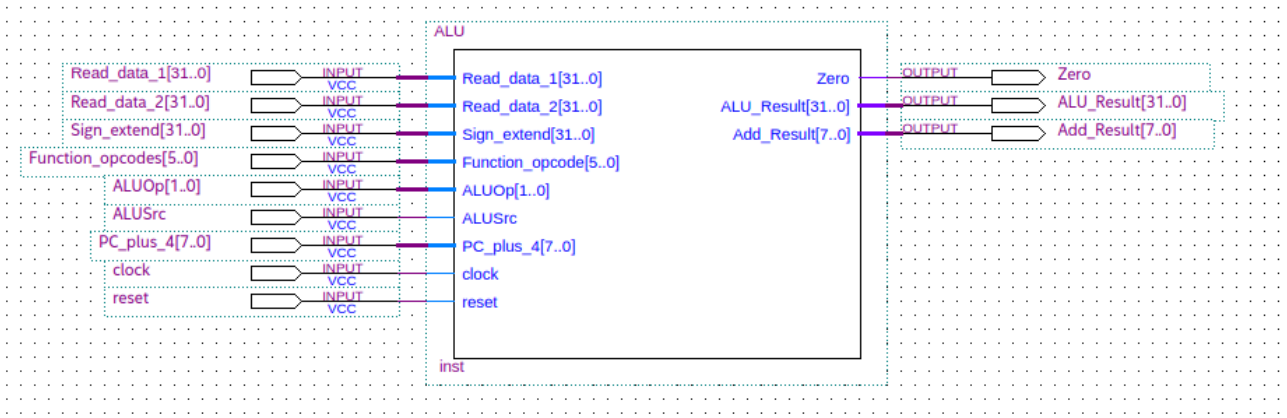


Figure 3: block_diagram_ALU.png

Κυματομορφές

Loadword Storeword

ALUOp == “00” FunField == “XXXXXX”

Desired ALU action → Add.

Read_data_1 (63) + Read_data_2 (3) == 66

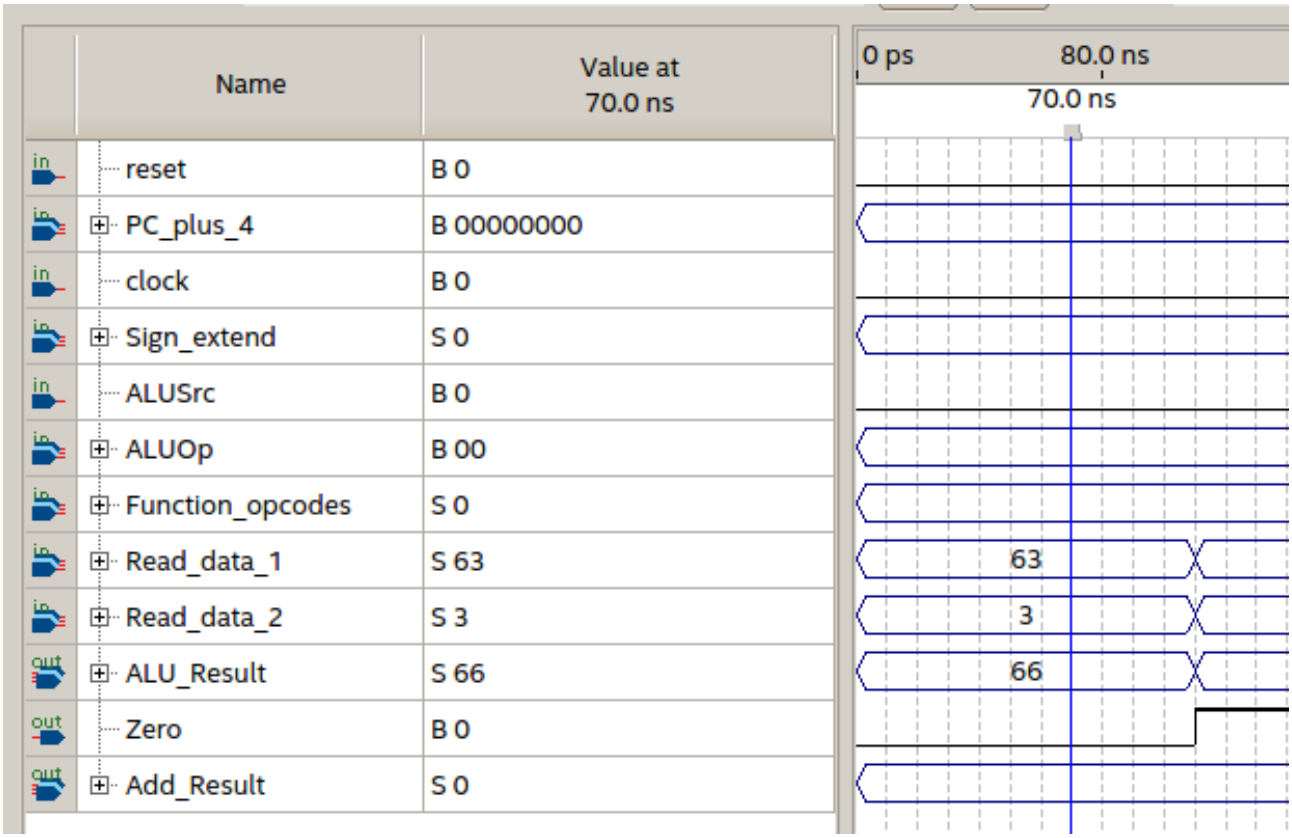


Figure 4: LWSW.png

Branch if equal

ALUOp == “01” FunField == “XXXXXX”

Desired ALU action → Sub.

Read_data_1 (257) - Read_data_2 (12) == 245

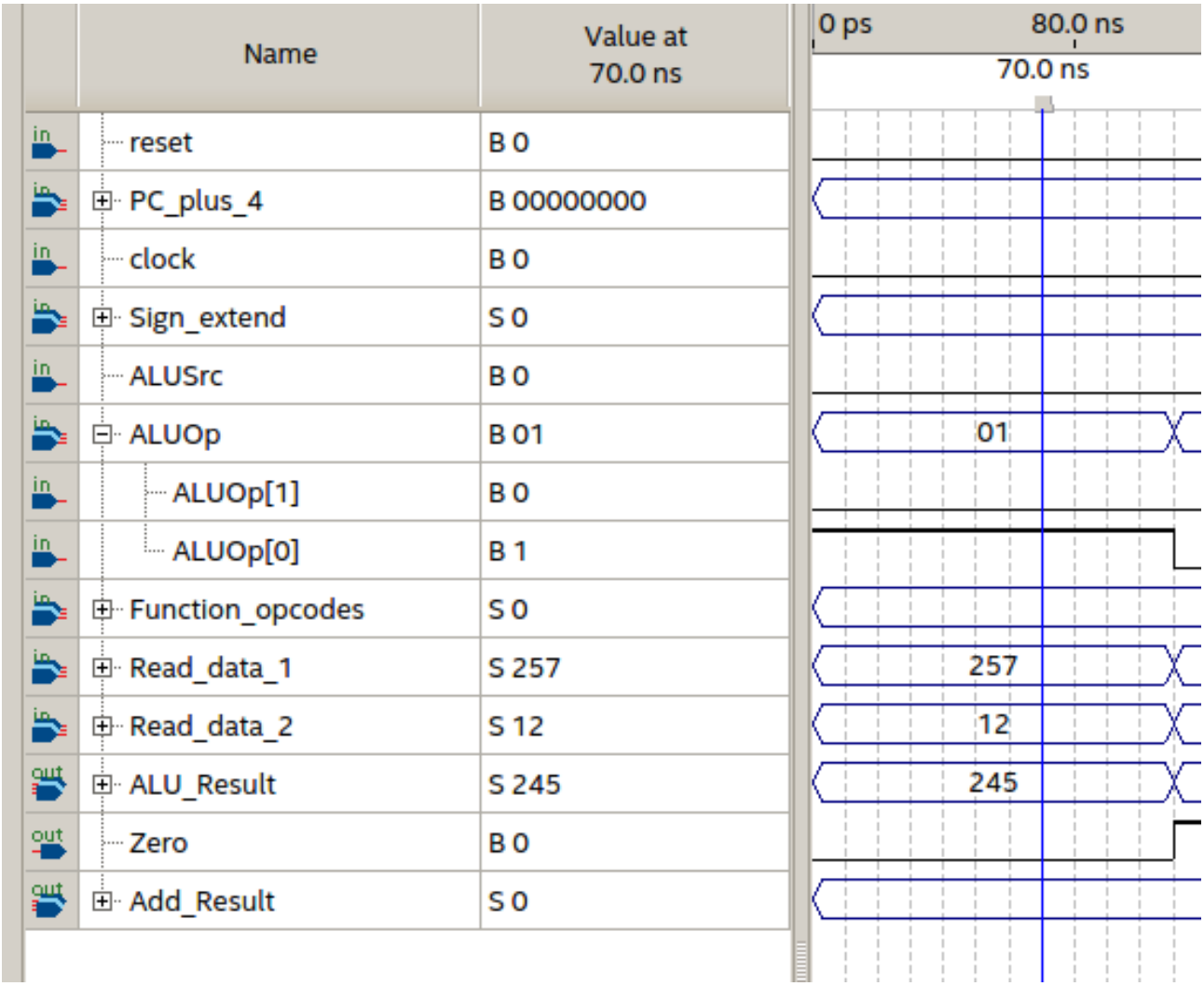


Figure 5: Beq.png

Πριν πάμε στον υπόλοιπο πίνακα, ας κάνουμε ένα μικρό “διάλειμμα” με την επόμενη κυματομορφή.

UNDEFINED

Με αυτήν την κυματομορφή, το σήμα ALU_ctl είναι “101”. Εν βάση τον κώδικα, στην έξοδο βγάζει X”00000000”, δηλαδή, 8 μηδενικά σε δεκαεξαδικό, άρα 8 * 4 == 32, το μέγεθος του bus εξόδου, άρα όλα τα bit είναι ‘0’.

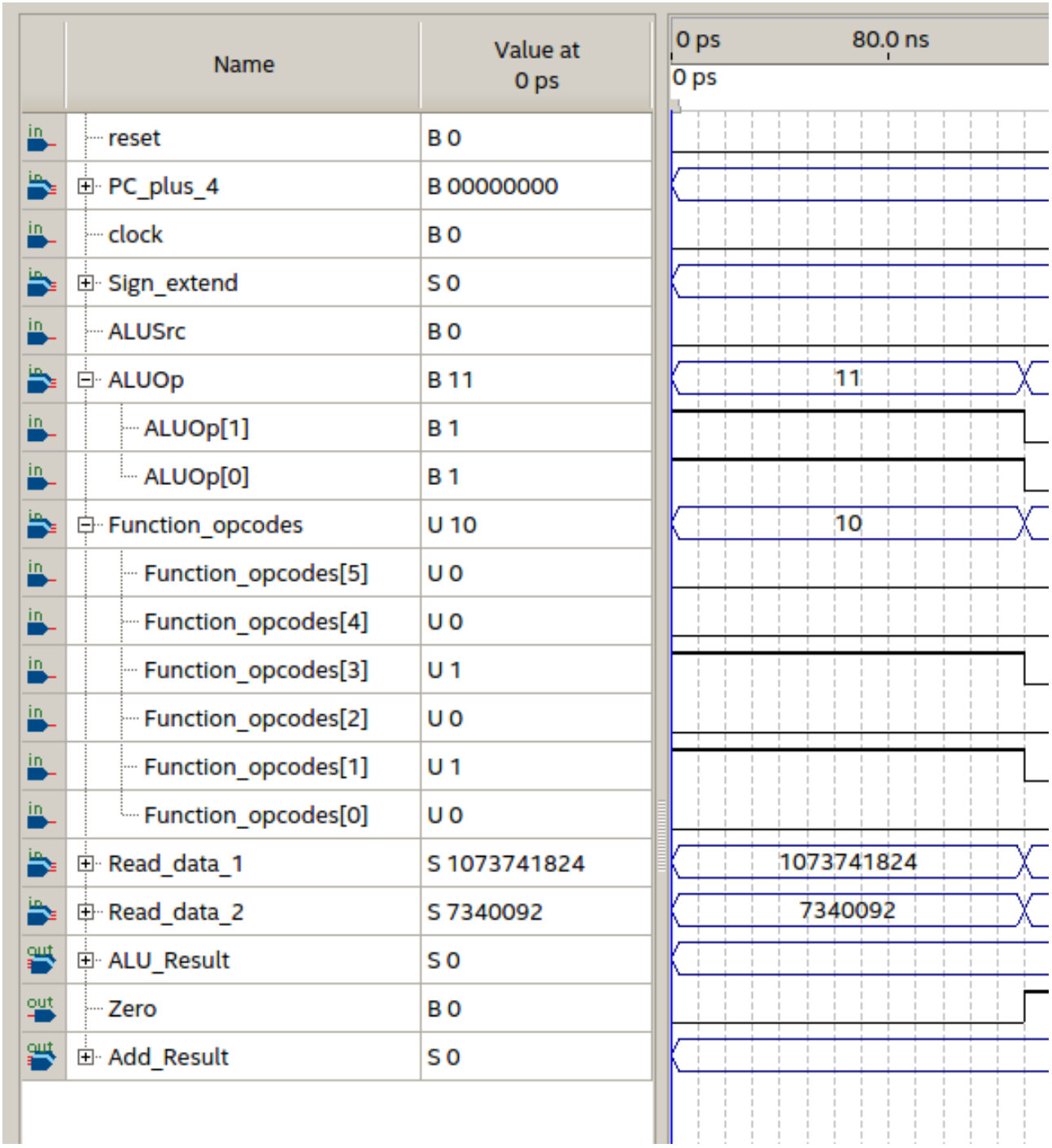


Figure 6: UNDEF.png

R type ⟶ Add

ALUOp == “10” FunField == “100000”

Έκανα κάτι παράξενο εδώ επίτηδες. Όταν ALUSrc == ‘1’ τότε αντί για Read_data_2 πιάνει το Sign_extend, πέρα από και άλλα πράγματα.

Read_data_1 (25165824) + Sign_extend (64) == 25165888

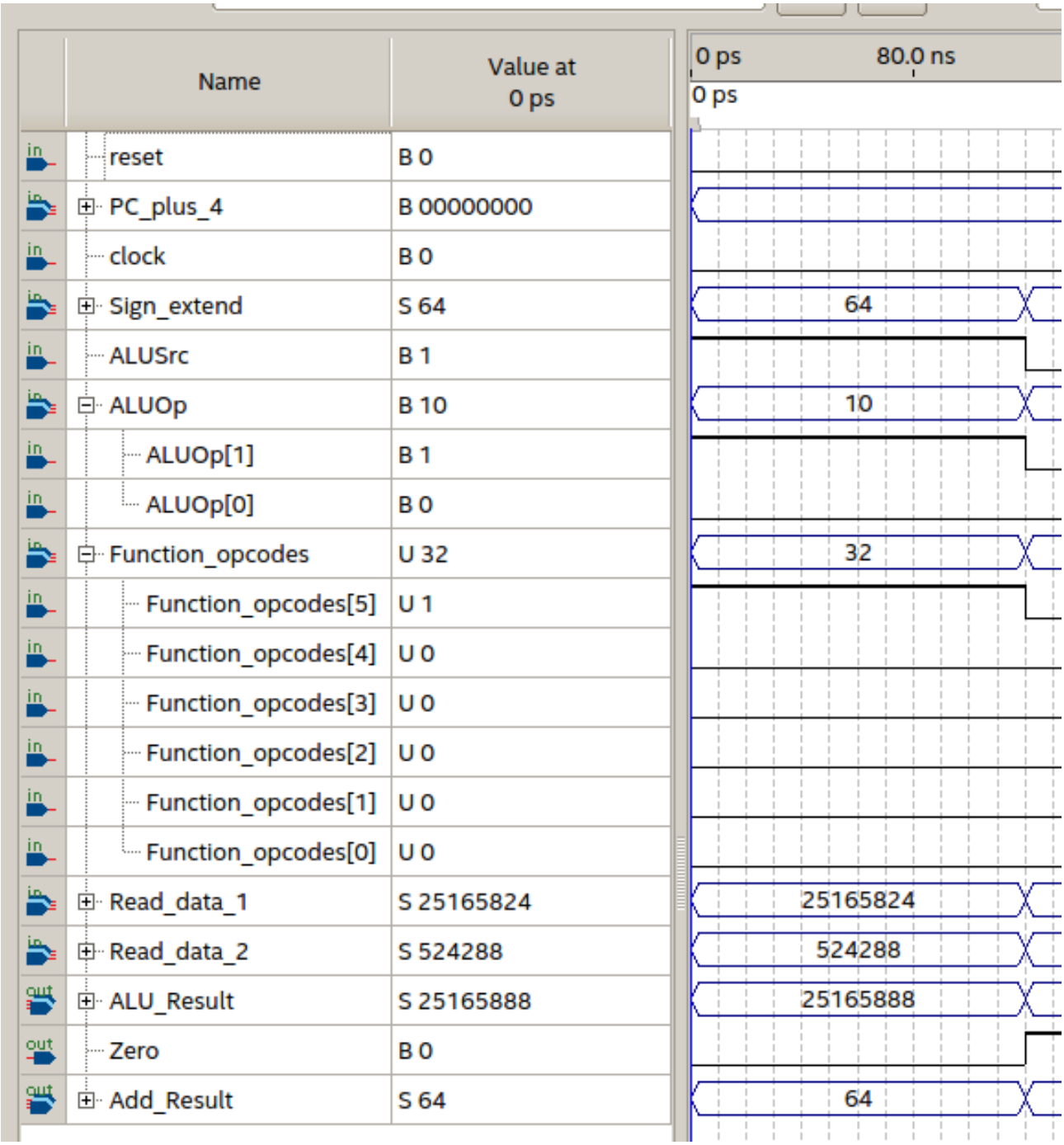


Figure 7: RADD_and_ALUSrc_with_Sign_extend.png

R type ⟶ Sub

ALUOp == “10” FunField == “100010”

Read_data_1 (32) - Read_data_2 (60) == -28

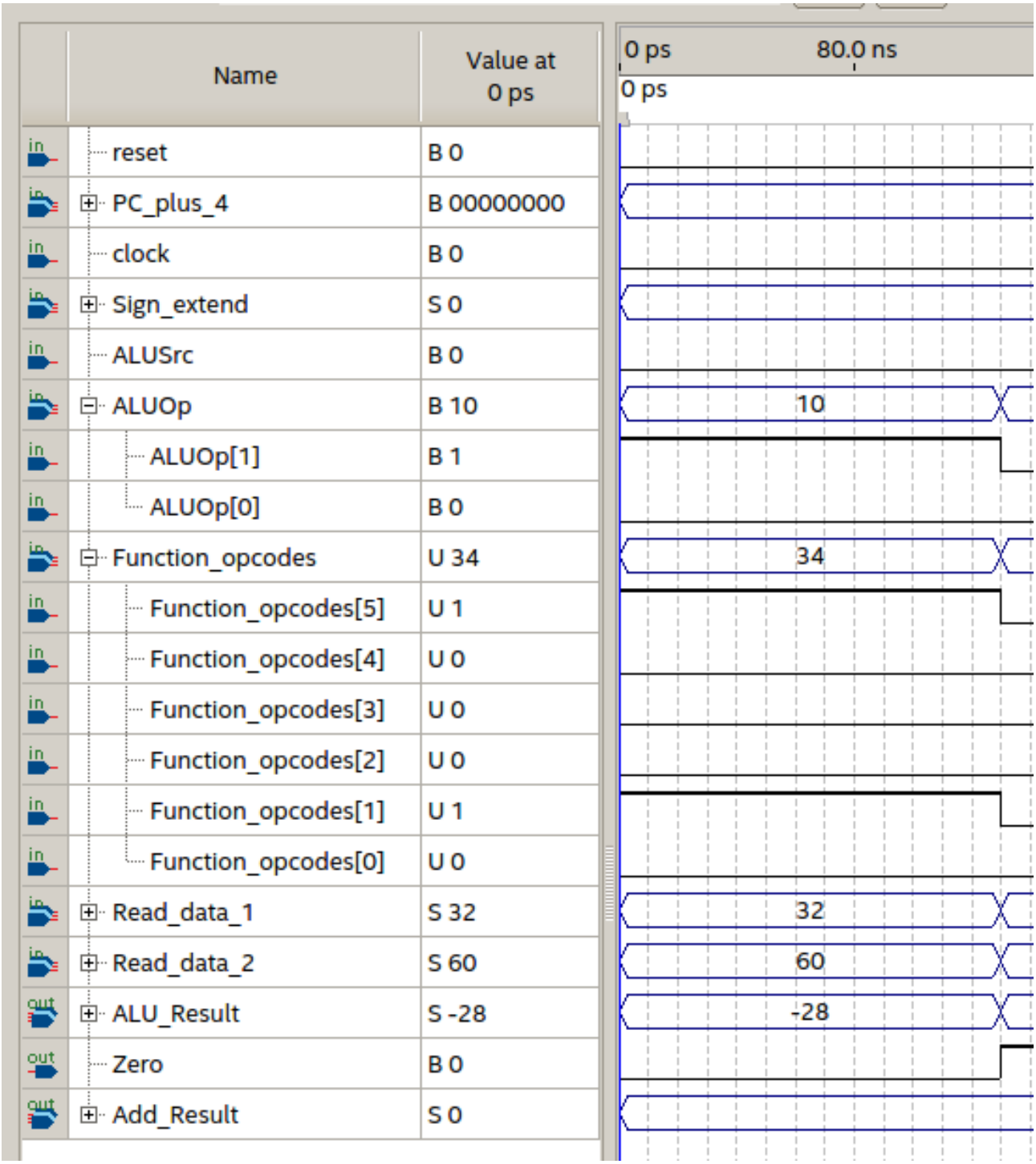
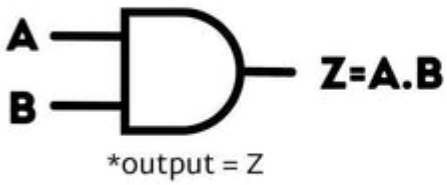


Figure 8: RSUB.png

R type ⟶ And

ALUOp = “10” FunField = “100100”

symbol + notation



truth table

A	B	Z=A.B
0	0	0
0	1	0
1	0	0
1	1	1

Πηγή

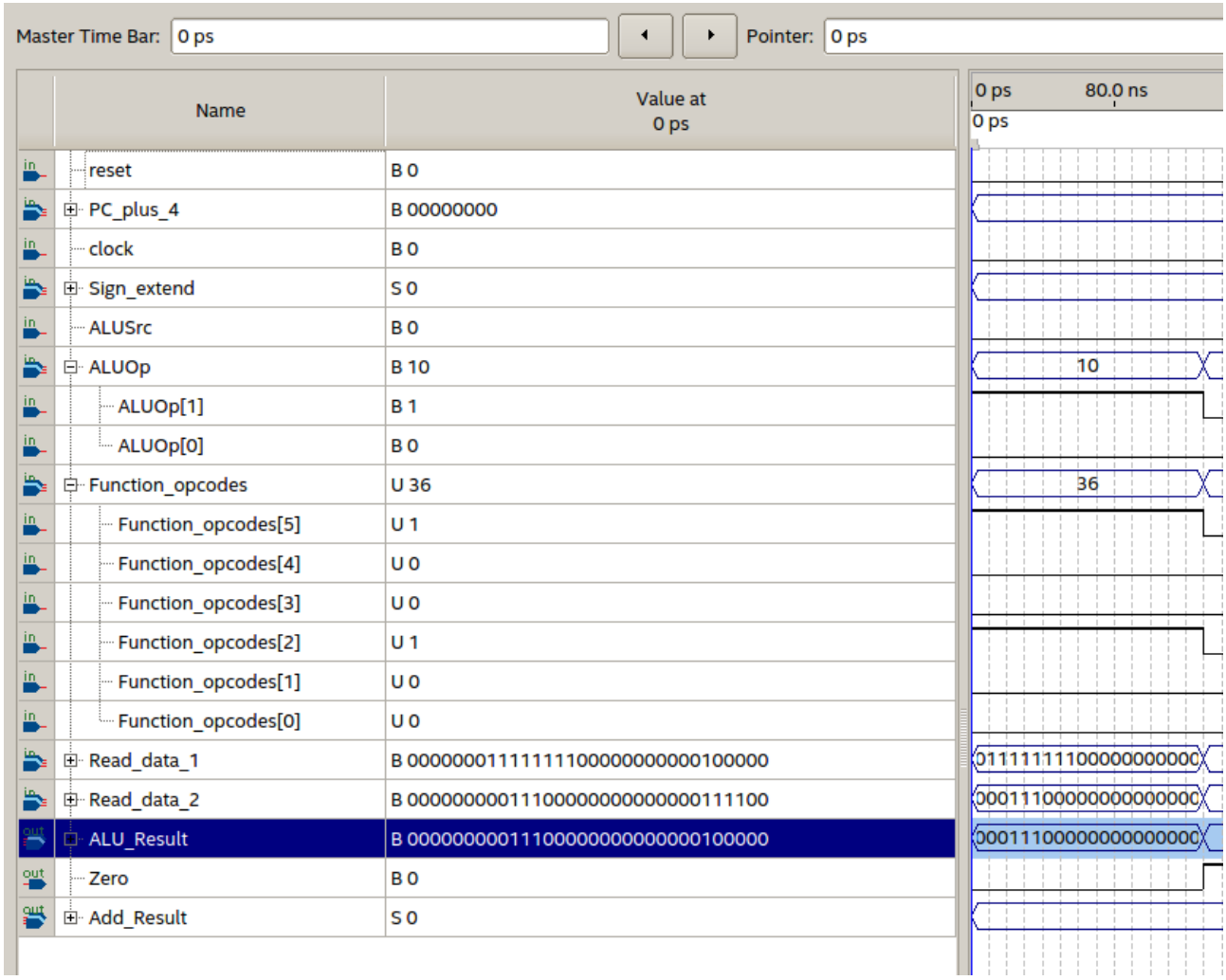
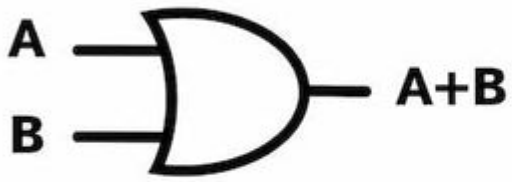


Figure 9: RAND.png

R type ⟶ Or

ALUOp = “10” FunField = “100101”



2 input OR gate

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

[Πηγή]

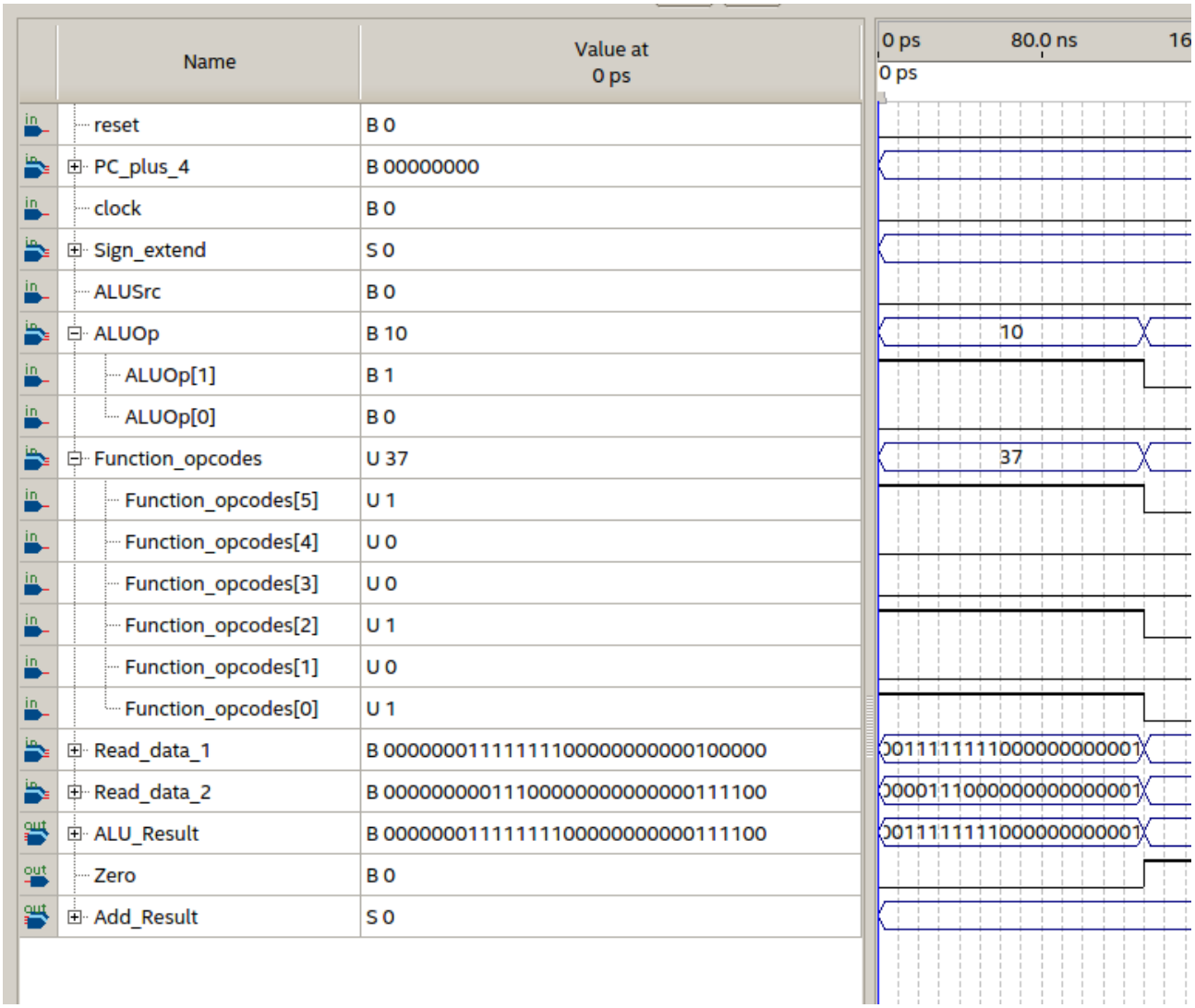
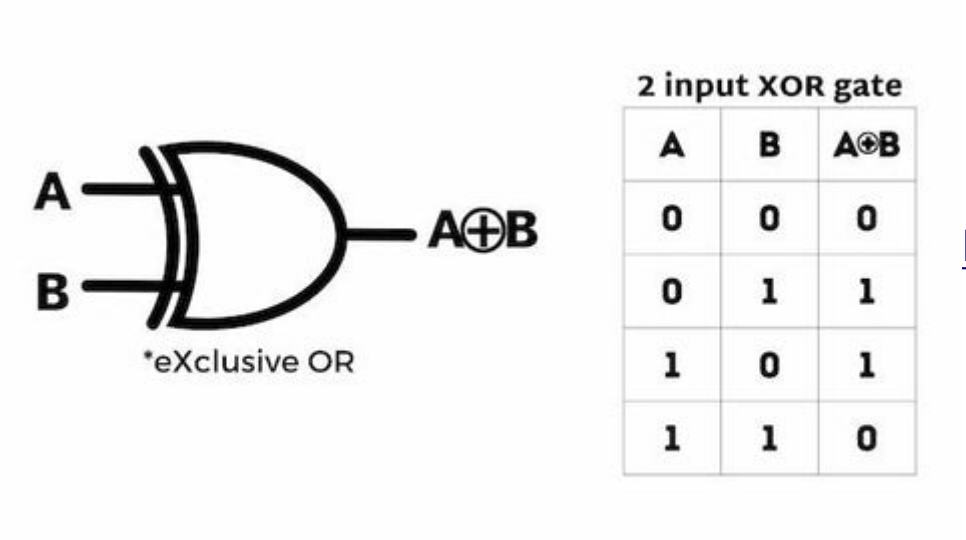


Figure 10: ROR.png

R type ⟶ Xor ⟶ (not A and B) or (A and not B)

ALUOp = “10” FunField = “101001”



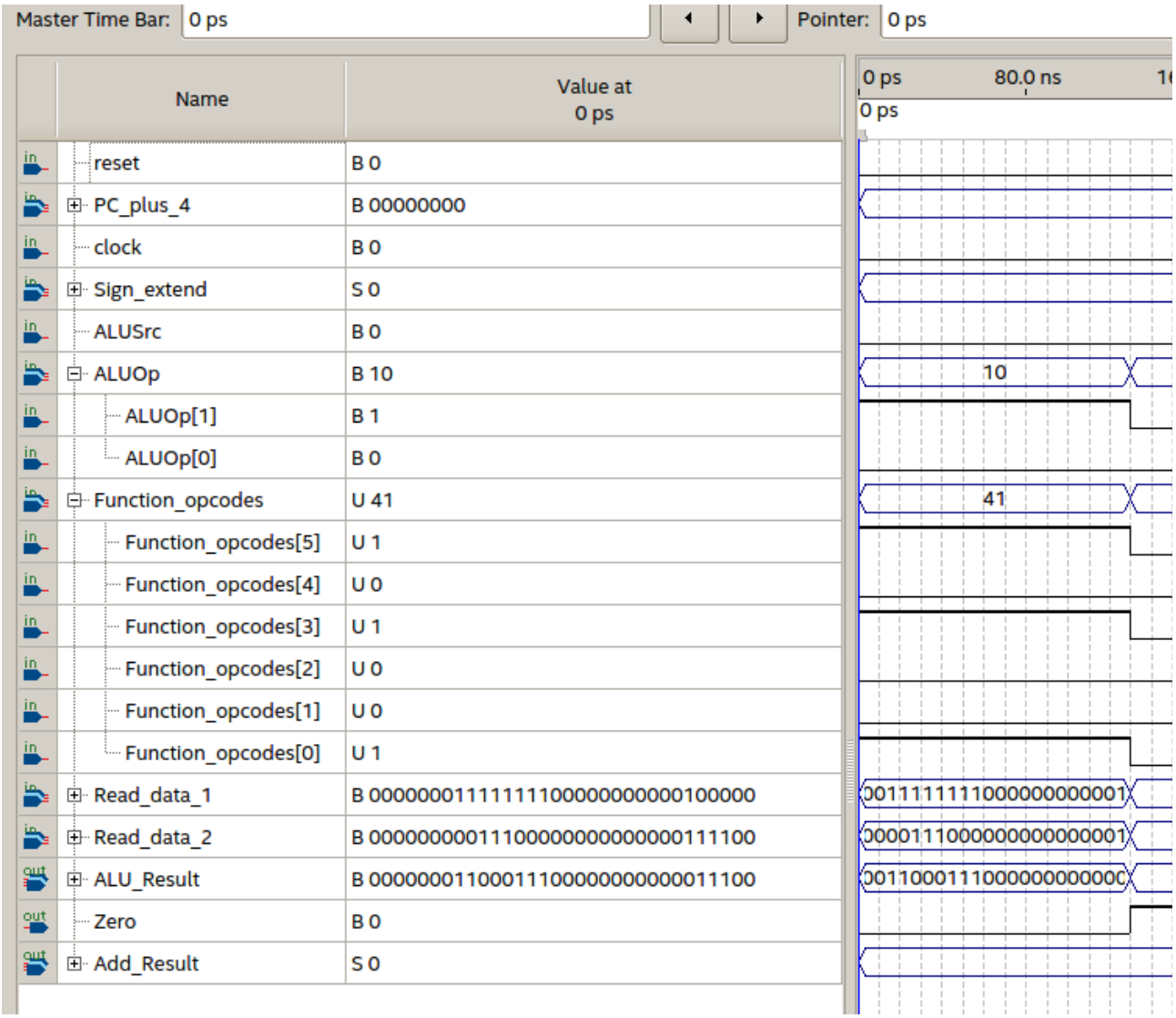


Figure 11: RXOR.png

R type ⟶ SLT ⟶ if (A < B) {return 1;} return 0;

ALUOp = “10” FunField = “101010”

Στην πρώτη εικόνα Read_data_1 < Read_data_2 άρα ALU_Result = 1

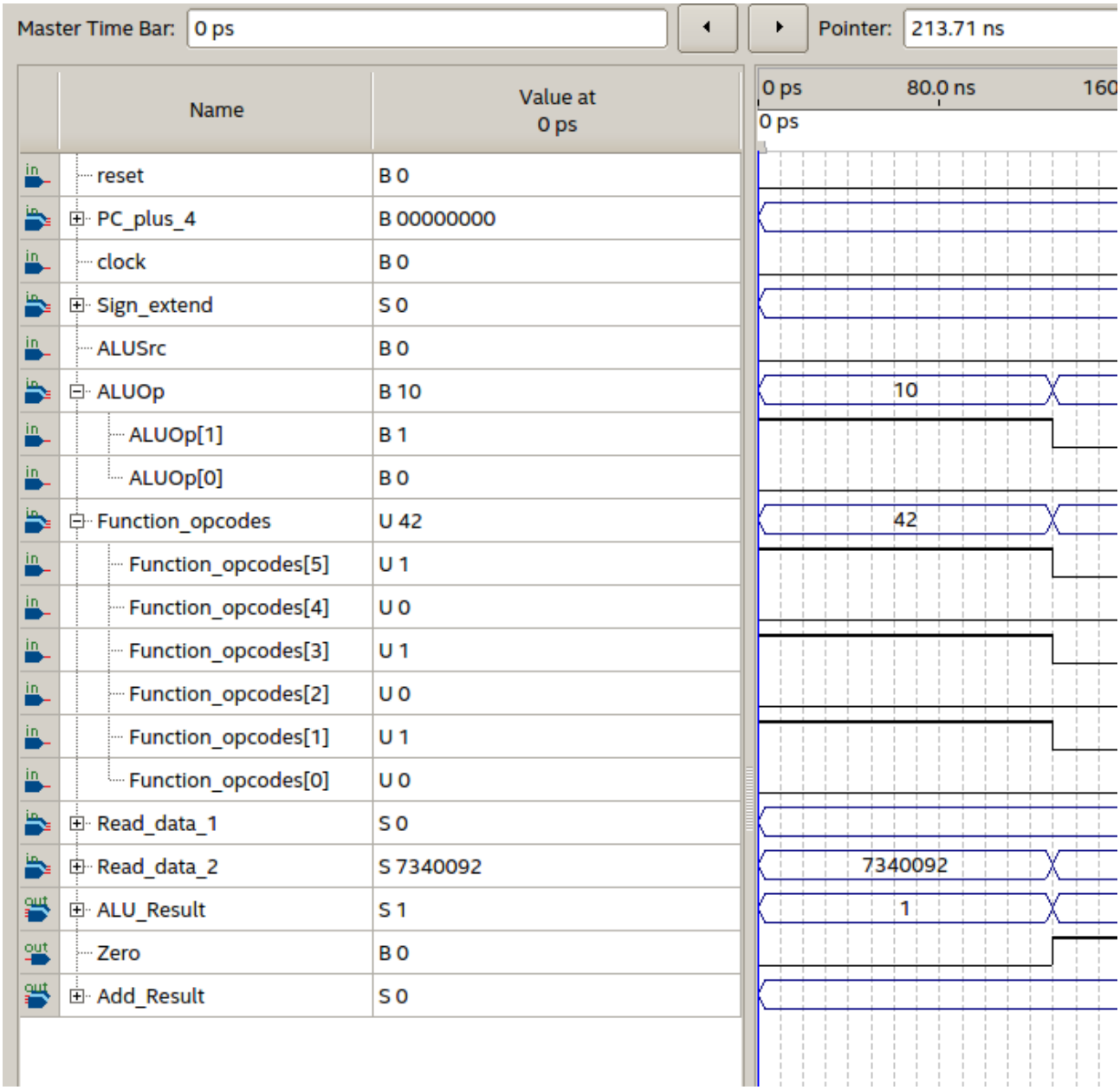


Figure 12: RSLTTRUE.png

Στην δεύτερη εικόνα όμως Read_data_1>Read_data_2, άρα SLT είναι false και ALU_Result == 0.

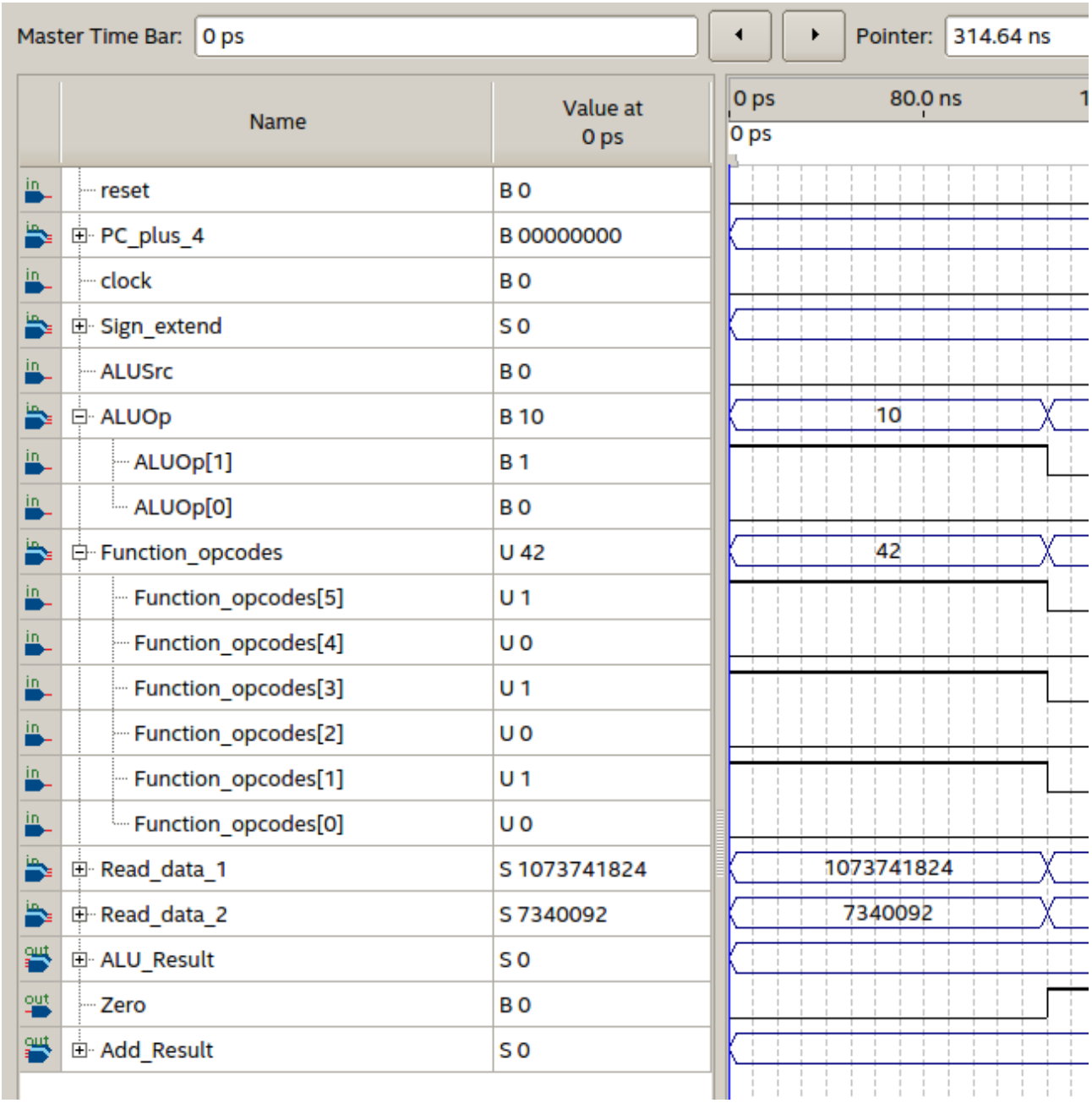


Figure 13: RSLTFALSE.png