ΟΡΓΑΝΩΣΗ ΥΠΟΛΟΓΙΣΤΩΝ
ΑΝΑΦΟΡΑ ΑΣΚΗΣΗΣ #1

SINGLE CYCLE MIPS PROCESSOR

Στρατάκης Ανδρέας - 2018030179
astratakis1@isc.tuc.gr

Github repository (public after due):
https://github.com/astratakis/Mips-Processor

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΟΛΟΓΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΑΠΡΙΛΙΟΣ 2022

ΜΕΡΟΣ Α – ΕΙΣΑΓΩΓΗ:

Σκοπός της εργαστηριακής άσκησης πρώτα είναι η μελέτη και η κατανόηση της λειτουργίας του επεξεργαστή ενός κύκλου και έπειτα η υλοποίηση του σε VHDL. Η αρχιτεκτονική (Instruction Set) που χρησιμοποιήθηκε είναι μια παραλλαγή της αρχιτεκτονικής MIPS η οποία περιέχει τις πολύ βασικές εντολές (arithmetic, memory access, branches). Λόγω της μεγάλης πολυπλοκότητας του προβλήματος, η υλοποίηση έγινε με τη μέθοδο "Διαίρει και Βασίλευε". Το τελικό πρόβλημα απλοποιήθηκε σε πολλά μικρότερα και απλούστερα. Στη συνέχεια υλοποιήθηκαν αυτά τα υποπροβλήματα κατασκευάζοντας τα απαραίτητα modules. Έπειτα modules συνδέθηκαν μεταξύ τους και δημιουργώντας μεγαλύτερα έως ότου δημιουργηθεί ο τελικός επεξεργαστής. Η υλοποίηση χωρίστηκε σε 3 μεγάλες φάσεις οι οποίες περιγράφονται αναλυτικά παρακάτω.

ΜΕΡΟΣ Β – ΥΛΟΠΟΙΗΣΗ:

ΠΡΩΤΗ ΦΑΣΗ:

Σε αυτή τη φάση σχεδιάστηκαν μια απλή Αριθμητική και Λογική Μονάδα (ALU) και το αρχείο καταχωρητών (Register File). Η ALU μπορεί να πραγματοποιήσει όλες τις πράξεις που περιγράφονται στο CHAR-Instruction-Set. Το Register File περιέχει 32 registers, έναν decoder 5:32 bits και δύο multiplexers 32:1 bits με 32-bit η κάθε είσοδος. Εδώ πρέπει να επισημανθεί ο τρόπος με τον οποίο υλοποιήθηκε ο decoder. Παρατηρήθηκε ότι η έξοδος του ισοδυναμεί με την πράξη bit shift left 1 << input.

```
constant one: std_logic_vector(31 downto 0) := x"00000001";
begin
    output <= std_logic_vector(unsigned(one) sll to_integer(unsigned(input))) after latency;
end decoder;
```

Οι multiplexers φτιάχτηκαν ορίζοντας logic vector array. Αυτό ορίστηκε σε ένα εξωτερικό package (βλ. utils.vhd) προκειμένου να μπορεί να χρησιμοποιηθεί από πολλά modules. Το input στον κώδικα παρακάτω είναι array από 32 bit logic vectors.

```
architecture Structural of mux_32 is
begin
    output <= input(to_integer(unsigned(sel))) after latency;
end Structural;
```

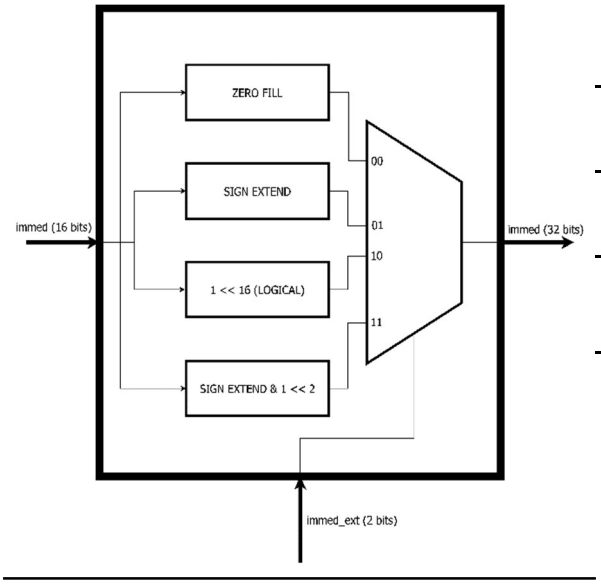
Το Register File έχει δυνατότητα σύγχρονης εγγραφής σε 1 καταχωρητή και ασύγχρονης ανάγνωσης 2 καταχωρητών. Ο καταχωρητής R0 έχει πάντα την τιμή 0x0000. Αυτό γίνεται έχοντας το write enable του πάντα στην τιμή '0'. Παρακάτω φαίνεται ο τρόπος με τον οποίο υλοποιήθηκε. Το write enable array είναι 32 bit logic vector με 1 bit για κάθε write enable του κάθε καταχωρητή. Τέλος στο Register File προστέθηκε και reset signal για αρχικοποίηση.

```
write_enable_array <= decoded_address_write and x"00000000" when write_enable = '0' else
decoded_address_write and x"fffffffe";
```

ΔΕΥΤΕΡΗ ΦΑΣΗ:

Σε αυτή τη φάση κατασκευάστηκαν τα 4 μεγάλα μέρη του DATAPATH. Αυτά είναι τα IFSTAGE, DECSTAGE, EXSTAGE MEMSTAGE. Το IFSTAGE η αλλιώς Instruction Fetch Stage είναι υπεύθυνο για την ανάρτηση εντολών προς εκτέλεση από τη μνήμη. Περιέχει τον καταχωρητή Program Counter (PC). Ωστόσο επειδή η μνήμη RAM μπορεί διευθυνσιτοδοτήσει μέχρι 2^{11} = 2048 θέσεις μνήμης η RAM δέχεται σαν είσοδο τα bits 12 έως και 2 του Program Counter, το οποίο ισοδυναμεί με την πράξη (PC >> 2) % 2^{11}.

Η αποκωδικοποίηση των εντολών γίνονται στο στάδιο DECSTAGE. Συγκεκριμένα σπάει η εντολή στα επιμέρους κομμάτια, δηλαδή στο rd, rt, rs και immediate και επιλέγονται οι επιμέρους καταχωρητές. Στον καταχωρητή rd γίνεται εγγραφή ενώ στους άλλους 2 ανάγνωση. Τέλος το immediate κομμάτι της εντολής από 16 bits μετατρέπεται σε 32 bits προκειμένου να μπορεί να χρησιμοποιηθεί από την ALU και το IFSTAGE. Η υλοποίηση του immediate extender φαίνεται παρακάτω.

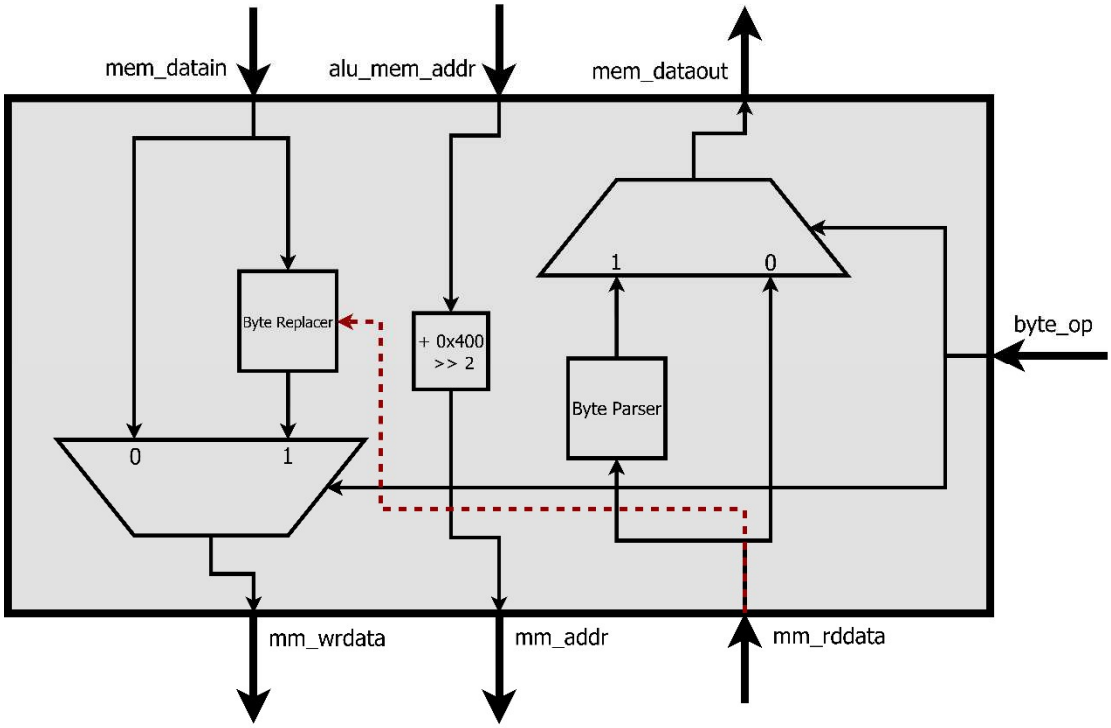


imm_ext (mux-select)	operation
00	zero fill
01	sign extend
10	<< 16 (logical)
11	sign extend and << 2

Η εκτέλεση των εντολών γίνεται στην βαθμίδα EXSTAGE (Execution Stage). Εδώ εκτελούνται όλες οι εντολές που έχουν να κάνουν με άμεση πράξη (add, sub κλπ.) αλλά και οι εντολές που πραγματοποιούν έμμεση πράξη (beq, sw κλπ). Η μόνη εντολή που δε χρησιμοποιεί την ALU είναι η b (branch).

Το στάδιο MEMSTAGE είναι υπεύθυνο για την προσπέλαση της μνήμης RAM. Η RAM χωρίζεται σε 2 τμήματα, η επιλογή των οποίων έγινε τυπικά. Προφανώς δόθηκε παραπάνω χώρος για το Data Segment (heap, stack):

- Code Segment (0 - 255).
- Data Segment (256 - 2047).



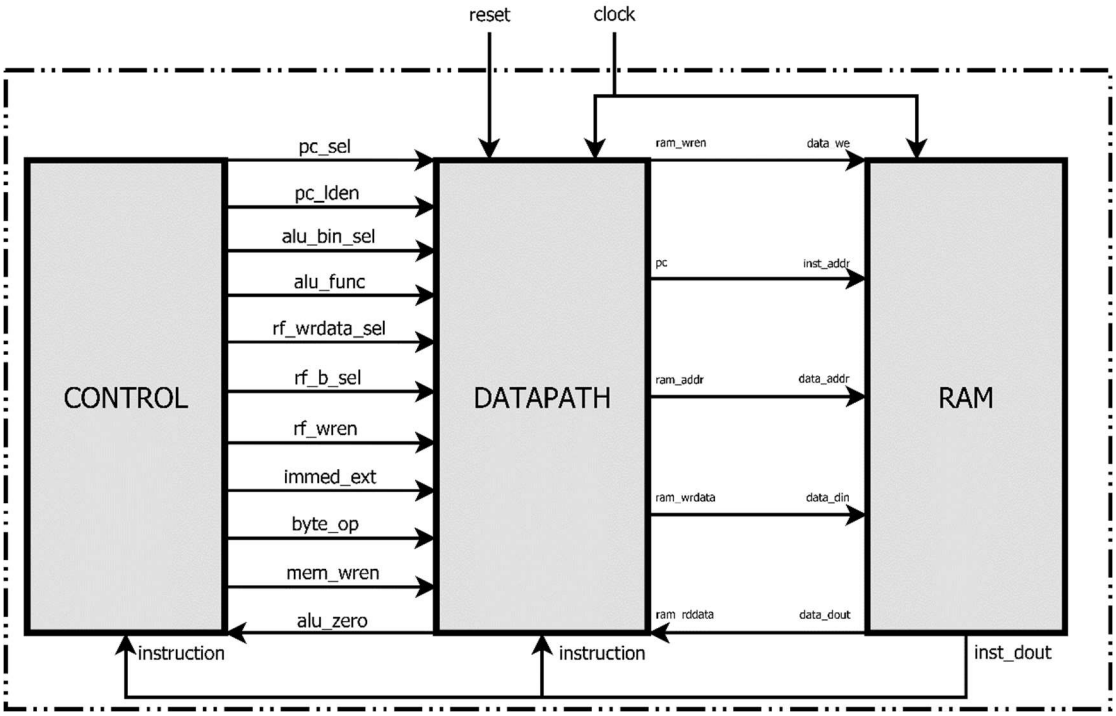
Block Diagram MEMSTAGE

Εδώ πραγματοποιούνται και οι αντίστοιχες τροποποιήσεις (χρήση μάσκας στους Byte Parser, Replacer) στη λέξη που διαβάζεται ή γράφεται στη μνήμη όταν εκτελούνται οι εντολές lb ή sb. Προκειμένου να μη διαγραφούν δεδομένα από τη μνήμη όταν γίνεται store byte, αντί για zero fill γίνεται προσθήκη της υπόλοιπης λέξης από τη συγκεκριμένη διεύθυνση.

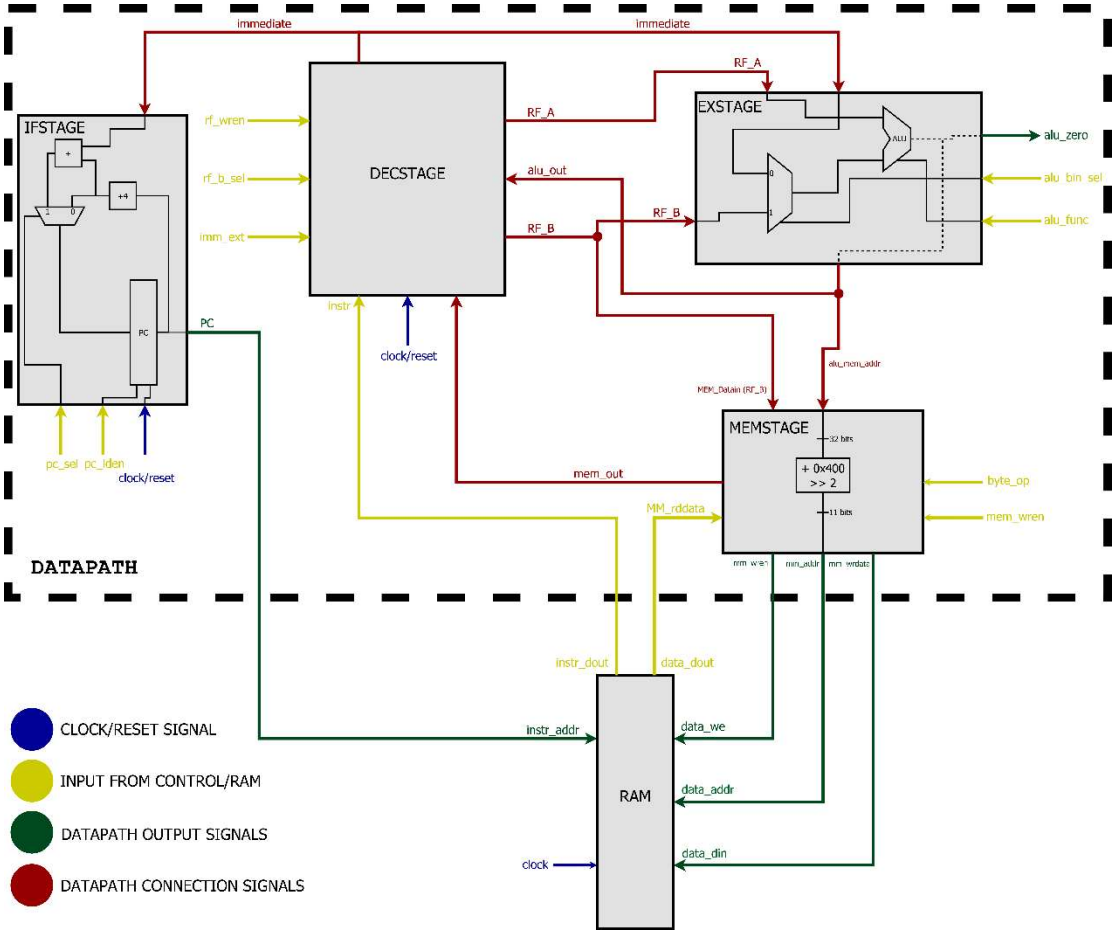
```
byte <= word(31 downto 0) and x"000000ff" when sel = "00"
        else x"00" & (word(31 downto 8) and x"0000ff") when sel = "01"
        else x"0000" & (word(31 downto 16) and x"00ff") when sel = "10"
        else x"000000" & (word(31 downto 24) and x"ff");

word_out <= ram_data(31 downto 8) & byte when sel = "00"
           else ram_data(31 downto 16) & byte & ram_data(7 downto 0) when sel = "01"
           else ram_data(31 downto 24) & byte & ram_data(15 downto 0) when sel = "10"
           else byte & ram_data(23 downto 0);
```

ΤΡΙΤΗ ΦΑΣΗ:

Τα 4 μεγάλα components που περιγράφονται παραπάνω ενώθηκαν και δημιουργήθηκε το DATAPATH. Παράλληλα με το DATAPATH δημιουργήθηκε και η μνήμη RAM μεγέθους 2048x4 bytes καθώς και το CONTROL. Το CONTROL τροφοδοτεί σχεδόν όλες τις εισόδους του DATAPATH. Παρακάτω τα Block Diagrams του DATAPATH και του τελικού επεξεργαστή.



Block Diagram processor single cycle



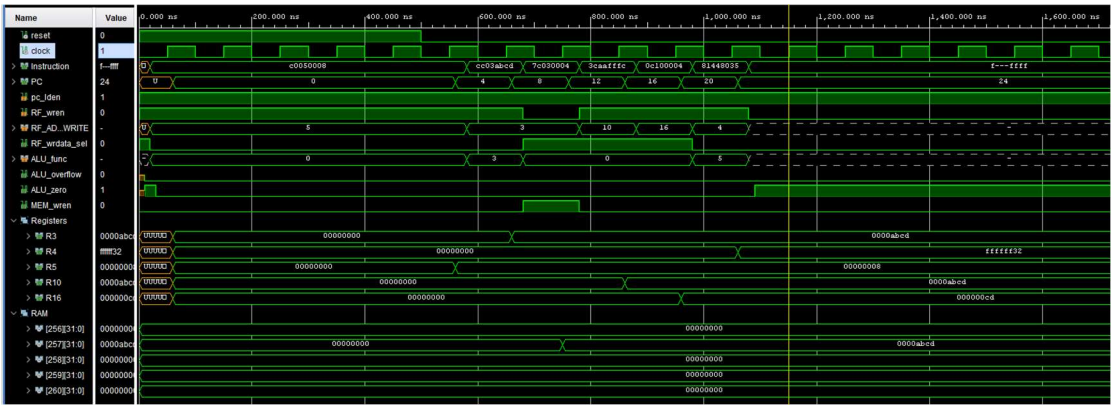
Block Diagram DATAPATH with RAM

ΜΕΡΟΣ Γ - BENCHMARKS:

Σε αυτή τη φάση κατασκευάστηκαν προγράμματα προκειμένου να γίνει έλεγχος της ορθότητας του επεξεργαστή. Τα τεστ χωρίζονται σε διάφορες κατηγορίες ανάλογα με τη δυσκολία. Τα 1 και 2 είναι τα προγράμματα της εκφώνησης. Το πρόγραμμα 3 έχει ως στόχο τον έλεγχο των load, store byte. Τέλος το 4 έχει ως στόχο να σφραγίσει την ορθότητα του επεξεργαστή. Στη μνήμη υπάρχει ένα string από χαρακτήρες ascii. Το πρόγραμμα διασχίζει όλο το string διαβάζοντας 1 λέξη τη φορά (4 χαρακτήρες τη φορά) και κρατάει μόνο τους λατινικούς χαρακτήρες αγνοώντας σύμβολα και αριθμούς. Οι χαρακτήρες που αγνοούνται αντικαθίστανται από το πολύ 1 space χαρακτήρα. Τέλος η αποκρυπτογραφμένη λέξη αποθηκεύεται στη μνήμη (4 χαρακτήρες κάθε φορά). (βλ. test.asm, memory_creator.py)

Program 1: (RAM: program1.data)

Instruction	Expectation
00: addi r5 r0 8	Write (0+8) = 8 in R5
04: ori r3 r0 0xABCD	Write (0 or 0xABCD) = 0xABCD in R3
08: sw r3 4(r0)	Write R3 in RAM[(4+0x400)/4] = RAM[257]
0C: lw r10 -4(r5)	Load RAM[(R5-4+0x400)/4] = RAM[257] into R10
10: lb r16 4(r0)	Load RAM[(R0+4+0x400)/4] = RAM[257], byte (R0+4+0x400)%4 = byte 0 into R16
14: nand r4 r10 r16	Write (R10 nand R16) = 0xffffff32 in R4
18: b -1	Branch PC = PC + 4 - 4 (HALT PC)!!!



Program 1 Simulation

Program 2: (RAM: program2.data)

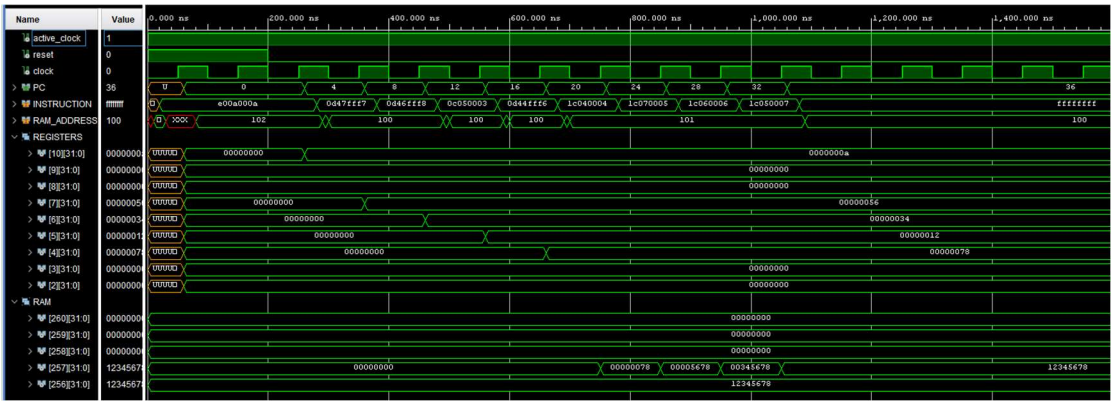
Instruction	Expectation
00: bne r5 r5 8	Failed Branch
04: b -2	PC = PC + 4 - 8 = 0
08: addi r1 r0 1	Never reached

Αναμένεται ατέρμονη επανάληψη των εντολών bne, b διότι για οποιαδήποτε τιμή του R5, ισχύει R5 = R5.

Program 2 Simulation

Program 3: (RAM: program3.data)

Instruction	Expectation
00: addi r10 r10 10	R10 = 10
04: lb r7 -9(r10)	R7 = RAM[256](15 downto 8) = 0x56
08: lb r6 -8(r10)	R6 = RAM[256](23 downto 16) = 0x34
0C: lb r5 3(r0)	R5 = RAM[256](31 downto 24) = 0x12
10: lb r4 -10(r10)	R4 = RAM[256](7 downto 0) = 0x78
14: sb r4 4(r0)	RAM[257](7 downto 0) = 0x78
18: sb r7 5(r0)	RAM[257](15 downto 8) = 0x56
1C: sb r6 6(r0)	RAM[257](23 downto 16) = 0x34
20: sb r5 7(r0)	RAM[257](31 downto 24) = 0x12



Program 3 Simulation

Το σημαντικό στην παραπάνω προσομοίωση είναι ότι η store byte δε διαγράφει τα υπόλοιπα bytes της λέξης. Χρησιμοποιώντας ακριβώς **4 sb** εντολές πραγματοποιήθηκε αντιγραφή των δεδομένων της διεύθυνσης **RAM[256]** στη διεύθυνση **RAM[257].**

Program 4: (RAM: decryptor.data)

Το πρόγραμμα αυτό περιέχει 100 instructions στο Code segment και 90 λέξεις φορτωμένες στη μνήμη. Κατασκευάστηκε επίσης και το πρόγραμμα register_simulator.py, το οποίο δέχεται data segment και code segment και προσομοιώνει το αποτέλεσμα των registers, του PC, της RAM (μόνο το data segment) και το συνολικό αριθμό instructions που έτρεξαν. Έχει γίνει η παραδοχή ότι το πρόγραμμα τερματίζει όταν εκτελείται η εντολή **b -1** όπου ο PC παραμένει σταθερός.

Expectations

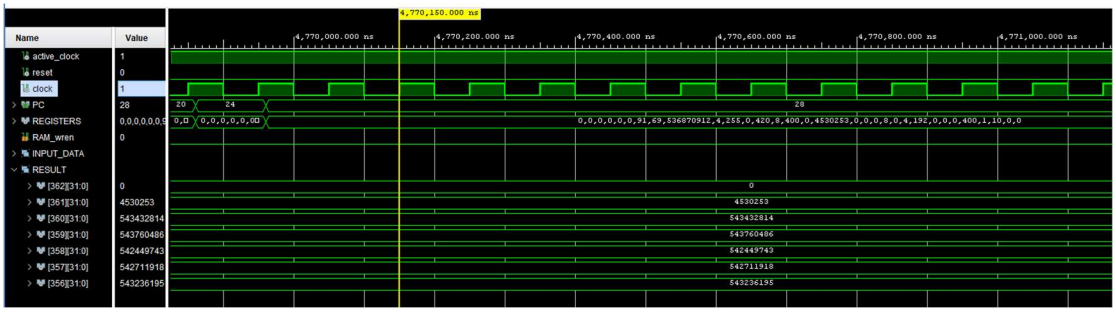
PC: 28

REGISTERS: [0, 0, 0, 0, 0, 0, 91, 69, 536870912, 4, 255, 0, 420, 8, 400, 0, 4530253, 0, 0, 0, 8, 0, 4, 192, 0, 0, 0, 400, 1, 10, 0, 0]

RAM: 356 543236195 | 357 542711918 | 358 542449743 | 359 543760486 | 360 543432814 | 361 4530253 | 362 0 |

Total instructions Executed: 47696

Hidden Word (little endian ASCII): C a n Y O U f i n d M E



Program 4 Simulation

Όπως φαίνεται απ' την παραπάνω προσομοίωση επικυρώνονται όλες οι αναμενόμενες τιμές των registers, του PC και της μνήμης στις συγκεκριμένες θέσεις. Όσον αφορά τον αριθμό των instructions που έτρεξαν, θεωρείται πως η τελευταία εντολή (**b -1**) τελειώνει στα 4.770.150 ns. Η πρώτη εντολή εκτελέστηκε στα 550 ns (εκτός rst). Υπολογίζοντας τη διαφορά και διαιρώντας με το χρόνο ανά κύκλο ρολογιού (100 ns) φαίνεται ότι έτρεξαν ακριβώς 47.696 εντολές.