ΠΡΟΧΩΡΗΜΕΝΗ ΛΟΓΙΚΗ ΣΧΕΔΙΑΣΗ

ΑΝΑΦΟΡΆ 5ου ΕΡΓΑΣΤΗΡΙΟΥ

LAB 20335359

Γιώργος Βιριράκης 2016030035
Μιχάλης Γαλάνης 2016030036

ΣΚΟΠΟΣ

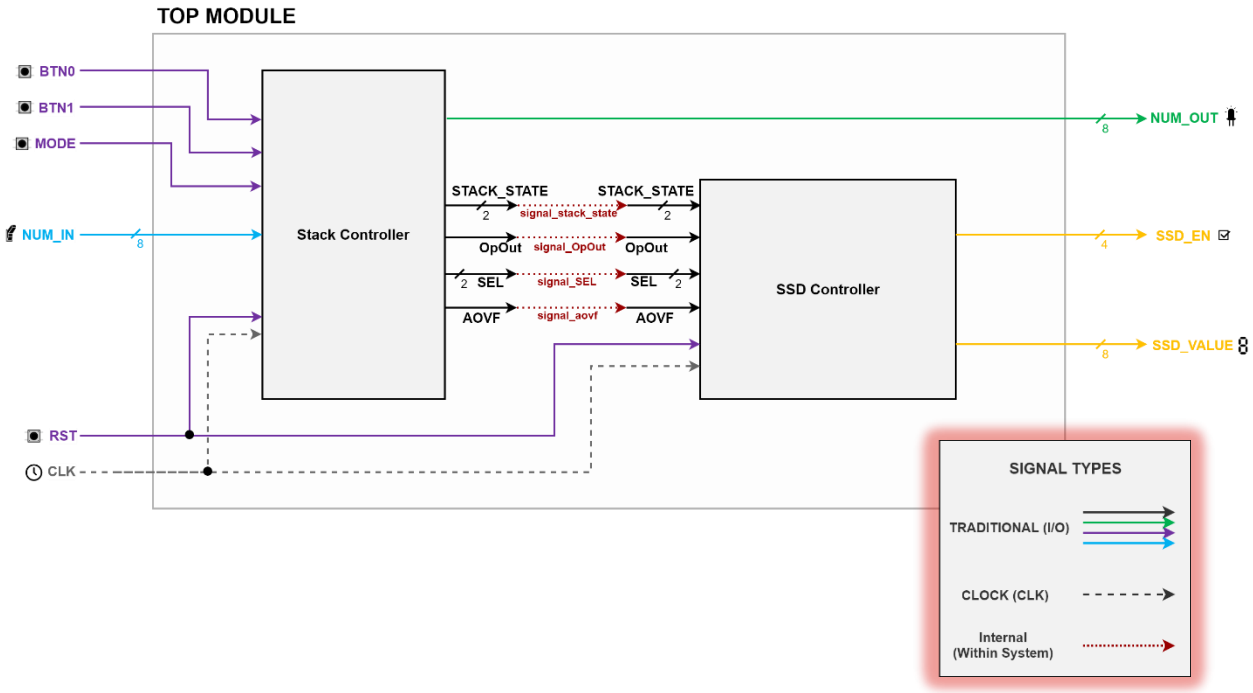
Στο 6ο και τελευταίο εργαστήριο, σκοπός της άσκησης ήταν η ολοκλήρωση της κατασκευής μιας λειτουργικής αριθμομηχανής που υλοποιεί πράξεις πάνω σε στοίβα αξιοποιώντας τα προηγούμενα δύο εργαστήρια. Οι πράξεις συμπεριλαμβάνουν τα **push, pop, add (2's complement), sub (2's complement), unary sub (2's complement** και **swap (X<>Y).**

ΠΡΟΕΤΟΙΜΑΣΙΑ - ΠΕΡΙΓΡΑΦΗ

Οι είσοδοι και έξοδοι του κυκλώματος του 5ου εργαστηρίου φαίνονται παρακάτω:

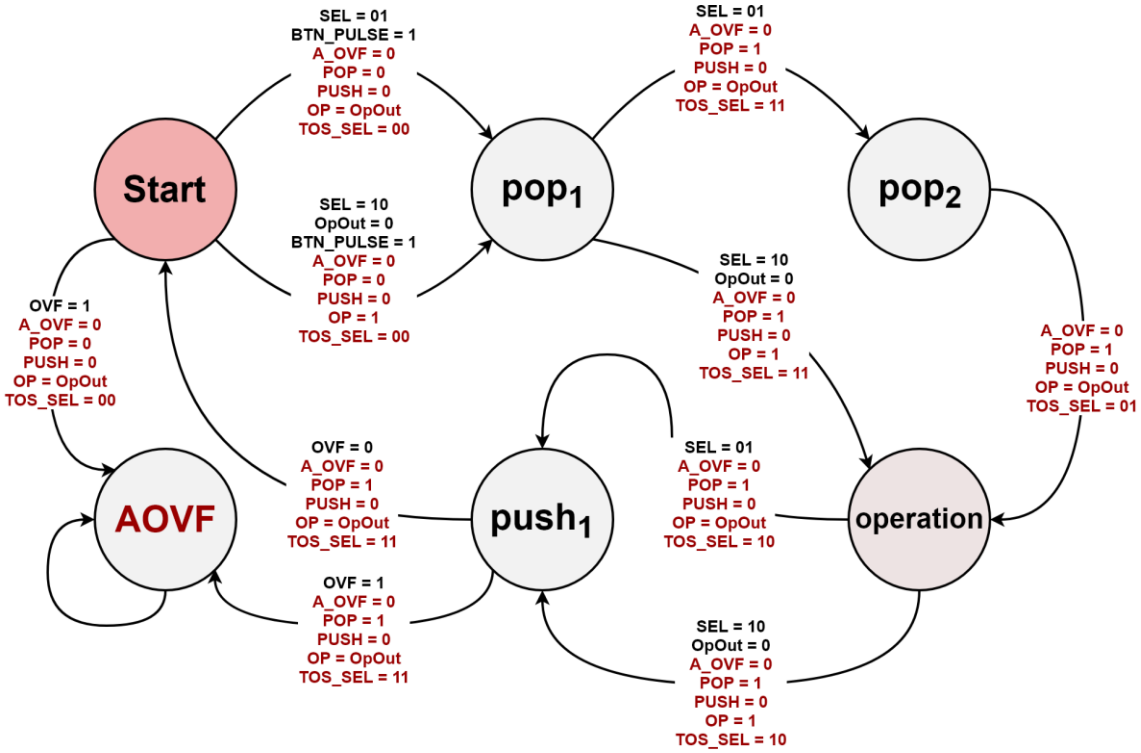
Name	IN / OUT	No. of Bits	FPGA Pins
Clock	in	1	MCLK
BTN_0	in	1	BTN_0
BTN_1	in	1	BTN_1
Mode	in	1	BTN_2
Reset	in	1	BTN_3
Num_In	in	8 (Bus)	$SW_{[7:0]}$
Num_Out	out	8 (Bus)	$LD_{[7:0]}$
SSD_En	out	4 (Bus)	$AN_{[3:0]}$
SSD_Value	out	8 (bus)	$SEG_{[7:0]}$

Εξωτερικά, δεν υπήρχαν αλλαγές στις εισόδους/εξόδους του κυκλώματος μας. Όλες οι καινούργιες λειτουργίες πραγματοποιήθηκαν με προσθήκη **εσωτερικών σημάτων.** Ακολουθεί το ανανεωμένο TOP Module:

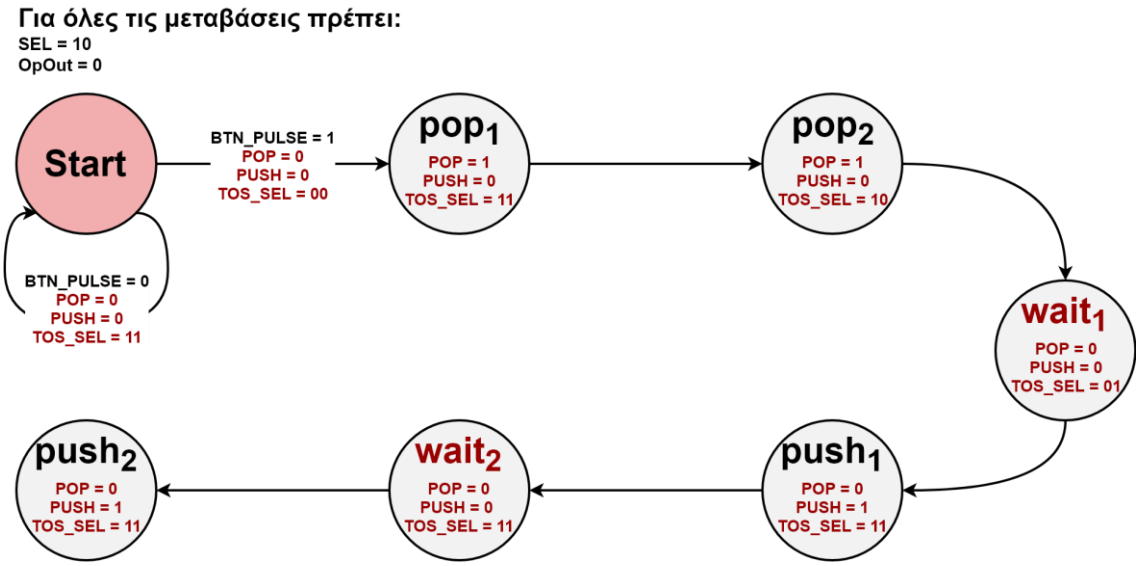


Όλες οι πράξεις πραγματοποιούνται στο **Stack Controller** μιας και υπάρχει άμεση επικοινωνία με τη μνήμη **Stack**. Για την υλοποιησή τους κατασκευάσαμε 2 καινούργιες FSM, την **ADD_SUB_UN_FSM** και τη **SWAP_FSM** οι οποίες εκτελούσαν την εκάστοτε πράξη και εισήγαγαν τα αποτελέσματά τους στη Stack. Οι πράξεις add, sub και unary sub ομαδοποιήθηκαν σε μια FSM καθώς αξιοποιούν και οι 3 το κύκλωμα του προσθαφαιρετή που θα περιγραφεί αργότερα. Οι δύο αυτές FSM είναι τύπου Mealy και τα διαγράμματα καταστάσεών τους παρουσιάζονται σε επόμενα σχήματα.

ADD_SUB_UN_FSM



SWAP_FSM



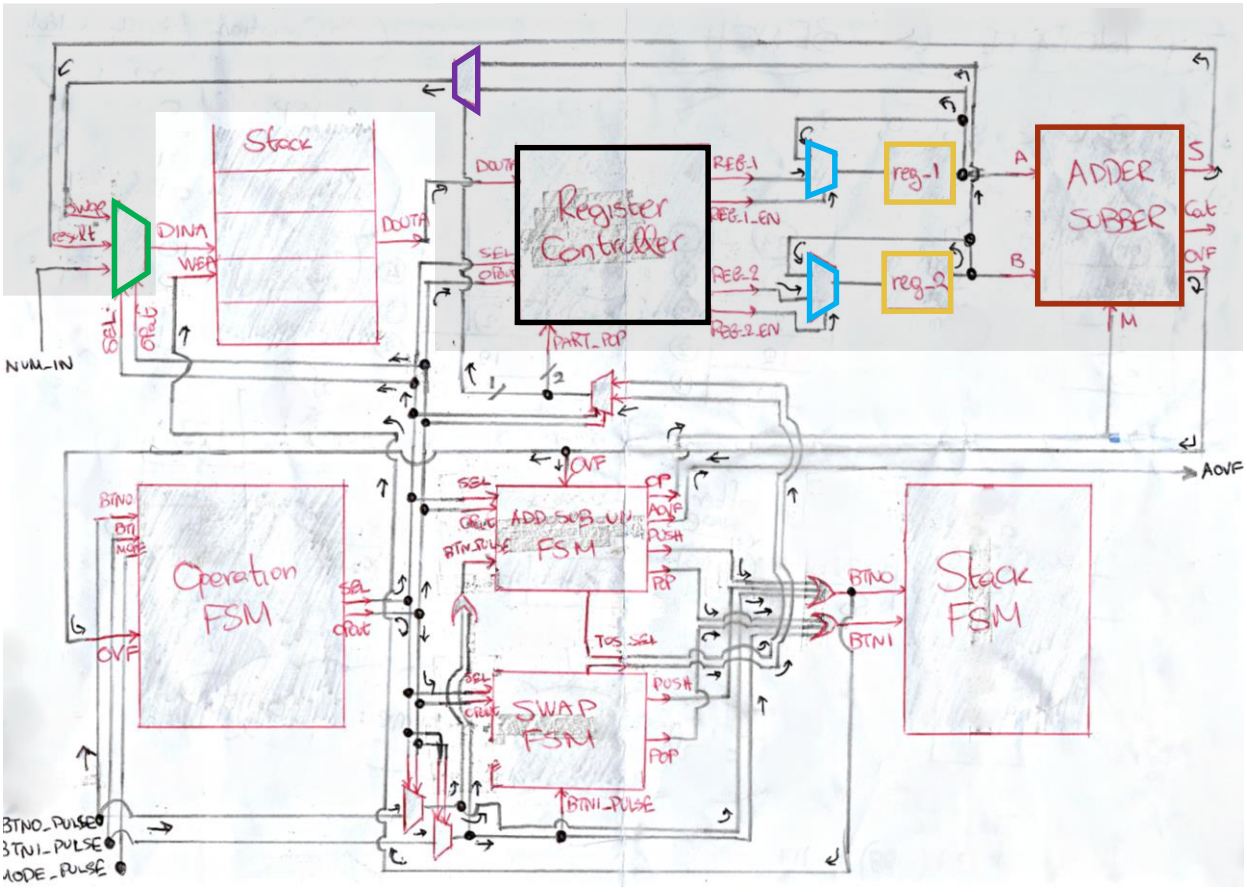
Παρατήρηση: Στις 2 παραπάνω FSM δεν αναγράφονται όλα τα σήματα για λόγους απλότητας. πχ για **RST='1'** η επόμενη κατάσταση θα είναι πάντα η **start.** Στη swap_FSM έπειτα απο έλεγχο με simulation, χρειάστηκε να προσθέσουμε 2 νέες καταστάσεις **wait$_1$** και **wait$_2$** για λόγους σωστού συγχρονισμού **(timings).**

DATA_PATH

Για την άσκηση αυτή, χρειαστήκαμε ένα **DataPath,** το οποίο περιλαμβάνει ένα σύνολο από modules, τα οποία διαχειρίζονται τη λογική της πράξης και περιλαμβάνουν τα:

- **Registers,** στα οποία αποθηκεύονται τα **TOS** και **TOS-1** της στοίβας.
- **Register Controller,** το οποίο δέχεται την έξοδο της στοίβας και τη διανέμει στα κατάλληλα register enablers.
- **Register Enablers,** τα οποία ανάλογα με μια συνθήκη, επιλέγουν είτε οι registers να πάρουν μια συγκεκριμένη τιμή είτε να επαναληφθεί η προηγούμενή τους (να θυμούνται).
- **Adder_Subber,** ο οποίος δέχεται 2 τιμές A, B (απο τους registers) και εκτελεί τη κατάλληλη πράξη που του έχει δοθεί. Το κύκλωμα αυτό είναι structural και αποτελεί επέκταση του 2ου εργαστηρίου.
- **Multiplexer_Pre_Stack,** ο οποίος περνάει τον κατάλληλο 8-Bit αριθμό στη Stack.
- **Multiplexer_Swap,** ο οποίος είναι αναγκαίος για τη πράξη της swap.

Το module του **DataPath** παρουσιάζεται παρακάτω στο γραμμοσκιασμένο τμήμα του **Stack Controller**:

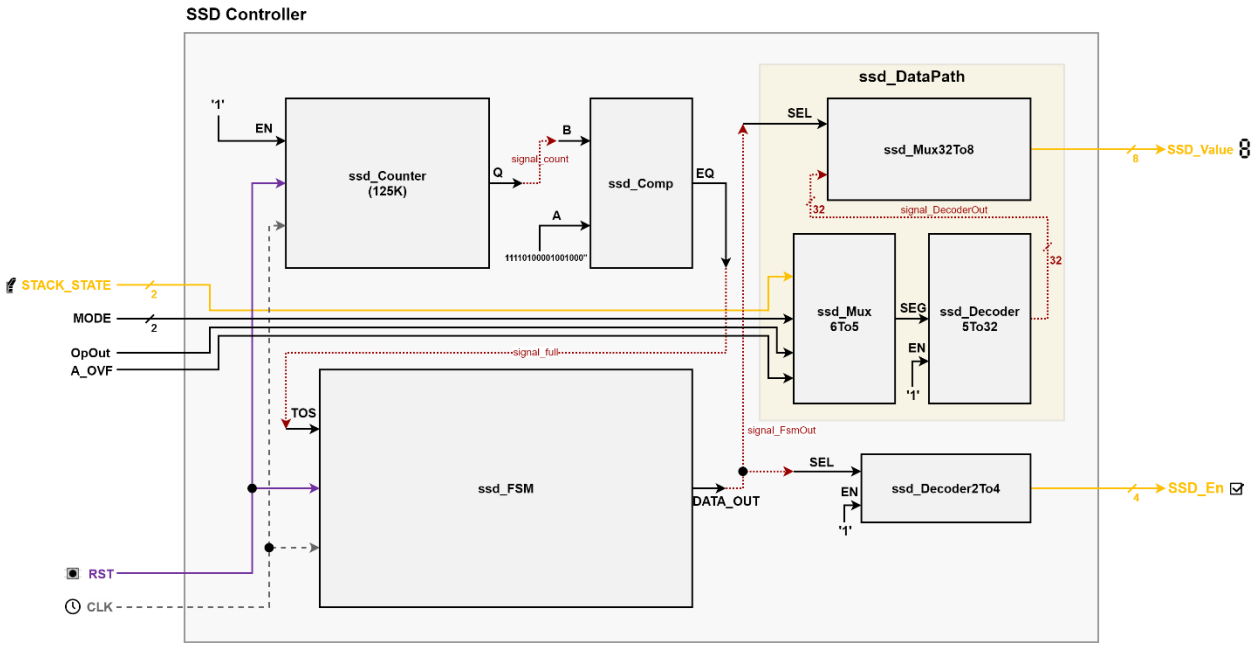


Παρατήρηση 1: Δε φαίνονται όλα τα τμήματα/σήματα του **Stack Controller** για αποφυγή πολυπλοκότητας.

Παρατήρηση 2: Για την υλοποίησή μας, χρησιμοποιούμε σήματα **SEL** και **OpOut** για την επιλογή των modes και είναι κρίσιμα για τη σωστή λειτουργία των δύο καινούργιων FSM. Πιο συγκεκριμένα, δίνονται στον παρακάτω πίνακα οι λειτουργίες ανάλογα με τις τιμές τους:

Operation	SEL	OpOut
push	00	0
pop		1
add	01	0
sub		1
unary sub	10	0
swap		1

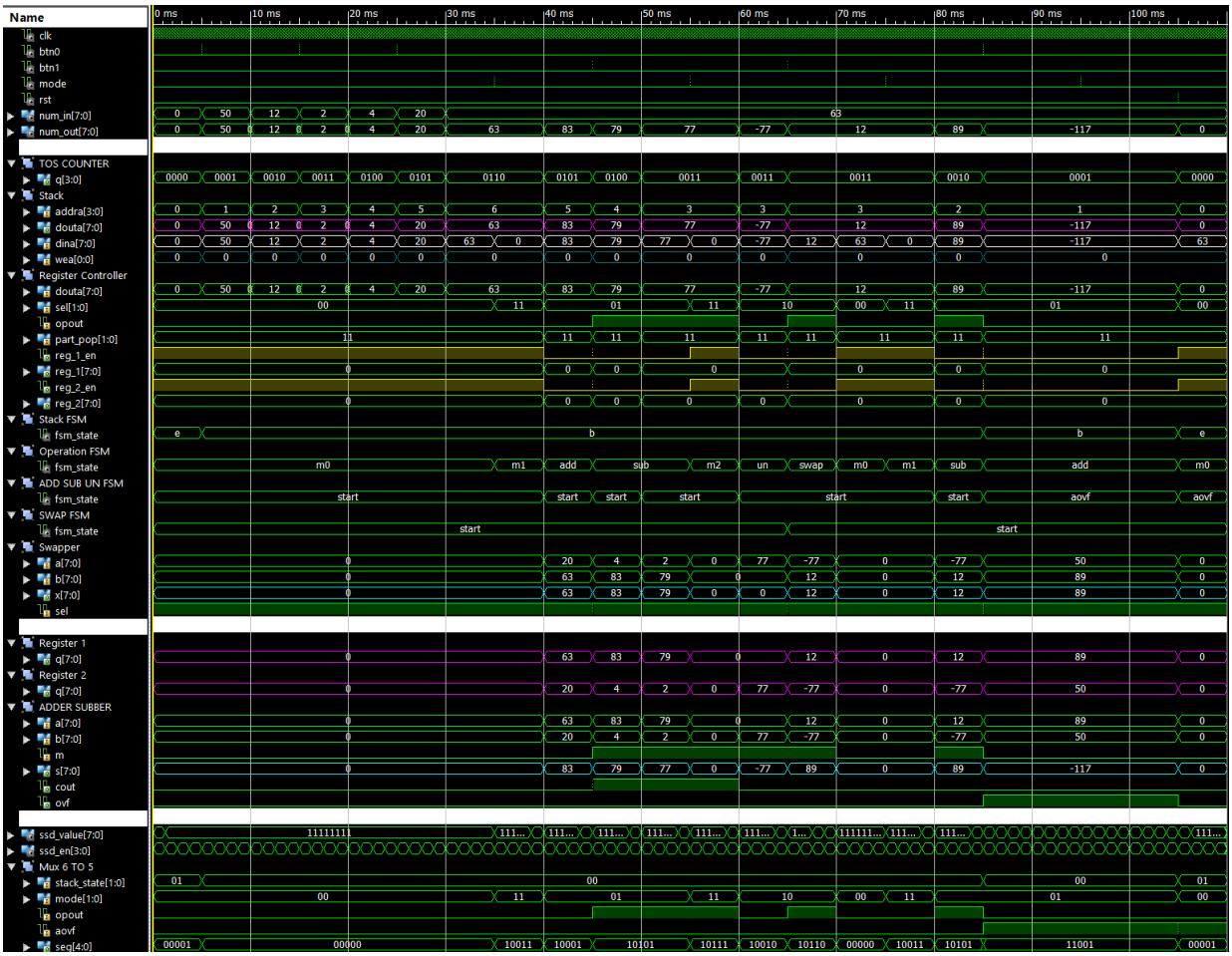
Η λειτουργία του **SSD Controller** δεν έχει αλλάξει, προστεθηκε απλώς το σήμα **AOVF** στον πολυπλέκτη του **SSD_DataPath.**



ΚΥΜΑΤΟΜΟΡΦΕΣ

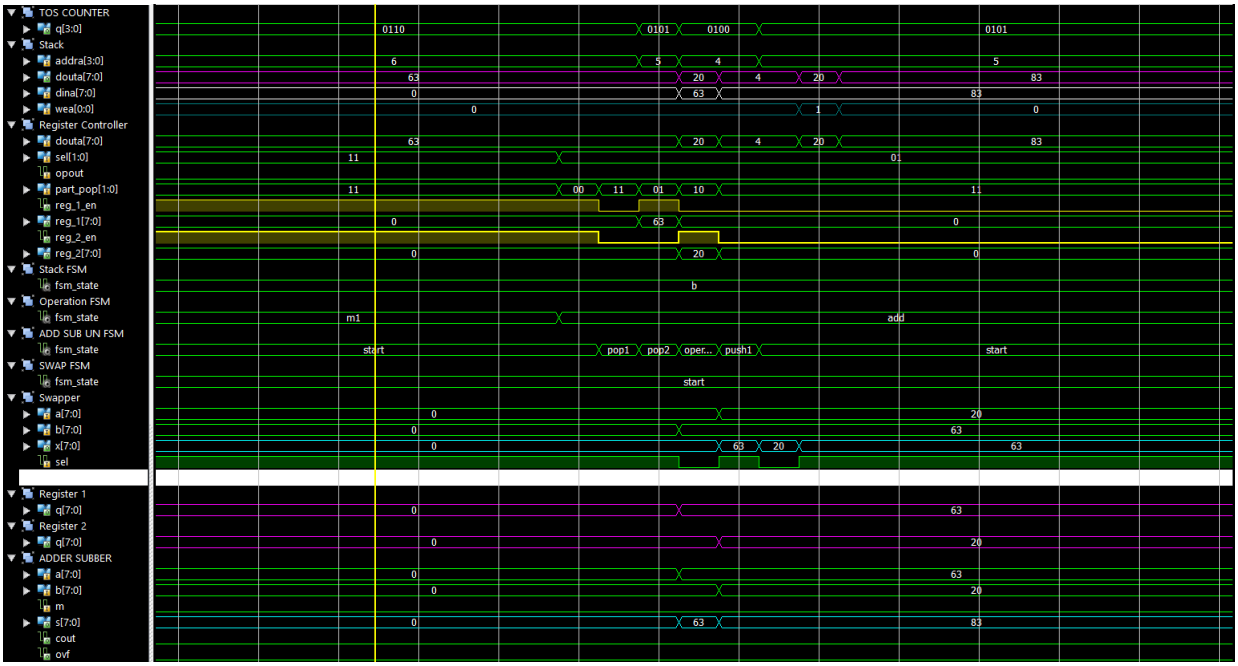
Παρακάτω φαίνεται γενικά η συμπεριφορά του κυκλώματος μας με τις παρκάτω ενέργειες (με τη σειρά):

- **[0 – 35ms]** : Εκτέλεση **push** για τους αριθμούς **50, 12, 2, 4, 20, 63.**
- **[40 – 110ms]** : Εκτέλεση **add, sub, sub, un, swap, sub, add** ανα 5 ms (με αλλαγή του mode όπου χρειάζεται).

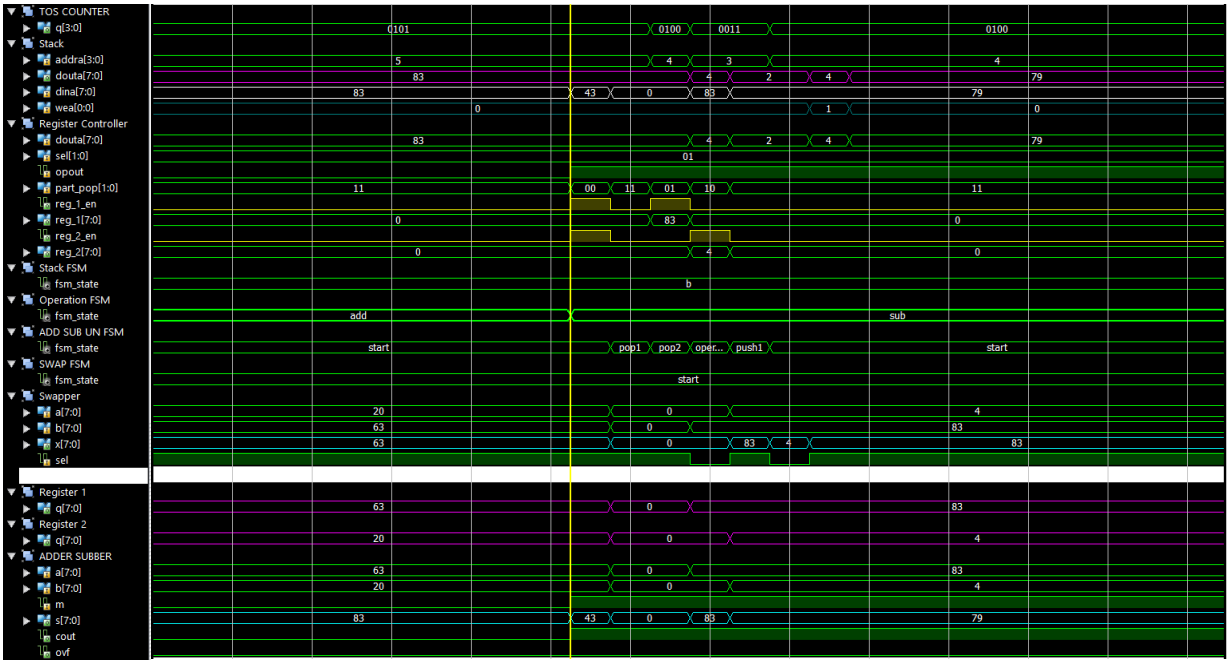


Για να ελέγξουμε πιο συγκεκριμένα την κάθε πράξη, μεγενθύνουμε τη κατάλληλη στιγμή.

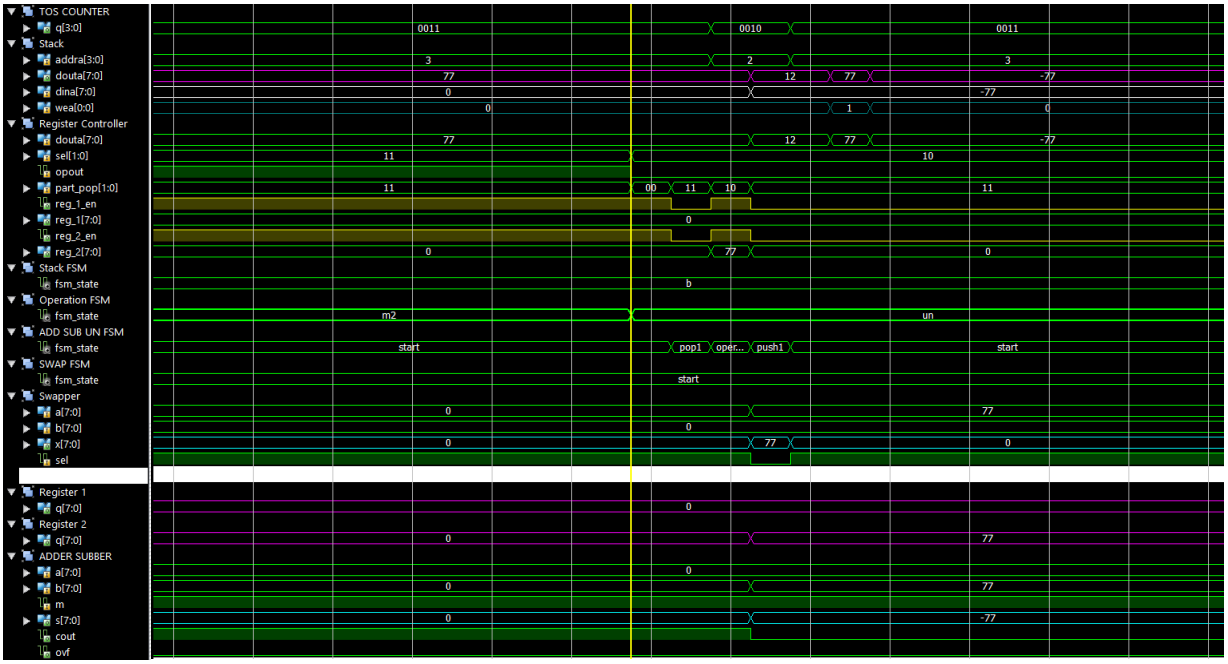
Για τη πράξη της **add** των αριθμών (63 + 20) βλέπουμε ότι αρχικά οι 2 προσθετέοι βρίσκονται στα addresses 6 (TOS) και 5 (TOS – 1) αντίστοιχα. Με το πρώτο pop το 63 μεταβαίνει στον register_1 και με το δεύτερο pop το 20 στον register_2. Η πράξη της πρόσθεσης εκτελείται από τον **adder_subber** και το αποτέλεσμα 83 αποθηκεύεται (μόλις το **WEA = 1**) στο address 5 (TOS).



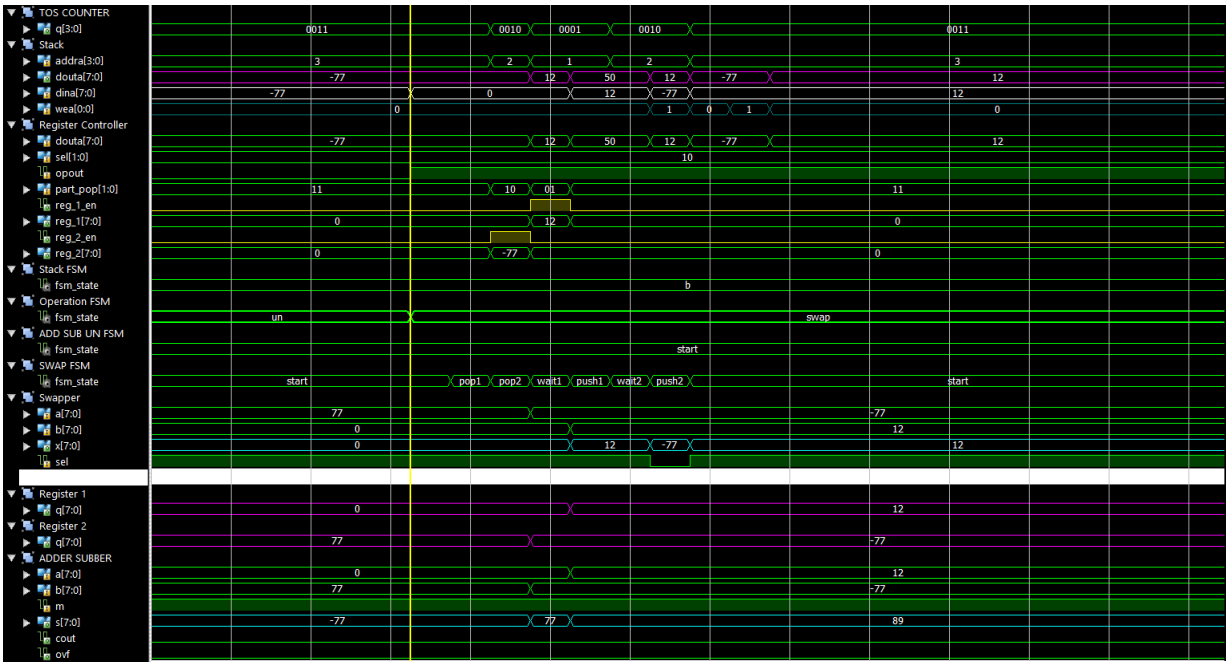
Για τη πράξη της **sub** των αριθμών (83 - 4) βλέπουμε ότι αρχικά οι 2 αριθμοί βρίσκονται στα addresses 5 (TOS) και 4 (TOS – 1) αντίστοιχα. Με το πρώτο pop το 83 μεταβαίνει στον register_1 και με το δεύτερο pop το 4 στον register_2. Η πράξη της αφαίρεσης εκτελείται από τον **adder_subber** και το αποτέλεσμα 79 αποθηκεύεται (μόλις το **WEA = 1**) στο address 4 (TOS).



Για τη πράξη της **un** του αριθμού 77 στη θέση 3, με pop ο register_1 παίρνει την τιμή 0 και το 77 περνάει στον register_2. Η πράξη της αφαίρεσης εκτελείται από τον **adder_subber** και το αποτέλεσμα -77 αποθηκεύεται (μόλις το **WEA = 1**) στο address 4 (TOS).



Για τη πράξη της **swap** των αριθμών (-77 και 12) βλέπουμε ότι αρχικά οι 2 αριθμοί βρίσκονται στα addresses 3 (TOS) και 2 (TOS – 1) αντίστοιχα. Με το πρώτο pop το -77 μεταβαίνει στον register_2 και με το δεύτερο pop το 12 στον register_1. Οι τιμές των 2 registers μεταφέρονται στον **Multiplexer_Swap** ο οποίος με τη σωστή σειρά τα αποθηκεύει στη στοίβα το -77 στη θέση 2 και το 12 στη θέση 3.



ΣΥΜΠΕΡΑΣΜΑΤΑ

Στο 5ο και τελευταίο εργαστήριο κατασκευάσαμε μια αριθμομηχανή με μέθοδο RPN. Κατα τη διάρκεια των 5 εργαστηριακών ασκήσεων μάθαμε να αντιμετωπίζουμε προβλήματα σε hardware και είμαστε πλέον σε θέση να σχεδιάζουμε πολύπλοκα κυκλώματα σε γλώσσα VHDL χρησιμοποιώντας behavioural και structural design.

ΠΑΡΑΡΤΗΜΑ – ΚΩΔΙΚΑΣ

Register Controller

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RegisterController is
    Port ( DOUTA : in  STD_LOGIC_VECTOR (7 downto 0);
           SEL : in  STD_LOGIC_VECTOR (1 downto 0);
           OpOut : in  STD_LOGIC;
           PART_POP : in  STD_LOGIC_VECTOR (1 downto 0);
           REG_1 : out  STD_LOGIC_VECTOR (7 downto 0);
           REG_1_EN : out  STD_LOGIC;
           REG_2 : out  STD_LOGIC_VECTOR (7 downto 0);
           REG_2_EN : out  STD_LOGIC);
end RegisterController;

architecture Behavioral of RegisterController is

begin


   Process (DOUTA, SEL, OPOut, PART_POP)
   begin

      if (SEL = "01") then
         if (PART_POP = "00") then
            REG_1 <= "00000000";
            REG_1_EN <= '1';
            REG_2 <= "00000000";
            REG_2_EN <= '1';
         elsif (PART_POP = "01") then
            REG_1 <= DOUTA;
            REG_1_EN <= '1';
            REG_2 <= "00000000";
            REG_2_EN <= '0';
         elsif (PART_POP = "10") then
            REG_1 <= "00000000";
            REG_1_EN <= '0';
            REG_2 <= DOUTA;
            REG_2_EN <= '1';
         else
            REG_1 <= "00000000";
            REG_1_EN <= '0';
            REG_2 <= "00000000";
            REG_2_EN <= '0';
         end if;
      elsif (SEL = "10") then
         if (OpOut = '0') then
               if (PART_POP = "00") then
                  REG_1 <= "00000000";
                  REG_1_EN <= '1';
                  REG_2 <= "00000000";
                  REG_2_EN <= '1';
               elsif (PART_POP = "01") then
                  REG_1 <= DOUTA;
                  REG_1_EN <= '1';
                  REG_2 <= "00000000";
                  REG_2_EN <= '0';
               elsif (PART_POP = "10") then
                  REG_1 <= "00000000";
                  REG_1_EN <= '1';
                  REG_2 <= DOUTA;
                  REG_2_EN <= '1';
               else
                  REG_1 <= "00000000";
                  REG_1_EN <= '0';
                  REG_2 <= "00000000";
                  REG_2_EN <= '0';
               end if;
         else
            if (PART_POP = "00") then
               REG_1 <= "00000000";
               REG_1_EN <= '1';
               REG_2 <= "00000000";
               REG_2_EN <= '1';
            elsif (PART_POP = "01") then
               REG_1 <= DOUTA;
               REG_1_EN <= '1';
               REG_2 <= "00000000";
               REG_2_EN <= '0';
            elsif (PART_POP = "10") then
               REG_1 <= "00000000";
               REG_1_EN <= '0';
               REG_2 <= DOUTA;
               REG_2_EN <= '1';
            else
               REG_1 <= "00000000";
               REG_1_EN <= '0';
               REG_2 <= "00000000";
               REG_2_EN <= '0';
            end if;
         end if;
      else
         REG_1 <= "00000000";
         REG_1_EN <= '1';
         REG_2 <= "00000000";
         REG_2_EN <= '1';
      end if;

   end Process;
```

Register

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Stack_Register is
    Port ( D : in  STD_LOGIC_VECTOR (7 downto 0);
           CLK : in  STD_LOGIC;
           Q : out  STD_LOGIC_VECTOR (7 downto 0));
end Stack_Register;

architecture Behavioral of Stack_Register is

begin

Process
begin

   Wait until CLK'Event and CLK = '1';
   Q <= D;

end Process;

end Behavioral;
```

DataPath

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Stack_DataPath is
    Port ( NUM_IN : in  STD_LOGIC_VECTOR (7 downto 0);
           DOUTA : in  STD_LOGIC_VECTOR (7 downto 0);
           SEL : in  STD_LOGIC_VECTOR (1 downto 0);
           OPOut : in  STD_LOGIC;
           PART_POP : in  STD_LOGIC_VECTOR (1 downto 0);
           M : in  STD_LOGIC;
           CLK : in STD_LOGIC;
           Cout : out STD_LOGIC;
           DINA : out  STD_LOGIC_VECTOR (7 downto 0);
           OVF : out  STD_LOGIC);
end Stack_DataPath;

architecture Structural of Stack_DataPath is

--Sorted based on bus width
signal signal_result_adder, signal_swapped : STD_LOGIC_VECTOR (7 downto 0);
signal signal_pre_register_1, signal_pre_register_2   : STD_LOGIC_VECTOR (7 downto 0);
signal signal_register_1, signal_register_2 : STD_LOGIC_VECTOR (7 downto 0);
signal signal_post_register_1, signal_post_register_2  : STD_LOGIC_VECTOR (7 downto 0);
signal signal_register_enabler_1, signal_register_enabler_2 : STD_LOGIC;

signal signal_ff, signal_ff2 : STD_LOGIC;

Component Mux_Pre_Stack is
    Port ( OP_RESULT : in  STD_LOGIC_VECTOR (7 downto 0);
           SWAP : in  STD_LOGIC_VECTOR (7 downto 0);
           NUM_IN : in  STD_LOGIC_VECTOR (7 downto 0);
           SEL : in  STD_LOGIC_VECTOR (1 downto 0);
           OPOut : in  STD_LOGIC;
           X : out  STD_LOGIC_VECTOR (7 downto 0));
end Component;

Component Stack_Register is
    Port ( D : in  STD_LOGIC_VECTOR (7 downto 0);
           CLK : in  STD_LOGIC;
           Q : out  STD_LOGIC_VECTOR (7 downto 0));
end Component;

Component ADDER_SUBBER is
    Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
           B : in  STD_LOGIC_VECTOR (7 downto 0);
           M : in STD_LOGIC;
           S : out  STD_LOGIC_VECTOR (7 downto 0);
           Cout : out  STD_LOGIC;
           OVF : out  STD_LOGIC);
end Component;

Component Mux_Swapper is
    Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
           B : in  STD_LOGIC_VECTOR (7 downto 0);
           X : out  STD_LOGIC_VECTOR (7 downto 0);
           SEL : in  STD_LOGIC);
end Component;


Component RegisterController is
    Port ( DOUTA : in  STD_LOGIC_VECTOR (7 downto 0);
           SEL : in  STD_LOGIC_VECTOR (1 downto 0);
           OpOut : in  STD_LOGIC;
           PART_POP : in  STD_LOGIC_VECTOR (1 downto 0);
           REG_1 : out  STD_LOGIC_VECTOR (7 downto 0);
           REG_1_EN : out  STD_LOGIC;
           REG_2 : out  STD_LOGIC_VECTOR (7 downto 0);
           REG_2_EN : out  STD_LOGIC);
end Component;

Component FlipFlop is
    Port ( D : in  STD_LOGIC;
           CLK : in STD_LOGIC;
           Q : out  STD_LOGIC);
end Component;

Component Double_FlipFlop is
    Port ( D : in  STD_LOGIC;
           CLK : in STD_LOGIC;
           Q : out  STD_LOGIC);
end Component;
```

```
82
83  begin
84
85  Mux_Pre_Stack_1: Mux_Pre_Stack port map( OP_RESULT => signal_result_adder,
86                                          SWAP => signal_swapped,
87                                          NUM_IN => NUM_IN,
88                                          SEL => SEL,
89                                          OPOut => OPOut,
90                                          X => DINA);
91
92  Controller : RegisterController port map (DOUTA => DOUTA,
93                                          SEL => SEL,
94                                          OpOut => OpOut,
95                                          PART_POP => PART_POP,
96                                          REG_1 => signal_pre_register_1,
97                                          REG_1_EN => signal_register_enabler_1,
98                                          REG_2 => signal_pre_register_2,
99                                          REG_2_EN => signal_register_enabler_2
100                                            );
101
102
103 Register_Enabler_1 : Mux_Swapper port map( A => signal_post_register_1,
104                                           B => signal_pre_register_1,
105                                           X => signal_register_1,
106                                           SEL => signal_register_enabler_1);
107
108 Stack_Register_1: Stack_Register port map(CLK => CLK,
109                                         D => signal_register_1,
110                                         Q => signal_post_register_1);
111
112
113
114
115 Register_Enabler_2 : Mux_Swapper port map( A => signal_post_register_2,
116                                           B => signal_pre_register_2,
117                                           X => signal_register_2,
118                                           SEL => signal_register_enabler_2);
119
120 Stack_Register_2: Stack_Register port map(CLK => CLK,
121                                         D => signal_register_2,
122                                         Q => signal_post_register_2);
123
124
125
126
127 ADDER_SUBBER_1: ADDER_SUBBER port map(A => signal_post_register_1,
128                                       B => signal_post_register_2,
129                                       M => M,
130                                       S => signal_result_adder,
131                                       Cout => Cout,
132                                       OVF => OVF
133                                       );
134
135 FF : Double_FlipFlop port map ( D=> PART_POP(1),
136                                 CLK => CLK,
137                                 Q => signal_ff);
138
139
140 FF2 : FlipFlop port map ( D=> signal_ff,
141                                 CLK => CLK,
142                                 Q => signal_ff2);
143
144 Swapper : Mux_Swapper port map (  A => signal_post_register_2,
145                                   B => signal_post_register_1,
146                                   X => signal_swapped,
147                                   SEL => signal_ff2);
148
149
150 end Structural;
```