2024

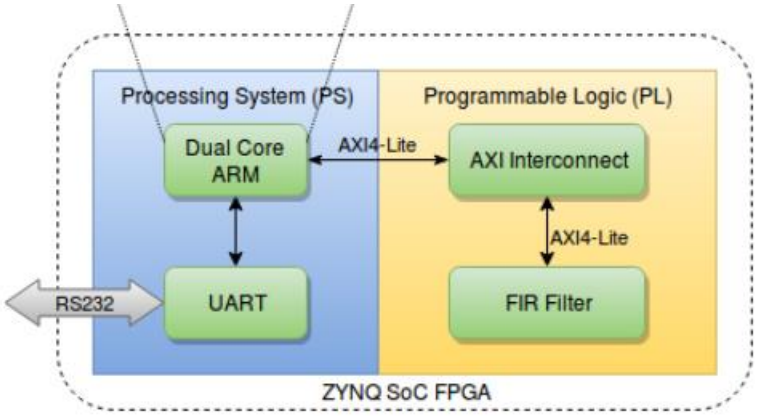
# Αναφορά 5ης εργαστηριακής Άσκησης

***Εργαστήριο VLSI***

ΠΑΠΑΔΟΠΟΥΛΟΣ ΣΠΥΡΙΔΩΝ (ΑΜ):03120033
ΕΜΜΑΝΟΥΗΛ ΞΕΝΟΣ (ΑΜ):03120850

## Υλοποίηση FIR φίλτρου με AXI διεπαφή σε ZYNQ SoC FPGA

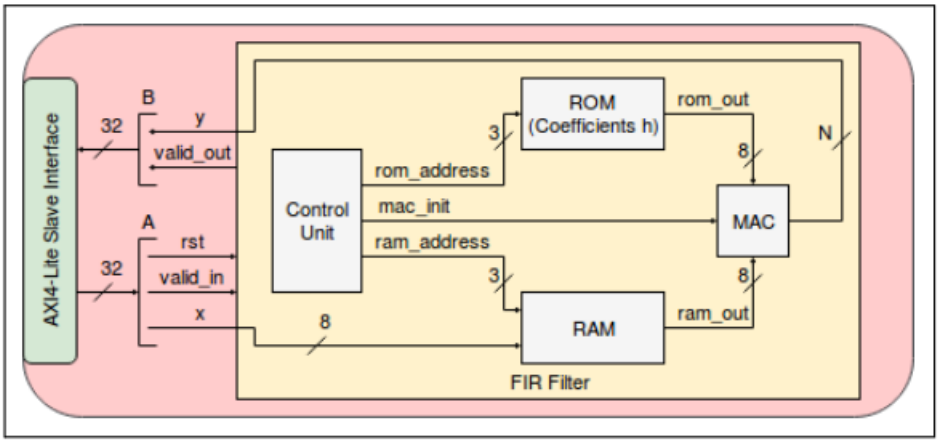
Στην άσκηση αυτή μας ζητήθηκε να εκμεταλλευτούμε την υλοποίηση του FIR που πραγματοποιήσαμε στο προηγούμενο εργαστήριο και χρησιμοποιώντας την διεπαφή AXI4-LIte να προγραμματίσουμε την αναπτυξιακή πλακέτα ZYBO , ώστε να υλοποιεί το φίλτρο , στο οποίο τα δεδομένα θα αποστέλλονται από τον ενσωματωμένο ARM επεξεργαστή που διαθέτει η πλακέτα. Ομοίως τα αποτελέσματα θα στέλνονται από το FPGA στον ARM και θα προβάλλονται στο terminal του SDK. Προφανώς για να γίνει αυτό απαιτείται ένα Hardware/Software Co-Design. Κάτι τέτοιο μας επιτρέπει να κάνουμε το Vivado χρησιμοποιώντας το κατάλληλο πρωτόκολλο επικοινωνίας, εν προκειμένου το AXI. Αρχικά θα πραγματοποιήσουμε μία θεωρητική ανάλυση του προβλήματος για βαθύτερη κατανόηση και στην συνέχεια θα παρουσιάσουμε αναλυτικά τα βήματα της υλοποίησης. Το σύστημα μας αποτελείται από δύο βασικά μέρη το PS (Processing System) και το PL ( Programmable logic).




**Εικόνα 2 :** Αρχιτεκτονική συστήματος.

Όπως φαίνεται και από την εικόνα το PS ουσιαστικά είναι ο ARM επεξεργαστής που διαθέτει η πλακέτα μας . Το PS παίρνει το input μέσω UART επικοινωνίας ( ξεφεύγει από τα όρια του μαθήματος η ανάλυση αυτού του πρωτοκόλλου επικοινωνίας, αναλύθηκε στο μάθημα των μικροϋπολογιστών ) και στην συνέχεια επικοινωνεί με το PL μέσω του πρωτοκόλλου AXI4-LIte. Εδώ είναι σημαντικό να σημειώσουμε ένα από τα κυριότερα προβλήματα που δημιουργούν την ανάγκη για την ύπαρξη ενός τέτοιου πρωτοκόλλου και αυτό είναι ο συγχρονισμός! Το PS δεν δουλεύει απαραίτητα ( προφανώς) με την ίδια συχνότητα με το PL ( το PL είναι σαφώς ταχύτερο) και αυτό δημιουργεί προβλήματα συγχρονισμού. Αυτά τα προβλήματα λύνει το AXI , εισάγοντας μία πληθώρα σημάτων τα οποία ουσιαστικά υλοποιούν επικοινωνία όπου τόσο ο παραλήπτης όσο και ο αποδέκτης ενημερώνουν με ready και valid σήματα την δυνατότητα που έχουν να διαβάσουν ή να παρέχουν μία ορθή έξοδο . Παράλληλα, υπάρχουν και διάφορα σήματα που υλοποιούν handshakes κατά την επικοινωνία των δύο συστημάτων, δηλαδή το ένα ενημερώνει το άλλο αν έλαβε ορθά ένα σήμα και ανάλογα αν έλαβε την επιβεβαίωση ή όχι. Στην περίπτωση μας ο καθορισμός όλων αυτών των σημάτων δεν είναι απαραίτητος όπως θα δούμε στην συνέχεια, καθώς το AXI4-LIte με την χρήση καταχωρητών, παρέχει κάποιες ευκολίες για την ανάγνωση και την εγγραφή, αλλά αυτά θα τα δούμε αναλυτικότερα στην συνέχεια. Είναι προφανές πλέον λοιπόν ότι η επικοινωνία μεταξύ PL και PS απαιτεί την ύπαρξη αυτού του πρωτοκόλλου, ώστε να επιτευχθεί ο συγχρονισμός . Οπότε στο πλαίσιο της εργασίας εμείς κληθήκαμε να υλοποιήσουμε το πρωτόκολλο στο FPGA χρησιμοποιώντας VHDL. Το AXI4-LIte που αναφέραμε είναι μία διεπαφή που υλοποιεί το πρωτόκολλο. Η ανάγνωση και η εγγραφή γίνεται σε 4 καταχωρητές εισόδου και εξόδου. Στο πλαίσιο της εργασίας απαιτείται μόνο 1 καταχωρητής για είσοδο και 1 για έξοδο, οπότε η διαδικασία απλοποιείται. Το Vivado παρέχει την default υλοποίηση του πρωτοκόλλου ( ουσιαστικά δεν υπάρχει κανένας συγχρονισμός απλώς ορίζονται τα σήματα εισόδου/ εξόδου και το διάβασμα και η εγγραφή στους καταχωρητές εισόδου και εξόδου) και εμείς καλούμαστε να υλοποιήσουμε ουσιαστικά την ανάγνωση των δεδομένων και την εγγραφή του αποτελέσματος στους κατάλληλους καταχωρητές ( 1 για είσοδο, 1 για έξοδο), καθώς και τον συγχρονισμό αυτών των διαδικασιών. Εφόσον αναλύσαμε τα απολύτως βασικά για την σημασία του AXI και τι καλούμαστε να υλοποιήσουμε στην συνέχεια θα περάσουμε στην δική μας υλοποίηση και την ανάλυση αυτής.

Το FIR μας στο σύστημα που περιγράψαμε έχει αυτή την αρχιτεκτονική:




**Εικόνα 1 :** Αρχιτεκτονική FIR φίλτρου με διεπαφή AXI4-Lite.

Το FIR ουσιαστικά διαβάζει και γράφει στους slave registers του πρωτοκόλλου. Συνεπώς, μπορούμε να τροποποιήσουμε ελάχιστα το FIR μας, ώστε να διευκολύνουμε τον συγχρονισμό. Συγκεκριμένα όπως αναφέραμε το κύριο πρόβλημα είναι ότι το PS έχει σημαντικά μικρότερη συχνότητα από το PL. Συνεπώς, το PL μπορεί να ολοκλήρωνε τον υπολογισμό της εξόδου του φίλτρου, όμως ακόμα να μην είχε αλλάξει η τιμή εισόδου του παρέχει το PS, δηλαδή θα είχαμε ακόμα valid_in=1 και είσοδο ίδια με την προηγούμενη. Η λύση για αυτό θα μπορούσε να έχει δύο οδούς είτε να εισάγουμε έναν επιπλέον buffer εντός του AXI( όπως θα κάνουμε αργότερα για την αντιμετώπιση του ανάλογου προβλήματος με την έξοδο) ή να τροποποιήσουμε το FIR. Επιλέξαμε το δεύτερο, καθώς απαιτούσε δύο απλές αλλαγές. Η πρώτη απαιτεί ένα συμβιβασμό που θα αναγκαστούμε να κάνουμε λόγω του προβλήματος συγχρονισμού. Για να δεχτούμε μία νέα είσοδο το valid_in θα πρέπει να έχει μεταβεί από 0->1 . Αυτό είναι απαραίτητο, ώστε να μην παίρνουμε πολλές φορές την ίδια είσοδο. Αυτό εύκολα μπορούμε να το υλοποιήσουμε στο Control_unit εισάγοντας μία τιμή prev_valid_in η οποία θα κοιτά το προηγούμενο valid_in και αν αυτό ήταν 0 τότε θα επιτρέπει την εγγραφή μίας νέας εισόδου. Επιπλέον ένα πρόβλημα που έχουμε είναι ότι το valid_out το θέσαμε ως το mac_init(που γίνεται 1 όταν έχουμε ορθή είσοδο) καθυστερημένο κατά 9 κύκλους, καθώς κάτι τέτοιο μας διευκόλυνε στην προηγούμενη τροποποίηση( δηλαδή να μην υπολογίζουμε στο control unit το valid_out). Ωστόσο αυτό οδήγησε στην ανάγκη να κάνουμε ένα gating του σήματος που θα μας δώσει το valid_out με το rst, ώστε αν κατά την διάρκεια κάποιου υπολογισμού έρθει reset να αναιρεθεί το valid_out. Επιπλέον, εφόσον πλέον περνάμε το valid_in από το control_unit και ελέγχουμε όλα τα υπόλοιπα MAC και we της RAM με το mac_init εισάγαμε ένα κύκλο καθυστέρηση στην πρώτη λειτουργία του FIR μας ( δηλαδή στην συνεχή λειτουργία βγάζει αποτέλεσμα ανά 8 κύκλους όμως αν δεν είμαστε σε συνεχή λειτουργία για το πρώτο αποτέλεσμα απαιτούνται 10 κύκλοι αντί για 9 που είχαμε στην προηγούμενη άσκηση) . Για αυτό τον λόγο καθυστερήσαμε κατά 1 κύκλο και την είσοδο X. Αυτές ήταν οι τροποποιήσεις που κάναμε στο FIR και για για αυτό παραθέτουμε ακολούθως τα αντίστοιχα component που τροποποιήθηκαν σε σχέση με την προηγούμενη άσκηση που υλοποιήσαμε πάλι το FIR.

Το Control unit:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;

entity Control_Unit is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           valid_in : in STD_LOGIC;
           rom_address : out std_logic_vector(2 downto 0) := "000";
           ram_address : out std_logic_vector(2 downto 0) := "000";
           mac_init : out STD_LOGIC :='1');
end Control_Unit;
```

```
architecture Behavioral of Control_Unit is
    signal counter : std_logic_vector (2 downto 0);
    signal prev_valid_in : std_logic := '0' ;
begin
create_output:  process (clk,rst)
                  begin
                  if rst = '1' then
                       counter <= "000";
                       mac_init <= '1';
                       rom_address<= "000";
                       ram_address <= "000";
                  elsif clk'event and clk = '1' then
                              prev_valid_in <= valid_in;
                              if counter = "000" and valid_in ='1' and prev_valid_in='0'
then
                                  mac_init <= '1';
                                  rom_address <= "000";
                                  ram_address <= "000";
                                  counter <= "001";
                              elsif counter/="000" then
                                  mac_init <= '0';
                                  rom_address <= COUNTER;
                                  ram_address <= counter;
                                  if counter = "111" then
                                      counter <= "000";
                                  else
                                      counter <= counter +"001";
                                  end if;

                               end
if;

                           end if;
               end process create_output;
end Behavioral;
```

Όπως αναφέραμε προσθέσαμε το prev_valid_in και τον αντίστοιχο έλεγχο για την επίτευξη του συγχρονισμού.

Στην συνέχεια παραθέτουμε το FIR

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

entity FIR is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           valid_in : in STD_LOGIC;
           X : in STD_LOGIC_VECTOR (7 downto 0);
           valid_out : out STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (18 downto 0));
end FIR;
```

```
architecture Behavioral of FIR is
    component D_flip_flop_N_seq is
        generic (
            N : positive
        );
        Port (
            Input : in std_logic ;
            Clk : in std_logic;
            Output : out std_logic
        );
    end component;
    component D_flip_flop_1_bit is
        Port ( Input : in std_logic;
               Clk : in std_logic ;
               Output : out std_logic);
    end component;
    component D_flip_flop_8_bit is
        Port ( Input : in std_logic_vector (7 downto 0);
               Clk : in std_logic ;
               Output : out std_logic_vector (7 downto 0));
    end component;
    component Control_Unit is
        Port ( clk : in STD_LOGIC;
               rst : in STD_LOGIC;
               valid_in : in STD_LOGIC;
               rom_address : out std_logic_vector(2 downto 0);
               ram_address : out std_logic_vector(2 downto 0);
               mac_init : out STD_LOGIC);
    end component;
    component mlab_rom is
             generic (
                coeff_width : integer :=8                  --- width of coefficients
(bits)
             );
            Port ( clk : in  STD_LOGIC;
                   en : in  STD_LOGIC;                 --- operation enable
                   addr : in  STD_LOGIC_VECTOR (2 downto 0);            -- memory address
                   rom_out : out  STD_LOGIC_VECTOR (coeff_width-1 downto 0));    -- output
data
        end component;
    component mlab_ram is
                generic (
                   data_width : integer :=8                --- width of data (bits)
                );
                port (clk  : in std_logic;
                      we   : in std_logic;                        --- memory write enable
                      en   : in std_logic;                --- operation enable
                      reset   : in std_logic;
                      addr : in std_logic_vector(2 downto 0);            -- memory address
                      di   : in std_logic_vector(data_width-1 downto 0);        -- input
data
                      do   : out std_logic_vector(data_width-1 downto 0));        --
output data
            end component;
```

```
    component MAC is
                    Port ( mac_init : in std_logic;
                           clk : in std_logic;
                           B : in STD_LOGIC_VECTOR (7 downto 0);
                           C : in STD_LOGIC_VECTOR (7 downto 0);
                           Y : out STD_LOGIC_VECTOR (18 downto 0));
     end component;
    signal rom_address,ram_address : std_logic_vector ( 2 downto 0);
    signal rom_out,ram_out,delayed_X: std_logic_vector (7 downto 0);
    signal mac_init,delayed_mac_init,delayed_reset,out_valid,valid_to_delay: std_logic;
begin
    control : Control_Unit port map (clk,rst,valid_in,rom_address,ram_address,mac_init);
    flip_1_bit : D_flip_flop_1_bit port map (mac_init,clk,delayed_mac_init);
    flip_1_bit_sec : D_flip_flop_1_bit port map (rst,clk,delayed_reset);
    flip_flop : D_flip_flop_8_bit port map (X,clk,delayed_X);
    valid_to_delay <= mac_init and not rst;
    flip_flops_delay : D_flip_flop_N_seq generic map (N => 9)  port map
(valid_to_delay,Clk,out_valid);
    valid_out <= out_valid and not delayed_reset;
    ROM : mlab_rom port map(clk,'1',rom_address,rom_out);
    RAM : mlab_ram port map(clk,mac_init,'1',rst,ram_address,delayed_X,ram_out);
    MAC_compon : MAC port map (delayed_mac_init,clk,rom_out,ram_out,Y);
end Behavioral;
```

Παραπάνω είναι η υλοποίηση του FIR με τα αντίστοιχα σημεία που τροποποιήσαμε σε σχέση με την προηγούμενη υλοποίηση.

Σε αυτό το σημείο τελειώσαμε με οποιαδήποτε τροποποίηση χρειάστηκε να κάνουμε στο FIR μας. Στην συνέχεια θα αναφέρουμε τις τροποποιήσεις και τις προσθήκες που κάναμε στην default υλοποίηση του AXI4-LIte που μας παρέχει το Vivado. Παραθέτουμε ολόκληρο τον κώδικα και όταν αναφερόμαστε σε κάποιο συγκεκριμένο κομμάτι θα το ξανά παραθέτουμε για μεγαλύτερη ευκολία στην ανάγνωση:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity myip_v1_0_FIR_AXI is
    generic (
        -- Users to add parameters here

        -- User parameters ends
        -- Do not modify the parameters beyond this line

        -- Width of S_AXI data bus
        C_S_AXI_DATA_WIDTH  : integer   := 32;
        -- Width of S_AXI address bus
        C_S_AXI_ADDR_WIDTH  : integer   := 4
    );
    port (
        -- Users to add ports here

        -- User ports ends
        -- Do not modify the ports beyond this line
```

```
        -- Global Clock Signal
        S_AXI_ACLK  : in std_logic;
        -- Global Reset Signal. This Signal is Active LOW
        S_AXI_ARESETN   : in std_logic;
        -- Write address (issued by master, acceped by Slave)
        S_AXI_AWADDR    : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
        -- Write channel Protection type. This signal indicates the
            -- privilege and security level of the transaction, and whether
            -- the transaction is a data access or an instruction access.
        S_AXI_AWPROT    : in std_logic_vector(2 downto 0);
        -- Write address valid. This signal indicates that the master signaling
            -- valid write address and control information.
        S_AXI_AWVALID   : in std_logic;
        -- Write address ready. This signal indicates that the slave is ready
            -- to accept an address and associated control signals.
        S_AXI_AWREADY   : out std_logic;
        -- Write data (issued by master, acceped by Slave)
        S_AXI_WDATA : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        -- Write strobes. This signal indicates which byte lanes hold
            -- valid data. There is one write strobe bit for each eight
            -- bits of the write data bus.
        S_AXI_WSTRB : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1 downto 0);
        -- Write valid. This signal indicates that valid write
            -- data and strobes are available.
        S_AXI_WVALID    : in std_logic;
        -- Write ready. This signal indicates that the slave
            -- can accept the write data.
        S_AXI_WREADY    : out std_logic;
        -- Write response. This signal indicates the status
            -- of the write transaction.
        S_AXI_BRESP : out std_logic_vector(1 downto 0);
        -- Write response valid. This signal indicates that the channel
            -- is signaling a valid write response.
        S_AXI_BVALID    : out std_logic;
        -- Response ready. This signal indicates that the master
            -- can accept a write response.
        S_AXI_BREADY    : in std_logic;
        -- Read address (issued by master, acceped by Slave)
        S_AXI_ARADDR    : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
        -- Protection type. This signal indicates the privilege
            -- and security level of the transaction, and whether the
            -- transaction is a data access or an instruction access.
        S_AXI_ARPROT    : in std_logic_vector(2 downto 0);
        -- Read address valid. This signal indicates that the channel
            -- is signaling valid read address and control information.
        S_AXI_ARVALID   : in std_logic;
        -- Read address ready. This signal indicates that the slave is
            -- ready to accept an address and associated control signals.
        S_AXI_ARREADY   : out std_logic;
        -- Read data (issued by slave)
        S_AXI_RDATA : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        -- Read response. This signal indicates the status of the
            -- read transfer.
```

```
        S_AXI_RRESP : out std_logic_vector(1 downto 0);
        -- Read valid. This signal indicates that the channel is
            -- signaling the required read data.
        S_AXI_RVALID    : out std_logic;
        -- Read ready. This signal indicates that the master can
            -- accept the read data and response information.
        S_AXI_RREADY    : in std_logic
    );
end myip_v1_0_FIR_AXI;

architecture arch_imp of myip_v1_0_FIR_AXI is
    constant CLK_PERIOD  : time := 0.05 ns; -- Clock period
    component FIR is
        Port ( clk : in STD_LOGIC;
               rst : in STD_LOGIC;
               valid_in : in STD_LOGIC;
               X : in STD_LOGIC_VECTOR (7 downto 0);
               valid_out : out STD_LOGIC;
               Y : out STD_LOGIC_VECTOR (18 downto 0));
    end component;
    signal fir_out_reg : std_logic_vector ( 31 downto 0);
    signal prev_valid_in : std_logic;
    -- AXI4LITE signals
    signal axi_awaddr   : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    signal axi_awready  : std_logic;
    signal axi_wready   : std_logic;
    signal axi_bresp    : std_logic_vector(1 downto 0);
    signal axi_bvalid   : std_logic;
    signal axi_araddr   : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    signal axi_arready  : std_logic;
    signal axi_rdata    : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal axi_rresp    : std_logic_vector(1 downto 0);
    signal axi_rvalid   : std_logic;

    -- Example-specific design signals
    -- local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
    -- ADDR_LSB is used for addressing 32/64 bit registers/memories
    -- ADDR_LSB = 2 for 32 bits (n downto 2)
    -- ADDR_LSB = 3 for 64 bits (n downto 3)
    constant ADDR_LSB  : integer := (C_S_AXI_DATA_WIDTH/32)+ 1;
    constant OPT_MEM_ADDR_BITS : integer := 1;
    ------------------------------------------------
    ---- Signals for user logic register space example
    ------------------------------------------------
    ---- Number of Slave Registers 4
    signal slv_reg0 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg1 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg2 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg3 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg_rden : std_logic;
    signal slv_reg_wren : std_logic;
    signal reg_data_out :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal byte_index   : integer;
    signal aw_en    : std_logic;
```

```
begin
    -- I/O Connections assignments

    S_AXI_AWREADY   <= axi_awready;
    S_AXI_WREADY    <= axi_wready;
    S_AXI_BRESP <= axi_bresp;
    S_AXI_BVALID    <= axi_bvalid;
    S_AXI_ARREADY   <= axi_arready;
    S_AXI_RDATA <= axi_rdata;
    S_AXI_RRESP <= axi_rresp;
    S_AXI_RVALID    <= axi_rvalid;
    -- Implement axi_awready generation
    -- axi_awready is asserted for one S_AXI_ACLK clock cycle when both
    -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
    -- de-asserted when reset is low.

    process (S_AXI_ACLK)
    begin
      if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
          axi_awready <= '0';
          aw_en <= '1';
        else
          if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1' and aw_en =
'1') then
            -- slave is ready to accept write address when
            -- there is a valid write address and write data
            -- on the write address and data bus. This design
            -- expects no outstanding transactions.
            axi_awready <= '1';
            elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then
                aw_en <= '1';
                axi_awready <= '0';
          else
            axi_awready <= '0';
          end if;
        end if;
      end if;
    end process;

    -- Implement axi_awaddr latching
    -- This process is used to latch the address when both
    -- S_AXI_AWVALID and S_AXI_WVALID are valid.

    process (S_AXI_ACLK)
    begin
      if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
          axi_awaddr <= (others => '0');
        else
          if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1' and aw_en =
'1') then
            -- Write Address latching
```

```
            axi_awaddr <= S_AXI_AWADDR;
          end if;
        end if;
      end if;
    end process;

    -- Implement axi_wready generation
    -- axi_wready is asserted for one S_AXI_ACLK clock cycle when both
    -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
    -- de-asserted when reset is low.

    process (S_AXI_ACLK)
    begin
      if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
          axi_wready <= '0';
        else
          if (axi_wready = '0' and S_AXI_WVALID = '1' and S_AXI_AWVALID = '1' and aw_en =
'1') then
              -- slave is ready to accept write data when
              -- there is a valid write address and write data
              -- on the write address and data bus. This design
              -- expects no outstanding transactions.
              axi_wready <= '1';
          else
            axi_wready <= '0';
          end if;
        end if;
      end if;
    end process;

    -- Implement memory mapped register select and write logic generation
    -- The write data is accepted and written to memory mapped registers when
    -- axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write strobes
are used to
    -- select byte enables of slave registers while writing.
    -- These registers are cleared when reset (active low) is applied.
    -- Slave register write enable is asserted when valid address and data are available
    -- and the slave is ready to accept the write address and write data.
    slv_reg_wren <= axi_wready and S_AXI_WVALID and axi_awready and S_AXI_AWVALID ;

    process (S_AXI_ACLK)
    variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
    begin
      if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
          slv_reg0 <= (others => '0');
--        slv_reg1 <= (others => '0');
--        slv_reg2 <= (others => '0');
--        slv_reg3 <= (others => '0');
        else
          loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
          if (slv_reg_wren = '1') then
            case loc_addr is
```

```
              when b"00" =>
                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
                  if ( S_AXI_WSTRB(byte_index) = '1' ) then
                    -- Respective byte enables are asserted as per write
strobes
                    -- slave registor 0
                    slv_reg0(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
                  end if;
                end loop;
--              when b"01" =>
--                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
--                  if ( S_AXI_WSTRB(byte_index) = '1' ) then
--                    -- Respective byte enables are asserted as per write
strobes
--                    -- slave registor 1
--                    slv_reg1(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
--                  end if;
--                end loop;
--              when b"10" =>
--                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
--                  if ( S_AXI_WSTRB(byte_index) = '1' ) then
--                    -- Respective byte enables are asserted as per write
strobes
--                    -- slave registor 2
--                    slv_reg2(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
--                  end if;
--                end loop;
--              when b"11" =>
--                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
--                  if ( S_AXI_WSTRB(byte_index) = '1' ) then
--                    -- Respective byte enables are asserted as per write
strobes
--                    -- slave registor 3
--                    slv_reg3(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
--                   -- end if;
--                  --end loop;
              when others =>
                slv_reg0 <= slv_reg0;
--              slv_reg1 <= slv_reg1;
--              slv_reg2 <= slv_reg2;
--              slv_reg3 <= slv_reg3;
            end case;
          end if;
        end if;
      end if;
    end process;

    -- Implement write response logic generation
    -- The write response and response valid signals are asserted by the slave
    -- when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
```

```
    -- This marks the acceptance of address and indicates the status of
    -- write transaction.

    process (S_AXI_ACLK)
    begin
      if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
          axi_bvalid  <= '0';
          axi_bresp   <= "00"; --need to work more on the responses
        else
          if (axi_awready = '1' and S_AXI_AWVALID = '1' and axi_wready = '1' and
S_AXI_WVALID = '1' and axi_bvalid = '0'  ) then
            axi_bvalid <= '1';
            axi_bresp  <= "00";
          elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then   --check if bready is
asserted while bvalid is high)
            axi_bvalid <= '0';                                 -- (there is a possibility
that bready is always asserted high)
          end if;
        end if;
      end if;
    end process;

    -- Implement axi_arready generation
    -- axi_arready is asserted for one S_AXI_ACLK clock cycle when
    -- S_AXI_ARVALID is asserted. axi_awready is
    -- de-asserted when reset (active low) is asserted.
    -- The read address is also latched when S_AXI_ARVALID is
    -- asserted. axi_araddr is reset to zero on reset assertion.

    process (S_AXI_ACLK)
    begin
      if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
          axi_arready <= '0';
          axi_araddr  <= (others => '1');
        else
          if (axi_arready = '0' and S_AXI_ARVALID = '1') then
            -- indicates that the slave has acceped the valid read address
            axi_arready <= '1';
            -- Read Address latching
            axi_araddr  <= S_AXI_ARADDR;
          else
            axi_arready <= '0';
          end if;
        end if;
      end if;
    end process;

    -- Implement axi_arvalid generation
    -- axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
    -- S_AXI_ARVALID and axi_arready are asserted. The slave registers
    -- data are available on the axi_rdata bus at this instance. The
    -- assertion of axi_rvalid marks the validity of read data on the
```

```
    -- bus and axi_rresp indicates the status of read transaction.axi_rvalid
    -- is deasserted on reset (active low). axi_rresp and axi_rdata are
    -- cleared to zero on reset (active low).
    process (S_AXI_ACLK)
    begin
      if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
          axi_rvalid <= '0';
          axi_rresp  <= "00";
        else
          if (axi_arready = '1' and S_AXI_ARVALID = '1' and axi_rvalid = '0') then
            -- Valid read data is available at the read data bus
            axi_rvalid <= '1';
            axi_rresp  <= "00"; -- 'OKAY' response
          elsif (axi_rvalid = '1' and S_AXI_RREADY = '1') then
            -- Read data is accepted by the master
            axi_rvalid <= '0';
          end if;
        end if;
      end if;
    end process;

    -- Implement memory mapped register select and read logic generation
    -- Slave register read enable is asserted when valid address is available
    -- and the slave is ready to accept the read address.
    slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not axi_rvalid) ;

    --process (slv_reg0, slv_reg1, slv_reg2, slv_reg3, axi_araddr, S_AXI_ARESETN,
slv_reg_rden)
    process (slv_reg1, axi_araddr, S_AXI_ARESETN, slv_reg_rden)
    variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
    begin
        -- Address decoding for reading registers
        loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
        case loc_addr is
--        when b"00" =>
--          reg_data_out <= slv_reg0;
          when b"01" =>
            reg_data_out <= slv_reg1;
--        when b"10" =>
--          reg_data_out <= slv_reg2;
--        when b"11" =>
--          reg_data_out <= slv_reg3;
          when others =>
            reg_data_out  <= (others => '0');
        end case;
    end process;

    -- Output register or memory read data
    process( S_AXI_ACLK ) is
    begin
      if (rising_edge (S_AXI_ACLK)) then
        if ( S_AXI_ARESETN = '0' ) then
          axi_rdata  <= (others => '0');
```

```
        else
          if (slv_reg_rden = '1') then
            -- When there is a valid read address (S_AXI_ARVALID) with
            -- acceptance of read address by the slave (axi_arready),
            -- output the read dada
            -- Read address mux
              axi_rdata <= reg_data_out;     -- register read data
          end if;
        end if;
      end if;
    end process;

    -- Add user logic here
my_fir: FIR port map (
        X => slv_reg0(7 downto 0),
        valid_in => slv_reg0(8),
        clk => S_AXI_ACLK,
        rst => slv_reg0(9),
        Y => fir_out_reg( 18 downto 0),
        valid_out => fir_out_reg (19)
    );


check_out:  process( S_AXI_ACLK ) is
    begin
      if (rising_edge (S_AXI_ACLK)) then
        if prev_valid_in ='0' and slv_reg0(8)='1' then
             slv_reg1(19)<='0';
        elsif fir_out_reg (19)='1' then
              slv_reg1<=fir_out_reg;
        end if;
        prev_valid_in <= slv_reg0(8);
      end if;
    end process;
-- User logic ends
end arch_imp;
```

Βλέπουμε όπως αναφέραμε ότι το Vivado ορίζει όλα τα σήματα που είναι απαραίτητα για την υλοποίηση του AXI στην αρχή. Στην συνέχεια εμείς εισάγουμε το component μας ( το FIR) στο ip που φτιάχνουμε ( για το ip θα αναφερθούμε εκτενέστερα στην συνέχεια) και ορίζονται διάφορα ενδιάμεσα σήματα για τον συγχρονισμό της εγγραφής και της ανάγνωσης από τους καταχωρητές. Επιπλέον ορίζονται οι 4 slave registers , όπου στον 0 θα γράφεται η είσοδος και στον 1 θα γράφεται η έξοδος του FIR.( Τα σήματα του AXI που ορίζονται εδώ έχουν και μία άλλη σημασία που θα την δούμε στην συνέχεια). Στην συνέχεια ορίζονται διάφορες διεργασίες( processes) που αφορούν τον συγχρονισμό αναγνώσεων και εγγραφών πάλι και στην συνέχεια πραγματοποιούμε την πρώτη μας τροποποίηση στην διεργασία που καθορίζει την εγγραφή της εισόδου.

```
process (S_AXI_ACLK)
    variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
    begin
      if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
```

```
          slv_reg0 <= (others => '0');
--        slv_reg1 <= (others => '0');
--        slv_reg2 <= (others => '0');
--        slv_reg3 <= (others => '0');
        else
          loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
          if (slv_reg_wren = '1') then
            case loc_addr is
              when b"00" =>
                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
                  if ( S_AXI_WSTRB(byte_index) = '1' ) then
                    -- Respective byte enables are asserted as per write
strobes
                    -- slave registor 0
                    slv_reg0(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
                  end if;
                end loop;
--              when b"01" =>
--                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
--                  if ( S_AXI_WSTRB(byte_index) = '1' ) then
--                    -- Respective byte enables are asserted as per write
strobes
--                    -- slave registor 1
--                    slv_reg1(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
--                  end if;
--                end loop;
--              when b"10" =>
--                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
--                  if ( S_AXI_WSTRB(byte_index) = '1' ) then
--                    -- Respective byte enables are asserted as per write
strobes
--                    -- slave registor 2
--                    slv_reg2(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
--                  end if;
--                end loop;
--              when b"11" =>
--                for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
--                  if ( S_AXI_WSTRB(byte_index) = '1' ) then
--                    -- Respective byte enables are asserted as per write
strobes
--                    -- slave registor 3
--                    slv_reg3(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
--                   -- end if;
--                  --end loop;
              when others =>
                slv_reg0 <= slv_reg0;
--              slv_reg1 <= slv_reg1;
--              slv_reg2 <= slv_reg2;
--              slv_reg3 <= slv_reg3;
            end case;
```

```
          end if;
        end if;
      end if;
    end process;
```

Στον παραπάνω κώδικα κάνουμε ( όπως φαίνεται ) comment out όλα τα σημεία όπου υπάρχει δυνατότητα επιλογής για την εγγραφή της εισόδου, και αφήνουμε μόνο την δυνατότητα εγγραφής στον slave register 0 , καθώς μόνο αυτός θα μας χρειαστεί. Έτσι αποφεύγονται τα conflicts και διευκολύνεται η διαδικασία, απλοποιείται ο κώδικας και εξοικονομείται υλικό εφόσον δεν υλοποιούνται οι ανάλογοι πολυπλέκτες για τα cases. Το ίδιο κάνουμε και πιο κάτω όπου υλοποιείται η επιλογή για τον καταχωρητή που θα διαβάζεται η έξοδος του AXI. Πάλι κάνουμε comment out τους registers που δεν χρησιμοποιούνται και συνεπώς αφήνουμε μόνο τον slave register 1.

```
    --process (slv_reg0, slv_reg1, slv_reg2, slv_reg3, axi_araddr, S_AXI_ARESETN,
slv_reg_rden)
    process (slv_reg1, axi_araddr, S_AXI_ARESETN, slv_reg_rden)
    variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
    begin
        -- Address decoding for reading registers
        loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
        case loc_addr is
--        when b"00" =>
--          reg_data_out <= slv_reg0;
          when b"01" =>
            reg_data_out <= slv_reg1;
--        when b"10" =>
--          reg_data_out <= slv_reg2;
--        when b"11" =>
--          reg_data_out <= slv_reg3;
          when others =>
            reg_data_out  <= (others => '0');
        end case;
    end process;
```

Στην συνέχεια εισάγουμε την δική μας λογική η οποία θα φροντίσει για το maping του FIR με τους αντίστοιχους slave registers και κάποιες προσθήκες για την υλοποίηση του συγχρονισμού

```
    -- Add user logic here
my_fir: FIR port map (
        X => slv_reg0(7 downto 0),
        valid_in => slv_reg0(8),
        clk => S_AXI_ACLK,
        rst => slv_reg0(9),
        Y => fir_out_reg( 18 downto 0),
        valid_out => fir_out_reg (19)
    );


check_out:  process( S_AXI_ACLK ) is
    begin
      if (rising_edge (S_AXI_ACLK)) then
        if prev_valid_in ='0' and slv_reg0(8)='1' then
            slv_reg1(19)<='0';
```

```
        elsif fir_out_reg (19)='1' then
              slv_reg1<=fir_out_reg;
        end if;
        prev_valid_in <= slv_reg0(8);
      end if;
    end process;
-- User logic ends
```

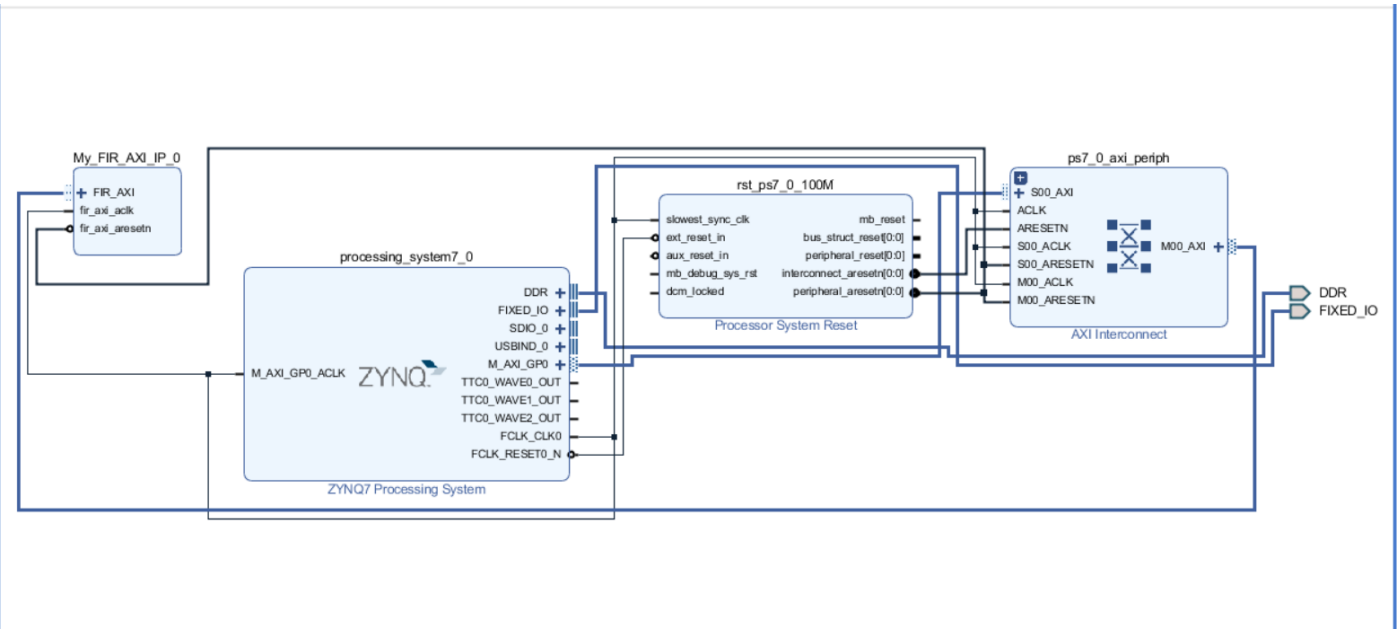
Όπως, αναφέραμε και πιο πάνω στο FIR υπήρχαν δύο επιλογές για την υλοποίηση του συγχρονισμού είτε να τροποποιήσουμε το FIR είτε να εισάγουμε κάποια λογική στο AXI αξιοποιώντας το πρωτόκολλο. Για την είσοδο επιλέξαμε να τροποποιήσουμε το FIR και για αυτό, όπως φαίνεται παραπάνω, πραγματοποιήσαμε ένα απλό mapping του slv_reg0 ( τον θέσαμε ως είσοδο) στις εισόδους του FIR , σύμφωνα με τις προδιαγραφές που μας δόθηκαν. Για τον συγχρονισμό της εξόδου προσθέσαμε έναν δικό μας καταχωρητή, τον fir_out_reg. Ο καταχωρητής αυτός θα έχει πάντα την τιμή της εξόδου του FIR και θα αντιγράφεται στον slv_reg1 μόνο όταν παίρνει ένα νέο valid_out. Έτσι ο slv_reg1 θα έχει ουσιαστικά θα έχει την σωστή τιμή για την έξοδο για πολλούς κύκλους και συγκεκριμένα μέχρι το FIR να λάβει την επόμενη είσοδο ( η έξοδος του FIR σε αυτ΄οτο διάστημα αλλάζει και δεν παραμένει απαραίτητα σταθερή, συνεπώς αναγκαστήκαμε να εισάγουμε αυτό τον επιπλέον καταχωρητή, ώστε η έξοδος να αλλάζει μόνο όταν έχω νέο valid out) . Το valid_out της εξόδου θα είναι 1 μέχρι να λάβουμε ένα νέο valid_in. Ένα νέο valid_in λαμβάνουμε όταν έχουμε μεταβολή του valid_in 0->1. Οπότε ομοίως με τον τρόπο που το κάναμε στο FIR ελέγχουμε το προηγούμενο valid_in και αν έχουμε μεταβολή από 0->1 τότε θέτουμε το valid_out=0 μέχρι να πάρουμε κάποιο νέο valid out από το FIR. Άρα το process που φτιάξαμε ουσιαστικά κάνει τα εξής:

1. Η τιμή της εξόδου του slv_reg1 ( δηλαδή τα bits 0 με 18) αλλάζει μόνο όταν έχουμε valid_out από το FIR, που σημαίνει ότι η τιμή της εξόδου διατηρείται μέχρι να λάβουμε μία νέα έξοδο
2. Το valid_out του slv_reg1 (δηλαδή το 19 bit) γίνεται από 0 σε 1 όταν από το FIR πάρουμε το valid_out. Το valid_out του slv_reg1 (δηλαδή το 19 bit) γίνεται από 1 σε 0 όταν πάρουμε ένα νέο ορθό valid in δηλαδή έχουμε valid_in=1 και prev_valid_in=0 . Αυτό σημαίνει ότι ένας νέος υπολογισμός ξεκινάει και συνεπώς η έξοδος πλέον δεν είναι έγκυρη μέχρι να λάβουμε την επόμενη.

Συνεπώς η έξοδος του PL μένει σταθερή για πολλούς κύκλους μέχρι το PS( που είναι πιο αργό) να την διαβάσει και να δώσει μία νέα είσοδο. Αν δεν κάναμε την παραπάνω προσθήκη , τότε το PL θα είχε valid_out μόνο για 1 κύκλο του PL και συνεπώς εφόσον το PS είναι πολύ πιο αργό είναι πολύ πιθανό πολλές φορές να μην κατάφερνε ποτέ να διαβάσει την έξοδο.

Σε αυτό το σημείο τελειώσαμε με το κομμάτι της υλοποίησης σε VHDL που έπρεπε να γράψουμε . Στην συνέχεια θα φτιάξουμε το δικό μας IP, ώστε να μπορούμε να το εισάγουμε στο block design και να υλοποιηθεί το block που θέλουμε μαζί με το AXI. Αρχικά για να φτιάξουμε το AXI ip μας πρέπει να φτιάξουμε ( το Vivado φτιάχνει τον κώδικα αυτόματα) ένα HDL wrapper το οποίο θα «περιτυλίγει» το AXI4-Lite και θα φροντίζει να πραγματοποιήσει το mapping των σημάτων του AXI4-Lite με τα αναγκαία σήματα του AXI , ώστε να μπορεί με βάση αυτά τα γνωστά και σαφώς καθορισμένα σήματα να πραγματοποιηθεί στην συνέχεια το block automation του Vivado ( δηλαδή η ένωση του PS με το IP μας).( ο κώδικας για το wrapper παράγεται αυτόματα από το Vivado και δεν κρίνεται αναγκαίο να τον παραθέσουμε εδώ). Πλέον έχουμε φτιάξει πλήρως το ip μας και μένει μόνο να το «φτιάξουμε» και στην πραγματικότητα, δηλαδή να το φτιάξει το Vivado για εμάς και να το προσθέσει στον κατάλογο με τα ip που έχει το Vivado. Και αυτό γίνεται αυτόματα (δηλαδή με κάποιες απλές επιλογές στα menu του Vivado, η εκτενής διαδικασία περιγράφεται και στο βοηθητικό υλικό της εκφώνησης). Πλέον έχουμε φτιάξει το ip μας και μπορούμε να το εισάγουμε σε ένα block design. Ένα ip ουσιαστικά είναι η πνευματική ιδιοκτησία της υλοποίησης ενός component. Η ακριβής υλοποίηση μπορεί να μην μας είναι άμεσα διαθέσιμη όμως το ip μπορεί να είναι , δηλαδή ένα block με εισόδους και εξόδους , το οποίο επιτελεί μία λειτουργία. Με αυτή την λογική σε ένα block design μπορούμε να εισάγουμε διάφορα ip και το μόνο που πρέπει να φροντίσουμε εμείς είναι το interconnection αυτών. Βέβαια τα περισσότερα από αυτά ( σχεδόν όλα) υποστηρίζουν το AXI , το οποίο έχει συγκεκριμένες εισόδους και εξόδους και συνεπώς μπορεί το vivado να πραγματοποιήσει ένα block automation και να φροντίσει για όλες τις συνδέσεις μεταξύ των component, εφόσον αυτές με βάση το πρωτόκολλο είναι αυστηρά καθορισμένες.

Με αυτή την λογική εισάγουμε σε ένα νέο block design το ip μας και το ip που αντιστοιχεί στο PS και το vivado πραγματοποιεί αυτόματα τις συνδέσεις για εμάς ( το block design για το PS είναι το ZYNQ7 Processing System).




Παραπάνω βλέπουμε το FIR και το αντίστοιχο AXI wrapper που ορίσαμε και την διασύνδεση αυτών με το PS ( έχει προστεθεί και μία μονάδα υπεύθυνη για το reset του PS).

Με τα παραπάνω βήματα έχουμε πλέον υλοποιήσει πλήρως στο Vivado το ζητούμενο της άσκησης και συνεπώς το μόνο που μένει είναι να περάσουμε αυτό το design στην πλακέτα και να την προγραμματίσουμε, ώστε να ελέγξουμε την λειτουργία της. Για να προγραμματίσουμε ένα FPGA πρέπει να πάρουμε το bitstream ( το οποίο καθορίζει τον προγραμματισμό του Chip, δηλαδή τις τιμές που θα πάρουν οι LUT, τις διασυνδέσεις κ.λ.π) και να το εισάγουμε στο FPGA. Οπότε κάνουμε generate το bitsream από το menu , και κάνουμε export hardware( ουσιαστικά ετοιμάσαμε τα αρχεία που θα προγραμματίσουν το FPGA). Τελευταίο βήμα είναι να ανοίξουμε το SDK και να προγραμματίσουμε το PS ( σε γλώσσα C) και να φορτώσουμε στο FPGA το bistream ( από το program FPGA του SDK).

Παρακάτω παραθέτουμε τον κώδικα σε C που εφαρμόζει σαν είσοδο στο FIR κάποια στοιχεία που δόθηκαν κατά την εξέταση της παρούσας εργασίας και τυπώνει στην έξοδο το αντίστοιχο αποτέλεσμα.

```c
#include <stdio.h>
#include "myip.h"
#include "xparameters.h"
#include <xil_io.h>

#define PIX_PROC_LIMIT 256
// Define delay length
#define DELAY 1000000

/* main function */
int main(void){
    /* unsigned 32-bit variables for storing current LED value */
    u32 i,input,result,reset,valid_in,valid_out,output;
    u32 A[12] = {112,97,195,203,47,125,114,165,181,193,70,174};
    valid_in =1<<8;
    reset =1<<9;
    xil_printf("FIR IP test begin\r\n");
    xil_printf("-----------------------------------------\r\n");
    input = reset ;
    xil_printf("Input value: reset\n");
    MYIP_mWriteReg(XPAR_MY_FIR_AXI_IP_0_FIR_AXI_BASEADDR, 0, input);
```

```
    for (i = 0; i < 12; i++)
    {       /* Print value to terminal */
        xil_printf("Input value: %lu\t", A[i]);
        input = A[i] | valid_in;
        MYIP_mWriteReg(XPAR_MY_FIR_AXI_IP_0_FIR_AXI_BASEADDR, 0, input);
        while(1){
            result = MYIP_mReadReg(XPAR_MY_FIR_AXI_IP_0_FIR_AXI_BASEADDR, 4); // 4 because
this is slave reg 1 (4 bytes after)
            valid_out = result & (1<<19);
            output = result & ((1<<19)-1);
            if (valid_out)
                break;

        }
        xil_printf("Output value: %lu\n", output);
        input = 0;
        MYIP_mWriteReg(XPAR_MY_FIR_AXI_IP_0_FIR_AXI_BASEADDR, 0, input);
    }
    return 1;
}
```

Αρχικά, πρέπει να φροντίσουμε να κάνουμε include τα αρχεία xparameters.h και myip.h ( το δεύτερο έχει το όνομα του ip που φτιάξαμε) . Το xparameters.h έχει την παράμετρο που καθορίζει το base address των slave registers του AXI4-LIte και συνεπώς και τον slv_reg0 και slv_reg1 . Συνεπώς αυτή η παράμετρος είναι απαραίτητη για να καθορίσουμε σε ποια θέση μνήμης θα διαβάσουμε και θα γράψουμε. Η διεύθυνση του slv_reg0 είναι αυτή που δίνεται στην παράμετρο XPAR_MY_FIR_AXI_IP_0_FIR_AXI_BASEADDR και το offset καθορίζει τον register. Για τους υπόλοιπου registers προσθέτουμε όσα και τα 8*χ bits διαφέρουν. Ο reg1 με την reg0 θα έχουνε 32 bit διαφορά συνεπώς offset 0. Βέβαια το αρχείο myip.h έχει το offset με μορφή παραμέτρου (εδώ δεν χρησιμοποιήσαμε τελικά την παράμετρο αλλά είδαμε το offset από εκεί ). Τέλος απαραίτητο είναι το xil_io.h το οποίο περιέχει τις αντίστοιχες συναρτήσεις για την ανάγνωση και εγγραφή στο terminal. Αναφορικά με τον κώδικα μας ορίζουμε αρχικά κάποιες βοηθητικές μεταβλητές, το valid_in θα αντιστοιχεί στο bit της εισόδου που αντιστοιχεί το valid_in του slv_reg0 και η ίδια ακριβώς λογική ισχύει και για το reset. Ο A είναι ο πίνακας με όλες τις εισόδους που θα δώσουμε στο FIR. Αρχικά, στον κώδικα πραγματοποιούμε ένα reset του FIR και στην συνέχεια με ένα for loop εφαρμόζουμε ένα-ένα τα στοιχεία του πίνακα A σαν είσοδο στο FIR. Για να το κάνουμε αυτό πρώτα θέτουμε το valid_in για την είσοδο και την τιμή που έχει ο πίνακας A για το αντίστοιχο iteration. Γράφουμε την είσοδο στον slv_reg0 . Στην συνέχεια γράφουμε στον slv_reg0 valid_in =0 ώστε στην επόμενη είσοδο που θα εφαρμόσουμε να έχουμε valid_in 0->1, που είναι απαραίτητο όπως εξηγήσαμε. Στην συνέχεια με ένα while loop διαβάζουμε την είσοδο μέχρι να πάρουμε valid_out. Όταν πάρουμε valid_out γράφουμε την έξοδο στο terminal και πάμε στο επόμενο iteration.

Ακολουθώντας όλα τα βήματα που περιγράψαμε παραπάνω καταφέραμε να πραγματοποιήσουμε το ζητούμενο της άσκησης, δηλαδή να χρησιμοποιήσουμε το πρωτόκολλο AXI και συγκεκριμένα την διεπαφή AXI-4LIte για να προγραμματίσουμε το Zybo, ώστε στο FPGA του να έχει πλέον περαστεί το bitstream του FIR ,να επιτελεί αυτή την λειτουργία και να επικοινωνεί ορθά με το PS.