Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

5η Σειρά Ασκήσεων

Μάθημα: Ψηφιακά Συστήματα VLSI

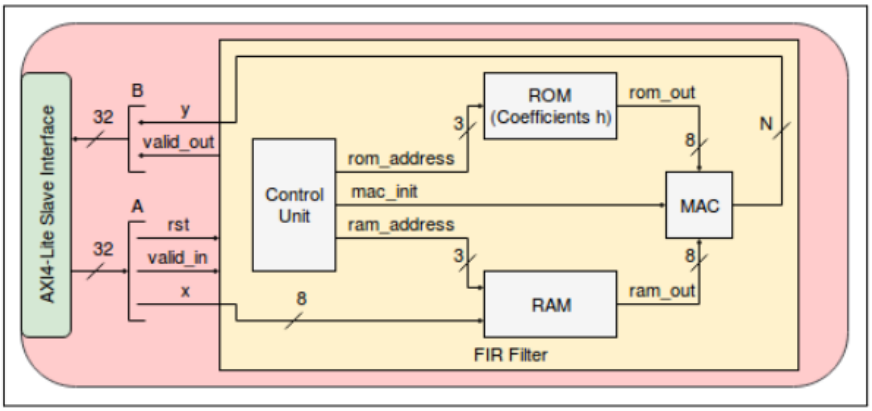
Εξάμηνο: 8ο

Ονοματεπώνυμο: Αλεξοπούλου Γεωργία, Γκενάκου Ζωή

Υλοποίηση FIR Φίλτρου σε ZYBO

Σε αυτή την εργαστηριακή άσκηση καλούμαστε να προγραμματίσουμε την αναπτυξιακή πλακέτα ZYBO, ώστε να υπολοποιεί ένα FIR φίλτρο στο οποίο τα δεδομένα εισόδου θα αποστέλλονται από τον ενσωματωμένο επεξεργαστή (ARM) προς το FPGA για επεξεργασία, και αντίστροφα για τα αντίστοιχα αποτελέσματα. Η επικοινωνία επεξεργαστή-FPGA θα βασίζεται στο πρωτόκολλο AXI.

Στα πλαίσια αυτής της εργαστηριακής άσκησης καλούμαστε να υλοποιήσουμε ένα 8-tap FIR φίλτρο (M = 7), το οποίο θα υλοποιεί διεπαφή AXI4-Lite για την επικοινωνία με το ZYNQ Processing System (PS). Ένα ενδεικτικό παράδειγμα του προς υλοποίηση FIR φίλτρου παρουσιάζεται στην παρακάτω εικόνα:



Όπως φαίνεται και από την εικόνα το FIR φίλτρο έχει ακριβώς την ίδια αρχιτεκτονική με αυτό της Εργαστηριακής Άσκησης 5.

Στην συνέχεια, συνδέουμε το FIR που έχουμε ήδη υλοποιήσει με μία AXI4-Lite διεπαφή (AXI4-Lite Slave Interface). Το πλήθος των bits του σήματος εισόδου x και των συντελεστών του φίλτρου h είναι ίσο με 8 bits. Η διασύνδεση των σημάτων εισόδου και εξόδου του φίλτρου με την AXI διεπαφή να υλοποιηθεί όπως φαίνεται στην Εικόνα. Συγκεκριμένα :

- A[7:0] = x
- A[8] = valid_in
- A[9] = rst
- A[31:10] = not used
- B[N-1:0] = y
- B[N] = valid_out
- B[31:N+1] = not used

Επιπλέον να αναπτύσσουμε την ανάλογη εφαρμογή λογισμικού για την αποστολή των σημάτων εισόδου και την λήψη των σημάτων εξόδου του φίλτρου από τον ενσωματωμένο επεξεργαστή. Επειδή η επικοινωνία μεταξύ PS-PL υλοποιείται μέσω διαύλου επικοινωνίας εύρους 32-bit, η αποστολή των σημάτων εισόδου και εγκυρότητας θα αποστέλλονται ταυτόχρονα, ομαδοποιημένα στην ίδια λέξη των 32-bit (αντίστοιχα και για τα σήματα εξόδου). Η εφαρμογή λογισμικού για κάθε αποστολή ενός δεδομένου προς το FIR θα πρέπει να περιμένει να λάβει το αντίστοιχο έγκυρο αποτέλεσμα και να το εμφανίζει στο τερματικό μέσω σειριακής επικοινωνίας πριν κάνει αποστολή του επόμενου δεδομένου εισόδου προς το FIR, διαφορετικά να μην εμφανίζει τίποτα. Για το λόγο αυτό η εφαρμογή λογισμικού θα πρέπει να ελέγχει την εγκυρότητα του αποτελέσματος που λαμβάνει.

Τώρα, θα περιγράψουμε τα βήματα που χρειάστηκαν για να υλοποιήσουμε το FIR φίλτρο στο Zynq.

Αρχικά, δημιουργούμε ένα καινούργιο project (New Project) και επιλέγουμε Tools → Create and Package New IP, ώστε να δημιουργήσουμε ένα νέο IP. Διαλέγουμε το Create new AXI4 Peripheral, ονομάζουμε το αρχείο μας, και πατάμε το Next μέχρι να δημιουργηθεί το IP Package. Στο παράθυρο Sources, επιλέγουμε το Add Sources και προσθέτουμε το VHDL αρχείο για το FIR (που φτιάξαμε στην προηγούμενη άσκηση). Επίσης, προσθέτουμε το αρχείο `FIR_IP_v1_0_S00_AXI.vhd` το οποίο βρίσκεται μέσο στον φάκελο `ip repo` και τροποποιούμε τον κώδικα για AXI.

Θα παραθέσουμε μόνο τα κομμάτια του κώδικα που τροποποιήσαμε:

Ο κώδικας του AXI περιλαμβάνει το declaration του entity `FIR_IP_v1_0_S00_AXI` με τις παράμετρους που ορίζουν τα δεδομένα και τα address bus widths του AXI, μαζί και IO ports και σήματα όπως το clock, reset, read/write signals κτλ.

Στο επόμενο κομμάτι περιγράφεται η εσωτερική αρχιτεκτονική του AXI slave module, όπου ορίζονται σήματα AXI όπως επίσης και σήματα που θα χρησιμοποιήσουμε για το φίλτρο μας. Σε αυτό το σημείο ορίζουμε κιόλας το component του FIR φίλτρου μας. Στον παρεχόμενο κώδικα VHDL, οι slave καταχωρητές υλοποιούνται χρησιμοποιώντας σήματα όπως `slv_reg0, slv_reg1, slv_reg2` και `slv_reg3`. Η πρόσβαση και ο χειρισμός αυτών των καταχωρητών γίνεται με βάση τα AXI transactions (ανάγνωση ή εγγραφή) που λαμβάνονται από την κύρια μονάδα AXI. Τα περιεχόμενα αυτών των καταχωρητών χρησιμοποιούνται για τη διαμόρφωση του φίλτρου FIR και τη διαχείριση της μεταφοράς δεδομένων μεταξύ του διαύλου AXI και του φίλτρου. Στην συνέχεια, υλοποιούνται διάφορα process blocks τα οποία χειρίζονται διάφορες πτυχές της επικοινωνίας AXI και της πρόσβασης των καταχωρητών.

```
architecture arch_imp of FIR_IP_v1_0_S00_AXI is

	-- 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;

    --our signals
    signal Aip : std_logic_vector (31 downto 0);
    signal Bip : std_logic_vector(31 downto 0);

    signal reset_ip : std_logic;
    signal valid_in_ip: std_logic;
    signal x_ip: std_logic_vector(7 downto 0);
    signal valid_out_ip: std_logic;
    signal y_ip: std_logic_vector(18 downto 0);

    signal en_ram_rom_ip: std_logic;
    signal rom_output_ip,ram_output_ip: std_logic_vector(7 downto 0);
    signal rom_address_ip,ram_address_ip: std_logic_vector(2 downto
0);
    signal mac_init_ip,we_out_ip: std_logic;

    component FIR is -- custom FIR added
        Port ( clock : in std_logic;
               reset : in std_logic;
               valid_in : in std_logic;
```

```
                en_ram_rom: in std_logic;
                x : in std_logic_vector(7 downto 0);
                valid_out : out std_logic;
                fir_output : out std_logic_vector (18 downto 0);
                rom_output,ram_output:out STD_LOGIC_VECTOR (7 downto
0);
                rom_address,ram_address:out STD_LOGIC_VECTOR (2 downto
0);
                mac_init : out std_logic;
                we_out:out std_logic
            );
    end component;
```

Το πρώτο process block που θα τροποποιήσουμε είναι αυτό που χειρίζεται την λογική για την εγγραφή δεδομένων σε memory-mapped registers. Συγκεκριμένα, κάνουμε comment out τον `slv_reg1` στον οποίο πρόκειται να αναθέσουμε την έξοδο του φίλτρου μας. Θέλουμε να αποφύγουμε εγγραφές που μπορούν αλλάξουν το σωστό αποτέλεσμα του κώδικα. Με αυτό τον τρόπο, εγγυόμαστε ότι τα δεδομένα στον καταχωρητή αυτόν θα μείνουν αναλλοίωτα, θα διαβάζονται μόνο όταν χρειάζεται και να μην έχουμε multi_drive error. Επιπλέον, σβήνουμε τα σημεία που γράφουμε την λογική του `slv_reg1` τον οποίο εμείς χρησιμοποιούμε για να διαβάζουμε, δηλαδή για να στέλνουμε από το hardware στο software. Χρειάζεται να προσθέσουμε στην συνθήκη if (όταν είναι έτοιμο, δηλαδή, να διαβάσει από το software) ένα else ώστε να κάνουμε το `valid_in = 0` (`else slv_reg0(8) <= '0'`), αυτό συμβαίνει ώστε το FIR να μην διαβάζει πολλές φορές την ίδια είσοδο αλλά να λαμβάνει και να επεξεργάζεται κάθε είσοδο μόνο μια φορά.

```
      -- 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" =>
                  slv_reg2 <= (others => '0');
                  --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;
              else slv_reg0(8) <= '0';
            end if;
          end if;
        end if;
      end process;
```

Το επόμενο process block που επεξεργαζόμαστε είναι αυτό που υλοποιεί την λογική που είναι υπεύθυνη να διαβάζει από memory-mapped καταχωρητές του hardware design. Σε αυτό το

κομμάτι δημιουργούμε τα δεδομένα εξόδου. Όταν το valid_out_ip είναι '1', το οποίο σημαίνει ότι το software (master) στέλνει έγκυρα δεδομένα εξόδου, το Bip γράφεται στον slv_reg1, δηλαδή το hardware τα αποθηκεύει στον slv_reg1. Τέλος, όταν έρθει το σήμα ότι ο master είναι έτοιμος να διαβάσει, ο slave (hardware) στέλνει τα δεδομένα του slv_reg1 μέσω του axi_rdata στον master (software) και μηδενίζει το bit που αντιπροσωπεύει valid_out για να μην αποθηκεύσει ξανά την ίδια τιμή, και ουσιαστικά ο slave περιμένει μέχρι ο master να στείλει δεδομένα με valid_in = 1, ώστε το FIR να παράγει δεδομένα εξόδου με valid_out = 1.

```
	-- 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)
	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 (valid_out_ip = '1') then
           slv_reg1 <= Bip;
        elsif (slv_reg_rden = '1') then
           axi_rdata <= reg_data_out;     -- register read data
           slv_reg1(19) <= '0';
        else
           slv_reg1 <= slv_reg1;
      end if;
    end if;
  end if;
end process;
```

Τέλος, γράφουμε το User Logic, το οποίο αποτελείται από το mapping των στοιχείων του component του FIR στα σήματα του AXI και ένα process. Σε αυτή την διαδικασία αναθέτουμε την τιμή του `slv_reg0`, του καταχωρητή που διαβάζουμε, στο σήμα `Aip`. Στην συνέχεια αναθέτουμε στο `x_ip`, τη είσοδο του φίλτρου τα τελευταία 8 bit του `Aip`, αναθέτουμε στο `valid_in_ip` το 9ο bit του `Aip` και στο `reset_ip` το 10ο bit του `Aip`. Τέλος, αναθέτουμε την τιμή του `Bip` που αποτελείται από το `valid_out_ip` (1 bit), την έξοδο του φίλτρου `y_ip` (19 bits) και 12 μηδενικά στην αρχή του σήματος για να συμπληρωθεί ο 32bit καταχωρητής.

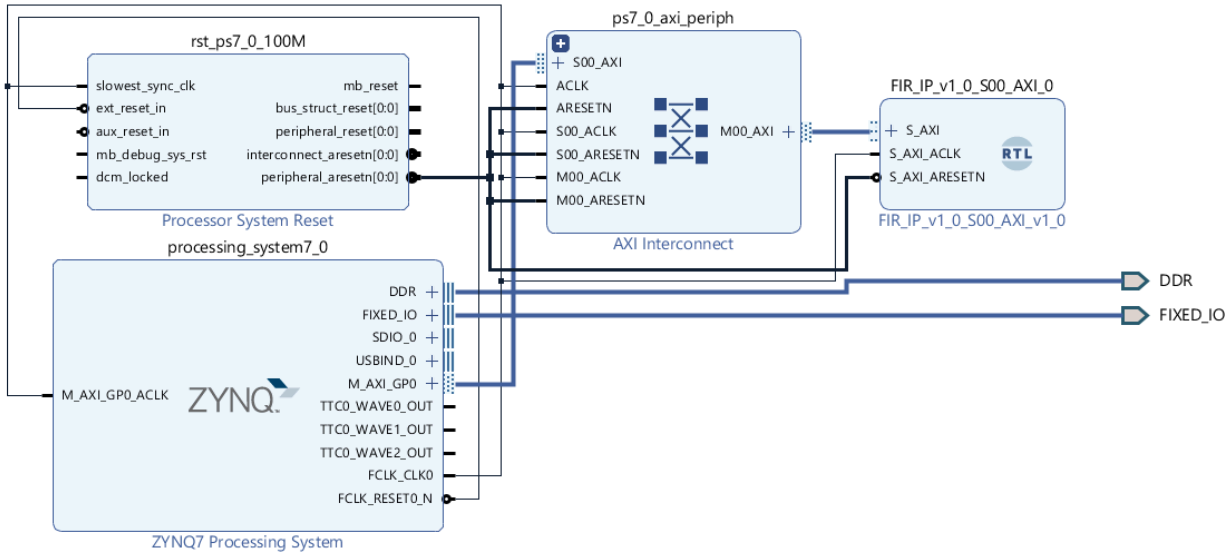
```
-- Add user logic here
FIR_label : FIR  port map (
      clock => S_AXI_ACLK,
      reset => reset_ip,
      x => x_ip,
      valid_in => valid_in_ip,
      valid_out => valid_out_ip,
      fir_output => y_ip,
      en_ram_rom => '1',
      we_out => we_out_ip,
      mac_init => mac_init_ip,
```

```
            rom_output => rom_output_ip,
            ram_output => ram_output_ip,
            rom_address => rom_address_ip,
            ram_address => ram_address_ip
        );

    process (S_AXI_ACLK) is
    begin

        Aip <= slv_reg0;
        x_ip <= Aip(7 downto 0);
        valid_in_ip <= Aip(8);
        reset_ip <= Aip(9);
        Bip <= "000000000000" &valid_out_ip & y_ip;


    end process;
      -- User logic ends

end arch_imp;
```

Αφού αποθηκεύσαμε τις αλλαγές μας, έχουμε ένα έτοιμο IP για το FIR φίλτρο μας.

Μέσω των επιλογών Settings → IP → Repository, ότι υπάρχει το path που δημιουργήσαμε για το FIR_IP προκειμένου να μπορέσουμε να το προσθέσουμε στο design.

Διαλέγουμε στο Project Manager την επιλογή Create Block Design, διαλέγουμε το Add IP και προσθέτουμε το ZYNQ7 Processing System. Μας εμφανίζεται η επιλογή Run Block Automation, το διαλέγουμε και στην συνέχεια προσθέτουμε και το IP που δημιουργήσαμε. Αυτό είτε γίνεται αυτόματα πατώντας άλλη μια φορά το Add IP, ή με δεξί κλίκ πάνω στο αρχείο `FIR_IP_v1_0_S00_AXI` και Add Module to Block Design. Επιλέγουμε άλλη μια φορά το Run Block Automation και προκύπτει το παρακάτω Block Design.



Κάνουμε Validate Design. Στο παράθυρο Sources, κάνουμε δεξί κλικ στο design μας και πατάμε την επιλογή Create HDL Wrapper. Στο αρχείο αυτό κάνουμε πάλι δεξί κλικ και επιλέγουμε Set as Top. Τρέχουμε το RTL Analysis, και Run Implementation για να βεβαιωθούμε ότι δεν υπάρχουν Errors ή σημαντικά Warnings. Στο Flow Navigator επιλέγουμε το Generate Bitstream και όταν τελειώσει αυτή η διαδικασία, πάμε File → Export → Export Hardware και τέλος File → Launch SDK και ανοίγει αυτόματα το SDK.

Ακολουθούμε τα εξής βήματα: File → New → Application Project, ονομάζουμε το πρόγραμμα μας Next → Hello World για να δημιουργηθεί αυτόματα το .c file μέσα στον φάκελο src που θα τροποποιήσουμε. Ο κώδικας του SDK φαίνεται παρακάτω:

```
******************************************************/
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "sleep.h"
#include <inttypes.h>
#include "xil_types.h"
#include "xil_io.h"
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "sleep.h"
#define MY_IP_BASEADDR 0x43C00000
```

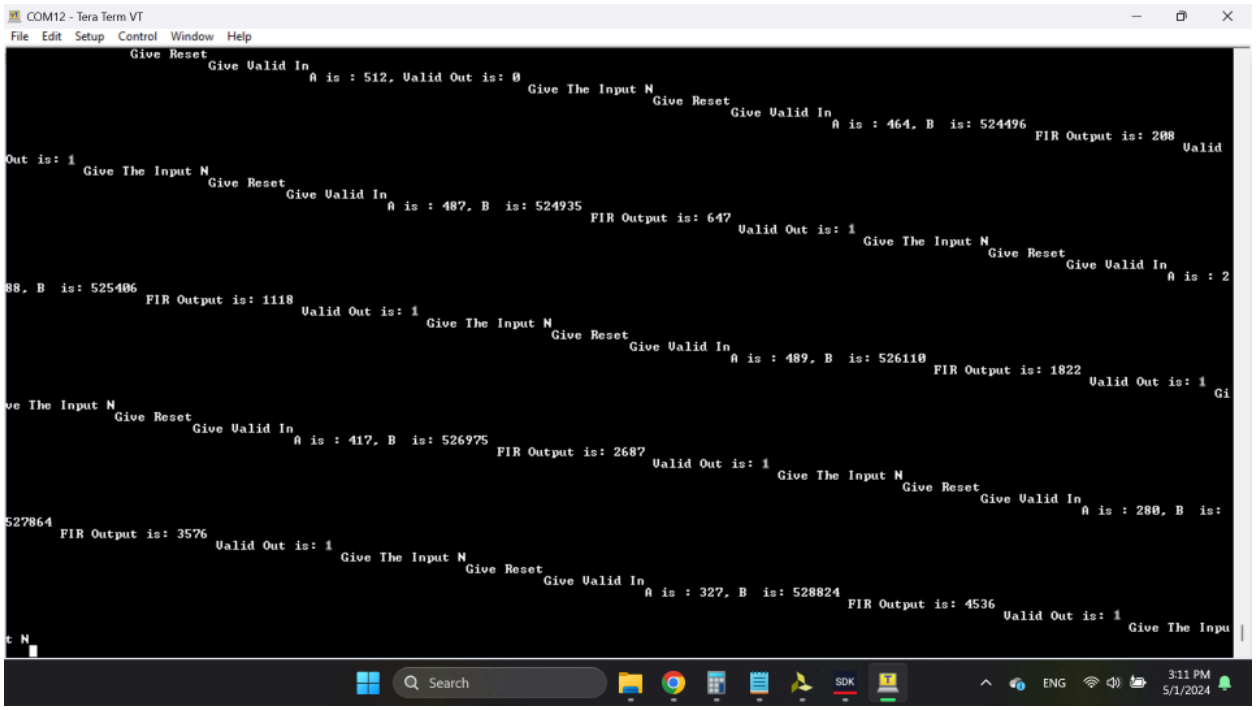
```c
int main() {

	init_platform();

	int32_t B, A, Y;
	int valid_out, valid_out_temp;
	int rst, reset, valid_in, data_in;

	while(1) {

		valid_out = 0;

		xil_printf("Give The Input N\n");
		scanf("%d",&data_in);

		xil_printf("Give Reset\n");
		scanf("%d",&reset);

		xil_printf("Give Valid In\n");
		scanf("%d",&valid_in);

		valid_in = valid_in << 8;
		rst = reset;
		reset = reset << 9;
		A = reset | valid_in | data_in;

		xil_printf("A is : %d, ", A);

		Xil_Out32((MY_IP_BASEADDR+0x00), A);

		usleep(500);

		if( valid_in == 0  || rst == 1 )  {
			B = Xil_In32(MY_IP_BASEADDR+0x04);
			valid_out = B &  0x80000;
			xil_printf("Valid Out is: %d \n",  valid_out);
		}
		else{
```

```
                if (valid_out == 0 ) {
                      B = Xil_In32(MY_IP_BASEADDR+0x04);
                      valid_out = B & 0x80000;
                      xil_printf("B  is: %d \n",  B);
                }

                Y = B & 0x7FFFF;
                xil_printf("FIR Output is: %u \n",Y);
                valid_out_temp = valid_out << 18;
                xil_printf("Valid Out is: %u \n", valid_out_temp);
          }
      }
      cleanup_platform();
      return 0;
}
```

Σε αυτόν τον κώδικα, ορίζουμε αρχικά το base address του IP που δημιουργήσαμε που ορίζεται στο Vivado Project, κάνουμε **`init_platform()`** που προετοιμάζει το ενσωματωμένο σύστημα μας για το software execution, επιβεβαιώνοντας ότι όλα τα απαραίτητα hardware components είναι σωστά configured. Αρχικοποιούμε τις μεταβλητές που θα κρατάνε τα δεδομένα εισόδου και εξόδου και ξεκινάμε μια while loop όπου υπάρχει το κυρίως πρόγραμμα μας. Μέσα στον βρόγχο αυτό, ζητάμε από τον χρήστη τις εισόδους `data_in`, `reset` και `valid_in` και τις διαβάζουμε μέσω της **`scanf()`**. Στην συνέχεια κατασκευάζουμε την είσοδο A έτσι όπως μας υποδεικνύεται στην εκφώνηση και γράφουμε τα Input Data στο IP Core μέσω της συνάρτησης `Xil_Out32((MY_IP_BASEADDR+0x00), A)`. Αν το `valid_in` είναι 0 ή το `reset` είναι 1 ή ακόμα και όταν το `valid_out` είναι 0, διαβάζουμε την έξοδο B από τους καταχωρητές του IP και απομονώνουμε και το `valid_out` από τον καταχωρητή B. Τέλος, απομονώνουμε το Y και το `valid_out` από τον καταχωρητή B και τα εκτυπώνουμε.

Παραθέτουμε την έξοδο στο Terminal βάζοντας τις εξής εισόδους:

Input N: 0	**Reset:** 1	**Valid In:** 0
Input N: 208	**Reset:** 0	**Valid In:** 1
Input N: 231	**Reset:** 0	**Valid In:** 1
Input N: 32	**Reset:** 0	**Valid In:** 1
Input N: 233	**Reset:** 0	**Valid In:** 1



- Για την πρώτη είσοδο εκτυπώνεται:

A = 512 (decimal) το οποίο είναι ισοδύναμο με 0010 0000 0000 (binary) το οποίο είναι σωστό καθώς έχουμε 8bit μηδενική είσοδο, 9ο bit μηδενικό γιατί το valid in είναι μηδενικό και το 10ο bit (reset) είναι ίσο με 1. Το valid out είναι 0.

- Για είσοδο N = 208:

Το A είναι 464 (decimal) το οποίο είναι 0001 1101 0000 (binary), όπου 1101 0000 (binary) = 208 (decimal) και το 9ο bit είναι 1 καθώς έχουμε valid in 1. Αντίστοιχα, βλέπουμε ότι το FIR Output = 208 και Valid Out = 1, που είναι η αναμενόμενη έξοδος.

- Για είσοδο N = 231:

Περιμένουμε την εξής έξοδο, 1101 0000 (208 decimal) · 0000 0010 + 1110 0111 (231 decimal) · 0000 0001 = 0010 1000 0111 (647 decimal) και Valid Out = 1. Αυτές οι έξοδοι εμφανίζονται όντως στο Terminal μας.

- Ακόμα ας δοκιμάσουμε για είσοδο N = 32:

Περιμένουμε την εξής έξοδο, 1101 0000 (208 decimal) · 0000 0011 + 1110 0111 (231 decimal) · 0000 0010 + 0010 0000 (32 decimal) · 0000 0001 = 0100 0101 1110 (1118 decimal) και Valid Out = 1. Αυτές οι έξοδοι εμφανίζονται όντως στο Terminal μας.

Άρα επιβεβαιώνουμε ότι όλα δουλεύουν όπως θα έπρεπε.