Módulo IP de Filtro FIR con Interfaz de comunicación AXI Lite

Jorge Contreras Ortiz – MEI – M18190

Para el proyecto de la asignatura de Design of Embedded Systems, se ha realizado un módulo IP consistente en un Filtro Fir de 4 coeficientes.

Para ello se ha descargado un código VHDL de la siguiente página web: [Código VHDL Filtro FIR].

Se ha modificado el código para dar una señal de Valid cuando el Dato de Salida esté disponible para lectura.

La entidad generada para el código del Filtro FIR es la siguiente:

```
entity Filtro_FIR_4in is
port (
  i_clk        : in  std_logic;
  i_rstb       : in  std_logic;
  -- coefficient
  i_coeff_0    : in  std_logic_vector( 7 downto 0);
  i_coeff_1    : in  std_logic_vector( 7 downto 0);
  i_coeff_2    : in  std_logic_vector( 7 downto 0);
  i_coeff_3    : in  std_logic_vector( 7 downto 0);
  -- data input
  i_data       : in  std_logic_vector( 7 downto 0);

  fir_valid    : out std_logic;
  -- filtered data
  o_data       : out std_logic_vector( 9 downto 0)
   );

end entity Filtro_FIR_4in;
```

Ilustración 1 Entidad Filtro FIR

Para la comunicación con el procesador y la escritura de los valores de los coeficientes y dato de entrada y lectura del dato de salida, se ha usado comunicación a través de la interfaz AXI Lite. La instanciación y conexión con el IP del AXI ha sido la siguiente:

```
component Filtro_FIR_4in is
    port (
    i_clk        : in  std_logic;
    i_rstb       : in  std_logic;
-- coefficient
    i_coeff_0    : in  std_logic_vector( 7 downto 0);
    i_coeff_1    : in  std_logic_vector( 7 downto 0);
    i_coeff_2    : in  std_logic_vector( 7 downto 0);
    i_coeff_3    : in  std_logic_vector( 7 downto 0);
-- data input
    i_data       : in  std_logic_vector( 7 downto 0);
    fir_valid    : out std_logic;
-- filtered data
    o_data       : out std_logic_vector( 9 downto 0)
    );
end component Filtro_FIR_4in;
```

Ilustración 2 Declaración Módulo FIR

```
FiltroFirAxi : Filtro_FIR_4in
    port map (
        i_clk        => S_AXI_ACLK,
        i_rstb       => S_AXI_ARESETN,
  -- coefficient
        i_coeff_0    => slv_reg0(7 downto 0),
        i_coeff_1    => slv_reg1(7 downto 0),
        i_coeff_2    => slv_reg2(7 downto 0),
        i_coeff_3    => slv_reg3(7 downto 0),
  -- data input
        i_data       => slv_reg4(7 downto 0),
        fir_valid    => s_valid,
  -- filtered data
        o_data       => so_data
    );
```

Ilustración 3 Instanciación y Conexionado

A pesar de que en los registros quedaban muchos bits “libres”, se eligió separar cada coeficiente y dato de entrada en registros diferentes para el mejor manejo en el SDK a la hora de escribir en dichos registros.

Para verificar que solo filtre cuando entra un nuevo dato, y puesto que se van a filtrar señales sinusoidales, se ha introducido una señal lógica que valdrá ‘1’ cuando el valor de entrada cambie y ‘0’ cuando este no cambie, que serán los momentos entre envío y envío del dato de entrada. Se condicionará todo el código relacionado con el filtrado a esta señal, haciendo así que solo se filtre cuando entra un nuevo dato. En la siguiente imagen se verifica el dato anterior leído (a_data) frente al nuevo dato de entrada (i_data).

```
if(a_data<i_data) then
    newdata<='1';
elsif(a_data>i_data) then
    newdata<='1';
else
    newdata<='0';
end if;
```

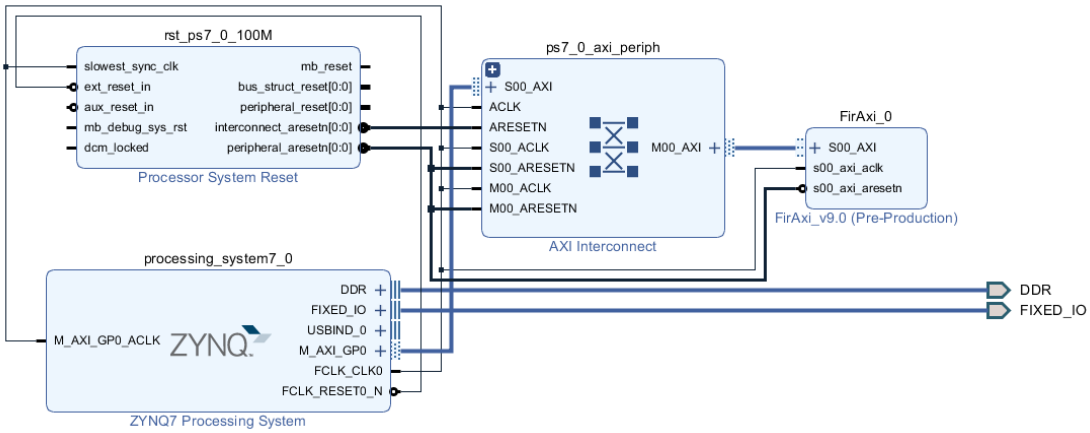
Ilustración 4 Asignación valor señal newdata

Para el dato de salida, se ha asignado a una señal su valor. Dicho valor se escribe en el registro 5 (***slv_reg5***) usando solo los 10 primeros bits. Dicha escritura en el registro 5 (para posteriormente leerlo en el proceso de lectura) se hace al final del proceso de escritura de los registros, de manera que, si por error se escribe en el registro 5, el valor que finalmente quedará guardado será el que queremos, el dato de salida.

```
  end if;
  slv_reg5(9 downto 0) <= so_data;
end process;
```

Ilustración 5 Escritura Dato de Salida

Realizado todo el conexionado de señales, se finaliza el módulo AXI de nuestro IP, por lo que se pasa al diseño de Bloques. El diseño de bloques generado es el siguiente:



El siguiente paso, fue la realización del Bare metal en SDK, donde se generó un código sencillo para la comprobación del correcto funcionamiento de nuestro módulo.

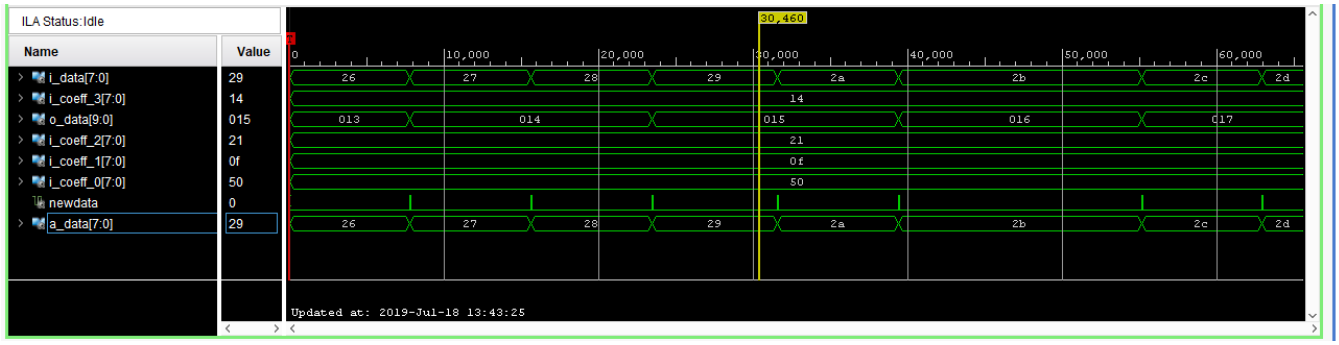
Dicha comprobación se ha realizado mediante una señal seno como dato de entrada, generada en un bucle for para cada punto de la señal y la escritura de los coeficientes.

```
#define PI 3.1415

int main()
{
    init_platform();
    int coeff_0 = 80;
    int coeff_1= 15;
    int coeff_2=33;
    int coeff_3=20;
    int in_data;
    int out_data;
    FIRAXI_mWriteReg(XPAR_FIRAXI_0_S00_AXI_BASEADDR, 0,coeff_0);
    FIRAXI_mWriteReg(XPAR_FIRAXI_0_S00_AXI_BASEADDR, 4,coeff_1);
    FIRAXI_mWriteReg(XPAR_FIRAXI_0_S00_AXI_BASEADDR, 8,coeff_2);
    FIRAXI_mWriteReg(XPAR_FIRAXI_0_S00_AXI_BASEADDR, 12,coeff_3);
    for(int i=0;i<360;i++)
    {
       in_data = 50*sin((i*PI)/180)+50;
       FIRAXI_mWriteReg(XPAR_FIRAXI_0_S00_AXI_BASEADDR, 16,in_data);
       printf("\n Data In = %d\n",in_data);
       for(int j=0; j<10000;j++)
       {
           ;
       }
       out_data = FIRAXI_mReadReg(XPAR_FIRAXI_0_S00_AXI_BASEADDR, 20);
       printf("Data Out %d = %d",i,out_data);
     }

     cleanup_platform();
     return 0;
}
```

Ilustración 6 Código SDK

Para reducir la complejidad con el tratamiento de valores negativos, se suma un valor de 50 (el mismo a la amplitud) para llevar la señal seno en su totalidad a la parte positiva.

En la siguiente imagen se ve el comportamiento de las diferentes señales del módulo VHDL gracias a que se han debuggeado con una ILA.



Posteriormente se ha pasado a implementar un sistema de Linux para sistemas embebidos para lanzarlo desde una SD en nuestra PYNQ. El sistema operativo de Linux escogido ha sido LINARO, ya que fue el que se usó en la práctica de la asignatura.

Tras implementar y configurar todo el sistema operativo con el Device Tree y un fichero boot, el cuál incorporaba el fichero bitstream, se pasa a generar un archivo .c. Este fichero será el equivalente al *helloworld.c* generado en el SDK, se encargará de generar la señal sinosoidal y de escribir en los registros de memoria usados para los coeficientes del filtro y el dato de entrada y finalmente leer el registro donde el filtro escribirá el dato de salida.

El código realizado ha sido el siguiente:

```
void main()
{
    int fd;
        FILE *fin,*fout;
    fd= open("/dev/uio0",O_RDWR);
        printf("\n HERE %d\n", fd);
    fin = fopen("DATAIN.txt","w");
    fout = fopen("DATAout.txt","w");


    int coeff_0 = 80;
    int coeff_1= 15;
    int coeff_2=33;
    int coeff_3=20;
    int in_data;
    int out_data;

    unsigned int *ptr = mmap(NULL, 0x1000, PROT_READ|PROT_WRITE, MAP_SHARED, fd,0);

    *ptr = coeff_0;
    *(ptr+1) = coeff_1;
    *(ptr+2) = coeff_2;
    *(ptr+3) = coeff_3;

        for(int i=0;i<360;i++)
        {

            in_data = 50*sin((i*PI)/180)+50;

            *(ptr+4)=in_data;
            printf("\n Data In = %d\n",in_data);
            for(int j=0; j<10;j++)
            {
                ;
            }
            out_data = *(ptr+5);
            printf("Data Out %d = %d",i,out_data);
                fprintf(fin,"%d\n", in_data);
                fprintf(fout,"%d\n", out_data);

         }
        fclose(fin);
        fclose(fout);
}
```

Ilustración 7 Código Linux

Se han guardado tanto los datos de entrada de la señal seno como los de salida del filtro para después verificar el filtrado mediante plots en Matlab. Las gráficas generadas se ven en la siguiente imagen:

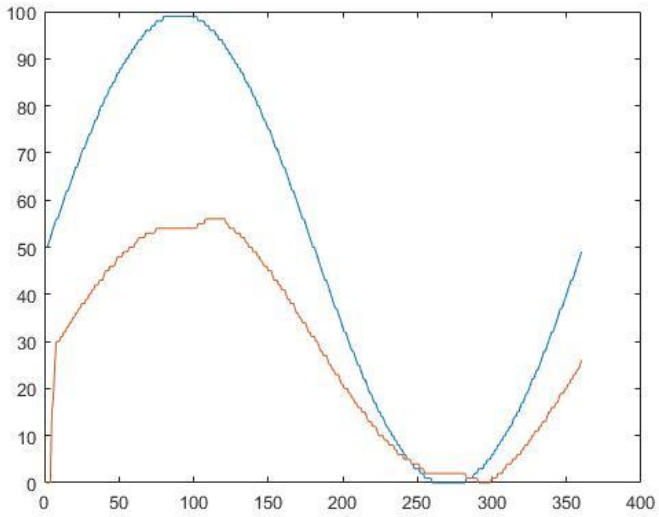


Ilustración 8 Señal de entrada con señal filtrada

Teniendo en cuenta que la señal de entrada es la señal azul y la de salida es la naranja, se apreciar que hay cierto retraso, pero es un retraso lógico debido a las etapas del filtro. También se puede apreciar el filtrado de la señal.