Universidad Politécnica de Cartagena MIEMBRO DE EUROPEAN UNIVERSITY OF TECHNOLOGY

Arquitectura Hardware de Comunicaciones

Proyecto Final:

Picoblaze de 16 Bits

Proyecto realizado por:

José David Guerrero Romero

Enrique Morales Bermejo

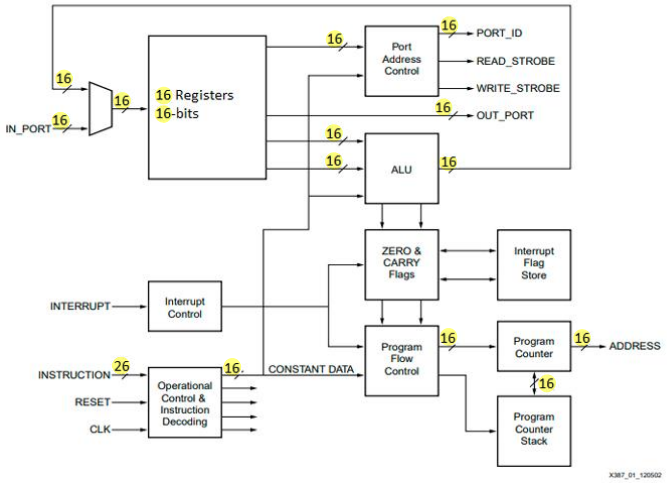
INDICE:

1. Visión general del proyecto

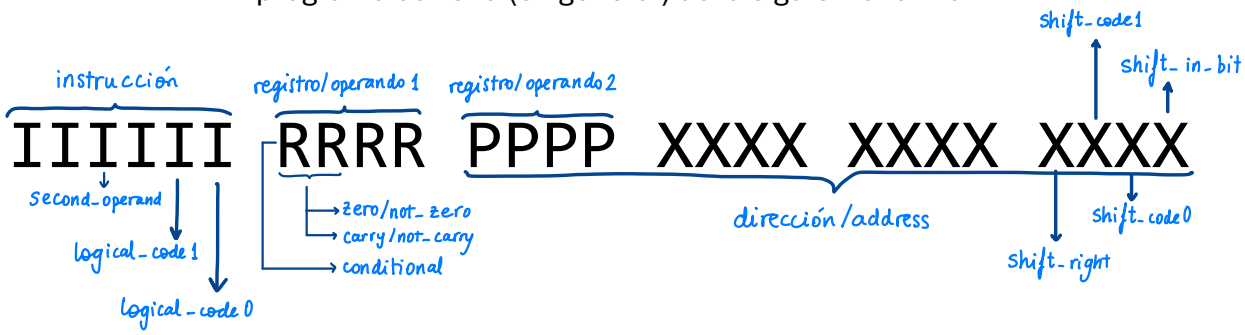
En este proyecto nos hemos propuesto llevar nuestra placa Spartan-3E de ***8 bits*** a ***16 bits***. En este primer apartado hablaremos en rasgos generales en los que enmarcamos el proyecto, siguiendo los siguientes cambios:

Característica	Picoblaze original	Picoblaze modificado
Arquitectura	8 bits	16 bits
Direccionamiento E/S	8 bits	16 bits
Tamaño de pila	4	8
Instrucciones adicionales	(FLIP)	MIX
Número de instrucciones	30	31
Tamaño instrucción	16	26
Longitud fichero ensamblador	1000	65536
Longitud del programa	256	65536
Número de registro	8	16

Nuestro programa, por tanto, se vería de la siguiente forma:



El programa se vería (en general) de la siguiente forma:



2. Modificación del proyecto

2.1 Modificaciones del código VHDL

Hay cambios bastante lógicos, como son cambiar todo lo que se ve con *(7 downto 0)* a *(15 downto 0)*, pero más allá de dichos cambios, hemos tenido que realizar las modificaciones que a continuación explicamos:

Uno de los primeros cambios, sugerido por el profesor José Javier, fue comentar los pines que mostramos en esta captura. Al cambiar la arquitectura a 16 bits, estos pines se acoplaban y nos limitaban el espacio para poder trabajar con nuestra arquitectura ampliada. Por ello, y por ser pines de depuración, el cambio que se nos sugirió fue comentarlos. De esta manera nos evitamos posibles problemas de solapamiento.

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

entity toplevel is
   Port (     -- port_id : out std_logic_vector(15 downto 0);  --solo para depurar
--           write_strobe : out std_logic;                     --solo para depurar
--            read_strobe : out std_logic;                     --solo para depurar
--               out_port : out std_logic_vector(15 downto 0); --solo para depurar
--                in_port : out std_logic_vector(15 downto 0); --solo para depurar
                 reset : in std_logic;
                   clk : in std_logic;
                    rx : in std_logic;
                    tx : out std_logic;
                    LED : out std_logic);    --led de comprobacion y reset
end toplevel ;

architecture behavioral of toplevel is
---------------------------------------------------------------
-- declaracion del picoblaze
---------------------------------------------------------------
  component picoblaze
    Port (      address : out std_logic_vector(15 downto 0);
            instruction : in std_logic_vector(25 downto 0);
                port_id : out std_logic_vector(15 downto 0);
           write_strobe : out std_logic;
               out_port : out std_logic_vector(15 downto 0);
            read_strobe : out std_logic;
                in_port : in std_logic_vector(15 downto 0);
              interrupt : in std_logic;
                  reset : in std_logic;
                    clk : in std_logic);
    end component;

---------------------------------------------------------------
-- declaración de la ROM de programa
---------------------------------------------------------------
  component programa_helloworld_int_FLIP
    Port (      address : in std_logic_vector(15 downto 0);
                   dout : out std_logic_vector(25 downto 0);
```

Además, adaptamos el nuevo tamaño de instrucción acorde a nuestros 16 bits (ahora siendo 26 bits en total). En varias de estas capturas se puede apreciar el cambio general de los 8 bits a los 16 bits en los *std_logic_vector*.

En la siguiente captura podemos comprobar como hemos adaptado también las *address* (direcciones) y el tamaño de las instrucciones en *instruction* de manera que sigan el esquema planteado anteriormente. También podemos apreciar que las señales previamente comentadas por el tema de los pines aparecen como “debugging”, de ahí la razón de no tenerlas en cuenta para nuestro programa.

```
-----------------------------------------------------------------
-- Signals usadas para conectar el picoblaze y la ROM de programa
-----------------------------------------------------------------
signal     address : std_logic_vector(15 downto 0);
signal instruction : std_logic_vector(25 downto 0);

-----------------------------------------------------------------
-- Signals para debugging
-----------------------------------------------------------------
signal readstrobe: std_logic;
signal writestrobe: std_logic;
signal portid: std_logic_vector(15 downto 0);
signal inport: std_logic_vector(15 downto 0);
signal outport: std_logic_vector(15 downto 0);
signal picoint: std_logic;
```

Para personalizar nuestro programa, decidimos cambiar las letras que mostraba la RAM (aunque no su tamaño, ya que no lo vimos necesario para escribir el texto que teníamos en mente) como mostramos a continuación:

```
type ram_type is array (0 to 63) of std_logic_vector (15 downto 0);      --Parte Declarativa  (se define un tipo) de la memoria RAM (le damos en la Bombilla, Síntesis, coding example, RAM, Dis
signal RAM : ram_type := (                                               --Mirar siempre los templates
x"0050", x"0049", x"0043", x"0020", x"0031", x"0036", x"0020", x"0042",
x"0049", x"0054", x"0053", x"0020", x"002D", x"0020", x"0032", x"0030",
x"0032", x"0035", x"000A", x"000D", x"004A", x"004F", x"0053", x"0045",
x"0044", x"0041", x"0020", x"003A", x"0029", x"0020", x"0045", x"004E",
x"0052", x"0049", x"0051", x"0055", x"0045", x"0020", x"000A", x"000D",
x"0050", x"0052", x"0045", x"0053", x"0053", x"0020", x"0041", x"004E",
x"0059", x"0020", x"004B", x"0045", x"0059", x"0020", x"0054", x"004F",
x"0020", x"0053", x"0054", x"0041", x"0052", x"0054", x"000A", x"000D");

                                                                         --En esta parte se declara el contenido de la RAM, cada celda tendra un valor inicial (en HEX)
                                                                         --Creamos una matriz de celdas de 0 a 64 bytes (En HEX de 0x00 a 0x3F), formadas por celdas de 8 bits
signal rxbuff_out,RAM_out: std_logic_vector(15 downto 0);
```

El mensaje que esconde nuestra RAM es el siguiente:

PIC 16 BITS – 2025
JOSEDA :) ENRIQUE
PRESS ANY KEY TO START

Dando un poco de personalidad al código…

Para el manejo de la recepción, como es de esperar, también hemos hecho varias modificaciones. Por supuesto, hemos añadido ceros hasta completar los 16 bits en total. Tendremos 15 ceros, y el bit más significativo es el bit de la recepción, bit *rx*, por lo que (si mantenemos como tenemos el código en ensamblador actualmente), de primeras, la recepción no se hará exitosamente... Más adelante iremos viendo las soluciones que hemos propuesto ante este problema.

```
   --añade 7ceros a rx para meterlos al puerto de entrada cuando se lea
   rxbuff:process(reset, clk)
   begin
      if (reset='1') then
         rxbuff_out <= (others=>'1');
      elsif rising_edge(clk) then
         if (readstrobe = '1' and portid =x"00FF") then
            rxbuff_out <= rx & "000000000000000";  ---anadir mas ceros duda
         end if;
      end if;
   end process;

   -- Memoria RAM (escritura sincrona / lectura asincrona)           --Aqui instanciamos la memoria RAM (previamente declarada en la zona Declarativa antes del begin)
   process (clk)                                                     --Si se declara y no se instancia, no se puede utilizar
   begin
      if (clk'event and clk = '1') then
         if (writestrobe = '1' and portid<x"0040") then
            RAM(to_integer(unsigned(portid))) <= outport;           --En todo este proceso escribimos valores de "outport" en nuestra memoria RAM
         end if;
      end if;
   end process;                                                      --Desde el principio es un proceso sincrono
   RAM_out <= RAM(to_integer(unsigned(portid)));                     --Este es asincrono, es la linea que se utiliza para LEER de la RAM

-- Multiplexor inport
inport <= RAM_out when (readstrobe = '1' and portid<x"0040") else    --Creamos un nuevo Multiplexor (MUX) haciendo uso de las plantillas (es el de 2-to-1(assign)) es como un conmutador
          rxbuff_out when (readstrobe = '1' and portid=x"00FF") else --La salida del multiplexor (input) sera RAM_out (de la memoria RAM) mediante una serie de condiciones (selector)
          x"0000";

end behavioral;
```

Todo lo visto hasta el momento corresponde al archivo *toplevel*.

Lo que vemos a continuación corresponde al archivo *processor*.

$$2^4 = 16\, registros.$$

$$2^3 = Tamaño\ de\ la\ pila$$

$$2^{16} = 65536\, posiciones\ de\ la\ memoria\ de\ programa.$$

```
architecture Behavioral of picoblaze is
--
-- Size of register bank, stack counter can be changed here
--
constant register_bank_address : natural := 4; -- 16 registers
constant stack_counter_address : natural := 3; -- 8 program stack address
--
-- size of program counter should not be changed
--
constant program_counter_address : natural := 16; -- 65536 program word
```

```
architecture v1 of programa_helloworld_int_FLIP is

   constant ROM_WIDTH: INTEGER:= 26;
   constant ROM_LENGTH: INTEGER:= 65536;
```

Para adaptar las instrucciones a la nueva longitud (de 6 bits, antes 5), hemos añadido un bit a la izquierda de todas y cada una de las instrucciones (bit más significativo).

```
constant jump_id : std_logic_vector(5 downto 0) := "011010";
constant call_id : std_logic_vector(5 downto 0) := "011011";
constant return_id : std_logic_vector(5 downto 0) := "010010";
--
-- logical group
constant load_k_to_x_id : std_logic_vector(5 downto 0) := "000000";
constant load_y_to_x_id : std_logic_vector(5 downto 0) := "001000";
constant and_k_to_x_id : std_logic_vector(5 downto 0) := "000001";
constant and_y_to_x_id : std_logic_vector(5 downto 0) := "001001";
constant or_k_to_x_id : std_logic_vector(5 downto 0) := "000010";
constant or_y_to_x_id : std_logic_vector(5 downto 0) := "001010";
constant xor_k_to_x_id : std_logic_vector(5 downto 0) := "000011";
constant xor_y_to_x_id : std_logic_vector(5 downto 0) := "001011";
--
-- arithmetic group
constant add_k_to_x_id : std_logic_vector(5 downto 0) := "000100";
constant add_y_to_x_id : std_logic_vector(5 downto 0) := "001100";
constant addcy_k_to_x_id : std_logic_vector(5 downto 0) := "000101";
constant addcy_y_to_x_id : std_logic_vector(5 downto 0) := "001101";
constant sub_k_to_x_id : std_logic_vector(5 downto 0) := "000110";
constant sub_y_to_x_id : std_logic_vector(5 downto 0) := "001110";
constant subcy_k_to_x_id : std_logic_vector(5 downto 0) := "000111";
constant subcy_y_to_x_id : std_logic_vector(5 downto 0) := "001111";

--
-- shift and rotate
constant shift_rotate_id : std_logic_vector(5 downto 0) := "010100";
--
```

En la captura siguiente comprobamos que hemos trasladado el bit *logical_code1* de su posición 12 (si contamos el primer bit como 0) a la posición 21 (como reflejamos en el esquema del principio). También hemos cambiado el bit *logical_code0* a la posición anterior de 21 (20). Ambos deben ir seguidos. Además, la selección de las instrucciones siguientes

```
-- set shift decoding bits
shift_right <= instruction(3);
shift_in_bit <= instruction(0);
shift_code1 <= instruction(2);
shift_code0 <= instruction(1);
logical_code1 <= instruction(21); --cambiamos de 12 a 21
logical_code0 <= instruction(20); --cambiamos de 11 a 20

-- set instruction decoding signals
i_jump <= '1' when instruction(25 downto 20) = jump_id else '0'; -- cambiams de 15 downto 11 a 25 downto 20
i_call <= '1' when instruction(25 downto 20) = call_id else '0';
i_return <= '1' when instruction(25 downto 20) = return_id else '0';
i_returni <= '1' when instruction(25 downto 20) = returni_id else '0';
     i_load_k_to_x <= '1' when instruction(25 downto 20) = load_k_to_x_id else '0';
     i_load_y_to_x <= '1' when instruction(25 downto 20) = load_y_to_x_id else '0';
     i_and_k_to_x <= '1' when instruction(25 downto 20) = and_k_to_x_id else '0';
     i_and_y_to_x <= '1' when instruction(25 downto 20) = and_y_to_x_id else '0';
     i_or_k_to_x <= '1' when instruction(25 downto 20) = or_k_to_x_id else '0';
     i_or_y_to_x <= '1' when instruction(25 downto 20) = or_y_to_x_id else '0';
     i_xor_k_to_x <= '1' when instruction(25 downto 20) = xor_k_to_x_id else '0';
     i_xor_y_to_x <= '1' when instruction(25 downto 20) = xor_y_to_x_id else '0';
     i_add_k_to_x <= '1' when instruction(25 downto 20) = add_k_to_x_id else '0';
     i_add_y_to_x <= '1' when instruction(25 downto 20) = add_y_to_x_id else '0';
     i_addcy_k_to_x <= '1' when instruction(25 downto 20) = addcy_k_to_x_id else '0';
     i_addcy_y_to_x <= '1' when instruction(25 downto 20) = addcy_y_to_x_id else '0';
     i_sub_k_to_x <= '1' when instruction(25 downto 20) = sub_k_to_x_id else '0';
     i_sub_y_to_x <= '1' when instruction(25 downto 20) = sub_y_to_x_id else '0';
     i_subcy_k_to_x <= '1' when instruction(25 downto 20) = subcy_k_to_x_id else '0';
     i_subcy_y_to_x <= '1' when instruction(25 downto 20) = subcy_y_to_x_id else '0';
     i_input_p_to_x <= '1' when instruction(25 downto 20) = input_p_to_x_id else '0';
     i_input_y_to_x <= '1' when instruction(25 downto 20) = input_y_to_x_id else '0';
     i_output_p_to_x <= '1' when instruction(25 downto 20) = output_p_to_x_id else '0';
     i_output_y_to_x <= '1' when instruction(25 downto 20) = output_y_to_x_id else '0';
     i_interrupt <= '1' when instruction(25 downto 20) = interrupt_id else '0';
i_shift_rotate <= '1' when instruction(25 downto 20) = shift_rotate_id else '0';
```

Ahora existen 65536 posiciones de la memoria de programa por lo que la interrupción que antes se ejecutaba en la última posición de memoria (256), ahora se ejecuta en la posición 65535 (como existe líneas de código anteriores a la memoria ROM en la figura se muestra como si fuese la línea 65555).

```
65553    "00000000000000000000000000000000",
65554    "00000000000000000000000000000000",
65555    "01101000000000000000011100000");
           jump                  interrupt

  47     "10100110000000000000000000000000",
           mix
```

Aquí vemos cómo se traduce, correctamente, la instrucción MIX.

Esta parte corresponde al *Arithmetic Group* (Grupo Aritmético). Aquí los cambios se reflejan de nuevo en el tamaño que ahora pasa a ser de 16 bits.

```
entity arithmetic_process is
    Port (  first_operand : in std_logic_vector(15 downto 0);
           second_operand : in std_logic_vector(15 downto 0);
                 carry_in : in std_logic;
                    code1 : in std_logic;
                    code0 : in std_logic;
                        Y : out std_logic_vector(15 downto 0);
                carry_out : out std_logic;
                      clk : in std_logic);
    end arithmetic_process;
--
```

Dichos cambios también se reflejan en el *Arithmetic process*

```
-- An 16-bit arithmetic process
--
component arithmetic_process
    Port (first_operand : in std_logic_vector(15 downto 0);
          second_operand : in std_logic_vector(15 downto 0);
          carry_in : in std_logic;
          code1 : in std_logic;
          code0 : in std_logic;
          Y : out std_logic_vector(15 downto 0);
          carry_out : out std_logic;
          clk : in std_logic);
    end component;
```

En el fragmento *zero_logic* hemos realizado los siguientes cambios acorde a la arquitectura actual de 16 bits.

	8 bits	***16 bits***
Lower_Zero	*Data(3) - Data(0)*	*Data(7) - Data(0)*
Upper_Zero	*Data(7) - Data(4)*	*Data(15) - Data(8)*

```
-- Detect all bits in data are zero using wired NOR gate

lower_zero <= (not data(7)) and (not data(6)) and (not data(5)) and (not data(4)) and (not data(3)) and (not data(2)) and (not data(1)) and (not data(0));
upper_zero <= (not data (15)) and (not data(14)) and (not data(13)) and (not data(12)) and (not data(11)) and (not data(10)) and (not data(9)) and (not data(8));

      lower_zero_carry <= lower_zero;
      data_zero <= (upper_zero) and lower_zero_carry;
```

2.2. Modificaciones del código C

Una modificación lógica, pero a la vez sumamente importante, es realizar los cambios en el formato de las instrucciones, de manera que al compilar el código en C se traduzcan de manera correcta. Por tanto, vemos que es idéntico a como lo tenemos en nuestro código de VHDL.

```
/* program control group */
char *jump_id = "011010";
char *call_id = "011011";
char *return_id = "010010";

/* logical group */
char *load_k_to_x_id = "000000";
char *load_y_to_x_id = "001000";
char *and_k_to_x_id = "000001";
char *and_y_to_x_id = "001001";
char *or_k_to_x_id = "000010";
char *or_y_to_x_id = "001010";
char *xor_k_to_x_id = "000011";
char *xor_y_to_x_id = "001011";

/* arithmetic group */
char *add_k_to_x_id = "000100";
char *add_y_to_x_id = "001100";
char *addcy_k_to_x_id = "000101";
char *addcy_y_to_x_id = "001101";
char *sub_k_to_x_id = "000110";
char *sub_y_to_x_id = "001110";
char *subcy_k_to_x_id = "000111";
char *subcy_y_to_x_id = "001111";

/* shift and rotate */
char *shift_rotate_id = "010100";
```

Otro cambio a destacar es que, al tener 16 bits, podemos tener hasta 16 registros del tipo s (que van de *s0* a *s15*). Por ello, nos hemos visto en la necesidad de cambiar la parte que adjuntamos a continuación del código en C, de manera que sepa distinguir ahora para el caso de tener “tres caracteres”. Concretamente, distingue si el primer carácter es una “s”; si el segundo carácter está comprendido entre el “0” y el “9”; y si el tenemos tres caracteres verifica que el segundo carácter sea un “1” siempre, y que tercer carácter esté comprendido entre “0” y “5” (para llegar a los 15).

```
/* Only S0 - S15 are valid */                                   // Comprueba la sintaxis de los registros.
int register_number(char *s)
{
    if(*s != 'S') return (-1);
    if(strlen(s) < 2 || strlen(s) > 3) return (-1);
    if(strlen(s) == 2){
        if((*(s+1) >= '0') && (*(s+1) <= '9')) return (*(s+1) - '0');
        else return( -1 );
    }

    if(strlen(s) == 3){
        if(*(s+1) == '1' && (*(s+2) >= '0') && (*(s+2) <= '5')) return (10*(*(s+1) - '0') + (*(s+2) - '0'));
        else return( -1 );
    }
    else return( -1 );
}
```

Para poder adaptar ahora el desplazamiento y encontrar correctamente en el programa qué bits corresponden a las instrucciones, ahora tenemos que hacer un desplazamiento

de 20 posiciones (de manera que empiecen en la posición 26, que en realidad sería la 25 si pensamos con la lógica de que el primer bit es 0), para que cada instrucción se lea, compruebe y ejecute correctamente. Vemos también cambios de posiciones relevantes para los bits que se muestran.

```
/*====================================== */
void insert_instruction(char *s, int p)                         // crea codigo máquina: campo de instrucciones
{
    int i, l;
    unsigned n = 0;

    l = strlen(s);
    for(i = 0; i < l; i++)
        if(*(s+i) == '1')
            n = n + (unsigned) pow(2, (l-i-1));

    program_word[p] = program_word[p] | (n << 20);  // Esto es debido a que la nueva instruccion tiene 6 bits y hay que despalazarla 20 sitios para que empiece en el 26
}

/*====================================== */
void insert_sXX(int c, int p)                                   // crea codigo máquina: campo 1er registro
{
    program_word[p] = program_word[p] | (unsigned) (c << 16);
}

/*====================================== */
void insert_sYY(int c, int p)                                   // crea codigo máquina: campo 2er registro
{
    program_word[p] = program_word[p] | (unsigned) (c << 12);
}

/*====================================== */
void insert_constant(int c, int p)                              // crea codigo máquina: campo Inmediato
{
    program_word[p] = program_word[p] | (unsigned) (c);
}

/*====================================== */
void insert_flag(int c, int p)                                  // crea codigo máquina: campo Flag
{
    program_word[p] = program_word[p] | (unsigned) (c << 17);           //para que llegue a la nueva longitud
}
```

Para conseguir que al compilar y ejecutar nuestro asm.exe nos devuelva el programa correctamente en formato VHDL, tenemos que realizar los mismos cambios vistos en el código VHDL de manera que coincidan y sea todo correcto en cuanto a longitudes de los formatos.

```
void write_vhd(void)
{
    int i, j;
    char *ptr;
    char basename[200];

    ptr = strstr(filename, ".mcs");
    *ptr = '\0';
    strcpy(basename, filename);
    strcat(filename,".vhd");

    ffp = fopen(filename, "w");
    if (ffp == NULL){
        printf("\nCan not open vhd file\n");
        exit(1);
    }

    fprintf(ffp,"library ieee;\nuse ieee.std_logic_1164.all;\n");
    fprintf(ffp,"use ieee.std_logic_unsigned.all;\n\n");

    fprintf(ffp,"entity %s is\n",basename);
    fprintf(ffp,"\tport( address : in std_logic_vector(15 downto 0);\n");
    fprintf(ffp,"\t\tclk : in std_logic;\n\t\tdout : out std_logic_vector(25 downto 0));\n\tend;\n\n");
    fprintf(ffp,"architecture v1 of %s is\n\n", basename);

    fprintf(ffp,"\tconstant ROM_WIDTH: INTEGER:= 26;\n");           //Cambiamos el size que tendra acada elemento de la ROM
    fprintf(ffp,"\tconstant ROM_LENGTH: INTEGER:= 65536;\n\n");     //En total tendremos 65536 lineas para almacenar instrucciones del codigo C en nuestra ROM
    fprintf(ffp,"\tsubtype rom_word is std_logic_vector(ROM_WIDTH-1 downto 0);\n");
    fprintf(ffp,"\ttype rom_table is array (0 to ROM_LENGTH-1) of rom_word;\n\n");
    fprintf(ffp,"constant rom: rom_table := rom_table'(\n");
    for(i = 0; i < PROGRAM_COUNT-1; i++){
        fprintf(ffp, "\t\"");
        for(j = 25; j >= 0; j--)
            fprintf(ffp, "%d", (program_word[i]>>j) & 1); //print binary
        fprintf(ffp, "\",\n");
    }
    fprintf(ffp, "\t\"");
    for(j = 25; j >= 0; j--)
        fprintf(ffp, "%d", (program_word[i]>>j) & 1); //print binary
    fprintf(ffp, "\");\n\n");

    fprintf(ffp,"begin\n\nprocess (clk)\nbegin\n", basename);
    fprintf(ffp,"\tif clk'event and clk = '1' then\n\t\tdout <= rom(conv_integer(address));\n");
    fprintf(ffp,"\tend if;\nend process;\nend v1;\n");

    fclose(ffp);
}
```

2.3. Modificaciones del código Ensamblador

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
CONSTANT        rs232, 00FF             ; puerto comunicacion serie es el FF
                                        ; rx es el bit 0 del puerto FF(entrada)
                                        ; tx es el bit 7 del puerto FF(salida), esto es porque
;el hyperterminal envia primero el LSB, por eso vamos desplazando a la
;izquierda al recibir, y al enviar tambien, con lo que enviamos de nuevo
;el LSB primero como corresponde para que lo entienda el hyperterminal
NAMEREG         s12, txreg              ;buffer de transmision cambiado para probar nuevas registros
NAMEREG         s8, rxreg               ;buffer de recepcion cambiado para probar nuevas registros
NAMEREG         s3, contbit             ;contador de los 8 bits de datos
NAMEREG         s4, cont1               ;contador de retardo1
NAMEREG         s5, cont2               ;contador de retardo2
;
ADDRESS         0000                    ;el programa se cargara a partir de la dir 0000
.............................................
```

	8 bits	16 bits
Tamaño Hexadecimal	0x00	0x0000
Registros	S0 - S7	S0-S15, nuevos registros
Rotaciones	8 rotaciones	16 rotaciones
Rxbuff	0x80	0x8000

Uno de los fallos que cometimos al principio (y por el que no nos funcionaba correctamente la recepción al pulsar una tecla) era que pusimos 0x0080, pero teníamos el bit de recepción *rx* en la posición 16, por lo que no nos estaba tomando el valor recibido si no el de un "0". Este fue nuestro paso a tomar dos vías:

- Cambiar posteriormente el número de iteraciones para que avanzase las 16 posiciones y tomase correctamente el valor de *rx*, como veremos más adelante.

- Mantener el código de la misma forma que teníamos originalmente, pero creando una nueva instrucción (que como vimos en los trabajos de años anteriores colgados en el corcho de la clase, la llamaban SWAP), que nosotros decidimos llamar MIX, de manera que se cambiasen los 8 primeros bits con los 8 últimos, poniendo en la posición 16 el bit de *rx* que estaba en la posición 8 (0x0080) pase a estar donde corresponde para ser leído correctamente.

```
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;Rutina de recepcion de caracteres
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
recibe:         ;esperamos a que se reciba un bit de inicio
                INPUT           rxreg, rs232
                AND             rxreg, 8000     ;ahora que usamos MIX se vuelve a tomar el valor del bit numero 0080 antes 8000 (no del final) ?
                JUMP            NZ, recibe
                CALL            wait_05bit
                ;almacenamos los 8 bits de datos
                LOAD            contbit,0011    ;sin mix, esto nos permite llevar el bit 16 a la posicion deseada
next_rx_bit:    CALL            wait_1bit
                SR0             rxreg
                INPUT           s0, rs232
                AND             s0,8000 ;ahora que usamos MIX se vuelve a tomar el valor del bit numero 0080 antes 8000 (no del final) ?
                OR              rxreg, s0
                SUB             contbit, 0001
                JUMP            NZ, next_rx_bit
                RETURN
```

3. Nueva instrucción MIX

Pasos a seguir para añadir una instrucción.

- Modificar herramienta ensambladora (asm.ccp):
  - 1º Añadir nuevo código de operación.
  - 2º Añadir nuevo nemónico al juego de instrucciones.
  - 3º Incrementar el número de instrucciones reconocibles
  - 4º Añadir instrucción en el parser de chequeo de sintaxis.
  - 5º Añadir instrucción en el parser de decodificación de Código Máquina

3.1. Modificaciones del código VHDL

101001 ahora tienen 6 bits las instrucciones, el tercer bit más significativo corresponde a 0 si no se utiliza segundo registro, y 1 si se utiliza segundo registro. En nuestro caso, aunque esté a 1 y tras varias pruebas, hemos concluido que es mejor utilizar un solo registro para operar en la instrucción.

```
-- decode second operand
second_operand <= sY_register when instruction(23) = '1' else instruction(15 downto 0); --tendremos que tomar el valor referente al bit 23 de instruction!
```

```
-- added new instruction
-- flip
constant flip_id : std_logic_vector(5 downto 0) := "011111";        --nos permite identificar la instruccion flip con sus respectivos bits 11111
constant mix_id : std_logic_vector(5 downto 0) := "101001";         -- Nueva instruccion MIX con su respectivo identificador
```

Una primera parte para añadir una instrucción es crear su definición, como mostramos a continuación (vemos que tenemos *operand*, que es el valor que recibe, e *Y,* que es el valor final que nos dará la instrucción MIX).

```
-- Definition of mix process
--
component mix  -- Zona de definicion del componente MIX
    Port (operand : in std_logic_vector(15 downto 0);
          Y : out std_logic_vector(15 downto 0);
          clk : in std_logic);
    end component;
```

En esta captura podemos comprobar que para el componente de *register_and_flag_enable* que se encarga de generar banderas (de las señales que contiene) para detectar si se está realizando alguna de dichas acciones en el código.

```
-- Decoding and timing of write enable for register bank and clock enable for flags
--
component register_and_flag_enable
    Port (i_logical: in std_logic;
          i_arithmetic: in std_logic;
          i_shift_rotate: in std_logic;
          i_flip: in std_logic;              -- added new instruction
          i_mix: in std_logic;               -- added new instruction
          i_returni: in std_logic;
          i_input: in std_logic;
          active_interrupt : in std_logic;
          T_state : in std_logic;
          register_enable : out std_logic;
          flag_enable : out std_logic;
          clk : in std_logic);
    end component;
```

Conexión auxiliar para llevar la señal de MIX.

```
-- added new instruction
signal i_flip : std_logic;
signal i_mix : std_logic;        -- Creamos la conexion para llevar la señal MIX
```

Creación de las señales que utilizará la ALU para las distintas instrucciones (añadida la de MIX).

```
-- ALU signals
--
signal second_operand          : std_logic_vector(15 downto 0);
signal logical_result          : std_logic_vector(15 downto 0);
signal shift_and_rotate_result : std_logic_vector(15 downto 0);
signal shift_and_rotate_carry  : std_logic;
signal arithmetic_result       : std_logic_vector(15 downto 0);
signal arithmetic_carry        : std_logic;
signal ALU_result              : std_logic_vector(15 downto 0);
signal flip_result             : std_logic_vector(15 downto 0);
signal mix_result              : std_logic_vector(15 downto 0);
```

Aquí mostramos el Mapeado de puertos de MIX.

```
mix_group: mix
port map (operand => sX_register,
          Y => mix_result,
          clk => clk);
```

```
Port map (i_logical => i_logical,
          i_arithmetic => i_arithmetic,
          i_shift_rotate => i_shift_rotate,
          i_flip => i_flip,          -- added new instruction
          i_mix => i_mix,            -- added new instruction
          i_returni => i_returni,
          i_input => i_input,
          active_interrupt => active_interrupt,
          T_state => T_state,
          register_enable => register_write_enable,
          flag_enable => flag_clock_enable,
          clk => clk);
```

Se comprueban los bits que se encuentran en la posición de la instrucción. Si los bits coinciden con los de la instrucción MIX se pone a 1, en otro caso es 0.

```
-- added new instruction
   i_flip <= '1' when instruction(25 downto 20) = flip_id else '0';
   i_mix <= '1' when  instruction(25 downto 20) = mix_id else '0';      -- added new instruction
```

Aquí entramos en el bucle de la ALU para poder obtener el resultado esperado de la instrucción que se haya realizado, y guardarlo en *ALU_result*.

```
   -- get ALU result
   --
   ALU_loop: for i in 0 to 15 generate
   begin
      ALU_result(i) <= (shift_and_rotate_result(i) and i_shift_rotate)
                    or (in_port(i) and i_input)
                    or (arithmetic_result(i) and i_arithmetic)
                    or (flip_result(i) and i_flip)       -- added new instruction
                    or (mix_result (i) and i_mix)        -- added new instruction
                    or (logical_result(i) and i_logical);
   end generate ALU_loop;
```

A partir de aquí, nos centramos en el código VHDL de *mix_group*.

Hacemos uso de un solo operando que intercambia los 8 bits más significativos con los 8 bits menos significativos cada flanco de reloj.

```
--------------------------------------------------------------------------------
-- Definition of a 16-bit mix process
-- Operation:
-- The input operand is split and mixed. The lower 8 bits are moved to the upper part,
-- and the upper 8 bits are moved to the lower part.
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mix is
    Port (
        operand : in std_logic_vector(15 downto 0); -- 16-bit input
        Y : out std_logic_vector(15 downto 0);      -- 16-bit output
        clk : in std_logic                          -- Clock signal
    );
end mix;

architecture low_level_definition of mix is
begin
    FF: process (clk)
    begin
        if rising_edge(clk) then
            -- Intercambia los 8 ultimos con los 8 primeros bits
            Y <= operand(7 downto 0) & operand(15 downto 8);
        end if;
    end process FF;
end low_level_definition;
```

En el fragmento *reg_and_flag_enables* si no hay ninguna interrupción activa, se decodifica la instrucción MIX en este caso de que esté a 1.

```
begin
  --
  -- Register enable
  --
  -- added new instruction, uncomment this instruction and comment next instruction
  -- to enable the new instruction
   reg_instruction_decode <= (i_logical or i_arithmetic or i_shift_rotate or i_input or i_flip or i_mix)    -- added new instruction
                      and (not active_interrupt);
```

Comrpobamos en el proceso que manejamos con *reg_instruction_decode* que tengamos, por ejemplo, *i_mix* funcionando y no tengamos ninguna *active_interrupt* (ninguna interrupción activa).

```
entity register_and_flag_enable is
    Port (i_logical: in std_logic;
          i_arithmetic: in std_logic;
          i_shift_rotate: in std_logic;
          i_flip: in std_logic;              -- added new instruction
          i_mix: in std_logic;               -- added new instruction
          i_returni: in std_logic;
          i_input: in std_logic;
          active_interrupt : in std_logic;
          T_state : in std_logic;
          register_enable : out std_logic;
          flag_enable : out std_logic;
          clk : in std_logic);
    end register_and_flag_enable;
```

3.2. Modificaciones del código C

Aumentamos en 1 el número de instrucciones totales (antes con FLIP, que partimos de dicho código, teníamos un total de 30 instrucciones), que ahora pasa a ser 31.

```
/* increase instruction_count for added new instruction */
#define instruction_count 31 /* total instruction set */            // Numero maximo que tenemos en el codigo para usar
```

Especificamos el id de MIX para que se haga correctamente la traducción tras compilar y ejecutar *asm.exe*.

```
*mix_id = "101001";                    // Identificador para la nueva instruccion MIX, lo hemos hecho distinto al resto de manera que sea facilmente reconocible
                                       //(sobre todo para comprobaciones)/* input/output group */
```

Añadimos la instrucción en la posición 30 (última posición, haciendo el total de 31 instrucciones).

```
char *instruction_set[] = {
    "JUMP",      /* 0 */
    "CALL",      /* 1 */
    "RETURN",    /* 2 */
    "LOAD",      /* 3 */
    "AND",       /* 4 */
    "OR",        /* 5 */          //Se van recorriendo co
    "XOR",       /* 6 */
    "ADD",       /* 7 */
    "ADDCY",     /* 8 */
    "SUB",       /* 9 */
    "SUBCY",     /* 10 */
    "SR0",       /* 11 */
    "SR1",       /* 12 */
    "SRX",       /* 13 */
    "SRA",       /* 14 */
    "RR",        /* 15 */
    "SL0",       /* 16 */
    "SL1",       /* 17 */
    "SLX",       /* 18 */
    "SLA",       /* 19 */
    "RL",        /* 20 */
    "INPUT",     /* 21 */
    "OUTPUT",    /* 22 */
    "RETURNI",   /* 23 */
    "ENABLE",    /* 24 */
    "DISABLE",      /* 25 */
    "CONSTANT", /* 26 */
    "NAMEREG",   /* 27 */
    "ADDRESS",   /* 28 */
    "FLIP",      /* 29 */ /* added new instruction */
    "MIX"};      /* 30 */ /* added new instruction */
```

Aprovechamos el código de los anteriores case para operandos sobrantes o escasos para la instrucción MIX.

```
case 29: /* FLIP */ /* added new instruction, same syntax with shift/rotate */
case 30: /* MIX */ /* added new instruction*/
if(op[i].op2 != NULL){
        printf("ERROR - Too many Operands for %s on line %d\n", op[i].instruction, i+1);
        fprintf(ofp,"ERROR - Too many Operands for %s on line %d\n", op[i].instruction, i+1);
        error++;
    } else if(op[i].op1 == NULL){
        printf("ERROR - Missing operand for %s on line %d\n", op[i].instruction, i+1);
        fprintf(ofp,"ERROR - Missing operand for %s on line %d\n", op[i].instruction, i+1);
        error++;
    } break;
```

Aprovechamos el código de FLIP para MIX. Se inserta su código máquina de operación.

```
case 29: /* FLIP */ /* added new instruction */                    // para el caso de instrucciones FLIP.
case 30: /* MIX */ /* added new instruction */
    if(j == 29) insert_instruction(flip_id, op[i].address);              // inserta codigo máquina del codigo de operacion.
    if(j == 30) insert_instruction(mix_id, op[i].address);
    if((reg_n = find_namereg(op[i].op1)) != -1)                     // Solo tiene un operando que siempre sera un registro o un namereg, sino error.
        insert_sXX(reg_n, op[i].address);                           // inserta codigo maquina del 2do operando si es namereg
    else if((reg_n = register_number(op[i].op1)) != -1)
        insert_sXX(reg_n, op[i].address);                           // inserta codigo máquina del 2do operando si es registro
    else {
        printf("ERROR - Invalid operand %s on line %d\n",op[i].op1, i+1);
        fprintf(ofp,"ERROR - Invalid operand %s on line %d\n",op[i].op1, i+1);
        error++;
    }
    break;
```

Como al final no utilizamos dos registros para efectuar la operación de MIX, no fue necesario implementar *sptr* para el caso de j == 30m, ya que nos hemos apañado con un solo registro para lograrlo (mucho más sencillo, eficiente, y fácil de entender de un solo vistazo).

```
if(j == 3){ kptr = load_k_to_x_id; sptr = load_y_to_x_id;}
if(j == 4){ kptr = and_k_to_x_id; sptr = and_y_to_x_id;}
if(j == 5){ kptr = or_k_to_x_id; sptr = or_y_to_x_id;}
if(j == 6){ kptr = xor_k_to_x_id; sptr = xor_y_to_x_id;}
if(j == 7){ kptr = add_k_to_x_id; sptr = add_y_to_x_id;}
if(j == 8){ kptr = addcy_k_to_x_id; sptr = addcy_y_to_x_id;}
if(j == 9){ kptr = sub_k_to_x_id; sptr = sub_y_to_x_id;}
if(j == 10){ kptr = subcy_k_to_x_id; sptr = subcy_y_to_x_id;}
```

3.3. Modificaciones del código Ensamblador

Como hemos comentado anteriormente, ahora "cometemos a propósito" el error que surgía al comienzo de la implementación de los 16 bits. Se hace un total de 8 rotaciones dejando la información deseada a mitad de la trama. Para recibir la información correctamente habría que cambiar los 8 bits de más significativos con los 8 bits menos significativos. Realizamos una llamada a la instrucción MIX en *rxreg* una vez hayan finalizado las rotaciones, quedando finalmente en el bit más significativo la información que queremos recibir.

```
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;Rutina de recepcion de caracteres
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
recibe:         ;esperamos a que se reciba un bit de inicio
                INPUT           rxreg, rs232
                AND             rxreg, 8000     ;ahora que usamos MIX se vuelve a tomar el valor del bit numero 0080 antes 8000 (no del final) ?
                JUMP            NZ, recibe
                CALL            wait_05bit
                ;almacenamos los 8 bits de datos
                LOAD            contbit,0009    ;sin mix, esto nos permite llevar el bit 16 a la posicion deseada
next_rx_bit:    CALL            wait_1bit
                SR0             rxreg
                INPUT           s0, rs232
                AND             s0,8000 ;ahora que usamos MIX se vuelve a tomar el valor del bit numero 0080 antes 8000 (no del final) ?
                OR              rxreg, s0
                SUB             contbit, 0001
                JUMP            NZ, next_rx_bit
                MIX             rxreg
                RETURN
```