

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería de
Telecomunicación

Implementación de una microarquitectura de 32 bits basada en el microcontrolador PicoBlaze para la aceleración por hardware del algoritmo de codificación Hamming

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA TELEMÁTICA



Autor: Pablo Ruiz Abellán

Director: Jose Javier Martínez Álvarez
Co-director: F. Javier Garrigós Guerrero
Cartagena, 09/09/2022

Agradecimientos

A mis compañeros y amigos, por regalarme todos estos años buenos momentos y apoyo, a mis profesores y todo personal encargado de hacer posible formarnos y aprender, pero, sobre todo, a mis padres, la paciencia y dedicación personificados que me han permitido llegar hasta aquí.

Tabla de contenido

1.	Introducción y objetivos.....	3
1.1.	Introducción	3
1.2.	Propósito	3
1.3.	Fases del trabajo	4
1.4.	Visión global del proyecto	4
2.	Plataforma de implementación.....	5
3.	Descripción del microcontrolador PicoBlaze a 8 bits	6
3.1.	Estructura interna	6
3.2.	Set de instrucciones disponibles	7
3.3.	Programa del microcontrolador.....	11
3.4.	Diseño de la herramienta ensambladora	13
3.5.	Puertos de entrada y salida del sistema	19
3.6.	Herramientas adicionales.....	20
4.	Migración de PicoBlaze a 32 bits.....	21
4.1.	Rediseño del PicoBlaze	21
4.2.	Modificación del código VHDL	22
4.2.1.	Modificación del fichero toplevel.....	22
4.2.2.	Modificación del bloque picoblaze	24
4.2.3.	Resultados de implementación del PicoBlaze a 32 bits	29
4.3.	Adaptación de la herramienta ensambladora	31
4.4.	Ejemplo de programa ensamblador.....	39
4.5.	Prueba de funcionamiento del microcontrolador a 32 bits.....	41
5.	Implementación del módulo SECDEC.....	61
5.1.	Diseño del periférico	61
5.2.	Migración del módulos SECDEC de 8 bits a 32 bits	62
5.3.	Conexión del Módulo SECDEC al Sistema	65
5.4.	Resultados y test de funcionamiento.....	67
6.	Conclusiones y líneas futuras	75
	Referencias.....	77
	Índice de figuras	78
	Índice de tablas	80

1. Introducción y objetivos

1.1. Introducción

El microcontrolador PicoBlaze es un dispositivo utilizado con fines educativos. Está desarrollado en VHDL y cuenta con una herramienta con código en C. El código de esta herramienta, una vez compilada, genera un fichero de extensión .exe que se encargará de, mediante la consola de comandos de Windows, generar el fichero VHDL que contiene el código máquina del programa ensamblador que queremos introducir en el microcontrolador. El objetivo es mejorarlo y ampliar así sus posibilidades, ya que originalmente, es un sistema que trabaja a 8 bits, circunstancia que limita en gran medida el número de acciones que se pueden realizar con él, ya que no puede manejar números superiores a 2^8 (256).

Para poder conseguir que el microcontrolador ofrezca unas capacidades mayores, se va a realizar una ampliación del tamaño de los datos con los que se trabaja, concretamente la idea consiste en ampliar el tamaño de los datos usados de 8 bits a 32. Esto permitirá realizar operaciones de mayor complejidad, permitiendo así obtener un aprendizaje más completo, así como permitir un mayor abanico de opciones a la hora de usar la herramienta.

Para completar las funcionalidades del microcontrolador, es posible añadir periféricos que se conectan al sistema y complementando así las nuevas funciones y posibilidades que este ofrece.

1.2. Propósito

Uno de los objetivos planteados en este trabajo no es ampliar y mejorar las características del microcontrolador PicoBlaze de 8 bits, así como dotarlo de una mayor capacidad, migrando su diseño a una arquitectura de 32 bits, lo que permitirá manejar datos de mayor tamaño. En este documento se recoge el proceso seguido para la migración del microcontrolador, exponiendo su diseño inicial y mostrando los pasos seguidos para ello, así como una serie de pruebas para verificar que los cambios son válidos y el sistema funciona correctamente tras las modificaciones realizadas y las nuevas funcionalidades añadidas.

Además de migrar la arquitectura a 32 bits, en este trabajo se propone añadir un periférico al sistema que incluya una pequeña memoria ECC (Error Correcting Code), que permitirá almacenar y leer datos en una pequeña memoria, la cual contará con un mecanismo uno-corrector (detecta un error y es capaz de corregirlo) y doble-detector (es capaz de detectar hasta dos errores sin corrección). El objetivo es la adaptar este componente para que funcione también con datos de 32 bits, en consonancia con el microcontrolador, además de verificar el correcto funcionamiento del componente tras la adaptación. Asimismo, se añadirán una serie de nuevas funcionalidades que permitirán comprender y visualizar de una manera sencilla la lógica de funcionamiento del algoritmo de corrección de errores implementado (SECDED).

1.3. Fases del trabajo

Para la realización del proyecto, este se divide en 3 fases:

- En primer lugar, se dará una introducción a una serie de conceptos que permitan entender la estructura interna del microcontrolador. También se revisará el repertorio de instrucciones disponibles y como las instrucciones se organizan en diferentes grupos según la operación que ejecuten. Aquí también se explica como el microcontrolador PicoBlaze interactúa con los demás elementos que componen el sistema, la memoria de programa, que contiene las instrucciones a ejecutar y la herramienta ensambladora, que es la que se encarga de trasladar la instrucciones en ensamblador a código máquina. Por último, se analizará el conexionado que hay que realizar con el FPGA, que es la plataforma sobre la que se implementa el sistema. Se hará mención a los programas utilizados en el desarrollo de este sistema, explicando su uso y la implicación en el desarrollo del nuevo microcontrolador.
- Una vez conocido el sistema con el que se va a tratar, se procederá a explicar la migración realizada del sistema de 8 bits a 32 bits, comentando los cambios para hacer esto posible. Se comentará del rediseño que sufre el PicoBlaze a nivel interno y de cómo estos cambios se reflejan en el código VHDL que lo define, analizando cada uno de los bloques de código existentes. También se comentarán los cambios realizados en la herramienta ensambladora, así como los que han de realizarse en la memoria de programa para completar la migración. Tras esto, se realizarán unas pruebas de testeo de las instrucciones, a fin de asegurar que todos los cambios realizados son satisfactorios y permiten el correcto funcionamiento del sistema.

Finalmente, se analizará el diseño del periférico que se asociará al sistema. Se trata de una pequeña memoria con corrección de errores SECDED, que permitirá leer y escribir datos en 16 posiciones de memoria a las que se apunta con una señal de 4 bits. Tras observar su diseño, se procederá a explicar el proceso de la migración de 8 bits a 32 bits, para que esté en consonancia con los cambios realizados al microcontrolador. Tras esto, se expondrá los cambios que hay que realizar en uno de los ficheros VHDL para poder conectar el periférico al microcontrolador. Por último, se realizarán una serie de pruebas que mostrarán el uso y funcionamiento del periférico.

1.4. Visión global del proyecto

Este proyecto tiene la intención de mostrar cómo proceder a la hora de aumentar las capacidades de un microcontrolador. Esto puede servir como guía para futuros proyectos y establecer un método de trabajo sólido que permita realizar estas modificaciones y añadir nuevas funcionalidades para el desarrollo del propio microcontrolador o bien como herramienta académica que permita al usuario comprender y conocer el proceso que hay que seguir para conseguirlo.

Se mostrará cómo se puede modificar el diseño de una arquitectura definida en VHDL, añadiendo mayor capacidad a sus registros y conectando un periférico para mostrar como este puede desarrollar tareas en conjunto con el microcontrolador, al tiempo que ambos se aprovechan de las ventajas de la ampliación de las capacidades del microcontrolador. Se añadirán nuevas instrucciones al set ya existente en el microcontrolador, siendo todas ellas adaptadas a la nueva arquitectura y se mostrará la manera de interactuar con el periférico conectado al microcontrolador.

Asimismo, se implementarán nuevas funcionalidades en el periférico conectado, además de mostrar el proceso de adaptación de las ya existentes a fin de facilitar la comprensión del algoritmo de corrección de errores que se implementan en él.

2. Plataforma de implementación

Se trata de una FPGA, la ZedBoard, como se puede observar en la Figura 1, que está basada en la Xilinx Zynq-7000. Se la puede dividir en dos partes diferenciadas, como es la Programmable Logic (lógica programable) y un Processing System compuesto de un Cortex-A9 dual. De estas dos partes, para la implementación del Picoblaze, se utilizará únicamente la parte de lógica programable.

La ZedBoard cuenta con numerosos puertos y conexiones diferentes. Tres conexiones importantes que se utilizarán son: la alimentación, el conector micro-USB para programar la placa y en último lugar la conexión con la UART. Para este último elemento, no se usará el puerto micro-USB que hay dedicado en la placa, sino que se hará a través de un módulo PMOD USBUART (figura 2), que se conecta al grupo de conectores JA, que se explicará cómo se conecta más adelante.

La preparación de la placa para poder implementar el microcontrolador es la siguiente: se debe conectar un cable micro-USB al puerto PROG (J17) y el otro extremo, al ordenador desde el que se enviará el código máquina que definirá el circuito. Para la transmisión y recepción de datos, se debe conectar el módulo PMOD en el puerto JA. Este módulo consta de un conector micro-USB, el cual se conectará con un cable también al ordenador. El uso de este módulo será de ayuda, ya que tiene también dos pequeños diodos LEDs que indican la transmisión y recepción de datos. A continuación, en la figura 1 se pueden ver que elementos hay que conectar:



Figura 1. Módulo PMOD. Fuente: Mouser

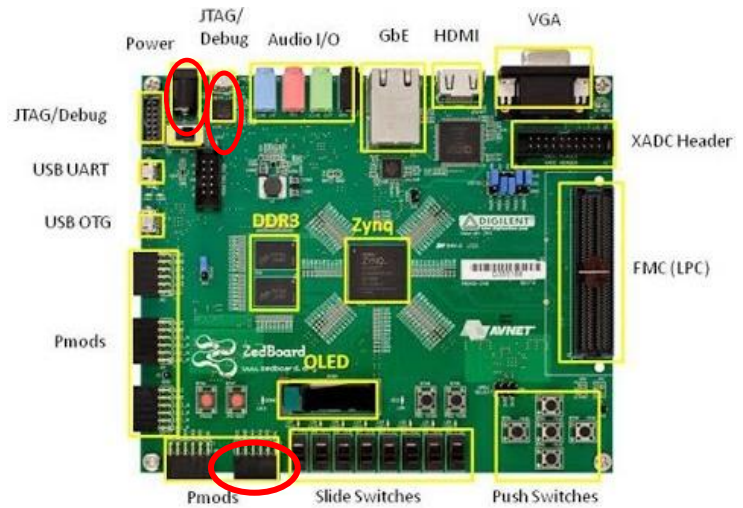


Figura 2. FPGA Zedboard. Fuente: Avnet

3. Descripción del microcontrolador PicoBlaze a 8 bits

3.1. Estructura interna

En la figura 3, se puede observar un esquema que muestra la estructura interna del microcontrolador. Se distinguen 2 bloques principales, la ALU y la UC (Unit Control), en los que se pueden ver los elementos de control permiten al Picoblaze funcionar. Las diferentes partes se comunican por medio de señales de control para poder gestionar los datos pertinentes a cada operación con el componente que corresponda.

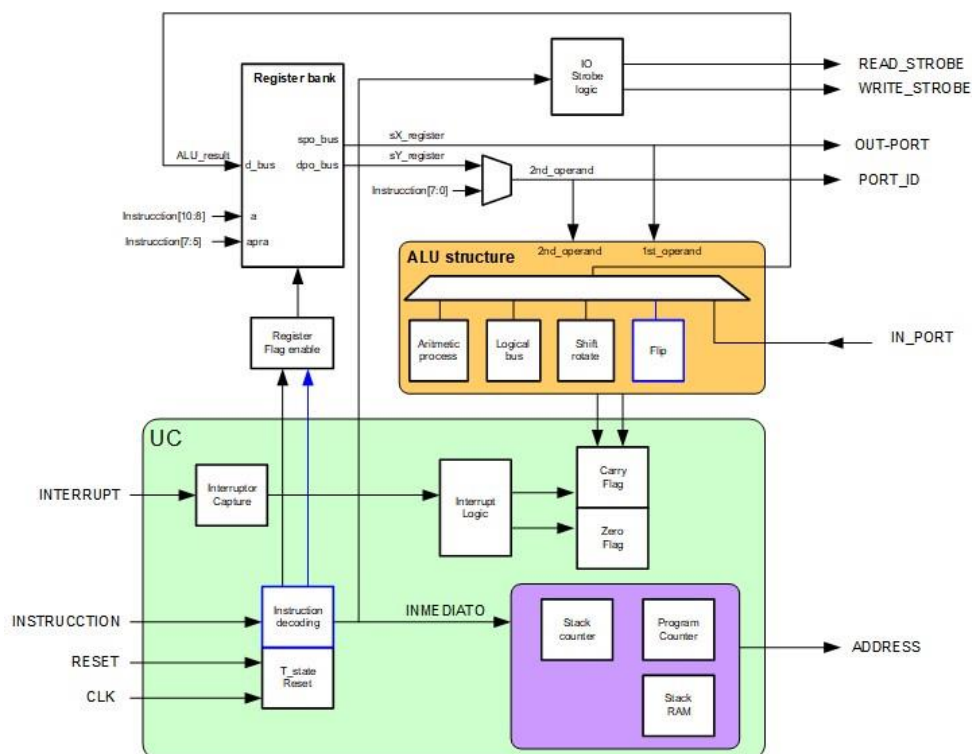


Figura 3. Diagrama del PicoBlaze

Estos dos bloques principales cumplen dos funciones marcadas. La ALU se encarga de procesar los datos obtenidos de los operandos de la instrucción del programa a ejecutar, así como de manejar los datos que le llegan por el puerto inport (canal de comunicación con los periféricos).

La unidad de control se encarga de coordinar todas estas operaciones y que en todo momento se ejecuten por orden y de acuerdo a las circunstancias de la ejecución, mediante el uso de flags y diversas señales que sirven para saber en qué estado se encuentra el programa y el microcontrolador y realizar así correctamente las instrucciones que se le indican.

3.2. Set de instrucciones disponibles

El PicoBlaze, en su versión original de 8 bits, cuenta con un total de 26 instrucciones, con las cuales se pueden realizar operaciones sencillas. Entre ellas se encuentran operaciones lógicas, tales como la OR, XOR o AND y unas pocas operaciones aritméticas, como ADD y SUB o sus variantes ADDCY y SUBCY, que tienen en cuenta el bit de acarreo para el resultado de la operación.

Program control group	Shift and rotate group	Logical group	Arithmetic group	Input/Output group	Interrupt group
JUMP, CALL, RETURN	SRO, SR1, SRX, SRA, RR, SLO, SL1, SLX, SLA, RL	LOAD, AND, OR, XOR	ADD, ADDCY, SUB, SUBCY	INPUT, OUTPUT	RETURNI, ENABLE INTERRUPT, DISABLE INTERRUPT

Tabla 1. Conjunto de instrucciones originales del PicoBlaze

Antes de pasar a explicar las acciones que ejecutan las diferentes instrucciones, se explicará la lógica de funcionamiento para las instrucciones en el microcontrolador:

La instrucción va contenida en un dato que, originalmente es de 16 bits de longitud, llamado palabra de programa. La palabra de programa contiene, entre otros datos, el código de instrucción que se quiere ejecutar. Para este caso, el código de instrucción en la palabra de programa ocupa un total de 5 bits (lo que permite hasta un máximo de 32 códigos de instrucciones). Estos 5 bits son los más significativos y son los que lee el microcontrolador para saber qué acción se quiere realizar, tal y como se ve en la figura 4:

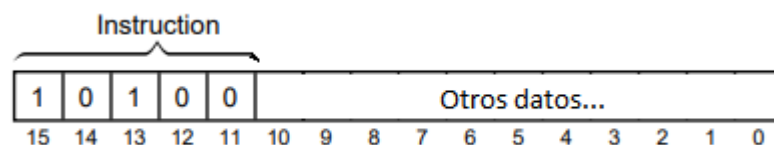


Figura 4. Esquema de una palabra de programa del PicoBlaze a 8 bits. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices

En función del grupo al que pertenezcan las instrucciones, los 11 bits restantes tendrán un significado u otro:

- Program control group: Para este grupo, quedan dos grupos de bits que rellenar. Los que ocupan las posiciones 10, 9 y 8 actúan como flags en función de las circunstancias en las que se produzca la operación, indicando con el bit 10 si se trata de una operación condicional (si el resultado era cero o no) e indica también con los bits 9 y 8 si se tiene en cuenta el bit de acarreo. Los 8 bits restantes se utilizan como dirección de la memoria de programa (ya que esta tiene un tamaño de hasta 256 program words).

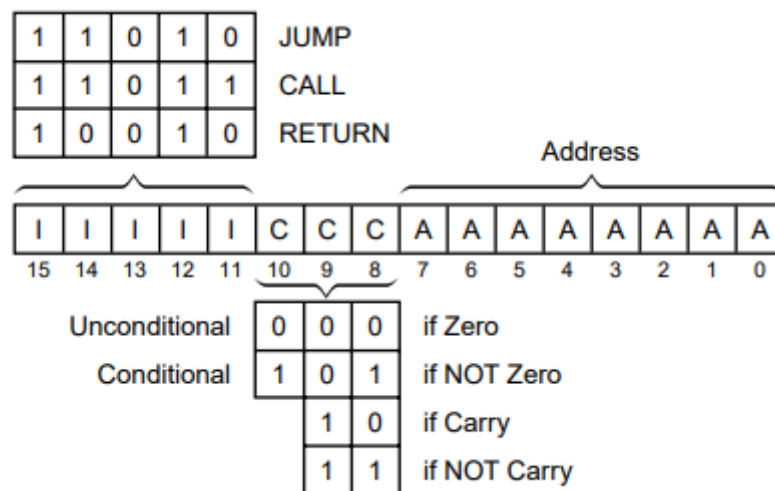


Figura 5. Grupo de instrucciones de control de programa. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices

- Shift and rotate group: Este grupo funciona diferente a los demás. En los 5 bits más significativos se encuentra un código de instrucción fijo, hace referencia a todo el grupo de instrucciones. Tras estos, están los bits 10, 9 y 8, que indican el registro que contiene al dato al cual se le va a aplicar la operación indicada por los bits de instrucción (hay 3 bits para esto porque el número total de registros disponibles es 8). Las posiciones 7, 6, 5 y 4 no se usan y están fijas a 0. La posición 3 se usa para determinar si se avanza a la derecha o a la izquierda y los tres bits menos significativos se usan para determinar la operación en concreto que se desea realizar (dentro de las disponibles para este grupo).

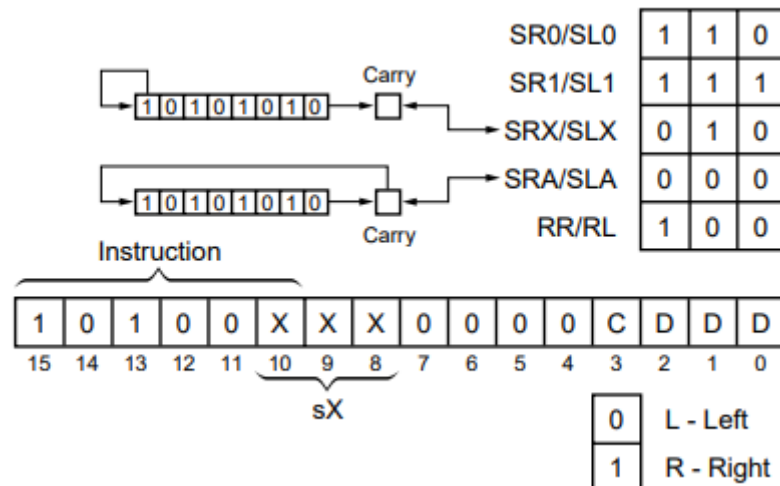


Figura 6. Grupo de instrucciones de desplazamiento y rotación de bits. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices

- Logical group:** Como en todos los grupos, los 5 bits más significativos indican el código de instrucción (que no el grupo) que se va a realizar. En este caso, se tiene en cuenta el bit 14 como modificador de la instrucción, ya que en este grupo de instrucciones se trabaja con dos datos. Este bit indicará la procedencia del segundo dato, si es una constante valdrá 0, si es otro registro ya existente, será 1. Las posiciones 10, 9 y 8 indicarán el primer registro y, en función del valor del bit 14, se toman los 8 restantes como segundo dato (una constante) o solamente los bits 7, 6 y 5 para indicar el registro que se usarán como segundo dato.

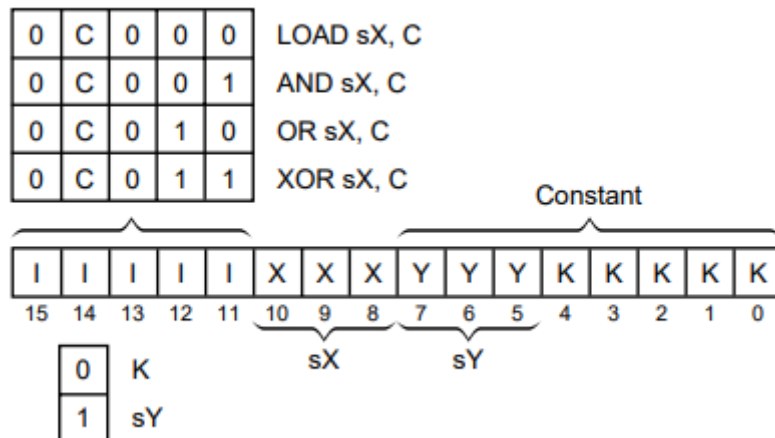


Figura 7. Grupo de instrucciones de operaciones lógicas. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices

- Arithmetic group:** La palabra de programa formada por este grupo es igual a la anterior. El bit 14 indicará si el segundo operando es un valor constante introducido o por el contrario se toma el valor de un registro ya existente. Las posiciones 10, 9 y 8 se reservan para indicar que registro contiene el primer operando. Los bits 7, 6 y 5 indicarán el segundo operando, siempre que sea un registro (lo marca el bit 14).

En caso de que sea una constante, el segundo operando será el valor de los 8 bits menos significativos.

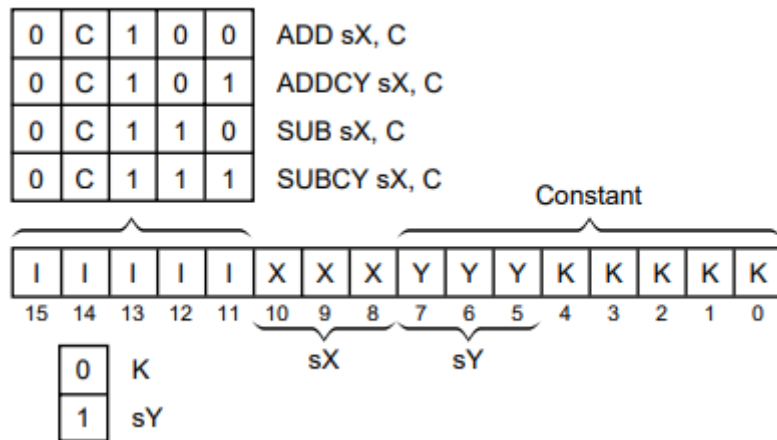


Figura 8. Grupo de instrucciones de operaciones aritméticas. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices

- Input/Output group: Este grupo se compone de los mismos elementos que los anteriores, con la diferencia de que para este caso los 8 bits menos significativos se usarán para identificar un puerto concreto. El bit 14 marcará si el segundo operando es un registro o una constante. En caso de ser un registro se usarán las posiciones 10, 9 y 8 para el primer operando y la 7, 6 y 5 para indicar el segundo operando (siendo un registro). En caso de indicar el bit 14 que se trata de una constante, los 8 bits menos significativos indicarán la dirección del puerto con el que el microcontrolador se comunicará.

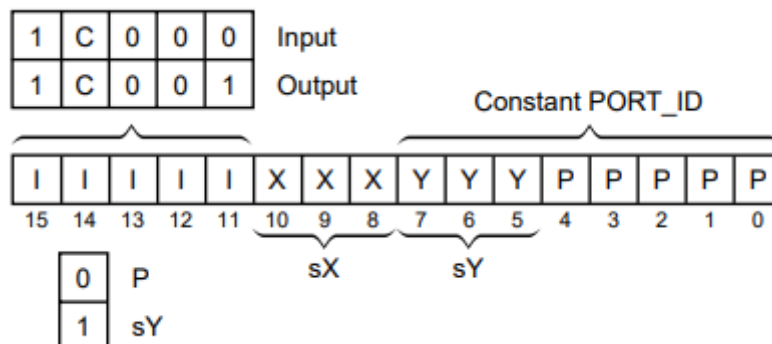


Figura 9. Grupo de instrucciones de entrada y salida de datos. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices

- Interrupt group: Este grupo de instrucciones es más simple que los anteriormente mencionados. Solo importan los 5 bits más significativos, que indicarán la instrucción a realizar y el menos significativo, que tendrá la función de actuar como un toggle para activar o desactivar las interrupciones. El resto de bits no se tienen en cuenta.

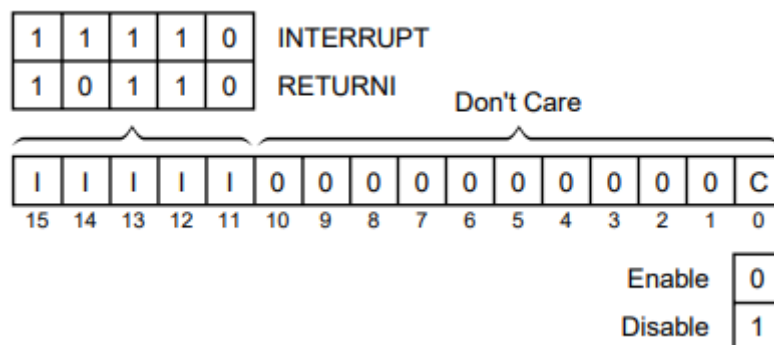


Figura 10. Grupo de instrucciones encargadas de las interrupciones. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices

3.3. Programa del microcontrolador

Todas estas instrucciones vistas anteriormente, son las que se utilizan a la hora de diseñar un programa para este microcontrolador. Para hacerlo, es necesario, en primer lugar, escribir un programa en ensamblador.

Tras escribir el programa, se necesita convertir dicho programa en un fichero VHDL que sea capaz de leer el microcontrolador y este debe contener la memoria de programa al completo, en el formato de instrucciones que se ha visto anteriormente. Para ello se necesita una herramienta que permita hacer la adaptación del código ensamblador a VHDL. Dicha herramienta es el resultado de compilar un programa escrito en C++ que se encarga de traducir este código ensamblador a un fichero VHDL que contiene el código máquina con el que se programa el microcontrolador. El código máquina que resulta contiene las palabras de programa, que se ejecutan de manera secuencial.

Como se puede observar, en la declaración de la entidad, aparecen tres señales, que serán las encargadas de recibir y transmitir datos con el microcontrolador. En concreto, se reciben del microcontrolador dos señales, que son la señal de reloj clk, que establecerá la sincronía entre ambos elementos y la señal address, la cual indica qué línea de la memoria de programa se encuentra el PicoBlaze. Por otra parte, se encuentra la señal dout, que será la encargada de transportar el dato que el microcontrolador solicita.

Si la operación tiene éxito, se generará un fichero con el siguiente formato, el cuál será el que se añadirá al microcontrolador:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity programa_helloworld is
    port( address : in std_logic_vector(7 downto 0);
          clk : in std_logic;
          dout : out std_logic_vector(15 downto 0));
end;

architecture v1 of programa_helloworld is

    constant ROM_WIDTH: INTEGER:= 16;
    constant ROM_LENGTH: INTEGER:= 256;
    subtype rom_word is std_logic_vector(ROM_WIDTH-1 downto 0);
    type rom_table is array (0 to ROM_LENGTH-1) of rom_word;

    constant rom: rom_table := rom_table'(
        "1111000000000000",
        "1101100000000111",
        "0000011100000000",
        "1100000111100000",
        "0010000100000000",
        "1101010000001001",
        "1101100000011100",
        "0010011100000001",
        "1101000000000011",
        .
        .
        .
        "0000000000000000",
        "0000000000000000",
        "1101000000110111");

begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            dout <= rom(conv_integer(address));
        end if;
    end process;
end v1;

```

3.4. Diseño de la herramienta ensambladora

El programa ensamblador parte de un fichero escrito en lenguaje C++. Es al compilar este fichero cuando se genera el archivo ejecutable de Windows (.exe) que permite transformar el programa escrito en ensamblador a código máquina para introducirlo en la memoria de programa del microcontrolador.

Este código C++ contiene numerosos métodos y todos están dirigidos desde un método main, que es el que los gobierna y define el comportamiento de la herramienta, el cual se va a analizar a fin de comprender los pasos ejecutados para poder obtener el fichero con el código máquina.

En primer lugar, el método main de la herramienta recibe dos argumentos, un entero y una cadena de caracteres. Dado que este programa se maneja usando CMD, el entero será el número de argumentos que se han introducido en el comando. Esto simplemente verifica que la sintaxis ha sido correctamente escrita, que ha de ser de la forma: *asm file.asm*. Si esto fuera distinto, devolvería un error.

```
if(argc != 2){  
    printf("\nCommand line syntax:\n\nasm file.asm\n");  
    exit(1);  
}
```

El siguiente paso, está relacionado también con el comando introducido por consola. En este caso se verifica que el archivo que se ha pasado como argumento tenga extensión .asm. Para ello, de la variable char pasada como argumento al método main, se usa el segundo elemento (argv[1]), que es el que contiene el nombre del fichero. Si no contuviese en el nombre del fichero la cadena de texto ".asm", devolvería NULL y detectaría el error, abortando la ejecución.

```
strcpy(filename, argv[1]);  
ptr = strstr(filename, ".asm");  
if (ptr == NULL){  
    printf("\nInvalid file type, use .asm extension\n");  
    exit(1);  
}  
*ptr = '\0';  
strcat(filename, ".log");
```

A partir de aquí, se empieza a trabajar con los ficheros que contienen la información. En primer lugar, se intenta leer el fichero que se indica en la consola de comandos al invocar al programa. Si esta acción presentase algún tipo de problema, la variable ifp, que es la referencia en el código para el fichero que se quiere leer, devolvería un NULL, indicando que no es posible leerlo y abortando la ejecución. Si esta comprobación fuese correcta, se procedería a abrir el fichero que contiene los registros de la ejecución, con la intención de escribir información en él que puede usarse para depurar el programa o identificar y comprender los fallos. De nuevo, si no fuese posible acceder a ella, se obtendría un error y se abortaría la ejecución.

```

ifp = fopen(argv[1], "r");
if (ifp == NULL){
    printf("\nCan not open input file\n");
    exit(1);
}

ofp = fopen(filename, "w");
if (ofp == NULL){
    printf("\nCan not open output file\n");
    exit(1);
}
fprintf(ofp, "Crypto asm logfile\n\n");

```

Tras realizar estas dos comprobaciones, se informa tanto por consola como en el archivo .log (registros de ejecución) que se empieza a leer el fichero que contiene el programa, almacenando cada línea para después procesarla e incrementando un contador, para al final saber exactamente el número de líneas del programa. Durante este proceso, se verifica que no se excede el límite máximo de líneas de programa. Si esto sucediese, se produciría un error y se abortaría la operación.

```

printf("Reading input file...\n");
fprintf(ofp, "Reading input file...\n");
while (fgets(linebuf, 128, ifp) != NULL) {
    if(line_count >= MAX_LINE_COUNT){
        printf("\nInput exceed maximum line number\n");
        error_out();
    }
    parse_linebuf();
    line_count++;
}

```

Una vez se obtenido la información del fichero, se invoca al método `init_program_word`, que se encarga de inicializar el array donde escribirá la memoria de programa. En él, cada posición del array será una palabra de programa completa. A continuación, el programa informa de que se van a someter a pruebas las instrucciones contenidas en el programa asm. Si se encontrase algún error, se informaría al usuario (tanto por CMD como por el fichero log) y se abortaría la ejecución. Si el código analizado supera las pruebas, se procede a llamar al método `write_program_word`, encargado de analizar cada línea y convertirla en una palabra de programa acorde al formato establecido. De nuevo, si se encuentran errores, se abortará la ejecución.

```

init_program_word();
printf("Testing instructions...\n");
fprintf(ofp, "Testing instructions...\n");
test_instructions();
if(error > 0){
    printf("Program aborted...error count %d\n", error);
    fprintf(ofp, "Program aborted...error count %d\n", error);
    error_out();
}
write_program_word();
if(error > 0){

```

```

printf("Program aborted...error count %d\n", error);
fprintf(ofp, "Program aborted...error count %d\n", error);
error_out();
}

```

Tras haber completado la fase anterior, el programa informa que va a comenzar a escribir en los diferentes ficheros que genera el programa. Lo anuncia tanto en pantalla como por el fichero log, y comienza a llamar a los métodos encargados de la escritura de ficheros, en total son 4. De estos, destaca el método `write_vhd`, que se encarga de generar el fichero VHDL que finalmente se incluye en la memoria de programa.

```

printf("Write output files...\n");
fprintf(ofp, "Write output files...\n");
write_fmt();
write_bin();
write_mcs();
write_vhd();

```

Finalmente, tras haber completado todos los procesos mencionados, se procede a la liberación de recursos, llamando al método `free_mem`, que se encarga de vaciar todos los elementos usados en la construcción de la memoria de programa. Tras esto, se informa de que se ha completado el proceso y se cierran los streams que manejan los ficheros necesarios para ensamblar el programa.

```

free_mem();
printf("Program completed...\n");
fprintf(ofp, "Program completed...\n");
fclose(ifp);
fclose(ofp);
return(0);

```

Una vez conocido el comportamiento del programa y que métodos sigue, se van a observar a continuación una serie de métodos clave para que el programa pueda funcionar.

En primer lugar, se van a comentar los métodos encargados de insertar en la palabra de programa los valores correspondientes a cada posición. Como ya se ha comentado anteriormente, la palabra de programa está organizada de una manera determinada, donde según la posición que ocupe un bit, este tiene un significado u otro. Estos métodos son los que se encargan de disponer en el lugar correcto la información.

El primero de ellos es el método `insert_instruction`, que se encarga de introducir los 5 primeros bits que corresponden al código de instrucción, que ocupan las posiciones de la 15 a la 11.

```
void insert_instruction(char *s, int p)
{
    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 << 11);
}
```

A continuación, se puede ver el método `insert_sXX`, que se encarga de introducir una cadena de tres bits que identifica el registro al que hace referencia esa línea de programa.

```
void insert_sXX(int c, int p)
{
    program word[p] = program word[p] | (unsigned) (c << 8);
}
```

Tras este, se encuentra uno muy similar, el método `insert_sYY` que, en caso de que exista un segundo argumento y este sea también un registro, se encarga de introducir la cadena de tres bits que lo identifica en la posición adecuada.

```
void insert_sYY(int c, int p)
{
    program word[p] = program word[p] | (unsigned) (c << 5);
}
```

Si el segundo argumento fuese una constante, se usaría el siguiente método llamado `insert_constant`, que introduce el dato como una cadena de 8 bits en las posiciones correspondientes.

```
void insert_constant(int c, int p)
{
    program word[p] = program word[p] | (unsigned) (c);
}
```

Finalmente, se encuentra el método `insert_flag`, que introduce una cadena de 3 bits con los flags relativos a esa línea de programa, en la posición correspondiente.

```
void insert_flag(int c, int p)
{
    program word[p] = program word[p] | (unsigned) (c << 8);
}
```

Otro de los métodos más importantes es el que permite leer el programa del fichero asm y almacenarlo en la posición correspondiente para después convertir ese código escrito en código máquina, que se insertará en el array que contiene la memoria de programa.

Se trata del método `parse_linebuf`. En primer lugar, el método busca en la línea que esta procesando en ese momento el carácter “;”, que indica el comienzo de un comentario y lo identifica. Tras esto, busca el carácter “:”, que indica el comienzo de una rutina y es identificado como una label (etiqueta) para el programa. A continuación, se leen en orden la instrucción, el primer operando y el segundo operando, tal y como dicta la sintaxis del fichero ASM. Finalmente, para evitar fallos inesperados, existe en el método un trozo de código que se asegura de que no haya más que una instrucción y dos operandos (como mucho). En caso de que esto ocurra, se incrementa el contador de errores.

```
int parse_linebuf(void)
{
    char *ptr;
    char seps[] = " ;, \t\n";
    char *token;

    /* get comment */
    if( (ptr = strchr(linebuf, ';')) != NULL ){
        op[line_count].comment = strdup(ptr);
        *ptr = '\0';
        op[line_count].comment[strlen(op[line_count].comment)-1] =
'\0';
    }

    /* get label */
    if( (ptr = strchr(linebuf, ':')) != NULL ){
        token = strtok( linebuf, seps );
        op[line_count].label = strdup(token);
        strupr(op[line_count].label);
    }

    /* get instruction */
    if (ptr == NULL)
        token = strtok( linebuf, seps );
    else token = strtok( NULL, seps );
    if (token != NULL){
        op[line_count].instruction = strdup(token);
        strupr(op[line_count].instruction);
    } else return (0);

    /* get op1 */
    token = strtok( NULL, seps );
    if (token != NULL){
        op[line_count].op1 = strdup(token);
        strupr(op[line_count].op1);
    } else return (0);

    /* get op2 */
    token = strtok( NULL, seps );
    if (token != NULL){
        op[line_count].op2 = strdup(token);
        strupr(op[line_count].op2);
    } else return (0);
}
```

```

/* make sure nothing left */
token = strtok(NULL, seps);
if (token != NULL) {
    printf("\nToo many operands in line %d\n", line_count+1);
    fprintf(ofp, "\nToo many operands in line %d\n",
line_count+1);
    error++;
}
return (0);
}

```

Otro método muy útil es el llamado `htoi`, el cual se encarga de transformar datos hexadecimales en enteros, para poder manejarlos de una manera cómoda en el programa. El método recibe una cadena de caracteres y se encarga de, elemento a elemento, transformarlo en un valor entero, que se suma a una variable por cada iteración del bucle que recorre la variable.

```

int htoi(char *s)
{
    int i, l, n = 0;
    char *p;

    l = strlen(s);
    for(i = 0; i < l; i++){
        p = s+l-1-i;
        if(isdigit(*p) || (*p >= 'A' && *p <= 'F')){
            if(isdigit(*p)) n += (*p - '0') * (int) pow(16, i);
            else n += (*p - 'A' + 10) * (int) pow(16, i);
        } else return (-1);
    }
    return(n);
}

```

Los dos últimos métodos relevantes dentro de la herramienta ensambladora, son los ya comentados `test_instruction` y `wite_program_word`. El primero se encarga de verificar que todas las instrucciones que se leen del fichero ASM con el programa no tienen errores de sintaxis y se hace un correcto uso de ellas. El segundo, se encarga de validar los datos contenidos en estas líneas e introducirlos en la palabra de programa. Ambos son métodos extensos, ya que tienen código escrito para cada instrucción presente en el microcontrolador.

3.5. Puertos de entrada y salida del sistema

Para poder interactuar con la FPGA, plataforma sobre la cual se implementa el microcontrolador, se debe realizar una serie de conexiones para que el sistema funcione. En concreto, el sistema cuenta con 6 puertos, 4 de ellos son de entrada y 2 de ellos de salida:

- **reset:** Esta señal tiene un tamaño de un bit, es la encargada de restablecer el estado del microcontrolador a su estado inicial. Por defecto tiene valor de '0'. Cuando se acciona un pulsador, esta señal pasa a valer '1' el tiempo que esté pulsado. Basta con que esté pulsado durante un ciclo de reloj (10 ns) para devolver el microcontrolador a su estado original. Está conectado con la placa al pin R16, que toma su valor del pulsador inferior (BTND).
- **clk:** Es la señal encargada de recibir la señal de reloj de la FPGA (100MHz) y transmitirla al resto de componentes. Es una señal de un bit y se conecta al pin Y9 de la FPGA.
- **interrupt:** Esta señal sirve para generar una interrupción en el procesador. Esta influye directamente en el desarrollo de la memoria de programa (siempre que estén habilitadas las interrupciones), llamando a la rutina de atención a la interrupción. Se conecta con el pin T18, que toma su valor, que por defecto es 0, de la señal BTNU, pulsador superior.
- **rx:** Esta señal de entrada será la que reciba los bits (de manera individual, ya que es una señal de un bit) de la comunicación que se realiza por medio de la UART. Esta señal se conecta a la FPGA al pin Y10, que corresponde a uno de los pines de los módulos PMOD, en concreto al JA3.
- **tx:** Señal de salida un bit, se usa como señal de salida para transmitir información por medio de la UART instalada. Se conecta al pin AA11 que corresponde en el FPGA con el pin JA2 de los PMOD.
- **LED:** Señal de salida de un bit. Toma el valor de la señal reset, de manera que según esta sea '1' o '0', activará un LED de la FPGA indicando el estado de reset (Un '1' enciende el LED y un '0' lo apaga o mantiene apagado). Se conecta al pin T22, que corresponde en la placa al LDO.
- **err_en:** Señal de entrada de un bit. Toma el valor del switch SW0, para lo que hay que conectar la señal con el pin F22. Esta señal se utiliza para activar ('1') o desactivar ('0') la corrección de errores en el periférico conectado al microcontrolador.
- **led_err:** Señal de dos bits de salida. Se conecta con los leds de la Zedboard LD7 y LD6, por lo que la conectamos a los pines U14 y U19 respectivamente. Por medio de esta señal podremos mostrar visualmente en la placa los códigos de error devueltos por el periférico según el dato que estemos analizando en ese momento.

3.6. Herramientas adicionales

Para poder completar el desarrollo de este microcontrolador, se han necesitado una serie de herramientas, cuyo uso se detalla a continuación:

- Vivado 2019.1: Esta ha sido la herramienta principal en el desarrollo del microcontrolador. Este programa proporciona un entorno de desarrollo en VHDL que permite editar en él y visualizar el código. En ella también se definen las conexiones del PicoBlaze con la FPGA y permite establecer una conexión con ella para programarla. Proporciona la herramienta de síntesis y se encarga de implementar el diseño a nivel de hardware, así como de generar el fichero .bit, que contiene el código máquina que se introduce en la FPGA para su programación.
- Dev-C++ v. 5.10: Este es un entorno de desarrollo de C++ que ha permitido visualizar y modificar el código necesario para generar el compilador de código asm que genera el fichero VHDL con la memoria de programa que se agrega al microcontrolador.
- Hyperterminal v. 5.1.26: Esta aplicación crea una comunicación serie con un puerto COM seleccionado desde el ordenador donde lo se esté ejecutando, permitiendo así la comunicación entre el microcontrolador y la máquina en la que se ejecuta.
- Windows CMD: Esta ha sido la interfaz usada para interactuar con el programa generado por la herramienta Dev-C++, a la cual le introducimos los comandos en la sintaxis requerida para compilar la memoria de programa en ensamblador y obtener su fichero VHDL correspondiente.

4. Migración de PicoBlaze a 32 bits

4.1. Rediseño del PicoBlaze

Una posibilidad muy interesante que ofrece el PicoBlaze es la posibilidad de modificar su diseño, permitiendo así añadirle nuevas funcionalidades.

Para poder adaptar el microcontrolador a 32 bits, es necesario un rediseño de la mayoría de señales usadas en los ficheros VHDL que definen el microcontrolador. Para ello, se han revisado todos y cada uno de los ficheros que componen el diseño y se han adaptado para el funcionamiento al nuevo tamaño de paquete.

La principal novedad que trae esto consigo es que el tamaño de los ficheros ha crecido, habiendo pasado de 8 a 32 bits, con lo cual por cada registro se tiene 4 veces más capacidad de almacenar información, pero no es el único cambio relevante. El tamaño de la palabra de programa varía también. Se pasa de un tamaño de 16 bits a uno de 43 bits. Este incremento se debe principalmente a que el espacio reservado para las constantes ha crecido en gran medida (este ha de ser igual que el tamaño de los registros), pero también lo han hecho el número de registros disponibles (incremento de 8 registros a 32) y se ha aumentado el número máximo de instrucciones posibles de 32 a 64. El número de palabras de programa que se pueden introducir en la memoria de programa ha crecido también (hasta 2048 palabras de programa – 11 bits), pero dado que es un tamaño menor que el de una constante, no añade bits extra a la palabra de programa.

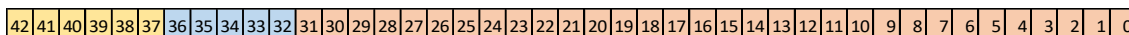


Figura 11. Formato de la palabra de programa del PicoBlaze a 32 bits

Este es el nuevo aspecto de la palabra de programa. Se mantiene la misma estructura, simplemente se ha ampliado el espacio disponible. Los bits sombreados en amarillo, hacen referencia al código de instrucción, que ahora tiene un tamaño de 6 bits.

Los sombreados en azul serían los encargados de indicar de que registro se debe cargar el primer operando (excepto para las instrucciones del Program Control Group, en ellas se ignoran los bits 36 y 35, funcionando así de la misma manera que a 8 bits.)

Finalmente, se encuentra el grupo de 32 bits, sobre el cual se pueden cargar datos de 32 bits de tamaño. Las direcciones de la memoria de programa usarán solo del bit 10 hasta el 0 (debido a su menor tamaño). En otro caso, si se va a seleccionar un segundo registro, su referencia se indicará en los bits 31 a 27.

Respecto a la herramienta compiladora, también se han de hacer unas cuantas modificaciones, ya que esta es la encargada del formarla cuando la herramienta procesa el programa en asm para obtener el fichero VHDL con la memoria de programa. Para ello hay que tener en cuenta los cambios realizados en la palabra de programa e implementarlos en el código C++, para después compilarlo y obtener la herramienta actualizada.

4.2. Modificación del código VHDL

4.2.1. Modificación del fichero toplevel

A continuación, se van a analizar aquellos fragmentos de código que han tenido que ser modificados para realizar la adaptación:

En primer lugar, se encuentra la declaración del picoblaze en el toplevel. No ha sido necesario añadir señales, solo modificar aquellas que no fueran señales de un bit. La señal address se ha adaptado a 8 a 11 bits para la nueva longitud de la memoria de programa, la señal instruction ha sido ampliada ahora de 16 a 43 bits y el resto han sido ampliadas de 8 a 32 bits.

```
component picoblaze
  Port (
    address : out std_logic_vector(10 downto 0);
    instruction : in std_logic_vector(42 downto 0);
    port_id : out std_logic_vector(31 downto 0);
    write_strobe : out std_logic;
    out_port : out std_logic_vector(31 downto 0);
    read_strobe : out std_logic;
    in_port : in std_logic_vector(31 downto 0);
    interrupt : in std_logic;
    reset : in std_logic;
    clk : in std_logic);
end component;
```

La declaración de la memoria de programa también cambia, se pasa de 8 a 11 bits para address y de 16 a 43 para dout, que es la señal que devuelve la palabra de programa contenida en la memoria.

```
component programa_helloworld_100mhz
  Port (
    address : in std_logic_vector(10 downto 0);
    dout : out std_logic_vector(42 downto 0);
    clk : in std_logic);
end component;
```

Aquí se encuentra el proceso de transmisión de datos por medio del pin conectado a la UART. En este caso, solo se ha modificado el valor que debe coincidir con portid, ya que la señal ha crecido de 8 a 32 bits (que equivalen a 8 caracteres hexadecimales).

```
txbuff:process(reset, clk)
begin
  if (reset='1') then
    tx <= '1';
  elsif rising_edge(clk) then
    if (writestrobe = '1' and portid=x"00001111") then
      tx <= outport(0);
    end if;
  end if;
end process;
```

El proceso de recepción de datos, al igual que el de transmisión, cambia el valor del portid. Además, en este caso el valor que se asigna a la señal rxbuff_out cambia, ya que ahora los datos que usa el sistema son de 32 bits, pero dado que la comunicación que se realiza por la UART es de tipo serie, se debe colocar el bit recibido en la posición 7, ya que los datos recibidos son de 8 bits.

```
rxbuff:process(reset, clk)
begin
    if (reset='1') then
        rxbuff_out <= (others=>'1');
    elsif rising_edge(clk) then
        if (readstrobe = '1' and portid=x"00001111") then
            rxbuff_out <= "0000000000000000000000000000" & rx &
"00000000";
        end if;
    end if;
end process;
```

En último lugar, se ha añadido un multiplexor para manejar los datos provenientes de los periféricos, dirigiendo los datos al inport según la dirección del puerto por el que se recibe. Por el momento, solo se ha conectado una salida, la que recibe los datos del puerto serie, pero será más útil en adelante.

```
mux:
    inport <= rxbuff_out when (readstrobe = '1' and portid <=
x"00000040")
```


4.2.2. Modificación del bloque picoblaze

Ahora, se van a revisar los cambios realizados en el PicoBlaze, que es un bloque que contiene todos los módulos del microcontrolador

En primer lugar, se pueden ver los cambios realizados en la declaración de la entidad. Son las mismas modificaciones que las realizadas en la declaración del componente picoblaze en el toplevel.

```
entity picoblaze is
  Port (
    address : out std_logic_vector(10 downto 0);
    instruction : in std_logic_vector(42 downto 0);
    port_id : out std_logic_vector(31 downto 0);
    write_strobe : out std_logic;
    out_port : out std_logic_vector(31 downto 0);
    read_strobe : out std_logic;
    in_port : in std_logic_vector(31 downto 0);
    interrupt : in std_logic;
    reset : in std_logic;
    clk : in std_logic;
  )
end picoblaze;
```

Aquí se encuentra la declaración de tres constantes que serán utilizadas a lo largo del código y que han sido modificadas para el nuevo diseño. La primera define el número de bits que se utilizan para las direcciones de los registros, que pasan de ser 3 a 5 bits. En segundo lugar, se encuentra la declaración de la constante que define el tamaño de la pila, que pasa de 2 a 4 bits. En tercer lugar, se declara el contador de la memoria de programa. Como se ha mencionado anteriormente, la memoria de programa creció de 256 líneas a 2048, lo que supone un incremento de 8 a 11 bits.

```
constant register_bank_address : natural := 5;
constant stack_counter_address : natural := 4;
constant program_counter_address : natural := 11;
```

En este cuadro se pueden ver todos los códigos de instrucción que reconoce el microcontrolador, separados por grupos. La modificación recibida por toda esta parte del código, es contar ahora con un bit más, pasando de 5 a 6 bits. Dado que todos los anteriores no contaban con un sexto bit, se ha añadido un '0' a todas las constantes en su posición más significativa.

```
-- program control group
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_v_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_v_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_v_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_v_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_v_to_x_id : std_logic_vector(5 downto 0) := "001100";
constant addcv_k_to_x_id : std_logic_vector(5 downto 0) := "000101";
constant addcv_v_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_v_to_x_id : std_logic_vector(5 downto 0) := "001110";
constant subcv_k_to_x_id : std_logic_vector(5 downto 0) := "000111";
constant subcv_v_to_x_id : std_logic_vector(5 downto 0) := "001111";
--
-- shift and rotate
constant shift_rotate_id : std_logic_vector(5 downto 0) := "010100";
--
-- added new instruction
-- flip
constant flip_id : std_logic_vector(5 downto 0) := "011111";
--shift rs232
constant shiftrs232_id : std_logic_vector(5 downto 0) := "010011";
--
-- input/output group
constant input_p_to_x_id : std_logic_vector(5 downto 0) := "010000";
constant input_v_to_x_id : std_logic_vector(5 downto 0) := "011000";
constant output_p_to_x_id : std_logic_vector(5 downto 0) :=
"010001";
constant output_v_to_x_id : std_logic_vector(5 downto 0) :=
"011001";
--
-- interrupt group
--
constant interrupt_id : std_logic_vector(5 downto 0) := "011110";
constant return_id : std_logic_vector(5 downto 0) := "010110";
```

Este componente del picoblaze, ha recibido algunos cambios en su declaración. Como se puede observar, las señales first_operand, second_operand e Y, han aumentado su tamaño de 8 a 32 bits para poder manejar los nuevos datos.

```
component arithmetic_process
    Port (first_operand : in std_logic_vector(31 downto 0);
          second_operand : in std_logic_vector(31 downto 0);
          carry_in : in std_logic;
          code1 : in std_logic;
          code0 : in std_logic;
          Y : out std_logic_vector(31 downto 0);
          carry_out : out std_logic;
          clk : in std_logic);
end component;
```

Este componente es el encargado de rotar los bits de los registros del microcontrolador para las distintas operaciones internas que este puede realizar. Mediante las señales code1 y code0 controla la operación que ha de realizar en cada momento. El componente recibe el dato por la señal de entrada operand y lo devuelve por la señal Y una vez realizadas las operaciones pertinentes. Estas son las únicas señales que han tenido que ser modificadas, pasando ambas de un tamaño de 8 a 32 bits.

```
component shift_rotate
    Port (operand : in std_logic_vector(31 downto 0);
          carry_in : in std_logic;
          inject_bit : in std_logic;
          shift_right : in std_logic;
          code1 : in std_logic;
          code0 : in std_logic;
          Y : out std_logic_vector(31 downto 0);
          carry_out : out std_logic;
          clk : in std_logic);
end component;
```

Esta funcionalidad del picoblaze habilita una instrucción que permite intercambiar las posiciones de todos los bits de la siguiente manera: mueve el bit menos significativo a la posición del bit más significativo, con el segundo bit, de nuevo intercambia el segundo bit más bajo con el segundo más alto. Esto se ejecuta un total de 16 veces, lo que permite voltear el registro entero con una sola instrucción. Tanto la señal operand que lleva el dato original como la señal Y que lleva el de salida, han sido ampliadas de 8 a 32 bits.

```
component flip
    Port (operand : in std_logic_vector(31 downto 0);
          Y : out std_logic_vector(31 downto 0);
          clk : in std_logic);
end component;
```

Este componente del picoblaze no está en el diseño original. Ha sido añadido como complemento a la modificación de 8 a 32 bits, motivado por el hecho de que, a pesar de que ahora los registros tienen un tamaño de 32 bits, las comunicaciones y los datos que se reciben por el puerto serie son de 8 bits. Esto proporciona al usuario una instrucción más a la hora de crear la memoria de programa. Su función es la de tomar los 8 bits más significativos y colocarlos en las 8 posiciones menos significativas, desplazando los otros 24 datos hacia la izquierda. Así, tras recibir un dato en un registro, con una sola instrucción este se desplaza, dejando libre las 8 posiciones más bajas para introducir otro dato. Gracias a esto, es posible aprovechar un registro para almacenar 4 datos recibidos por el puerto serie de una manera cómoda.

```
component shift_rs232 is
    Port (operand : in std_logic_vector(31 downto 0);
          Y : out std_logic_vector(31 downto 0);
          clk : in std_logic);
end component;
```

Este componente es el encargado de realizar las operaciones lógicas en el bus de datos del microcontrolador. Se controla mediante las señales code0 y code1, lo que da un total de 4 operaciones posibles: carga un dato en el second_operand, o bien realiza una operación and entre el first_operand y el second_operand, un or o bien una xor, siempre entre estas dos señales. Las únicas modificaciones necesarias aquí han sido las de las señales de entrada ya mencionadas (first_operand y second_operand) y la señal de salida Y, que devuelve el dato final.

```
component logical_bus_processing
    Port (first_operand : in std_logic_vector(31 downto 0);
          second_operand : in std_logic_vector(31 downto 0);
          code1 : in std_logic;
          code0 : in std_logic;
          Y : out std_logic_vector(31 downto 0);
          clk : in std_logic);
end component;
```

Este elemento del picoblaze se encarga de determinar cuándo un dato recibido por este componente es cero. Se encarga de comprobarlo y en caso de ser cierto, activa una señal de salida de un bit que actúa como flag, indicando que el dato que se le ha pasado es nulo.

```
component zero_flag_logic
    Port (data : in std_logic_vector(31 downto 0);
          returni : in std_logic;
          shadow_zero : in std_logic;
          reset : in std_logic;
          flag_enable : in std_logic;
          zero_flag : out std_logic;
          clk : in std_logic);
end component;
```

Esta funcionalidad del microcontrolador es la encargada de permitir el control del contador de programa, se encarga de, según los diferentes flags (aquellas señales que empiezan por i) de entrada que tiene, determinar a qué posición debe ir el contador de programa o a qué posición debe volver. Las señales modificadas aquí están relacionadas con la dirección de la memoria de programa, por lo que estas han sido ampliadas de 8 a 11 bits, ya que como se verificará más adelante, se ha aumentado el número de direcciones de memoria de 256 a 2048.

```

component program_counter

  Port (i_jump      : in std_logic;
        i_call      : in std_logic;
        i_return     : in std_logic;
        i_returni    : in std_logic;
        conditional  : in std_logic;
        low_instruction : in std_logic_vector(10 downto 0);
        stack_value  : in std_logic_vector(10 downto 0);
        flag_condition_met : in std_logic;
        T_state      : in std_logic;
        reset        : in std_logic;
        interrupt     : in std_logic;
        program_count : out std_logic_vector(10 downto 0);
        clk          : in std_logic);

end component;

```

Aquí se encuentra la definición de una memoria RAM de dos puertos, que tiene un total de 2^M posiciones. En el mapeo de puertos que tiene lugar en la descripción del picoblaze, se indica que se asocia a la constante `register_bank_address`, cuyo valor es de 5. Así, se tiene que las señales `a` y `dpra` tendrán un tamaño de (4 downto 0). Teniendo en cuenta que la señal `a` obtiene el valor del registro `sX` que se estén manejando y `dpra` el registro `sY`, se puede ver que el diseño se ajusta en todo momento a esta implementación a 32 bits del microcontrolador. El resto de señales están encargadas de manejar datos propiamente dichos, por lo que han de ser ampliadas de 8 a 32 bits.

```

component register_bank

  generic(M: natural);

  Port (we      : in std_logic;
        d_bus   : in std_logic_vector(31 downto 0);
        wclk    : in std_logic;
        a       : in std_logic_vector(M-1 downto 0);
        dpra    : in std_logic_vector(M-1 downto 0);
        spo_bus : out std_logic_vector(31 downto 0);
        dpo_bus : out std_logic_vector(31 downto 0));

end component;

```

4.2.3. Resultados de implementación del PicoBlaze a 32 bits

Tras realizar los cambios en el diseño del microcontrolador, existen una serie de parámetros que pueden indicar si el diseño es correcto y si este puede funcionar. Esto no analiza las características del código introducido, lo que hace es evaluar el diseño y si este es correcto, arrojando unos datos que permitirán saber si es factible que el diseño sea válido.

En primer lugar, se podrá observar el timing report. Este informe permite saber si el diseño funciona correctamente observando una serie de valores. Se divide en tres partes: Setup, Hold y Pulse Width.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0,147 ns	Worst Hold Slack (WHS): 0,178 ns	Worst Pulse Width Slack (WPWS): 3,750 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 1374	Total Number of Endpoints: 1374	Total Number of Endpoints: 959

Figura 12. Timing report

En la parte de setup se puede observar que el valor Worst Negative Slack (WNS), tiene un valor de 0,147 ns. Que esta medida sea positiva y cercana a 0 ns es positivo. Se encuentra también que la medida Total Negative Slack. Como se puede observar, su valor es 0 ns, lo que es un hecho positivo ya que valores negativos indicarían un mal diseño. Finalmente, en esta parte, se puede ver que el dato Number of Failing Endpoints es 0, indicando 0 errores en esta parte.

Si se analiza ahora de la parte de hold, se encuentran resultados muy parecidos. En la primera medida, Worst Hold Slack, se obtiene un valor de 0,178 ns. Como se puede observar es ligeramente superior al WNS, pero aun así es un valor bajo y sigue siendo cercano a 0 ns, de nuevo es un dato positivo. Similarmente a lo que sucede en el reporte de setup, la segunda medida, Total Hold Slack (THS), es 0. También el dato Number of Failing Endpoints es el mismo, 0, lo que supone que en este apartado se tienen buenas noticias respecto a la calidad del diseño del microcontrolador.

Por último, se encuentra el apartado de pulse width, en el que se ve que el primer dato es Worst Pulse Width Slack (WPNS) con un valor de 3,750 ns. Como se ha mencionado anteriormente, la frecuencia de FPGA es de 100MHz, lo que supone que cada ciclo de reloj dura 10 ns, por lo que un pulso (positivo o negativo) de reloj debe durar 5 ns. Esto indica que, en el peor de todos los casos, la duración es de 3,75 ns, un valor para nada igual pero bastante cercano. Si se observan el resto de parámetros se puede ver que son todos 0, lo que indica que no se encuentran fallos de ningún tipo, y por tanto esta medida de WPNS es aceptable.

En definitiva, el timing report resulta positivo, lo que hace pensar que es perfectamente factible que el diseño funcione, cosa que se verificará posteriormente.

Tras el timing report, se va a proceder a valorar la utilización de recursos de la FPGA que se han precisado para implementar el microcontrolador.

Resource	Utilization	Available	Utilization %
LUT	511	53200	0.96
LUTRAM	71	17400	0.41
FF	393	106400	0.37
BRAM	2.50	140	1.79
IO	9	200	4.50
BUFG	1	32	3.13

Figura 13. Utilización de recursos de la FPGA (formato tabla)

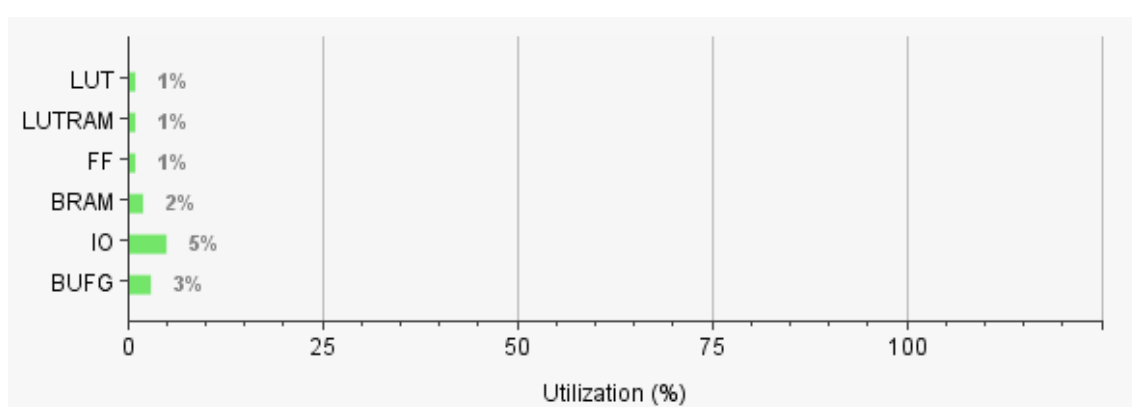


Figura 14. Utilización de recursos de la FPGA (formato gráfica)

Aquí se pueden observar los datos de utilización desde dos perspectivas diferentes, en una tabla y en una gráfica. Como se puede observar, los porcentajes de utilización son muy bajos. A pesar de haber ampliado las capacidades (lo que supone un aumento en el uso de recursos) y haber añadido un componente extra, los porcentajes son muy bajos, lo que hace ver que las capacidades de la FPGA sobre la que se implementa el sistema son mucho mayores que la exigencia de este microcontrolador.

En definitiva, estos reportes arrojan resultados positivos sobre el diseño planteado para el microcontrolador, tanto el timing report como el escaso uso de recursos de la FPGA que supone implementar el PicoBlaze en ella, lo que hace pensar que, siempre que no se cometan fallos al programar, todo debería funcionar correctamente.

4.3. Adaptación de la herramienta ensambladora

El compilador es un elemento fundamental para poder usar y programar el picoblaze. Es el que se encarga de realizar la conversión de un programa asm (ensamblador) a un fichero VHDL que contiene la memoria de programa al completo, en un formato que el microcontrolador pueda reconocer.

A continuación, se van a mostrar los cambios que han sido necesario realizar para que el compilador pueda proporcionar la memoria de programa de una manera adecuada para el nuevo diseño de 32 bits del picoblaze.

En primer lugar, se puede apreciar que los códigos de instrucción han sido modificados, que son los mismos que los que se pusieron en la modificación del picoblaze. Estos han de coincidir ya que serán los que el compilador escriba en la memoria de programa y los que el picoblaze tendrá que leer.

```
/* 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_v_to_x_id = "001000";
char *and_k_to_x_id = "000001";
char *and_v_to_x_id = "001001";
char *or_k_to_x_id = "000010";
char *or_v_to_x_id = "001010";
char *xor_k_to_x_id = "000011";
char *xor_v_to_x_id = "001011";

/* arithmetic group */
char *add_k_to_x_id = "000100";
char *add_v_to_x_id = "001100";
char *addcv_k_to_x_id = "000101";
char *addcv_v_to_x_id = "001101";
char *sub_k_to_x_id = "000110";
char *sub_v_to_x_id = "001110";
char *subcv_k_to_x_id = "000111";
char *subcv_v_to_x_id = "001111";

/* shift and rotate */
char *shift_rotate_id = "010100";
char *flip_id = "011111";

/* added new instruction */
char *shiftrs232_id = "010011";

/* input/output group */
char *input_p_to_x_id = "010000";
char *input_v_to_x_id = "011000";
char *output_p_to_x_id = "010001";
char *output_v_to_x_id = "011001";
```



```

/* interrupt group */
char *interrupt_id = "011110";
char *return_id = "010110";

```

A continuación, se encuentra la declaración de unos parámetros que han tenido que ser modificados:

- El primero define el número máximo de líneas permitidas en el fichero ensamblador, que aumenta de 1000 a 4096.
- El segundo, indica el número máximo de palabras de programa posibles que se pueden incluir en la memoria de programa, que como se ha comentado anteriormente en otros puntos del documento, es de 2048 (2^{11} bits).
- El tercer parámetro es el tamaño máximo de cada variable. Si bien esta variable no existía antes, ha sido necesario incluirla debido a una complicación con los tipos de variables usados en C++. Su valor asociado corresponde a 2^{32} , que es el número más grande que se puede guardar en un registro.
- El cuarto parámetro, hace referencia al número de registros que pueden existir, que ha pasado de 8 a 32.

```

#define MAX_LINE_COUNT 4096
#define PROGRAM_COUNT 2048 /* total program word */
#define VAR_SIZE 4294967296
#define REG_COUNT 32 /* max 8 namereg can be declared */

```

Esta es la declaración del array que contiene la memoria de programa que genera el compilador. Ha sido necesario modificar el tipo de variable de unsigned (16 bits) a long long int (64 bits), ya que cada palabra de la memoria de programa tiene un total de 43 bits.

```

long long int program_word[PROGRAM_COUNT]; /* program word array */

```

Aquí se muestra ahora el conjunto total de instrucciones que reconoce el compilador. Se muestran las cadenas de texto que están asociadas al uso de cada instrucción. En este caso, se ha añadido la instrucción SHIFTRS232 que es la mencionada anteriormente en la modificación del bloque picoblaze, la cual ha sido creada desde cero y pensada para aprovechar mejor el potencial que otorga el nuevo diseño a 32 bits.

```
char *instruction_set[] = {
    "JUMP" /* 0 */
    "CALL" /* 1 */
    "RETURN" /* 2 */
    "LOAD" /* 3 */
    "AND" /* 4 */
    "OR" /* 5 */
    "XOR" /* 6 */
    "ADD" /* 7 */
    "ADDCY" /* 8 */
    "SUB" /* 9 */
    "SUBCY" /* 10 */
    "SR0" /* 11 */
    "SR1" /* 12 */
    "SRX" /* 13 */
    "SRA" /* 14 */
    "RR" /* 15 */
    "SLO" /* 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 */
    "SHIFTRS232" /* 30 */ /* added new instruction */
};
```

Este método es el que se encarga de convertir valores hexadecimales en valores decimales, teniendo estos valores un tamaño máximo de 64 bits. Su funcionamiento apenas se ha visto alterado respecto al original, lo único que ha sido necesario modificar es el tipo de las variables usadas, pasando de variables tipo int a variables long long int. Esto es necesario para el buen funcionamiento del programa en C++ y evitar errores de compilación.

```
long long int htoi(char *s)
{
    long long int i, l, n = 0;
    char *p;

    l = strlen(s);
    for(i = 0; i < l; i++){
        p = s+l-1-i;
        if(isdigit(*p) || (*p >= 'A' && *p <= 'F')){
            if(isdigit(*p)) n += (*p - '0') * (long long int) pow(16, i);
            else n += (*p - 'A' + 10) * (long long int) pow(16, i);
        } else return (-1);
    }
    return(n);
}
```

Este método es uno de los que más modificaciones ha tenido. Es el encargado de devolver en formato int el número del registro que el sistema esté usando. Para ello, en primer lugar, comprueba el formato con el que le pasan los datos. Si cumple, entonces intenta discernir si el registro que le han pasado tenga dos elementos (registro entre S0 y S9) o tres elementos (registro de S10 a S31). Llegados a este punto, el método distingue si se trata de decenas, veintena o treintenas. Para cada caso, se realiza una operación que acaba devolviendo el dígito en decimal del registro que se hace referencia. Esto lo consigue restando el valor hexadecimal del carácter ASCII que se sitúa en la unidad al valor hexadecimal del 0 ASCII, a lo que se le suma 10, 20 o 30 en función de la clasificación realizada anteriormente.

```
int register_number(char *s)
{
    if(*s != 'S') return(-1);
    if(strlen(s) != 2 && strlen(s) != 3) return(-1); /*Here we
check the length of the reg introduced, from 2 digits up to 3*/
    if(strlen(s) == 2){
        if((*s+1) >= '0' && (*s+1) <= '9')) /*If there are
two digits, it is form S0 to S9*/
            return (*s+1 - '0'); /*s+1 makes reference to
the second char of s*/
    } else if(strlen(s) == 3) {
        if(*s+1 == '1'){ /* If first numeric char is 1 */
            if((*s+2) >= '0' && (*s+2) <= '9')) /* And the
last char is between 0 and 9 */
                return(10 + *s+2 - '0'); /* It returns 10 +
(0-9) usign decimal value of ascii characters*/
        } else return(-1);
    }
}
```

```

        if(*s+1 == '2'){
            if((*s+2) >= '0') && (*s+2) <= '9'){
                return(20 + *s+2 - '0');
            } else return(-1);
        }
        if(*s+1 == '3'){
            if((*s+2) >= '0') && (*s+2) <= '1'){
                return(30 + *s+2 - '0');
            } else return(-1);
        }
    } else return(-1);
}

```

Con este se rellena una parte de la palabra de programa, concretamente los primeros 6 bits que la forman, que hacen referencia al código de instrucción. Para ello, ha sido necesario modificarlo un poco. Se han actualizado los tipos de datos a long long int y se ha ajustado la posición a la que se introducen los valores. Concretamente, se pasa de desplazar el bit menos significativo del código de instrucción de la posición 11 a la 37. Con esto, se completan las 6 primeras posiciones con el código de la instrucción que esté procesando.

```

void insert_instruction(char *s, int p) {
    int i, l;
    long long int n = 0;

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

    program_word[p] = program_word[p] | (n << 37); /*43-6=37*/
}

```

Con esta función el programa se encarga de introducir en la palabra de programa que se esté montando en ese momento el valor binario del registro sX con el que se esté trabajando en ese momento. Para ello, introduce los datos a partir de la posición 32 de la palabra de programa. El tipo de datos usado en este método es long long int.

```

void insert_sXX(long long int c, int p){
    program_word[p] = program_word[p] | (unsigned long long int) (c
<< 32);
}

```

Esta función es muy similar a la anterior. Se encarga de introducir los alores binarios del registro sY que se esté analizando en ese momento. Lo que hace es introducir el código binario que hace referencia a ese registro sY a partir de la posición 27 de la palabra de programa. Los tipos de datos también cambian y pasan a ser de tipo long long int.

```
void insert_sYY(long long int c, int p)
{
    program_word[p] = program_word[p] | (unsigned long long int) (c
<< 27);
}
```

Con este método, se introduce en la palabra de programa la constante con la que se esté trabajando en ese momento. Este método es alternativo con el anterior, el *insert_sYY*, ya que no se podrán usar en una misma palabra de programa. Como con prácticamente todos los métodos modificados, el tipo de datos pasa a ser long long int.

```
void insert_constant(long long int c, int p)
{
    program_word[p] = program_word[p] | (unsigned long long int)
(c);
}
```

Otra posibilidad a la hora de formar una palabra de programa, es el tener que introducir los flags. Para ello, este método se encarga de introducir el valor binario de dichos flags a partir de la posición 32 cuando se invocado. También, es necesario modificar los tipos de datos, que pasan a ser long long int.

```
void insert_flag(long long int c, int p)
{
    program_word[p] = program_word[p] | (unsigned long long int) (c
<< 32);
}
```

Esta función del programa comprueba la sintaxis de las líneas de código del programa que se desea cargar en el microcontrolador. Verifica que las expresiones sean correctas para cada caso leyendo los parámetros introducidos en cada caso y comprobando que sean válidos y estén expresados correctamente. En este caso, no ha sido necesaria una modificación, pero si ha sido necesario añadir un caso para la nueva instrucción. En ella se verifica que solo exista un parámetro al usar la instrucción, de lo contrario, devolvería un error de compilación.

```
void test_instructions(void)
{
case 30: /*SHIFT_RS232*/
    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;
}
```

A continuación, se puede ver uno de los métodos más extensos que componen la herramienta ensambladora, el write_program_word. Este método no ha requerido muchos cambios. El primero de ellos está relacionado con una variable que se define al inicio del fichero, PROGRAM_COUNT. El método analiza sus operandos, decidiendo de que tipo es y llamando a la función correspondiente en cada caso, pudiendo ser una etiqueta, una constante o un valor hexadecimal. Es cuando se introduce un valor hexadecimal cuando se produce una comparación. En la versión de 8 bits, se compara el dato introducido con PROGRAM_COUNT, ya que el tamaño máximo del contador de programa y el tamaño máximo de un registro coincidían (originalmente existían máximo 256 líneas de memoria de programa y 2^8 era el tamaño máximo de un registro).

Tras haber realizado la modificación, ahora se puede observar que esos tamaños no coinciden, ya que mientras la memoria de programa admite un máximo de 2048 líneas (2^{11}), el tamaño máximo de un registro es ahora mucho mayor (4294967296, 2^{32}), por lo que fue necesario incluir la variable VAR_SIZE, mencionada anteriormente, con el valor máximo que puede tener un registro. En cada comparación, se sustituye la variable PROGRAM_COUNT por VAR_SIZE.

También ha sido necesario incluir un nuevo caso en el switch para la nueva instrucción añadida.

```

case 30: /* SHIFT_RS232 */ /* added new instruction */

    insert_instruction(shiftrs232_id, op[i].address);
    if((reg_n = find_namereg(op[i].op1)) != -1)
        insert_sXX(reg_n, op[i].address);
    else if((reg_n = register_number(op[i].op1)) != -1)
        insert_sXX(reg_n, op[i].address);
    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;

```

Este es el último método que ha requerido cambios. Se trata de la función que escribe en el fichero VHDL con la memoria de programa que se cargará en el microcontrolador. Se han actualizado las cadenas de texto que se escriben como código en el fichero a los nuevos tamaños (ROOM_WIDTH y ROM_LENGTH) y se ha actualizado el número de iteraciones de los bucles de 15 a 42. Este método actúa cuando se invoca al programa .exe generado con el código C++, cuya sintaxis es “(nombre del fichero .exe) (nombre del fichero con la extensión .asm)”. Para hacerlo, se debe estar en el mismo directorio en CMD que el fichero ejecutable.

```
void write_vhd(void)
{
    :
    :
    fprintf(ffp,"entity %s is\n",basename);
    fprintf(ffp,"\tport( address : in std_logic_vector(10 downto 0);\n");
    fprintf(ffp,"\t\tclk : in std_logic;\n\t\t\tdout : out\n");
    fprintf(ffp,"std_logic_vector(42 downto 0));\n\t\t\tend;\n\n");
    fprintf(ffp,"architecture v1 of %s is\n\n", basename);

    fprintf(ffp,"\tconstant ROM_WIDTH: INTEGER:= 43;\n");
    fprintf(ffp,"\tconstant ROM_LENGTH: INTEGER:= 2048;\n\n");
    fprintf(ffp,"\tsubtype rom word is std_logic_vector(ROM_WIDTH-1 downto\n");
    fprintf(ffp,"0);\n");
    fprintf(ffp,"\t\ttype rom table is array (0 to ROM_LENGTH-1) of\n");
    fprintf(ffp,"rom word;\n\n");
    fprintf(ffp,"constant rom: rom table := rom_table'(\n");
    for(i = 0; i < PROGRAM_COUNT-1; i++){
        fprintf(ffp, "\t\t");
        for(j = 42; j >= 0; j--) /*Updated from 15 to 42*/
            fprintf(ffp, "%d", (program_word[i]>>j) & 1); //print binary
        fprintf(ffp, "\",\n");
    }
    fprintf(ffp, "\t\t");
    for(j = 42; j >= 0; j--) /*Updated from 15 to 42*/
        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\t\tdout <=\n");
    fprintf(ffp,"rom(conv_integer(address));\n");
    fprintf(ffp,"end if;\nend process;\nend v1;\n");
    fclose(ffp);
}
}
```

4.4. Ejemplo de programa ensamblador

El programa en ensamblador, como se ha mencionado en otros puntos del documento, es lo que finalmente acabará en el microcontrolador como memoria de programa. En este caso, las modificaciones han sido necesarias hacerlas en la rutina encargada de coordinar la tasa a la que se transmiten y reciben datos.

El programa que existía originalmente, estaba pensado para funcionar a una frecuencia de reloj de 50MHz, de manera que debían existir unas determinadas configuraciones para cada una de las posibles tasas de envío y recepción de datos por el puerto serie. Dado que la FPGA sobre la que se implementa el microcontrolador funciona a 100MHz, se deben adaptar estas rutinas con valores que permitan mantener las tasas de comunicación por el puerto serie, pero a una frecuencia de reloj mayor.

En concreto, hay que modificar dos rutinas, llamadas wait_1bit y wait_05bit:

```
;PARA PICOBLAZE A 100MHz  
:clk=100MHz, 115200bps, cont1=01, cont2=D6  
:esta rutina ejecuta  $1 + (1 + 1*(1 + 214*2 + 2)) + 1 = 434$   
instrucciones,  
:aproximandose al numero teorico de  $(8,68\mu s/bit)/(0,02$   
us/instruc) = 434 instr/bit necesarias.  
  
wait_1bit:  LOAD      cont1, 00000001  
espera2:   LOAD      cont2, 000000D6  
espera1:   SUB        cont2, 00000001  
           JUMP       NZ, espera1  
           SUB        cont1, 00000001  
           JUMP       NZ, espera2  
           RETURN
```

Esta rutina es la encargada de generar el tiempo suficiente entre bit y bit para que se ajuste a una tasa determinada. En concreto, el programa está ajustado para funcionar en una comunicación a 115200 baudios sobre un reloj de 100MHz. Para poder ajustar esto, es necesario calcular cuantas instrucciones se precisan que se ejecuten entre bit y bit. En este caso 115200 baudios corresponden a 8,68 μs entre bit y bit. Sabiendo también la duración de un ciclo de reloj, que son 20 ns, se puede calcular el número de instrucciones necesarias para rellenar un espacio de 8,68 μs , que son 434 instrucciones por baudio.

Sabiendo esto, se deben que asignar unos valores a las variables cont1 y cont2 para que el resultado de la fórmula: $1 + (1 + cont1 * (1 + cont2 * 2 + 2)) + 1$ dé un resultado lo más cercano posible a las 434 instrucciones por baudio que se necesitan.

La siguiente rutina es muy similar a la anterior, su funcionamiento se basa en la misma lógica y también pretende introducir un retardo de tiempo controlado, en este caso de medio bit (o baudio) de duración:

Dado que se ha de encontrar la solución a una ecuación con dos incógnitas, se ha optado por escribir un pequeño y sencillo programa en lenguaje Java que pruebe con un gran rango de valores para encontrar los valores exactos, y si no fuera posible, los más cercanos posibles.

```
; PARA PICOBLAZE A 100MHz
      :clk=100MHz, 115200bps, cont1=02, cont2=34
      ; 1 + (1 + 2*(1 + 52*2 + 2)) + 1 = 217; aprox = 217

wait_05bit: LOAD      cont1, 00000002
espera4:   LOAD      cont2, 00000034
espera3:   SUB       cont2, 00000001
           JUMP      NZ, espera3
           SUB       cont1, 00000001
           JUMP      NZ, espera4
           RETURN
```

El programa utilizado es el siguiente:

```
public class App {
    public static void main(String[] args) {

        int num1[] = new int[512];
        int num2[] = new int[512];

        for(int i = 0; i<512; i++){
            num1[i] = i;
            num2[i] = i;
        }

        int numCiclos = 0;

        for(int i = 0; i<num1.length; i++){
            for(int j = 0; j<num2.length; j++){
                numCiclos = (1+(1+num1[i]*(1+num2[j]*2+2))+1);
                if(numCiclos == 217 || numCiclos == 434){
                    System.out.println(numCiclos);
                    System.out.println("Para un cont1: "+num1[i]+" y para
un cont2: "+num2[j]);
                }
            }
        }
    }
}
```

Para lo que devuelve la siguiente salida:

```
434
Para un cont1: 1 y para un cont2: 214
217
Para un cont1: 2 y para un cont2: 52
```

Como se ha podido ver en los anteriores cuadros de texto donde estaban descritas las rutinas del programa en ensamblador, los valores obtenidos por medio de este sencillo programa son los que finalmente se han implementado, ya que dan una tasa exacta de espera de 20 ns, justo el periodo de reloj del microcontrolador.

Para la rutina wait_1bit obtenemos para cont1 el valor 1 y para cont2 el valor 214, que en hexadecimal equivalen a 01 y D6 respectivamente. Para el caso de la rutina wait_05bit, obtenemos para la variable cont1 el valor 2 y para cont2 el valor 52, cuyos equivalentes hexadecimales son 02 y 34.

4.5. Prueba de funcionamiento del microcontrolador a 32 bits

Para verificar que la adaptación del microcontrolador de 8 a 32 bits ha funcionado correctamente, se van a someter a test todas las instrucciones que hay disponibles. Para ello, se usarán rutinas en ensamblador en las que se usarán cada una de las instrucciones o con un grupo de ellas, con unos parámetros concretos, para los cuales se puede predecir el resultado y comparar después este con el resultado de la ejecución. Las instrucciones JUMP, CALL, RETURN y LOAD no tienen un test específico para ellas ya que se usan en todas las demás pruebas, por lo que se puede concluir que, si las demás superan los test, significa que estas también lo hacen correctamente.

La configuración es exactamente la misma para todas las pruebas, ofreciendo así más rigor con el resultado:

datos	NAMEREG	s1, txreg	;buffer de transmision
	NAMEREG	s2, rxreg	;buffer de recepcion
	NAMEREG	s3, contbit	;contador de los 8 bits de
	NAMEREG	s4, cont1	;contador de retardo1
	NAMEREG	s5, cont2	;contador de retardo2
	NAMEREG	s6, aux	
	NAMEREG	s7, cont3	
	NAMEREG	s8, show_reg	
	NAMEREG	s9, aux2	
	NAMEREG	s10, aux3	
	NAMEREG	s11, aux4	
ADDRESS	00	;el programa se carga a partir de la dir 00	

Como apoyo para estas pruebas, se van a usar una serie de rutinas de elaboración propia a fin de facilitar la realización y comprobación de los test:

Esta rutina permitirá representar con unos y ceros ASCII el valor de los registros, cargando el valor a representar en la variable aux y llamando a la rutina show.

```
show:
    LOAD    cont3, 00000020 ;32
show_start:
    LOAD    show_reg, aux
    AND     show_reg, 80000000 ;nos quedamos solo con el bit
mas significativo
    OR      show_reg, 00000000 ;comparamos el valor del bit mas
significativo, si es cero, la or dara cero, si no , es un uno
    JUMP    NZ, uno ;si no es cero, vamos a imprimir un uno
    JUMP    cero
show_aux:
    RL      aux
    SUB     cont3, 00000001
    JUMP    NZ, show_start
    LOAD    txreg, 0000000A
    CALL    transmite
    LOAD    txreg, 0000000D
    CALL    transmite
    RETURN

uno:
    LOAD    txreg, 00000031 ;1 ascii
    call    transmite
    JUMP    show_aux

cero:
    LOAD    txreg, 00000030 ; 0 ascii
    call    transmite
    JUMP    show_aux
```

Esta función servirá de ayuda a la hora de representar los datos en el programa Hyperterminal, espaciando y haciendo más legibles los resultados que arrojen los test.

```
saltoLinea: LOAD    txreg, 0A
              CALL    transmite
              LOAD    txreg, 0D
              CALL    transmite
              RETURN
```

- AND:

Para esta prueba se plantea un caso sencillo con la operación AND. Se carga en un registro el valor hexadecimal cuyos 4 bits menos significativos son 0101 y otro en el que sus 4 bits menos significativos son 0011; el resto de bits son 0. Si la operación funciona correctamente, devolverá treinta y un ceros y un uno en el bit menos significativo. La lógica del test viene descrita con el siguiente código:

```
start:      ;esperamos a recibir un caracter
CALL       recibe      ;control
LOAD       aux3, 00000005
LOAD       aux4, 00000003
AND        aux3, aux4
LOAD       aux, aux3
CALL       show
CALL       saltoLinea
```

Para comprobar el resultado, se carga el test en la memoria de programa y se introduce en el sistema. Aquí se puede ver la respuesta obtenida, que es el resultado esperado:

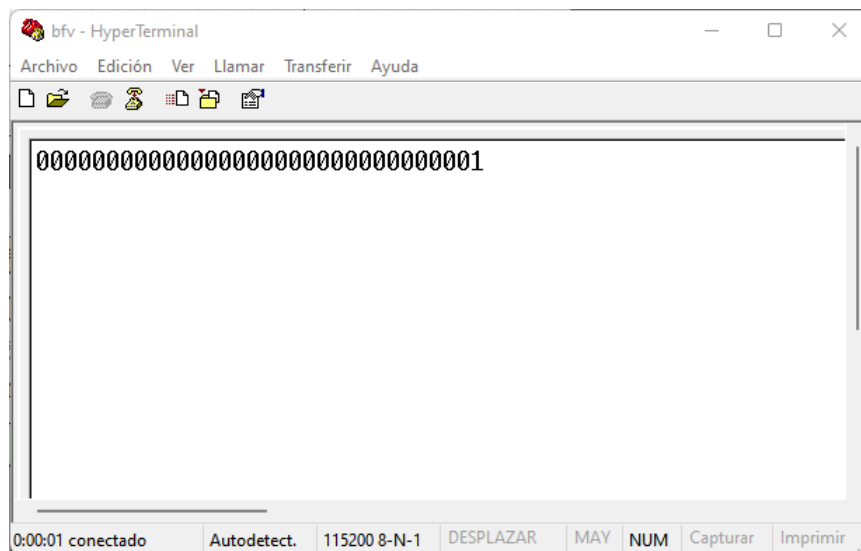


Figura 15. Ventana del programa Hyperterminal con el resultado del test de la operación AND

- OR:

Para comprobar la operación lógica OR, se va a realizar una prueba similar a la anterior, con los mismos parámetros, pero esta vez se espera un resultado distinto, que deben ser tres unos en las tres posiciones menos significativas y el resto han de ser ceros.

```
LOAD       aux3, 00000005
LOAD       aux4, 00000003
OR        aux3, aux4
LOAD       aux, aux3
CALL       show
CALL       saltoLinea
```

El resultado del test es el correcto, como se puede ver el microcontrolador devuelve lo esperado:

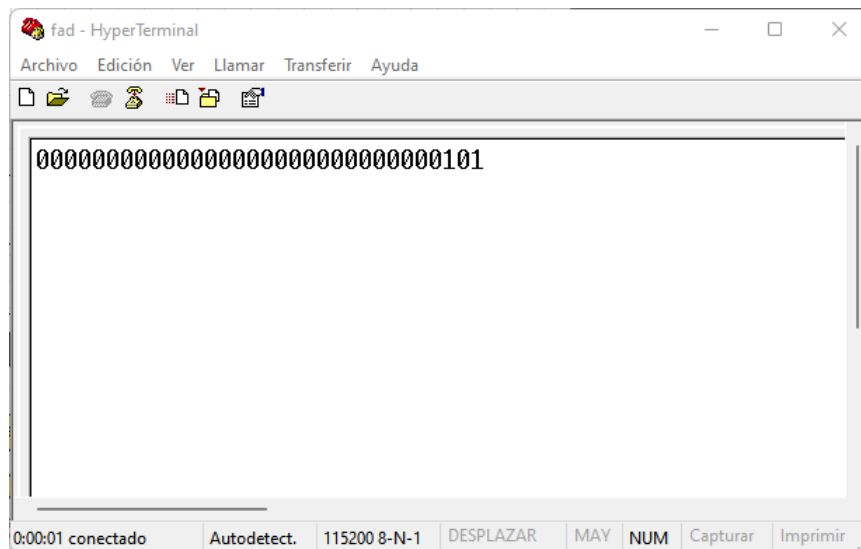


Figura 19. Ventana del programa Hyperterminal con el resultado del test de la operación SUB

- ADDCY:

Esta instrucción realiza una suma, pero introduce una variación. Se trata de una instrucción que suma los dos parámetros que se le indiquen y además suma el bit de acarreo, sea cual sea su valor. Para comprobarlo se va a cargar un registro cuyos bits están todos a 1. A este, le añadirá una unidad, provocando el desborde y estableciendo el bit de acarreo a 1 justo en el momento de la ejecución. Una vez hecho eso, se ejecutará una suma con los parámetros del test de ADD, esperando el mismo resultado sumando un uno (el bit de acarreo). El resultado, por tanto, debe ser 21 (10101 en binario). Si tras esto, se repite la suma (el bit de acarreo vale 0), el resultado debe ser 20 (10100 en binario).

```

LOAD      aux3, FFFFFFFF ;ADDCY -> 21
LOAD      aux2, 0000000A
ADD       aux3, 00000001 ;generamos carry --> 1
ADDCY     aux2, 0000000A
LOAD      aux, aux2
CALL      show
CALL      saltoLinea

LOAD      aux2, 0000000A
ADDCY     aux2, 0000000A
LOAD      aux, aux2
CALL      show
CALL      saltoLinea

```

El test resulta satisfactorio, ya que los resultados coinciden con lo esperado.

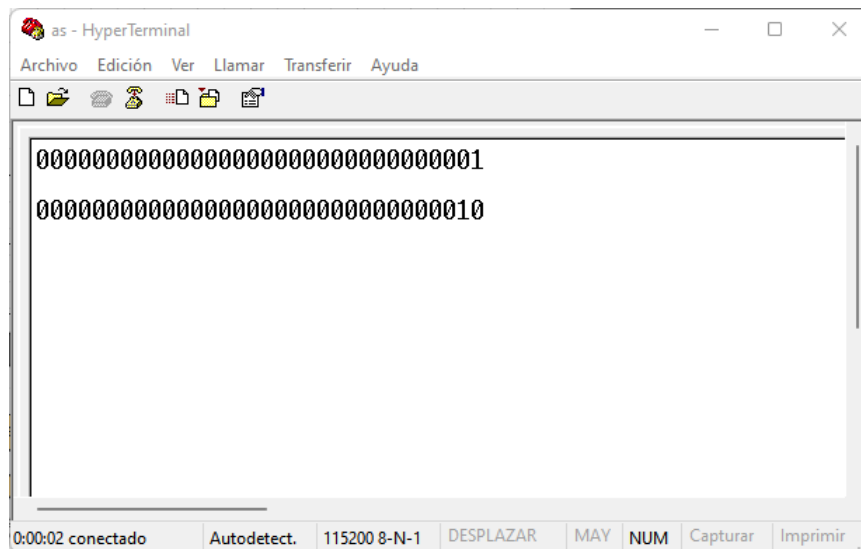


Figura 21. Ventana del programa Hyperterminal con el resultado del test de la operación SUBCY

- SHIFTRS232

Esta instrucción de elaboración propia se ha añadido al microcontrolador a fin de mostrar un ejemplo de aprovechamiento de la modificación de 8 a 32 bits. Se encarga de rotar 8 posiciones a la izquierda un registro dado. Para este test, se van a cargar en un registro 4 caracteres ASCII: H, O, L, A. La idea de la prueba es aplicar un pequeño bucle para poder disponer en cada iteración de un bloque de bits diferente. Esto va a permitir que, usando un solo registro, se pueda enviar la palabra HOLA mediante el puerto serie sin necesidad de cargar cada una de las letras por separado.

	LOAD	aux, 484F4C41 ;HOLA
	LOAD	aux2, 00000004
bucle:	SHIFTRS232	aux
	LOAD	txreg, aux
	CALL	transmite
	SUB	aux2, 00000001
	JUMP	NZ, bucle

Tras la ejecución de este test, aparece el siguiente mensaje por el Hyperterminal, mostrando como con una simple rutina se puede conseguir aprovechar de manera completa un registro de 32 bits con datos de 8 bits. Se asemejaría al funcionamiento de un array de un entorno de desarrollo de software.



Esta función permite rotar un registro hacia la derecha en una posición. El bit menos significativo será desplazado y será ahora el más significativo. Para probarla, se cargará un valor en un registro (0x0000000F) y se le aplicará la instrucción, debiendo obtener 0x80000007.

```
LOAD      aux, 0000000F ;RR
RR        aux
CALL      show
CALL      saltoLinea
```



- RL

Del mismo modo que la función RR, RL realiza una rotación del registro, pero en este caso, hacia la izquierda. Para comprobarlo, se carga un valor en un registro (0xF0000000) y se ejecuta la instrucción. El resultado esperado es 0xE0000001.

```
LOAD      aux, F0000000 ;RL
RL        aux
CALL      show
CALL      saltoLinea
```

El resultado se podrá ver usando la rutina show cargando el valor en la variable aux, que es la que toma como referencia la rutina a la hora de mostrar un dato.

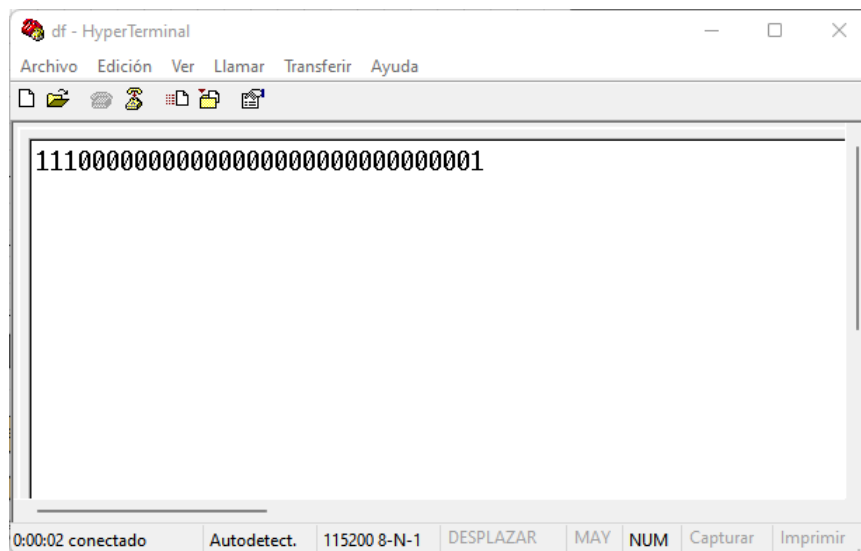


Figura 24. Ventana del programa Hyperterminal con el resultado del test de la operación RL

- SR0

Esta instrucción que incorpora el microcontrolador realiza una acción muy simple, como es el desplazamiento de un registro en una dirección y rellenando con un bit (0 en este caso). El registro se desplaza una posición a la derecha, desplazando “fuera” del registro al bit menos significativo. Como esto dejaría una posición vacía en el registro (es una instrucción pensada para registros de 8 bits), se rellena con un 0 en la posición más significativa.

```
LOAD      aux2, 000000C3 ;SR0 -> a
SR0       aux2
LOAD      txreg, aux2
CALL      transmite
CALL      saltoLinea
```

Para poder hacer la prueba de funcionamiento, se usarán valores hexadecimales que representen caracteres ASCII. Para esta prueba, se cargará el valor C3 (hex.) que equivale a 1100 0011 y se ejecutará la instrucción. Tras esto, se debe obtener el carácter 61 (hex.), que equivale a 0110 0001. En ASCII, este carácter es la “a”, que es justo el dato que se recibe por el Hyperterminal.

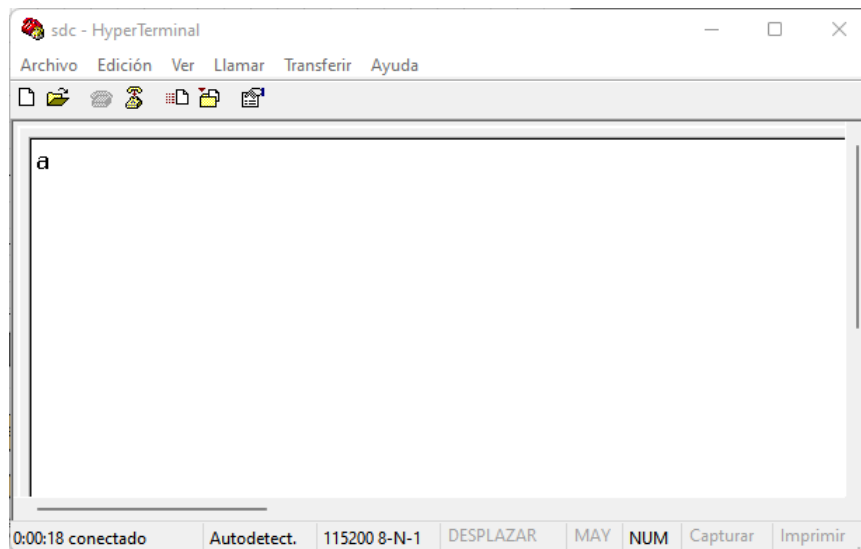


Figura 25. Ventana del programa Hyperterminal con el resultado del test de la operación SRO

- SR1

Para esta instrucción ha sido necesario emplear algunas de las instrucciones que se han verificado hasta el momento (RR y SHIFTRS232). Esta instrucción desplaza el registro hacia la derecha, haciendo que este desborde por la posición más baja y dejando un hueco en el bit más significativo, el cual se completa con un uno en este caso. Para esta prueba, se espera obtener en el Hyperterminal el símbolo @.

LOAD	aux2, 00000000 ;SR1 -> @
SR1	aux2
RR	aux2
SHIFTRS232	aux2
LOAD	txreg, aux2
CALL	transmite
CALL	saltoLinea

Para esta prueba se ha reiniciado un registro y se le ha aplicado la instrucción SR1, lo que deja al registro con un uno en el bit más significativo y el resto lleno de ceros. Tras esto, se rota el bit una posición a la derecha con la instrucción RR (el dato en las 8 posiciones más altas sería 0100 0000) y tras esto se llama a la instrucción SHIFTRS232. Esto hará que el dato pase a las 8 posiciones más bajas y pueda ser transmitido y reflejando el resultado, que es el esperado.

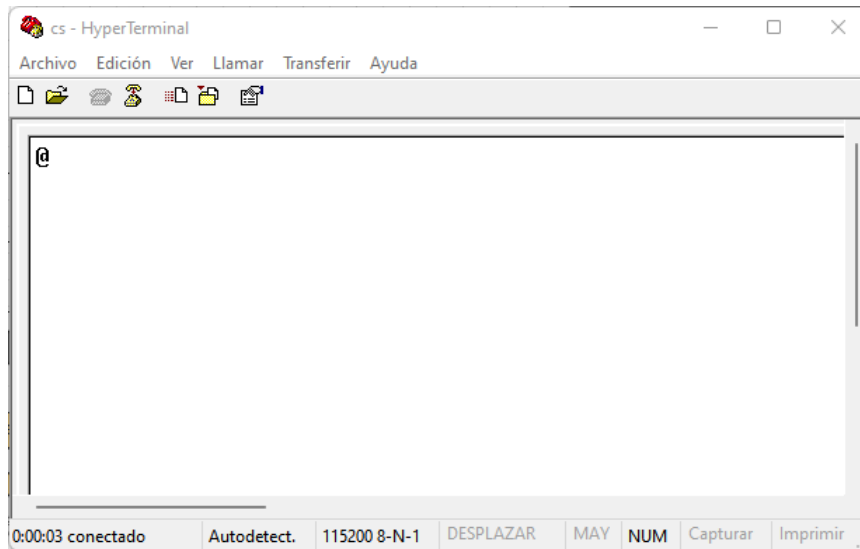


Figura 26. Ventana del programa Hyperterminal con el resultado del test de la operación SR1

- SLO

Esta instrucción es muy similar a SR0. La funcionalidad es la misma, solo cambia el sentido de desplazamiento. En este caso, se desplaza el registro a la izquierda, lo que deja una posición vacía en el bit menos significativo, que en este caso se completa con un cero.

LOAD	aux2, 00000033 ;SLO -> f
SLO	aux2
LOAD	txreg, aux2
CALL	transmite
CALL	saltoLinea

Para esta prueba, se cargará en un registro el valor 0x00000033. Al aplicar la instrucción, el registro se desplazará a la izquierda, añadiendo un 0 por la derecha. El resultado esperado es el doble del introducido, 0x00000066, que corresponde en ASCII a una "f".

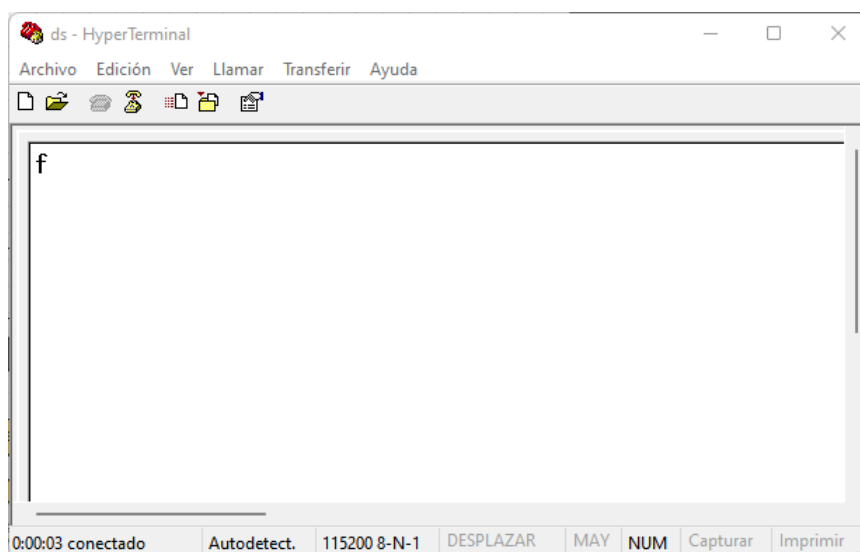


Figura 27. Ventana del programa Hyperterminal con el resultado del test de la operación SLO

- SL1

Esta es una instrucción muy similar a la que se ha verificado antes. En este caso, al desplazar a la izquierda, se rellena por la derecha con un uno. El procedimiento de la prueba, como se puede observar, es muy similar.

LOAD	aux2, 00000030 ;SL1 -> a
SL1	aux2
LOAD	txreg, aux2
CALL	transmite
CALL	saltoLinea

En este caso, se carga en un registro el valor hexadecimal 0x00000030. Al aplicarle la instrucción, de igual manera que en caso anterior, se obtendría el valor doble de lo que se cargó en un principio, y dado que el hueco que queda tras la ejecución de la instrucción se rellena con un uno, se tiene que el resultado obtenido debería ser 0x00000061, que equivale a la “a” en ASCII.

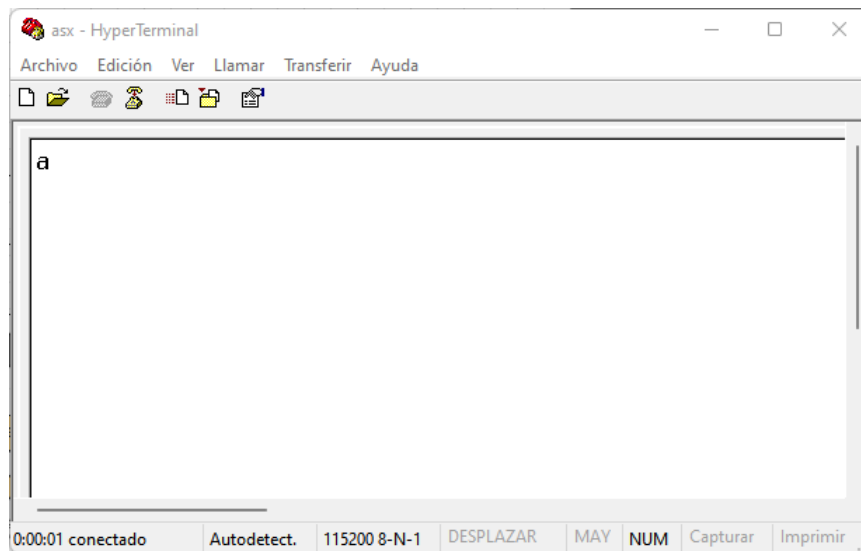


Figura 28. Ventana del programa Hyperterminal con el resultado del test de la operación SL1

- SRX

Esta instrucción funciona un poco distinta a las que se han visto en las últimas pruebas. También realiza una función de desplazamiento a la derecha, de manera que deja un hueco en el bit más significativo que hay que rellenar, pero en este caso, se completará con una réplica del bit más significativo. De modo, que un registro esta contiene el siguiente dato: ABCDEF89, tras la aplicación de la instrucción se debería obtener AABCDEF9.

LOAD	aux2, 70000000 ;SRX -> 8
SRX	aux2
SHIFTRS232	aux2
LOAD	txreg, aux2
CALL	transmite
CALL	saltoLinea

En el test se carga en un registro el valor 0x70000000, el cual tiene como bit más significativo un 0. Tras aplicar la instrucción, los 8 bits más significativos son 0011 1000. Al aplicar

la función SHIFTRS232, estos 8 bits pasan a las ocho posiciones más bajas, permitiendo enviar el dato por el Hyperterminal, el cual muestra el carácter “8” como estaba previsto.

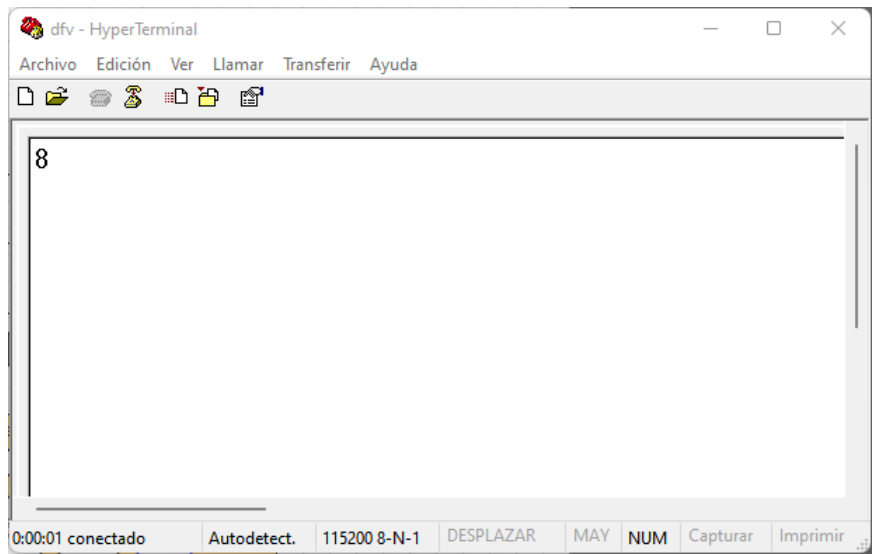


Figura 29. Ventana del programa Hyperterminal con el resultado del test de la operación SRX

- SLX

Funciona de una manera idéntica a la instrucción SRX, solo que este caso se invierte el sentido. El registro se desplazará a la izquierda dejando el hueco en el bit menos significativo. Será el que previamente al aplicar la instrucción se encontraba en esa posición el que se replique.

LOAD	aux2, 00000019 ;SLX -> "3"
SLX	aux2
LOAD	txreg, aux2
CALL	transmite
CALL	saltoLinea

En esta prueba, se carga en un registro el valor 0x00000019 y se le aplica la instrucción a comprobar. En teoría, si se observan los 8 bits menos significativos, que son 0001 1001, tras la ejecución de la instrucción se debería obtener 0011 0011 en las 8 posiciones más bajas (el resto no se ven afectadas, siguen siendo ceros). El resultado esperado es el carácter “3”, que es lo que muestra el Hyperterminal.

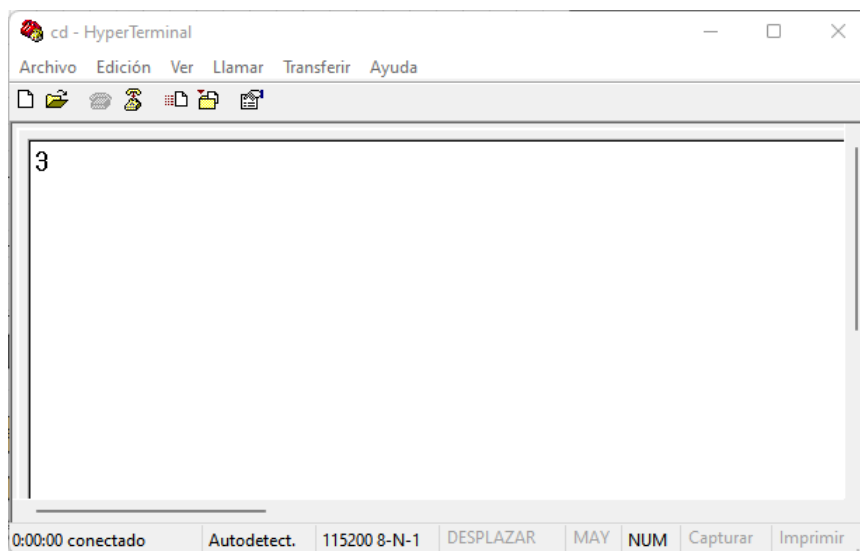


Figura 30. Ventana del programa Hyperterminal con el resultado del test de la operación SLX

- SRA

Esta instrucción, al igual que las últimas que se han visto, desplaza a un registro en un sentido. En este caso, hacia la derecha, lo que genera un hueco en el bit más significativo del registro al que se le aplica la instrucción que en este caso, se rellena con el bit de acarreo que haya en el instante de la ejecución. Para este tipo de pruebas se van a plantear dos casos, uno en el que el bit de acarreo valga 1 y otro en el que valga 0.

LOAD	aux 000000FF ;SRA -> 0xC000003F
SR1	aux
SRA	aux
CALL	show
CALL	saltoLinea
LOAD	aux 000000FF ;SRA -> 0x0000007F
SRA	aux
CALL	show
CALL	saltoLinea

Para el primer caso, se desborda un registro asegurando que el bit que desborda sea un 1, para que el bit de acarreo valga 1. Tras el primer paso, queda que el dato intermedio es 0x8000007F. Tras aplicar la instrucción SRA, se debe desplazar otra vez a la derecha, introduciendo en el bit más significativo el valor del bit de acarreo, que debe ser uno y por tanto devolver un 0xC000003F.

Para el segundo caso, se parte del mismo registro que en la anterior prueba, pero en este caso no va a ser desbordado, por lo que el bit de acarreo debe ser 0 y eso es lo que debe introducir la instrucción al aplicarse, siendo el dato esperado 0x0000007F.

Los resultados esperados para ambas pruebas coinciden con lo que devuelve el Hyperterminal.

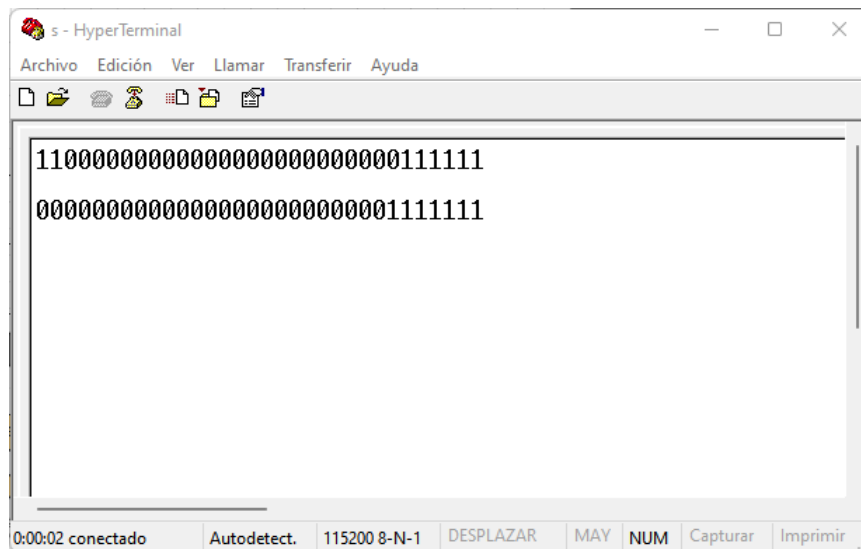


Figura 31. Ventana del programa Hyperterminal con el resultado del test de la operación SRA

- SLA

Esta instrucción es muy idéntica a SRA, solo que en este caso se invierte el sentido de desplazamiento hacia la izquierda. De nuevo, se deben que plantear dos situaciones en las que el bit de acarreo valga 1 y 0 en dos pruebas diferentes, para verificar el correcto funcionamiento.

LOAD	aux, FF000000 ;SLA -> 0xFC000003
SL1	aux
SLA	aux
CALL	show
CALL	saltoLinea
LOAD	aux, FF000000 ;SLA -> 0xFE000000
SLA	aux
CALL	show
CALL	saltoLinea

Para el primer caso, se va a desbordar el registro desplazando el bit más significativo, que es un 1, fuera de este, haciendo que el bit de acarreo pase a valer 1. El valor intermedio será 0xFE000001. Tras esto, se aplica la instrucción con el bit de acarreo a 1 y el resultado debe ser 0xFC000003, ya que desplaza de nuevo a la izquierda introduciendo un bit a 1 por el bit de acarreo.

Para el segundo caso, se realiza la misma prueba, pero esta vez sin desbordar el registro, por lo que la instrucción introducirá un 0 en la posición menos significativa, por lo que el resultado esperado es 0xFE000000.

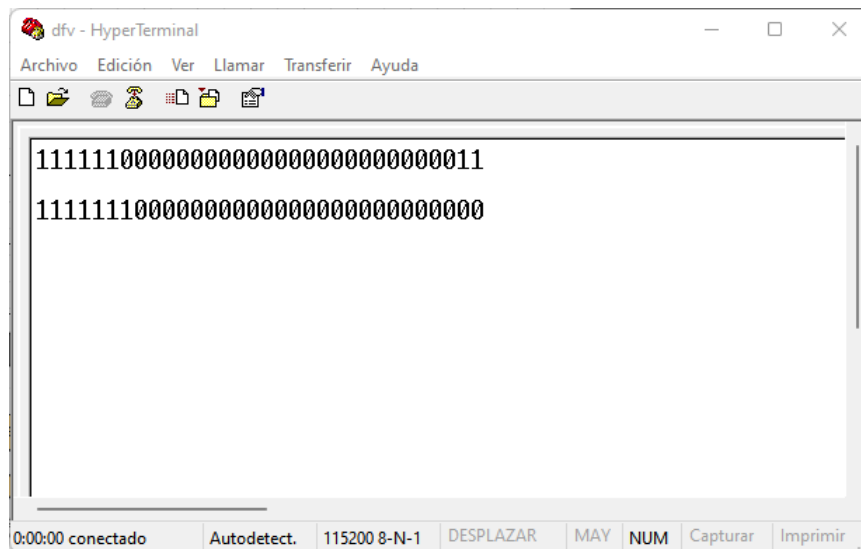


Figura 32. Ventana del programa Hyperterminal con el resultado del test de la operación SLA

- INPUT, OUTPUT

Este par de instrucciones están relacionadas con la entrada y salida de datos del microcontrolador con cualquier periférico que esté conectado. Estas instrucciones, indicándole un registro del que saldrán o donde se recibirán los datos y un valor que es el identificador de puerto al que está conectado el periférico en cuestión, se encargan de recibir y transmitir datos. Para poder comprobar el correcto funcionamiento de estas instrucciones, se va a usar el periférico que se ha agregado al microcontrolador y que será analizado en profundidad en el siguiente apartado.

INPUT	aux, 0000005F
CALL	show
CALL	saltoLinea
LOAD	aux2, FF0000FF
OUTPUT	aux2, 00000055
INPUT	aux, 00000055
CALL	show
CALL	saltoLinea

En este código se va a probar el funcionamiento de ambas instrucciones. Para INPUT, se debe indicar que se desea recibir los datos en una variable, en este caso la variable aux. Tras la coma, se indica que estos datos los debe obtener del puerto 0x0000005F. Tras ejecutar la instrucción, se llama a la subrutina show para que muestre por el Hyperterminal el valor del registro aux, que coincidirá con uno de los datos contenidos en el periférico.

Para el segundo caso, se cargará un valor cualquiera en uno de los registros del microcontrolador y a continuación se ejecutará la instrucción OUTPUT, indicándole la variable que contiene los datos a enviar y la dirección del puerto donde se desea que lo envíe. Para verificar que el dato se ha escrito correctamente, se usará INPUT para leer del puerto al que se le había enviado el dato, para verificar que este se envió correctamente.

Como se puede observar, los datos obtenidos son correctos, ya que consultando en la tabla RAM del periférico, se puede ver que el último dato (seleccionado con la primera prueba), se corresponde con lo que se obtiene por el Hyperterminal, y el segundo dato es el mismo que se introdujo en él mediante el uso de la instrucción OUTPUT.

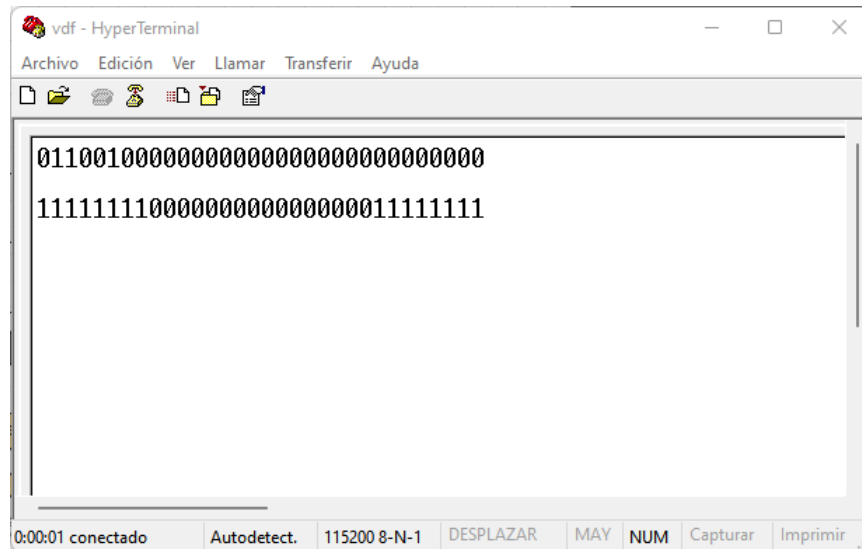


Figura 33. Ventana del programa Hyperterminal con el resultado del test de las operaciones INPUT y OUTPUT

- ENABLE, DISABLE, RETURNI

Estas instrucciones están relacionadas con el manejo de las interrupciones en el microcontrolador. Sirven para bien habilitarlas o deshabilitarlas, siendo RETURNI la instrucción que se ejecuta cuando se termina la rutina de interrupción. Cuando una interrupción se produce en el PicoBlaze, se almacena la dirección de la memoria de programa en la que se encuentra el microcontrolador en el momento previo a la interrupción, permitiendo así volver al punto en el que se encontraba el programa de manera sencilla.

Para poder probar su funcionamiento es necesario interactuar con la FPGA, ya que el pulsador BTNU está asignado a la señal que genera una interrupción en el PicoBlaze.

```
rutina: LOAD      aux, 00000005
          LOAD      aux3, 00000040

rutina1: CALL      transmite
          SUB        aux, 00000001
          JUMP       Z, rutina2
          ADD        aux3, 00000001
          LOAD       txreg, aux3
          JUMP       rutina1

rutina2: ENABLE    INTERRUPT
          LOAD       aux2, 0000000A
bucle:  SUB        aux2, 00000001
          JUMP       NZ, bucle
          LOAD       aux2, 0000000A
          JUMP       bucle

interrup:          ;ESTA SUBROUTINA HA DE COLOCARSE AL FINAL DEL PROGRAMA
          DISABLE   INTERRUPT
```

```

CALL      recibe
LOAD      txreg, rxreg
call      transmite
LOAD      txreg, aux2
ADD       txreg, 00000030
call      transmite
RETURNI   ENABLE
LOAD      txreg, 0000002F
CALL      transmite
ADDRESS   000007FF      ;aquí salta cuando hay una
interrupcion
JUMP      interrup

```

Este conjunto de rutinas va a permitir verificar el funcionamiento de este grupo de instrucciones de una manera veraz. En primer lugar, se carga el valor ASCII del carácter “@” junto con un contador a 5. Mediante rutina1 se van a escribir cuatro caracteres: A, B, C y D. Tras esto, el programa salta a rutina2 donde se habilita la interrupción y el programa entra en un bucle infinito. Aquí el programa espera hasta que se produce una interrupción (pulsando BTNU en la FPGA), lo que hace que el contador de programa salte a la última línea (ADDRESS 000007FF) y de ahí entre a la rutina de atención a la interrupción. En ella, se hace lo siguiente: en rutina2 se ha diseñado un bucle infinito y en él se introduce un contador que va de 0 a 10, que se va cargando en la variable aux2. Una vez el programa llega a la rutina de interrupción, toma ese valor, le suma 0x30 (dirección a partir de la cual comienzan las letras del abecedario en la tabla ASCII) y el resultado es un carácter numérico ASCII aleatorio. La prueba de que funcionan las interrupciones es que solo cuando se pulsa el BTNU, el microcontrolador responde a una pulsación del teclado (está esperando debido a *CALL recibe* en la rutina interrup) devolviendo un carácter numérico aleatorio.

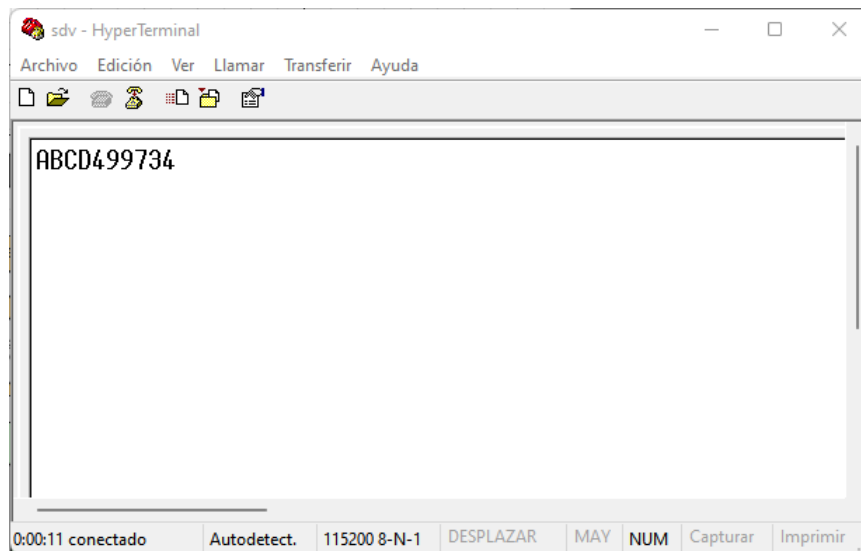


Figura 34. Ventana del programa Hyperterminal con el resultado del test de la operación ENABLE, DISABLE y RETURNI

- FLIP

Esta instrucción tiene la funcionalidad de alternar por completo un registro, invirtiendo las posiciones de los bits que lo conforman. Se aplica sobre un registro y en este mismo se graba el resultado de la instrucción.

```

LOAD      aux, 0F00000F
CALL      show
CALL      saltoLinea
FLIP      aux
CALL      show
CALL      saltoLinea

LOAD      aux, 3C0000FF
CALL      show
CALL      saltoLinea
FLIP      aux
CALL      show
CALL      saltoLinea

```

Para este test, se cargan unos valores en unas posiciones determinadas de un registro, se les aplica la instrucción, y como se podrá observar las posiciones se han invertido.

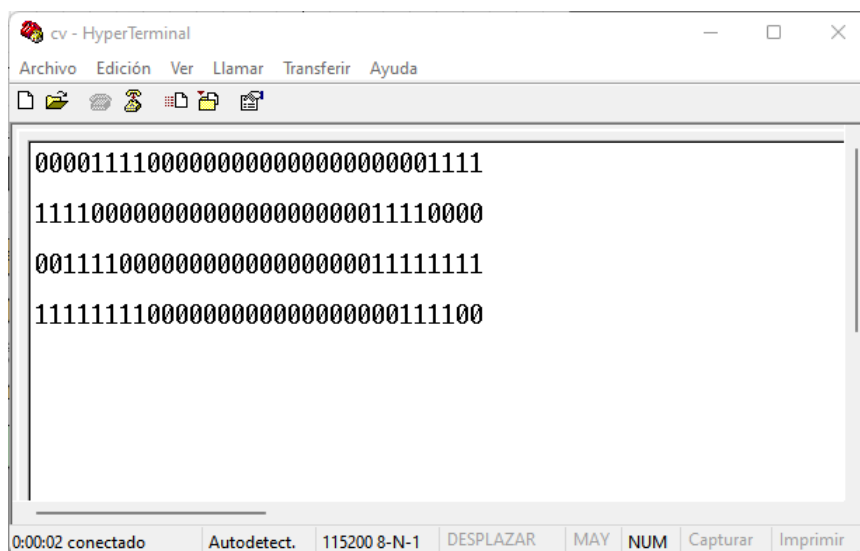


Figura 35. Ventana del programa Hyperterminal con el resultado del test de la operación FLIP

5. Implementación del módulo SECDEC

5.1. Diseño del periférico

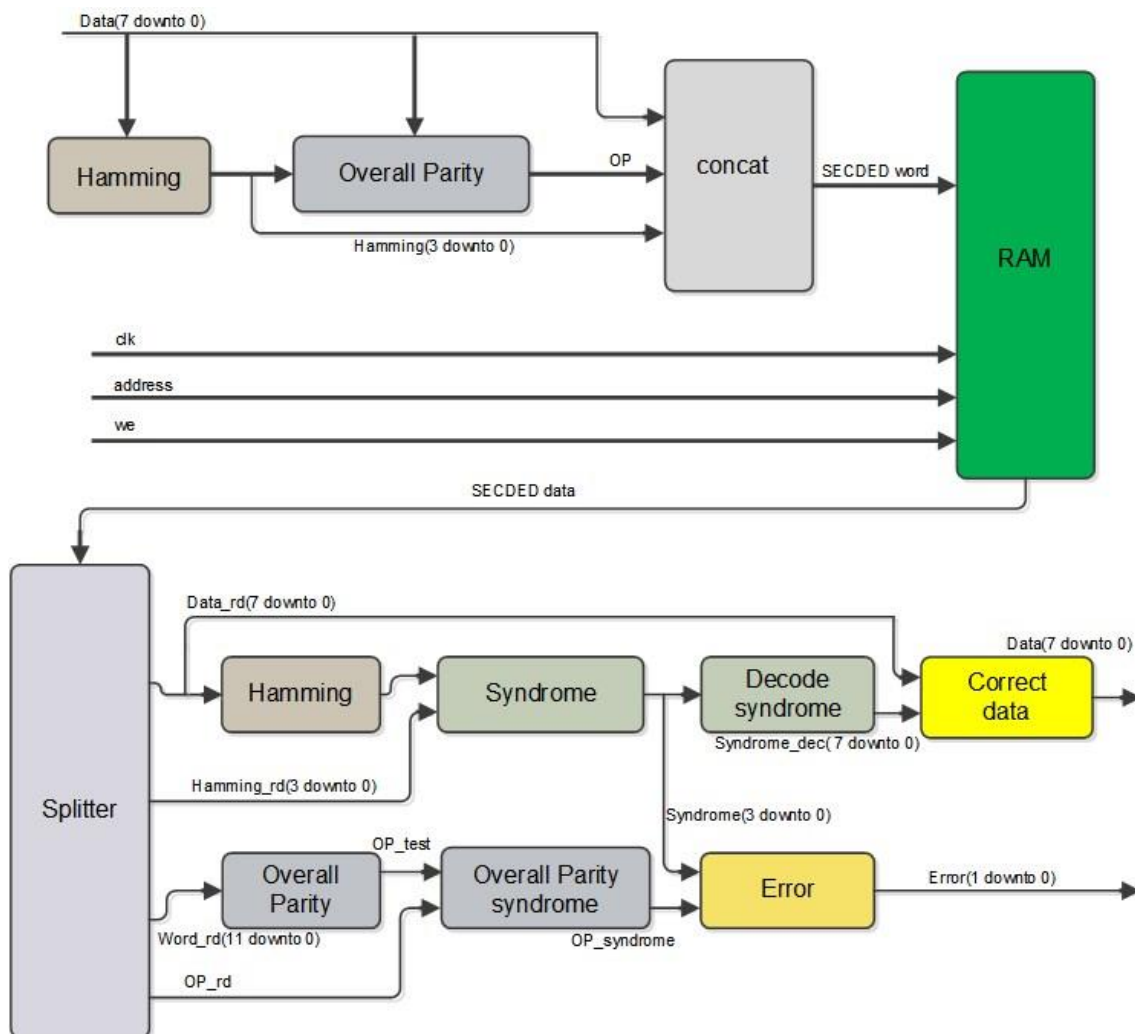


Figura 36. Diagrama del funcionamiento del módulo SECDED a 8 bits

El periférico comienza sus operaciones recibiendo un dato por su puerto de entrada. A dicho dato, se le calcula su código hamming, su bit de paridad general (a partir de los bits hamming) y se concatena toda la información según se ha visto en el Tabla 2. Tras esto, se introduce el dato en la memoria RAM en el siguiente ciclo de reloj, en la dirección que se le indique y siempre que la señal write enable esté habilitada (we).

Se ha completado la codificación y almacenamiento de un dato en codificación SECDED. Si en cambio, se busca obtener un dato de la memoria, el proceso es el siguiente.

Se indica el dato al que se quiere acceder usando la señal address y se carga en SECDED data. El dato llega a un divisor, que se encarga de repartir diferentes partes del dato codificado a los diferentes módulos que participan en la decodificación del dato. Se parte del SECDED data

y se introduce en un módulo Hamming los bits correspondientes al dato original, y el SECDED data al completo en el módulo Overall Parity. Estos procesos se ejecutan de manera paralela.

Tras este primer paso, del módulo hamming, se pasa el resultado al módulo síndrome, que comparará los bits de hamming recién calculados con los que había almacenados. Paralelo a esto, se ejecuta un proceso similar con los datos que se obtienen del módulo Overall Parity, introduciendo el bit OP calculado en el módulo Overall Parity Syndrome, que comparará el bit OP recién obtenido con el almacenado.

El resultado de ambos syndrome lo recibe el módulo error, que se encarga de analizar los resultados y según su lógica establecida, determina si hay error y si lo hay, de que tipo es. Paralelamente, se decodifica el síndrome en otro módulo, llamado Decode Syndrome, que devolverá un dato de 8 bits indicando en qué posición se ha producido una alteración. Tras esto, se compara el dato con lo que había almacenado y si fuera necesario, se realiza la corrección.

5.2. Migración del módulos SECDEC de 8 bits a 32 bits

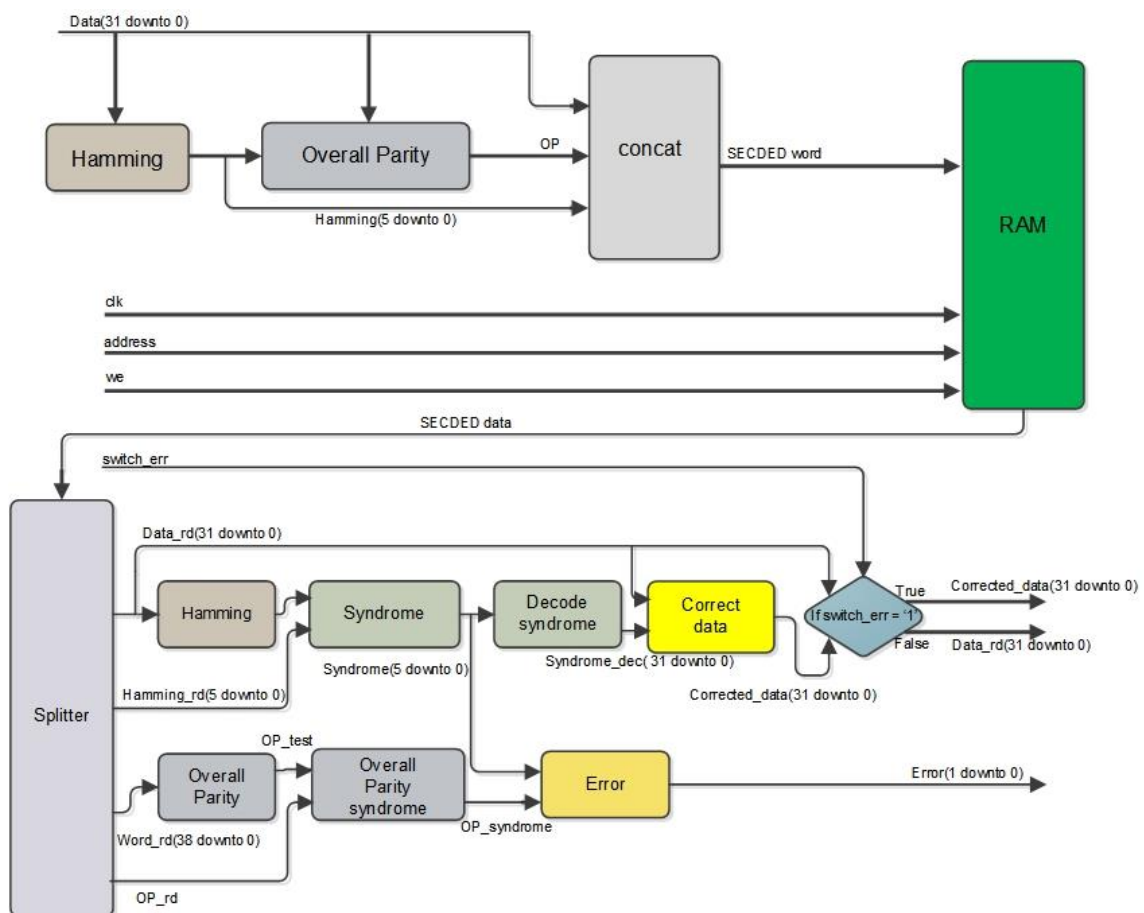


Figura 37. Diagrama de funcionamiento del módulo SECDED a 32 bits

Para hacer funcionar al módulo a 32 bits, es necesario cambiar el tamaño de las señales que componen el periférico. En este caso, la complejidad es menor que en la adaptación del PicoBlaze a 32 bits, debido a que aquí no se modifica el tamaño de la memoria, solo el de sus registros.

En primer lugar, se ha de identificar aquellas señales cuyo no coincida con las nuevas medidas. En el caso de las señales originalmente de 8 bits, se puede afirmar que manejan el dato real, sin bits de paridad. Por tanto, se pueden ampliar el tamaño de todas y cada una de estas señales de 8 a 32 bits (31 down to 0).

Con esto, se han conseguido que los nuevos datos puedan ser manejados por las señales internas del periférico, sin bits de paridad. Pero aún hace falta modificar más señales para que funcione. A continuación, se deben encontrar todas aquellas señales que se encarguen de manejar el dato con los bits de paridad incluidos. Originalmente, son todas aquellas señales cuyo tamaño sea de 13 bits. Para ellas, hay que modificar su declaración e indicar su nuevo tamaño, que es de 39 bits.

El último tipo de señales a modificar son aquellas relacionadas con la paridad, para las cuales se produce un salto de 4 bits a 6 bits. El resto de señales no han de ser modificadas.

Modificadas todas las señales necesarias, es el momento de modificar instancia de la memoria RAM, que es una tabla con los datos pre almacenados. En este caso, basta con ampliar hacia la izquierda el número de ceros hasta que cada dato tenga 39 bits. Más adelante será modificado para hacer test.

En cuanto a los procesos combinacionales, la manera de proceder viene determinada por la siguiente tabla:

Coded word	Bit position	P5	P4	P3	P2	P1	P0
P0	0	0	0	0	0	0	1
P1	1	0	0	0	0	1	0
D0	2	0	0	0	0	1	1
P2	3	0	0	0	1	0	0
D1	4	0	0	0	1	0	1
D2	5	0	0	0	1	1	0
D3	6	0	0	0	1	1	1
P3	7	0	0	1	0	0	0
D4	8	0	0	1	0	0	1
D5	9	0	0	1	0	1	0
D6	10	0	0	1	0	1	1
D7	11	0	0	1	1	0	0
D8	12	0	0	1	1	0	1
D9	13	0	0	1	1	1	0
D10	14	0	0	1	1	1	1
P4	15	0	1	0	0	0	0
D11	16	0	1	0	0	0	1
D12	17	0	1	0	0	1	0
D13	18	0	1	0	0	1	1
D14	19	0	1	0	1	0	0
D15	20	0	1	0	1	0	1
D16	21	0	1	0	1	1	0
D17	22	0	1	0	1	1	1
D18	23	0	1	1	0	0	0
D19	24	0	1	1	0	0	1
D20	25	0	1	1	0	1	0
D21	26	0	1	1	0	1	1
D22	27	0	1	1	1	0	0
D23	28	0	1	1	1	0	1
D24	29	0	1	1	1	1	0
D25	30	0	1	1	1	1	1
P5	31	1	0	0	0	0	0
D26	32	1	0	0	0	0	1
D27	33	1	0	0	0	1	0
D28	34	1	0	0	0	1	1
D29	35	1	0	0	1	0	0
D30	36	1	0	0	1	0	1
D31	37	1	0	0	1	1	0
PB	38	0	0	0	0	0	0

Tabla 2. Codificación SEDED de los bits de paridad

Las filas representan el total de bits necesarios entre bits de dato y bits de paridad. Para cada fila que contenga un bit de dato (aquellas que no están en negro), se indica con un 1 si este se usa para calcular un determinado bit de paridad o un 0 si no tiene influencia en el cálculo.

Conocida ya la lógica de funcionamiento del componente, las modificaciones en el código deben ir regidas por este criterio, de manera que si para calcular el bit de paridad 3 (P3) antes se usaba hasta el octavo bit del dato (D7), en cambio ahora se usarán todos aquellos datos que en la columna P3 estén marcados con un 1. Al mismo tiempo, se puede usar la columna nombrada como Coded word para saber el orden de los bits en cada palabra codificada.

5.3. Conexión del Módulo SECDEC al Sistema

Para poder hacer funcionar este módulo, es necesario integrarlo con el microcontrolador. Esta integración consiste en declarar el periférico como un componente en el bloque toplevel y conectar sus señales con las del PicoBlaze.

En primer lugar, se debe declarar el componente para que pueda ser reconocido.

```
component SECDEDRAM is
  port (clk          : in std_logic;
        we           : in std_logic;
        din          : in std_logic_vector (31 downto 0);
        address      : in std_logic_vector (31 downto 0);
        dout         : out std_logic_vector (31 downto 0);
        error_int    : out std_logic_vector (1 downto 0);
        switch_err   : in std_logic);
end component;
```

Una vez declarado el componente y todas sus señales de entrada y salida, se procede a mapear dichas señales con otras señales internas del bloque toplevel para poder manejar los datos.

```
secded_ram_core : SECDEDRAM
  port map (clk => clk,
            we => writestrobe,
            din => outport,
            address => portid,
            dout => secded_out,
            error_int => secded_err,
            switch_err => err_en);
```

En el mapeo de estas señales, se pueden destacar algunas sobre el resto por la complejidad que tienen. Una de las más complejas es la señal address. Esta señal es de 32 bits, pero el periférico solo toma en cuenta los 4 bits menos significativos que se obtienen de la señal portid. Esto se debe a que el número de datos almacenados es de 16, con lo que con 4 bits se puede abarcar todo el rango de posiciones. Esto, como se podrá ver más adelante, es clave para apuntar a los datos deseados y recibirlos por el multiplexor del bloque toplevel. A la vez que entraña otro problema, y es que la señal portid es usada constantemente por el PicoBlaze para numerosas acciones, por lo que ese valor puede variar mucho en un espacio pequeño de tiempo hacer que se apunten a datos no deseados. La solución a este problema será mostrada más adelante, en las modificaciones realizadas al código VHDL del microcontrolador

Otra señal destacable, es la señal error_int la cual contiene el código de error del dato que se está analizando en cada momento dentro del periférico y que escribe su valor en la señal secded_err.

Para poder mostrar el código de error que se obtiene de la señal `error_int`, se ha diseñado un proceso integrado en el bloque toplevel, que se encarga de transmitir el valor que se obtiene a dos leds integrados en la placa, para mostrar de manera visual el código obtenido.

```
showErr:process(reset, clk)
begin
    if(reset = '1') then
        led_err <= "00";
    elsif(clk'event and clk= '1') then
        led_err <= secded_err;
    end if;
end process;
```

Hablando del resto de señales, se tiene que en primer lugar la señal `clk`, que se conecta al reloj del sistema, permitiéndole estar sincronizado con el resto de los elementos del diseño. La señal `we` se conecta a `writestrobe`, que se activa cuando se intenta escribir o enviar un dato a través del microcontrolador usando cualquiera de sus puertos, permitiendo así enviar datos desde el picoblaze hasta el periférico. La señal de entrada `din` es el canal por el que el periférico recibe los datos que se le envían desde bus de salida del picoblaze.

Para las dos señales de salida, se han creado dos señales internas a las que asignar los datos de salida del periférico, para poder manejarlos e introducirlos en el picoblaze. Para `dout` se ha declarado la señal `secded_out`, que almacenará el dato y se lo pasará a `inport`, que es el bus de entrada de datos del picoblaze, cuando se active la lectura de datos del multiplexor. Para la segunda señal de salida, que es la que devuelve el código de error de operación del periférico, se ha asignado a otra señal interna llamada `secded_err` para que almacene el dato.

Finalmente, resta por explicar el funcionamiento del multiplexor y como interacciona con el nuestro periférico. Para que se reciba un dato del periférico, antes se deben dar dos condiciones. La primera es que se cumpla que la señal `readstrobe` del picoblaze esté a '1', esto es, que el microcontrolador esté listo para recibir un dato a través de un puerto en uno de sus registros. La segunda condición es que el valor del `portid` esté dentro de unos valores determinados. En este caso, el `portid` debe estar entre la dirección `0x00000050` y la dirección `0x0000005F`, abarcando así un total de 16 puertos, tantos como entradas tiene la tabla RAM del periférico.

De modo que, si desde la memoria de programa contenida en el picoblaze se ejecuta, por ejemplo, la instrucción *INPUT data, 00000050* (siendo `data` un registro reconocido por el

```
--Multiplexor inport
mux:

    inport <= rxbuff_out when (readstrobe = '1' and portid =
x"00001111") else

    secded_out when (readstrobe = '1' and (portid >= x"00000050"
and portid < x"00000060")) else

    x"00000000"&"00"&secded_err when (readstrobe = '1' and portid
= x"00000070") else

    x"00000000";
```

microcontrolador y 00000050 la dirección de la que se desea recibir el dato), en el multiplexor se cumplirá la condición de readstrobe = '1', ya que desde la perspectiva del picoblaze se está intentando obtener un dato de fuera. En lo que a la dirección del puerto se refiere, se cumple la condición anteriormente mencionada, por lo que se enviará el dato del multiplexor al microcontrolador, que recibirá el dato que indique el último dígito escrito en la instrucción. De modo que, si el valor del portid está entre los límites que se han mencionado anteriormente, será el último dígito hexadecimal el que determine cuál de los 16 registros está elegido.

Debido a que la señal address del periférico está directamente conectada a la señal portid, debemos realizar un filtrado de señales, ya que, si se estuviera ejecutando alguna otra rutina, esto provocaría que el valor de la señal address apuntara a otros datos y no controlaríamos que valor estamos rescatando de la memoria, lo que haría que el valor de la señal de salida (dout) cambiase de manera descontrolada. Esto también nos impediría visualizar el código de error correctamente, ya que a medida que el dato varía, el código de error también lo hace, impidiendo visualizarlo correctamente en los leds de la Zedboard.

Para solucionar esto, se impone una condición al proceso del periférico que apunta a una posición de memoria según el valor de address, indicando que solo debe actuar cuando el valor de la señal se encuentra entre el valor x"00000050" y x"0000005F". De esta manera se asegura que la RAM solo apunta a una posición de memoria cuando queremos leer o escribir un dato. Esto es fundamental ya que el proceso que apunta a una posición de memoria solo toma en cuenta los cuatro bits menos significativos en el rango completo (en hexadecimal) de x"0" a x"F", lo que provocaría que por cada cambio de la señal portid (conectada a address), se apuntase a un dato diferente, devolviendo un valor diferente de dout y en error_int por cada cambio. El proceso modificado resulta así.

```
RD_WR_RAM:process (clk)
begin
    if (clk'event and clk = '1') then
        if(address >= x"00000050" and address < x"00000060") then
            if (we = '1') then
                ram_mem(to_integer(unsigned(address(3 downto 0) ))) <=
code_2_mem;
            end if;
            rd_mem_code <= ram_mem(to_integer(unsigned(address(3
downto 0) )));
        end if;
    end if;
end process;
```

5.4. Resultados y test de funcionamiento

Para poder comprobar el correcto funcionamiento del periférico y verificar que tanto las modificaciones realizadas como la integración con el PicoBlaze es válida, se han realizado algunas pruebas usando el programa asm compilado que se agrega al microcontrolador.

Se desea comprobar que el periférico permite escribir datos, leerlos y que tiene sus propiedades correctoras. A continuación, se va a comprobar si permite introducir datos en la memoria del periférico y posteriormente, leerlos.

más, modificando el dato introducido y el que se empleará (se usarán la cuarta y onceava posición).

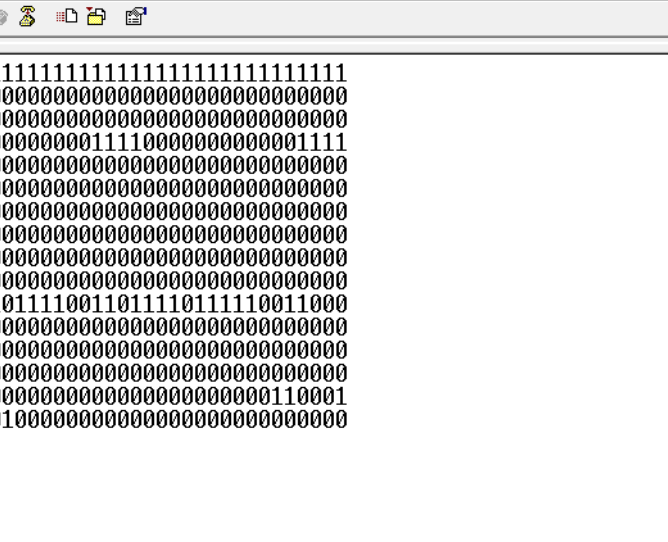
Para poder verificar que las instrucciones aplicadas han funcionado, se ejecuta a continuación una prueba de lectura de la memoria, para lo cual se leerán cada una de las 16 posiciones que tiene la tabla de la memoria RAM del periférico. Esto lo se realizará justo

```
recorrer:

    INPUT    rxreg, 00000050 ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 00000051 ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 00000052 ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 00000053 ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 00000054 ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 00000055 ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 00000056 ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 00000057 ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 00000058 ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 00000059 ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 0000005A ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 0000005B ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 0000005C ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 0000005D ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 0000005E ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    INPUT    rxreg, 0000005F ;obtengo un dato
    LOAD     aux, rxreg; cargo el dato en aux
    CALL     show ;lo muestro
    CALL     saltoLinea

RETURN
```

Dentro de la rutina recorrer, se invoca a la subrutina show, gracias a la que aparecerán en la pantalla los 16 registros que conforman la memoria del periférico, permitiendo comprobar si los datos relevados son los mismos que en el momento previo a la programación de la placa (revisándolos en el editor VHDL) o, por el contrario, aquellos datos que habían sido modificados se mantienen.



The screenshot shows a HyperTerminal window titled "sc - HyperTerminal". The menu bar includes "Archivo", "Edición", "Ver", "Llamar", "Transferir", and "Ayuda". The toolbar contains icons for file operations and communication. The main text area displays a binary sequence consisting of 15 lines of characters. The first line is a string of 16 '1's. The subsequent lines are mostly '0's, with some '1's appearing in the 11th, 12th, and 14th lines. The status bar at the bottom shows "0:00:05 conectado", "Autodetect.", "115200 8-N-1", "DESPLAZAR", "MAY", "NUM", "Capturar", and "Imprimir".

```

11111111111111111111111111111111
00000000000000000000000000000000
00000000000000000000000000000000
00000000000001111000000000000111
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
101010111001101110111101110011000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
01100100000000000000000000000000

```

0:00:05 conectado Autodetect. 115200 8-N-1 DESPLAZAR MAY NUM Capturar Imprimir

La siguiente comprobación, pretende demostrar que el periférico tiene propiedades correctoras, para lo cual se ha diseñado una prueba en la que se altera de manera manual un bit de dato en uno de los registros de la memoria RAM. Si funciona correctamente, el periférico deberá devolver el dato correcto, y no el dato modificado, ya que la codificación SECDED del periférico permite corregir un fallo o detectar dos.

70

Para la prueba se ha modificado el dato que aparece sombreado en rojo, pasando de un 0 a un 1. Al no haber recalculado los bits de paridad para esa modificación, el periférico se encontrará con que el bit de paridad P5 devuelve un '1', cuando para la configuración actual, El dato correcto sería un '0'. Al detectar esa diferencia, la señal syndrome tomará un valor determinado, en concreto para el D31 (corresponde al bit trigésimo octavo) será "100110", indicando así en que bit está el fallo y alternando su valor para a la salida del componente tener el dato corregido.

```
type ram_type is array (0 to 15) of std_logic_vector (38 downto 0);
-- 2D Array Declaration for RAM signal

signal ram_mem : ram_type --;
--
PBD31D30D29D28D27D26P5D25D24D23D22D21D20D19D18D17D16D15D14D13D12
D11P4D10D9D8D7D6D5D4P3D3D2D1P2D0P1P0

:= ("00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "00000000000000000000000000000000000000000000000000000",
    "01100110000000000000000000000000000000000000000000000"
);
```

Si el periférico hace bien su función, devolverá "01100100000000000000000000000000". Por el contrario, si no aplicase la corrección de errores, el resultado devuelto sería "11100100000000000000000000000000".

Con el propósito de facilitar la visualización de la corrección de errores, se ha añadido una señal (err_en) controlada por uno de los interruptores de la placa. Esta señal permite que se habiliten o no la corrección de errores a la hora de rescatar el dato de la memoria RAM para enviarlo por la señal de salida dout. Gracias a esta funcionalidad añadida, será posible visualizar en la consola Hyperterminal la diferencia entre el dato con errores (sin corregir) y cuando este es corregido. Para implementar esta característica, ha sido necesario modificar el código del proceso de la salida de datos del periférico. Se ha decidido implementarlo en la última fase de funcionamiento del periférico ya que nos permite mantener el código de error, mostrando

claramente el estado del dato que estamos manejando y cuál es su corrección. El código modificado es el siguiente.

```

COR_ERROR: process (rd_data, dec_syndrome, switch_err)
begin
  if (switch_err = '1') then
    -- XOR data from memory with dec_syndrome
    corrected_data <= rd_data xor dec_syndrome;
    --dout <= corrected_data;
  else
    corrected_data <= rd_data;
  end if;
end process COR_ERROR;

```

Utilizando el interruptor habilitado para ello, se puede activar o desactivar la corrección de errores. Para el caso en el que la corrección está desactivada, vemos como devuelve el dato incorrecto (segundo dato). En la segunda línea, con la corrección ya activada, vemos como devuelve el dato como se ha indicado en el primer caso.

Para comprobarlo, se ha introducido en la memoria de programa el siguiente código, donde se rescatará el mismo dato en ambas ejecuciones, pero activando el interruptor antes de la segunda lectura.

INPUT	rxreg, 0000005F ;obtengo un dato
LOAD	aux, rxreg; cargo el dato en aux
CALL	show ;lo muestro
CALL	recibe
INPUT	rxreg, 0000005F ;obtengo un dato
LOAD	aux, rxreg; cargo el dato en aux
CALL	show ;lo muestro
CALL	recibe

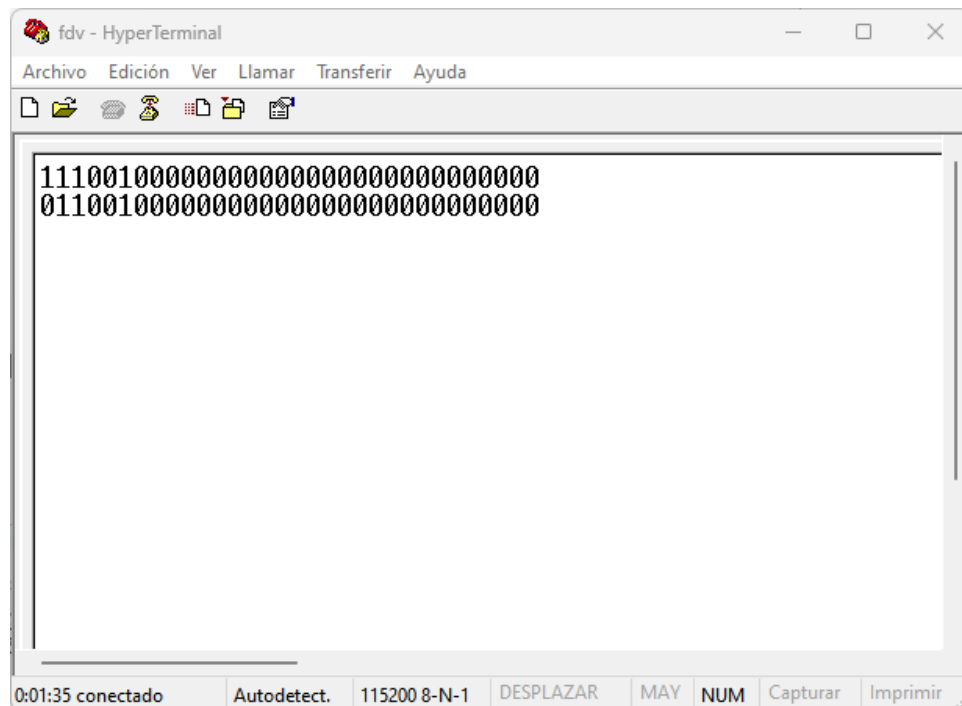


Figura 39. Ventana del programa Hyperterminal que muestra la diferencia entre la corrección de errores desactivada y activada

Como se puede ver, los datos devueltos por el microcontrolador son correctos, por lo que se verifica que esta nueva funcionalidad está operativa.

Finalmente, tal y como se ha indicado en el apartado anterior, se implementó un proceso para poder visualizar los códigos de error en los leds integrados en la FPGA. A continuación, se encuentran las imágenes que muestran cómo se visualiza cada código de error en la FPGA:

- Código de error "00": Este primer estado indica que no se han encontrado errores en la posición de memoria a la que se está apuntando en este momento.

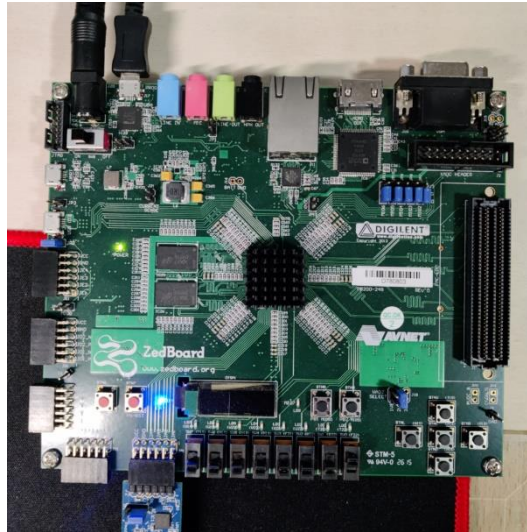


Figura 40. Imagen de la FPGA mostrando el código de error "00"

- Código de error "01": Este código es el primero que indica que existe algún tipo de error en la posición de memoria a la que se está apuntando en el momento de su visualización. Significa que el algoritmo de corrección de errores ha detectado un error bien en alguno de los bits de datos de la memoria (y será corregido) o bien en alguno de los bits de paridad de esta (no necesitará ser corregido).

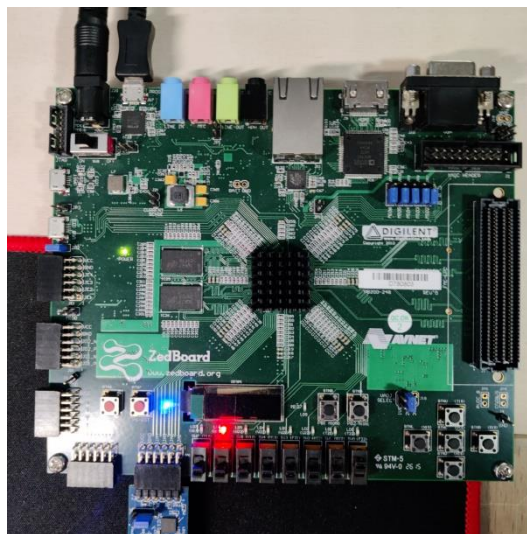


Figura 41. Imagen de la FPGA mostrando el código de error "01"

- Código de error “10”: De nuevo, este código de error indica que existen errores con el dato al que se apunta en este instante. En este caso, la gravedad del error sube de categoría, ya que indica que se ha detectado un error doble. Para este tipo de error, el algoritmo SECEDED es capaz de indicar que existe un error, pero no será capaz de corregirlo. Para este caso, mediante las nuevas funcionalidades añadidas al periférico y a la memoria de programa (interruptor switch_err y rutina show), se podrá aislar y detectar que tipo de bits son los afectados.

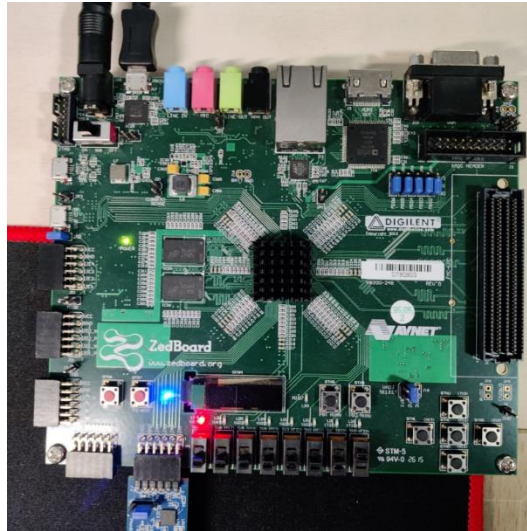


Figura 42. Imagen de la FPGA mostrando el código de error "10"

- Código de error “11”: Este último código afecta al bit de paridad general (OP). Se producirá siempre que en el dato al que se esté apuntando tenga el bit más significativo de los almacenados en la memoria (posición 39) erróneo. Se trata de un error corregible, aunque no será necesario aplicarle ningún tipo de cambio.

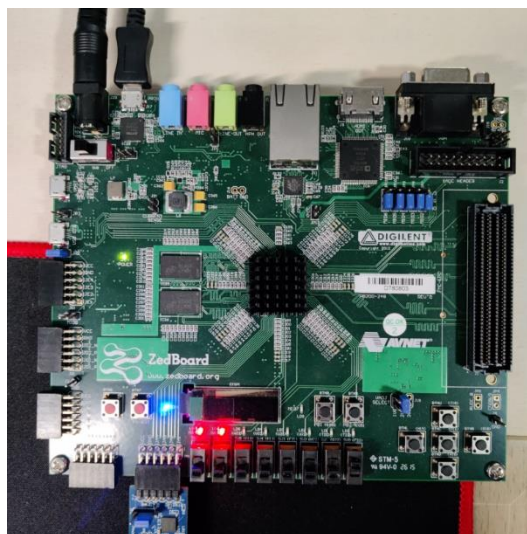


Figura 43. Imagen de la FPGA mostrando el código de error "11"

6. Conclusiones y líneas futuras

El objetivo principal de este proyecto era conseguir migrar el microcontrolador Picoblaze de 8 bits a 32 bits y verificar su correcto funcionamiento. Durante este proceso se ha hecho una descripción esquemática y analítica del diseño de 8 bits y como este ha sido migrado al nuevo diseño. En el desarrollo de la migración ha sido necesario verificar el correcto funcionamiento del microcontrolador debido a las numerosas modificaciones realizadas para lo que ha sido necesario diseñar una batería de pruebas que nos han permitido verificar el correcto funcionamiento de las instrucciones presentes originalmente en el microcontrolador, así como las nuevas que se han implementado.

Así mismo, el otro objetivo era implementar un periférico diseñado en VHDL con una pequeña memoria con un mecanismo de corrección de errores de tipo SECDED. Para ello, se ha analizado la estructura del componente y su diseño, explicando su funcionamiento y cómo funciona la migración de 8 a 32 bits, pretendiendo aprovechar las capacidades añadidas al microcontrolador. Se ha analizado su correcto funcionamiento con una batería de pruebas, pretendiendo demostrar que se conecta correctamente al microcontrolador y el nuevo diseño implantado es válido. Así mismo, se han añadido algunas nuevas funcionalidades al periférico que permiten observar su correcto funcionamiento y facilitan su comprensión. Esto añade un valor extra al periférico implementado, ya que no solo se permite operar con él, sino que además las nuevas características añadidas permiten ver y comprender su funcionamiento, siendo útil como herramienta académica a la hora de estudiar y desarrollar la implementación de circuitos digitales sobre una FPGA.

Ambos objetivos han sido completados, pues en ambos casos se han superado todas las pruebas propuestas que pretendían mostrar el correcto funcionamiento del sistema que conforman microcontrolador y periférico, mostrando así un método de cómo se pueden ampliar las capacidades de un microcontrolador así como añadir nuevas funciones que permiten comprender el funcionamiento interno de uno de estos dispositivos así como comprender e interactuar con el mecanismo de control y corrección de errores SECDED.

Este nuevo sistema, ofrece también nuevas posibilidades. Dado que estamos ante un sistema de hardware que trabaja con datos de 32 bits, se abre un gran abanico de posibles implementaciones que permitirían aprovechar la nueva arquitectura como base para crear sistemas o componentes más sofisticados.

Uno de las posibles futuras aplicaciones de este sistema es la posibilidad de realizar operaciones en coma flotante, ya que un registro de 32 bits permitiría trabajar con este tipo de operaciones de manera eficaz. Esto permitiría realizar operaciones aritméticas con una precisión muy elevada, lo que ofrece una gran cantidad de posibles aplicaciones prácticas, lo que combinado con el periférico que se ha implementado, permite almacenar estos datos y conservarlos de manera segura en entornos donde el fenómeno trastorno de un solo evento (SEU) pueda ser un problema, provocado por el impacto de partículas de las radiaciones electromagnéticas presentes en el espacio exterior.

Otra posible aplicación que pasa a ser factible, es realizar una transformada de Fourier, en el campo del tratamiento y análisis de señales continuas. Esta posible implementación, habilitaría al microcontrolador a realizar filtrados de señales de una gran precisión en el dominio de la frecuencia, lo que podría usarse como base para crear componentes que realicen esta

operación. Asimismo, su uso podría facilitar la comprensión de cómo se realiza esta compleja operación a personas que estén realizando algún tipo de formación académica relacionada con ello.

El desarrollo de este proyecto ha estado enfocado también a la formación académica de alumnos que deseen ahondar en el área del desarrollo de sistemas digitales basados en VHDL, estando preparado para implementar un gran número de nuevos periféricos, así como para ampliar las funcionalidades del microcontrolador añadiendo nuevas instrucciones (tras el desarrollo, han quedado 36 códigos de instrucción sin usar que pueden ser aprovechados por cualquiera que quiera ampliar las funciones de este microcontrolador).

Se convierte así en una herramienta de formación preparada para continuar su desarrollo y con medios para comprobar el correcto funcionamiento de las nuevas implementaciones que se introduzcan por parte de cualquier usuario, así como una herramienta para el desarrollo académico de aquellos interesados en esta área de conocimiento.

Referencias

- [1]
Xilinx, «PicoBlaze 8-bit Embedded Microcontroller User Guide,» 22 Junio 2011. [En línea].
Available: https://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf.
- [2]
Xilinx, «Zynq-7000 SoC Technical Reference Manual,» 2 Abril 2021. [En línea].
Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.
- [3]
Diligent, «ZedBoard - Hardware User's Guide,» 27 Enero 2014. [En línea].
Available: https://files.digilent.com/resources/programmable-logic/zedboard/ZedBoard_HW_UG_v2_2.pdf.
- [4]
K. Champman, «PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IE Devices,» 4 Febrero 2003. [En línea].
Available: https://www.xilinx.com/support/documentation/application_notes/xapp213.pdf.
- [5]
Microsoft, «C and C++ Integer Limits,» 8 Marzo 2021. [En línea].
Available: <https://docs.microsoft.com/en-us/cpp/c-language/cpp-integer-limits?view=msvc-170>.

Índice de figuras

Figura 1. Módulo PMOD. Fuente: Mouser.....	6
Figura 2. FPGA Zedboard. Fuente: Avnet.....	6
Figura 3. Diagrama del PicoBlaze.....	6
Figura 4. Esquema de una palabra de programa del PicoBlaze a 8 bits. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices	7
Figura 5. Grupo de instrucciones de control de programa. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices.....	8
Figura 6. Grupo de instrucciones de desplazamiento y rotación de bits. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices	9
Figura 7. Grupo de instrucciones de operaciones lógicas. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices.....	9
Figura 8. Grupo de instrucciones de operaciones aritméticas. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices.....	10
Figura 9. Grupo de instrucciones de entrada y salida de datos. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices.....	10
Figura 10. Grupo de instrucciones encargadas de las interrupciones. Fuente: PicoBlaze 8-Bit Microcontroller for CPLD Devices	11
Figura 11. Formato de la palabra de programa del PicoBlaze a 32 bits.....	21
Figura 12. Timing report.....	29
Figura 13. Utilización de recursos de la FPGA (formato tabla).....	30
Figura 14. Utilización de recursos de la FPGA (formato gráfica).....	30
Figura 15. Ventana del programa Hyperterminal con el resultado del test de la operación AND	43
Figura 16. Ventana del programa Hyperterminal con el resultado del test de la operación OR	44
Figura 17. Ventana del programa Hyperterminal con el resultado del test de la operación XOR	44
Figura 18. Ventana del programa Hyperterminal con el resultado del test de la operación ADD	45
Figura 19. Ventana del programa Hyperterminal con el resultado del test de la operación SUB	46
Figura 20. Ventana del programa Hyperterminal con el resultado del test de la operación ADDCY	47
Figura 21. Ventana del programa Hyperterminal con el resultado del test de la operación SUBCY	48
Figura 22. Ventana del programa Hyperterminal con el resultado del test de la operación SHIFTRS232 ..	49
Figura 23. Ventana del programa Hyperterminal con el resultado del test de la operación RR	49
Figura 24. Ventana del programa Hyperterminal con el resultado del test de la operación RL.....	50
Figura 25. Ventana del programa Hyperterminal con el resultado del test de la operación SR0.....	51
Figura 26. Ventana del programa Hyperterminal con el resultado del test de la operación SR1.....	52
Figura 27. Ventana del programa Hyperterminal con el resultado del test de la operación SLO	52
Figura 28. Ventana del programa Hyperterminal con el resultado del test de la operación SL1	53
Figura 29. Ventana del programa Hyperterminal con el resultado del test de la operación SRX.....	54
Figura 30. Ventana del programa Hyperterminal con el resultado del test de la operación SLX	55
Figura 31. Ventana del programa Hyperterminal con el resultado del test de la operación SRA	56
Figura 32. Ventana del programa Hyperterminal con el resultado del test de la operación SLA	57
Figura 33. Ventana del programa Hyperterminal con el resultado del test de las operaciones INPUT y OUTPUT	58
Figura 34. Ventana del programa Hyperterminal con el resultado del test de la operación ENABLE, DISABLE y RETURN.....	59
Figura 35. Ventana del programa Hyperterminal con el resultado del test de la operación FLIP	60
Figura 36. Diagrama del funcionamiento del módulo SECDED a 8 bits.....	61
Figura 37. Diagrama de funcionamiento del módulo SECDED a 32 bits.....	62
Figura 38. Ventana del programa Hyperterminal con la lectura de todas las posiciones de memoria	70
Figura 39. Ventana del programa Hyperterminal que muestra la diferencia entre la corrección de errores desactivada y activada	72

<i>Figura 40. Imagen de la FPGA mostrando el código de error "00"</i>	<i>73</i>
<i>Figura 41. Imagen de la FPGA mostrando el código de error "01"</i>	<i>73</i>
<i>Figura 42. Imagen de la FPGA mostrando el código de error "10"</i>	<i>74</i>
<i>Figura 43. Imagen de la FPGA mostrando el código de error "11"</i>	<i>74</i>

Todas las imágenes son de elaboración propia excepto en aquellas en las que se indique su fuente.

Índice de tablas

<i>Tabla 1. Conjunto de instrucciones originales del PicoBlaze</i>	<i>7</i>
<i>Tabla 2. Codificación SECDED de los bits de paridad.....</i>	<i>64</i>

Todas las tablas son de elaboración propia excepto en aquellas en las que se indique su fuente.