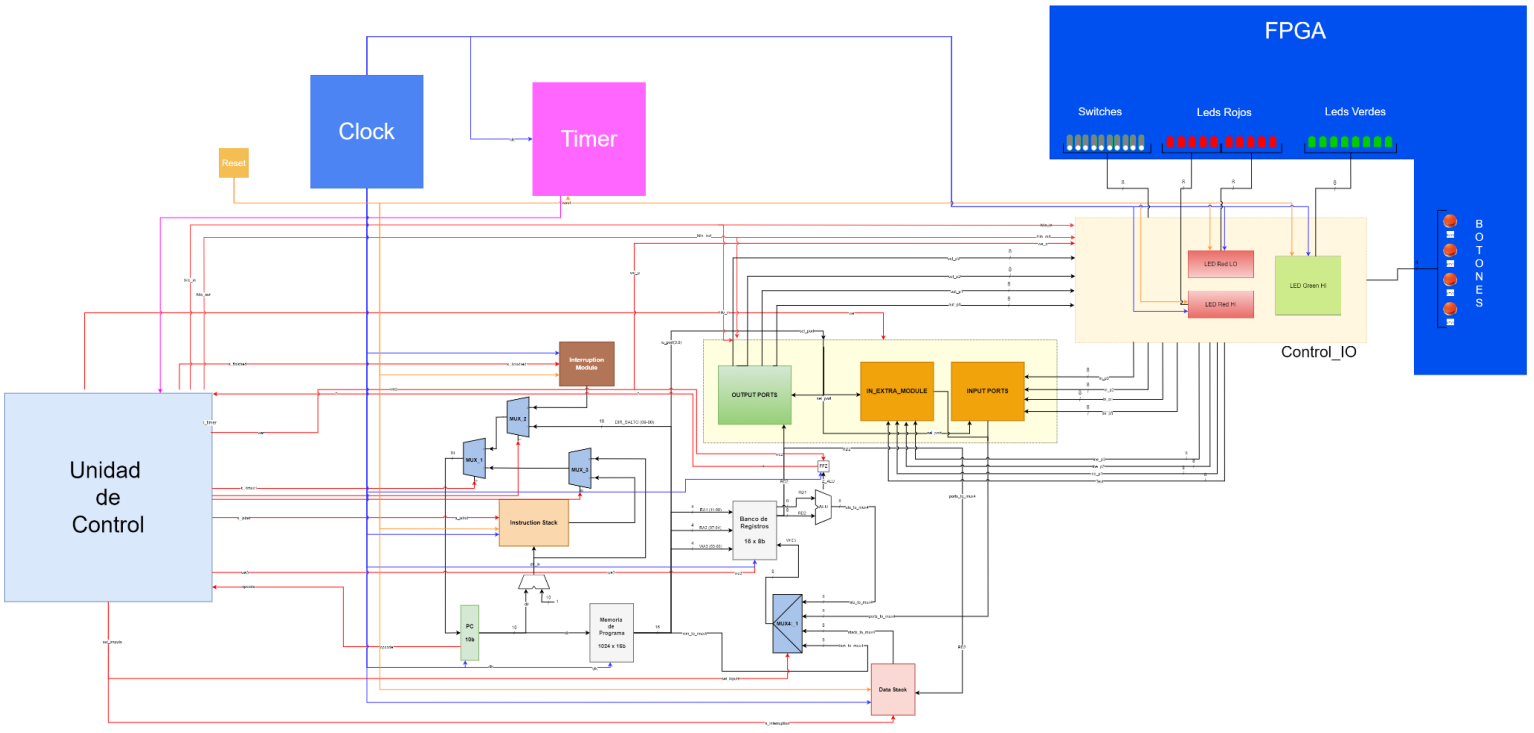
Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Proyecto CPU Monociclo

Diseño de Procesadores



Ángel Julián Bolaño Campos

Índice:

1. Introducción.

En este informe se dará detalle del proceso de diseño de una CPU monociclo. Comenzaremos por explicar algunas decisiones tomadas para la implementación de algunos de los componentes de la CPU como por ejemplo la unidad de control, puertos y pila.

En este diseño no se contempla la ampliación de 16 a 32 bits

2. Decisiones de diseño tomadas.

En un primer momento se tomaron decisiones de diseño que hubo que cambiar a medida que el proyecto iba madurando.

2.1. Unidad de Control.

Después de algunos cambios al principio, se tomó la decisión de programar la Unidad de Control para que ésta fuese fácil de modificar si había que añadir señales e instrucciones.

```
reg [14:0] signals;

parameter ARITH   = 15'b100001100000000;
parameter LOADINM = 15'b100111000000000;
parameter JUMP    = 15'b010000000000000;
parameter NOJUMP  = 15'b110000000000000;
parameter IN      = 15'b100011000000000;
parameter OUT     = 15'b100000010000010;
parameter NOP     = 15'b000000000000000;
parameter JAL     = 15'b010000001000000;
parameter RET     = 15'b101000001100000;
parameter PUSH    = 15'b100000000010000;
parameter POP     = 15'b100101000011000;
parameter INTERR  = 15'b000000001000000;
parameter FNSH    = 15'b101000001100100;
parameter OUTPUTR = 15'b100011000000001;
```

```
assign {
    s_mux1,         //1
    s_mux2,         //2
    s_mux3,         //3
    sel_inputs[1],  //4
    sel_inputs[0],  //5
    we3,            //6
    wez,            //7
    we_port,        //8
    we_istack,      //9
    s_jret,         //10
    we_dstack,      //11
    s_ppop,         //12
    s_finish_interr, //13
    we_o,           //14
    s_special_port  //15

} = signals;
```

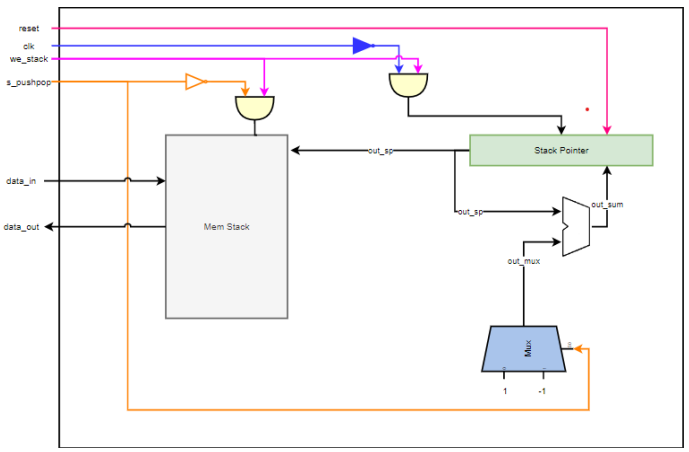
De esta forma resultó muy eficiente añadir o modificar señales a la Unidad de Control. Luego el switch case se ocupaba de la asignación dependiendo del opcode.

```
always @(*) begin
    if (i_timer & ~s_interruption) begin
        signals = INTERR;
        op_alu = 3'bx;
    end
    else begin
        op_alu = opcode[4:2];
        casez (opcode)
            6'b0?????: // operacion aritmetica
                signals = ARITH;
    . . . . . .
    . . . . . .
                signals = JAL;
            6'b101010:
                signals = RET;
            6'b101011:
                signals = PUSH;
            6'b101100:
                signals = POP;
            6'b101110:
                signals = FNSH;
            6'b101111:
                signals = OUTPUTR;
            default:
                signals = NOP;
        endcase
    end
end
```

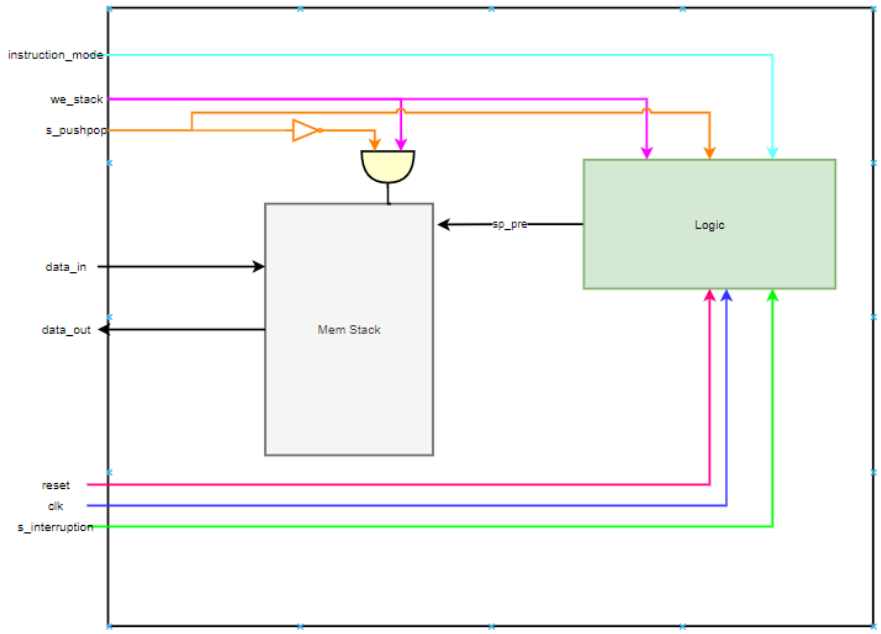
2.2. Pila de instrucciones y datos.

En un primer diseño la pila utilizaba un registro para almacenar el StackPointer. Esto tuvo que cambiarse cuando se simuló en la FPGA ya que en un primer momento la pila guardaba el valor en el flanco de subida y en el flanco de bajada aumentaba el StackPointer. La simulación con GTKWave funcionaba perfectamente pero los tiempos en “la realidad” eran muy cortos y esto daba problemas.

2.2.1 Primer diseño de pila



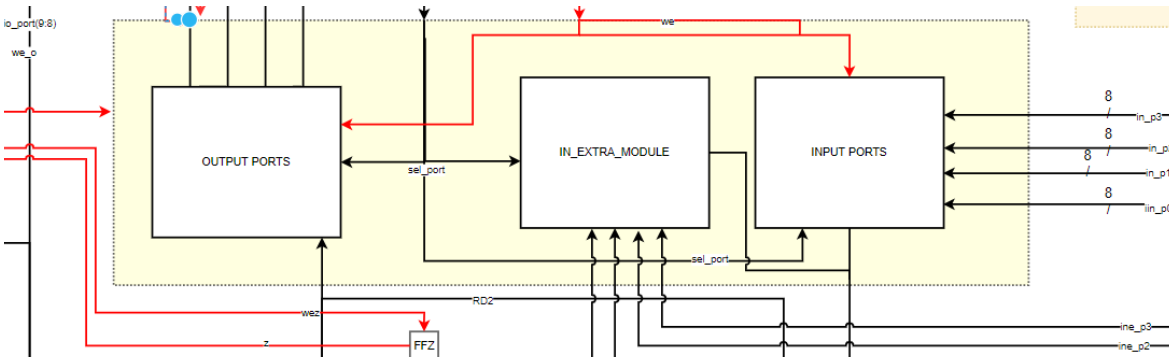
2.2.2 Segundo diseño de pila



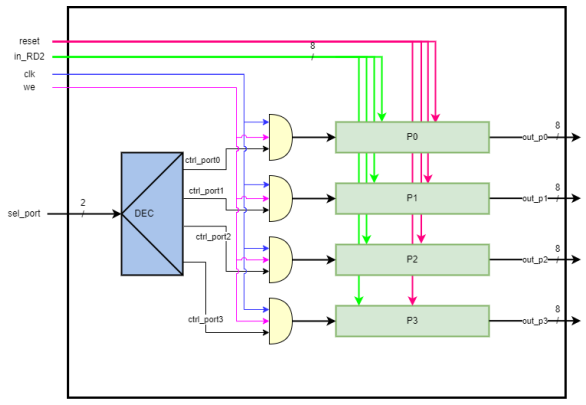
2.3. Puertos de I/O.

Para el tema de los puertos ocurrió algo similar pero relacionado con los registros. Los registros, que se implementaron para intentar mantener los datos por si en algún momento fuese necesario, no estaban respondiendo bien en el momento de solicitar o recibir los datos. En el GTKWave si funcionaba pero en la placa no. Por lo que se optó por sacar los registros para que los datos entren o salgan en el mismo ciclo.

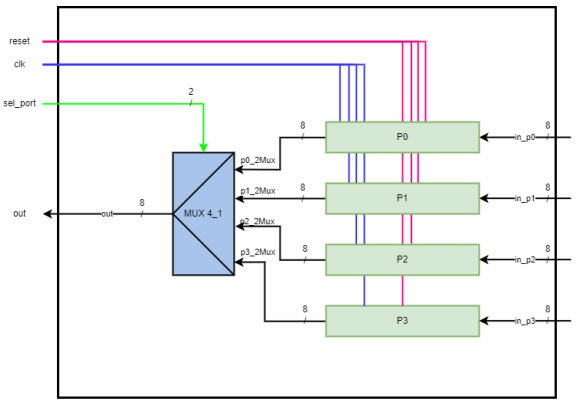
Se añadió un puerto extra con una peculiaridad, se trata de una instancia del puerto de entrada al que conectamos el registro de los leds verdes. Esto está pensado para poder leer el estado de los puertos de salida.



2.3.1 Primer diseño de entrada/salida

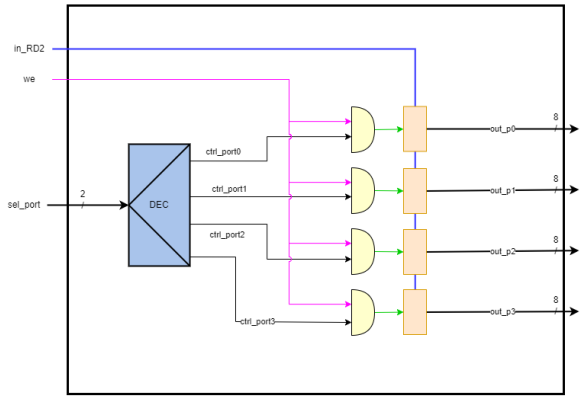


Output

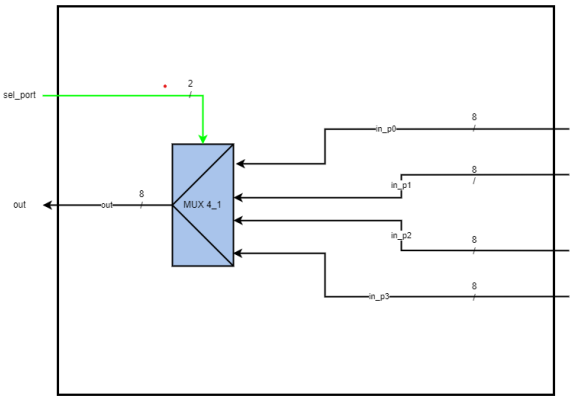


Input

2.3.2 Segundo diseño de entrada/salida



Output



Input

2.3.4 Código

```
module output_module ( input wire we, input wire [1:0] sel_port, input wire [7:0] in_RD2,
                       output wire [7:0] out_from_p0, out_from_p1, out_from_p2, out_from_p3);

    wire [3:0] ctrl_port;

    deco #(4) DECO_FOR_OUTPUTS(sel_port, ctrl_port);

    assign out_from_p0 = ctrl_port[0] & we ? in_RD2 : 8'b0;
    assign out_from_p1 = ctrl_port[1] & we ? in_RD2 : 8'b0;
    assign out_from_p2 = ctrl_port[2] & we ? in_RD2 : 8'b0;
    assign out_from_p3 = ctrl_port[3] & we ? in_RD2 : 8'b0;

endmodule
```

```
module input_module ( input wire clk, reset, input wire [1:0] sel_port,
                      input wire [7:0] in_p0, in_p1, in_p2, in_p3, output wire [7:0] out);

    mux4 MUX_PORT(in_p0, in_p1, in_p2, in_p3, sel_port, out);

endmodule
```

2.4. Módulo de interrupciones.

El módulo de interrupciones se pensó para poder atender y gestionar diferentes tipos de interrupciones y prioridades pero al solo estar usando el Timer como interrupción se simplificó el código. Su tarea es cargar la dirección en el PC dependiendo del tipo de interrupción que reciba. Como sólo estamos trabajando con una interrupción se eliminó el resto de direcciones.

```
module interruption_module (  input wire clk, reset, i_timer, s_finished,
                              output wire[9:0] dir_out, output wire s_interruption);

//Direcciones de subrutina
parameter dir_timer = 10'b1000000000;

wire in, we;

assign in = s_finished ? 1'b0 : i_timer | s_interruption;
assign we = i_timer | s_finished;

ffd ffinterruption(clk, reset, in, we, s_interruption);

assign dir_out = i_timer & ~s_finished ? dir_timer : 10'bx;

endmodule
```

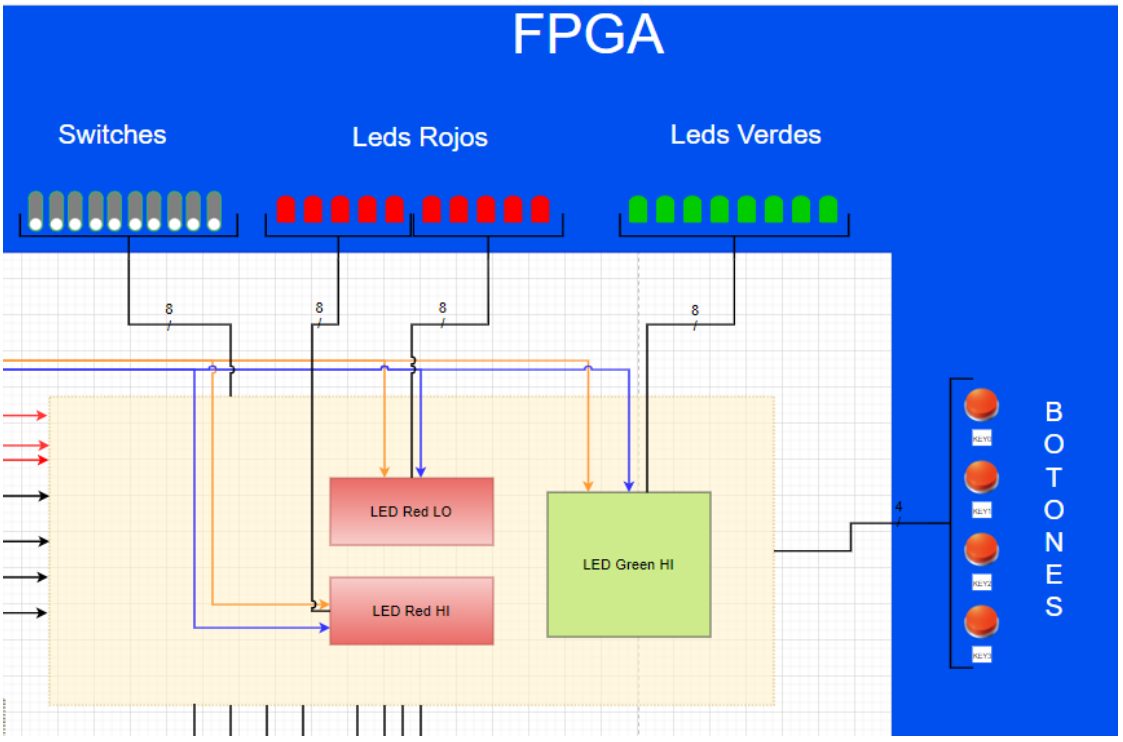
2.5. Control_IO.

En el momento de llevar el diseño a la placa nos encontramos con el obstáculo de que los switches son de 10 bits y los puertos que tenemos son de 8 bits. Para suplir esto, los puertos se conectan a los 5 primeros switches al puerto de entrada 1 y los 5 siguientes al puerto de entrada 2 (la idea era que fuesen 5 y 5 pero finalmente se dejó 5 y 4 para dejar un switch de reset).

Los puertos de entrada quedan asignados de la siguiente manera:

- in_p0 = Botones 0, 1, 2 y 3, el resto de bits a cero.
- in_p1 = Parte baja de los switch [4:0]
- in_p2 = Parte alta de los switch [5:8]
- in_p3 No se usa

En los puertos de salida usamos 3 registros, 2 de ellos para albergar los dos pares de 5 bits de las luces rojas de los switch y uno para los leds verdes.



2.5. Componentes.

Se añadieron algunos componentes básicos como un decodificador, multiplexores de 4 a 1, una memoria para la pila, un registro con WE y algún otro de menor importancia.

3. Codificación de instrucciones.

Para la codificación de instrucciones se decidió usar los siguientes nemónicos:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	0	0	0	0	R1	R1	R1	R1	N	N	N	N	Rd	Rd	Rd	Rd
NOT	0	0	0	1	R1	R1	R1	R1	N	N	N	N	Rd	Rd	Rd	Rd
ADD	0	0	1	0	R1	R1	R1	R1	R2	R2	R2	R2	Rd	Rd	Rd	Rd
SUB	0	0	1	1	R1	R1	R1	R1	R2	R2	R2	R2	Rd	Rd	Rd	Rd
AND	0	1	0	0	R1	R1	R1	R1	R2	R2	R2	R2	Rd	Rd	Rd	Rd
OR	0	1	0	1	R1	R1	R1	R1	R2	R2	R2	R2	Rd	Rd	Rd	Rd
NEG1	0	1	1	0	R1	R1	R1	R1	N	N	N	N	Rd	Rd	Rd	Rd
NEG2	0	1	1	1	N	N	N	N	R2	R2	R2	R2	Rd	Rd	Rd	Rd
LI	1	0	0	0	C	C	C	C	C	C	C	C	Rd	Rd	Rd	Rd
BEZ	1	0	0	1	0	0	D	D	D	D	D	D	D	D	D	D
BNZ	1	0	0	1	0	1	D	D	D	D	D	D	D	D	D	D
JUMP	1	0	0	1	1	0	D	D	D	D	D	D	D	D	D	D
INPUT	1	0	0	1	1	1	P	P	N	N	HL	HL	Rd	Rd	Rd	Rd
OUTPUT	1	0	1	0	0	0	P	P	Rd	Rd	Rd	Rd	N	N	HL	HL
JAL	1	0	1	0	0	1	D	D	D	D	D	D	D	D	D	D
RET	1	0	1	0	1	0	N	N	N	N	N	N	N	N	N	N
PUSH	1	0	1	0	1	1	N	N	Rd	Rd	Rd	Rd	N	N	N	N
POP	1	0	1	1	0	0	N	N	N	N	N	N	Rd	Rd	Rd	Rd
FNSH	1	0	1	1	1	0	N	N	N	N	N	N	N	N	N	N
OUTPUTR	1	0	1	1	1	1	P	P	N	N	HL	HL	Rd	Rd	Rd	Rd

Como se puede observar en la codificación, la mayoría de las instrucciones son comunes. Pasaremos pues a explicar las instrucciones INPUT, OUTPUT, OUTPUTR, FNSH, JAL, RET, PUSH y POP.

3.1. Input, Output y OutputR.

La forma de utilización input/ouput serían las que podríamos haber llamado read/write. Como se implementaron cuatro puertos se debe seleccionar el puerto y si el valor leído será de la parte alta o baja.

Por ejemplo, si queremos leer el puerto 1 para meter el valor en un registro deberemos recordar que tenemos asignado el in_p1 para la parte baja (LO) y el in_p2 para la parte alta (HI) de los switches. Usamos 1 para la parte baja, 2 para la parte alta y 3 para el registro de 8 bits completo.

Sintaxis de INPUT

- INPUT 1 1 R2 → Lee lo que está en el puerto 1, establece que será la parte baja de los 10 switches y lo mete en R2.
- INPUT 2 2 R3 → Lee lo que está en el puerto 2, establece que será la parte alta de los 10 switches y lo mete en R3.
- INPUT 0 3 R1 → Lee los 8 bits del puerto 3 y lo pone en R1.

Sintaxis de OUTPUT

- OUTPUT 0 R0 3 → Pone el los 8 bits del registro cero en el puerto cero.
- OUTPUT 1 R0 1 → Pone los primeros 5 bits del registro R0 en la parte baja de los bits que se conectan a los switches
- OUTPUT 2 R0 2 → Pone los 4 primeros bits del registro R0 en la parte alta baja de los bits que se conectan a los switches (Recordar que solo 4 porque el último esta reservado para el reset)

OUTPUTR	1	0	1	1	1	1	P	P	N	N	HL	HL	Rd	Rd	Rd	Rd

La sintaxis de esta instrucción es la misma que la del INPUT, solo que realmente lo que hacemos es saber el estado de los puertos de salida, por eso se llama OUTPUTR de Output read.

- OUTPUTR 0 3 R5 → Coloca en R5 todos los bits del puerto 0.

3.2. FNSH.

Esta instrucción se utiliza para delimitar el final de una subrutina. Su principal función es que cuando se entra en una subrutina ésta no pueda ser interrumpida hasta que dicha subrutina pase por la instrucción “FNSH”. En este código sólo se contemplan las interrupciones del timer.

```
ledsv_off:
OUTPUT 0 R0 3
FNSH
```

3.3. JAL, RET, PUSH Y POP.

Todas estas instrucciones trabajan de la misma forma solo que existen dos pilas. JAL y RET utilizan una pila de instrucciones y PUSH y POP utilizan una pila aparte de datos.

La sintaxis de JAL y RET

- JAL subrutina → Salta a subrutina guardando la dirección actual para luego volver
- RET → Volverá a la última posición guardada por JAL en la pila.

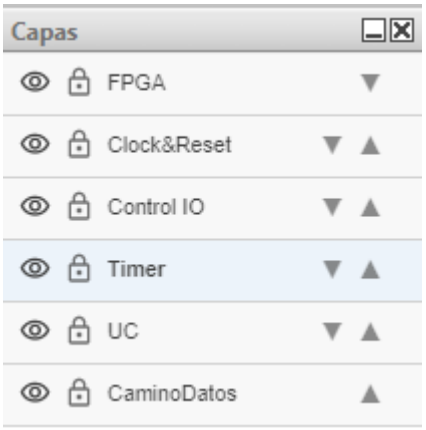
La sintaxis de PUSH y POP

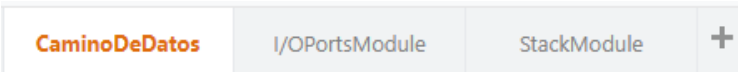
- PUSH R3 → Carga en la pila el contenido de R3
- POP R2 → Mete en R2 el último valor introducido en la pila.

4. Diagrama de Camino de Datos .

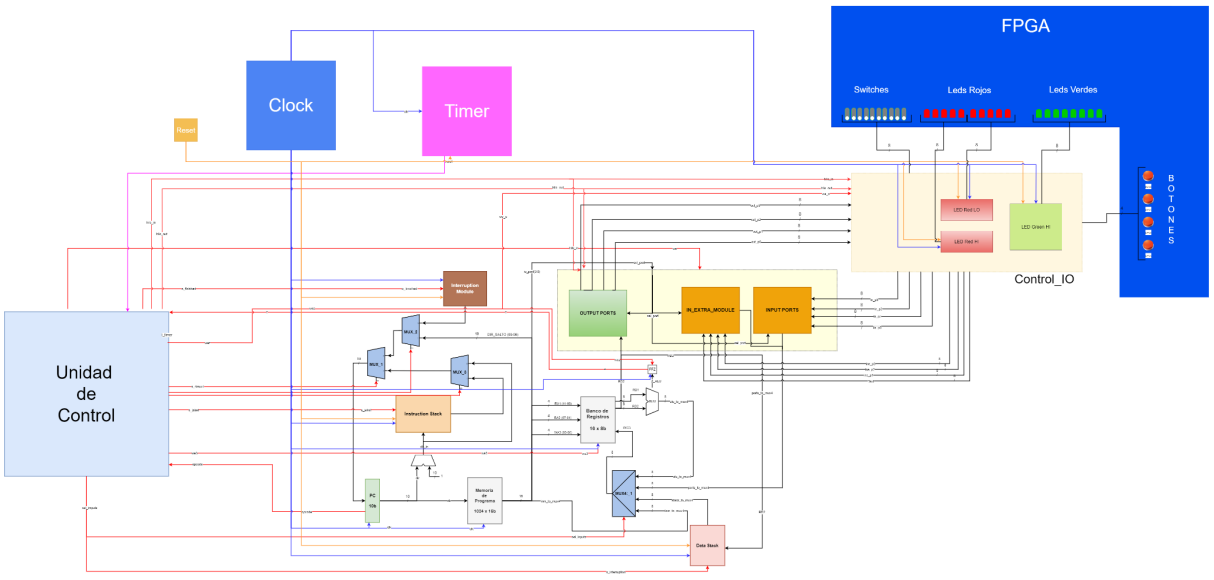
Dada la complejidad del diagrama, para poder observar mejor el diseño se deja aqui el enlace a la plataforma donde se creó. No obstante se incorpora una imagen a continuación. El enlace está disponible para lectura de correos institucionales de la Universidad de La Laguna. Si no fuese posible acceder, hágamelo saber para buscar otra solución.

El diseño esta implementado en capas, por lo que el profesorado podrá activar o desactivar las capas que quiera visualizar. Además, en la parte inferior del proyecto, hay pestañas donde se encuentran los otros módulos creados.





4.1. Diagrama.



5. Utilidades implementadas.

Lo más destacable para mencionar es la creación de un pequeño programa ensamblador para poder crear el programa. Se intentó que el programa sea lo más versátil posible haciendo que, para crear el juego de instrucciones que después deberá interpretar, lea las “normas” desde un fichero de texto plano con una sintaxis concreta.

De esta forma se evita tener que compilar el programa por cada nuevo cambio. Las normas a seguir son las siguientes:

- Los primeros dígitos especifican el tamaño de los registros, por ejemplo en la instrucción MOV hay 4 registros de 4 bits
- Deben ser 4 registros siempre aunque se tengan que poner a 0 para completar los 16 bits
- El siguiente parametro declara el nombre de la instrucción.
    - Deben ser nombres en MAYÚSCULAS
- El último parámetro especifica el OPCODE

Así pues un fichero de texto con la siguiente codificación nos ayudará a crear el repertorio de instrucciones.

4	4	4	4	MOV	0000
4	4	4	4	NOT	0001
4	4	4	4	ADD	0010
4	4	4	4	SUB	0011
4	4	4	4	AND	0100
4	4	4	4	OR	0101
4	4	4	4	NEG1	0110
4	4	4	4	NEG2	0111
4	8	0	4	LI	1000
6	10	0	0	BEZ	100100
6	10	0	0	BNZ	100101
6	10	0	0	JUMP	100110
6	2	4	4	INPUT	100111

6	2	4	4	OUTPUT	101000
6	10	0	0	JAL	101001
16	0	0	0	RET	101010
6	2	4	4	PUSH	101011
6	2	4	4	POP	101100
16	0	0	0	FNSH	101110
6	2	4	4	OUTPUTR	101111

6. Implementación de programa.

El programa consta de dos partes. La primera es la parte del código que podríamos entender como “main” y a partir de la linea 512 de la memoria estarán las “funciones” que se activan con el pulso del timer.

6.1. Main.

En esta primera parte empezamos cargando unos valores previos en registros para tener valores con los que realizar operaciones y poder determinar la secuencia del programa.

```
LI 1 R2  # Registro para comprobar boton0
LI 2 R4  # Registro para comprobar boton1
LI 4 R5  # Registro para comprobar boton2
LI 8 R6  # Registro para comprobar boton3
LI 0 R11 # Contador para lectura

# Comprobamos los valores de P0 y vamos restando sus valores dependiendo de la posición
# que tienen en el puerto --> 00000001|00000010|00000100|00001000 guardamos valores en
los # registros R14 y R15

inicio:
INPUT 0 3 R1  # Se guarda lo que haya entrado por los botones en R1
SUB R1 R2 R3  # Se resta 1
BEZ boton0    # Si es 0 vamos al boton0

SUB R1 R4 R3  # Se resta 2
BEZ boton1    # Si es 0 vamos al boton1

SUB R1 R5 R3   # Se resta 4
BEZ boton2     # Si es 0 vamos al boton2

SUB R1 R6 R3   # Se resta 8
BEZ boton3     # Si es 0 vamos al boton3
```

```
JUMP inicio

boton0:      # Modo lectura
LI 1 R15     # Cargamos un 1 en el R15 (será como un flag)
LI 0 R14     # R14 lo usamos para saber si toca guardar, debe actualizar el valor
JUMP inicio

boton1:     # Guardar
LI 1 R14    # Cargamos un 1 en el R14
JUMP inicio

boton2:     # Apagar
LI 2 R15    # Cargamos un 2 en el R15 (será como un flag)
LI 0 R14    # R14 lo usamos para saber si toca guardar, debe actualizar el valor
JUMP inicio

boton3:     # Mostrar
LI 8 R15    # Cargamos un 8 en el R15 (será como un flag)
LI 0 R14    # R14 lo usamos para saber si toca guardar, debe actualizar el valor
JUMP inicio
```

6.2. Función.

A partir de la posición 512 se encuentra la subrutina que activa el timer cada medio segundo.

```
# R12 - Registro que enciende leds verdes

#Comprobación de botones
SUB R15 R2 R3       # Si al restar 1 da 0 significa que el boton de lectura de los
switches fue activado
BEZ lectura         # Vamos a "lectura" si la resta dio 0
SUB R15 R4 R3       # Restamos 2 a R15
BEZ apagar_todo     # Si 0 entonces vamos a apagar_todo
SUB R15 R6 R3       # Restamos un 8 a R15
BEZ mostrar         # Si 0 entonces mostramos
FNSH

lectura:
SUB R14 R2 R3         # Restamos 1 a R14
BNZ ctrl_switches     # Si no es 0 vamos a ctrl_switches

ADD R11 R2 R11      # Incrementamos contador
INPUT 1 1 R3        # Leemos el puerto 1 para la parte baja y metemos en R3
PUSH R3             # Guardamos en pila R3
INPUT 2 2 R3        # Leemos el puerto 2 para la parte alta y metemos en R3
PUSH R3             # Guardamos R3
LI 0 R14            # Actualizamos el estado de R14
                    # Para confirmar la lectura encenderemos las luces
OUTPUTR 0 3 R13     # Cargamos en R13 el estado de los leds verdes
LI 63 R12           # Cargamos en R12 un valor que tenga muchos unos
JUMP ledsv_on       # Encendemos leds verdes
FNSH


ctrl_switches:
```

```
INPUT 1 1 R3      # Leemos el puerto 1 para la parte baja y metemos en R3
PUSH R3           # Guardamos en pila R3
INPUT 2 2 R3      # Leemos el puerto 2 para la parte alta y metemos en R3
PUSH R3           # Guardamos R3
POP R3            # Sacamos el contenido de la pila en R3
OUTPUT 2 R3 2     # Ponemos la parte alta en el puerto2
POP R3            # Sacamos el contenido de la pila en R3
OUTPUT 1 R3 1     # Ponemos la parte baja en el puerto1

                  # Hacemos que parpadee estando en modo lectura
OUTPUTR 0 3 R13   # Guardamos estado de puerto salida P0 en R3
LI 1 R12          # Carga de inmediato
SUB R12 R13 R3    # Restamos valor leido a R12
BEZ ledsv_off     # Si es cero apagamos luces
JUMP ledsv_on     # Encendemos luces
FNSH

ledsv_off:
OUTPUT 0 R0 3    # Metemos en puerto0 el valor de R0 = 0
FNSH

ledsv_on:
OUTPUT 0 R12 3   # Metemos en puerto0 el valor de R12
FNSH

mostrar:
POP R3               # Sacamos el contenido de la pila en R3
OUTPUT 2 R3 2        # Metemos para la parte alta en el puerto 2
POP R3               # Sacamos el contenido de la pila en R3
OUTPUT 1 R3 1        # Metemos para la parte baja en el puerto 2
SUB R11 R2 R11       # Decrementamos el contador
BEZ termino_mostrar # Si el contador es cero ya no hay valores en la pila
FNSH

termino_mostrar:     # Encendemos todos los leds cuando ya no hayan valores en la pila
OUTPUTR 0 3 R13      # Cargamos el estado del P0 en R3
LI 255 R12           # Cargamos en R12 un 255 para encender todos los leds
SUB R12 R13 R3       # Restamos el valor de P0 a R3 para saber el estado en el que esta
BEZ ledsv_off        # Si es cero significa que estaban encendidos y hay que apagarlos
JUMP ledsv_on        # Si no es cero habrá que encenderlos
FNSH

apagar_todo:    # Cargamos en todos los puertos un cero
OUTPUT 0 R0 3
OUTPUT 1 R0 1
OUTPUT 2 R0 2
FNSH
```

7. Problemas y observaciones.

Los problemas más significativos que aparecieron fueron todos relacionados con sincronización y los tiempos de lectura en registros y en la memoria de la pila. Debemos aclarar que digo que fueron los más significativos porque fueron los más complicados de detectar. La mayoría de los problemas se solucionaban comprobando las simulaciones en el GTKWave pero era muy habitual que las cosas que funcionaban en la simulación no reflejasen el comportamiento en la placa real. (Fácilmente se compilo el programa más de 200) Todos los cambios significativos fueron por problemas de sincronización. Las cosas no ocurrían cuando “debían”

Durante el desarrollo de toda la práctica, y teniendo en cuenta que soy un alumno que había empezado la práctica el curso anterior, otro inconveniente con el que me tropecé fue la ampliación de los buses de 16 a 32 como se solicitó. Este cambio afectaba no sólo al diseño sino también al ensamblador, así que con el proyecto avanzado como lo tenía, decidí no adaptar la ampliación. Esta decisión tomó relevancia cuando se necesitaban guardar 10 bits en registros de 8 bits. Reconozco que las instrucciones pueden resultar confusas cuando se quiere “leer el puerto p1 para la parte baja” pero finalmente me quedé satisfecho de poder solucionar un problema con las “herramientas que tenía a mano”.

Otro problema fue que el ensamblador que programé en C++ no está completamente funcional, por lo que tuve que corregir algunos errores “a mano” en binario sobre el progfile.mem final. Algunos de los errores son el no interpretar bien la línea donde debe saltar y tampoco acepta comentarios. Fueron añadidos a posteriori para explicar el programa.