PROCESADOR MONOCICLO EN VHDL

rpcr213

Este proyecto consiste en la elaboración de un procesador monociclo con **VHDL** para una **FPGA** (*Basys3*).
El procesador consta de las siguientes especificaciones:

- **4 registros** (R0, R1, R2, R3) de **8 bits**
- Frecuencia de **1Hz**
- **256 bytes** de memoria **ROM** donde van almacenadas las instrucciones
- **256 bytes** de memoria **RAM** donde mediante la programación de la memoria ROM, se podrán guardar aquellos valores que se deseen

COMPONENTES

Top module (core.vhd)

```
entity core is
    Port (
        clk : in std_logic;
        rst : in std_logic;
        led : out std_logic_vector(15 downto 0);
        seg : out std_logic_vector(6 downto 0);
        an : out std_logic_vector(3 downto 0)
    );
end core;
```

Es la entidad de nivel superior que agrupa el controlador, la ruta de datos, el divisor de frecuencia y el conversor de 7 segmentos.
Interconecta las salidas obtenidas de la ruta de datos con los leds que se ven en la Basys3, la entrada al conversor de 7 segmentos para mostrar el número de instrucción por el display de 7 segmentos y las señales de control y estado entre la ruta de datos y el controlador.
Une el clk con el clk del divisor de frecuencia y la frecuencia que este da (que en realidad es un pulso) se usa junto al clk en la ruta de datos para la sincronización de los registros y la escritura de la RAM.
También permite la entrada de datos (botón rst), para volver al estado inicial (todos los registros a 0 y la RAM vacía).

Controlador (controller.vhd)

```
entity controller is
Port (
    data : in std_logic_vector(4 downto 0);
    ctrl : out std_logic_vector(10 downto 0)
);
end controller;
```

Dependiendo de su entrada data (la cuál consta de Z y el OPCODE), enviará señales por ctrl que harán que el circuito trabaje de diferente manera:
alias sel_mux2a1_1 : std_logic is ctrl(0);
alias write_banco_regs : std_logic is ctrl(1);
alias sel_mux2a1_2 : std_logic is ctrl(2);
alias ctrl_alu : std_logic_vector(2 downto 0) is ctrl(5 downto 3);
alias we_ram : std_logic is ctrl(6);
alias sel_mux2a1_3 : std_logic is ctrl(7);
alias sel_mux2a1_4 : std_logic is ctrl(8);
alias sel_mux2a1_5 : std_logic is ctrl(9);
alias sel_mux2a1_51 : std_logic is ctrl(10);

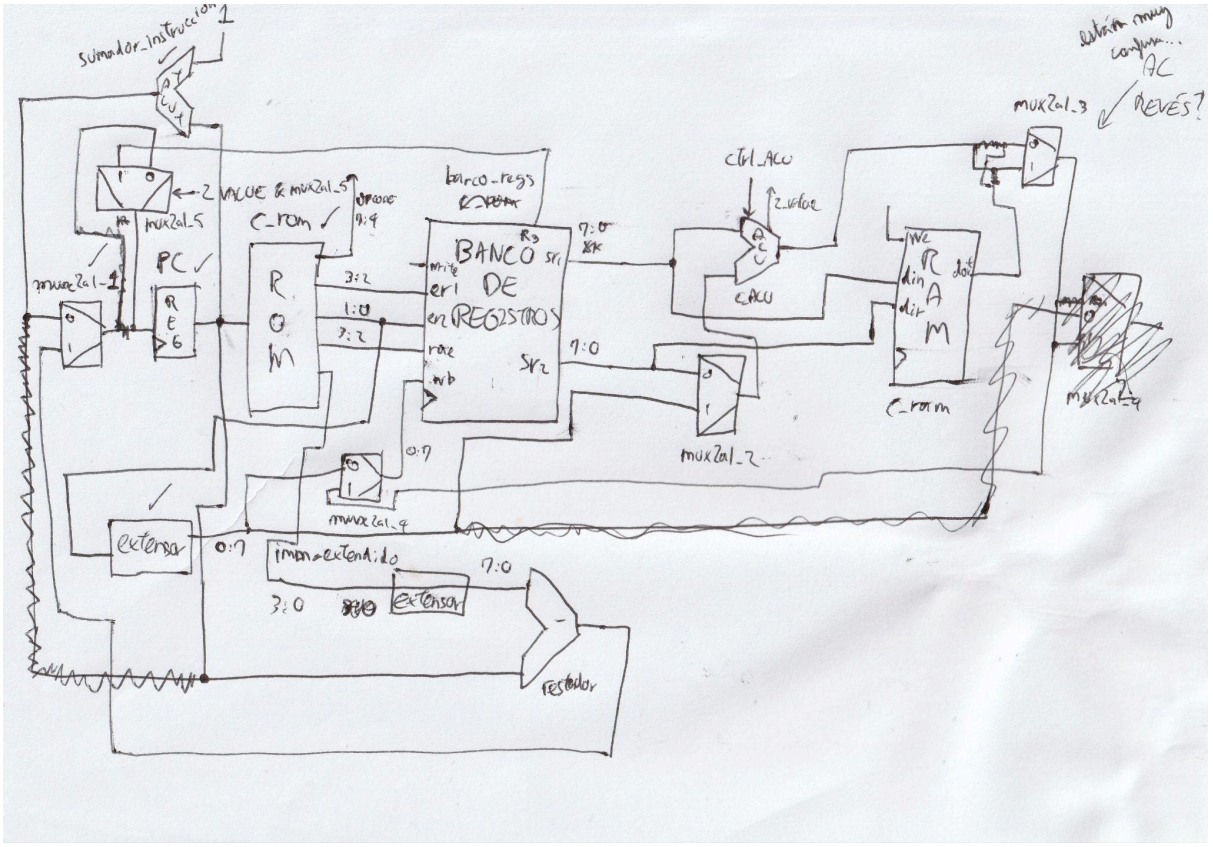
Divisor de frecuencia (divisor.vhd)

```
entity divisor is
    port (
        rst: in STD_LOGIC;
        clk_entrada: in STD_LOGIC;
        clk_salida: out STD_LOGIC
    );
end divisor;
```

Es un generador de pulsos. Tiene una clk_entrada que entra a 100MHz y una salida clk_salida que genera un pulso cada segundo. Los registros operan a una frecuencia de 100MHz pero se actualizan cuando el pulso (clk_p) emitido por el divisor de frecuencia para mantener un funcionamiento a 1Hz.

Ruta de datos (ruta_de_datos.vhd)

```
entity data_path is
Port (
    clk : in std_logic;
    clk_p : in std_logic;
    rst : in std_logic;
    ctrl : in std_logic_vector(10 downto 0);
    data : out std_logic_vector(4 downto 0);
    leds : out std_logic_vector(15 downto 0);
    numero_instruccion : out std_logic_vector(7 downto 0)
);
end data_path;
```



(Está un poco sucia, la reharé digitalmente)

Como entradas tiene:

- clk, clk_p, rst: señal de reloj y reset
- ctrl: señales de control definidas por el controller.

Como salidas tiene:

- data: contiene el OPCODE de la instrucción y Z (si sr1 y sr2 son equivalentes).
- leds: muestra por los 8 leds más significativos de la FPGA la instrucción que se está ejecutando y por los 8 menos significativos salida_mux2a1_3.
- numero_instruccion: el número de instrucción que se está ejecutando.

Componentes:

- PC (Program Counter)

Es un registro síncrono especial que se encarga de llevar la cuenta de las instrucciones, su salida va conectada a la ROM para extraer la instrucción a ejecutar y a la señal numero_instruccion para mostrar por pantalla el número de la instrucción que se está ejecutando.

- ROM (Read Only Memory)

Memoria combinacional de 256 bytes que contiene las instrucciones a ejecutar, los bits de su salida se conectan a diferentes componentes y sus 4 bits más significativos al controller para definir el comportamiento de cada instrucción.

- Banco de Registros

Banco con 4 registros de 8 bits.
Según sus entradas er1 y er2 se seleccionan los valores de salida de sr1 y sr2, que pueden contener los valores de R0 a R3.
Cuando write (we) vale '1', se escribe el valor de wb en el registro er1.
sr3 es la salida del valor del registro R3, osea la dirección a la que se saltará cuando se cumplan ciertas condiciones.

- ALUs

En este circuito hay 3 ALUs:

- sumador_instrucciones: es la ALU que sumará 1 al número de instrucción actual para pasar a la siguiente instrucción
- restador_instrucciones: es la ALU que restará el imm [3:0] de la instrucción j (jump) para retroceder tantas instrucciones como se desee
- c_ALU: es la ALU que según los datos de entrada y la señal de control ctrl [5:3], hará una suma, resta, (and, or, xor, not) lógicos o desplazamientos a la derecha o a la izquierda. Si sus 2 entradas son equivalentes, pone Z a '1'.

- RAM (Random Access Memory)

Es una memoria síncrona para la escritura y combinacional para la lectura de 256 bytes que cuando su entrada we = '1' se guardarán los valores pasados por din en la dirección dir. Por dout siempre saldrá el valor almacenado en dir.

- Extensores

Ambos extensores hacen posible las operaciones con inmediatos, rellenando con ceros los inmediatos de 2 y 4 bits para llegar a los 8 bits.

- Multiplexores

En esta ruta de datos hay 5 multiplexores:

- mux2a1_1: filtra entre si necesitamos avanzar 1 instrucción o retroceder inmediato [3:0] instrucciones definidas en la instrucción j (jump).
- mux2a1_2: filtra entre si la c_ALU va a operar con el valor de Rb o con el inmediato [0:1].
- mux2a1_3: filtra entre el dout de la SRAM o el valor de la c_ALU que llegará a mux2a1_4.
- mux2a1_4: filtra entre el valor del inmediato de li o el valor del mux2a1_3 para escribir en el banco de registros.
- mux2a1_5: filtra entre la dirección de salto calculada por el mux2a1_1 o la dirección de salto almacenada en R3 en función de la instrucción que se está ejecutando y el valor de Z de la c_ALU:
  (sel_mux2a1_5 and z_value) or (sel_mux2a1_51 and not(z_value))

Conversor 7 segmentos (conv_7seg.vhd)

```
entity conv_7seg is
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           display : out  STD_LOGIC_VECTOR (6 downto 0));
end conv_7seg;
```

Permite la conversión para mostrar el número de instrucción por los 4 pantallas de 7 segmentos.

INSTRUCCIONES

Cada instrucción será de 8 bits, y se seguirá el siguiente esquema para su especificación:
Ra -> registro a
Rb -> registro b

NOMBRE	OPCODE [7:4]	V0 [3:2]	V1 [1:0]	INFO
add	0000	Ra	Rb	Ra <- Ra + Rb
sub	0001	Ra	Rb	Ra <- Ra - Rb
and	0010	Ra	Rb	Ra <- Ra & Rb
or	0011	Ra	Rb	Ra <- Ra | Rb
xor	0100	Ra	Rb	Ra <- Ra ⊕ Rb
not	0101	Ra	-	Ra <- not(Ra)
ld	0110	Ra	Rb	Ra <- dir(Rb)
st	0111	Ra	Rb	dir(Rb) <- Ra
bne	1000	Ra	Rb	if (Ra = Rb) -> (PC <- R3)
beq	1001	Ra	Rb	if (Ra ≠ Rb) -> (PC <- R3)
j	1010	imm [3:0]		PC <- imm [3:0]
sr	1011	Ra	imm [1:0]	Ra <- Ra >> imm [1:0]
sl	1100	Ra	imm [1:0]	Ra <- Ra << imm [1:0]
addi	1101	Ra	imm [1:0]	Ra <- Ra + imm [1:0]
nop	1110	-	-	-
li	1111	Ra	imm [1:0]	Ra <- imm [1:0]

Se seguirá para la mayoría de instrucciones el esquema de:

1. Los 4 bits más significativos son el OPCODE
2. Los siguientes 2 bits seleccionan el registro Ra
3. Los 2 bits menos significativos seleccionan al registro Rb

Por ejemplo:

- add r1, r2 -> 0000 01 10
- sub r2, r0 -> 0001 10 00

Sin embargo si bien todas las instrucciones coinciden en que los 4 primeros bits pertenecen al OPCODE, según qué instrucción pueden variar los 4 bits menos significativos:

- ld (load)

ld se encargará de cargar el valor almacenado en la dirección contenida en el Rb en el Ra.

- st (store)

st se encargará de guardar el valor contenido en Ra en la dirección contenida en Rb.

- sr (shift right) y sl (shift left)

sr hará tantos desplazamientos a la derecha al valor contenido en Ra como digan los 2 bits menos significativos.
sl hará tantos desplazamientos a la izquierda al valor contenido en Ra como digan los 2 bits menos significativos.

- li (load immediate)

li cargará en Ra el valor definido por los 2 bits menos significativos.

- addi (add immediate)

addi sumará en Ra el valor contenido en Ra y el valor especificado por los 2 bits menos significativos.

- bne (branch if not equal) y beq (branch if equal)

bne hará un salto a la dirección contenida en R3 cuando los valores contenidos en los registros Ra y Rb sean diferentes.
beq hará un salto a la dirección contenida en R3 cuando los valores contenidos en los registros Ra y Rb sean iguales.

- j (jump)

j retrocederá tantas instrucciones como se le especifique en los 4 bits menos significativos.

SEÑALES DE CONTROL

OPCODE	sel_mux2a1_1	write_banco_regs	sel_mux2a1_2	ctrl_alu	we_ram	sel_mux2a1_3	sel_mux2a1_4	sel_mux2a1_5	sel_mux2a1_51
0000	0	1	0	000	0	1	1	0	0
0001	0	1	0	001	0	1	1	0	0
0010	0	1	0	010	0	1	1	0	0
0011	0	1	0	011	0	1	1	0	0
0100	0	1	0	100	0	1	1	0	0
0101	0	1	0	101	0	1	1	0	0
0110	0	1	0	000	0	0	1	0	0
0111	0	0	0	000	1	0	0	0	0
1000	not(z_value)	0	0	000	0	0	0	0	1
1001	z_value	0	0	000	0	0	0	1	0
1010	1	0	0	000	0	0	0	0	0
1011	0	1	1	110	0	1	1	0	0
1100	0	1	1	111	0	1	1	0	0
1101	0	1	1	000	0	1	1	0	0
1110	0	0	0	000	0	0	0	0	0
1111	0	1	1	000	0	1	0	0	0

ENSAMBLADOR

He creado un ensamblador sencillo en Python que escribiendo las instrucciones en el formato correcto en un archivo, lo lee y escribe sobre otro archivo (codigo_maquina.txt) el código en binario ya listo para escribir en la ROM.

A la hora de escribir en la memoria ROM (rom.vhd), se pegará directamente el código en el archivo de esta manera:

1. Escribimos el código ensamblador que queramos en un archivo, y le ponemos un nombre, por ejemplo “multiplicación.txt”
2. Ejecutamos el archivo “ensamblador.py”, nos pedirá el nombre del archivo, introduciremos “archivo.txt”
3. El ensamblador sobre el archivo “codigo_maquina.txt” escribirá las instrucciones en binario.
4. Copiamos esas instrucciones y las pegamos dentro del archivo “rom.vhd”.
5. Generamos un nuevo bitstream y el programa se ejecutará automáticamente en la placa Basys3.

```
nop
nop
nop
nop
li r2, 3
sl r2, 2
addi r2, 1
li r1, 0
li r0, 0
li r3, 3
sl r3, 2
addi r3, 3
addi r3, 2
beq r1, r2
addi r1, 1
add r0, r2
j 3
li r3, 0
st r0, r3
nop
nop
nop
nop
ld r1, r3
ld r1, r3
ld r1, r3
nop
nop
nop
```

archivo.txt

```
0 => b"11100000",
1 => b"11100000",
2 => b"11100000",
3 => b"11100000",
4 => b"11111011",
5 => b"11001010",
6 => b"11011001",
7 => b"11110100",
8 => b"11110000",
9 => b"11111111",
10 => b"11001110",
11 => b"11011111",
12 => b"11011110",
13 => b"10010110",
14 => b"11010101",
15 => b"00000010",
16 => b"10100011",
17 => b"11111100",
18 => b"01110011",
19 => b"11100000",
20 => b"11100000",
21 => b"11100000",
22 => b"11100000",
23 => b"01100111",
24 => b"01100111",
25 => b"01100111",
26 => b"11100000",
27 => b"11100000",
28 => b"11100000",
```

codigo_maquina.txt

```
constant ROM_CONTENT : rom_array := (
0 => b"11100000",
1 => b"11100000",
2 => b"11100000",
3 => b"11100000",
4 => b"11111011",
5 => b"11001010",
6 => b"11011001",
7 => b"11110100",
8 => b"11110000",
9 => b"11111111",
10 => b"11001110",
11 => b"11011111",
12 => b"11011110",
13 => b"10010110",
14 => b"11010101",
15 => b"00000010",
16 => b"10100011",
17 => b"11111100",
18 => b"01110011",
19 => b"11100000",
20 => b"11100000",
21 => b"11100000",
22 => b"11100000",
23 => b"01100111",
24 => b"01100111",
25 => b"01100111",
26 => b"11100000",
27 => b"11100000",
28 => b"11100000",
others => (others => '0')
);
```

rom.vhd