Fernando Arribas

Carné: 19030

Lab 10

Ejercicio 1

```
module ffd4(input clk, input reset, input en, input [3:0] d, output reg [3:0] q);

  always @ (posedge clk, posedge reset) begin
  if (reset) q <= 4'b0;
  else if (en) q <= d;
  end

endmodule

module fetch(input clk, reset, en, input [7:0] program_byte, output [3:0] instr, oprnd);

  ffd4 f1(clk, reset, en, program_byte[7:4], instr);
  ffd4 f2(clk, reset, en, program_byte[3:0], oprnd);

endmodule
```

El Fetch se realizo con la utilización de 2 módulos, aunque se puede realizar a la perfección solo con uno debido a que el fetch es un flip flop tipo D de 8 bits. Pero para facilitar la muestra de las salidas Instruction y Opernad del fetch se decidió a realizar en un módulo un FFD de 4 bits para luego en un segundo módulo juntar 2 FFD de 4 bits.

```
module ROM(input wire [11:0] PC, output wire [7:0] program_byte);
        //Definimos las variables de entrada y salida.

    reg [7:0] m[0:4095]; //Asignamos el tamaño de la memoria
                         //la cual es de 4k con 8 bits de ancho

    initial begin
        $readmemb("memoria.lab10", m); //guardamos un valor
            //binario en la memoria
    end

    assign program_byte = m[PC];

endmodule
```

Primero declaramos las variables de entrada, siendo una entrada L de 12 bits, la cual nos servirá para buscar en las 4096 localidades de nuestra memoria y una salida Y que nos desplegará lo que esta guardado en cierta localidad de nuestra memoria. Lo primero a realizar fue un array de memoria, el cual será de 4k localidades y 8 bits de ancho, la cual será una variable M. Luego por medio de la instrucción $readmemb leeremos el archivo externo que guardaremos en nuestra memoria, la cual será guardada en el arreglo m. Por último, asignamos que el valor de la salida será la localidad que seleccionemos con L.

```
32    module counter #(parameter N = 12) //Definimos variables
33                    (input wire clk, reset, en, load,
34                     input wire [N-1:0] val,
35                     output reg [N-1:0] q);
36
37      always @ (posedge clk or posedge load or posedge reset) begin
38        if (reset == 1) //flanco de reloj se colocara la salida en 0
39          q <= 12'b0;
40
41        else if (load == 1)//flando de reloj se cargara un valor al
42          q <= val;        //contador
43
44        else if (en == 1)//cuando enable sea igual a 1
45          q <= q + 1;   //comenzara a contar
46
47      end
48
49    endmodule
```

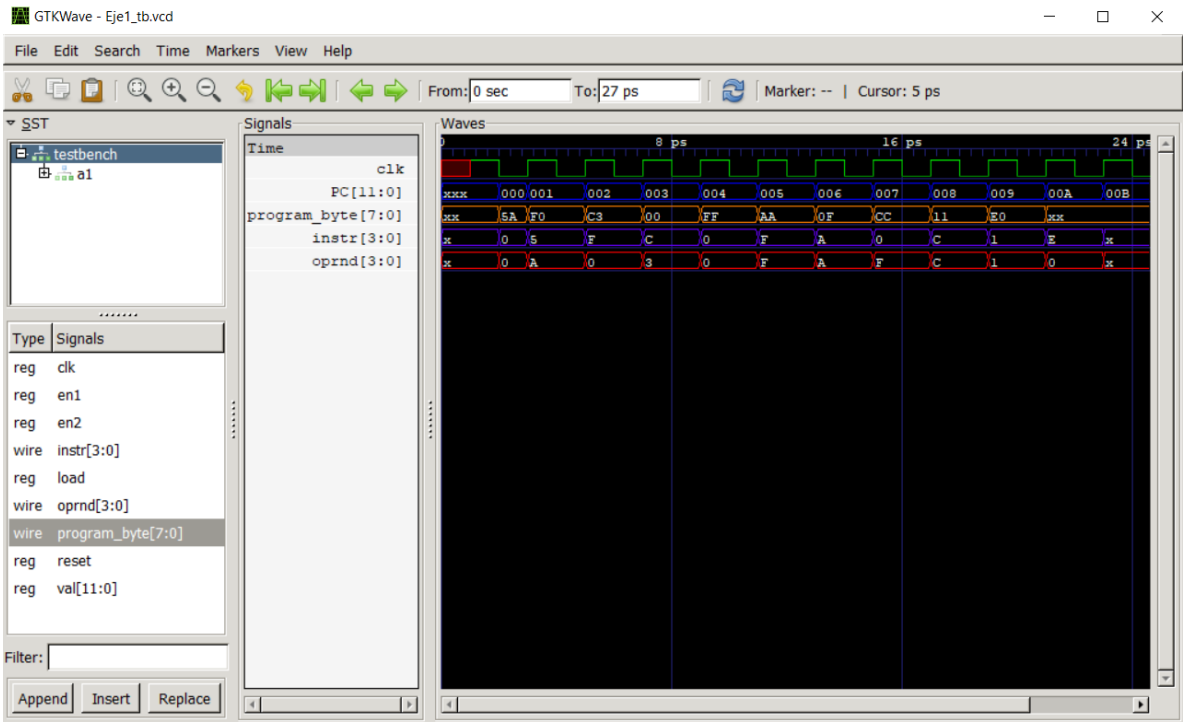
Primero definimos las variables a utilizar las cuales son un clock, un reset, un enable (en), un load, un variable de 12 bits llamada val, y la salida de 12 bits; adicionalmente definimos un parámetro, el cual nos indicará de cuantos bits será nuestro contador, en este caso 12 bits. Luego todo eso se ingresa a un always, en donde el reloj, el reset y load trabajaran con flancos de reloj positivos. Dentro existen un total de 3 condicionales, el primero indica si hay un cambio de reloj para reset la salida se colocara en 0. Luego si existe un flanco de reloj en load la salida pasara a ser los que tengamos en la variable val. Por último, si enable es igual a 1 la salida del contador será el valor pasado del mismo más uno. Hay que mencionar que todas estas operaciones están realizadas con non-blocking assignment, para que el contador pueda saltar entre las distintas opciones sin ninguna restricción.

```
51    module Eje1(input reset, en1, en2, load, clk, input [11:0] val, output [7:0] program_byte, output [3:0] instr, oprnd);
52
53      wire [11:0] PC;
54
55      counter a1(clk, reset, en1, load, val, PC);
56      ROM a2(PC, program_byte);
57      fetch a3(clk, reset, en2, program_byte, instr, oprnd);
58
59    endmodule
```

Luego se juntan los módulos fetch, contador y ROM en un mismo módulo para poder interconectarlos entre si y poder obtener los resultados deseados.

```
module testbench();

  reg clk, reset;
  reg [2:0] en, f;
  reg [3:0] b;
  wire carry, zero;
  wire [3:0] y;

  eje2 a1(clk, reset, en, f, b, carry, zero, y);

  initial begin
  #1

  clk = 0; en[2] = 1; en[1] = 1; en[0] = 1; b = 4'b0; f = 3'b010;
  reset = 0;
  $display("Ejercicio 2: Lab 10");
  $display("B | Y | C | Z");
  $monitor("%b | %b | %b | %b", b, y, carry, zero);
  #1 reset = 1;
  #1 reset = 0; f = 3'b010; b = 4'b1111;
  #1 f = 3'b000; b = 4'b0001;
  #1 f = 3'b001; b = 4'b0101;
  #1 f = 3'b011; b = 4'b0100;
  #1 f = 3'b100; b = 4'b0000;

  #30 $finish;
  end
```

```
    always
    begin
      #1 clk = ~clk;
      end

      initial begin
        $dumpfile("Eje2_tb.vcd");
        $dumpvars(0, testbench);
      end

  endmodule
```



Ejercicio 2

```
56   module buftri(input wire en, input wire [3:0]in, output wire [3:0]out);
57
58     assign out = (en == 1) ? in :
59                  (en == 0) ? 4'bz : 4'bx;
60
61   endmodule
```

Primero se llamaron las variables, de entrada, tenemos el enable y una de 4 bits llamada in, mientras que de salida tenemos una variable de 4 bits llamada out. Luego empezamos a construir el buffer triestado, asignamos a out un condicional en donde si en es igual a 1 deja pasar a in, de lo contrario estará en alta impedancia.

```
63   module Accu(input clk, input reset, input en, input [3:0] d, output reg [3:0] q);
64
65     always @ (posedge clk, posedge reset) begin
66     if (reset) q <= 4'b0;
67     else if (en) q <= d;
68     end
69
70   endmodule
```

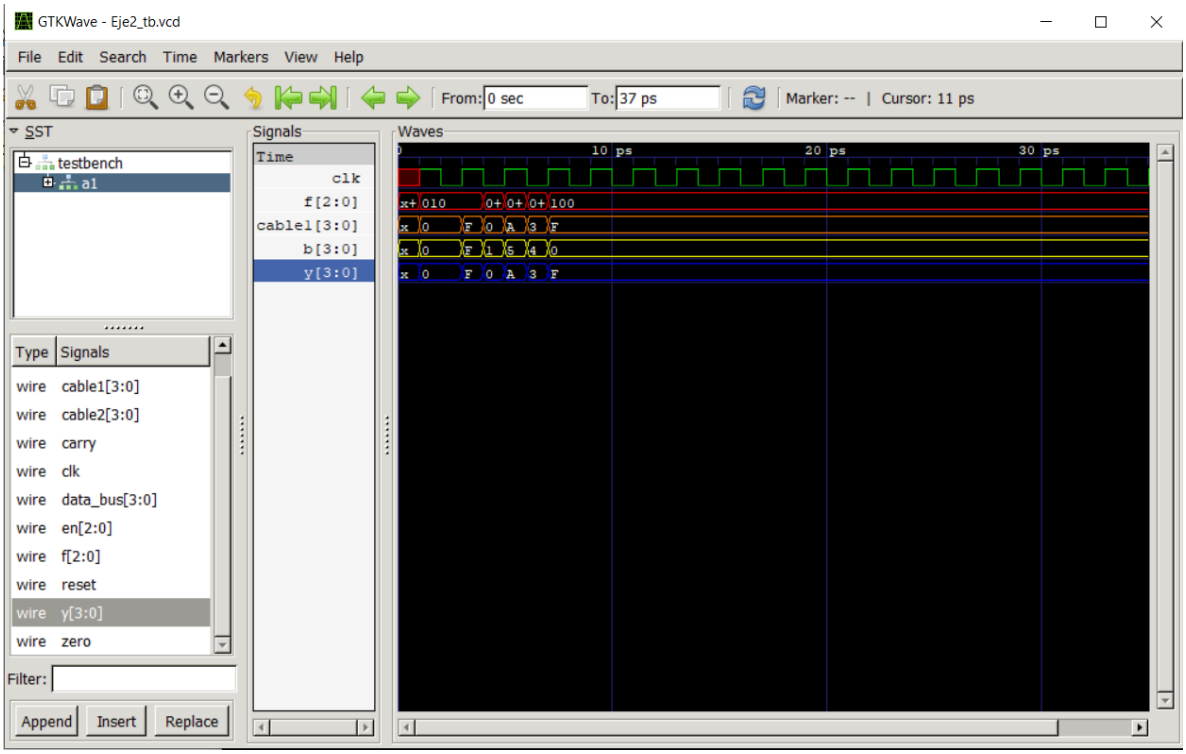
Para el accumulador se construyo un flip flop tipo de 4 bits.

```
1    module ALU(input wire [3:0] a, b,
2               input wire [2:0] f,
3               output reg [3:0] y,
4               output reg carry, zero);
5
6        reg [4:0] c;
7        always @ (a or b or f) begin
8
9            case(f)
10               3'b000: begin
11                 c = 5'b0;
12                 c = a; //Deja pasar lo que este en el Accumulador
13                 carry = (c[4] == 1) ? 1:0; //Si el bit mas significativo es 1 carry = on
14                 zero = (c[4] == 0) ? 1:0; //SI el bit mas significativo es 0 zero = on
15                 y = c[3:0];
16                 end
17
18               3'b001: begin
19                 c = 5'b0;
20                 c = (a - b); //Resta de A menos B
21                 carry = (c[4] == 1) ? 1:0; //Si el bit mas significativo es 1 carry = on
22                 zero = (c[4] == 0) ? 1:0; //SI el bit mas significativo es 0 zero = on
23                 y = c[3:0];
24                 end
25
26               3'b010: begin
27                 c = 5'b0;
28                 c = b; //Deja pasar lo que este en B
29                 carry = (c[4] == 1) ? 1:0; //Si el bit mas significativo es 1 carry = on
30                 zero = (c[4] == 0) ? 1:0; //SI el bit mas significativo es 0 zero = on
31                 y = c[3:0];
32                 end
```

```
                3'b011: begin
                  c = 5'b0;
                  c = (a + b); //Suma de A mas B
                  carry = (c[4] == 1) ? 1:0; //Si el bit mas significativo es 1 carry = on
                  zero = (c[4] == 0) ? 1:0; //SI el bit mas significativo es 0 zero = on
                  y = c[3:0];
                  end

                3'b100: begin
                  c = 5'b0;
                  c = ~(a & b); //NAND entre A y B
                  carry = (c[4] == 1) ? 1:0; //Si el bit mas significativo es 1 carry = on
                  zero = (c[4] == 0) ? 1:0; //SI el bit mas significativo es 0 zero = on
                  y = c[3:0];
                  end

                default: y = 3'b0; //Valor por defecto por si la
                                   //opción no se encuentra en la ALU
            endcase
        end
    endmodule
```

Para la ALU se modifico la original para colocar una con solo cinco instrucciones, dejar pasar A, dejar pasar B, suma, comparación y una compuerta NAND o NOR. Adicionalmente se le agregaron las salidas Carry y Zero, en donde carry se activa cuando hay overflow en alguna operación y zero se activa cuando todo es igual a 0.

```
module eje2(input clk, reset, input [2:0] en, f, input [3:0] b, output carry, zero, output [3:0] y);

  wire [3:0] data_bus, cable1, cable2;

  buftri a1(en[2], b, data_bus);
  ALU a2(cable2, data_bus, f, cable1, carry, zero);
  Accu a3(clk, reset, en[0], cable1, cable2);
  buftri a4(en[1], cable1, y);

endmodule
```



Link Repositorio: https://github.com/fernando19030/LaboratoriosElectronica_Digital_1-19030.git