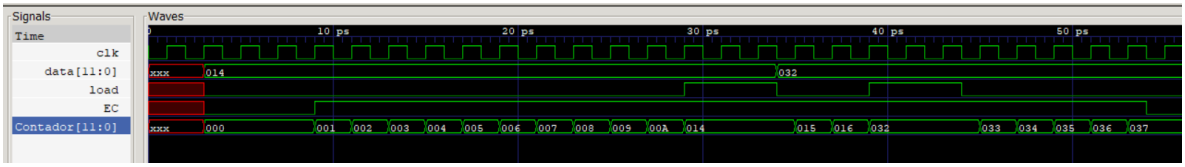
LABORATORIO 8

EJERCICIO 1

```
module CounterLoad(input wire [11:0]data, input wire load, enable, clk, reset, output reg [11:0]out);
    always @(posedge clk or posedge load or posedge reset)
        if (reset) begin
            out <= 12'b0 ;
        end else if (load) begin
            out <= data;
        end else if (enable) begin
            out <= out + 1;
        end
endmodule
```

El código para programar un counter con load es muy similar a como se programa un FlipFlop D, en donde se utiliza el always con el posedge para monitorear cuando el clk, load o reset tienen el valor de 1. Se programa un if para cada caso, en donde si el reset esta prendido la salida es 0, si load esta prendido la salida es el valor que data posea y si el enable está encendido el contador ya actúa y aumenta en 1 el valor de la salida.



En el siguiente diagrama se puede observar como el contador únicamente funciona si el enable esta encendido (10ps), en dado caso el load se enciende la salida pasa a ser el valor de data (30ps) y una vez se apaga load el contador empieza a funcionar desde el valor en el que se encuentra (35ps).

EJERCICIO 2

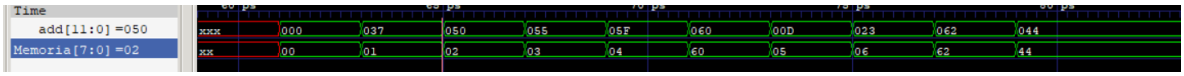
```
module MemoriaRom (input wire [11:0]address, output wire [7:0]data);
    reg [11:0] mem [0:4095] ;

    assign data = mem[address];

initial begin
  $readmemb("memoria.txt", mem);
end

endmodule
```

Esta memoria ROM es únicamente de lectura por lo que no necesita de ningún enable o un valor externo para saber si debe leer o escribir. Ya que se deseaba una memora de 4K x 8 se colocó un input de 12 bits para poder seleccionar las 4096 posiciones y una salida de 8 bits. Para poder lograr esto se registro una variable

mem que era una arreglo de 1 dimensión y sus datos se obtenían de un archivo externo. En este caso se utilizó $readmemb ya que se quería una representación en binario, en dado caso se desee una representación en hexadecimal la instrucción sería $readmemh.

```
1  00000000    96  00000100    99  01100010
56 00000001    97  01100000    69  01000100
81 00000010    14  00000101
86 00000011    36  00000110
```

Para comprobar que si funcionaba en el testbench se leyó la posición 0, 55, 80, 85, 95, 96, 13, 35, 98 y 68 en donde sus valores en el archivo llamado memoria.txt poseen esos valores y en su respuesta se pueden observar que si son los mismos



En el diagrama se puede observar que al llamar a la posición deseada su valor es el correcto y es el que se encuentra guardada en el archivo memoria.txt

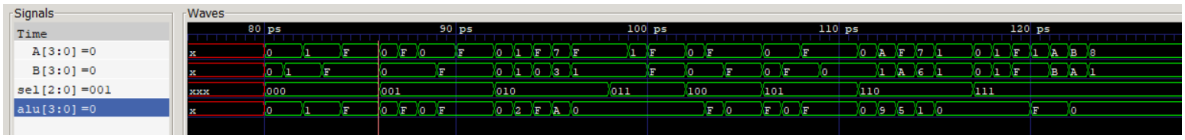
EJERCICIO 3

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

```
module ALU (input wire [3:0]A, input wire [3:0]B, input wire [2:0]sel, output reg [3:0]Y);
reg [4:0]X;
reg menor;
    always @ (A or B or sel)
        case (sel)
            0 : Y = A&B;
            1 : Y = A | B;
            2 : begin
                X = A + B;
                Y[0] = X[0];
                Y[1] = X[1];
                Y[2] = X[2];
                Y[3] = X[3];
                end
            3 : Y = 0;
            4 : Y = A&~B;
            5 : Y = A | ~B;
            6 : begin
                X = A - B;
                Y[0] = X[0];
                Y[1] = X[1];
                Y[2] = X[2];
                Y[3] = X[3];
                end
            7 : begin
                menor = (A < B);
                if (menor)begin
                Y = 4'b1111;
                end else begin
                Y = 0;
                end
                end
            default : $display("Error in SEL");
        endcase
```

Para la programación de la ALU se realizó por medio de cases en donde en base a la tabla descrita arriba se realizó un case diferente para cada caso, el sel era el encargado de seleccionar que case actuaría sobre Y. Para el case 0, 1, 4 y 5 únicamente se programó las compuertas de AND y OR para A y B. Para el caso 2 y 6 en donde se realiza una operación aritmética se creo un wire de 5 bits para poder contener el overflow, pero nuestra ALU lo ignora por lo que las salidas únicamente

son del bit 0 hasta el cuarto bit del wire que se creó. Para el último caso nuestra ALU únicamente compara si A es menor que B en donde si esto se cumple nuestra salida se coloca en 1 todos los bits, de lo contrario nuestra salida es 0.



En el diagrama podemos observar como de lo 80ps a los 85ps esta funcionando el case 0 en donde el AND funciona para cada bit individual. De los 85ps a los 92ps funciona el case 1, en donde el OR aplica para cada bit individual. De los 92ps a los 98ps funciona el case 2 en donde se puede observar que las sumas se hacen correctamente. De los 98ps a los 102ps se realiza el case 3 en donde sin importar el valor de A y B la salida siempre es 0. De los 102ps a los 107ps funciona el case 4 en donde de nuevo el AND se aplica a cada bit individual de A y B. De los 107ps a los 111ps funciona el case 5 en donde de igual forma el AND se aplica a cada bit. De los 111ps a los 118ps aplica el case 6 en donde se puede observar que las restas se realizan correctamente. Por último tenemos el case 7 en donde se observa que la salida vale F si A < B.

LINK GITHUB

https://github.com/lan19175/Electronica_Digital_1-19175.git