Guatemala, 24/Octubre/2020 Electrónica Digital
Ricardo Pellecer Orellana Carné: 19072

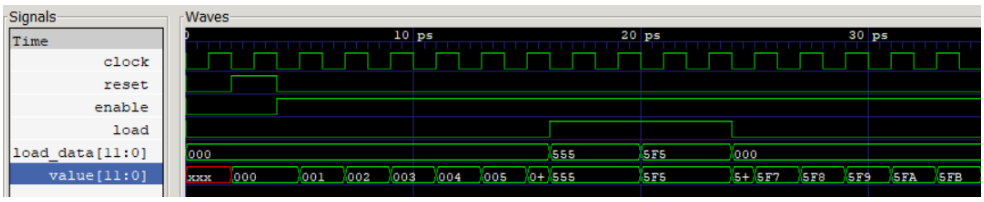
Laboratorio 8

Ejercicio 1:

Para la creación del contador se utilizaron bloques always para que pueda funcionar con el cambio del clock, el reset, el load o algún cambio en el dato que se está cargando. Por esto, como puede verse, se utilizaron output reg. Al mismo tiempo, como se observa, tanto el dato para cargar como el valor de salida son de 12 bits de ancho. Finalmente, se diseñó para que el valor se coloque en 0 si hay un reset, para que tome el valor del dato de carga cuando el load está activado y para que cuente 1 cada flanco de reloj si el enable está activado. Estas funciones pueden verse en el diagrama de timing de GTKWave y la tabla mostrada en la terminal.

```
module Counter (input wire clock, reset, load, enable, input wire [11:0]load_data, output reg [11:0]value);
  always @ (posedge clock, posedge reset, load, load_data) begin
    if (reset) begin
      value <= 12'b0;
    end
    else if (load) begin
      value <= load_data;
    end
    else if (enable) begin
      if (value < 12'b111111111111) begin
        value <= value + 1;
      end
    end
  end
endmodule
```

```
CONTADOR
clock reset load enable   load_data    | value
---------------------------------------|--------
0     1     0     0     000000000000   |   000000000000
1     1     0     0     000000000000   |   000000000000
0     0     0     1     000000000000   |   000000000000
1     0     0     1     000000000000   |   000000000001
0     0     0     1     000000000000   |   000000000001
1     0     0     1     000000000000   |   000000000010
0     0     0     1     000000000000   |   000000000010
1     0     0     1     000000000000   |   000000000011
0     0     0     1     000000000000   |   000000000011
1     0     0     1     000000000000   |   000000000100
0     0     0     1     000000000000   |   000000000100
1     0     0     1     000000000000   |   000000000101
0     0     0     1     000000000000   |   000000000101
1     0     0     1     000000000000   |   000000000110
0     0     1     1     010101010101   |   010101010101
1     0     1     1     010101010101   |   010101010101
0     0     1     1     010101010101   |   010101010101
1     0     1     1     010101010101   |   010101010101
0     0     1     1     010111110101   |   010111110101
1     0     1     1     010111110101   |   010111110101
0     0     1     1     010111110101   |   010111110101
1     0     1     1     010111110101   |   010111110101
0     0     0     1     000000000000   |   010111110110
1     0     0     1     000000000000   |   010111110111
0     0     0     1     000000000000   |   010111110111
1     0     0     1     000000000000   |   010111111000
```



Ejercicio 2:

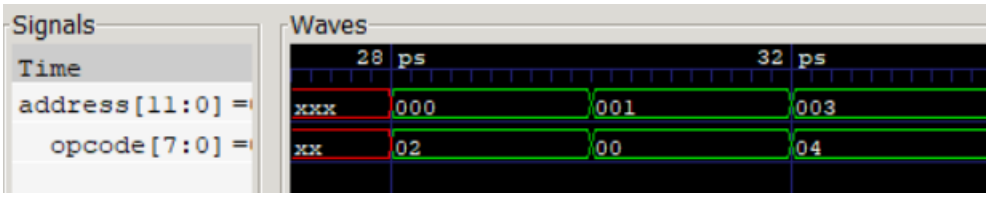
Para crear la memoria ROM se utilizó el arreglo de dos dimensiones de verilog y la función readmemh, la cual toma un documento con datos en hexadecimal y los coloca en la variable del arreglo 2D. Al módulo se le colocó un input que representa a la dirección que escoge qué byte se lee de la memoria y un output que representa la instrucción u opcode, el cual muestra el byte seleccionado. Como puede verse en la tabla de la terminal y en el diagrama de timing, al introducir las direcciones 0, 1 y 3, se muestran los datos que corresponden a estas direcciones en el documento ejemplo.list.

```
module ROM_Memory (input wire [11:0]address, output wire [7:0]opcode);
  reg [7:0] ROM [0:4095];
  initial begin
      $readmemh("ejemplo.list", ROM);
  end
  assign opcode = ROM[address];
endmodule
```

code.v | ejemplo.list

```
02
00
00
04
00
00
FA
02
00
00
00
00
29
D5
04
00
```

```
MEMORIA ROM
address         |    opcode
----------------|-------------
000000000000    |   00000010
000000000001    |   00000000
000000000011    |   00000100
```



Ejercicio 3:

Para la creación de la ALU se utilizó un bloque condicional case, el cual muestra todos los escenarios posibles que puede tomar una variable. Este bloque se coloca en un bloque always para que pueda ejecutarse a cada momento. En las condiciones se colocaron todas las opciones que muestra la ALU presentada en el libro.

```
module ALU (input wire [3:0]A, input wire [3:0]B, input wire [2:0]selector, output reg [3:0]result);
  always @ ( * ) begin
    case (selector)
    0:
      result <= A & B;
    1:
      result <= A | B;
    2:
      result <= A + B;
    3:
      result <= 4'b0;
    4:
      result <= A & ~B;
    5:
      result <= A | ~B;
    6:
      result <= A - B;
    7:
      result <= (A < B) ? 4'b1111:4'b0;
    default:
      result <= 4'b0;
    endcase
  end
endmodule
```

```
ALU
Primera Prueba
A       B    Selector |  Resultado
---------------------|-------------
1010   0101     000   |   0000
1010   0101     001   |   1111
1010   0101     010   |   1111
1010   0101     011   |   0000
1010   0101     100   |   1010
1010   0101     101   |   1010
1010   0101     110   |   0101
1010   0101     111   |   0000

Segunda Prueba
0110   1100     000   |   0100
0110   1100     001   |   1110
0110   1100     010   |   0010
0110   1100     011   |   0000
0110   1100     100   |   0010
0110   1100     101   |   0111
0110   1100     110   |   1010
0110   1100     111   |   1111
```