INSTITUTO TECNOLÓGICO DE BUENOS AIRES

Trabajo Práctico Nº 1

Electrónica III - 2019

Grupo 1:

Farall, Facundo
Gaytan, Joaquín
Kammann, Lucas
Maselli, Carlos

1 de septiembre de 2019

1. Ejercicio 1

Se realizó un programa bajo el nombre run.py, dentro de la carpeta EJ_1 del proyecto, el cual calcula el rango y resolución para una cierta convención de números en punto fijo. Recibe tres parámetros por línea de comando: un 1 o un 0 indicando si el sistema es signado, cantidad de bits de la parte entera, y cantidad de bits de la parte fraccionaria, en ese órden. Ante algún error en el formato en que se le pasan los parámetros, o ya sea porque los mismos exceden las capacidades del programa, se imprimirá en pantalla el mensaje ERROR. Los errores pueden estar ocasionados por alguna de las siguientes razones:

- No se pasa ningún argumento.
- Los argumentos no son números enteros.
- Se pasa una cantidad de argumentos distinta a 3.
- El primer argumento (que indica si el sistema es signado o no) es distinto de 0 o 1.

1.1. Limitaciones del programa

También puede devolverse ERROR a causa de que la cantidad de bits asignados a la parte entera o a la fraccionaria exceden los límites del lenguaje utilizado para el programa. El código se escribió en Python y se buscaron los casos límite para los cuales el programa deja de devolver valores con sentido:

- La cantidad de bits de la parte entera debe ser menor o igual a 1023.
- La cantidad de bits de la parte fraccionaria debe se menor o igual a 1074.

Además de las limitaciones mencionadas, existe un error en la representación de los números fraccionarios, inherente al sistema por el cual las computadoras guardan los números. La mayoría de los números que en sistema decimal tienen una representación con una cantidad finita de dígitos detrás de la coma, no poseen tal característica en un sistema binario. Tal es el caso del número 0,1 en base 10, que en Python, así como muchos otros lenguajes de programación que representan a los números fraccionarios bajo el formato normalizado de IEEE-754, es guardado como 0,1000000000000000055511151231257827021181583404541015625. En la mayoría de estos casos, al imprimir el número en pantalla, el lenguaje de Python decidirá aproximar el valor real guardado en memoria, a uno que considere más amigable para el usuario humano. Un caso en el cual estas limitaciones pueden verse reflejadas, es si se ingresan como parámetros 0, 5 y 740 (no signado, con 5 bits de parte entera y 740 bits de parte fraccionaria), la salida del programa será:

- Res: 1.7290327071306454e-223
- Ran: 32.0

A estos se les puede calcular el error absoluto de la siguiente manera:

$$E_{Res} = 2^{-740} - 1{,}7290327071306454 \cdot 10^{-223} \approx 1 \cdot 10^{-236} \tag{1.1}$$

$$E_{Ran} = 32 - \left(2^5 - 2^{-740}\right) \approx 1{,}7290327071286 \cdot 10^{-223} \tag{1.2}$$

Para la resolución, se observarán diferencias recién luego de 13 cifras significativas, mientras que para el rango, luego de 223. Ambos errores cometidos pueden considerarse despreciables para casi cualquier sistema real de medición.

2. Ejercicio 3

De los dos módulos pedidos para implementar en Verilog se tomó la decisión de realizar uno mediante las compuertas lógicas que lo componen, y otro a través de la descripción de su comportamiento (behavioural). La razón detrás de esta decisión fue para hacer uso de las variantes provistas por Verilog, e interiorizarnos en su estilo de programación.

2.1. Decoder de 4 salidas

Esta implementación de un decoder de 4 salidas recibe una arreglo de 2 bits, que determinan cual de las cuatro salidas accionar. La misma se realizó a través de lógica de compuertas, donde únicamente la combinación correcta de los dos bits de entrada, ponen a la salida con ese número, en 1 lógico. Las relaciones lógicas son sencillas de comprender y quedan explicitadas en el mismo código. A continuación se presenta el código en Verilog:

```
module decoder4out(coded, y0, y1, y2, y3);
    input [1:0] coded;
    output y0, y1, y2, y3;

    assign y0 = ~coded[1] & ~coded[0];
    assign y1 = ~coded[1] & coded[0];
    assign y2 = coded[1] & ~coded[0];
    assign y3 = coded[1] & coded[0];

endmodule
```

2.2. MUX de 4 entradas

Esta implementación de un mux de 4 entradas recibe un arreglo de 4 bits con las 4 "fuentes", otro arreglo de 2 bits que hace las veces de selector, y cuenta con una salida de 1 bit. La salida copiará el valor de la entrada determinada por el selector. El módulo se logró mediante una descripción del comportamiento del mismo, en el cual se le especificó qué debía realizar ante cambios en alguna de sus entradas. A continuación se presenta el código en Verilog:

```
module mux4in (x, sel, y);
    input [3:0] x;
    input [1:0] sel;                    // sel selects the exit (x[3], x[2], x[1], x[0]).
    output reg y;

    always @(sel or x) begin
        if (sel == 0)
            assign y = x[0];
        else if (sel == 1)
            assign y = x[1];
        else if (sel == 2)
            assign y = x[2];
        else if (sel == 3)
            assign y = x[3];
```

```
        end

    endmodule
```

2.3. TEST-BENCH

Se sometió a los dos módulos a un testeo de su respuesta a cada una de las posibles entradas, y los resultados fueron los esperados. Los mismos pueden ser replicados ejecutando los comandos:

```
user@computer: path/to/EJ_3/folder$ make
user@computer: path/to/EJ_3/folder$ ./run
```