Proyecto Final
2023-2

Electrónica Digital II
Carlos Ivan Camargo Bareño

Elaborado por:
Andres Felipe Quijano Montenegro, aqujano@unal.edu.co
Alejandro Mahecha Arango, almahechaa@unal.edu.co
Julian Leonardo Robles Cabanzo, juroblesc@unal.edu.co

Grupo 1
6 de diciembre de 2023

Índice

1. Introducción

Este informe detalla el diseño de un proyecto centrado en el control de un helicóptero mediante señales NEC generadas por una FPGA Colorlight 5A-75E. La esencia de este proyecto reside en la ingeniería aplicada, destacando el diseño del protocolo de comunicación y las etapas cruciales del circuito del helicóptero.

En lugar de buscar la innovación por sí misma, nos sumergimos en los aspectos técnicos que definen la funcionalidad de este sistema. Desde la configuración del protocolo de comunicación hasta el desarrollo del circuito, cada fase se presenta con un enfoque práctico y funcional, resaltando la viabilidad y eficacia del control remoto del helicóptero a través de la FPGA Colorlight 5A-75E.

De igual manera, es de suma importancia mencionar que se llevara a cabo el diseño del protocolo de transmisión NEC con el fin de poder generar señales propias y con base a estos controlar el circuito diseñado para el helicóptero a nuestro gusto, evidenciando que errores se pueden producir a lo largo del diseño e implementación, generando así una contextualización más a fondo y logrando un aprendizaje tanto práctico como teórico, todo esto haciendo uso adicionalmente de la arquitectura de conjunto de instrucciones abierta RISC-V, promoviendo así el desarrollo de hardware y software.

2. Marco teórico

El protocolo NEC (National Electronic Code) es un estándar ampliamente utilizado en la comunicación de dispositivos electrónicos mediante señales infrarrojas. Este protocolo se encuentra presente en múltiples de los dispositivos de nuestra cotidianidad, tales como los televisores, reproductores DVD o aires acondicionados.

El protocolo de comunicación NEC se basa en la modulación por amplitud de pulso para modular la señal infrarroja, codificando la señal transmitida en varios pulsos. Generalmente, la frecuencia de transmisión suele ser de 38 kHz, esto con el de discriminar ruido ambiente. El formato de datos en un comando NEC típico consta de un encabezado, una dirección, la inversa lógica de la dirección, un código de comando y finalmente la inversa lógica del comando. A continuación, se va a dar una contextualización más a fondo del protocolo de transmisión:

- **Protocolo de Funcionamiento NEC:**
  El protocolo de transmisión IR NEC utiliza la codificación de distancia de impulsos de los bits de mensaje. El tiempo definido para cada cada impulso es el siguiente:
  - 0 Lógico, es representado por un pulso de 562.5 μs seguido por un espacio 562.5 μs, con un tiempo de transmisión total de 1.125 ms.
  - 1 Lógico, es representado por un pulso de 562.5 μs seguido por un espacio 1.6875 ms, con un tiempo de transmisión total de 2,25 ms.

Sin embargo, cuando una tecla de un mando a distancia infrarrojo es presionada, el mensaje transmitido consiste en el siguiente orden:

1. Un impulso líder de 9 ms.
2. Un espacio de 4.5 ms.
3. La dirección de 8 bits para el dispositivo receptor.
4. La inversa lógica de la dirección.
5. El comando enviado de 8 bits.
6. La inversa lógica del comando.

Un ejemplo de lo anteriormente descrito se presenta en la siguiente imagen

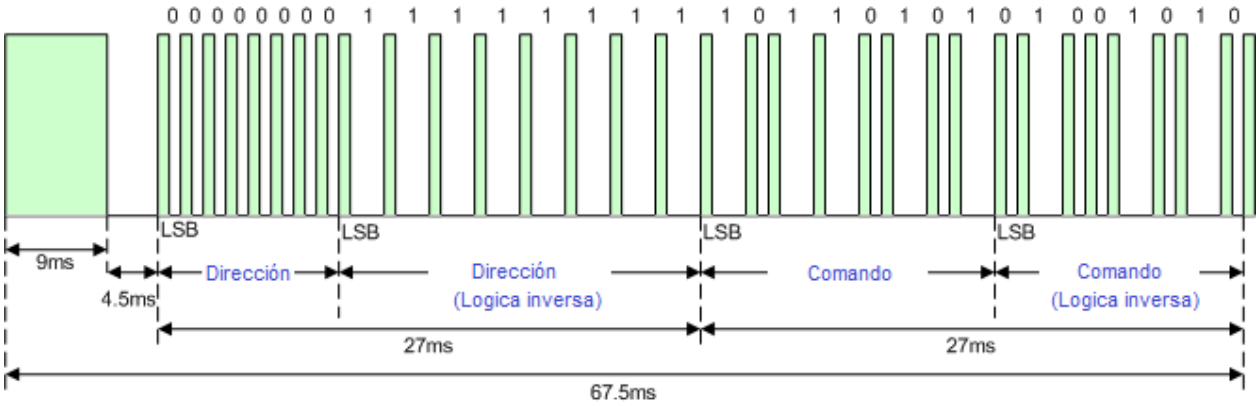

Figura 1: Ejemplo de transimisión IR NEC. [1]

3. Procedimiento realizado:

Nuestro proyecto consistía en el control de un helicóptero a control remoto por medio del protocolo NEC. La transmisión de la señal se buscaba realizar por medio de uno de los pines de la tarjeta FPGA Colorlight 5a-75e a un diodo infrarrojo y la recepción seria llevada a cabo a través de un receptor infrarrojo y sería decodificada en Arduino para generar las señales que movieran el helicóptero. Lo primero que fue realizado fue el diseño del protocolo con los tiempos anteriormente mencionados en Verilog:

- **Diseño del protocolo en Verilog:**

El algoritmo diseñado para generar los pulsos con sus respectivos tiempos dependiendo del bit de entrada (1 o 0), fue el siguiente:

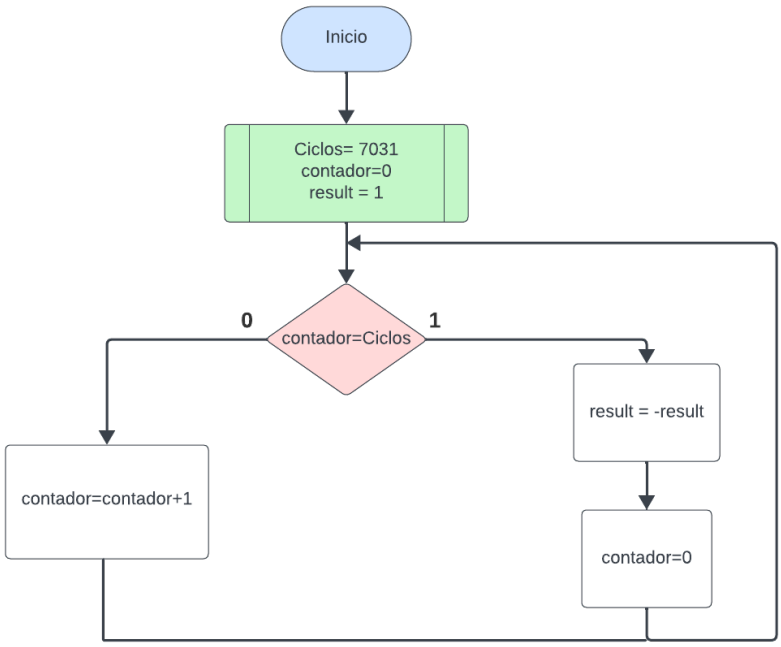

Figura 2: Algoritmo para la implementación del protocolo NEC.

Mediante el algoritmo implementado mediante un diagrama de flujo en la figura 2, se diseñó la siguiente máquina de estados basado en el funcionamiento mencionado de manera previa:

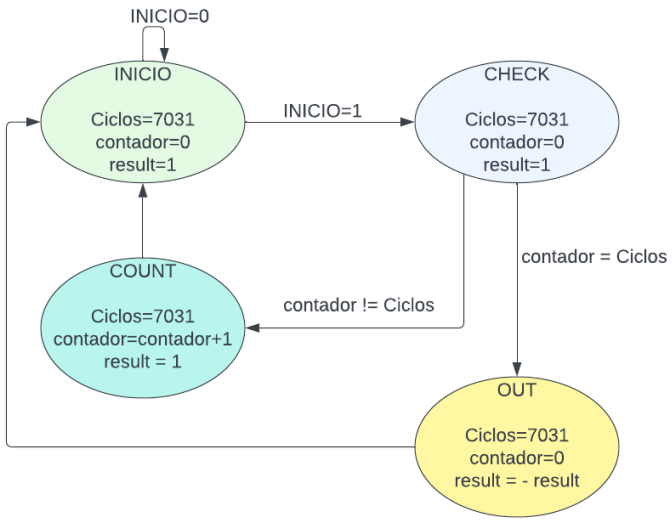

Figura 3: Máquina de estados para la implementación del protocolo NEC.

Su funcionamiento se basa en la división de la frecuencia del reloj presente en la tarjeta (25 MHz) sobre el inverso del tiempo para el bit de cero (562.5 μs), esto con el fin de tener un conteo de ciclos del reloj de la FPGA necesarios para cambiar el estado de la salida e ir enviando los pulsos:

$$\frac{25MHz}{2 * \frac{1}{562{,}5\mu s}} = \frac{14062{,}5}{2}$$

Esto quiere decir, que para enviar por ejemplo un pulso de cero, era necesario contar 14062 ciclos del reloj de la tarjeta manteniendo la salida en ALTO (1), y otros 14062 ciclos de reloj de la tarjeta manteniendo en la salida en BAJO (0), para transmitir completamente el pulso de cero. Para el pulso en uno, simplemente era necesario que la señal de salida se mantuviera en ALTO (1), un valor igual a 14062*3 ciclos (esto debido a que el tiempo en encendido para cuando se envia un bit de 1 es igual a 562.5 μs*3).

Para el encabezado inicial presente en el protocolo NEC, el conteo de ciclos de reloj de la FPGA fue igual a:

$$\frac{25MHz}{2 * \frac{1}{4{,}5ms}} = \frac{112500}{2}$$

Cabe aclarar que fue necesario en ambos casos la multiplicación por 2 del denominador, debido a que en el código de Verilog se estaba verificando los flancos de subida del reloj. Una vez establecido esto, se definieron las señales de entrada y de salida del código:

```
1  module infra (
2          input              reset,
3          input              clk,
4          // Control lines
5          input              init,
6          output reg         done,
7          //
8          output reg         result,
9          output reg [31:0]  result_infra,
10         input      [31:0]  op_A
11 );
12 //parameter contador_cero = 13872.925/2; //62 :V
13 parameter contador_cero = 14062/2; //62 :V
14 parameter pulso_cero=112500/2; // PULSO DE 4.5ms
```

Figura 4: Entradas del código.

Los estados definidos para la máquina de estados fueron:

```
16 parameter START  = 5'b00000;
17 parameter PULSE1 = 5'b00001;
18 parameter PULSE0 = 5'b00010;
19 parameter SALIDA0 = 5'b00011;
20 parameter SALIDA1 = 5'b00100;
21 parameter SUMAR = 5'b00101;
22 parameter SUMAR1 = 5'b00110;
23 parameter SHIFT = 5'b00111;
24 parameter COMP = 5'b01000;
25 parameter CHECK0 = 5'b01001;
26 parameter SUMAR2 = 5'b01010;
27 parameter SALIDA2 = 5'b01011;
28 parameter CHECK1 = 5'b01100;
29 parameter CHECK2 = 5'b01101;
30 parameter SUMAR3 = 5'b01110;
31 parameter SUMAR4 = 5'b01111;
32 parameter SALIDA3 = 5'b10000;
33 parameter END = 5'b10001;
34 parameter END0 = 5'b10010;
35 parameter SUMAR5 = 5'b10011;
36 parameter START1 = 5'b11111;
```

Figura 5: Estados de la maquina.

Posteriormente, se definieron las señales intermedias que se iban a utilizar en el transcurso del código, estableciendo los registros de un tamaño adecuado para que pudieran llegar al valor del conteo de ciclos mencionado anteriormente, junto con las condiciones de RESET del código.

```
initial begin
        result = 0;
        done   = 0;
end
        reg [21:0] count; // Contador de pulsos 1/0
        reg [21:0] count0; // Contador pulsos 1 y 0 iniciales
        reg [0:0] msb;
        reg [5:0] contador; // Contador para 32 bits
        reg [1:0] contador2;

always @(posedge clk)
begin
        if (reset) begin
                done   <= 0;
                result <= 1;
                result_infra <= 0;
                state   = START;
        end else begin
```

Figura 6: Señales intermedias.

Continuando, se va a presentar el código usado para realizar el conteo de ciclos y cambiar el estado de la señal de salida a lo largo del código. Para este ejemplo, se va a presentar el conteo para el encabezado inicial de 9 ms en ALTO y que posteriormente pasa al estado para enviar el pulso de 4.5 ms en BAJO. Para este caso, PULSE1 realiza el conteo para determinar si la señal bandera count0 es igual al número de pulsos necesarios para la señal de 9 ms (el doble de 4.5 ms) y si se cumple la condición, cambia la señal de result y pasa al estado para empezar el conteo para el pulso de 4.5 ms. De lo contrario, pasa al estado de suma para incrementar el contador.

```
PULSE1:begin
        if(count0==(pulso_cero*2))
                state = SALIDA0;
        else
                state = SUMAR;
end

SUMAR: begin
        count0=count0+1;
        state   = PULSE1;
end

SALIDA0: begin
        result <= ~result;
        count0 = 0;
        state = PULSE0;

PULSE0:begin
        if(count0==pulso_cero)
                state = SALIDA1;
        else
                state = SUMAR1;
end
```

Figura 7: Código realizado para conteos.

Ahora, para ir recorriendo el arreglo del código de entrada de 32 bits (8 bits de dirección, 8 bits de dirección negados, 8 bits de comando y 8 bits de comando negado), se implementó el siguiente código:

```
SHIFT: begin
        contador2=2;
        msb=A[31];
        A = A<<1;
        contador=contador-1;
        if (A==0 && contador==0)
                state = END0;
        else
                state = COMP;
end
```

Figura 8: Código realizado para recorrer el registro.

En este código, se reinicia el valor de contador2 (debido a que cada bit ya sea 0 o 1 tiene un ciclo en ALTO y otro en BAJO), se almacena el valor del bit más significativo de A (señal de entrada) y se realiza un corrimiento de la misma a la izquierda, y finalmente se resta al contador del número de bits

de la entrada (32) en 1, esto con el fin de evidenciar si la entrada es igual a cero y si el contador es igual a cero (representando que se realizaron los 32 corrimientos), para pasar al estado final de la máquina, de lo contrario entra en el estado de COMP, el cual determina si el bit ingresado es 1 o 0 y en base a a esto se envía los pulsos con sus tiempos estimados, siendo este el código en términos generales.

El código del testbench realizado para la maquina de estados fue el siguiente:

```
`timescale 1ns / 1ps
`define SIMULATION
module infra_TB;
  reg  clk;
  reg  reset;
  reg  init;
  reg  [31:0] op_A;
  wire result;
  wire [31:0] result_infra;
  wire done;

  infra uut (
     .clk(clk),
     .reset(reset),
     .init(init),
     .op_A(op_A),
     .result(result),
     .result_infra(result_infra),
     .done(done)
  );

  parameter PERIOD = 20;
  initial begin  // Initialize Inputs
     clk = 0; reset = 0; init = 0; op_A = 32'b00000000111111010000001011111101;
  end
  // clk generation
  initial         clk <= 0;
  always #(PERIOD/2) clk <= ~clk;

  initial begin // Reset the system,
       // Reset
       @ (negedge clk);
            reset = 1;
            @ (negedge clk);
            reset = 0;
       #(PERIOD*4)
       init = 0;
       @ (posedge clk);
       init = 1;
       #(PERIOD*2)
       init = 0;
       #(PERIOD*50);
  end
```

Figura 9: Testbench de la maquina de estados

Para este caso, se definieron las señales de entrada y de salida de la máquina de estados, incluyendo la instanciación de la máquina de estados y definiendo una señal de 32 bits establecida para evidenciar que el codigo estuviera funcionando correctamente, de igual manera, se definió el funcionamiento de las señales de init y reset para el testbench. A continuación se presenta la señal de result del código realizado (resaltando que fue necesario la negación de la señal de result para que fuera decodificada correctamente por Arduino).

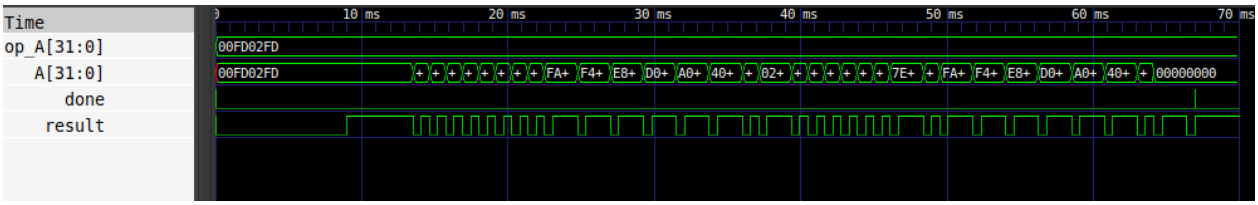

Figura 10: Testbench de la maquina de estados

Continuando, se realizó el diseño del periférico para poder implementar el código diseñado en la tarjeta. Para este caso se definieron las señales de lectura y escritura, la señal de chip-select y demás señales presentes también en la máquina de estados (entrada y salida). Posteriormente, se definieron registros y wires usados en el código, estableciendo el selector del multiplexor, los registros de entrada y de salida. Adicionalmente, se definieron las ubicaciones en memoria del decodificador de direcciones.

```
module peripheral_infra(clk , reset , d_in , cs , addr , rd , wr, d_out, d_outinfra);

  input clk;
  input reset;
  input [31:0] d_in;
  input cs;
  input [4:0]  addr;
  input rd;
  input wr;
  output d_out;
  output reg[31:0]d_outinfra;


reg [4:0] s;
reg [31:0] A;
reg init;
wire [31:0] result_infra;
wire done;

always @(*) begin
if (cs) begin
  case (addr)
    5'h04: s =  5'b00001; // A
    5'h08: s =  5'b00010; // result_infra
    5'h0C: s =  5'b00100; // init
    5'h10: s =  5'b01000; // result
    5'h14: s =  5'b10000; // done
    default: s = 5'b00000;
  endcase
end
else
  s = 5'b00000;
end
```

Figura 11: Código del periférico.

Finalmente, para el periférico, se definió la escritura de registros y se multiplexó las salidas del periférico.

```
always @(posedge clk) begin

  if(reset) begin
    init = 0;
    A    = 0;
  end
  else begin
    if (cs && wr) begin
                  A    = s[0] ? d_in    : A;
                  init = s[2] ? d_in[0] : init;
    end
  end

end


//----------

always @(posedge clk) begin
  if(reset)
    d_outinfra = 0;
  else if (cs && rd) begin
    case (s[4:0])
      5'b00010: d_outinfra    =  result_infra;
      5'b10000: d_outinfra    = {31'b0, done};
    endcase
  end
end
infra infra1 (
      .clk(clk),
      .reset(reset),
      .init(init),
      .op_A(A),
      .result(d_out),
      .result_infra(result_infra),
      .done(done)
);
```

Figura 12: Código del periférico.

Mediante el análisis de código implementado para el periférico, se determinó el siguiente diagrama de bloques en el cual se resume la interacción de las señales de entrada, salida e intermedias del mismo y la asignación correspondiente de las direcciones de memoria.

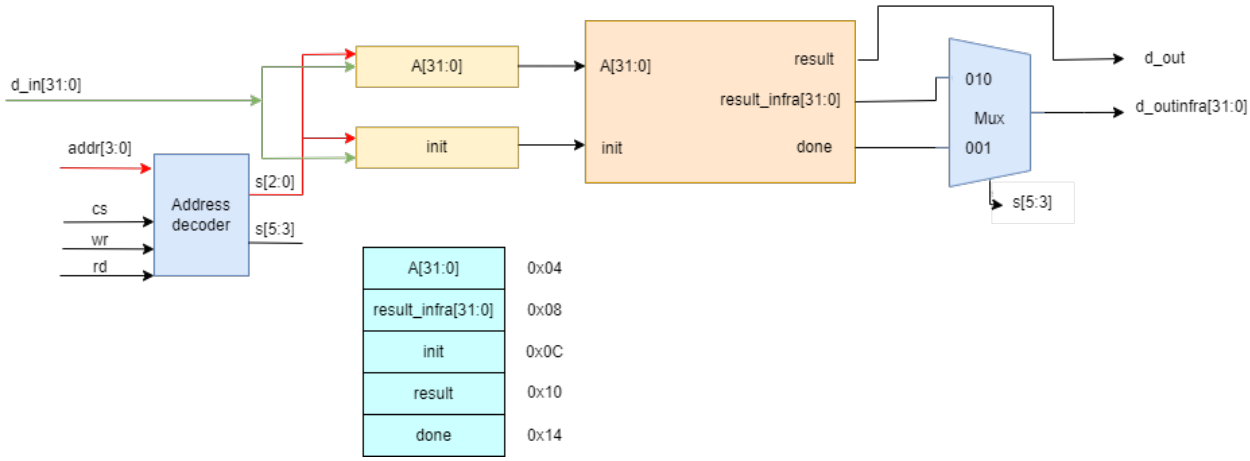

Figura 13: Diagrama de bloques del periférico.

Una vez comprobado que el periférico funcionaba adecuadamente por medio de su testbench, se instanció el mismo en el SOC con las señales definidas. Es importante recalcar, que se indicó en el encabezado del SOC la señal de salida, esto con el fin de establecer la señal de salida como uno de los pines de la tarjeta en el archivo .lpf.

```
peripheral_infra infra1 (
        .clk(clk),
        .reset(!resetn),
        .d_in(mem_wdata[31:0]),
        .cs(cs[2]),
        .addr(mem_addr[4:0]),
        .rd(rd),
        .wr(wr),
.d_outinfra(infra_dout),
.d_out(INFRA)
);
```

Figura 14: Instanciación en el SOC del periférico.

Posteriormente, se definió la parte de hardware del programa en Assembler. En este código, se cargó las ubicaciones en memoria definidas tanto en el SOC como en el código del periférico, estableciendo posteriormente las ubicaciones del puntero global, la escritura de la señal de entrada y se establecieron las señales de INIT y RESULT.

```
.equ INFRA_BASE,   0x430000
.equ INFRA_OP_A,   0x04
.equ INFRA_INIT,   0x0C
.equ INFRA_RESULT, 0x10
.equ INFRA_DONE,   0x14

.section .text
.globl infra_hw
.globl infra

infra_hw:
  li   gp, INFRA_BASE
  sw   a0, INFRA_OP_A(gp)
  li   a0, 1
  sw   a0, INFRA_INIT(gp)
  sw   zero, INFRA_INIT(gp)
.L0:
   li   t0, 1
   lw   t1, INFRA_DONE(gp)
   and  t1, t1, t0
   bnez t1, .L0
   lw   a0, INFRA_RESULT(gp)
   ret
```

Figura 15: Definición del periférico en Assembler.

Para complementar el funcionamiento del periférico, se diseñó un código en Assembler para la recepción del comando mediante la consola, considerando una dirección por defecto. Inicialmente, se instancia los espacios de memoria correspondientes a la tarjeta, y se define la dirección deseada (dirección + dirección inversa).

```
.equ IO_BASE, 0x400000
.equ IO_LEDS, 4
.equ IO_UART_DAT, 8
.equ IO_UART_CNTL, 16
.section .text
.globl start
.globl blinker
.globl put_hello

calculator:
    li gp,IO_BASE
    li sp,0x1000
    li s2,0xBF00 #direccion fija
    li s3,0x00FF #constante para negar

```

Figura 16: Definición de las variables e instanciación en memoria.

Luego, se emite un mensaje (“Enter command”) indicando que ya se puede teclear el comando, este se compone de cuatro números, que son leídos por parejas, y concatenados al final.

```
loop:
    ################# Primer digito
    la   a0, commando
    call putstring
    call getchar    #
    call putchar    # echo
    addi a0, a0, -0x30
    li a1,10 # Cargamos el x10
    call mult_hw # Llamamos la multiplicacion para decenas
    mv   a4, a0             # first operand

    ################# Segundo digito
    call getchar    #
    call putchar    # echo
    addi a3, a0, -0x30 # Segundo operando
    add a3,a4,a3 #sumar

    ################ Tercer digito ######################################
    call getchar
    call putchar
    addi a0, a0, -0x30
    li a1,10 # Cargamos el x10
    call mult_hw # Llamamos la multiplicacion para decenas
    mv a1,a0

    ################# Cuarto digito ############################
    call getchar    #
    call putchar    # echo
    addi a5, a0, -0x30 # Segundo operando
    add a5,a1,a5
```

Figura 17: Recepción del comando por consola.

Con el comando ya completo, se procede a hallar su forma negada a través de la compuerta XOR. Después, al ser corrido el comando a la izquierda ocho bits, se suma con el comando negado.

```
    sll a3,a3,4 #correr primer numero izquierda 4
    add a3,a3,a5 #completar comando

    xor a4,s3,a3 #comando negado
    sll a6,a3,8 #correr comando izquierda 8
    add a5,a6,a4 #comando + comando negado

    sll a7,s2,16 #correr direccion 16 iz
    add a5,a7,a5 #completo
```

Figura 18: Establecimiento de la salida completa.

Finalmente, se imprimen dos mensajes (“Transmitted” y “Command =”), indicando que la recepción ha sido completada y que el comando se está transmitiendo de forma correcta..

```
    ## CODIGO PARA RESULTADO########
    la   a0, mej
    call putstring
    la   a0, result
    call putstring
    ################################
    mv a0,a5 #copia
    call infra_hw
    mv a0,a3
    call bin_to_bcd
```

Figura 19: Transmisión de la salida y vista en consola.

A continuación se muestra la sección ".data", que incluye los mensajes que se usaron en la comunicación:

```
87 .section .data
88 mej:
89     .asciz "\n\rTransmitted! \n\r"
90 commando:
91     .asciz "\n\rEnter command \n\r"
92 result:
93     .asciz "\n\rCommand = "
```

Figura 20: Mensajes usados para la comunicación.

Presentamos un ejemplo de la recepción y transmisión del comando por medio de la consola:

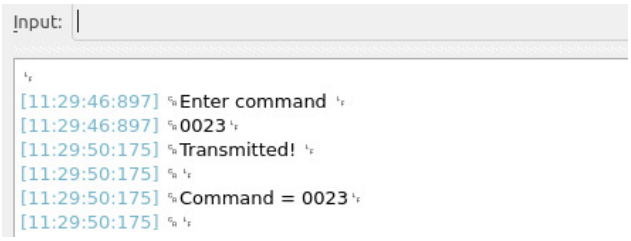

Figura 21: Ejemplo de recepción y transmisión por consola.

En cuanto al diseño del circuito del helicóptero, se planteó el siguiente diagrama de bloques:

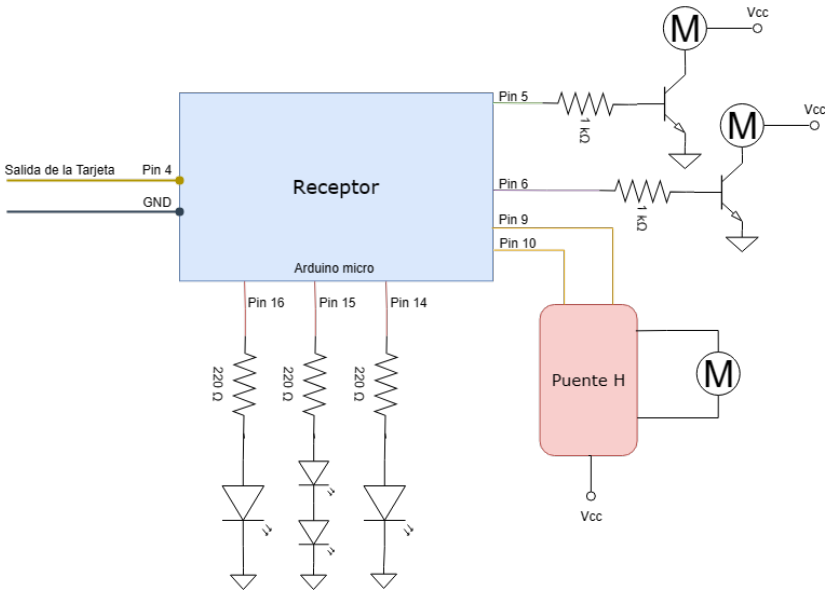

Figura 22: Diagrama del circuito del helicóptero.

Como receptor, se utilizó un Arduino Micro Pro debido a su tamaño reducido, y se implementó la librería IRremote, la cual permite interpretar los comandos transmitidos a través del protocolo NEC.

```
2  #define DECODE_NEC          // Includes Apple and Onkyo
3  #include <Arduino.h>
4
5  #include "PinDefinitionsAndMore.h"
6  #include <IRremote.hpp> // include the library
7
```

Figura 23: Librería implementada para recepción del protocolo NEC.

Las lecturas de la señal se realizan cuando el receptor está disponible y, en dado caso de ser una señal con protocolo NEC, se muestran en pantalla la dirección y el comando ya descifrados. A través de esta librería, es sencillo extraer el comando y asignarle una función en el código.

```
59    if (IrReceiver.decode()) {
60
61        IrReceiver.printIRResultShort(&Serial);
62        IrReceiver.printIRSendUsage(&Serial);
63
64        Serial.println();
65        IrReceiver.resume(); // Enable receiving of the next value
```

Figura 24: Instanciación de la recepción de datos.

4. Resultados

La primera prueba realizada fue conectando un analizador lógico a un control de televisión convencional, esto con el fin de evidenciar como eran los patrones de dirección y comando que enviaba para cada uno de los botones y así poderlos decodificar posteriormente. Los 32 bits determinados para cada uno de los botones del control de televisión fueron los siguientes:

00FB02FB		00FB24DB		00FBD12E		00FB51AB		00FB10EF		00FBA06F		00FB609F		00FB40DF		00FB50AF		00FBD02F	
IZQUIERDA		DERECHA		ARRIBA		ABAJO		BOTÓN 2		BOTÓN 8		BOTÓN 6		BOTÓN 4		BOTÓN 5		BOTÓN 0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0		0		0		0		0		0		0		0		0		0	
0		0		0		0		0		0		0		0		0		0	
0		0		0		0		0		0		0		0		0		0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0		0		0		0		0		0		0		0		0		0	
0		0		0		0		0		0		0		0		0		0	
0		0		0		0		0		0		0		0		0		0	
1	F	1	F	1	F	1	F	1	F	1	F	1	F	1	F	1	F	1	F
1		1		1		1		1		1		1		1		1		1	
1		1		1		1		1		1		1		1		1		1	
1		1		1		1		1		1		1		1		1		1	
1	B	1	B	1	B	1	B	1	B	1	B	1	B	1	B	1	B	1	B
1		1		1		1		1		1		1		1		1		1	
0		0		0		0		0		0		0		0		0		0	
1		1		1		1		1		1		1		1		1		1	
0	0	0	2	1	D	1	5	1	1	1	A	0	6	0	4	1	5	1	D
0		1		0		0		0		0		1		0		0		0	
0		0		1		1		0		0		1		1		1		1	
0		0		1		0		0		1		0		0		0		1	
0	2	0	4	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0		0		0		0		0		0		0		0		0		0	
1		1		0		1		0		0		0		0		0		0	
0		0		0		0		0		0		0		0		0		0	
1	F	1	D	0	2	0	A	1	E	1	6	1	9	1	D	0	A	1	2
1		0		1		1		1		1		0		1		1		0	
1		1		0		0		1		1		0		0		0		1	
1		1		0		1		1		0		1		1		1		0	
1	B	1	B	0	E	1	B	1	F	1	F	1	F	1	F	1	F	1	F
1		1		1		1		1		1		1		1		1		1	
0		0		1		0		1		1		1		1		1		1	
1		1		1		1		1		1		1		1		1		1	

Figura 25: Señales obtenidas del control remoto.

Teniendo las direcciones y comandos establecidos que se iban a utilizar, se comprobó que el código diseñado en Verilog/Assembler funcionara adecuadamente, evidenciando que enviara los mismos comandos y direcciones que el control de televisión y tuviera tiempos adecuados, representando que el protocolo NEC realizado fue adecuado.

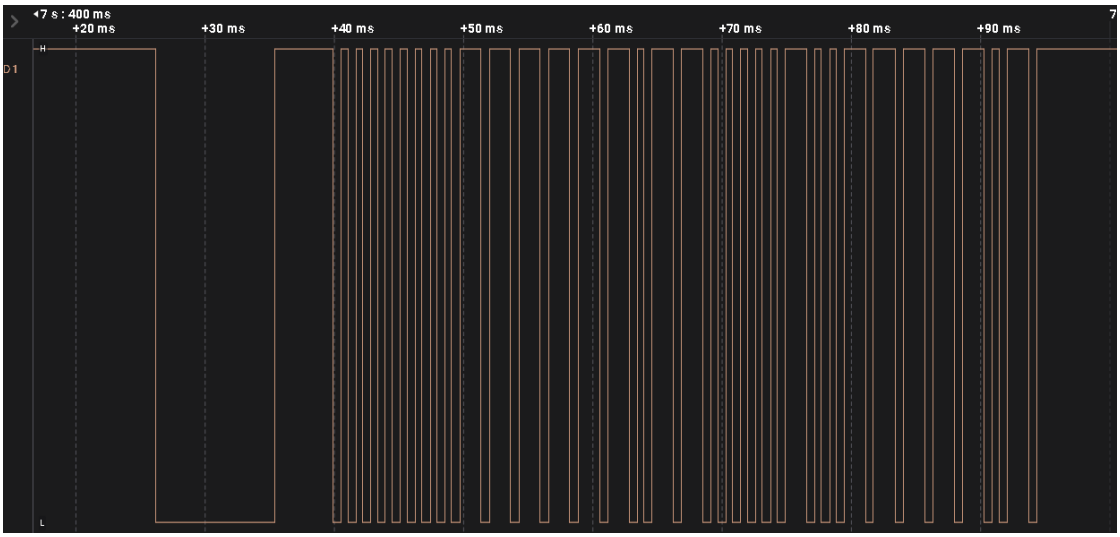

Figura 26: Señales obtenidas de la FPGA.

Cabe resaltar, que fue necesario negar la señal de salida de la FPGA, esto con el fin de que Arduino

lo decodificara correctamente.

A continuación, se presenta la prueba de decodificación de una señal de prueba enviada por la FPGA a través de Arduino, constando que la señal enviada por la FPGA es correcta y que está enviando los bits adecuados con sus respectivos comandos y direcciones.

```
Protocol=NEC Address=0xFD Command=0x17 Raw-Data=0xE81700FD 32 bits LSB first
23Send with: IrSender.sendNEC(0xFD, 0x17, <numberOfRepeats>);
```

Figura 27: Decodificación hecha con Arduino.

El circuito del helicóptero implementado físicamente quedo de la siguiente manera:

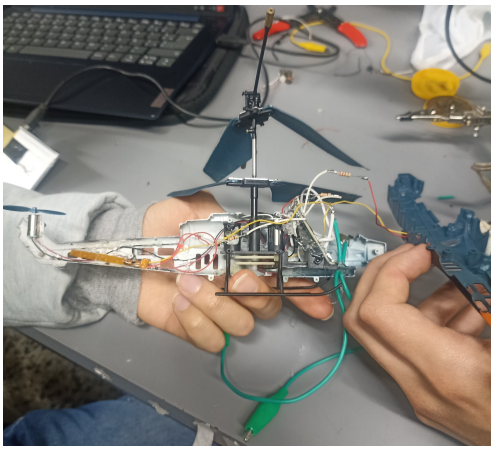

Figura 28: Circuito implementado.

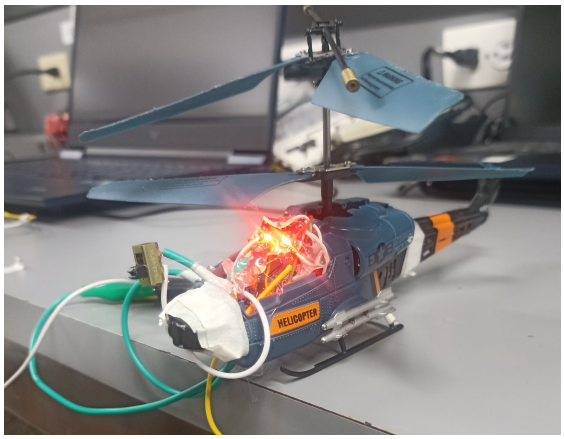

Figura 29: Diseño final.

El código implementado basado en la carpeta proporcionada para el curso se encuentra en el siguiente repositorio de GitHub, el cual es de acceso público: `https://github.com/JulianRobles/Implementaci-n-del-protocolo-NEC`.

5. Dificultades encontradas

La primera dificultad encontrada fue en la transmisión de la señal por medio del uso de diodos infrarrojos, realmente se buscaron múltiples formas de transmitir la señal, pero no se llegaba a decodificar correctamente, siendo un problema mayormente de los diodos usados y su limitación de potencia, ya que, la señal que le llegaba de la FPGA era adecuada. Para intentar transmitir la señal se planteó el siguiente circuito, el cual corresponde a un transistor polarizado como interruptor, por las limitaciones de corriente de la placa, pero que no funciono con éxito:

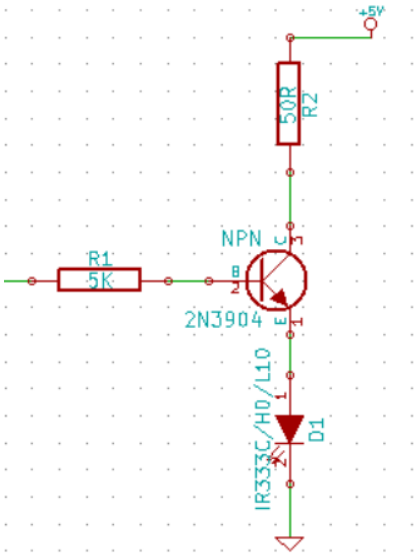

Figura 30: Circuito planteado para transmisión.

De igual manera, se cambió la posición del diodo, colocándolo en el colector del transistor, pero la señal recibida por el receptor tampoco era adecuada. Adicionalmente, se decidió comprar un módulo que trajera integrados los diodos infrarrojos, pero la señal recibida tampoco fue adecuada. Por lo cual se tomó la decisión de simplemente conectar por medio de un cable la tarjeta con el helicóptero para la recepción de señal.

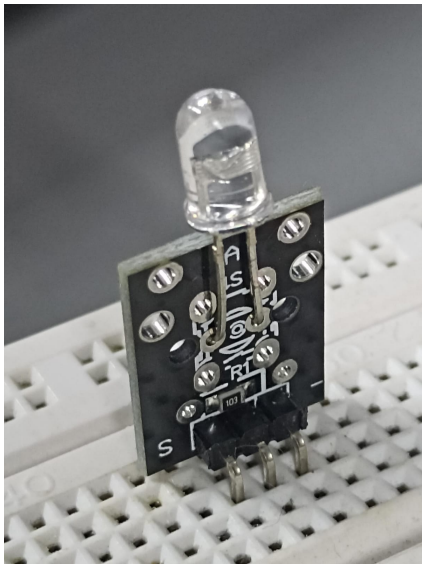

Figura 31: Módulo comprado para la transmisión.

Otro error que fue una limitante en cuanto a tiempo, fue la decodificación de la señal, no era

adecuada inicialmente, así fuera por cable directo, y luego de múltiples pruebas, se comprobó que era necesario negar la señal enviada por la FPGA y se solucionó este error, y junto con este error, habían ciertos problemas de tierra entre la FPGA y el programador secundario, lo cual no permitía enviar la señal correctamente, pero fueron solucionados.
Adicionalmente, se presentó un error a la hora de soldar el circuito del helicóptero, esto debido a que en uno de los motores DC que posee, se desoldó un cable interno, el cual no era accesible y no permitía el giro de las hélices del helicóptero, pero afortunadamente se pudo arreglar abriendo el motor.

Figura 32: Motor DC.

Finalmente, se puede indicar que a pesar de que el PWM diseñado para el helicóptero movía sus hélices a una velocidad considerable, este no fue capaz de ascender debido a que todo el circuito interno fue modificado y diseñado por nosotros, y aumento bastante el peso en la relación con el helicóptero original, debido principalmente al uso de una powerbank como fuente de tensión.

Referencias

[1] "Comunicación Por infrarrojos," ardubasic, `https://ardubasic.wordpress.com/2013/07/14/comunicacion-por-infrarrojos/` (accessed Dec. 6, 2023).

[2] "Irremote," IRremote - Arduino Reference, `https://www.arduino.cc/reference/en/libraries/irremote/` (accessed Dec. 6, 2023).