Proyecto:
Implementación del sistema de control de un robot de desplazamiento omnidireccional con ruedas Mecanum (Gyro)

Estudiantes: Jairo David Díaz Luna, Juan Ángel Pinzón López
Universidad Nacional de Colombia
Electrónica Digital I

1. Introducción

En el contexto actual de la automatización y la robótica, el desarrollo de sistemas de desplazamiento eficientes y versátiles representa un desafío clave para la ingeniería. Este proyecto se enfoca en la implementación del sistema de control de un robot de desplazamiento omnidireccional, diseñado para operar con un alto grado de maniobrabilidad en entornos restringidos. A través del uso de tecnologías digitales y estrategias de control preciso, se busca dotar al prototipo de la capacidad para moverse libremente en múltiples direcciones sin necesidad de realizar giros intermedios, optimizando así su desempeño en tareas dinámicas.

2. Arquitectura

El proyecto se compone de varias máquinas de estados finitos, integradas dentro de una maquina para controlar los distintos sistemas del robot.

2.1 Top-level maquina de estados: control de Gyro

Inicialmente esta maquina funciona con la activacion de un switch de accion rapida, donde con una configuracion PULL, se conecta a un input de la FPGA.

De modo que en esta maquina tambien se indica cuando lo demas funciona y no, con un IDLE y un WORK, siendo el WORK el estado donde Gyro funciona y entran las demas maquinas de estados.

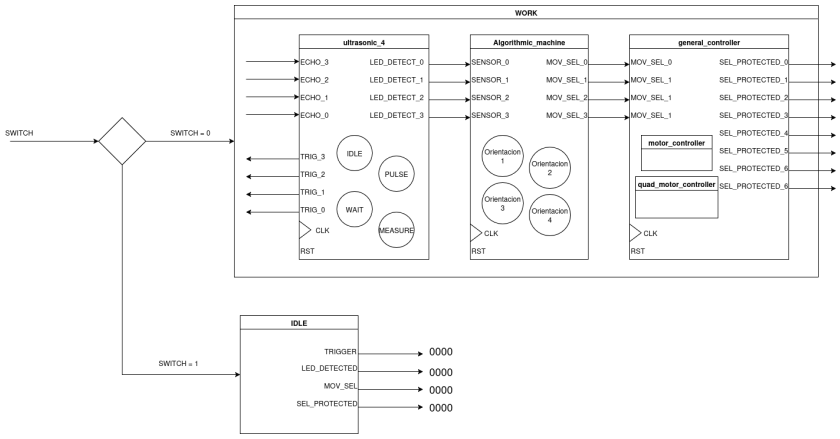

Figure 1: FSM funcionamiento de Gyro

2.2 Control movimiento de motores

2.2.1 Máquina de estados : Control de puente H

Esta máquina gestiona un puente H a partir de dos señales de entrada y dos de salida. Incluye una protección para evitar la activación simultánea de ambas entradas (estado 1-1), anulando esta combinación a 0-0.

```
module motor_controller(
    input clk,
    input rst,
    input wire [1:0] sel,
    output reg [1:0] sel_protected
);

    localparam IDLE                = 2'b00;
    localparam CLOCK_WISE          = 2'b01;
    localparam COUNTER_CLOCK_WISE  = 2'b10;
    localparam PROTECTION          = 2'b11;

    reg [1:0] fsm_state, next_state;

    always @(negedge clk or posedge rst) begin
        if (rst)
            fsm_state <= IDLE;
        else
            fsm_state <= next_state;
    end

    always @(*) begin
        case (fsm_state)
            IDLE, CLOCK_WISE, COUNTER_CLOCK_WISE, PROTECTION: next_state = sel;
            default:            next_state = IDLE;
        endcase
    end

    always @(*) begin
        if (fsm_state == PROTECTION)
            sel_protected = 2'b00;
        else
            sel_protected = sel;
    end

endmodule
```

Listing 1: Código Verilog - FSM Control de Puente H

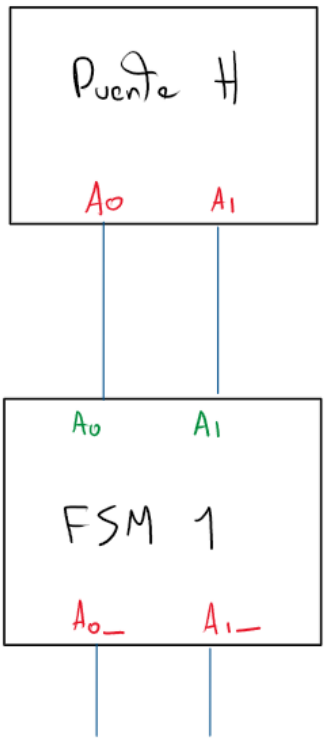

Figure 2: Máquina de estados para control de puente H

2.2.2 Máquina de estados 2: Ejecutora de movimientos

Recibe un input de 4 bits que define un movimiento preestablecido. Genera una salida de 8 bits, organizada en pares para controlar directamente los cuatro puentes H.

```
module move_machine(
    input clk,
    input rst,
    input wire [3:0] movement_sel,
    output reg [7:0] sel
);

    localparam IDLE                              = 4'b0000;
    localparam FORWARD                           = 4'b0001;
    localparam BACK                              = 4'b0010;
    localparam RIGHT                             = 4'b0011;
    localparam LEFT                              = 4'b0100;
    localparam RIGHT_UPPER_DIAGONAL              = 4'b0101;
    localparam RIGHT_DOWN_DIAGONAL               = 4'b0110;
    localparam LEFT_UPPER_DIAGONAL               = 4'b0111;
    localparam LEFT_DOWN_DIAGONAL                = 4'b1000;
    localparam RADIUS_VERT_ROT_CLOCKWISE         = 4'b1001;
    localparam RADIUS_VERT_ROT_COUNTERCLOCKWISE  = 4'b1010;
    localparam RADIUS_HORIZ_ROT_CLOCKWISE        = 4'b1011;
    localparam RADIUS_HORIZ_ROT_COUNTERCLOCKWISE= 4'b1100;
    localparam CENTER_ROT_CLOCKWISE              = 4'b1101;
    localparam CENTER_ROT_COUNTERCLOCKWISE       = 4'b1110;

    reg [3:0] fsm_state, next_state;

    // Logica de transicion de estado sincronizada
    always @(posedge clk or posedge rst) begin
```

```
28          if (rst)
29              fsm_state <= IDLE;
30          else
31              fsm_state <= next_state;
32      end
33
34      // Logica para determinar el proximo estado segun la entrada
35      always @(*) begin
36          case (movement_sel)
37              IDLE:                                 next_state = IDLE;
38              FORWARD:                              next_state = FORWARD;
39              BACK:                                 next_state = BACK;
40              RIGHT:                                next_state = RIGHT;
41              LEFT:                                 next_state = LEFT;
42              RIGHT_UPPER_DIAGONAL:                 next_state = RIGHT_UPPER_DIAGONAL
                    ;
43              RIGHT_DOWN_DIAGONAL:                  next_state = RIGHT_DOWN_DIAGONAL;
44              LEFT_UPPER_DIAGONAL:                  next_state = LEFT_UPPER_DIAGONAL;
45              LEFT_DOWN_DIAGONAL:                   next_state = LEFT_DOWN_DIAGONAL;
46              RADIUS_VERT_ROT_CLOCKWISE:            next_state =
                    RADIUS_VERT_ROT_CLOCKWISE;
47              RADIUS_VERT_ROT_COUNTERCLOCKWISE: next_state =
                    RADIUS_VERT_ROT_COUNTERCLOCKWISE;
48              RADIUS_HORIZ_ROT_CLOCKWISE:           next_state =
                    RADIUS_HORIZ_ROT_CLOCKWISE;
49              RADIUS_HORIZ_ROT_COUNTERCLOCKWISE:next_state =
                    RADIUS_HORIZ_ROT_COUNTERCLOCKWISE;
50              CENTER_ROT_CLOCKWISE:                 next_state = CENTER_ROT_CLOCKWISE
                    ;
51              CENTER_ROT_COUNTERCLOCKWISE:          next_state =
                    CENTER_ROT_COUNTERCLOCKWISE;
52              default:                              next_state = IDLE;
53          endcase
54      end
55
56      // Logica de salida para cada estado de la FSM
57      always @(*) begin
58          case (fsm_state)
59              IDLE:                                  sel = 8'b0000_0000; // Motores
                    apagados
60              FORWARD:                               sel = 8'b0101_0101; // Avanzar
                    recto
61              BACK:                                  sel = 8'b1010_1010; //
                    Retroceder recto
62              RIGHT:                                 sel = 8'b0110_1001; // Girar
                    derecha
63              LEFT:                                  sel = 8'b1001_0110; // Girar
                    izquierda
64              RIGHT_UPPER_DIAGONAL:                  sel = 8'b0100_0001; // Diagonal
                    sup. derecha
65              RIGHT_DOWN_DIAGONAL:                   sel = 8'b0010_1000; // Diagonal
                    inf. derecha
66              LEFT_UPPER_DIAGONAL:                   sel = 8'b1000_0010; // Diagonal
                    sup. izquierda
67              LEFT_DOWN_DIAGONAL:                    sel = 8'b1001_0100; // Diagonal
                    inf. izquierda
68              RADIUS_VERT_ROT_CLOCKWISE:             sel = 8'b0001_0001; // Rotar
                    vertical, reloj
69              RADIUS_VERT_ROT_COUNTERCLOCKWISE:  sel = 8'b0100_0100; // Rotar
                    vertical, antireloj
70              RADIUS_HORIZ_ROT_CLOCKWISE:            sel = 8'b0000_0101; // Rotar
                    horizontal, reloj
71              RADIUS_HORIZ_ROT_COUNTERCLOCKWISE: sel = 8'b1001_1010; // Rotar
```

```
                horizontal, antireloj
72              CENTER_ROT_CLOCKWISE:               sel = 8'b1001_1001; // Giro
                    centro, reloj
73              CENTER_ROT_COUNTERCLOCKWISE:        sel = 8'b0110_0110; // Giro
                    centro, antireloj
74              default:                            sel = 8'b0000_0000; // Seguridad
75          endcase
76      end
77
78  endmodule
```

Listing 2: Código Verilog - FSM Control de movimientos

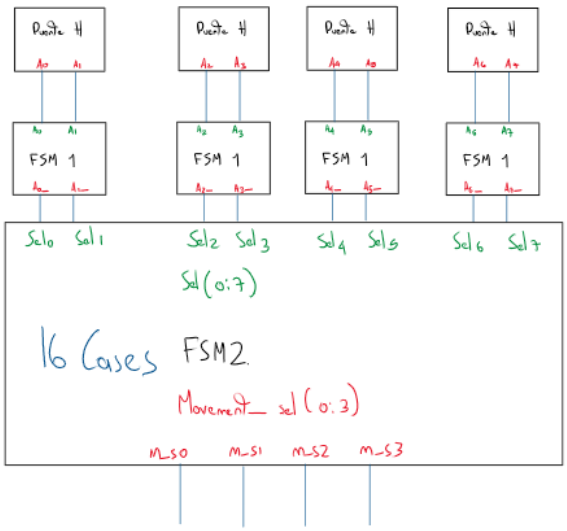

Figure 3: Máquina de estados ejecutora de movimientos

2.2.3 Mid-level: control general de los motores

Aquí se integran las dos maquinas de estado anteriores

```
1       module general_controller (
2       input clk,
3       input rst,
4       input [3:0] movement_sel,  // Entrada de movimiento
5       output [7:0] sel_protected  // Salida protegida de los motores
6   );
7
8       // signals intermedias para conectar la salida de move_machine a
            motor_controller
9       wire [7:0] sel;
10
11      // Instanciar el modulo move_machine (motor_controller)
12      move_machine move_machine_instance (
13          .clk(clk),
14          .rst(rst),
15          .movement_sel(movement_sel),
16          .sel(sel)
17      );
18
19      // Instanciar el modulo quad_motor_controller
20      quad_motor_controller quad_motor_instance (
21          .clk(clk),
22          .rst(rst),
```

```
        .sel(sel),              // Conectamos la salida de move_machine a la entrada de quad_motor_controller
        .sel_protected(sel_protected) // Salida protegida de los motores
    );

endmodule
```

Listing 3: Código Verilog - controlador general de los motores

2.3 Control de los sensores ultrasonicos HC-SR04

2.3.1 Maquina de estados 3: control de un sensor ultrasonido

Esta FSM gestiona los pines `Echo` y `Trigger` para un sensor, y entrega una salida binaria que indica si un objeto está a una distancia menor o igual a, por ejemplo, 15 cm.

```

module ultra_controller #(
  parameter CLOCK_FREQ        = 50_000_000, //clk de la FPGA que trabaja a 50 MHz
            SOUND_SPEED       = 343, // Usado para calcular la distancia o el set point del sensor
            DETECT_CM         = 15, //distancia que queremos medir
            TRIG_US           = 10, //tiempo que se manda el trigger al sensor
            INTERVAL_MS       = 50, // cada cuanto se manda el pulso de trigger al sensor
            WAIT_TIMEOUT_US   = 30_000,
            COUNT_TIMEOUT_US  = 50_000
)(
  input  wire       clk,
  input  wire       rst,
  input  wire       echo_i,
  output reg        trigger_o,
  output reg        object_detected,
  output reg        timeout_error
);

  // clock-cycle counts
  localparam integer TRIG_CYCLES = (CLOCK_FREQ/1_000_000) * TRIG_US; //ciclos necesarios para mantener el trigger en 1
  localparam integer INTERVAL_CYCLES     = (CLOCK_FREQ/1_000)     * INTERVAL_MS; //ciclos entre cada disparo del trigger
  localparam integer WAIT_TIMEOUT_CYCLES = (CLOCK_FREQ/1_000_000) * WAIT_TIMEOUT_US; //ciclos que espera la FPGA antes de que llege el primer posedge de echo
  localparam integer COUNT_TIMEOUT_CYCLES= (CLOCK_FREQ/1_000_000) * COUNT_TIMEOUT_US; // ciclos limite que espera la FPGA antes de que arroje un timeout error
  localparam integer DETECT_CYCLES       = (CLOCK_FREQ * DETECT_CM * 2)/ (SOUND_SPEED * 100); // ciclos correspondientes a la distancia establecida (basicamente el set-point)

  // estados de la FSM
  localparam [1:0]
    IDLE    = 2'd0,
    PULSE   = 2'd1,
    WAIT    = 2'd2,
    MEASURE = 2'd3;

  reg [1:0]   state;
  reg [31:0]  cnt_int, cnt_trig, cnt_wait, cnt_echo, measured_echo;
  reg         echo_s0, echo_s1;
  wire        echo_rise =  echo_s0 & ~echo_s1; // detecta el flanco de subida
```

```
37  wire        echo_fall = ~echo_s0 &  echo_s1; // detecta el flanco de bajada
38
39  always @(posedge clk or posedge rst) begin //aqui empieza la FSM
40    if (rst) begin //si rst esta en alto toda la maquina de estados esta quieta
          y todo queda en 0
41      state           <= IDLE;
42      cnt_int         <= 0;
43      cnt_trig        <= 0;
44      cnt_wait        <= 0;
45      cnt_echo        <= 0;
46      measured_echo   <= 0;
47      trigger_o       <= 0;
48      object_detected <= 0;
49      timeout_error   <= 0;
50      echo_s0         <= 0;
51      echo_s1         <= 0;
52    end else begin
53      // sinc. echo_i
54      echo_s0 <= echo_i;
55      echo_s1 <= echo_s0;
56
57      case (state)
58        //---------------------------------
59        IDLE: begin
60          trigger_o <= 0;
61          if (cnt_int < INTERVAL_CYCLES)
62            cnt_int <= cnt_int + 1;
63          else begin
64            cnt_int         <= 0;
65            cnt_trig        <= 0;
66            object_detected <= 0;
67            timeout_error   <= 0;
68            state           <= PULSE;
69          end
70        end
71
72        //---------------------------------
73        PULSE: begin
74          trigger_o <= 1;
75          if (cnt_trig < TRIG_CYCLES-1) // Cuenta hasta llegar al ciclo de
              trigger
76            cnt_trig <= cnt_trig + 1;
77          else begin // cuando el contador llega  los ciclos de trigger -1 pasa
                al siguiente estado y resetea el contador de trigger y de wait
78            cnt_trig  <= 0;
79            cnt_wait  <= 0;
80            trigger_o <= 0; //manda el trigger a 0 por que ya se mando la
                signal
81            state     <= WAIT;
82          end
83        end
84
85        //---------------------------------
86        WAIT: begin
87          if (echo_rise) begin // cuando echo_rise es 1 hace lo siguiente
88            cnt_echo <= 0; // restea echo si en dado caso tenia algun valor
                guardado
89            state    <= MEASURE;
90          end
91          else if (cnt_wait < WAIT_TIMEOUT_CYCLES)
92            cnt_wait <= cnt_wait + 1; //cuenta hasta los ciclos de imeout en
                espera
93          else begin
```

```
94              timeout_error <= 1; //arroja error y devuelve a IDLE, pero nunca
                    para
95              state         <= IDLE;
96            end
97          end
98
99          //---------------------------------
100         MEASURE: begin
101           if (echo_fall) begin // cuando ya se recibio todo pulso de echo.
102             measured_echo   <= cnt_echo;
103             object_detected <= (cnt_echo <= DETECT_CYCLES);
104             state           <= IDLE;
105           end
106           else if (cnt_echo < COUNT_TIMEOUT_CYCLES)
107             cnt_echo <= cnt_echo + 1;
108           else begin
109             measured_echo   <= COUNT_TIMEOUT_CYCLES;
110             object_detected <= 0;
111             timeout_error   <= 1;
112             state           <= IDLE;
113           end
114         end
115
116         default: state <= IDLE;
117       endcase
118     end
119   end
120
121 endmodule
122 endmodule
```

Listing 4: Código Verilog - FSM Sensor HCSR04

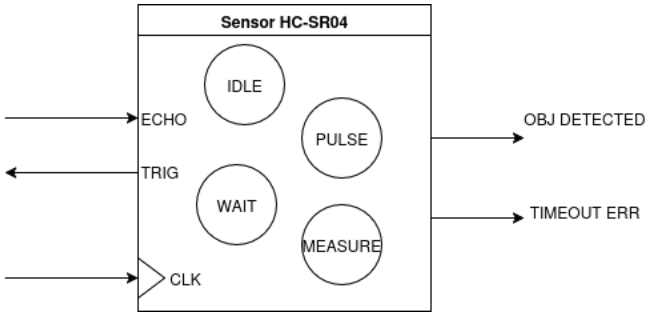

Figure 4: Máquina de estados para el sensor ultrasónico

2.3.2 Mid-level: instancia para cuatro sensores ultrasonicos

como son 4 sensores se hizo un codigo donde se instancia 4 veces ultra_controller llamado ultrasonic_4 donde por medio de un ciclo for se crean las 4 estancias para los sensores.

```
1
2  module ultrasonic_4(
3    input  wire        clk,
4    input  wire        rst,
5    input  wire [3:0]  echo_i,          // echo inputs para los 4 sensores
6    output wire [3:0]  trigger_o,       // trigger outputs para los 4 sensores
7    output wire [3:0]  led_detect       // 1 LED por cada sensor si detecta un
         objeto
8  );
9
10   genvar i; // genera la variable i
```

```
  generate
    for (i = 0; i < 4; i = i + 1) begin : U_SENSOR
      wire obj_det;
      wire to_err;

      ultra_controller #(
        .CLOCK_FREQ       (50_000_000),
        .SOUND_SPEED      (343),
        .DETECT_CM        (15),
        .TRIG_US          (10),
        .INTERVAL_MS      (50),
        .WAIT_TIMEOUT_US  (30_000),
        .COUNT_TIMEOUT_US(50_000)
      ) u_ultra (
        .clk              (clk),
        .rst              (rst),
        .echo_i           (echo_i[i]),
        .trigger_o        (trigger_o[i]),
        .object_detected  (obj_det),
        .timeout_error    (to_err)
      );

      // Mapea el led a object_detected
      assign led_detect[i] = obj_det;
    end
  endgenerate

endmodule
```

Listing 5: Código Verilog - Mid-level control 4 sensores

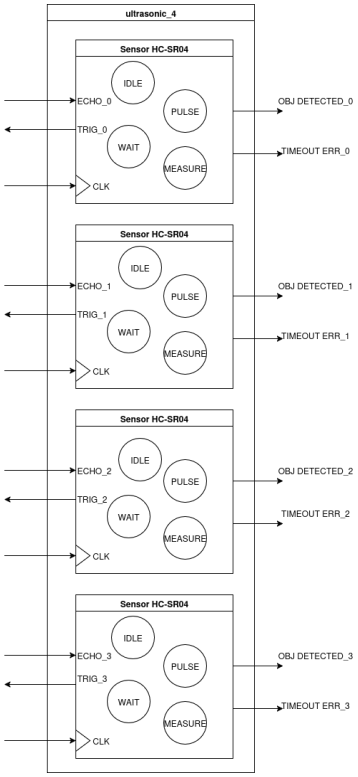

Figure 5: Diagrama de estados controlador de los 4 sensores utrasonicos

2.4 Máquina de estados 4: Resolución de laberintos

Implementa un algoritmo de navegación que decide el movimiento según los obstáculos detectados. Toma como entradas 4 bits provenientes de sensores ultrasónicos y entrega una salida de 4 bits, usada por la máquina 2.

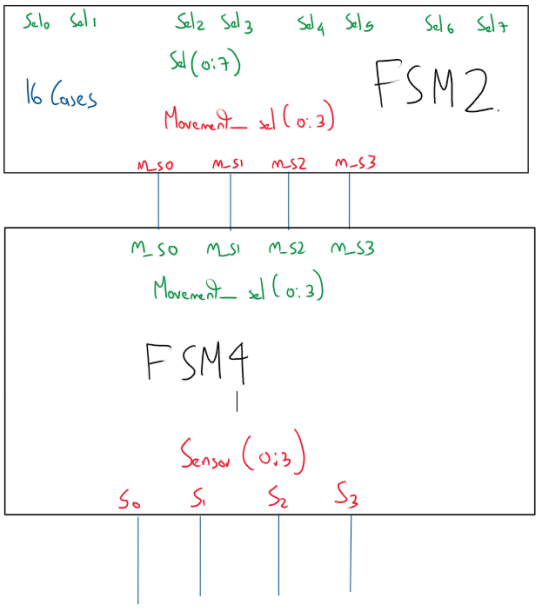

Figure 6: Máquina de estados para resolución de laberintos

3. Uso de la IA

El uso de la IA fue útil como asistente y corrector de sintaxis, ya que al usar solo maquinas de estado básicas, básicamente fue editar una plantilla a nivel de código y ajustarla de acuerdo a la maquina de estados planteada. Irónicamente esta actividad sirvió como ejemplo de como la IA no es tan capaz ya que ni fue capaz de entender el diagrama de estados de la maquina de estados que se encarga del movimiento automático, siendo esta maquina con 17 estados y un cambio recurrente de orientación.

4. Conclusiones

Si bien, al poder finalizar el proyecto de forma exitosa demuestra que es posible usar FPGAS para control de robots, para la creación de algoritmos de búsqueda de salidas de laberintos resulta mucho mas complejo de lo que aparenta, ya que en este algoritmo, que de por si es sencillo, resulta engorroso tratar con el cambio de orientación, por lo que algún algoritmo mas robusto debe tener en cuenta esta misma lógica y por ende, maquinas de estados con demasiados estados, difícil de planear a mano.

En este proyecto se prefirió usar solo 5 estados de movimiento (IDLE, FORWARD, BACK, LEFT, RIGHT) y no los 17 planteados ya que para un giro respecto a un eje se debe temporizar o identificar cuando se realizo el giro a 90 grados respecto al punto inicial, ademas de que debe ser preciso ya que pequeñas desviaciones de ángulos harían a Gyro torpe o incapaz de ir en linea recta, añadiendo también que, la gracia de usar un robot Mecanum es poder cambiar de orientación sin rotar.

Dicho lo anterior, también se puede añadir a la conclusión que de la manera en que se plantearon mas salidas, se podría adaptar un control manual o bluetooth para controlar el robot, de modo que en caso de continuar el proyecto, se puede reutilizar código cambiando solo una maquina de estados.

5. Bibliografía

- Wikipedia, *Mecanum wheel control principle*, [En línea]. Disponible en: `https://en.wikipedia.org/wiki/File:Mecanum_wheel_control_principle.svg`. [Accedido: 27-jun-2025].