UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Sistemas Electrónicos Digitales Avanzados

PRÁCTICA FINAL

Diseño de SoC para Control de Motores

Presentado por:

Ihor Yasnyy

Curso 2020/21

Índice

Introducción

En esta práctica se desarrollará un System on Chip (SoC) para una aplicación de control de un motor de corriente continua, empleado la placa Zedboard Zynq.

El motor, cuya velocidad teórica máxima es de 789 rpm, se alimentará a una tensión de 12 V y dispone de una reductora con encoder magnético fijado al eje del motor para determinar la posición del eje. Este se encarga de generar dos señales en cuadratura, proporcionando 12 flancos por vuelta, lo que permitirá determinar el sentido de giro además de la velocidad.

Para controlar el motor se utilizará un módulo de expansión PmodHB5, que incorpora un puente en H encargado de controlar el sentido de giro. Además, integra circuitos para adecuar las señales procedentes del encoder.

El puente en H se controla mediante una señal de PWM cuyo ciclo de trabajo determinará la velocidad y otra señal que determinará el sentido de giro. Cuando el sentido de giro es 0, están conduciendo los transistores A y D, mientras si el sentido es 1, conducen los transistores B y C. Debido a que los transistores presentan un tiempo de conmutación no nulo, si durante este tiempo la señal PWM está activa, los cuatro transistores podrían estar conduciendo por lo que se produciría un cortocircuito, lo que puede dañar el dispositivo. Es por ello, que se deberá de tener en cuenta los tiempos de conmutación a la hora de diseñar el generador de PWM.

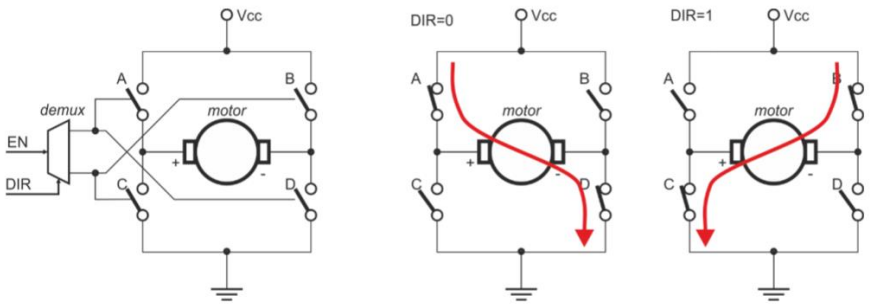

Figura 1: Esquema del funcionamiento del Puente en H

Desarrollo de los IP

Para adecuar la plataforma HW a las necesidades de la aplicación, se desarrollarán dos IPs a medida con ayuda del asistente de Vivado.

Los IPs constarán de dos ficheros VHDL, el primero (Top) corresponde al módulo de mayor jerarquía en el que se incluyen los parámetros y genéricos del módulo completo además de los elementos asociados al interface que se desea añadir, mientras que el segundo (Adapter), de menor jerarquía, contiene la declaración del diseño del usuario, la lógica necesaria para implementar los bancos de registros y acceder a ellos desde el bus AXI.

La aproximación de la solución diseñada por el usuario (Core), se incluirá como un módulo de menor jerarquía dentro de los dos ficheros descritos anteriormente, para ello estos deberán de ser modificados acorde al core a implementar y a los registros deseados.

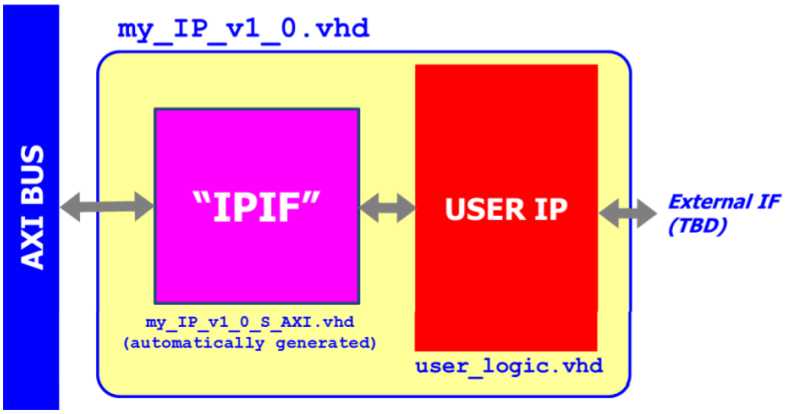

Figura 2: Diagrama de creación de un IP

Generador PWM

El primer periférico a desarrollar es el generador PWM. El core del módulo se denominará pmod5HB_pwm.vhd, este a su vez incluirá dos submódulos que harán más fácil su implementación y su simulación mediante Vivado o QuestaSim. Estos submódulos se denominarán pwm_generator.vhd, encargado de generar la señal de PWM y deadtime_cont.vhd, que implementará un contador de tiempos muertos para no dañar el dispositivo a la hora de conmutar de dirección de giro.

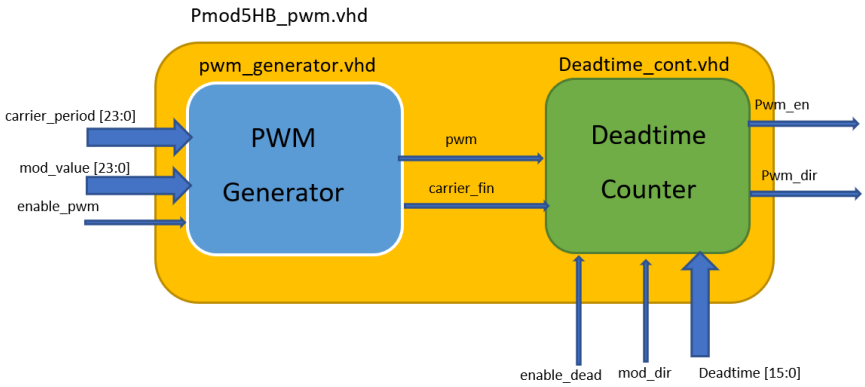

Figura 3: Diagrama de bloques del periférico generador PWM

Generador de Pulsos

El generador de PWM presentará las siguientes señales de entrada:

- carrier_period. Señal mediante la cual se fija el comienzo del contador descendente.
- mod_value. Señal mediante la cual se fija el ciclo de trabajo del PWM.
- enable_pwm. Señal de habilitación del generador PWM.

Para evitar la aparición de glitches en la señal PWM, se implementará un registro shadow denominado mod_value_reg para almacenar el valor de mod_value, este valor se actualizará al comienzo de cada periodo, pudiéndose detectar con la señal carrier_fin.

Para la generación de la señal PWM se implementará un contador descendente desde el valor fijado mediante carrier_period hasta 0. La señal de salida pwm estará a nivel bajo hasta que el valor del contador sea igual o menor al valor registrado de la señal mod_value, poniéndose en este caso a nivel alto. Se realizará de esta manera para que posteriormente el generador de tiempos muertos deshabilite el mínimo periodo de la señal PWM.

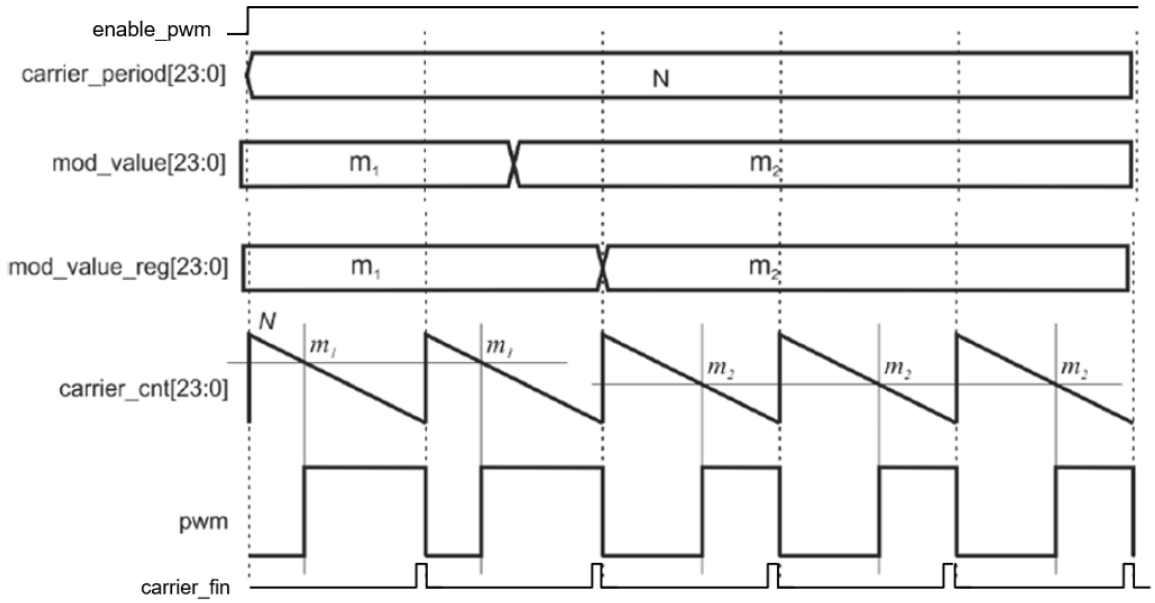

Figura 4: Generación de la señal PWM

Generación de Tiempos Muertos

El bloque generador de tiempos muertos recibirá las siguientes entradas:

- deadtime. Señal encargada de establecer el tiempo en alto de la señal dt_enable.
- enable_dead. Señal de habilitación del bloque generador de tiempos muertos.
- pwm_in. Señal de PWM que se recibe bel bloque pwm_generator.
- carrier_fin. Señal de fin de periodo.
- mod_dir. Señal que representa el sentido de giro deseado, introducido por el usuario.

A partir de estas entradas, en el bloque se crearán internamente otras señales adicionales con el objetivo de generar las salidas deseadas. Una de ellas es prev_dir que corresponde al valor registrado de mod_dir, su valor se actualiza cada vez que finaliza un periodo. Otra de estas señales es detect_start que corresponde a una señal de detección de flanco de subida y bajada de prev_dir.

Las señales dt_enable se activará cuando se detecte que detect_start y carrier_fin están a nivel alto, además, con esta condición se actualiza el valor de la señal que determina el sentido de giro, dir, tomando el valor de prev_dir. La señal dt_enable tendrá la duración establecida mediante la señal deadtime, mientras que dir mantendrá su valor hasta volver a ser actualizado.

La señal de salida pwm_out se generará mediante una puerta and a partir de la señal de pwm_in y la señal negada de dt_enable, de esta forma se logrará transmitir los pulsos PWM generados en el bloque anterior hasta que se detecte un cambio de sentido de giro, en este caso la salida tomará el valor de 0.

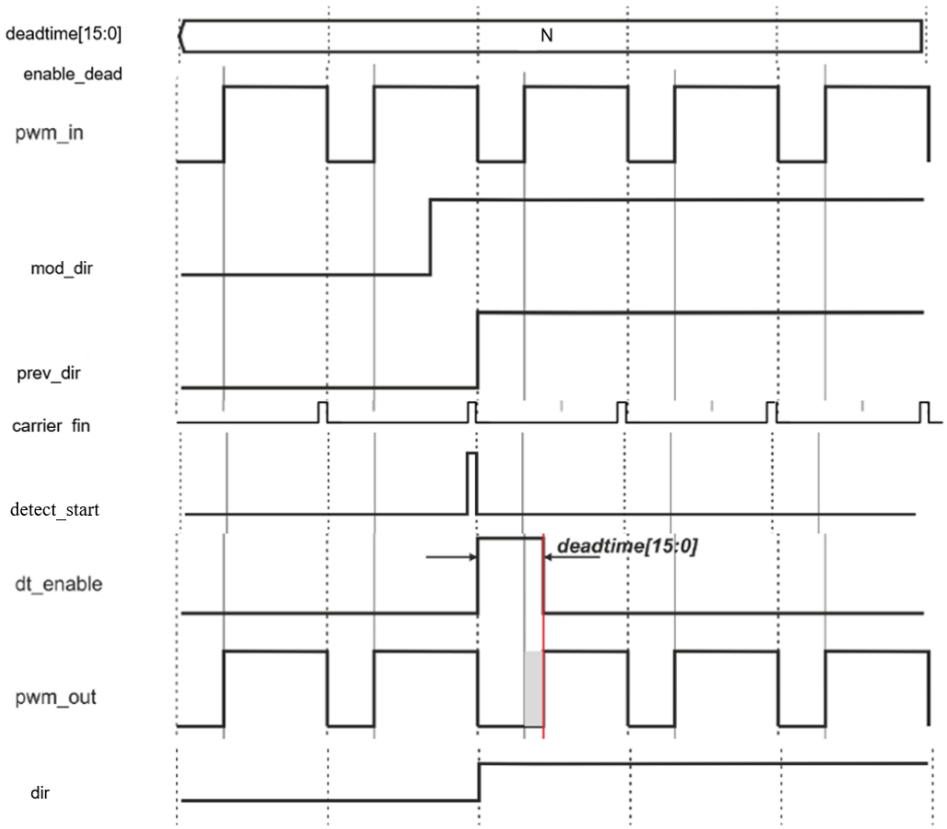

Figura 5: Generación de tiempos muertos y señales de salida

Simulación completa

Para la verificación del core se creó un nuevo proyecto en Vivado y se añadió los ficheros que contenían el diseño del generador pwm, el generador de los tiempos muertos, un módulo global que integra los dos bloques anteriores en un único diseño y el fichero de simulación (Test Bench). Las siguientes imágenes corresponden con una simulación ideal sin realizar la implementación ni la síntesis el diseño. Se escogió esta manera debido a que se ven mejor las señales y se puede llegar a entender el diseño de una manera más rápida. Aunque posteriormente se hizo una simulación post síntesis.

La Figura 6 es una vista general de la simulación, que permite observar el funcionamiento del dispositivo y que no presenta errores obvios.

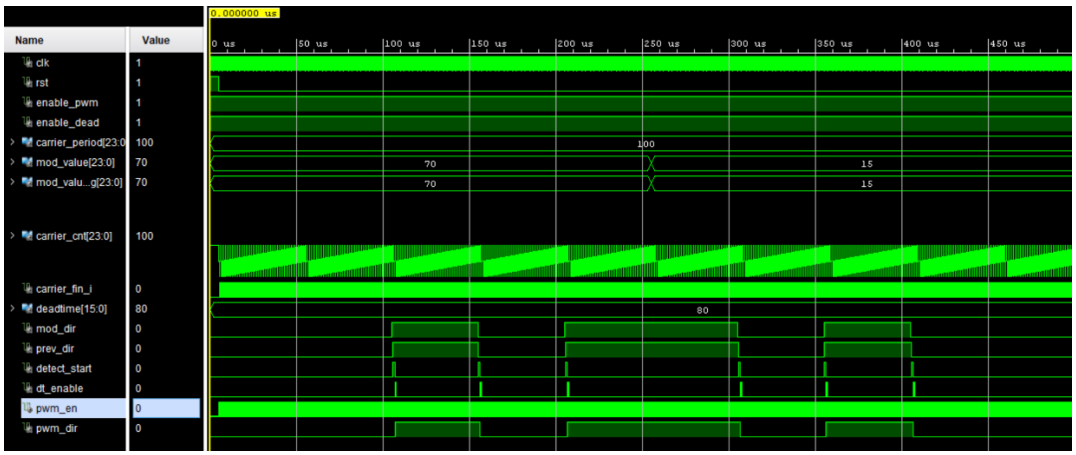

Figura 6: Simulación mediante Vivado del bloque pmod5HB_pwm

A continuación, se amplió la simulación en cualquier tramo para verificar el funcionamiento del módulo encargado de la generación de los pulsos PWM.

En la Figura 7 además de la señal pwm_en se aprecia la generación señal de carrier_fin que indica la finalización de un periodo, como con ella se actualiza el valor de mod_value_reg y la forma de onda del contador descendente carrier_cnt.

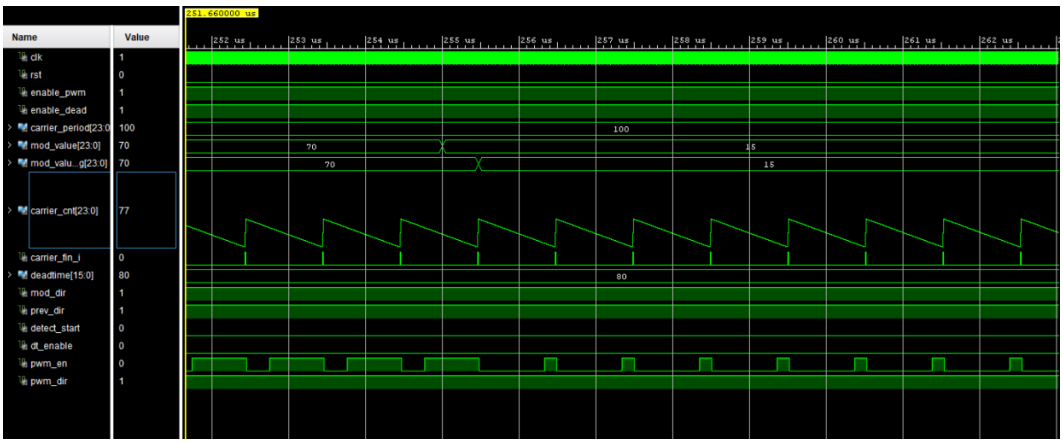

Figura 7: Simulación ampliada para observar la actualización de los registros y de la señal PWM

Desplazando la simulación al primer cambio de sentido de giro, se obtiene la vista de la Figura 8.

En la que se aprecia como el valor de prev_dir se actualiza con el siguiente flanco de la señal de fin de periodo tras detectar un cambio en mod_dir. Al modificarse el valor de prev_dir se detecta su flanco de subida mediante la señal detect_start y en el siguiente flanco de carrier_fin, al coincidir esta señal y detect_start a nivel alto, se produce la activación de dt_enable. Lo que a su vez provoca la activación del contador de tiempos muertos, que al finalizar conmuta la señal dt_enable permitiendo la salida de los pulsos PWM.

Se aprecia como durante el periodo en el que dt_enable se encuentra a nivel alto, la señal PWM está a 0, además, al inicio se conmuta la señal de salida de dirección de giro cuando los pulsos PWM están inactivos.

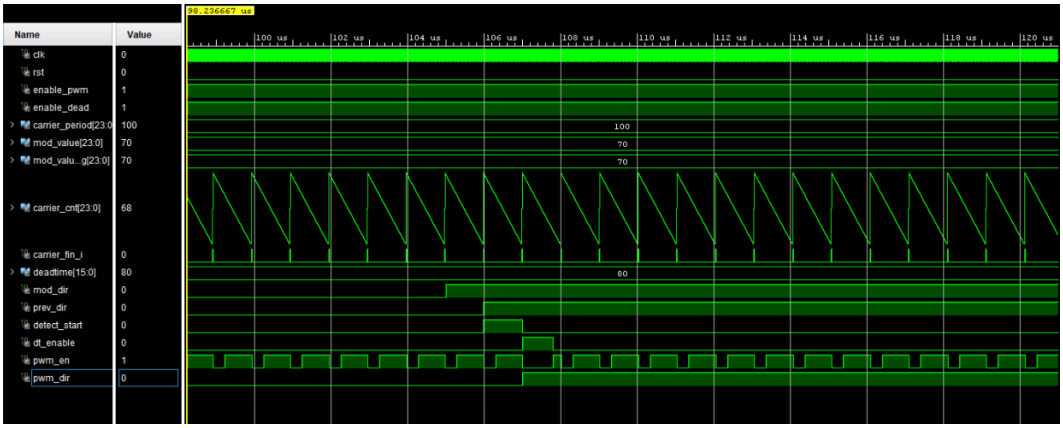

Figura 8: Parte de la simulación en la que aparece la primera conmutación de sentido de giro

Tras volver a desplazar la simulación al siguiente cambio de sentido, se obtiene la vista de la Figura 9. En la que se puede observar que el diseño también detecta el cambio de dirección en sentido contrario e implementa correctamente los tiempos muertos.

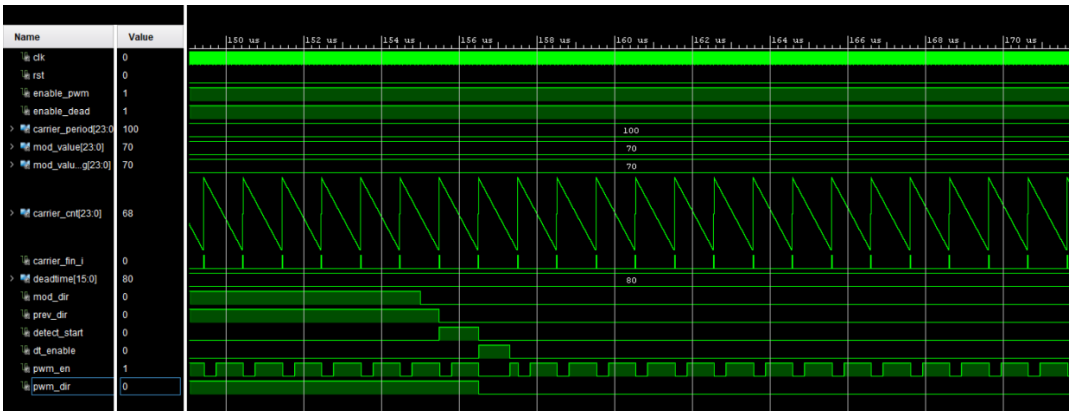

Figura 9: Parte de la simulación en la que aparece la siguiente conmutación de sentido de giro

Creación del IP

Tras la comprobación del funcionamiento del diseño mediante la simulación, se procede a generar el IP a medida a través de Vivado, dotando al IP de cuatro registros para su manejo, como aparece en la siguiente tabla.

Nombre	Offset	Descripción
control_reg	0x0	Registro de control.
period_reg	0x4	Registro de periodo.
mod_reg	0x8	Registro de ciclo de trabajo (moduladora).
dead_time_reg	0xC	Registro de tiempo muerto.

Figura 10: Tabla de registros del periférico axi_pwm_generator

Cabe destacar que del registro de control se emplearán los tres primeros bits.

- El bit 0 será mediante el cual se controle el reset del IP
- El bit 1 para para habilitar el PWM
- El bit 2 para habilitar los tiempos muertos

Una vez creada la estructura, el asistente generará los módulos Top y Adapter y a continuación se deberá añadir el core e instanciarlo en los dos ficheros anteriores.

En el Adapter aparece toda la lógica de adaptación que permite implementar el interface AXI Lite con los bancos de registros. Sobre esta estructura se deberá de instanciar el módulo pmod5HB_pwm, añadir como puertos de salida las señales pwm_en y pwm_dir y declarar las señales internas para conectar la instanciación del componente con los registros. A continuación, se puede observar la interconexión entre el core y el Adapter respetando los offset.

```
pwm_rst          <= not(S_AXI_ARESETN) or slv_reg0(0);
 pwm_enable       <= slv_reg0(1);
 dead_enable      <= slv_reg0(2);
 pwm_mod_dir      <= slv_reg1(31);
 pwm_carrier_period <= slv_reg1(27 downto 4);
 pwm_mod_value    <= slv_reg2(31 downto 8);
 pwm_deadtime     <= slv_reg3(27 downto 12);

  Conexion_PWM : pmod5HB_pwm
     port map(
        clk          => S_AXI_ACLK,
        rst          => pwm_rst,
        enable_pwm   => pwm_enable,
        enable_dead  => dead_enable,
        carrier_period => pwm_carrier_period,
        mod_value    => pwm_mod_value,
        mod_dir      => pwm_mod_dir,
        deadtime     => pwm_deadtime,
        pwm_en       => pwm_en,
        pwm_dir      => pwm_dir);
```

Dentro del fichero Top se declarará el interface externo añadiendo los mismos puertos de salida que en el Adapter, además se llevará a cabo su instanciación.

Decodificador de Señales

El último periférico a desarrollar es el encoder del motor. Deberá de leer los pulsos del encoder magnético que incorpora el motor y proporcionar el valor de la posición y los pulsos de velocidad para la posterior obtención de la velocidad en rpm y el sentido de giro.

El core del módulo se denominará axi_motor_enc.vhd, este a su vez estará compuesto por cuatro submódulos que harán más fácil su implementación y simulación de cada bloque por separado.

- Filtro_Digital.vhd, encargado de eliminar los posibles glitches que puedan aparecer en las señales CHA y CHB.
- Contador_Periodo.vhd, encargado de enviar un pulso cuando concluye el periodo de observación fijado.
- FSM.vhd, este módulo será encargado de enviar la dirección a la que se detecta que gira el motor, un pulso por flanco de la señal anterior y un pulso por cada flanco de subida o bajada de CHA_out y CHB_out.
- Contador_Posicion.vhd, el cual proporciona el valor en pulsos de la posición y de la velocidad, además, en el bit de mayor peso cada señal integra la dirección de giro.

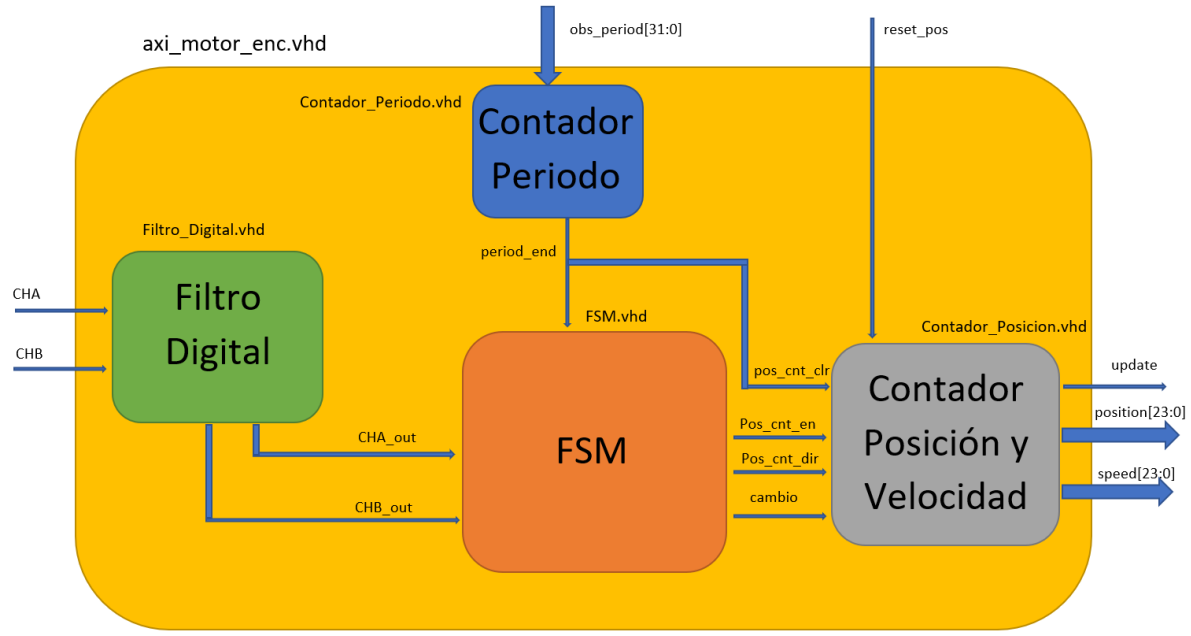

Figura 11: Diagrama de bloques del periférico axi_motor_enc

Filtro Digital

Se comienza el diseño del periférico desarrollando el filtro digital, encargado de eliminar los posibles glitches que pueden aparecer debido al ambiente ruidoso a que está sometida la aplicación.

Este módulo contendrá tres submódulo, uno dedicado a filtrar la señal del encoder A, otro para la señal B y un Prescaler.

El intervalo de muestreo se controla mediante el Prescaler. Este módulo se trata de un contador de pulsos de reloj, que en cuanto llega al valor establecido genera un pulso. Se eligió un valor de contador de 100 para el prescaler, debido a que es suficientemente grande para evitar glitches, de esta manera, la señal de salida que se genera es de 1MHz, que es suficientemente grande para filtrar los glitches y suficientemente pequeña para generar cuatro pulsos por cada medio periodo de la señal del encoder .

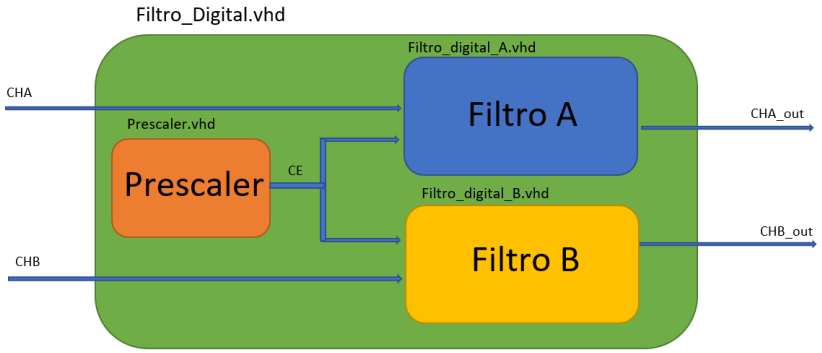

Figura 12: Diagrama de bloques de Filtro Digital

El Filtro A, al igual que el Filtro B, permitirán la conmutación de la salida si la señal de entrada permanece estable durante cuatro muestras consecutivas, en caso contrario, la señal mantendrá el último valor. Su estructura se muestra en la Figura 13 tratándose de un registro de desplazamiento junto con un biestable JK mediante el cual se genera la señal filtrada.

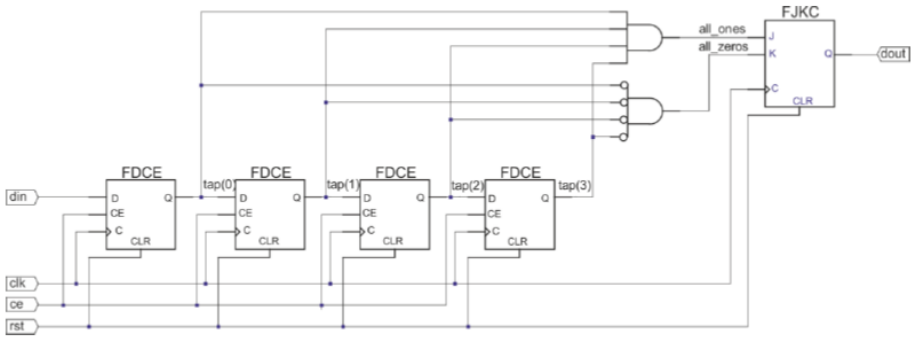

Figura 13: Estructura del Filtro A y Filtro B

A continuación, se simulará el bloque empleando un valor de 4 para el contador del prescaler. En la Figura 14 se observa una vista general de la simulación, pudiendo comprobar que se generan las señales de salida.

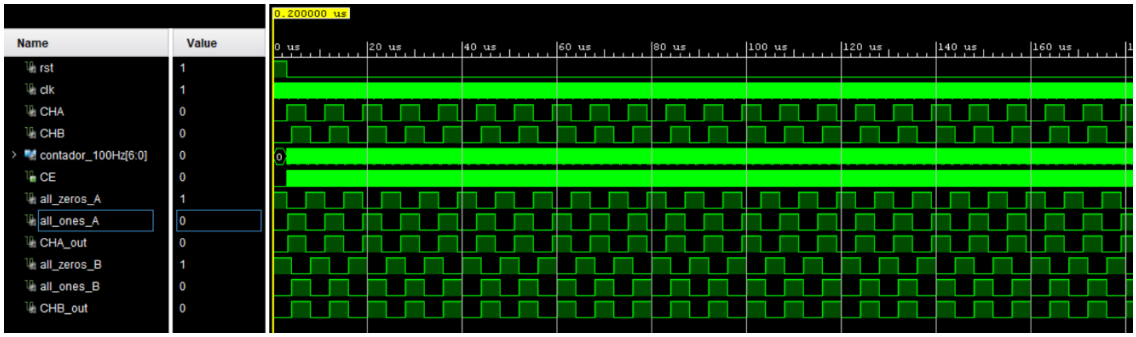

Figura 14: Simulación mediante Vivado del bloque Filtro_Digital

Ampliando la simulación se puede apreciar el valor de las distintas señales internas de cada bloque, además que las señales de los encoder están retrasadas cuatro pulsos de señal CE como consecuencia del filtro.

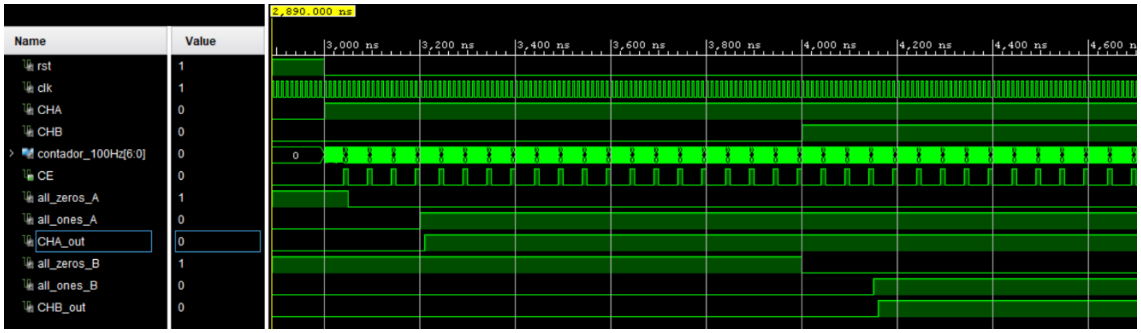

Figura 15: Simulación ampliada

Además, se simuló el comportamiento del bloque introduciendo glitches. Esto se realizó generando señales CHA y CHB de menor periodo, como se puede apreciar en la siguiente imagen.

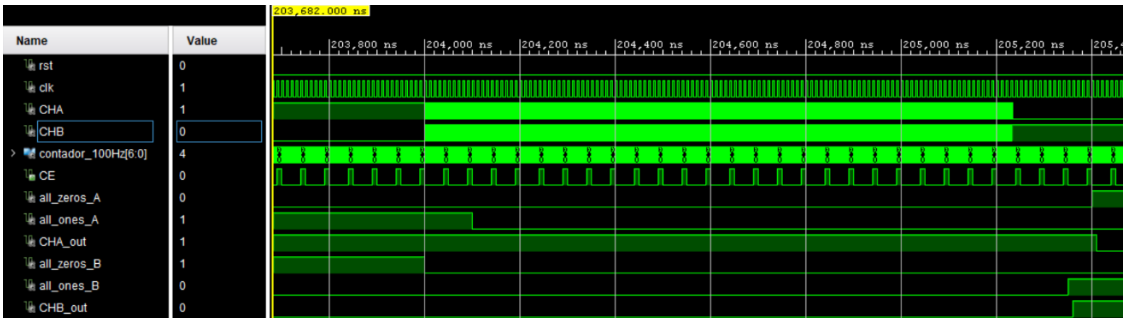

Figura 16: Parte de la simulación donde aparecen glitches

Ampliando la parte anterior se observan la conmutación de las señales de entrada con una mayor frecuencia, en cambio no se produce cambio alguno a la salida, debido a que las señales de entrada no duran más de un periodo de la señal CE, por lo que se puede afirmar que el filtro realiza su trabajo.

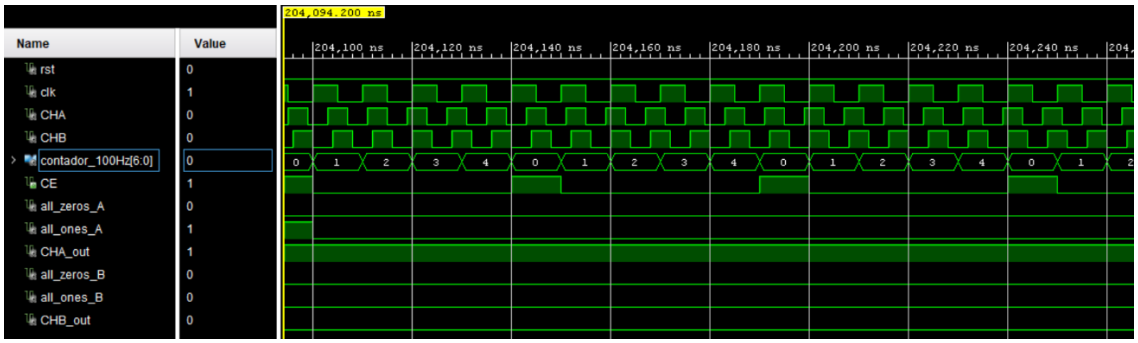

Figura 17: Ampliación de la Figura 16

Contador de Periodo

Una vez se encuentra operativo el filtro digital, se procede al diseño del contador de periodo. Este se encargará de proporcionar un pulso al finalizar el periodo de observación, posteriormente esta señal estará encargada de limpiar los pulsos de velocidad del último bloque además de generar una interrupción en el procesador.

Este bloque tendrá una única señal de entrada, obs_period, que será proporcionada por el usuario a través de un registro. Internamente consistirá en un contador descendente, que contará desde el valor fijado hasta 0, proporcionando al concluir un pulso indicando el fin del periodo.

A continuación, se puede observar la simulación de este bloque comprobando su correcto funcionamiento.

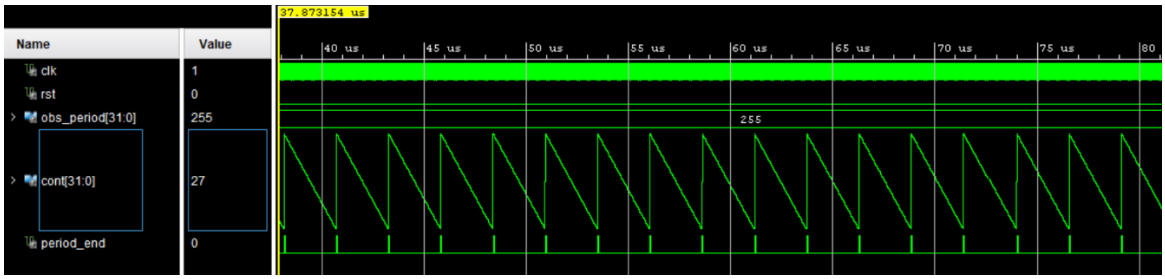

Figura 18: Simulación bloque Contador de Periodo

FSM

Una vez filtradas las señales en cuadratura y generado el bloque contador de periodo, se procede a diseñar una máquina de estado que tiene por finalidad detectar cuándo se produce un cambio de la posición del eje del motor y el sentido del giro.

Esta máquina de estados presentará las siguientes señales de entrada:

- CHA_out, señal CHA filtrada.
- CHB_out, señal CHB filtrada.
- period_end, pulso que indica el fin del periodo de observación.

Se deberá de seguir el siguiente esquema para la obtención del bloque.

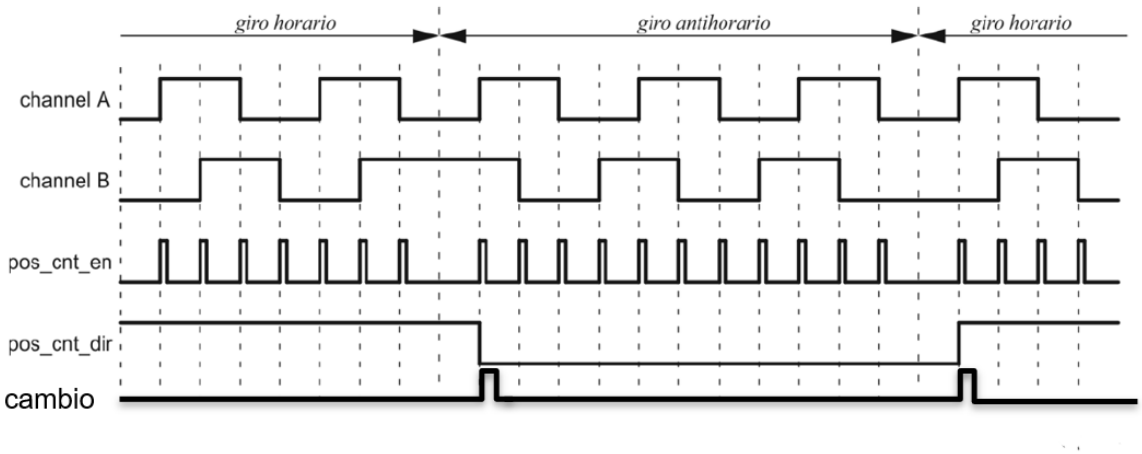

Figura 19: Generación de las señales de salida del bloque FSM

Este bloque, internamente genera una señal de control denominada pos_cnt_i, cuyo valor indica la posición del eje del motor. Dicha señal se generará mediante una operación de concatenación "&" de las señales CHA_out y CHB_out. Una vez obtenida la señal de control se implementa la máquina de estados descrita en la Figura 14, mediante la cual se obtiene el sentido de giro.

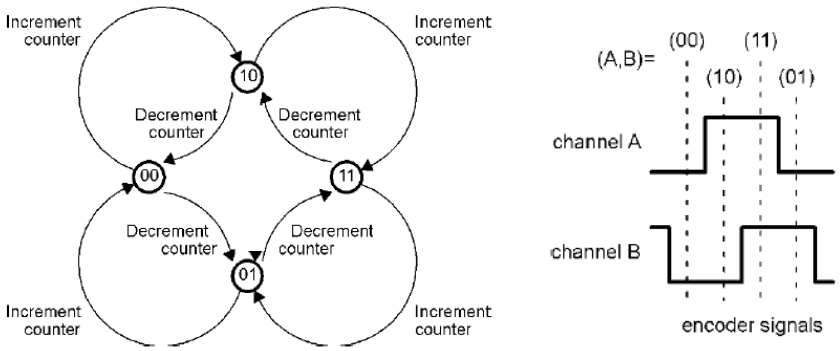

Figura 20: Máquina de estados para la decodificación de las señales en cuadratura

Tal y como se muestra en el cronograma de la Figura 19, la dirección de cuenta pos_cnt_dir depende de la relación de fases, correspondiendo con la señal obtenida a través de la máquina de estados, en

cuanto a la señal de habilitación de cuenta pos_cnt_en, se activa cada vez que aparece un flanco en alguno de los canales. Para ello se hace uso de la siguiente estructura para su obtención.

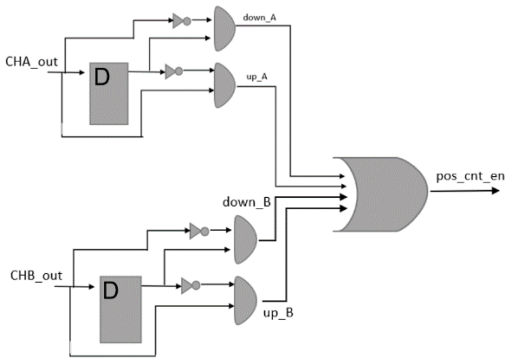

Figura 21: Estructura de detección de flancos de subida y bajada de las señales en cuadratura filtradas

En la primera imagen de la simulación se puede comprobar el correcto funcionamiento del módulo, pudiéndose apreciar la detección del sentido de giro a través de la señal pos_cnt_dir y cuando se produce este cambio.

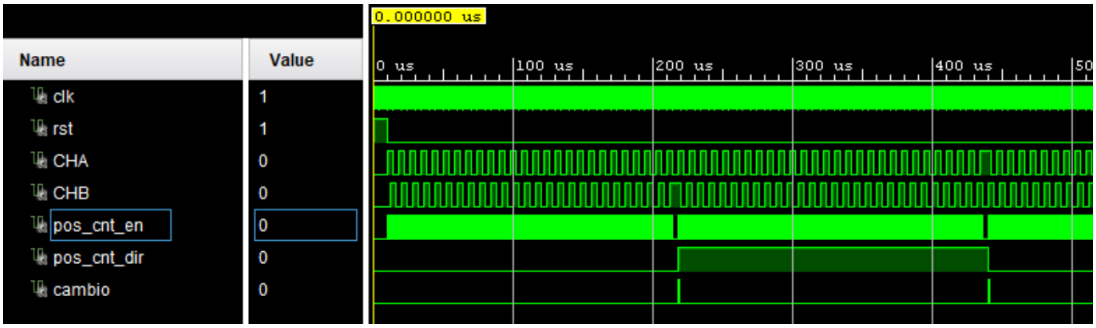

Figura 22: Simulación del bloque FSM

Ampliando la simulación al primer cambio de dirección se obtiene la siguiente imagen. En ella a través de TestBench se cambió la secuencia de las señales del encoder lo que provocó que el módulo detectase un cambio de dirección de giro. Además, se aprecia como la señal pos_cnt_en detecta todos los flancos de las señales en cuadratura

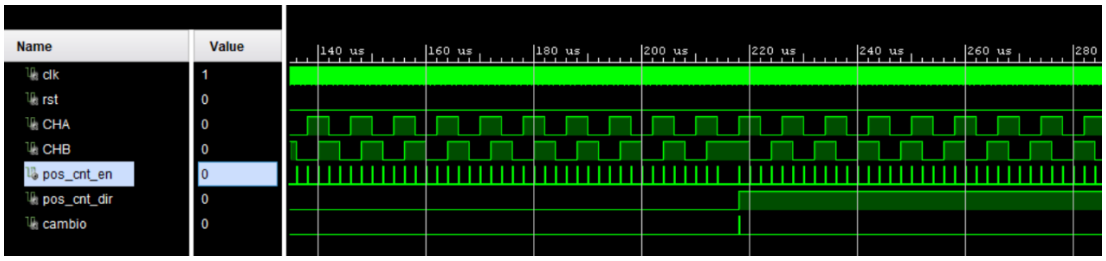

Figura 23: Ampliación de la simulación en la zona donde se produce un cambio de dirección de giro

Desplazando la simulación, se aprecia que el módulo también es capaz de detectar los cambios de sentido en la otra dirección, así comprobando su correcto funcionamiento

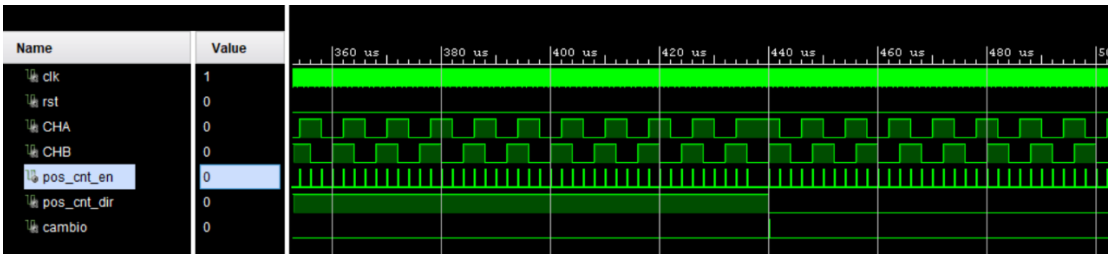

Figura 24: Ampliación de la simulación en la zona donde se produce otro cambio de dirección de giro

Contador de Posición

Por último, para concluir con el diseño del core del IP, se implementará el módulo contador de posición, que será el encargado de devolver el valor de la posición del motor, los pulsos generados en el periodo de observación permitiendo de esta manera la obtención de la velocidad en rpm y la señal registrada de pos_cnt_clr generada por el bloque contador de periodo.

El bloque presentará las siguientes señales de entrada:

- reset_pos. Señal procedente del registro de control del IP encargada de limpiar el contador de posición.
- pos_cnt_en. Pulsos recibidos del bloque FSM mediante los cuales se incrementan los contadores.
- pos_cnt_dir. Señal generada por el módulo FSM que indica el sentido de giro detectado.
- cambio. Señal procedente del FSM de detección de flancos de la señal pos_cnt_dir.
- pos_cnt_clr. Señal de fin de periodo de observación generada por el bloque Contador Periodo.

Primero se implementará el contador de posición bidireccional, este dependiendo del sentido de giro se incrementará o decrementará, dentro se le añadirán restricciones para que no se produzcan desbordamientos. Se llegaría al mismo objetivo convirtiendo el contador binario a complemento a dos.

Por otro lado, para facilitar el cálculo de la velocidad se creó un contador propio ascendente, el cual se reinicia al llegarle una señal a nivel alto de cambio de dirección o fin de periodo de observación. Su salida pasa por un biestable de tipo D, el cual registra el valor de la velocidad al llegarle las mismas señales que para el reset del contador de velocidad. Además, se encarga de generar un pulso indicando que ya se dispone de un nuevo valor de velocidad.

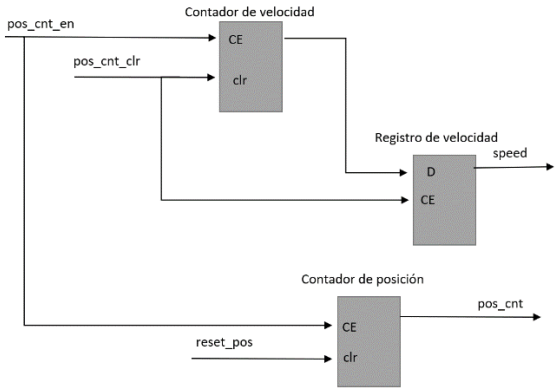

Figura 25: Estructura del contador de posición y contador de pulsos de velocidad

Tanto para el valor de la posición como para los pulsos de velocidad, se les añade en el último bit la dirección a la que gira el motor mediante una operación de concatenación "&".

Debido a que este último módulo requiere de señales generadas en los bloques anteriores, se comprobará su funcionamiento mediante una simulación global del bloque axi_motor_enc.

Simulación completa

Tras finalizar el diseño del core se unen los distintos bloques y se realiza la simulación completa, obteniendo como resultado las siguientes imágenes.

En la Figura 26 se aprecia una vista completa de toda la simulación, en la que aparecen las señales filtradas de CHA y CHB, los pulsos de period_end que indica la finalización del periodo de observación, la detección de cada flanco de las señales en cuadratura mediante pos_cnt_en y la detección del sentido de giro mediante pos_cnt_dir.

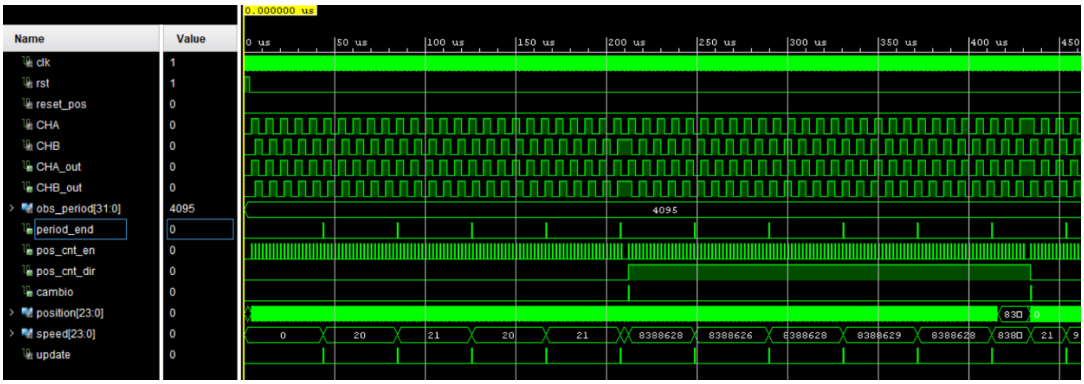

Figura 26: Simulación del core Decodificador de Señales

A continuación, se amplía la simulación con el objetivo de ver con detalle el funcionamiento del bloque contador de posición.
En la Figura 27 se aprecia los valores que van tomando el contador de posición y el contador de pulsos de velocidad, cada uno toma un valor diferente debido a que el contador de velocidad solo se reinicia cuando le llega la señal de reset_pos, en cambio, el contador de pulsos de velocidad se reinicia cada vez que el motor cambia de sentido de giro o termina un periodo de muestreo.

Cabe destacar que los valores de speed y position incluyen en su ultimo bit la dirección de giro, es por ello por lo que cuando esta cambia, los contadores adquieren valores muy elevados.

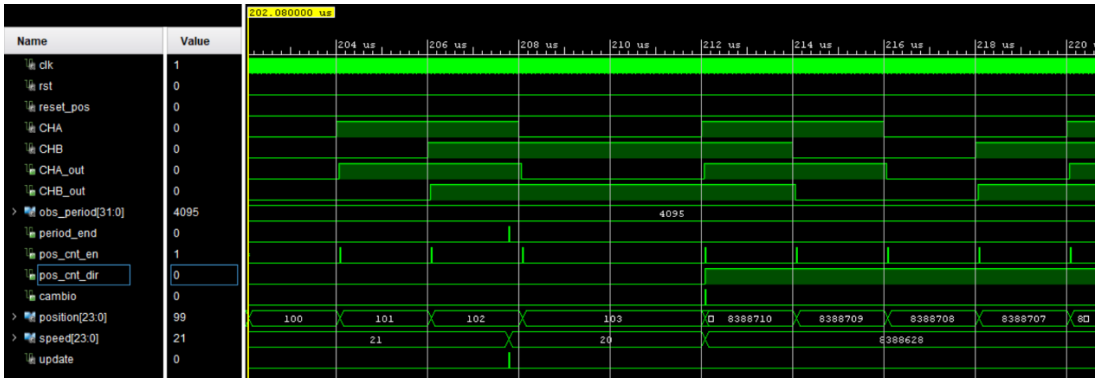

Figura 27: Ampliación de la simulación donde se produce el primer cambio de sentido de giro

Por último, se desplaza la simulación al siguiente cambio de sentido de giro donde se aprecia su correcto funcionamiento, además justo en ese momento se genera un pulso de la señal reset_pos lo que provoca que se reinicie el contador de posición.

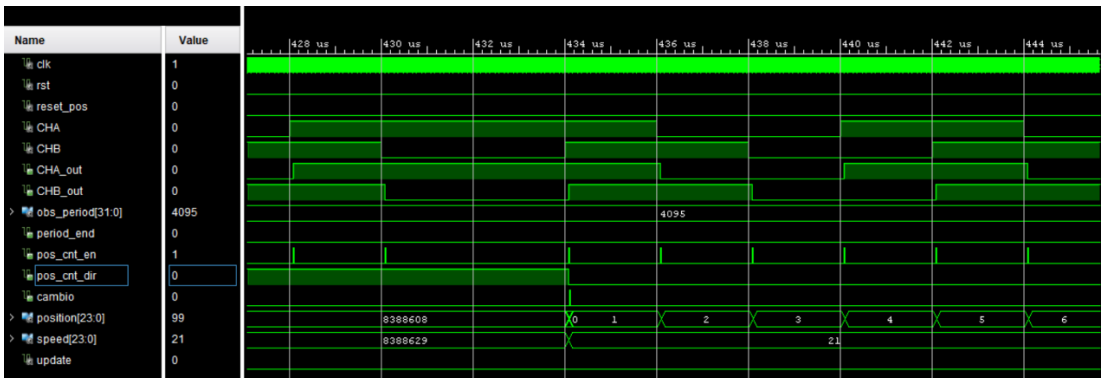

Figura 28: Simulación desplazada hasta el siguiente cambio de sentido de giro

Creación del IP

Tras la comprobación del funcionamiento del diseño mediante las simulaciones, se procede a generar el IP a medida con ayuda de Vivado, dotándolo de cuatro registros para su manejo, como aparece en la siguiente tabla. No se tuvo en consideración los offset debido a que dificultarían el manejo de los registros debido a que se necesitarían 6 en total, dos de period_reg y dos de speed_reg.

Nombre	Offset	Descripción
control_reg	0x0	Registro de control.
period_reg	0x4	Registro de periodo de observación para el cálculo de la velocidad.
position_reg	0x8	Registro de posición.
speed_reg	0xC	Registro de velocidad.

Figura 29: Tabla de registros del periférico axi_encoder

En el registro de control se emplearán los cuatro primeros bits.

- El bit 0 será mediante el cual se controle el reset del IP.
- El bit 1 será mediante el que se controle el reset del contados de posición. Este deberá de ser de tipo Write One Clear (WOC), lo que significa que adquirirá el valor escrito y seguidamente se pondrá a cero.
- El bit 2 para habilitar la interrupción.
- El bit 3 para limpiar el flag de la interrupción. También será de tipo WOC.

La interconexión se realizará de la siguiente manera para los tres primeros bits del registro de control y el registro de periodo de observación.

```
enc_rst  <= not(S_AXI_ARESETN) or slv_reg0(0); --Reset the Core
int_enable     <= slv_reg0(2);  --IRQ Enable R/W
enc_obs_period  <= slv_reg1(31 downto 0); --Periodo de Observacion R/W

process(S_AXI_ACLK) is    --Proceso de Reset de Posicion WOC
begin
 if (rising_edge (S_AXI_ACLK)) then
   if ( S_AXI_ARESETN = '0') then
    enc_reset_pos <= '0';
   else --Si llevamos un acceso de escritura, se realiza sobre dicha direccion y lase??al de strobe
    if(slv_reg_wren = '1' and S_AXI_WSTRB(0) = '1') then
       enc_reset_pos <= S_AXI_WDATA(1);
    else
       enc_reset_pos <= '0';
    end if;
   end if;
  end if;
end process;
```

Cabe resaltar que las interrupciones en el procesador se producen a nivel, por lo que la señal de salida update no será suficiente para generar una interrupción, por lo que se deberá de crear un proceso para fijar una señal a nivel alto, siendo esta (interrupt_i), cuando se detecte el pulso de update y la interrupción esté habilitada. Esta señal se pondrá a cero cuando se detecte que el flag esté a nivel alto. A continuación, se muestra dicho proceso junto al proceso encargado de establecer el bit 3 de la señal de control de tipo WOC.

```
process(S_AXI_ACLK) is
   begin
    if (rising_edge (S_AXI_ACLK)) then
      if ( S_AXI_ARESETN = '0') then
        interrupt_i <= '0';
      else
        if(int_enable = '1' and enc_update = '1') then
          interrupt_i <= '1';
        elsif(interrupt_flag = '1') then
            interrupt_i <= '0';
        end if;
      end if;
    end if;
   end process;

   process(S_AXI_ACLK) is
   begin
    if (rising_edge (S_AXI_ACLK)) then
      if ( S_AXI_ARESETN = '0') then
        interrupt_flag <= '0';
      else
        if(slv_reg_wren = '1' and S_AXI_WSTRB(0) = '1') then
          interrupt_flag <= S_AXI_WDATA(3);
        else
            interrupt_flag <= '0';
        end if;
      end if;
    end if;
   end process;

   interrupt <= '1' when((interrupt_i and not(interrupt_flag)) = '1') else '0';

   Conexion_Encoder : axi_motor_enc
      port map(
         clk          => S_AXI_ACLK,
         rst          => enc_rst,
         CHA          => CHA,
         CHB          => CHB,
         reset_pos    => enc_reset_pos,
         obs_period   => enc_obs_period,
         speed        => enc_speed,
         position     => enc_position,
         update       => enc_update);
```

En cuanto a los registros position_reg y speed_reg se deberán de implementar como registros de solo lectura, es por ello por lo que será necesario romper el multiplexor de salida de estos registros y poner directamente las señales, tal y como aparece a continuación.

```
process (enc_position, enc_speed, slv_reg0, slv_reg1, axi_araddr, S_AXI_ARESETN, slv_reg_rden)
  variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
  begin
    -- Address decoding for reading registers
   loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
   case loc_addr is
        when b"00" =>
          reg_data_out <= slv_reg0;
        when b"01" =>
          reg_data_out <= slv_reg1;
        when b"10" =>
          reg_data_out(22 downto 0)  <= enc_position(22 downto 0); --Posicion
          reg_data_out(30 downto 23) <= (others => '0');          --Ceros
          reg_data_out(31)           <= enc_position(23);         --Bit de signo
        when b"11" =>
          reg_data_out(22 downto 0)  <= enc_speed(22 downto 0);    --Velocidad
          reg_data_out(30 downto 23) <= (others => '0');          --Ceros
          reg_data_out(31)           <= enc_speed(23);            --Bit de signo
        when others =>
          reg_data_out  <= (others => '0');
   end case;
end process;
```

Por último, se deberán de añadir los puertos necesarios, en este caso las señales de entrada CHA y CHB y una salida interrupt, generada en el proceso descrito anteriormente.

Simulación BFM

Con el objetivo de comprobar el funcionamiento del IP incluyendo el decodificador de señales junto con sus registros se realizará una simulación Bus Functional Models (BFM). Para ello se modifican los archivos .tcl y se añaden las señales CHA y CHB manualmente, además de configurar los registros de control y tiempo de observación.

En la Figura 30 aparece una vista completa de la simulación en la que se aprecia como se generan las interrupciones activándose la señal interrupt a nivel alto y como esta misma se limpia mediante el flag de interrupción interrupt_flag.

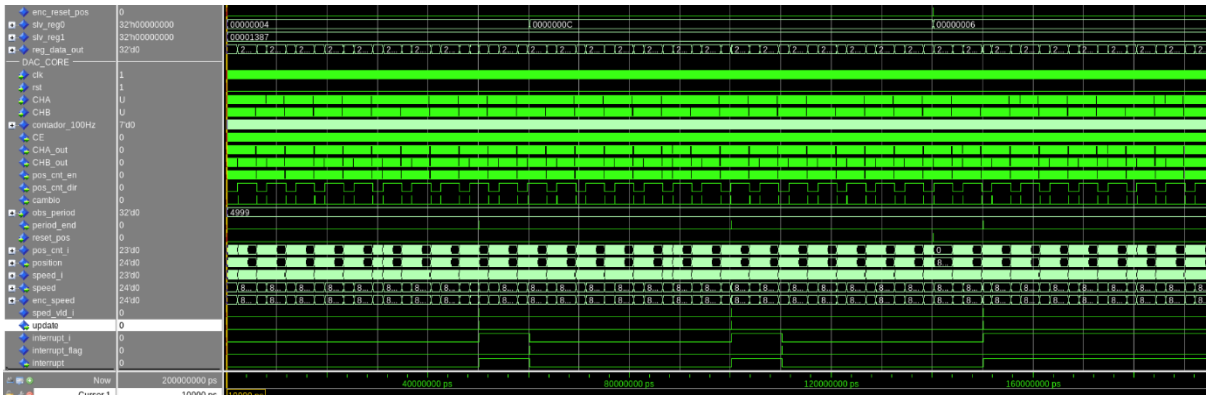

Figura 30: Simulación BFM Decodificador de Señales

Ampliando la simulación al comienzo, Figura 31, se aprecia la activación del reset en el cual se configuran todos los registros. También, se comprueba el correcto funcionamiento de las señales del core, así como las procedentes del Filtro Digital y FSM, siendo estas idénticas a las de la simulación mediante Vivado. A través de estas señales internas se incrementan los contadores de posición y velocidad.

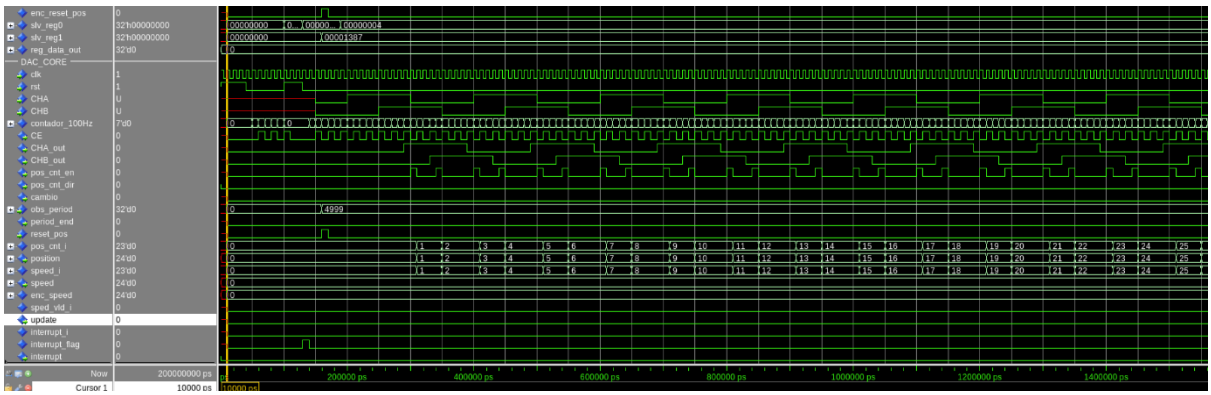

Figura 31: Inicio de la simulación

Desplazando la simulación se aprecia mediante period_end, un pulso que indica el fin del periodo con el cual se reinicia el contador de pulsos de velocidad y se activa la interrupción mediante interrupt. En esta figura el bloque FSM ha detectado que el sentido de giro es antihorario por lo que el contador de posición pos_cnt_i se decrementa hasta llegar a cero y mantiene este último valor sin desbordar.

Además, se observa como el registro de posición (position) y de velocidad (speed) se van actualizando y adquiriendo el valor de los respectivos contadores junto con el bit de signo, es por ello por lo que aparezcan números tan elevados como por ejemplo 8388608. Por último, el valor de los pulsos de velocidad registrados también se adquiere por la señal reg_data_out, siendo esta mediante la cual se obtendrá estos valores en la API accediendo al registro correspondiente.

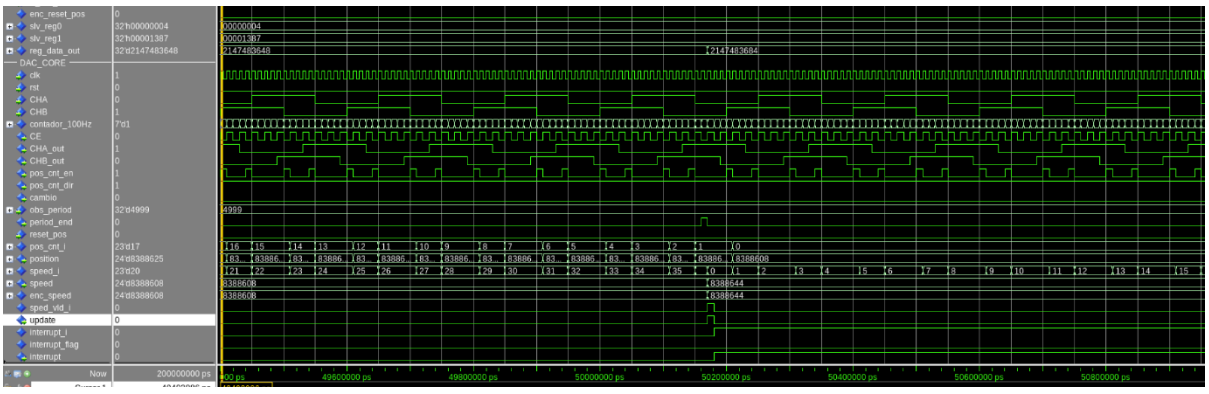

Figura 32: Simulación BFM donde acaba el periodo de observación

Por último, en la Figura 33 se aprecia como escribiendo en el bit 1 del registro de control se limpia el contador de posición mediante la señal reset_pos.

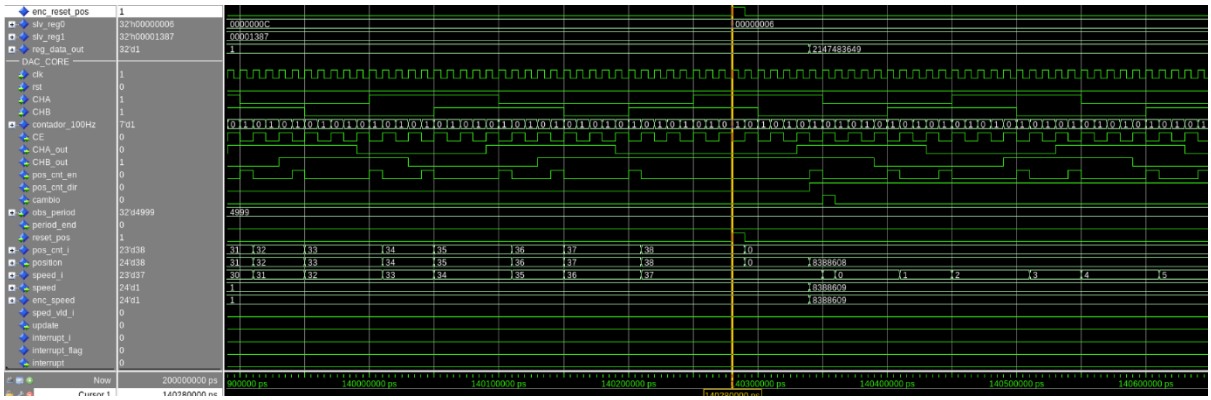

Figura 33: Simulación BFM donde se limpia el contador de posición

Desarrollo de la Aplicación

Una vez terminado el diseño se procederá a escribir la aplicación main.c encargada de controlar los dos IP. Para ello se crearán dos punteros encargados de apuntar a los registros de los IP, mediante los cuales se darán valores a los registros y se leerá su valor.

En cuanto al módulo PWM se crearán las siguientes funciones para su manejo:

- Set_PWM. Mediante la cual se configuran los distintos parámetros de funcionamiento del dispositivo.
- PWM_Duty_Cycle. Función encargada de fijar el ciclo de trabajo.
- set_left_turn_direction. Función encargada de cambiar el sentido de giro del motor en dirección antihoraria.
- set_right_turn_direction. Encargada de cambiar el sentido de giro del motor en dirección horaria.
- cambio: Función encargada de extraer del dato adquirido el ciclo de trabajo y la dirección de giro y llamar a las tres funciones anteriores para modificar estos parámetros.

En cuanto al módulo Decodificador de Señales se crearán las siguientes funciones:

- set_Encoder. Mediante la cual se configuran los distintos parámetros de funcionamiento del dispositivo.
- cnt_encoder_IRQ. Función de configuración del GIC para habilitar las interrupciones procedentes del encoder.
- encoder_clr_position. Función que se encarga de limpiar el contador de posición escribiendo un 1 en el bit 1 del registro de control.
- encoder_clr_flag. Función encargada de limpiar el flag de interrupción.
- encoder_set_observation_period. Función encargada de fijar el periodo de observación.
- encoder_get_position. Función encargada de obtener la velocidad junto con el sentido de giro.
- encoder_get_speed. Función encargada de obtener los pulsos de velocidad junto con el sentido de giro.

- encoder_IRQ_Handler. Función de atención a la interrupción, en la cual se limpia el flag, se recoge el valor de pulsos de velocidad, el sentido de giro, se calcula el valor de velocidad y se activa la variable token = 1 para que en la función main se detecte la adquisición de una nueva variable de velocidad.

Para el cálculo de la velocidad se empleará la siguiente expresión, donde 12 son los pulsos por vuelta que genera el encoder y 19,225 es la reductora del motor.

$$V_{RPM} = 60 \cdot V_{RPS}$$

$$V_{RPM} = 60 \cdot \frac{n_{pulsos}}{12 \cdot 19{,}225 \cdot T_{observacion} \cdot 10^{-3}}$$

$$V_{RPM} = \frac{60 \cdot 10^3}{12 \cdot 19{,}225} \cdot \frac{n_{pulsos}}{T_{observacion}}$$

Por último, se hará uso del puerto de comunicación serie, UART, mediante el cual se imprimirá por pantalla los valores de velocidad, sentido de giro y se adquirirán distintos parámetros, como la frecuencia del PWM, los tiempos muertos, el tiempo de observación y el ciclo de trabajo. Debido a que en la aplicación SDK ya viene inicializada la UART, solamente se deberá de hacer uso de la función xil_printf para imprimir por pantalla y scanf para enviar algún parámetro.

Conclusión

Durante la realización de la práctica de laboratorio se puso en práctica los conocimientos adquiridos de las asignaturas Sistemas Electrónicos Digitales, Diseño Electrónico y Sistemas Electrónicos Digitales Avanzados.
El proyecto consistió en el diseño de un generador de pulsos PWM que incorporaba la posibilidad de cambio de giro del motor respetando los tiempos muertos del puente en H utilizado y el diseño de un decodificador de señales, que mediante un encoder que integraba el motor se obtenía la velocidad de giro de este. Tras el diseño del core se procedió a crear las IP a medida mediante el asistente de Vivado haciendo modificaciones pertinentes en los ficheros Top y Adapter en lenguaje VHDL, tras lo cual se creó un proyecto Vivado implementando un nuevo diseño integrando los IP anteriores y exportando el BitStream a la aplicación SDK. Dentro de esta herramienta se generó una aplicación en lenguaje C con el objetivo de poder enviar y adquirir valores de los registros, controlando así de esta manera la plataforma hardware. En cada uno de los pasos anteriores se realizaron distintas simulaciones tanto mediante el asistente de Vivado para la simulación del core como mediante QuestaSim implementando un BFM con el fin de simular el comportamiento de la plataforma junto a sus registros.

Para el diseño de todo lo referente al lenguaje VHDL, se intentó realizar de manera mas sencilla, evitando estructuras exóticas y a medida de lo posible implementando bloques vistos en Electrónica Digital.

Durante la realización de la práctica se topó con numerosos problemas, fallos y diferentes solución a estos. Uno de los errores graves fue la implementación de todos los resets del core de manera asíncrona, lo que generaba un número considerable de latches, lo que a su vez provocaba un funcionamiento indeterminado a la hora de cargar la aplicación sobre la placa. Esto se solucionó implementado los resets de forma síncrona, lo que redujo a cero todos los latches e hizo funcionar de manera correcta la descarga de la aplicación PWM sobre la placa.
Otro de los fallos fue introducir un valor para el prescaler del Filtro Digital demasiado alto, lo que generaba una señal con un periodo demasiado alto como para que en cada nivel de la señal de cuadratura se captasen cuatro ciclos de la señal de prescaler, por lo que la señal de salida era constantemente cero, lo que impedía el correcto funcionamiento de los siguientes bloques. Una vez solucionado este problema fijando el valor del contador a 100, el IP se comportó de manera esperada.

Por último, mediante el script .tcl el cual contenía funciones de carga y descarga de datos del SDK, se obtuvo la respuesta del motor a un vector de ciclos de trabajo. Con este vector descargado, se abrió Matlab mediante la cual se obtuvo los parámetros de la planta y se generó un sistema de control PID basado en Ziegler-Nichols. Por desgracia, a falta de tiempo no se pudo implementar este controlador en el código C de manera exitosa.

Anexo I: Diseño VHDL del Generador de PWM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity pmod5HB_pwm is
  port (
                clk : in std_logic;
                rst : in std_logic;
                enable_pwm : in std_logic;
                enable_dead : in std_logic;
                carrier_period : in std_logic_vector(23 downto 0);
                mod_value : in std_logic_vector(23 downto 0);
                mod_dir : in std_logic;
                deadtime : in std_logic_vector(15 downto 0);
                pwm_en : out std_logic;
                pwm_dir : out std_logic);

end entity ; -- pmod5HB_pwm

architecture rtl of pmod5HB_pwm is

        component pwm_generator is
                port(
                        clk : in std_logic;
                rst : in std_logic;
                enable_pwm : in std_logic;  --Señ?al enable PWM
                carrier_period : in std_logic_vector(23 downto 0); --Valor maximo
                mod_value : in std_logic_vector(23 downto 0); --Ciclo de trabajo
                pwm : out std_logic; --Salida PWM
                carrier_fin : out std_logic); --Pulsos fin de periodo

        end component pwm_generator;

        component deadtime_cont is
                port(
       clk : in std_logic;
                        rst : in std_logic;
                enable_dead : in std_logic; --Señ?al enable tiempos muertos
                mod_dir : in std_logic;  --Cambiar direccion
                        pwm_in : in std_logic;  --Señ?al PWM entrada
                        start_dt : in std_logic; --Señ?al fin periodo
                        deadtime : in std_logic_vector(15 downto 0); --Cuentas de deadtime
                        pwm_out : out std_logic;  --Salida PWM
                dir : out std_logic); --Salida direccion

        end component deadtime_cont;

        signal pwm_i : std_logic;
        signal carrier_fin_i : std_logic;

begin

        --JUNTO LOS DOS COMPONENTES, pwm_generator Y deadtime--

        Conexion_pwm_generator : pwm_generator
                port map(
                        clk => clk,
```

```
                rst => rst,
                enable_pwm => enable_pwm,
                carrier_period => carrier_period,
                mod_value => mod_value,
                pwm => pwm_i,
                carrier_fin => carrier_fin_i);


    Conexion_deadtime : deadtime_cont
        port map(
                clk => clk,
                rst => rst,
                pwm_in => pwm_i,
                mod_dir => mod_dir,
                enable_dead => enable_dead,
                start_dt => carrier_fin_i,
                deadtime => deadtime,
                pwm_out => pwm_en,
                dir => pwm_dir);


end architecture ; -- rtl
```

Generador de Pulsos

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity pwm_generator is
  port (
     clk : in std_logic;
     rst : in std_logic;
     enable_pwm : in std_logic;  --Seal enable PWM
     carrier_period : in std_logic_vector(23 downto 0); --Valor maximo
     mod_value : in std_logic_vector(23 downto 0); --Ciclo de trabajo
     pwm : out std_logic; --Salida PWM
     carrier_fin : out std_logic); --Pulsos fin de periodo

end entity pwm_generator;

architecture rtl of pwm_generator is

  signal pwm_i : std_logic;
  signal carrier_cnt : unsigned(23 downto 0);
  signal mod_value_reg : std_logic_vector(23 downto 0);
  signal carrier_fin_i : std_logic;

begin

  Actualizacion_Registros : process(clk) is
  begin
    if(clk = '1' and clk'event)then  --En cada flanco de CLK
      if (rst = '1') then               --El reset lo pone todo a un valor inicial
        mod_value_reg <= mod_value;
      elsif(carrier_fin_i = '1') then --compruebo si ha llegado a fin de cuenta y si esta activo deadtime_enable
        mod_value_reg <= mod_value; --Actualizo el valor del ciclo de trabajo
      end if;
    end if;
```

```
 end process; -- Actualizacion_Registros

 carrier_fin_i <= '1' when (carrier_cnt = 0) else '0';

 MR0 : process(clk) is
 begin
  if (clk = '1' and clk'event) then
   if(rst = '1') then
    carrier_cnt <= unsigned(carrier_period);
   elsif(enable_pwm = '1') then
    if(carrier_fin_i = '1') then        --Cuando ha finalizado el periodo del pwm
     carrier_cnt <= unsigned(carrier_period); --Vuelvo a empezar a contar
    else
     carrier_cnt <= carrier_cnt - 1; --Si no, decremento el contador
    end if;
   end if;
  end if;

 end process; -- MR0

 MR1 : process(clk)
 begin
  if(clk = '1' and clk'event) then
   if(rst = '1') then
     pwm_i <= '0';
   elsif(enable_pwm = '1') then
    if(carrier_cnt > unsigned(mod_value_reg)) then --Si el contador esta por debajo del ciclo de trabajo fijado,
PWM = 1
      pwm_i <= '0';  --la salida sera 1
     elsif (carrier_fin_i = '0') then
      pwm_i <= '1';  --Si, no, PWM = 0
     end if;
   end if;
  end if;

 end process ; -- MR1

  carrier_fin <= carrier_fin_i; --Saco la señal de fin de periodo
  pwm <= pwm_i; --Saco la señal PWM

end architecture rtl;
```

Generador de Tiempos Muertos

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity deadtime_cont is
 port (
          clk : in std_logic;
           rst : in std_logic;
          enable_dead : in std_logic; --Señ?al enable tiempos muertos
           pwm_in : in std_logic;  --Señ?al PWM entrada
          mod_dir : in std_logic;  --Cambiar direccion
           start_dt : in std_logic; --Señ?al fin periodo
           deadtime : in std_logic_vector(15 downto 0); --Cuentas de deadtime
           pwm_out : out std_logic;  --Salida PWM
```

```
            dir : out std_logic); --Salida direccion

end entity ; -- deadtime_cont

architecture rtl of deadtime_cont is

  signal dt_enable : std_logic;
  signal detect_start : std_logic;
  signal deadtime_i : unsigned(15 downto 0);
  signal prev_dir : std_logic;
  signal dir_i : std_logic;

  signal dir_up : std_logic;
  signal dir_down : std_logic;

begin

  Actualizacion_direccion : process(clk)
  begin
   if(clk = '1' and clk'event) then
    if(rst = '1') then
      prev_dir <= mod_dir;
      dir_up <= '0';
      dir_down <= '0';
    elsif (enable_dead = '1' and start_dt = '1')then --Si detecta cambio en la direcci???n y es fin de periodo
      prev_dir <= mod_dir;
      dir_up <= not(prev_dir) and mod_dir;
      dir_down <= prev_dir and not(mod_dir);
    end if;
   end if;

  end process ; -- Actualizacion_direccion

  detect_start <= dir_down or dir_up; --Cuando detecta que las señ?ales de dir_in y prev_dir son diferentes se
pone a 1

  Enable_deadtime : process(clk)
  begin
   if(clk = '1' and clk'event) then
    if(rst = '1') then
       dt_enable <= '0';
       dir_i <= '0';
    else
      if (detect_start = '1' and start_dt = '1')then --Si detecta cambio en la direcci???n y es fin de periodo
        dt_enable <= '1';  --Activa el deadtime
        dir_i <= prev_dir;
      elsif(deadtime_i = 0) then  --Si llega a fin de cuenta
      dt_enable <= '0';  --Lo desactiva
      end if;
    end if;
   end if;

  end process ;

  Dead_time_cont : process(clk)
   begin
    if(clk = '1' and clk'event) then
      if(rst = '1') then
         deadtime_i <= unsigned(deadtime);  --Actualizo valor deadtime
      else
```

```
            if (dt_enable = '1') then  --Si la se???al de enable del deadtime eta activa
            deadtime_i <= deadtime_i - 1; -- Decremento el contador
            else
            deadtime_i <= unsigned(deadtime);  --El contador vuelve a la posicion inicial
            end if;
          end if;
        end if;

    end process ; -- Dead_time_cont

    pwm_out <= pwm_in and not(dt_enable);
    dir <= dir_i;

end architecture ; -- rtl
```

Anexo II: Diseño VHDL del Decodificador de Señales

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity axi_motor_enc is
  port (
          clk       : in std_logic;
          rst       : in std_logic;
          reset_pos  : in std_logic;
          CHA       : in std_logic;
          CHB       : in std_logic;
          obs_period : in std_logic_vector(31 downto 0);
          speed     : out std_logic_vector(23 downto 0); --Posicion para el calculo de velocidad
          position   : out std_logic_vector(23 downto 0); --Posicion
          update     : out std_logic);                --Señ?al de actualizacion para la interrupcion

end entity ; -- axi_motor_enc

architecture rtl of axi_motor_enc is

   component contador_periodo is
      port(
         clk       : in std_logic;
         rst       : in std_logic;
         obs_period : in std_logic_vector(31 downto 0);
         period_end : out std_logic);

   end component contador_periodo;

          component Filtro_Digital is
                    port(
                     clk        : in std_logic;
                     rst        : in std_logic;
                     CHA        : in std_logic;
                     CHB        : in std_logic;
                     CHA_out    : out std_logic;
                     CHB_out    : out std_logic);

          end component Filtro_Digital;

          component FSM is
                    port(
                    clk        : in std_logic;
                     rst        : in std_logic;
                     CHA        : in std_logic;
                     CHB        : in std_logic;
                     pos_cnt_en  : out std_logic;
                     pos_cnt_dir : out std_logic;
                     cambio     : out std_logic);

          end component FSM;

          component contador_posicion is
      port(
          clk       : in std_logic;
          rst       : in std_logic;
          reset_pos   : in std_logic;
```

```
        pos_cnt_en  : in std_logic;
        pos_cnt_dir : in std_logic;
        pos_cnt_clr : in std_logic;
        cambio      : in std_logic;
        pos_cnt     : out std_logic_vector(23 downto 0);
        speed       : out std_logic_vector(23 downto 0);
        speed_vld   : out std_logic);

    end component contador_posicion;

        signal CHA_i        : std_logic;
        signal CHB_i        : std_logic;
        signal period_end_i  : std_logic;
        signal pos_cnt_en_i  : std_logic;
        signal pos_cnt_dir_i : std_logic;
        signal pos_cnt_i     : std_logic_vector(23 downto 0);
        signal cambio_i      : std_logic;
        signal speed_i       : std_logic_vector(23 downto 0);
        signal speed_vld_i   : std_logic;

begin

    Conexion_contador_periodo : contador_periodo
      port map(
        clk        => clk,
        rst        => rst,
        obs_period  => obs_period,
        period_end  => period_end_i);

    Conexion_Filtro_Digital : Filtro_Digital
      port map(
        clk        => clk,
        rst        => rst,
        CHA         => CHA,
                   CHB         => CHB,
                   CHA_out     => CHA_i,
                   CHB_out     => CHB_i);

    Conexion_FSM : FSM
      port map(
        clk        => clk,
        rst        => rst,
        CHA         => CHA_i,
                   CHB         => CHB_i,
                   pos_cnt_en  => pos_cnt_en_i,
                   pos_cnt_dir => pos_cnt_dir_i,
                   cambio      => cambio_i);


Conexion_contador_posicion : contador_posicion
      port map(
        clk        => clk,
        rst        => rst,
        reset_pos   => reset_pos,
        pos_cnt_en  => pos_cnt_en_i,
        pos_cnt_dir => pos_cnt_dir_i,
        pos_cnt_clr => period_end_i,
        cambio      => cambio_i,
        pos_cnt     => position,
        speed       => speed,
```

```
            speed_vld   => speed_vld_i);
            update   <= speed_vld_i;

end architecture ; -- rtl
```

Filtro Digital

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Filtro_Digital is
  port (
    clk     : in std_logic;
    rst     : in std_logic;
    CHA     : in std_logic;   --Señ?al A entrada
    CHB     : in std_logic;   --Señ?al B entrada
    CHA_out : out std_logic;  --Señ?al A filtrada
    CHB_out : out std_logic); --Señ?al B filtrada

end entity ; -- Filtro_Digital

architecture rtl of Filtro_Digital is

  component Filtro_digital_A is
    port(
      clk     : in std_logic;
      rst     : in std_logic;
      CHA     : in std_logic;
      CE      : in std_logic;
      CHA_out : out std_logic);

  end component Filtro_digital_A;

  component Filtro_digital_B is
    port(
      clk     : in std_logic;
      rst     : in std_logic;
      CHB     : in std_logic;
      CE      : in std_logic;
      CHB_out : out std_logic);

  end component Filtro_digital_B;

  component Prescaler is
    port(
      clk : in std_logic;
      rst : in std_logic;
      CE  : out std_logic);

  end component Prescaler;

  signal CE_i : std_logic;

begin

  Conexion_FD_A : Filtro_digital_A
    port map(
      clk     => clk,
```

```
    rst     => rst,
    CHA     => CHA,
    CE      => CE_i,
    CHA_out => CHA_out);

  Conexion_FD_B : Filtro_digital_B
   port map(
    clk     => clk,
    rst     => rst,
    CHB     => CHB,
    CE      => CE_i,
    CHB_out => CHB_out);

 Conexion_Prescaler : Prescaler
  port map(
    clk     => clk,
    rst     => rst,
    CE      => CE_i);

end architecture ; -- rtl
```

Filtro A

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity filtro_digital_A is
 port (
                clk     : in std_logic;
                rst     : in std_logic;
                CHA     : in std_logic;
                CE      : in std_logic;   --Prescaler
                CHA_out : out std_logic);

end entity ; -- filtro_digital_A

architecture rtl of filtro_digital_A is

        signal CHA_out_i   : std_logic;
        signal tap         : std_logic_vector (3 downto 0);
        signal all_zeros_A : std_logic;
        signal all_ones_A  : std_logic;

begin

 Contador : process (clk) is --Contador de pulsos de CE
  begin  -- process
   if(clk = '1' and clk'event) then
    if(rst = '1') then
      tap <= (others => '0');
    elsif(CE = '1') then
      tap <= tap(2 downto 0) & CHA;
    end if;
   end if;

  end process;

 all_ones_A  <= '1' when (tap = "1111") else '0';
 all_zeros_A <= '1' when (tap = "0000") else '0';
```

```
Biestable_JK : process (clk) is  --Si se producen suficientes pulsos, se pasa a la salida
begin  -- process
 if(clk = '1' and clk'event) then
      if(rst = '1') then
        CHA_out_i <= '0';
      else
        if(all_ones_A = '1') then
           CHA_out_i <= '1';
        elsif(all_zeros_A = '1') then
           CHA_out_i <= '0';
        end if;
      end if;
 end if;
end process;

 CHA_out <= CHA_out_i;

end architecture ; -- rtl
```

Filtro B

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity filtro_digital_B is
  port (
    clk     : in std_logic;
    rst     : in std_logic;
    CHB     : in std_logic;
    CE      : in std_logic;   --Prescaler
    CHB_out : out std_logic);

end entity ; -- filtro_digital_B

architecture rtl of filtro_digital_B is

  signal CHB_out_i   : std_logic;
  signal tap         : std_logic_vector (3 downto 0);
  signal all_zeros_B : std_logic;
  signal all_ones_B  : std_logic;

begin

    Contador : process (clk) is  --Contador de pulsos de CE
    begin  -- process
     if(clk = '1' and clk'event) then
      if(rst = '1') then
        tap <= (others => '0');
      elsif(CE = '1') then
           tap <= tap(2 downto 0) & CHB;
      end if;
     end if;

end process;

  all_ones_B  <= '1' when (tap = "1111") else '0';
  all_zeros_B <= '1' when (tap = "0000") else '0';
```

```
  Biestable_JK : process (clk) is  --Si se producen suficientes pulsos, se pasa a la salida
  begin  -- process
   if(clk = '1' and clk'event) then
    if(rst = '1') then
    CHB_out_i <= '0';
    else
     if(all_ones_B = '1') then
      CHB_out_i <= '1';
     elsif(all_zeros_B = '1') then
      CHB_out_i <= '0';
     end if;
    end if;
   end if;

  end process;

  CHB_out <= CHB_out_i;

end architecture ; -- rtl
```

Prescaler

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Prescaler is
  port (
                clk : in std_logic;
                rst : in std_logic;
                CE  : out std_logic); --Será? de 25MHz

end entity ; -- Prescaler

architecture rtl of Prescaler is

  constant Fin_cuenta   : integer :=99; --No se actualiza instantaneamente
  --constant Fin_cuenta   : integer :=1; --No se actualiza instantaneamente
  signal CE_i          : std_logic;
  signal contador_100Hz : unsigned(6 downto 0);

begin

   cnt_100Hz : process (clk) is  --Me crea una señ?al de frecuencia de 100Hz
     begin
     if (clk = '1' and clk'event) then
      if (rst = '1') then
       contador_100Hz <= (others => '0');
       CE_i <= '0';
      else
       if (contador_100Hz = Fin_cuenta) then
           CE_i <= '1';
           contador_100Hz <= (others => '0');
       else
          contador_100Hz <= contador_100Hz + 1;
          CE_i <= '0';
       end if;
      end if;
     end if;
   end process;
```

```
   CE <= CE_i;
end architecture ; -- rtl
```

Contador de Periodo

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity contador_periodo is
  port (
            clk       : in std_logic;
            rst       : in std_logic;
            obs_period : in std_logic_vector(31 downto 0); --Periodo de observacion
            period_end : out std_logic);                --Señ?al fin de periodo de observacion

end entity ; -- contador_periodo

architecture rtl of contador_periodo is

            signal period_end_i : std_logic;
            signal cont         : unsigned(31 downto 0);

begin

   Contador_descendente : process (clk) is
   begin  -- process
    if(clk = '1' and clk'event) then
      if(rst = '1') then
      cont <= unsigned(obs_period);
      else
            if(cont = 0) then               --Cuando llega a 0, se reinicia
               cont <= unsigned(obs_period);
       else
          cont <= cont - 1;           --Si no, se decrementa
       end if;
      end if;
    end if;

            end process;

            Pulso_fin : process (clk) is
   begin  -- process
    if(clk = '1' and clk'event) then
      if(rst = '1') then
      period_end_i <= '0';
      else
            if(cont = 1) then       --Cuando el contador este a 1, porque va con retraso
                     period_end_i <= '1';   --Se ha acabado el periodo
            else
                     period_end_i <= '0';   --Si no, a 0
            end if;
      end if;
    end if;

            end process;

            period_end <= period_end_i;
```

```
end architecture ; -- rtl
```

FSM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity FSM is
  port (
    clk       : in std_logic;
    rst       : in std_logic;
    CHA       : in std_logic;   --Señ?al A filtrada
    CHB       : in std_logic;   --Señ?al B filtrada
    pos_cnt_en  : out std_logic;  --Sañ?al de incremento del contador
    pos_cnt_dir : out std_logic;  --Señ?al de direccion de giro
    cambio      : out std_logic); --Señ?al de deteccion de cambio de direccion

end entity ; -- FSM

architecture rtl of FSM is

  signal pos_cnt_dir_i : std_logic;
  signal pos_cnt_i     : unsigned(1 downto 0);

  signal cambio_1      : std_logic;
  signal cambio_up     : std_logic;
  signal cambio_down   : std_logic;

  signal CHA_1         : std_logic;
  signal CHB_1         : std_logic;

  signal up_A          : std_logic;
  signal down_A        : std_logic;
  signal up_B          : std_logic;
  signal down_B        : std_logic;

  type stateFSM is (S1,S2,S3,S4); --Creaccion de estados
  signal state : stateFSM ;  -- estado de la maquina secuencial

begin

    pos_cnt_i <= CHA & CHB;  --Contador de combinaciones del encoder

    process (clk, rst) is
    begin  -- process
    if (rst = '1') then                 --Lo inicializo en el estado S1
       state <= S1;
       pos_cnt_dir_i <= '0';
    elsif (clk = '1' and clk'event) then
      case state is
        when S1 =>
        if (pos_cnt_i="10")then          --Si la combinacion se cumple pasa al figuiente estado
          state <= S2;
          pos_cnt_dir_i <= '0';
        elsif(pos_cnt_i="01") then       --Si va al reves, pasa al estado anterior
          state <= S4;
          pos_cnt_dir_i <= '1';
```

```
      end if;
      when S2 =>
      if (pos_cnt_i="11")then
        state <= S3;
        pos_cnt_dir_i <= '0';
      elsif(pos_cnt_i="00") then
        state <= S1;
        pos_cnt_dir_i <= '1';
      end if;
      when S3 =>
      if (pos_cnt_i="01")then
        state <= S4;
        pos_cnt_dir_i <= '0';
      elsif(pos_cnt_i="10") then
        state <= S2;
        pos_cnt_dir_i <= '1';
      end if;
      when S4 =>
      if (pos_cnt_i="00")then
        state <= S1;
        pos_cnt_dir_i <= '0';
      elsif(pos_cnt_i="11") then
        state <= S3;
        pos_cnt_dir_i <= '1';
      end if;
    end case ;
  end if;
end process;

 Cabio_direccion : process (clk) is
  begin  -- process
   if(clk = '1' and clk'event) then
    if(rst = '1') then                          --Puesta de todas las señ?ales a 0
      cambio_1    <= '0';
      cambio_up   <= '0';
      cambio_down <= '0';
    else
      cambio_1    <= pos_cnt_dir_i;                 --Señ?al del contador retrasada
      cambio_up   <= not(cambio_1) and pos_cnt_dir_i; --Detector de flanco de subida
      cambio_down <= cambio_1 and not(pos_cnt_dir_i); --Detector de flanco de bajada
    end if;
  end if;

 end process;

Biestable_A : process (clk) is
   begin  -- process
    if(clk = '1' and clk'event) then
     if(rst = '1') then                 --Puesta de todas las señ?ales a 0
       CHA_1  <= '0';
       up_A   <= '0';
       down_A <= '0';
     else
       CHA_1  <= CHA;                  --Señ?al encoder A retrasada
       up_A   <= not(CHA_1) and CHA;    --Detector de flanco de subida
       down_A <= CHA_1 and not(CHA);    --Detector de flanco de bajada
     end if;
   end if;
          end process;
```

```
Biestable_B : process (clk, rst, CHB) is
   begin  -- process
    if(clk = '1' and clk'event) then
     if(rst = '1') then               --Puesta de todas las señ?ales a 0
       CHB_1  <= '0';
       up_B   <= '0';
       down_B <= '0';
     else
       CHB_1  <= CHB;               --Señ?al encoder B retrasada
       up_B   <= not(CHB_1) and CHB;  --Detector de flanco de subida
       down_B <= CHB_1 and not(CHB);  --Detector de flanco de bajada
     end if;
   end if;
          end process;

  pos_cnt_en  <= up_A or up_B or down_A or down_B; --Si alguna está? a 1, se pone a 1 el incremento del
contador
  pos_cnt_dir <= pos_cnt_dir_i;                  --Se devuelve la direccion de giro
  cambio      <= cambio_up or cambio_down;       --Se devuelve los cambios de direccion detectado

end architecture ; -- rtl
```

Contador de Posición

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity contador_posicion is
  port (
    clk       : in std_logic;
    rst       : in std_logic;
    reset_pos  : in std_logic;                  --Señ?al de reset de posicion(Registro)
    pos_cnt_en  : in std_logic;
    pos_cnt_dir : in std_logic;
    pos_cnt_clr : in std_logic;
    cambio      : in std_logic;                 --Pulso de deteccion de cambio de sentido
    pos_cnt     : out std_logic_vector(23 downto 0);  --Posicion del motor
    speed       : out std_logic_vector(23 downto 0); --Posicion para el calculo de velocidad
    speed_vld   : out std_logic);

end entity ; -- contador_posicion

architecture rtl of contador_posicion is

    signal pos_cnt_i : unsigned(22 downto 0);
    signal speed_i   : unsigned(22 downto 0);
    signal speed_reg : unsigned(22 downto 0);
    signal sped_vld_i  : std_logic;


begin

    Contador_velocidad : process (clk) is
    begin  -- process
     if(clk = '1' and clk'event) then
       if(rst = '1' or pos_cnt_clr = '1' or cambio = '1') then --Si detecta un reset, un cambio de direccion o fin de
periodo de muestreo
          speed_i <= (others => '0');                  --Se pone a 0
```

```
            elsif(pos_cnt_en = '1') then                    --En cada flanco detectado
          if(pos_cnt_i < 8388597) then
            speed_i <= speed_i + 1;                  --Se actualiza la posicion
          end if;
        end if;
      end if;

          end process;

    Registro_Velocidad: process (clk) is
    begin  -- process
     if(clk = '1' and clk'event) then
        if(rst = '1') then
          speed_reg <= (others => '0');
          sped_vld_i  <= '0';
        else
          if(pos_cnt_clr = '1' or cambio = '1') then
            speed_reg <= speed_i;
            sped_vld_i  <= '1';
          else
            sped_vld_i  <= '0';
          end if;
        end if;
      end if;

   end process;


Contador : process (clk) is
    begin  -- process
     if(clk = '1' and clk'event) then
        if(rst = '1' or reset_pos = '1') then
            pos_cnt_i <= (others => '0');
        elsif(pos_cnt_en = '1') then          --Al detactar el flanco
                if(pos_cnt_dir = '0') then              --Si gira a la derecha
             if(pos_cnt_i < 8388597) then    --Si no ha llegado al limite, todo 1
                pos_cnt_i <= pos_cnt_i + 1; --Se incrementa la posicion
             end if;
          else                            --Si gira a la izquierda
                if(pos_cnt_i > 0) then         --Si no ha llegado al limite
                pos_cnt_i <= pos_cnt_i - 1; --Se decrementa la posicion
             end if;
          end if;
        end if;
     end if;

end process;

   speed     <= pos_cnt_dir & std_logic_vector(speed_reg);   --Se añ?ade la direccion en el ú?ltimo bit
   pos_cnt   <= pos_cnt_dir & std_logic_vector(pos_cnt_i); --Se añ?ade la direccion en el ú?ltimo bit
   speed_vld <= sped_vld_i;

end architecture ; -- rtl
```

Anexo III: Código de la aplicación en C

```
#include <stdio.h>
#include <stdint.h>
#include <xil_printf.h>
#include <math.h>
#include "xscugic.h"
#include "xil_exception.h"
#include "xuartps.h"
#include <xparameters.h>

////////////////////////////////////PWM/////////////////////////////////////////////////////////
// Register indexes PWM
#define PWM_CTRL_REG_INDEX    0
#define PWM_PERIOD_REG_INDEX  1
#define DUTY_CYCLE_REG_INDEX  2
#define DEADTIME_REG_INDEX    3
//Bit Mask PWM
#define PWM_RST             0x01
#define PWM_ENABLE          0x02
#define DEADTIME_ENABLE     0x04
//Related to PWM
#define AXI_CLK_FREQ_HZ     100000000

#define LEFT_TURN           0x80000000
#define RIGHT_TURN          0x7FFFFFFF

////////////////////////////////////ENCODER/////////////////////////////////////////////////////////
//Register indexes Encoder
#define ENCODER_CTRL_REG_INDEX      0
#define ENCODER_PERIOD_REG_INDEX    1
#define ENCODER_POSITION_REG_INDEX  2
#define ENCODER_SPEED_REG_INDEX     3
//Bit Mask Encoder
#define ENCODER_RST             0x01
#define ENCODER_RST_POS         0x02
#define ENCODER_IRQ_ENABLE      0x04
#define ENCODER_IRQ_FLAG        0x08

//Declaracion de registros
volatile uint32_t *reg  = (volatile uint32_t *) XPAR_AXI_PWM_GENERATOR_0_S_AXI_BASEADDR; //To acess to
PWM Registers
volatile uint32_t *reg2 = (volatile uint32_t *) XPAR_AXI_ENCODER_0_S_AXI_BASEADDR;      //To acess to Encoder
Registers

XScuGic   scu_GIC_drv;          // The instance of the interrupt controller

//Global Variables
uint32_t PWM_FREQ_HZ = 10000;   //Frecuencia PWM
float    dead_time_us = 0.16;   //Tiempos muertos en us
uint32_t DEAD_TIME;
uint32_t dc_code;               //Variable que se escribe en el registro de Duty-Cycle
float    duty_UART;             //Variable Duty que se recoge de la UART
uint32_t period_code;           //Periode of PWM

uint32_t FREC_OBS_ENCODER;
uint8_t  T_obs = 10;            //Tiempo de observacion en ms
uint32_t period_encoder = 0;    //Period of Encoder
float    speed = 0;             //Variable Speed in RPM
uint32_t pulsos_position = 0;   //Position
uint32_t pulsos_speed = 0;      //Speed Position
uint8_t  direction_speed = 0;   //Derection from Speed
uint8_t  direction_position = 0; //Derection from Position
uint8_t  token = 0;             //Se dispone de nuevo valor de velocidad
```

```
//////////////////////CONTROL/////////////////////////////////////////////////////
#define N_SAMPLES  1700
#define Ko         789
#define Tau        0.13e-3
#define Gamma      5.41e-6
#define Kp         (1.2*Gamma)/(Ko*Tau)
#define Ti         2*Tau
#define Td         0.5*Tau

static float dc_ref[N_SAMPLES];
static float speed_log[N_SAMPLES];
static int   dc_ref_idx = 0;

float u = 0;
float u_prev = 0;
float e = 0;
float e_prev = 0;
float sum_e = 0;


void set_PWM()
{

  reg[PWM_CTRL_REG_INDEX] |= PWM_RST;           //Reset PWM
  reg[PWM_CTRL_REG_INDEX] &= ~PWM_ENABLE;       //Disable PWM
  reg[PWM_CTRL_REG_INDEX] &= ~DEADTIME_ENABLE;  //Disable Deadtime

  xil_printf("\nIntroduce frecuencia de la señ?al PWM [en Hz]: ");
  scanf("%d", &PWM_FREQ_HZ);

  period_code = (AXI_CLK_FREQ_HZ/PWM_FREQ_HZ)-1;
  reg[PWM_PERIOD_REG_INDEX] = (period_code<<4);  //Set PWM Period
  reg[DUTY_CYCLE_REG_INDEX] = 0;                 //Duty-Cycle 0

  xil_printf("\nIntroduce duracion de tiempo muerto [us]: ");
  scanf("%f", &dead_time_us);
  DEAD_TIME = (AXI_CLK_FREQ_HZ/(1/(dead_time_us * 10e-6))) - 1;

  reg[DEADTIME_REG_INDEX] = (DEAD_TIME<<12);     //Set Deadtime

  reg[PWM_CTRL_REG_INDEX] &= ~PWM_RST;          //Release the reset
  reg[PWM_CTRL_REG_INDEX] |= PWM_ENABLE;        //Enable PWM
  reg[PWM_CTRL_REG_INDEX] |= DEADTIME_ENABLE;   //Enable Deadtime
}

void PWM_Duty_Cycle(float duty_cycle)
{

  dc_code = (uint32_t) (period_code*duty_cycle);
  reg[DUTY_CYCLE_REG_INDEX] = (dc_code<<8);
}

void set_left_turn_direction()
{

  reg[PWM_PERIOD_REG_INDEX] |= (LEFT_TURN);  //Set left turn direction
}

void set_right_turn_direction()
{

  reg[PWM_PERIOD_REG_INDEX] &= (RIGHT_TURN); //Set right turn direction
}

void cambio(float valor)
{
```

```
  if(valor < 0)
  {
  set_left_turn_direction();
  }
  else
  {
  set_right_turn_direction();
  }

  valor = fabs(valor);

  PWM_Duty_Cycle(valor);

}

void encoder_clr_position()
{

  reg2[ENCODER_CTRL_REG_INDEX] |= ENCODER_RST_POS;     //Reset Position Counter
}

void encoder_clr_flag()
{

  reg2[ENCODER_CTRL_REG_INDEX] |= ENCODER_IRQ_FLAG;    //Quit IRQ Flag writing 1
}

void encoder_set_observation_period(int period_obs)
{

  reg2[ENCODER_PERIOD_REG_INDEX] = (period_obs <<0);   //Set Observation Period2 with offset
}

void encoder_get_position()
{

  direction_position = ((reg2[ENCODER_POSITION_REG_INDEX] & (0x1<<31))>>31);  //Get Direction from Position
  pulsos_position = (reg2[ENCODER_POSITION_REG_INDEX] & (0x7FFFFF<<0));       //Get Position from register
}

void encoder_get_speed()
{

  direction_speed = ((reg2[ENCODER_SPEED_REG_INDEX] & (0x1<<31))>>31);  //Get Direction
  pulsos_speed = (reg2[ENCODER_SPEED_REG_INDEX] & (0x7FFFFF<<0));       //Get Speed without Direction
}

void encoder_IRQ_Handler()
{

  encoder_clr_flag();              //Clear IRQ Flag

  encoder_clr_position();           //Clear Position

  encoder_get_speed();             //Get Pulsos

  speed = (pulsos_speed* FREC_OBS_ENCODER) * (60)/(12 * 19.225);

//////IMPLEMENTACION SISTEMA DE CONTROL////////////
  /*e = speed - u;
  sum_e += e;
  u = Kp*e+(Kp/Ti)*(sum_e+e)+Kp*Td*(e-e_prev);
  e_prev = e;

  if(e_prev > e)
  {
```

```
    duty_UART -= u;
    cambio(duty_UART);
  }
  else if(e_prev < e)
  {
    duty_UART += u;
    cambio(duty_UART);
  }*/

//////ADQUISICION MODELO DE LA PLANTA///////////

  /*if(dc_ref_idx < N_SAMPLES)
  {
    encoder_clr_flag();              //Clear IRQ Flag
    encoder_get_speed();               //Get Pulsos
    speed = (pulsos_speed* FREC_OBS_ENCODER) * (60)/(12 * 19.225);

    if(direction_speed == 0)
    {
      speed_log[dc_ref_idx] = speed;
    }
    else
    {
      speed_log[dc_ref_idx] = -speed;
    }

    duty_UART = dc_ref[dc_ref_idx];

    cambio(duty_UART);

    dc_ref_idx++;
  }*/

  token = 1;
}

void set_Encoder()
{

  reg2[ENCODER_CTRL_REG_INDEX] |= ENCODER_RST;          //Reset Encoder

  encoder_clr_position();                 //Reset Position Counter
  encoder_clr_flag();                    //Quit IRQ Flag

  xil_printf("\nIntroduce periodo de observacion en ms: ");
  scanf("%d", &T_obs);
  FREC_OBS_ENCODER = 1/(T_obs * 10e-3);

  period_encoder = (AXI_CLK_FREQ_HZ/FREC_OBS_ENCODER)-1;

  encoder_set_observation_period(period_encoder);       //Set Observation Period

  reg2[ENCODER_CTRL_REG_INDEX] |= ENCODER_IRQ_ENABLE;    //Enable IRQ

  reg2[ENCODER_CTRL_REG_INDEX] &= ~ENCODER_RST;          //Reset Encoder
}

void cnt_encoder_IRQ(XScuGic *intc_drv)
{

  XScuGic_Config *scu_GIC_cfg;

  //Finalizo las interrupciones
  Xil_ExceptionDisable();

  //Configuro GIC
  scu_GIC_cfg = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
```

```
  XScuGic_CfgInitialize(intc_drv, scu_GIC_cfg, scu_GIC_cfg->CpuBaseAddress);

  //Registramos el gestor de interrupciones del GIC
  Xil_ExceptionRegisterHandler(
     XIL_EXCEPTION_ID_IRQ_INT,
     (Xil_ExceptionHandler) XScuGic_InterruptHandler,
     intc_drv);

  //Inicializar periferico
  set_Encoder();

  //Realizo la conexion
  XScuGic_Connect(
        intc_drv,
        XPAR_FABRIC_AXI_ENCODER_0_INTERRUPT_INTR,
        (Xil_ExceptionHandler) encoder_IRQ_Handler,
        (void *) NULL);

  // Enable the interrupt for the device.
  XScuGic_Enable(intc_drv, XPAR_FABRIC_AXI_ENCODER_0_INTERRUPT_INTR);

  // Enable interrupts in the processor.
  Xil_ExceptionEnable();
}

int main()
{

  set_PWM();                    //Configurate PWM IP
  cnt_encoder_IRQ(&scu_GIC_drv);    //Configurate Encoder Interruption and Encoder IP

  while(1)
  {

    if(token == 1)
    {

      if(direction_speed == 1)
      {
        xil_printf("\n %d -> Direccion de giro antihorario", direction_speed);
      }
      else
      {
        xil_printf("\n %d -> Direccion de giro horario", direction_speed);
      }

      printf("\nSpeed: %.2f RPM\n", speed);

      token = 0;
    }

    xil_printf("\n Introduzca 'a' para modificar el ciclo de trabajo o ENTER para actualizar el valor de velocidad");

    if(XUartPs_RecvByte(XPAR_PS7_UART_1_BASEADDR) =='a')
    {
       xil_printf("\nIntroduce ciclo de trabajo [-1.0, 1.0]: ");
       scanf("%f", &duty_UART);

       cambio(duty_UART);
    }

  }

  return 0;
}
```