Universidad Autónoma de Querétaro

Facultad de ingeniería

Maestría en instrumentación y control automático

SISTEMA EMBEBIDO DE CONTROL DE MOVIMIENTO PARA ROBOT POLIARTICULADO

TESIS

Que como parte de los requisitos para obtener el grado de

Maestro en Ciencias (Instrumentación y control automático)

Presenta:
Salvador Manuel Malagón Soldara

Dirigido por:
Dr. Edgar Alejandro Rivas Araiza

SINODALES

Dr. Edgar Alejandro Rivas Araiza	
Presidente	Firma
MC. Carlos Ricardo Luna Ortiz	
Secretario	Firma
Dr. Manuel Toledano Ayala	
Vocal	Firma
MI. Luis Miguel Contreras Medina	
Suplente	Firma
Dr. Juevenal Rodríguez Reséndiz	
Suplente	Firma

Dr. Gilberto Herrera Ruíz
Director de la Facultad

Dr. Luis Gerardo Hernández Sandoval
Director de Investigación y Posgrado

Centro Universitario
Querétaro, Qro.
Noviembre de 2011
México

RESUMEN

En este trabajo se presenta un sistema híbrido de cómputo que contempla la interacción entre un DSP (Digital Signal Processor) *eZdsp TMS320F2812* y un FPGA (Field Programmable Gate Array) *DE1 Cyclone II Starter Board* por medio de la interfaz McBSP (Multichannel Buffered Serial Port), cada uno desempeñó diferentes tareas aprovechando las fortalezas de cada dispositivo para delegar las actividades. El sistema fue implementado como una tarjeta de control con módulos de interacción externa para un robot PUMA (Programmable Universal Machine for Assembly). Dentro del FPGA, se programó un algoritmo adaptivo para medición de velocidad en encoders, tomado de una propuesta teórica elaborada por Lygoras en 1998. En dicha referencia existen dos maneras para medir la velocidad: la primera es a través de la frecuencia de giro para altas velocidades, mientras que la segunda utiliza los ciclos de un canal del encoder como referencia para eliminar el error imprescindible en la medición de velocidad cuando se utilizan encoders para medir bajas velocidades. El algoritmo adaptivo se calcula a través de una tabla propuesta en esta misma referencia donde existen diferentes valores para el ciclo medido en un canal del encoder. Cada valor de este periodo representa una velocidad en RPM y un intervalo de muestreo para volver a realizar las mediciones, tomando como premisa que no existen cambios bruscos en la velocidad. Por otra parte, se desarrolló un control PID (Proportional-Integral-Derivative) dentro del DSP, el diseño fue obtenido a partir del *PMSM 3.1 Control Motor System* de *Texas Instruments*, proyecto que contiene las librerías necesarias para el control de motores. Del diseño se obtiene un alto rendimiento, comprobando la factibilidad de desarrollo para el mismo sistema con procesadores más avanzados que los utilizados para la validación de los resultados de este proyecto.

(**Palabra clave**: Medición de velocidad, Algoritmo Adaptivo, Control de motor, DSP, FPGA)

SUMMARY

This work presents a hybrid computing system that contemplates the interaction between a DSP (Digital Signal Processor) TMS320F2812 eZdsp and a FPGA (Field Programmable Gate Array) Cyclone II Starter Board DE1 through the interface McBSP (Multichannel Buffered Serial Port), each performed different tasks taking advantage of the strengths of every device to delegate the activities. The system was implemented as a control board with external interaction modules for a robot PUMA (Programmable Universal Machine for Assembly). Within the FPGA, an adaptive algorithm was programmed to measure speed encoders, taken from a theoretical proposal developed by Lygoras in 1998. In this reference there are two ways to measure the speed: the first is through the rotation frequency for high speeds, while the second uses the cycles of a reference channel encoder as necessary to eliminate the error in the measurement of speed when encoders are used to measure low speeds. The adaptive algorithm is calculated through a table proposed in this same reference where are different values for the cycle measured in a channel encoder. Each value of this period represents a speed in RPM and a sampling interval for re-measurements, on the premise that there are no sudden changes in speed. We developed a PID (Proportional-Integral-Derivative) within the DSP, the design was obtained from 3.1 Control PMSM Motor System of Texas Instruments, a project that contains the necessary libraries to the motor control. Design of high performance is obtained, proving the feasibility of developing the same system with advanced processors used for the validation of the results of this project.

(**Key words**: Velocity Measurement, Adaptive Algorithm, Motor Control, DSP, FPGA)

A mis padres, que con su apoyo y cariño hacen posible cada una de mis metas.

A mis hermanos, para quienes siempre he tratado de ser el mejor ejemplo.

AGRADECIMIENTOS

Al Consejo Nacional de Ciencia y Tecnología (CONACyT), por el apoyo proporcionado para mis estudios de posgrado. De la misma manera agradezco a la Universidad Autónoma de Querétaro (UAQ) por proporcionar el equipo y un lugar de trabajo agradable para mis estudios.

Agradecimiento especial a todos mis compañeros de generación, por crear la mejor de las atmósferas para trabajar. A mi asesor, el Dr. Edgar Rivas, por su paciencia y disponibilidad a lo largo de toda mi estadía. A mis compañeros de sala en el B9 del CEDIT.

A mi familia en general por el cariño brindado, en especial a Lucía Soldara por haberme dado un techo durante mis estudios.

Por último gracias a Dios, ya que gracias a él, el todo es posible.

ÍNDICE DE CONTENIDO

Página

RESUMEN i

SUMMARY ii

AGRADECIMIENTOS iv

ÍNDICE DE CONTENIDO v

ÍNDICE DE TABLAS vii

ÍNDICE DE FIGURAS viii

I INTRODUCCIÓN 1

I.1 Justificación 4

I.2 Planteamiento del problema 10

I.3 Hipótesis y objetivos 11

I.3.1 Hipótesis general 11

I.3.2 Objetivo general 12

I.3.3 Objetivos específicos 12

II REVISIÓN DE LITERATURA 13

II.1 Antecedentes 13

II.1.1 Nacimiento de los sistemas híbridos de control 13

II.1.2 Actualidad 15

II.2 Consecuentes 20

III METODOLOGÍA 21

III.1 Marco teórico 21

III.1.1 PMSM 3.1 Control Motor System 24

III.1.2 Teoría de controladores PID (Proporcional Integral Derivativo) 29

III.1.3 Sintonización de un PID 37

III.1.4 Motor de CD serie 38
III.2 Metodología 40
III.2.1 Características del módulo McBSP 40
III.2.2 Código en lenguaje C para comunicación McBSP 44
III.2.3 Transmisor en FPGA para McBSP 45
III.2.4 Receptor en FPGA para McBSP 47
III.2.5 Algoritmo adaptivo para medición de velocidad 50
III.2.1 Código en lenguaje C para control de motor 59
IV RESULTADOS Y DISCUSIÓN 63
IV.1 Resultados 63
IV.1.1 Algoritmo adaptivo de velocidad 63
IV.1.2 Interfaz McBSP 67
IV.1.3 Sintonización de controlador PID 70
IV.1.4 Etapa de potencia 75
IV.2 Conclusión 79
LITERATURA CITADA 80
APÉNDICE 86
A. Artículo publicado en 7mo. Congreso Nacional de Ingeniería 87
B. Póster publicado en 5to. Coloquio de Investigación (Postgrado, FI) 92
C. Código McBSP con loopback digital (McBSPLoopback.c) 93
D. Archivo de cabecera para McBSP con loopback digital (McBSPLoopback.h) 95
E. Código McBSP en FPGA (McBSP.vhd) 97
F. Código PID para control de posición de motor en DSP (PID_Tesis.c) 98
G. Código de Algoritmo adaptivo de velocidad (Speed.vhd) 103
H. Código Decodificador para Algoritmo adaptivo de velocidad (Decoder.vhd) 106

ÍNDICE DE TABLAS

Página

Tabla I.1 Principales características del DSP TMS320F2812 6

Tabla I.2 Principales características de FPGA Cyclone II 9

Tabla III.1 Variables de entradas y de salida en la estructura QEP_NO_INDEX 25

Tabla III.2 Variables de entradas y de salida en la estructura ESTIMATEDSPEED 26

Tabla III.3 Variables de entradas y de salida en la estructura PID_REG 27

Tabla III.4 Variables de entradas y de salida en la estructura BDC_PWM_DRV 28

Tabla III.5 16 archivos de cabecera incluidos por DSP281x_Device.h 44

Tabla III.6 Tabla de verdad para arreglo lógico de detección de dirección 53

Tabla III.7 Tabla de valores adaptivos para la velocidad 57

Tabla IV.1 Tabla de valores para Cp y Ct, obtenidos mediante praxis 63

Tabla IV.2 Tabla de velocidades dependiendo el voltaje de alimentación 65

ÍNDICE DE FIGURAS

Página

Figura I.1 Kit de desarrollo eZdspTMS320F2812 2

Figura I.2 Cyclone II FPGA Starter Board 3

Figura I.3 Diferentes usos para las familias de DSPs (C28x Workshop, TI) 5

Figura I.4 Diagrama general del sistema 7

Figura I.5 Pieza para realizar las pruebas de funcionalidad de robot PUMA 11

Figura III.1 Dimensiones físicas del DSP TMS320F2812 23

Figura III.2 Encapsulado de FPGA ALTERA EP2C20F484C7N 24

Figura III.3 Bloque de entradas y salidas de QEP_NO_INDEX_DRV 25

Figura III.4 Bloque de entradas y salidas de ESTIMATEDSPEED 26

Figura III.5 Bloque de entradas y salidas de PID_REG3 27

Figura III.6 Bloque de entradas y salidas de BDC_PWM_DRV 28

Figura III.7 Diagrama a bloques de programación en TMS320F2812 29

Figura III.8 Estructura del PID usado por PID_REG3 35

Figura III.9 Circuito más básico para un motor CD 39

Figura III.10 Triple buffer para la recepción y doble buffer para transmisión en McBSP (SPRU061A, TI) 41

Figura III.11 Número de bits por palabra en el McBSP (SPRU061A, TI) 42

Figura III.12 Palabras enviadas por una Estructura en el McBSP (SPRU061A, TI) 43

Figura III.13 Selección de canal dentro del McBSP (SPRU061A, TI) 43

Figura III.14 FSM_TransmisorMcBSP 46

Figura III.15 Diagrama a bloques del Transmisor por McBSP 47

Figura III.16 Muestreo para adquisición de datos de McBSP en FPGA 48

Figura III.17 FSM_ReceptorMcBSP 49

Figura III.18 Diagrama a bloques del Receptor por McBSP 50

Figura III.19 Aumento al cuádruple la resolución del encoder (Lygouras, 1998)..... 52

Figura III.20 Detección de giro dentro del algoritmo adaptivo 53

Figura III.21 Diagrama general a bloques del algoritmo adaptivo 54

Figura III.22 Medición de la velocidad, diagrama a bloques 55

Figura III.23 FSM_PMC..... 56

Figura III.24 FSM_SPEED..... 58

Figura III.25 Fotografía de validación del algoritmo adaptivo con motor de CD 59

Figura III.26 Diagrama de conexión DSP/Motor/Encoder 61

Figura III.27 Diagrama general del sistema..... 62

Figura IV.1 Coeficientes Ct y Cp, exhibidos en los Displays de 8 segmentos de la DE1..... 64

Figura IV.2 Diagrama RTL (Resistor Transistor Logic) de algoritmo adaptivo para medición de velocidad 66

Figura IV.3 Fotografía de sistema implementado para medición de velocidad 67

Figura IV.4 Captura de pantalla tomada en el TLA5201B que muestra 4 niveles en el FIFO..... 68

Figura IV.5 Señales provenientes del DSP adquiridas por medio del TLA5201B..... 69

Figura IV.6 Modelo a bloques del comportamiento del motor..... 70

Figura IV.7 Cálculo de 1/a para la caracterización del motor acorde a un impulso de 4 V 71

Figura IV.8 Diagrama a bloques en Simulink para graficar la respuesta del motor de CD..... 71

Figura IV.9 Gráfica de la simulación de la respuesta del motor, obtenida a partir de Simulink .. 72

Figura IV.10 Acercamiento en imagen para obtener el valor exacto de 1/a..... 73

Figura IV.11 Señal real de respuesta del motor adquirida a partir de Borland C y RS-232..... 74

Figura IV.12 Impresión del osciloscopio TDS2024B con las señales de PWM correspondientes a los valores en el Watch Window 76

Figura IV.13 Circuito interno del L298N 77

Figura IV.14 Fisiología del L298N (DataSheet, STMicroelectronics)..... 78

Figura IV.15 Uso de compuerta lógica OR para configuración de las salidas PWM..... 78

CAPÍTULO 1:

I INTRODUCCIÓN

Desde la existencia del ámbito tecnológico, constantemente se ha producido la necesidad de crear nuevos y mejores procesos, no importa cuál y dónde esté su aplicación siempre se buscará más comodidad, velocidad, potencia y flexibilidad. Por ejemplo, con el paso de los años los televisores en nuestros hogares han evolucionado en forma, tamaño y velocidad de procesamiento, todo esto para ser un aparato más cómodo ya que con una pantalla de mayor tamaño las imágenes son más fáciles de visualizar. El procesamiento de las mismas se realiza con mayor velocidad mejorando la nitidez, además, en relación a la forma de los aparatos se han hecho más delgados disminuyendo el espacio ocupado hasta en un 80%. De esta manera todos y cada uno de los sistemas que conforman el ambiente tecnológico, no importa si son industriales o domésticos, deben ir evolucionando conforme a las expectativas y necesidades de los usuarios.

Entre los diferentes ámbitos donde existe innovación tecnológica, el industrial es uno de los más importantes debido a la cantidad de dinero que se encuentra invertido, dicho dinero se ve reflejado en la velocidad y efectividad de los sistemas, provocando grandes ganancias para los dueños de las mejores tecnologías. Así, aprovechando la rentabilidad de productos orientados en este camino, el siguiente trabajo propone una tarjeta híbrida de control para un robot tipo PUMA (Programmable Universal Machine for Assembly), provista con una gran flexibilidad en su diseño y programación.

Esta tarjeta de control estará compuesta por dos elementos principales: un FPGA (Field Programmable Gate Array) y un DSP (Procesador Digital de Señales), que son circuitos integrados para procesamiento de datos, parecidos a los procesadores que contiene cualquier computadora. Los dispositivos van a interactuar por medio de una interfaz McBSP (Multi-Channel Buffered Serial Port), además de concebir un sistema híbrido debido a la heterogeneidad entre sus componentes. La elección de este proyecto surge a través de la necesidad de tener controladores fáciles de programar, como lo son los DSPs y un elemento con una alta velocidad de procesamiento (el FPGA), que permita la formación de un híbrido para la fácil interacción con otros dispositivos de cálculo y el rápido procesamiento de grandes cantidades de datos. Esta

unión entre dispositivos formará una tarjeta de control para un brazo robótico tipo PUMA de 6 grados de libertad (Sawhney, 2008).

La justificación para la búsqueda de innovación en tarjetas controladoras híbridas se basa en un hecho simple, las aplicaciones exigen cada vez cantidades más grandes de procesamiento de datos, además de sufrir un constante cambio estructural y programático debido a las constantes actualizaciones necesarias en cualquier proceso, complicando la existencia de un alto rendimiento dentro una programación homogénea (Abdin, 2008). Para responder a esta necesidad las empresas que elaboran procesadores han dividido sus productos para ser especializados y optimizados en una tarea en específico, por ejemplo, Texas Instruments tiene una amplia lista de modelos en su gama de procesadores digitales, sin embargo cada uno se especializa en una aplicación diferente (Fernández, 2010); los hay para procesamiento de voz, de video, de operaciones de punto fijo, etc.

Existe conjunto de procesadores bastante robusto que desempeña labores diferentes a pesar de todos ser procesadores de señales digitales de la misma casa productora. Esta especialización en una sola tarea, es provocada por la necesidad de crear sistemas que trabajen de manera optimizada, sacrificando flexibilidad pero con una estructura en su programación interna que incrementa su desempeño al máximo (Maguire, 2007).

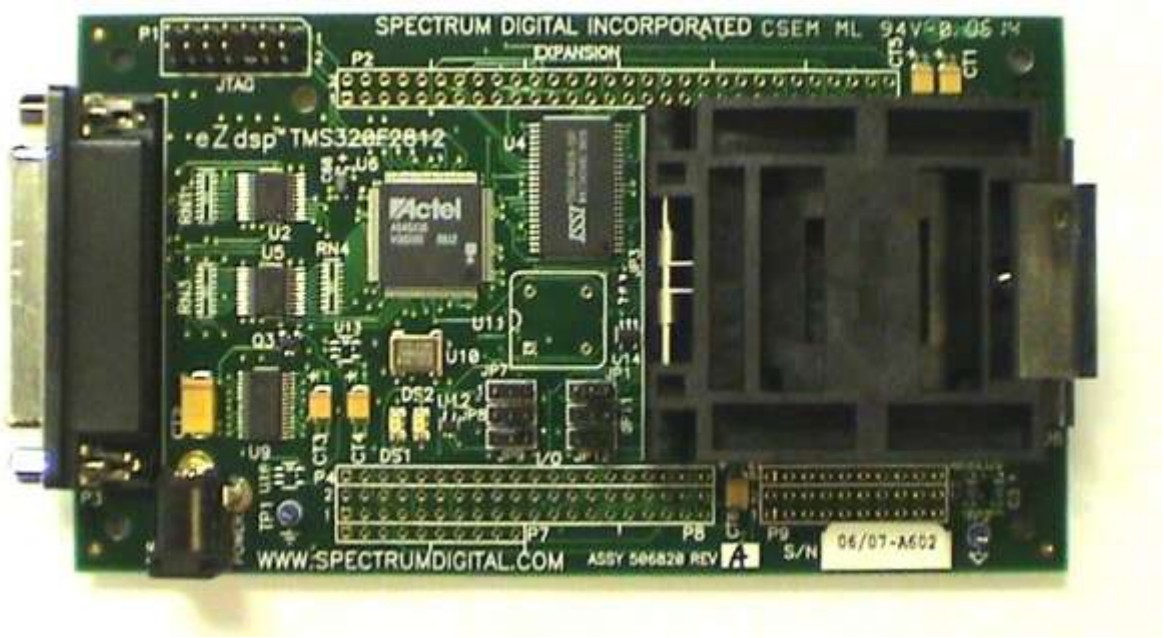


Figura I.1 Kit de desarrollo eZdspTMS320F2812

Para esto existe actualmente una nueva filosofía en el diseño de hardware, en la cual se interconectan diferentes tipos de procesadores especializados en hacer una tarea en específico (Hampel, 2008), así cada uno desempeña una actividad distinta (para la que fue creado, por ejemplo un TMS320DM6435 procesando video, la cual es su tarea innata), intercambiando resultados entre ellos, de esta manera se puede lograr una sinergia y explotar el rendimiento del conjunto de componentes sobre su capacidad tope (Devrim, 2003; Kayak, 2004; Faes, 2007). La contraparte y forma de diseño más común, es el programar todos los tipos de procesamiento en un mismo dispositivo, sin importar la naturaleza de ambos (procesamiento y dispositivo).

Los kits elegidos dentro de este trabajo fueron el eZdspF2812 (**Figura I.1**) y el Cyclone II Starter Board (**Figura I.2**) y debido a sus características de compatibilidad y especialización en control de motores, respectivamente. Además dentro del FPGA se programó un algoritmo adaptivo de medición de velocidad de alta precisión (Lygouras, 1998; Lygouras, 2009 y Batí, 1997). Así, realizando una sinergia debido a las características de ambos kits de desarrollo, se formó un sistema de control de motores con una gran flexibilidad de usos y fácil de implementar.

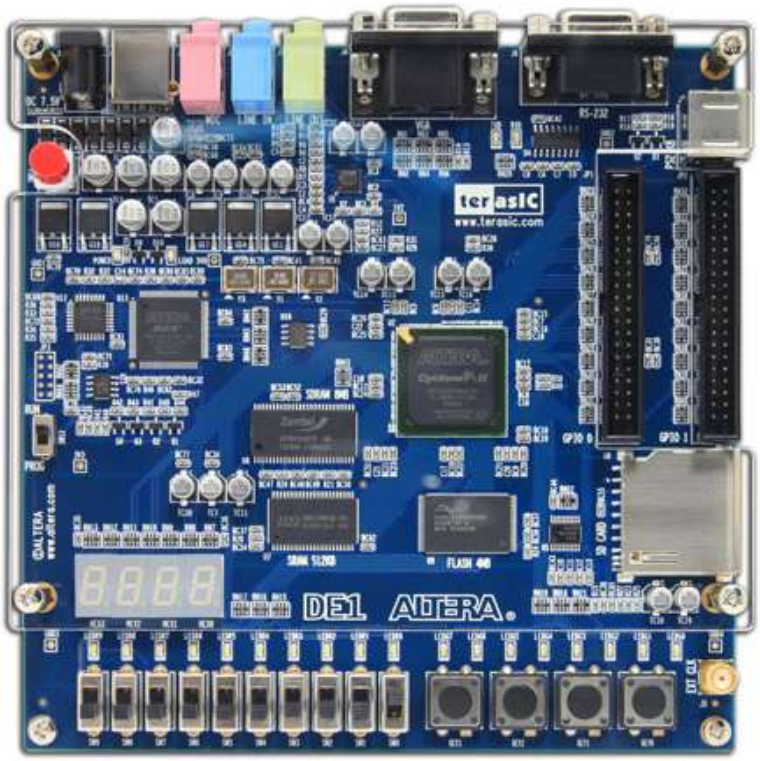


Figura I.2 Cyclone II FPGA Starter Board

I.1 Justificación

Los sistemas híbridos han adquirido gran auge debido a su poderosa capacidad de procesamiento, característica muy apreciada en las tarjetas controladoras de robots. Por otra parte se tiene la facilidad de utilizar procesadores distintos, obteniendo así una gran flexibilidad en el número de aplicaciones al poder añadir un dispositivo de procesamiento dependiendo de la actividad desempeñada. En cuanto a este trabajo la flexibilidad se logrará colocando un controlador (FPGA maestro) como núcleo, es decir, todos los periféricos controlables del sistema serán coordinados por él, ofreciendo una conexión amigable y formando un punto de recaudación de la información más indispensable para un fácil acceso hacia ella. Así, las tareas más complicadas pueden ser programadas sobre los dispositivos más sencillos (esclavos), entendiendo por sencillos, el poder ser programados en lenguajes de mayor nivel. Por su parte el núcleo permanece fijo para interconectarlos e intercambiarlos, logrando en conjunto una tarjeta de control tan robusta como la tarea lo demande (Seth Koehler, 2008).

Este concepto además puede facilitar el procedimiento para agregar mecanismos a un sistema ya estructurado, simplemente programando el controlador adecuado dentro de un procesador sencillo, de aquí sólo resta comunicarlo con nuestro núcleo para comenzar su funcionamiento.

El FPGA seleccionado fue el Cyclone II FPGA Starter Board debido a su espacio de trabajo GPIO con un gran número de pines, divididos en la tarjeta de desarrollo en dos puertos con 36 entradas/salidas de propósito general cada uno, además de alimentación de 5 y 3.3 V con dos tierras. Sin embargo en las investigaciones comprendidas entre 2009 y 2011 se encontraron kits de desarrollo más baratos en el mercado. Así las recomendaciones debido a su tamaño (un cuarto del tamaño de la Cyclone II), su compatibilidad por la capacidad de ser programada vía USB sin uso de cables y un bajo costo; son el FPGA MAX II Micro Kit y la DE0-Nano Development and Education Board (69 y 79 USD respectivamente, tarjetas de desarrollo en venta por Terasic.com al mes de Octubre de 2011). En trabajo posterior a este proyecto se podría realizar una PCB (Printed Circuit Board) con FPGAs mucho más baratas, como es el caso de la AT40K05-2RQC-ND de Atmel, la cual contiene 256 elementos lógicos (nuestro proyecto ocupa un total de 246 elementos lógicos) y tiene un precio de 1.056 USD (www.digikey.com, 2011).

Hablando ahora sobre la justificación en el uso de un elemento híbrido para la programación del sistema, se pueden mencionar algunas desventajas acerca del empleo de cada uno de los elementos por separado; el FPGA es un dispositivo en el cual necesitas algoritmos como el CORDIC (Dick, 2008; Volder, 1959) para programar funciones trigonométricas, por lo tanto cuando se desean programar diseños grandes, tales como: PIDs, RNs (Redes Neuronales), FUZZY, etc., el tiempo de programación alcanza semanas o hasta meses dependiendo de la experiencia del diseñador. Sin embargo, esto se puede evitar con el uso de lenguajes de programación de más alto nivel, ventaja obtenida al usar DSPs, ya que su lenguaje de programación está basado en una variación de C. Por otro lado, también es equivocado tratar de programar el sistema en toda su extensión dentro del DSP, debido a que el TMS320F2812 (DSP orientado al control de motores, **Figura I.3**) sólo contiene dos entradas de lectura para QEP (Cuadratura de Pulso de Encoder): CAP1 y CAP2 (Capure Input 1 y 2) para EVA (Event Manager A); CAP3 y CAP4 para EVB (SPRS174R, 2010; Toliyat, 2003). De esta manera, y tomando en cuenta que el número de articulaciones de nuestro robot asciende a 6, el uso exclusivo del DSP también está descartado. Sin embargo, cuenta con todas las características para desempeñar interfaces McBSP como se puede ver dentro de las características mostradas en la **Tabla I.1.**

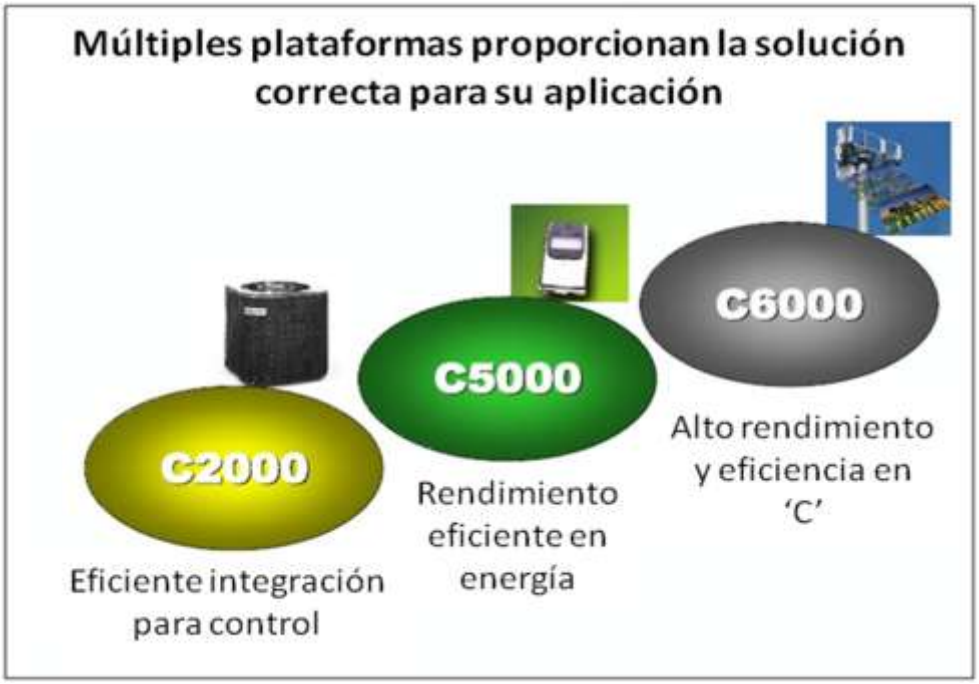


Figura I.3 Diferentes usos para las familias de DSPs (C28x Workshop, TI)

Tabla I.1 Principales características del DSP TMS320F2812

Características	TMS320F2812
Generación	28x Fixed-poing Series
CPU	1 C28x
Peak MMACS	150
Frecuencia (MHz)	150
SRAM	36 KB
OTP ROM	2 KB
Flash	256 KB
EMIF	1 16-Bit
PWM	16-Ch
CAP/QEP	06-feb
ADC	1 16-Ch 12-Bit
ADC Conversion Time	80 ns
McBSP	1
UART	2 SCI
SPI	1
CAN	1
Relojes	3 32-Bit GP, 1 WD
GPIO	56
Alimentación de núcleo (Volts)	1.9 V
IO Alimentación (V)	3.3 V
Rango de temperatura (°C)	40 to 85, 40 to 125

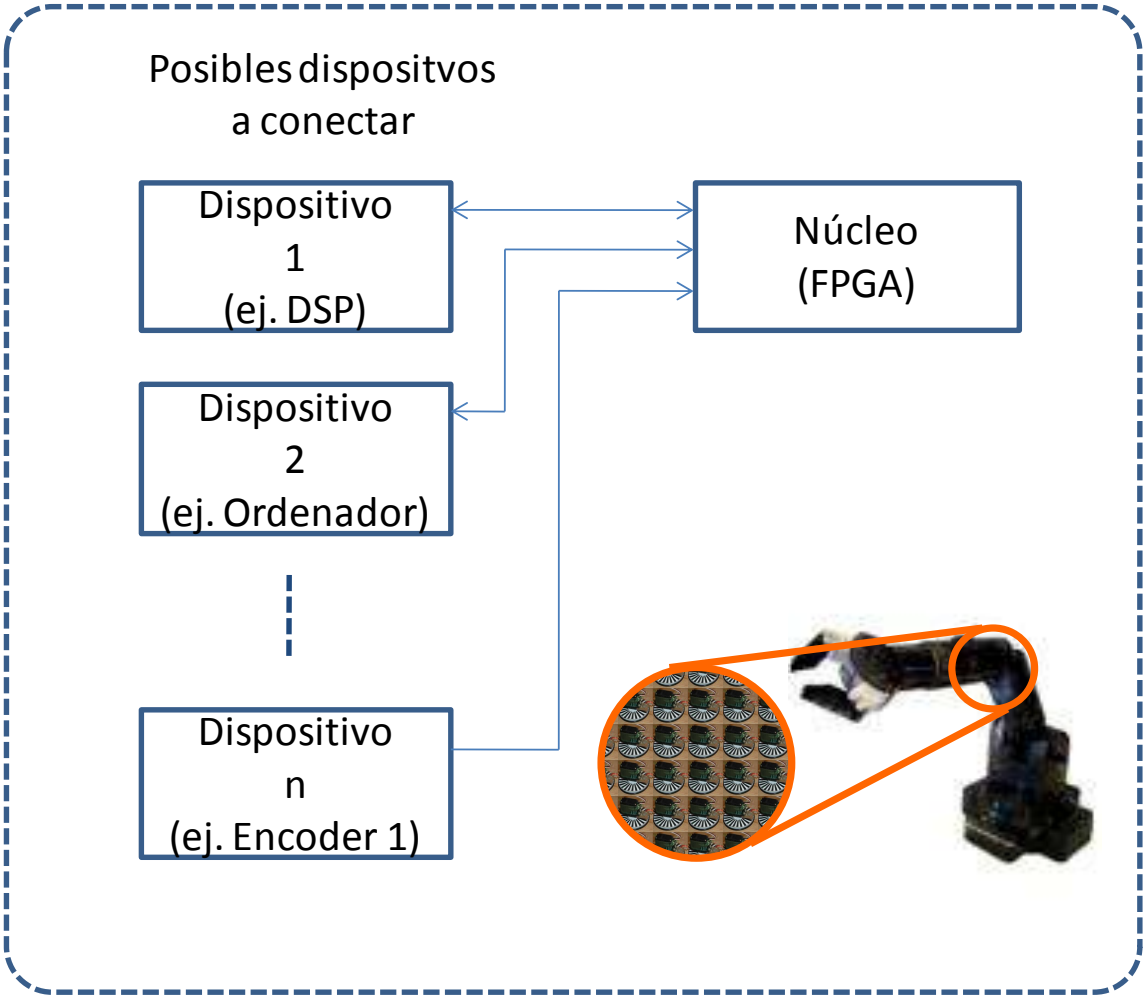


Figura I.4 Diagrama general del sistema

Desde otro punto de vista, pero dentro de las facilidades que ofrece la tarjeta híbrida es la elección del dispositivo controlador (modelo del microchip), ésta siempre es una labor difícil debido a la variedad de modelos, tomando en cuenta que cada uno es elaborado por las empresas comerciales para una tarea en específico, y normalmente cuando controlamos un sistema, utilizamos un solo controlador para desarrollar muchos tipos de tareas, eligiéndolo sólo por tipo de labor principal.

Por ejemplo, si se desea analizar un video para identificar a un criminal por medio de la identificación del rostro, los resultados se obtienen a partir del procesamiento de la imagen comparándola con una base de datos, sin embargo, podemos encontrar que también se requiere analizar el audio para identificar la voz y corroborar su identidad. Con un controlador común o

GPP (General Purpose Processor) la velocidad de procesamiento se reduce, en comparación al uso de un DSP orientado al procesamiento de imágenes para la parte visual y otro DSP para el sonido, ambos especializados en imágenes y voz, respectivamente. Así, podemos emplear cualquier cantidad de controladores para optimizar nuestro sistema (Pinto, 2006). Para detallar mejor las conexiones entre controladores, se ha creado un diagrama general del sistema el cual se exhibe en la **Figura I.4**.

Como se puede apreciar todas las terminales están conectadas con el núcleo, el cual dirige los datos hacia su destino, es decir, recibirá las lecturas de los encoders y las enviará hacia el DSP (Gao, 2006) especializado en control de motores, éste realiza una respuesta traducida en movimiento directamente en los motores por medio de un PWM (Pulse Width Modulation), obteniendo una velocidad, aceleración y energía de consumo controlada (Osornio, 2008).

Como antes fue mencionado, existe mucho dinero invertido dentro del ámbito industrial, así que para las empresas es de gran interés cualquier investigación o innovación que mejore sus líneas de producción, de esta manera solucionan problemas tales como: seguridad, consumo de energía, tiempo de fabricación, etc. Sin embargo, en la investigación no siempre se obtienen resultados satisfactorios, pese a mencionados fallos en la conclusión de algunos proyectos, el porcentaje de avances científicos y tecnológicos que logran instalarse son de gran ayuda para mejorar los tiempos y los costos en la producción (entre otros factores). Consecuentemente, un sistema automatizado en la mayoría de los casos ofrecerá una solución viable para producciones en masa, por lo tanto, crear e investigar dentro de este rubro siempre será un negocio bien remunerado. El concepto propuesto por este trabajo es un novedoso sistema de control que combina velocidad de procesamiento con flexibilidad en el número de aplicaciones; como puede ser el añadir un control de motores para un CNC, por supuesto, cambiando las debidas variables a controlar (Romero, 2004). Esto se logra colocando un circuito integrado que soporte la velocidad de procesamiento con el algoritmo de control adecuado, conectado al núcleo de procesamiento para obtener sus variables de entrada y reportar sus resultados (y la DE1 cumple con una gran compatibilidad para formar interfaces con otros dispositivos **Tabla I.2**). De manera similar, se puede lograr la implementación en cualquier sistema donde se requieran controlar actuadores mecánicos o simplemente obtener una serie de señales deseadas.

Tabla I.2 Principales características de FPGA Cyclone II

Altera Cyclone II 2C20 FPGA con 20000 LEs (Elementos lógicos)
Dispositivos con configuración serial Altera (EPCS4) para Cyclone II 2C20
USB Blaster construido sobre una tarjeta para programar y usar control API
Modo JTAG y AS son soportados
8Mbyte (1M x 4 x 16) SDRAM
4Mbyte Flash Memory
512Kbyte(256Kx16) SRAM
Puerto para tarjetas SD
4 Botones de presión
10 botones DPDT
8 LEDs verdes
10 LEDs rojos
4 pantallas de siete segmentos
Oscilador de 50MHz, 24MHz, 27MHz y soporte para oscilador externo
24-bit Calidad-CD Audio CODEC con línea de entrada, salida, y micrófono
VGA DAC (4-bit R-2R por canal) con puerto de salida VGA
Transceptor RS-232 y conector de 9 pines
Puerto PS/2 (ratón/teclado)
Dos puertos de expansión de 40 pines
DE1 Lab CD-ROM con códigos de ejemplo
26 multiplicadores (9x9 bits y 18x18 bits)

Básicamente, todo esto es lo que mejoraran las tarjetas híbridas con su diseño, ofertando una nueva manera de realizar distintos procesamientos dentro de una tarjeta de control. Esta tarjeta realiza una sinergia con ayuda de sus múltiples conexiones entre distintos procesadores, es decir, obtener un resultado en conjunto, mejor al que se obtendría con cualquiera de los procesadores por separado. En el siguiente capítulo se enlistarán las características más importantes de los sistemas híbridos, como son la flexibilidad, reprogramabilidad y el alto desempeño en su procesamiento (Pinto, 2006); así como su descripción y respaldo por trabajos publicados en revistas científicas.

Dentro de la plataforma propuesta por este documento existe un sustento teórico que permite la integración de varias tecnologías, además de innovar con la inclusión de algoritmos

adaptivos para validar su funcionamiento y rendimiento. Este trabajo pretende además formalizar la investigación acerca de tarjetas híbridas de desarrollo, el fomentar un interés por la investigación y dar a conocer el mundo embebido en un microchip.

I.2 Planteamiento del problema

Existen innumerables procesos donde se necesita analizar morfológicamente una pieza específica, y los diseñadores a partir de esta premisa deben considerar distintos factores como el medio que la rodea, los recursos con que se cuenta, entre otros. En ese momento surge la interrogativa, ¿cómo hacerlo?

En nuestro caso nos enfocaremos en solucionar el problema para piezas pequeñas con forma cúbica como la exhibida en la **Figura I.5** que será el modelo procesado para realizar las pruebas. Dentro de las soluciones provistas por el brazo manipulador será el poder transportar las piezas por medio de un actuador, además de tener la capacidad de visualizar todas sus superficies por medio de movimientos de acomodo de la pieza y para culminar con la toma de fotografías. El sistema ROSCOM (Robotic System Control and Manufacturing) abarca todas estas actividades y se implementó en un brazo robótico tipo PUMA con un montacargas en la punta, capaz de realizar los movimientos para levantar una base de aluminio que simula una tarima. Para lograr un perfecto control del brazo requiere mucha precisión y grandes cantidades de procesamiento de datos que los sistemas actuales no sustentan (Hasan, 2006). Si se quiere realizar estas dos actividades de manera eficiente se debe de tener un controlador capaz de hacer un veloz procesamiento y una rápida respuesta para mandar los resultados obtenidos a su destino.

Según la definición del “Robot Institute of America”, un robot manipulador PUMA puede ser empleado para mover materiales, piezas, herramientas o dispositivos especiales, mediante movimientos, programados para la ejecución de diferentes tareas, como los la pintura, soldadura, ensamble, corte, etc. (Ollero, 2001).

Figura I.5 Pieza para realizar las pruebas de funcionalidad de robot PUMA

I.3 Hipótesis y objetivos

Los objetivos a continuación se derivan de una serie de factores económicos, científicos y personales, es por esto que se dividen en generales y específicos. Como objetivo general entenderemos las consideraciones hechas para obtener un sistema que se desenvuelva de manera efectiva en un todo, llamado robot poliarticulado, desglosando así las funciones que debe desempeñar la tarjeta híbrida como un objeto particular dentro del sistema. Los objetivos particulares conciernen a las metas que se esperan obtener al desarrollar un sistema que amplíe el panorama de las tarjetas controladoras, así dicho componente debe representar una mejora de desempeño en el tratamiento de las variables en juego.

I.3.1 Hipótesis general

Es posible realizar una sistema de control para robots tipo PUMA que permita la flexibilidad de programar fácilmente controladores, cambiar tipos de motores y agregar articulaciones, creando una tarjeta híbrida que conste de un núcleo (FPGA) facilitador de información hacia diferentes procesadores intercambiables y reprogramables (DSP).

I.3.2 Objetivo general

Realizar un sistema híbrido de cómputo por medio de la conexión de un FPGA con un DSP utilizando el módulo McBSP para brindar control a un robot poliarticulado, anexando un sistema adaptativo para medición velocidad con alta precisión en encoders.

I.3.3 Objetivos específicos

Una meta de gran importancia que no se menciona explícitamente, es el comprobar el uso de tarjetas híbridas como interfaces de datos y no sólo como tarjetas controladoras, de ahí la gran importancia a la hora de elegir el protocolo de comunicación. Por otra parte algunas de las características deseadas en el sistema se mencionan a continuación:

- Contener una interfaz McBSP entre el FPGA Cyclone II y el DSP TMS320F2812.
- Validar el funcionamiento del sistema híbrido a través de la implementación de un algoritmo adaptivo para la medición de velocidad en el FPGA.
- Programar un control PID en el DSP.
- Calcular velocidades altas a través del uso de la frecuencia medida en los canales del encoder.
- Calcular velocidades bajas a través de los ciclos hechos por los encoders.

CAPÍTULO 2:

II REVISIÓN DE LITERATURA

II.1 Antecedentes

En el epígrafe en desarrollo se abordará el tema de sistemas híbridos de procesamiento, teniendo en cuenta los antecedentes y consecuentes que existen alrededor de esta filosofía de desarrollo.

II.1.1 Nacimiento de los sistemas híbridos de control

En 1967, Takeo et al. mencionan que desde ese año las tendencias de la tecnología requerían un control eficiente y con plataformas precisas basadas en subsistemas software-hardware conectados, sin embargo, los algoritmos implementados en estas plataformas son difíciles de probar y verificar. Por otra parte, se establece que en el diseño es necesario proveer información de los valores internos en registros y memorias de ambos; software y hardware durante la ejecución del sistema completo. La arquitectura final del diseño dirigido y estas capacidades de depurado, dependen fuertemente de cómo está conectado y registrado el sistema híbrido. En este artículo se presentan diferentes estrategias de arquitectura que han sido adoptadas por plataformas híbridas de hardware. Además se propone una estrategia de arquitectura diferente que debe ser adoptada por una plataforma híbrida de hardware.

Desde ese tiempo los dispositivos programables en campo han estado sustituyendo a los circuitos de costumbre. El diseño de los ASICs (Application-Specific Integrated Circuit) ha sido viable históricamente por el bajo costo de los sistemas, sin embargo el número de muestras usadas en un año puede ser excesivo (típicamente 100 000 unidades, según Aguirre, 2004). Hoy en día podemos encontrar en el mercado de la electrónica muchos FPGAs de bajo costo (por ejemplo el EP1C3T100C8N de Altera en 10.7 USD), mostrando una alternativa igual de económica pero reprogramable para disminuir el número de unidades adquiridas.

Aguirre ofrece como una opción el implantar tareas de hardware en un FPGA y conectar con un DSP que desarrolle tareas de software. Estos dispositivos han sido recientemente introducidos en el mercado bajo el nombre de sistemas programables en un solo chip (PSoCs). Por otro lado se establece que la codepuración de un sistema híbrido hardware-software, consiste en la solución de dos problemas clave; la **observabilidad** y la **controlabilidad** durante el tiempo de ejecución. Los beneficios de los esquemas de depuración dependen de qué tan bien estén implementadas estas dos tareas.

La presencia de la memoria conjunta en el dispositivo empleado implica que el desempeño total del sistema será reducido en comparación con la implementación del circuito original (Aguirre, 2004).

Según Aguirre las formas básicas de interfaz son las siguientes:

- Embebido en modelo de memoria.
- Embebido en modelo de enrutamiento.
- Modelo de mapeo de dispositivo.
- Modelo de memoria caché.

Además desde 1963 Hermann Schmidt ya tenía una idea clara de los alcances y propiedades de los sistemas híbridos, en su artículo "An operacional hybrid computing system provides analog-type computation with digital elements", establece:

"En general, los elementos de procesamiento híbrido son insensibles a la ambientación, las tolerancias del componente y los cambios de alimentación, tienen tamaño pequeño, bajo peso y consumo mínimo de potencia y pueden ser instrumentados con 100 por ciento de los circuitos integrados", Hermann Schmidt (1963).

Así, las técnicas híbridas propuestas no tienen una limitante de precisión básica. La justificación más importante en el uso de técnicas híbridas es que hay muchas aplicaciones que requieren una precisión superior a la analógica y no puede ser económicamente resuelta por las técnicas actuales, por ejemplo el procesamiento de señales y el empleo de punto flotante. Una segunda justificación se encuentra en los circuitos integrados, el impacto de estos microcircuitos

en la industria será mejor apreciado cuando los precios futuros de estos dispositivos estén comparados con los de los ASICs.

En este artículo se describe una técnica de procesamiento operacional que emplea cinco elementos básicos de procesamiento. La operación de cada elemento en el procesamiento híbrido está divido en dos modos discretos, a lo largo del tiempo en el periodo. En las operaciones de las señales de control cambia la sincronización entre dos modos de operación con estos elementos de procesamiento.

Existen muchas tareas de procesamiento donde los sistemas variables de entrada están en forma análoga, y por lo tanto se emplean sistemas análogos para su procesamiento. Cuando el número de entradas y salidas es demasiado grande, el costo, el tamaño y el peso pueden exceder la capacidad de procesamiento. En este caso particular se propuso un sistema híbrido ofreciendo más ventajas que un sistema convencional, ya que cada modulador de ancho de pulso es requerido para un convertidor digital en una conversión. Y porque muchos sistemas de control análogos y circuitos de visión pueden aceptar una señal de ancho de pulso como entrada.

II.1.2 Actualidad

"En años recientes se ha visto una disminución en la tasa de aumento de velocidad para relojes de procesadores de uso general, sin embargo la demanda de mejorar el desempeño computacional ha continuado sin cesar. Un nuevo enfoque para satisfacer esta necesidad es el uso de sistemas híbridos. Un **sistema híbrido** se considera a aquel que emplea más de un tipo de motor de cálculo para llevar a cabo sus tareas", Chamberlain (2008). Dicho aumento en la tasa de velocidad puede ser provocada por dos características en los sistemas: la primera y más natural es el aumento en la velocidad del CPU, mientras que existen otras técnicas embebidas en los circuitos integrados para provocar un paralelismo en el procesamiento de las instrucciones, como puede ser el *Pipeline* o un determinado número de Multiplicadores Embebidos.

Chamberlain (2008) en su publicación *"Visions for application development on hybrid computing systems"*, define sistema híbrido de cálculo como un número de componentes de cálculo heterogéneos conectados unos entre sí para crear un recurso de cálculo más robusto, por lo general (pero no necesariamente) empaquetados en un solo recinto.

Este tipo de tecnología es frecuentemente usada en aplicaciones de Sistemas embebidos de cálculo con alto desempeño (HPEC, por sus siglas en inglés). De forma similar y para ampliar el conocimiento acerca de sistemas híbridos, también podemos encontrar clusters que son grandes colecciones de nodos híbridos en red, sin embargo este tipo de sistemas no se desarrollarán en este trabajo.

Otro enfoque para utilizar sistemas híbridos de cálculo es reducir el tamaño del cluster requerido para realizar alguna tarea (Pereira, 2006), así también se obtienen beneficios al realizar el mantenimiento de un sistema más pequeño.

Proshanta et al. (2007), proponen que librerías portátiles de núcleos de hardware altamente optimizados pueden reducir significativamente el tiempo de diseño de aplicaciones con lógica programable. Cuando aparecieron los primeros sistemas con lógica programable en la década de los 80s, utilizaban básicamente coprocesadores con tarjetas FPGA conectadas a una computadora convencional. Los recursos lógicos del FPGA son usados para crear coprocesadores que aceleran una aplicación dada, obteniendo significativos aumentos de velocidad por explotación de hardware de gran manera.

Existe la creencia de que los DSPs pueden desaparecer a partir del uso de Soft-DSP y Hard-DSP, es decir, la descripción en hardware de un DSP dentro de un PFGA y la implementación de la estructura de un DSP dentro de un FPGA (Plumlee, 2008). La principal diferencia radica en que la descripción en hardware o un programa que simule un DSP dentro del FPGA, tendrá menor rendimiento que un DSP real, debido a que el arreglo elegido por la síntesis del programador no es la más eficiente (Shuvra, 2010). Como trabajo futuro dentro de este proyecto se podría plantear la implementación de la estructura híbrida, pero emulando un DSP dentro del FPGA y cuantizar los resultados en ámbitos de eficiencia.

El concepto de procesamiento de lógica programable de alto desempeño (HPRC por sus siglas en inglés) aparece en la década de los 00s (Buell, 1996), cuando la idea del procesamiento reconfigurable fue aplicada primero a los Sistemas de procesamiento paralelo de alto desempeño (HPC, por sus siglas en inglés). Los HPRCs son máquinas de procesamiento paralelo que cuentan con múltiples procesadores y múltiples FPGAs, así ellas combinan hardware en paralelo de precisión fina con sistemas paralelos HPC. Los beneficios que provee el procesamiento

reconfigurable en términos de aceleración de hardware son amplificados con la escalabilidad ofertada por los sistemas HPC. De hecho, el incremento de la velocidad ha sido reportado que llega hasta un tercer o cuarto orden con mejoras en costos y área (El-Ghazawi, 2007 y 2008). Sin embargo, al día de hoy aún los sistemas se asemejan mucho a los antiguos, ya que para cada aplicación se selecciona un hardware acelerador. Por otra parte, esta metodología no toma ventaja sobre la gran cantidad de superposición entre las diferentes aplicaciones, por ejemplo, muchas aplicaciones de procesamiento de imágenes utilizan el mismo núcleo y algoritmos para encriptar llaves públicas, usando las mismas operaciones modulares. Esta falta de buena estrategia de diseño al reutilizar los algoritmos hace que una parte significativa del tiempo de desarrollo sea desperdiciado en la implementación de hardware similar al de proyectos más antiguos (Proshanta et al., 2008).

Proshanta (2008) propone el uso de librerías portátiles que optimicen altamente el núcleo de hardware para el direccionamiento. El beneficio de estas librerías ha sido demostrado a nivel de software por las mismas propiedades y la disponibilidad de recursos abiertos a la comunidad de HPC. Este algoritmo propuesto por el autor, podría representar la estructura medular para el manejo de los distintos protocolos de comunicación en el núcleo de la tarjeta de control (el FPGA).

"Las arquitecturas basadas en lógica programable están ganando la atención de los diseñadores, por los beneficios que ofrece en flexibilidad, reprogramabilidad y alto desempeño de procesamiento. La combinación de GPPs con FPGAs provee valiosas características que se han vuelto indispensables en los sistemas modernos", Marcelo Götz (2008).

Como se puede observar en el párrafo anterior, los sistemas híbridos han ganado gran terreno en el ámbito de diseño, esto gracias a los resultados obtenidos dentro del procesamiento de datos y la flexibilidad que ofrecen. Götz (2008), también menciona cómo los sistemas híbridos permiten la existencia de tareas de hardware, éstas pueden ser debidamente administradas por un sistema operativo y así permitir la coexistencia de varias tareas al mismo tiempo. El tiempo de ejecución de estas tareas en las arquitecturas de lógica programable parece ser una promesa para dar soporte a muchos sistemas, especialmente combinando Sistemas on-Chip (SoC) y Hardware reconfigurable (RH). La gran cantidad de componentes que podemos comunicar abre la posibilidad de construir sistemas de lógica reconfigurable dinámicos, donde

no sólo software, sino también el procesamiento de hardware puede ser cambiado en forma de ejecución. Por otra parte, algunas FPGAs son posibles de reconfigurar sólo en algunas partes del dispositivo, mientras que el resto de las partes permanecen en ejecución. Este tipo de aspectos permiten el concepto de *ablandamiento de hardware* (Upegui, 2008), así el procesamiento de hardware puede ser fácilmente cambiado por la carga de diferentes configuraciones en el FPGA. Sin embargo, a pesar de la flexibilidad obtenida también es necesario que los sistemas evolucionen al igual que todos los dispositivos actuales, es decir a la abstracción para un uso más fácil. De igual manera esta característica puede ser provista por un sistema operativo que redireccione todos los hilos entre hardware y software. Como proponen Mignolet (2002, 2003) y Pellizzoni (2006), se puede lograr de forma dinámica adelantarse o reanudar las tareas en ejecución, independientemente de sus entornos de ejecución, o también a través de límites de hardware/software.

Galanis en el año de 2007, publicó un artículo que presenta una macro de software que implementa una metodología formalizada para particionar aplicaciones de procesamiento digital de señales entre diferentes bloques de RH. Una arquitectura híbrida genérica de lógica programable es considerada y la metodología es aplicada a una gran variedad de sistemas híbridos de lógica reconfigurable. Aunque el macro es paramétrico con respecto al mapeo de procedimientos para las grandes y pequeñas unidades reconfigurables, el artículo provee un algoritmo de mapeo para este tipo de hardware.

El hardware de lógica reconfigurable tiene la capacidad de combinar el desempeño de ASICs y la flexibilidad ofertada por los microprocesadores (Hartenstein, 2001). Adicionalmente los sistemas reconfigurables de varias partes (granularity) ofertan ventajas en términos de rendimiento (Kastner, 2002; Wan, 2001; Rauwerda, 2004), disipación de potencia y gran flexibilidad. De esta manera se podría implementar eficientemente un procesamiento digital de señales.

El hardware de lógica programable de muchas partes, típicamente cumple un rango de cuatro a cinco bits y así, es usualmente programado en los FPGAs. Para este tipo de operaciones de muchas partes (Fine-grain), esta arquitectura es adecuada. Tal como los Bloques de lógica programable dividido en partes (CLBs) de un FPGA son típicamente de cuatro o cinco bits (Galanis, 2007).

La referencia plantea un macro automatizado que realice una metodología para aplicaciones particionadas en una mezcla de varios bloques de hardware de lógica programable. La meta es que la metodología sea para mejorar el rendimiento de sistemas para satisfacer la demanda en el tiempo, siendo esta última parametrizada respecto a un hardware de lógica programable.

Los DSPs además obedecen a ciertas fluctuaciones del sistema, una de ellas es la mala sincronía. Shuvra (2000) en su obra "Resynchronization for multiprocessor DSP systems", documenta la realización de una técnica llamada resincronía para reducir sincronización de sobrecarga en DSPs implementados en multiprocesadores. La técnica aplica una colección arbitraria de procesadores programables, configurables o dedicados, tales como combinaciones de DSPs, ASICs y subsistemas de FPGAs. Esto particularmente es bien recibido en sistemas de DSPs que tiendan a ser aplicados en multiprocesadores de un solo chip heterogéneo (SoCs).

La meta de la resincronía, es introducir nuevas sincronizaciones para que no redunden con respecto al nuevo número de señales añadidas. Un asunto crítico en el diseño de multiprocesadores embebidos, es administrar la comunicación y la sincronización entre los elementos de procesamiento heterogéneo. Shuvra et al. (2000), se enfocan en el problema de minimizar los gastos invertidos en la comunicación y sincronización. En el escrito se proponen algoritmos que automaticen el proceso de diseño de los puntos de sincronización en el sistema de multiprocesado.

En conclusión se diseñó una técnica llamada resincronización para reducir el rango en que las operaciones sincronizadas deben ser realizadas en un sistema multiprocesador de memoria compartida. La sincronización está basada en el concepto que puede haber redundancia en las funciones de sincronización en la implementación de un multiprocesador dado. La implementación del modelo implica una estrategia de auto-programación. Cada procesador ejecuta las tareas asignadas para él, conforme especifique el compilador. Para asegurarse de que la serialización impuesta por la resincronización no minimice el rendimiento, la nueva resincronía será restringida a trabajar fuera de todos los ciclos.

II.2 Consecuentes

"En el futuro los sistemas de procesamiento de alto desempeño pueden consistir en múltiples procesadores de varios núcleos y coprocesadores de lógica reconfigurable", menciona Pranav (2007) en su artículo "Simulation of hybrid computer architectures: simulators, methodologies and recommendations", dentro de él establece que las dos grandes tendencias evidentes en la industria del procesamiento son: primero, las limitaciones físicas han llevado a los principales fabricantes de microprocesadores a integrar múltiples núcleos de un solo chip; y segundo, los nuevos fabricantes de computadoras, así como los fabricantes de dispositivos programables están realizando productos para aplicaciones aceleradas.

De estas dos características anteriores surge el concepto de máquinas híbridas de procesamiento. Estas máquinas ofertan gran desempeño en el procesamiento más allá de las limitaciones de las máquinas Von Neumann.

Como establece Yi (2006), la metodología de simulación debe contener tres características principales:

1. *Eficiencia*. Es decir, que la metodología de simulación debe permitir utilizar si no completamente todas las capacidades del host, mínimo de manera muy satisfactoria.

2. *Elegancia*. La metodología de simulación elegida debe ser fácilmente entendible y modificable.

3. *Determinística y reproducible*. La simulación debe ser capaz de producir resultados idénticos con una misma condición inicial.

De manera similar el diseño de las decisiones asociadas con las técnicas de simulación deben tener consecuencias directas sobre la precisión y validación de la simulación.

Para finalizar se hacen dos recomendaciones:

- Usar técnicas de simulación paralela para los host de simulación presente.
- Abrir la arquitectura del FPGA junto con los recursos en las herramientas de diseño.

CAPÍTULO 3:

III METODOLOGÍA

III.1 Marco teórico

Como en cualquier proyecto tecnológico de actualidad, este trabajo se encuentra integrado por una parte de hardware y otra de software. El hardware utilizado está conformado por un Dispositivo Lógico Programable y un Procesador Digital de Señales que se interconectarán por medio de un protocolo de comunicación. El PLD será el administrador de la información delegando actividades tales como el análisis y adquisición de los datos (Gao, 2009).

Ambos circuitos integrados sirven para implementar algoritmos que sean capaces de realizar tareas sin importar su complejidad. Sin embargo, la mayor complejidad de un sistema híbrido radica en el dominar la codificación de cada uno de sus elementos, es decir, ser un especialista en el lenguaje de programación de los dispositivos que componen al sistema. En nuestro caso, el PLD se programa con VHDL (Very-High-Speed-Integrated-Circuit Hardware Description Language) y el DSP trabaja con una variante del lenguaje C. Consecuentemente, para el diseño se deben tomar en cuenta las características de cada uno de los procesadores para delegar las diferentes actividades.

La principal diferencia es en cuanto a los niveles de programación, ya que un algoritmo puede tornarse muy complicado de programar en VHDL a comparación de desarrollarlo en lenguaje C, es por esto que se decidió que el algoritmo de control es más conveniente programarlo en el DSP (Guha, 2008). Así mismo, un algoritmo en el FPGA se presenta con mayor velocidad de procesamiento, sin embargo, se emplearía mucho tiempo programando el algoritmo de control en lugar de comprobar la efectividad de un sistema híbrido, que es el objetivo de este trabajo.

Existen otras alternativas como los compiladores C a VHDL (ver, www.c-to-verilog.com), estos son muy útiles para diseños grandes o para la ejecución de código que podría cambiar en el futuro. El diseño de una aplicación de gran tamaño en su totalidad en HDL puede ser muy difícil y como se mencionó anteriormente requiere mucho tiempo, la abstracción de un

lenguaje de alto nivel para una aplicación tan grande, a menudo se reduce el tiempo desarrollo total. Además, una aplicación en código HDL es casi seguro que será más difícil de modificar de un código en un lenguaje de alto nivel. Si el diseñador tiene que añadir nuevas funcionalidades a la aplicación, la adición de unas pocas líneas de código C casi siempre será más fácil que la remodelación del código HDL equivalente. Los traductores más empleados para realizar estas tareas con el System-C y Handle-C. La principal desventaja de estos compiladores es obvia, en un lenguaje de bajo nivel existe una visión más amplia del sistema, con una capacidad más amplia de programación, de esta manera si se requiero trasladar un código de alto nivel hacia uno de más bajo nivel, se perderá eficiencia y sobre todo aumentará el tamaño del diseño (Benkrid, 2007).

Las características más importantes al buscar el PLD correcto para nuestra aplicación, son la velocidad de procesamiento, el ahorro monetario y la capacidad para formar interfaces con otros dispositivos. Y el FPGA utilizado en este proyecto contiene un puerto GPIO (General Pupose Input/Output) cumpliendo el último requerimiento. De manera similar el DSP puede ser adquirido ya soldado en una tarjeta con módulos de interfaz, además de extensiones a una tarjeta de trabajo para cada uno de sus pines, a este tipo de herramientas que cuentan ya con un área de trabajo se les denomina *kits de desarrollo*.

Los kits a su vez conforman una interfaz más amigable entre el circuito integrado y el mundo exterior, dicha interfaz puede venir en forma de buses completos que contengan una manera más fácil de conectar puertos completos del microchip; esto representa una gran ayuda a la hora de adquirir señales de pines específicos. Como se puede apreciar en la **Figura III.1**, debido al empaquetado, el espacio entre pines a veces es muy pequeño y esto puede ser una gran barrera para cablear, alimentar o soldar. La gráfica fue obtenida de la hoja de datos de un DSP de Texas Instrument de la familia C2000, en particular el TMS320F2812.

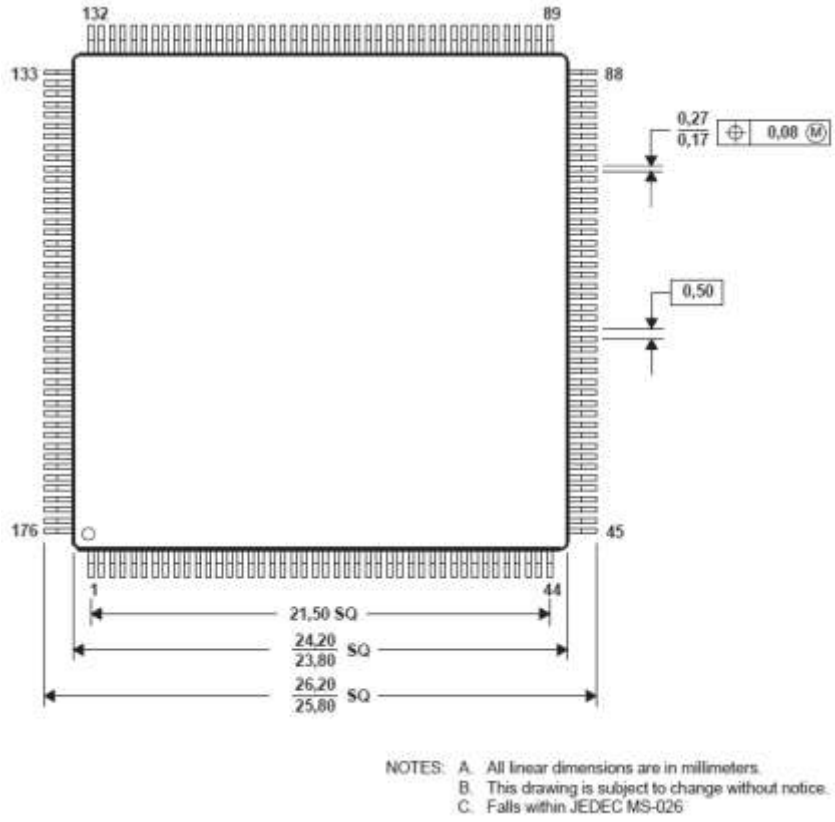


Figura III.1 Dimensiones físicas del DSP TMS320F2812

De esta manera uno de los grandes aportes del kit de desarrollo, es conectar con pistas en un PCB (Printed Circuit Board) cada pin del DSP hacia el exterior y exhibirlos en orificios con marco de cobre para realizar cualquier conexión de manera rápida y fácil.

Respecto al software utilizado, existen muchas compañías que ofrecen productos de alta calidad y con un alto nivel de abstracción, ya que la programación es en su mayoría en bloques, ejemplos de ello son productos como: *Simulink, Wonderware, labVIEW, o labWindows,* donde los dos últimos pertenecen a la empresa *National Instruments* que representa al actual dominante en este tipo de programación. Existen además pruebas elaboradas que verifican la eficacia de los diseños Matlab/Simulink/XSG (System Generador from Xilinx) (Simard, 2009) reduciendo el tiempo de diseño.

El encapsulado del FPGA es muy parecido a la del DSP, como lo podemos apreciar a continuación en la **Figura III.2**., por lo tanto volvemos a encontrarnos con la necesidad de usar

un kit de desarrollo para facilitar su conexión, abstraer nuestro diseño físico y no crear PCBs que pueden incrementar el costo en comparación con los kits ofrecidos por *Terasic* (véase www.terasic.com).

Figura III.2 Encapsulado de FPGA ALTERA EP2C20F484C7N

III.1.1 PMSM 3.1 Control Motor System

Texas Instruments, a manera de facilitar el uso de sus productos brinda soporte y una gran cantidad de literatura. En control de motores, ha realizado un conjunto de librerías para gestionar las señales de cuadratura del encoder y realizar un control PID.

El nombre de este proyecto es DCMOTOR y está orientado hacia la familia C2000 de DSPs. Para obtener en este proyecto en nuestro ordenador se instaló el PMSM 3.1 Control Motor System, el cual organiza en una carpeta todos los archivos de recurso y cabecera necesarios para el control de un motor de CD. Además, cada librería cuenta con un documento que describe cada uno de los registros de su estructura. Así, con la comprensión de cada módulo podemos modificar el proyecto para orientarlo hacia nuestra meta y material disponible. Dentro de este proyecto existen 4 módulos principales:

- BDCPWM
- QEP_NO_INDEX
- PID_REG3
- ESTIMATEDSPEED

El presente conjunto de módulos es el encargado de llevar paso a paso el algoritmo para el control de motores de CD. El módulo QEP_NO_INDEX adquiere las señas provenientes del encoder (los canales QEP_A y QEP_B, **Figura III.3**), además proporciona varias salidas útiles como es la dirección de giro (Direction Qep), un contador de índices y el número de cuentas que se desarrollan (MechTheta en pulsos y OutputTheta en revoluciones).

Tabla III.1 Variables de entradas y de salida en la estructura QEP_NO_INDEX

Estructura de QEP_NO_INDEX_DRV	Descripción
QEP_A	Pulsos de cuadratura para canal A
QEP_B	Pulsos de cuadratura para canal B
MechTheta	Ángulo de la flecha del motor (0-360°)
OutputTheta	Ángulo mecánico del motor (0-360°)
DirectionQep	Dirección de rotación

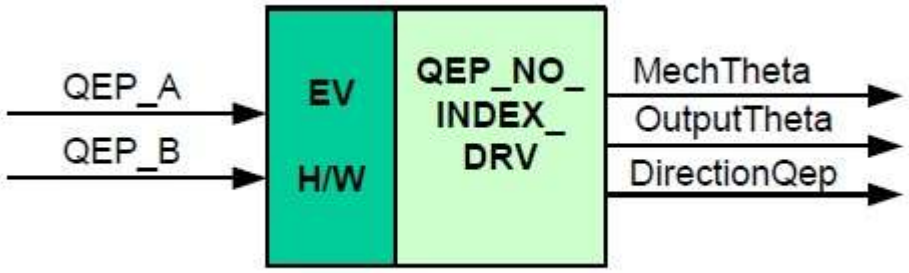


Figura III.3 Bloque de entradas y salidas de QEP_NO_INDEX_DRV

ESTIMATEDSPEED es el encargado de mediar la velocidad del motor basado en una estimación de la posición del rotor cuando la información de la dirección rotacional no está disponible, esto es posible por medio del número de cuentas que le proporciona

QEP_NO_INDEX. El módulo ESTIMATEDSPEED brinda como salida la velocidad en RPM (StimatedSpeedRpm, **Figura III.4**), además de una salida en PU (per-unit).

Tabla III.2 Variables de entradas y de salida en la estructura ESTIMATEDSPEED

Estructura de ESTIMATEDSPEED	Descripción
EstimatedTheta	Ángulo eléctrico (0-360°)
EstimatedSpeed	Velocidad estimada (PU)
EstimatedSpeedRpm	Velocidad estimada (RPM)

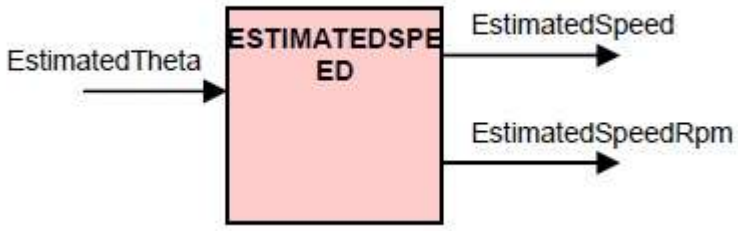


Figura III.4 Bloque de entradas y salidas de ESTIMATEDSPEED

El módulo PID_REG3 se analizará más a fondo en capítulos a continuación, sin embargo se trata de un PID simple, al cual se le puede proporcionar una referencia en velocidad o posición (**Figura III.5**) para obtener una salida controlada. En nuestro caso tenemos dos módulos conectados al Fdb (retroalimentación del PID), QEP_NO_INDEX y ESTIMATEDSPEED para realizar el control de las dos variables, posición y velocidad. Las referencias son adquiridas desde el código en CCS (Code Composer Studio).

Tabla III.3 Variables de entradas y de salida en la estructura PID_REG

Estructura de PID_REG3	Descripción
Ref	Referencia de entrada
Fdb	Retroalimentación
Out	Salida del PID

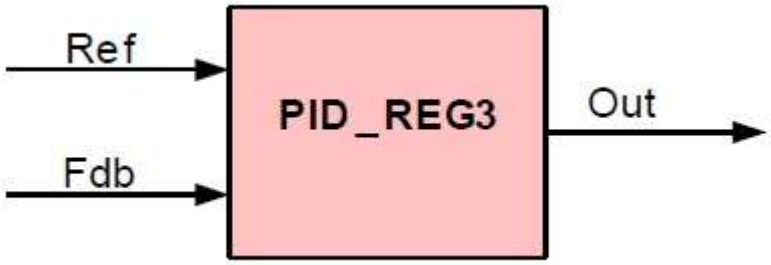


Figura III.5 Bloque de entradas y salidas de PID_REG3

Para finalizar debemos tener un módulo encargado de transmitir la orden del PID_REG3 hacia los motores, para esto se necesita una etapa de potencia administrada por 4 PWM, estas señales son generadas por BDCPWM, el cual le transmite las instrucciones al motor de CD por medio de señales de PWM generadas por pines en el DSP (**Figura III.6**), las variable de entrada para este bloque con la dirección de giro (Rotation), el ancho de pulso (DutyFunc) y la frecuencia del PWM (MfuncPeriod).

Tabla III.4 Variables de entradas y de salida en la estructura BDC_PWM_DRV

Estructura de BDC_PWM_DRV	Descripción
Rotation	0=PWM1 y 4 habilitados; 0=PWM2 y 3 habilitados
DutyFunc	Modulación de periodo PWM
MfuncPeriod	Escalador del periodo
PWM1	Señal de PWM
PWM3	Señal de PWM

Figura III.6 Bloque de entradas y salidas de BDC_PWM_DRV

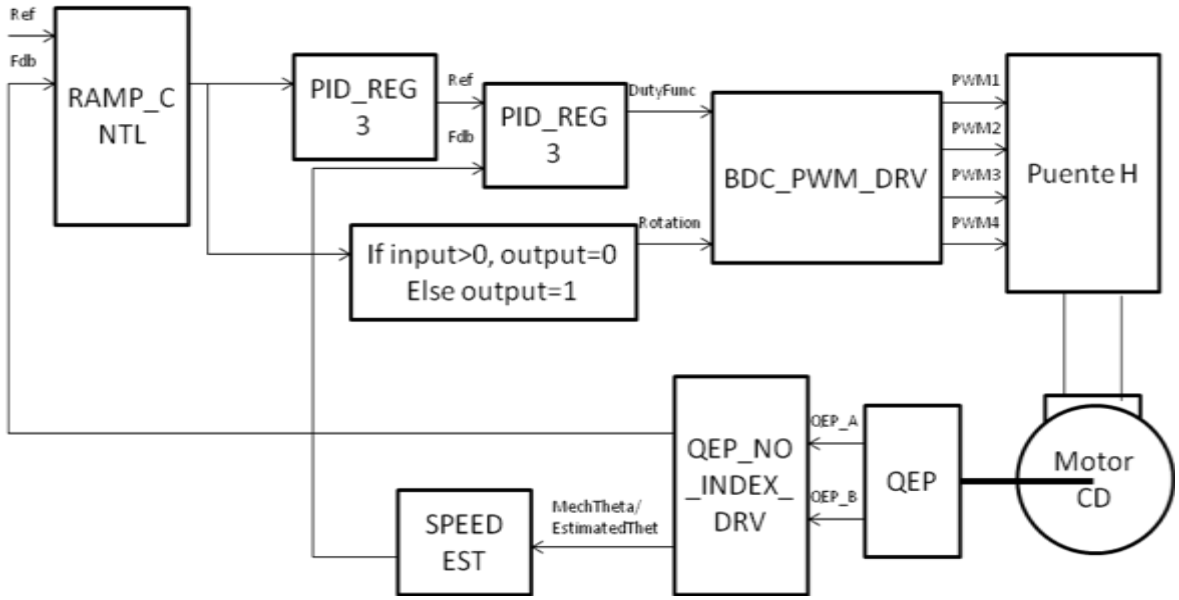


Figura III.7 Diagrama a bloques de programación en TMS320F2812

III.1.2 Teoría de controladores PID (Proporcional Integral Derivativo)

Para el control de motores es necesario un algoritmo capaz de hacer que una salida tienda a una referencia deseada. Y en general, para realizar cualquier algoritmo de control, se deben tener las tres consideraciones siguientes:

1. Determine qué debe hacer el sistema y cómo debe hacerlo (especificaciones de diseño).
2. Determine la configuración de compensador o el controlador relativa a cómo está conectado al proceso controlado.
3. Determine los valores de los parámetros del controlador para alcanzar los objetivos de diseño.

A menudo se emplean especificaciones de diseño para describir qué debe hacer el sistema y cómo hacerlo. Estas especificaciones son únicas para cada aplicación individual y con frecuencia incluyen especificaciones como estabilidad relativa, precisión en estado estable (error), respuesta transitoria, y características de respuesta en frecuencia. En algunas aplicaciones puede haber especificaciones adicionales sobre sensibilidad a variaciones de parámetros.

El diseño de sistemas de control lineales se puede realizar ya sean en el dominio del tiempo o en el dominio de la frecuencia. Por ejemplo, la precisión en estado estable a menudo se especifica con respecto a una entrada escalón unitario, una entrada rampa o a una entrada parábola, y el diseño para cumplir ciertos requisitos es más conveniente realizarlo en el dominio del tiempo. Otras especificaciones como el sobrepaso máximo, tiempo de levantamiento y tiempo de asentamiento, están definidas para una entrada escalón unitario, y por tanto se emplean para diseño en el dominio del tiempo. Se ha aprendido que la estabilidad relativa también se mide en términos del margen de ganancia, margen de fase, y M_r. Éstas son especificaciones típicas del dominio de frecuencia y deben emplearse junto con herramientas como la traza de Bode, la traza polar, la traza de ganancia-fase, y la carta de Nichols.

Se ha mostrado que para el sistema prototipo de segundo orden, existen relaciones analíticas simples entre estas especificaciones, en los dominios del tiempo y de la frecuencia. Sin embargo, para sistemas de orden superior, la correlación entre las especificaciones entre los dominios del tiempo y la frecuencia son difíciles de establecer. En resumen, el análisis y diseño de sistemas de control es más un ejercicio de selección entre varios métodos alternativos que resuelven un mismo problema. Por tanto la selección de si el diseño se debe realizar en el domino del tiempo o de la frecuencia depende de la preferencia del diseñador. Sin embargo, se debe señalar que, en la mayoría de los casos, las especificaciones en el dominio del tiempo tales como sobrepaso máximo, tiempo de levantamiento y tiempo de asentamiento se emplean normalmente como la medida final del desempeño del sistema.

La tecnología actual en sistemas computacionales facilita la implementación de algoritmos de control en el dominio del tiempo. Además de día con día reducir su tiempo de procesamiento gracias a las actualizaciones en hardware. Esto disminuye considerablemente la ventaja histórica del diseño en el dominio de la frecuencia, el cual está basado en la conveniencia de realizar el diseño gráfico en forma manual (que generalmente es difícil, excepto para el diseñador experimentado).

Uno de los controladores más sencillos y más utilizados en la actualidad es el PID, el cual aplica una señal al proceso que una combinación proporcional, integral y derivativa de la señal de actuación. Debido a que estos componentes de la señal se pueden realizar y visualizar

con facilidad en el dominio del tiempo, los controladores PID se diseñan comúnmente empleando métodos en el dominio del tiempo.

Los tipos de controladores disponibles para el diseño de sistemas de control están limitados sólo por la imaginación. Los ingenieros prácticos normalmente establecen que uno escoge el controlador más simple que cumpla con todas las especificaciones de diseño. En la mayoría de los casos, mientras más complejo sea un controlador, es más costoso, menos confiable, y más difícil de diseñar. El escoger un controlador determinado para una aplicación específica se basa a menudo en la experiencia del diseñador, y algunas veces en la intuición, e involucra inevitablemente tanto arte como ciencia (Kuo, 1996).

Una vez elegido el controlador, la siguiente tarea es determinar los valores de los parámetros del controlador. Estos parámetros don típicamente coeficientes de una o más funciones de transferencia que componen al controlador. El enfoque de diseño básico es emplear las herramientas de análisis discutidas en los capítulos anteriores para determinar cómo los valores de los parámetros individuales afectan las especificaciones de diseño, y finalmente, el desempeño del sistema. Con base a esta información, se determinan los parámetros del controlador para que se cumplan todas las especificaciones de diseño. Mientras algunas veces este proceso es directo, más frecuentemente no involucra muchas iteraciones de diseño, ya que normalmente los parámetros del controlador interactúan unos con otros y afectan las especificaciones de diseño en formas conflictivas. Por ejemplo, el valor de un parámetro en particular se puede seleccionar para el sobrepaso máximo sea satisfecho, pero en el proceso de variar el valor de otro parámetro, en un intento por cumplir el requisito de tiempo de asentamiento, la especificación de sobrepaso máximo puede ya no cumplirse. Es claro que mientras más especificaciones de diseño y más parámetros haya, el proceso de diseño se vuelve más complicado.

Para un diseño más intuitivo, Kuo (1996) en su libro “Sistemas de control automático”, dicta las siguientes consideraciones para modelos, indistintamente del tiempo y la frecuencia.

1. Los polos complejos conjugados de la función de transferencia en lazo cerrado producen una respuesta al escalón unitario que es subamortiguada. Si todos los polos son reales, la respuesta al escalón unitario es sobreamortiguada. Sin embargo, los ceros

de la función de transferencia en lazo cerrado pueden causar un sobrepaso aun si el sistema es sobreamortiguado.

2. La respuesta de un sistema está dominada por aquellos polos más cercanos al origen de plano s. Los transitorios debidos a aquellos polos a la izquierda decaen más rápido.

3. Mientras más alejados a la izquierda en el plano s estén los polos dominantes del sistema, el sistema responderá más rápido y mayor será el ancho de banda.

4. Mientras más alejados a la izquierda del plano s estén los polos dominantes del sistema, más caro será y más grandes serán sus señales internas. Aunque esto se puede justificar en forma analítica, es obvio que golpear más fuerte un clavo con un martillo hará que el clavo entre más rápido, pero requiere más energía por golpe. En forma similar, un carro de carreras puede acelerar más rápido, pero emplea más combustible que un carro promedio.

5. Cuando un polo y un cero de una función de transferencia de un sistema se cancelan uno con el otro, la porción de la respuesta del sistema asociada con el polo tendrá una magnitud más pequeña.

6. Las especificaciones en los dominios del tiempo y de la frecuencia están asociadas vagamente. El tiempo de levantamiento y el ancho de banda son inversamente proporcionales. El margen de fase, el margen de ganancia, Mr, y el amortiguamiento son inversamente proporcionales.

Los modelos de control más simples como compensadores, consisten únicamente en un amplificador simple con una ganancia K. Este tipo de acción de control se conoce formalmente como control proporcional, ya que la señal de control a la salida del controlador está relacionada con la entrada del controlador mediante una constante proporcional. En forma intuitiva, se debe ser capaz de emplear la derivada o la integral de la señal de entrada, además de la operación proporcional. En consecuencia, se puede considerar un controlador en tiempo continuo más general como aquel que contiene componentes tales como sumadores, amplificadores, atenuadores, diferenciadores e integradores.

En cuanto a la parte derivativa (de(t)/dt), representa la pendiente de e(t), para ejemplificarla mencionaremos el uso de un control PD o control anticipativo. Esto es, al conocer la pendiente el controlador puede anticipar la dirección del error y emplearla para controlar mejor el proceso. Normalmente, en sistemas lineales si la pendiente de e(t) o y(t) debida a la entrada escalón es grande, subsecuentemente ocurrirá un sobrepaso alto. El control derivativo mide la pendiente instantánea de e(t), predice el sobrepaso grande adelante en el tiempo, y hace un esfuerzo correctivo antes de que el sobrepaso excesivo ocurra. En forma intuitiva, el control derivativo afecta el error en estado estable de un sistema sólo si el error en estado estable varía con el tiempo. Si el error en estado estable de un sistema es constante con respecto al tiempo, la derivada con respecto al tiempo de este error es cero, y la porción derivativa del controlador no provee ninguna entrada al proceso. Pero si el error en estado estable se incrementa con el tiempo, se genera otra vez un par en proporción a de(t)/dt, lo cual reduce la magnitud del error. Las características obtenidas por un PD son las siguientes:

1. Mejora el amortiguamiento y reduce el sobrepaso máximo.
2. Reduce el tiempo de levantamiento y el tiempo de asentamiento.
3. Incrementa el BW.
4. Mejora el margen de ganancia, el margen de fase y M_r.
5. Puede acentuar el ruido en altas frecuencias.
6. No es efectivo para sistemas ligeramente amortiguados o inicialmente inestables.
7. Puede requerir un capacitor muy grande en la implementación del circuito.

Por último, la parte integral del controlador PID produce una señal que es proporcional a la integral con respecto al tiempo de la entrada del controlador. Para ejemplificar su funcionamiento, se hará alusión al comportamiento de un PI; como primer característica se tiene que mejora el error en estado estable a costa de la estabilidad. Sin embargo, si se tiene una buena selección del cero para eliminar uno de los polos, tanto el amortiguamiento como el error en estado estable se pueden mejorar. Ya que el controlador PI es en esencia un filtro pasa-bajas, el sistema compensado tendrá un tiempo de levantamiento más bajo y un tiempo de asentamiento

más largo. Un método factible para diseñar un PI es seleccionar el cero en $s=-K_1/K_p$ relativamente cerca del origen y lejos de los polos más significativos del proceso, y los valores de Kp y K_I deben ser relativamente pequeños.

De acuerdo a la teoría ya conocida acerca del PI, podemos resumir las siguientes características para su selección:

1. Mejora el amortiguamiento y reduce el sobrepaso máximo.
2. Incrementa el tiempo de levantamiento.
3. Disminuye el ancho de banda.
4. Mejora el margen de ganancia, el margen de fase y M_r.
5. Filtra el ruido de alta frecuencia.
6. El problema de seleccionar una combinación adecuada de K_I y K_p para que el capacitor en la implementación del circuito del controlador no sea excesivamente grande, es menos trivial que en el caso del controlador PD.

De las discusiones anteriores se observa que el controlador PD puede añadir amortiguamiento a un sistema, pero no afecta la respuesta en estado estable. El controlador PI puede mejorar la estabilidad relativa y el error en estado estable al mismo tiempo, pero el tiempo de levantamiento se incrementa. Esto conduce a emplear un controlador PID para que se empleen las mejores características de los controladores PI y PD.

Así, de la eficacia para el control de motores y las librerías que ofrece el TMS320F2812 para su construcción, se decidió emplear un PID para validar el proyecto. La teoría de control que maneja la famila de DSPs 28x es la siguiente.

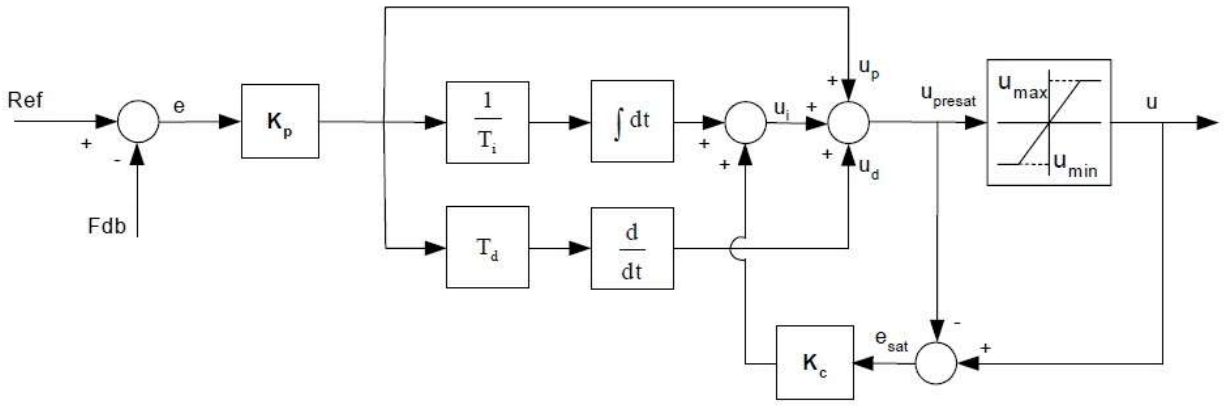


Figura III.8 Estructura del PID usado por PID_REG3

Como se puede observar la única diferencia al PID convencional es el uso de un Anti-WindUp a la salida, esto se utiliza para no tener salidas fuera de cierta proporción, proporción programada dentro de la estructura del PID_REG3.

Obedeciendo al diagrama a bloques anterior, el PID tiene la forma de la **Ecuación III.1**, sin embargo para deducirla, es necesario tomar como antes fue mencionado, el PID como una sumatoria de sus tres partes (Frankin, 1990; ver también la guía "Digital PID Controller with Anti-windup" de Texas Instruments).

$$u_{presat}(t) = u_p(t) + u_i(t) + u_d(t) \tag{III.1}$$

Y a continuación se realiza un desgloce de los componentes.

$$u_p(t) = K_p e(t) \tag{III.2}$$

$$u_i(t) = \frac{K_p}{T_i}\int_0^t e(\zeta)d\zeta + K_c\left(u(t) - u_{presat}(t)\right) \tag{III.3}$$

$$u_d(t) = K_p T_d \frac{d}{dt} e(t) \tag{III.4}$$

Donde:

$u(t)$ es la salida del controlador PID

$u_{presat}(t)$ es la salida antes de la saturación

$e(t)$ es el error entre la referencia y las variables de retroalimentación

K_p es la ganancia proporcional del controlador PID

T_i es la integral del tiempo del controlador PID

T_d es la derivada del tiempo del controlador PID

K_c es la ganancia de corrección integral del controlador PID

Por otra parte es necesario encontrar una salida presaturada, para esto se calcula.

$$u_{presat}(k) = u_p(k) + u_i(k) + u_d(k) \tag{III.5}$$

Y posteriormente hay que encontrar cada una de las ganancias con su respectiva corrección de saturación y hacer $K_i=T/T_i$ y $K_d=T_d/T$, para esto tenemos.

$$u_p(k) = K_p e(k) \tag{III.6}$$

$$u_i(k) = u_i(k-1) + K_p \frac{T}{T_i} e(k) + K_c\left(u(k) - u_{presat}(k)\right) \tag{III.7}$$

$$u_d(k) = K_p \frac{T_d}{T}\left(e(k) - e(k-1)\right) \tag{III.8}$$

III.1.3 Sintonización de un PID

La utilidad de los controladores PID estriba en que se aplican en forma general a la mayoría de los sistemas de control. En el campo de los sistemas de control, los esquemas de control PID básicos y modificados han demostrado su utilidad al aportar un control satisfactorio (Ogata, 2003).

Para describir el proceso de selección de los mejores valores para el controlador se usa el término sintonización (Bolton, 2001). La sintonía de un algoritmo de control consiste en seleccionar valores adecuados para sus parámetros. Por tanto, para el caso del controlador PID se trata de calcular los valores idóneos para sus parámetros (K_p, T_i y T_d) de forma que se asegure que el sistema se comporta siguiendo las especificaciones previamente definidas.

En las primeras aplicaciones de control PID el ajuste se basaba únicamente en la propia experiencia del operario o simplemente se utilizaban los ajustes del fabricante. Existen diferentes métodos para la sintonización, dentro de los más estudiados se encuentra el método de respuesta en frecuencia que se basa en los siguientes principios fundamentales (Osornio, 2003):

1. Se encuentra la función de transferencia en lazo abierto.

$$L(s) = G(s)H(s) \tag{III.9}$$

2. Se remplazan los parámetros s en L(s) por los parámetros jω. La función de transferencia resultante L(jω), es llamada respuesta en frecuencia en lazo abierto.

3. Se encuentra la frecuencia de cruce ω_c del sistema. La frecuencia de cruce es aquella en la que la magnitud L(jω) es uno, como se ilustra en la **Ecuación III.10**.

$$\left|L(jw)\right| = 1 \tag{III.10}$$

La importancia de la frecuencia de cruce es que indica la velocidad de respuesta, un valor alto de ω_c indica respuestas más rápidas. En general, la constante de tiempo de la respuesta, τ se aproxima mediante la **Ecuación III.11**.

$$\tau = \frac{1}{\omega_c} \tag{III.11}$$

4. Se encuentra la fase de L(jω). Dado que L(jω) es una función compleja se encontró su argumento (ϕ) por medio de la **Ecuación III.12** y el margen de fase se obtiene mediante la **Ecuación III.13**.

$$\phi = \arg[L(j\omega)] \tag{III.12}$$

$$\theta = 180 + \phi \tag{III.13}$$

5. El parámetro θ_m tiene la llave de la estabilidad y amortiguamiento del sistema. El sistema es estable cuando el margen de fase θ_m es positivo e inestable cuando θ_m es negativo. Los valores del margen de fase del intervalo de 30 a 45 grados indican una respuesta amortiguada; valores de margen de fase más pequeños indican respuestas de bajo amortiguamiento.

III.1.4 Motor de CD serie

Es un motor cuyo devanado de campo consta relativamente de unas pocas vueltas conectadas en serie con el circuito de armadura. El circuito equivalente de un motor de CD se muestra en la **Figura III.9**. En un motor de CD, la corriente de armadura, la corriente de campo y la corriente del suministro de potencia son iguales. El comportamiento básico de un motor de CD serie se debe al hecho de que el flujo magnético de campo es directamente proporcional a la corriente de armadura, al menos, hasta antes de llegar a la saturación de flujo magnético en el

núcleo. Cuando se incremente la carga del motor, también aumenta el flujo magnético. Las variables que intervienen en el sistema son:

R_{in} Resistencia en serie con inductancia.

L_{in} Embobinado del motor.

T_m Torque desarrollado por la flecha del motor.

J_m Inercia.

$\theta(t)$ Ángulo de la flecha.

I_{ex} Corriente de exitación

$U_{in}(t)$ Voltaje de alimentación.

$U_b(t)$ Voltaje en bornes del motor.

B_m Amortiguamiento presentado por carga.

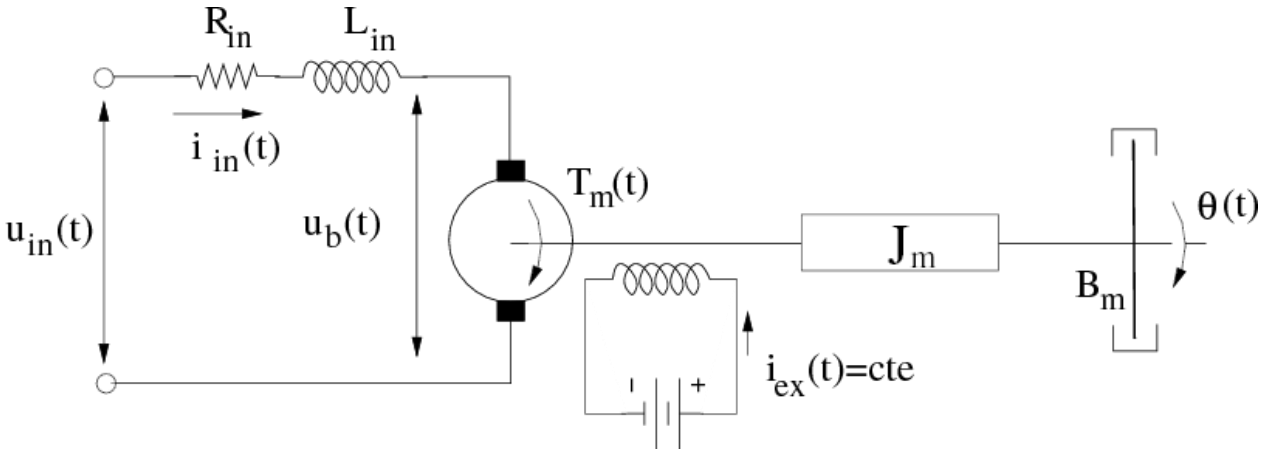


Figura III.9 Circuito más básico para un motor CD

III.2 Metodología

En particular, la fusión que se realizará como se ha visto a lo largo de este informe, será entre un Cyclone II FPGA Starter Board (de Altera y por lo tanto todos los diseños están desarrollados Quartus II 9.0 sp2) y un DSP sZdspTMS32028F2812 de Texas Instruments. El FPGA realizará las funciones de enlace por medio de los módulos de su kit de desarrollo, así que se programarán las máquinas de estado necesarias para el uso de sus diferentes puertos.

El material utilizado será elegido entre los contenidos por la Universidad Autónoma de Querétaro (UAQ) ya que es demasiado costoso adquirir nuevo equipo. De entre ellos, el procesador más completo y adecuado para nuestra aplicación es el TMS320F2812, ya que éste se especializa en el control de motores. Su creador es Texas Instruments, y el laboratorio escolar lo contiene en su presentación didáctica eZdsp Start Kit (**Figura III.10**). Este Kit cuenta con una serie de conexiones directas a los puertos del DSP, además de relojes, convertidores y una serie de elementos que facilita el desarrollo de prácticas muy especializadas.

Entre el FPGA y el DSP se realizará un protocolo de comunicación McBSP (Multi-Channel Buffered Serial Port), éste módulo del DSP provee una interfaz serial directa entre un Procesador Digital de Señales y otro dispositivo en un sistema, para sincronizadamente poder transmitir/recibir datos de 8/12/32-bit en forma serial.

III.2.1 Características del módulo McBSP

El McBSP provee una interfaz entre dos dispositivos, un 28x u otro compatible con mencionado protocolo y tales como VBAP (Vector Based Amplitude Panning), AIC (Advanced Interface Converter) y Combo Codecs. Las principales características que son:

- Comunicación full-duplex.
- Doble-buffer para transmisión y triple-buffer para recepción, así se permite un continuo flujo de datos.
- Relojes independientes y etiquetas para recepción y transmisión.
- 128 canales para transmisión y recepción.

- Modo de selección de multicanal que te habilita para permitir o bloquear transferencias en cada uno de los canales.
- DMA reemplazados con 2 FIFOs de16 niveles y 32 bits.
- Soporte para modo A-bis.
- Interfaz directa a los codecs standard-industry, interfaz análoga a chips (AICs), y otra conectada de manera serial a dispositivos A/D y D/A.
- Soporte para generación externa de señales de reloj y señales frame-synchronization (frame-sync).
- Reloj interno programable y generación de frame.
- Polaridad programable para frame-synchronization y reloj de datos.
- Soporte para dispositivos SPI.
- Soporte para T1/E1.
- Una amplia selección de tamaños de dato: 8, 12, 16, 20, 24 y 32 bits.
- Una opción para transmitir/recibir datos de 8 bits con el LSB o el MSB primero.
- La máxima tasa de transmisión es ½ de la velocidad del CPU, limitada por 50 Mbps.

En la siguiente imagen (**Figura III.11**) se puede apreciar el triple buffer para la recepción y el doble buffer para la transmisión.

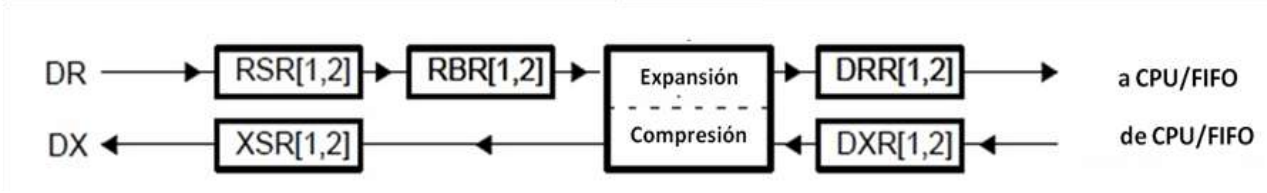


Figura III.10 Triple buffer para la recepción y doble buffer para transmisión en McBSP (SPRU061A, TI)

La Cyclone II recibirá la lectura de los encoders por medio del PROTO_A el cual tiene 68 pines GPIO asociados directamente al FPGA, dispuestos para ser utilizados como entradas o salidas. El FPGA mandará los datos de los canales del encoder hacia el DSP utilizando el mismo puerto, esto, debido a que en el DSP se encontrará programado el controlador PID para el procesamiento de las variables, y así regresar instrucciones con los movimientos adecuados (Kung, 2005).

El trabajo en conjunto de los dispositivos mencionados a lo largo del párrafo anterior puede ser tan substancial como difícil de diseñar, por ende también se transforma en un sistema difícil de depurar a la hora de hacer las pruebas, sin embargo existen propuestas para manipular los errores como la hecha por M. A. Aguirre (2004), en la cual se sostiene que manteniendo una misma fuente de reloj es posible hacer de manera significativa las mediciones en la depuración de un sistema híbrido. El McBSP trabaja enviando un conjunto de palabras antecedidas por un FS (Frame Start) como se muestra en la **Figura III.12**.

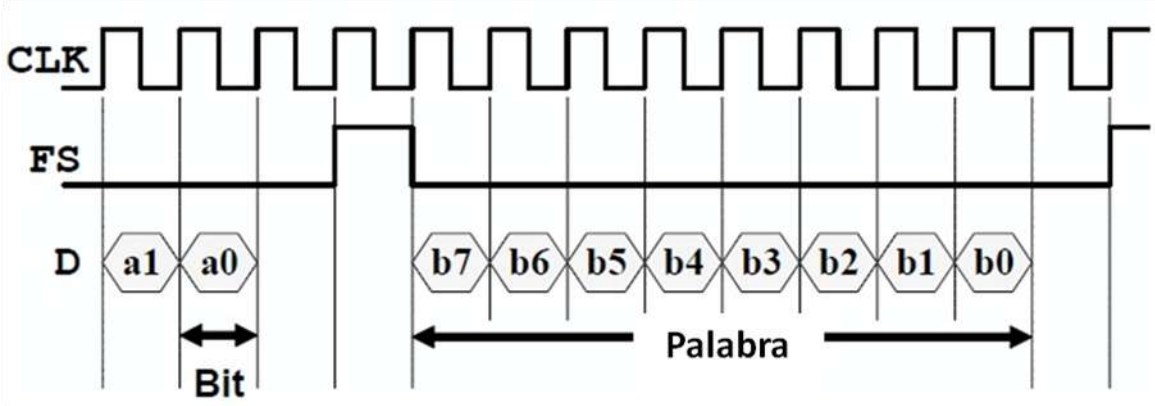


Figura III.11 Número de bits por palabra en el McBSP (SPRU061A, TI)

Como se puede apreciar, existe un número definido de bits contenido en cada palabra (o canal), sin embargo se puede configurar la palabra de acuerdo a las distintas necesidades, con la posibilidad de cambiar el número de bits contenidos por la palabra. Las diferentes configuraciones son 8, 12, 16, 20, 24 y 32 bits.

Yendo de lo particular a lo general, existe otro tipo de dato llamado estructura o frame, éste se encuentra compuesto por palabras (**Figura III.13**), las cuales pueden variar desde 1 hasta 128 dependiendo la configuración elegida.

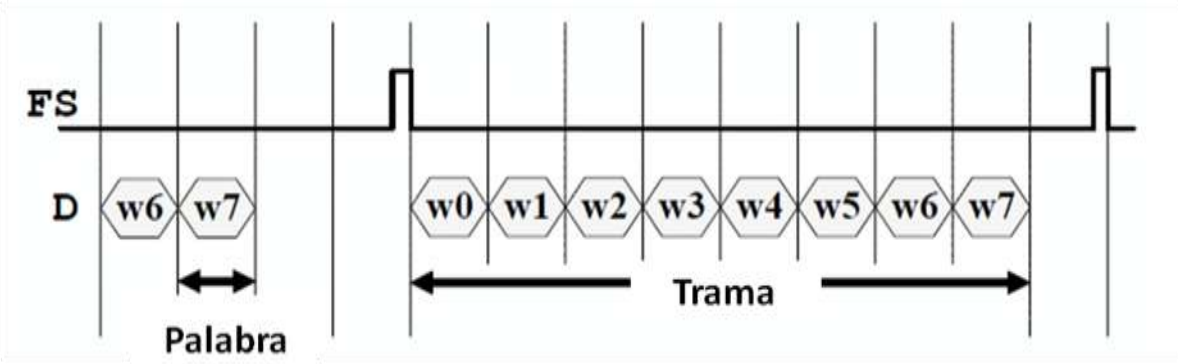


Figura III.12 Palabras enviadas por una Estructura en el McBSP (SPRU061A, TI)

El McBSP recibe su nombre debido a que brinda la oportunidad de elegir entre múltiples canales (o palabras, **Figura III.14**) para ser independientemente seleccionados por el transmisor y el receptor. El protocolo guarda el tiempo de sincronía con todos los canales, pero sólo recibe o transmite si el canal especificado está habilitado. Así, se optimiza la conexión procesamiento/bus.

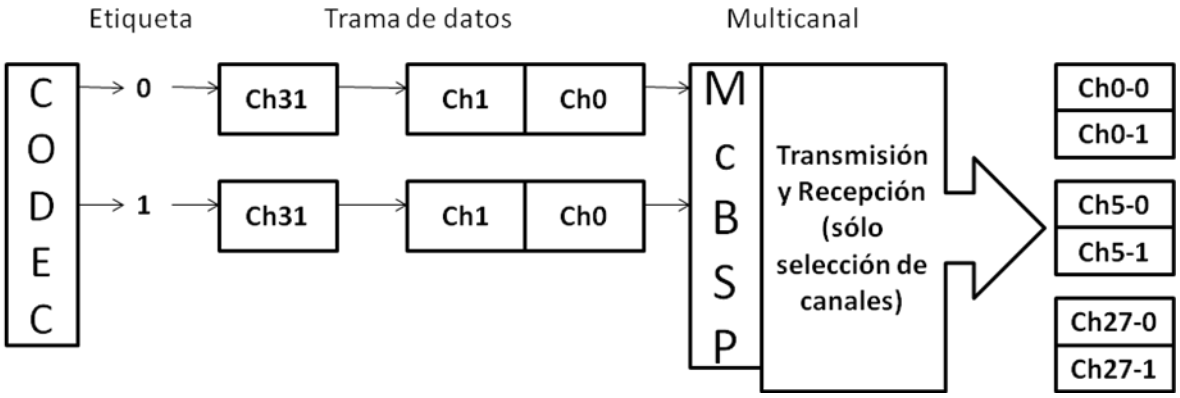


Figura III.13 Selección de canal dentro del McBSP (SPRU061A, TI)

El registro para habilitar el modo multicanal es el MCR (Multi-channel Control Reg) y el correspondiente a la elección de canales es el R/XCER(A-H) (Rec/Xmt Channel Enable Regs). Teniendo arriba de 128 canales para ser habilitados o deshabilitados.

III.2.2 Código en lenguaje C para comunicación McBSP

Como todo código en lenguaje C, es necesario conocer los archivos de cabecera que se emplearán. En particular para este programa es necesario analizar dos: DSP281x_Device.h y DSP281x_Examples.h. Este par de archivos contienen las instrucciones necesarias para realizar la comunicación. El archivo DSP281x_Device.h es una librería madre que llama a otras 16 correspondientes a la familia de DSPs 281x (**Tabla III.5**).

Tabla III.5 16 archivos de cabecera incluidos por DSP281x_Device.h

1. DSP281x_Device.h	2. DSP281x_DevEmu.h	3. DSP281x_Gpio.h	4. DSP281x_Mcbsp.h
5. DSP281x_SysCtrl.h	6. DSP281x_PieCtrl.h	7. DSP281x_Sci.h	8. DSP281x_Spi.h
9. DSP281x_Adc.h	10. DSP281x_CpuTimers.h	11. DSP281x_Xintf.h	12. DSP281x_XIntrupt.h
13. DSP281x_ECan.h	14. DSP281x_Ev.h	15. DSP281x_PieVect.h	16. DSP281x_DefaultIsr.h

Para este código la más importante es la número 4, de la cual, la primera función en sobresalir es *InitMcbspGpio()*, y se encarga de activar los pines F9, F8, F13, F12, F11 y F10, pines necesarios para la función del módulo. Posteriormente se realiza un reset a *FS*, bandera de arranque para la lectura de la trama (como se puede apreciar en la **Figura III.13**).

Se declaran dos variables de envío (sdata1, sdata2) y dos variables de recepción (rdata1, radata2); esto para realizar un loopback digital, es decir un bucle entre las recepciones y los envíos. El código empleado se encuentra dentro de los anexos con el nombre de McBSPLoopback.c. Uno de los aspectos más importantes a notar dentro de este anexo es la

deshabilitación de la interrupciones: IER (Interrupt Enable Register) e IFR (Interrupt Flag Register).

Para tener un flujo correcto de los datos es necesario activar el modo FIFO (First In First Out), el nombre de la función dentro de DSP281x_Mcbsp.h es *mcbsp_fifo_init()*, de esta manera se inicializa específicamente para el McBSP.

Como ya fue explicado anteriormente, el McBSP soporta diversas configuraciones para el número de bits enviados. Así, que en el código se colocó una condición de manera que puede ser modificado el número de bits desde una declaración *#define WORD_SIZE*, la cual se puede tener los valores 8, 16 y 32; específicamente para este código. Otra de las definiciones necesarias para empezar a trabajar es el nivel de FIFO que vamos a emplear, de modo que después de haberlo inicializado, se debe elegir el grado por medio de una declaración *#define FIFO_LEVEL*, teniendo como posibilidades desde 1 hasta 16.

La parte más importante del programa se encuentra dentro del *if* de 32 bits para tamaño de palabra, el cual fue elegido puesto que se necesitan enviar 11 bits para el cociente Ct de la velocidad y 7 para Cp, el número de pulsos. Ahí se realiza un mapeo entre los datos sdata1 y sdata1_point, para posteriormente ser concatenados y formar la palabra completa de 32 bits.

Para poder igualar las frecuencias de envío del FPGA y del DSP, es necesario bajar la frecuencia original del McBSP del DSP que es 50 MHz, esto se logra por medio del registro CLKGDV (archivo McBSP.c ubicado en los anexos), el cual se encarga de aplicar un divisor a la frecuencia de transmisión, como se muestra en la siguiente fórmula:

$$CLKG\_frecuency = \frac{Input\_clock\_frecuency}{CLKGDV + 1} \qquad \text{(III.14)}$$

III.2.3 Transmisor en FPGA para McBSP

El transmisor programado en el FPGA será el encardo en enviar la lectura de los encoder y ambos coeficientes para calcular la velocidad (Cp y Ct). La máquina de estados (**Figura III.15**) contenida en la programación debe marcar el protocolo de envío de datos, respetando el inicio marcado por FS y el término de la transmisión (EOT).

El diagrama a bloques del transmisor tiene como entradas el dato y FS, de esta manera se conoce el inicio de transmisión y el fin, el cual es indicado por la salida EOT (End Of Transmition). Además, se colocó un bloque de *Single Pulse* para evitar rebotes o recibir varias señales en el inicio de transmisión (**Figura III.16**).

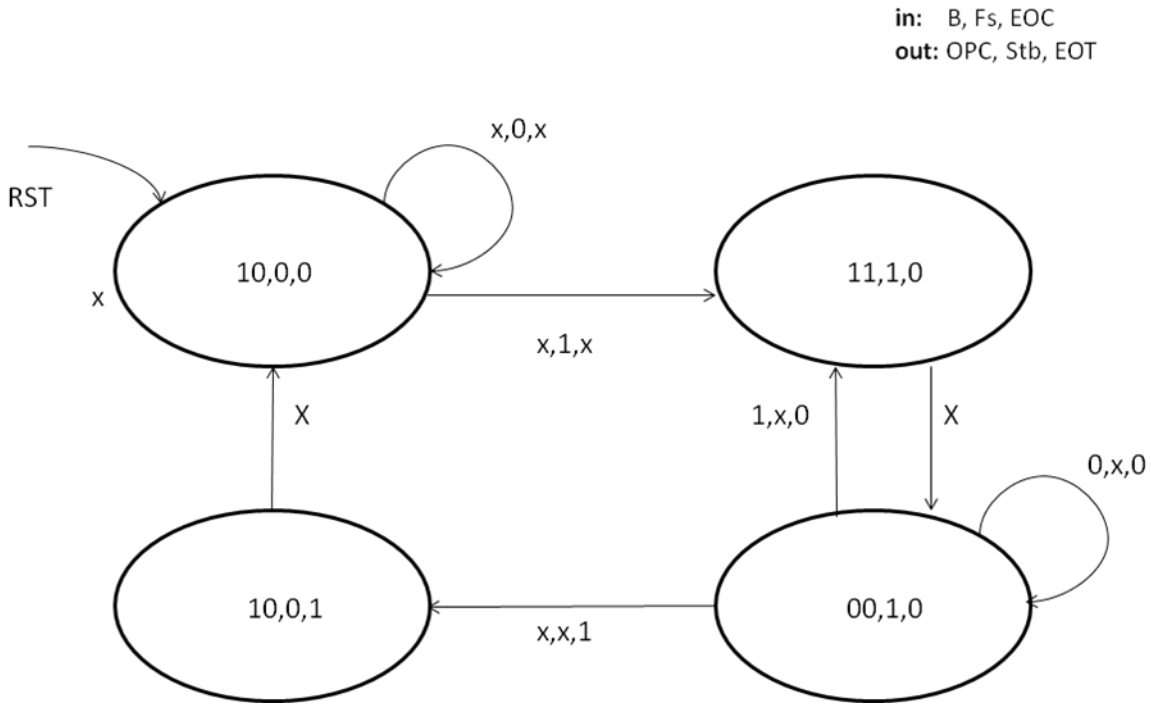


Figura III.14 FSM_TransmisorMcBSP

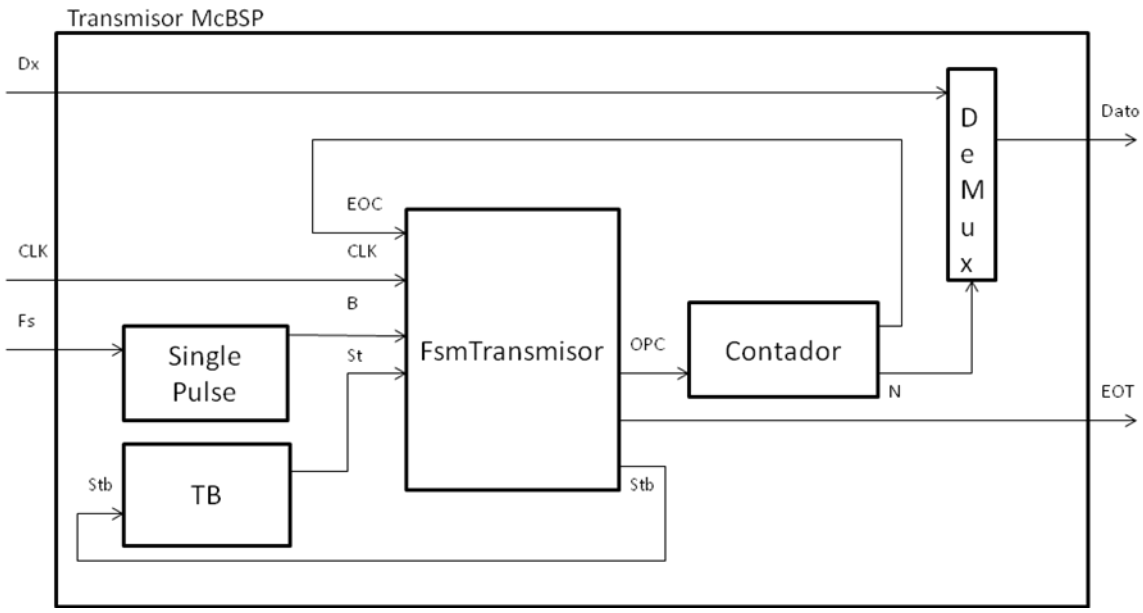


Figura III.15 Diagrama a bloques del Transmisor por McBSP

III.2.4 Receptor en FPGA para McBSP

En el receptor es necesario tener una base con el doble de la frecuencia de transmisión, así será posible conseguir tomar el dato a la mitad de su aparición (variable B en la **Figura III.17**), esto disminuye la posibilidad de recolectar datos errados o defasados. Una vez que se tiene la base correcta, es cuestión de programar una FSM (Finite State Machine) para coordinar todos los registros de los datos obtenidos.

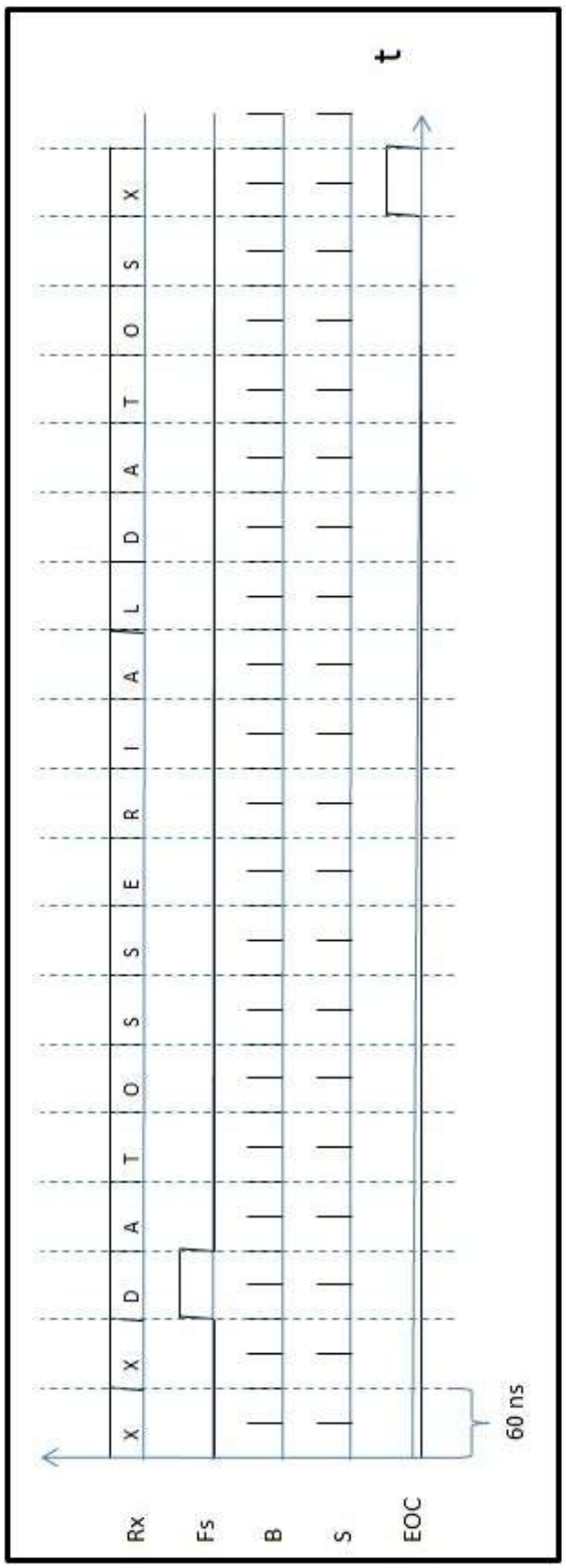


Figura III.16 Muestreo para adquisición de datos de McBSP en FPGA

La máquina de estados finitos (FSM en **Figura III.18**) tiene como función, capturar el momento en que FS esté activado, una vez recibida la señal comienza a recibir datos, haciendo la captura (B) a la mitad de su aparición por medio de S, así hasta que se active EOR por medio del contador.

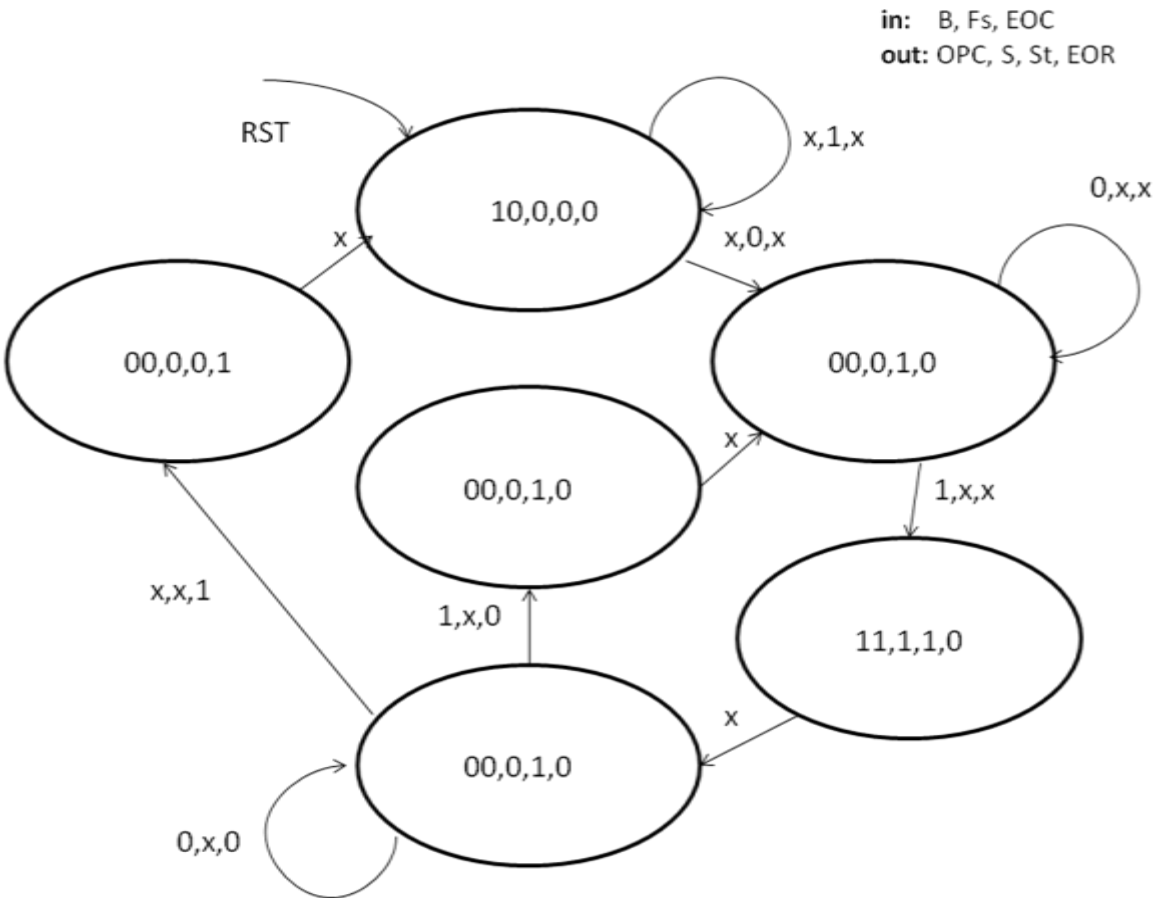


Figura III.17 FSM_ReceptorMcBSP

Dentro del diagrama a bloques (**Figura III.19**) para la programación en VHDL se puede encontrar un registro conectado a un flip-flip, el papel de este enlace es guardar los datos ya recibidos en una pequeña memoria, igual que la memoria FIFO en el DSP.

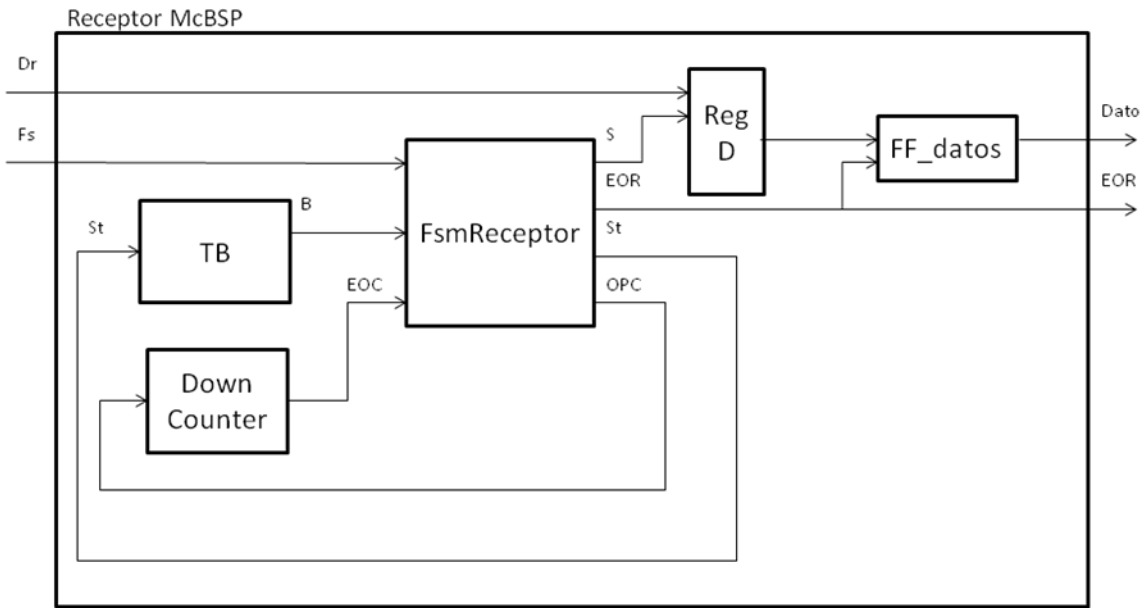


Figura III.18 Diagrama a bloques del Receptor por McBSP

III.2.5 Algoritmo adaptivo para medición de velocidad

El algoritmo adaptivo utilizado para medir la velocidad, fue propuesto en función de encontrar una solución para la medición de velocidad a bajas velocidades por medio de encoders. La problemática surge debido a que los encoders funcionan a través de un número finito de marcas percibidas por un foto-transistor, así que con un pequeño número de marcas o pulsos la resolución puede bajar, y por lo tanto las interrupciones entre pulsos son lo suficientemente largas para arrojar una medición errónea.

El encoder se utiliza para medir la posición (Sancho, 2006) o la velocidad angular, sin embargo es más común en la medición de la velocidad. Cuando ésta es baja los algoritmos de tratamiento de señales se hacen indispensables porque el error de la medida incrementa con la disminución de la velocidad. Regularmente el programador opta por calcular la velocidad por medio del índice que brinda el encoder, de esta manera sólo necesita contar el número de revoluciones y compararlas con la base de tiempo. Sin embargo cuando la velocidad es demasiado baja, el tiempo que permanece el índice del encoder en bajo será tan grande que el resultado de la resta en la derivada se torna muy grande, y si no existe una correcta elección de la base de tiempo, resulta una medición ruidosa o muy cercana a cero, como se puede observar en

la **Ecuación III.15**, donde la posición está dada por X_n (en la ecuación X_2 es la posición actual y X_1 la posición anterior) y un periodo de muestreo por t.

$$\dot{y} = \frac{x_2 - x_1}{t} \tag{III.15}$$

Para el cálculo de la velocidad son necesarias varias consideraciones antes de empezar a trabajar, por ejemplo, el mínimo de resolución aceptable para mediciones de velocidad, fue establecido por Lygouras (1998) y es de 0.09°, esta resolución se puede calcular dividiendo la rotación entre el número de pulsos que brinda el encoder durante ese recorrido (**Ecuación III.16**).

$$r = \frac{360}{PPR} \tag{III.16}$$

Así, tomando en cuenta que los PPR (Pulsos Por Revolución) de nuestro encoder son 1024, la resolución resultante es 0.35156°, teniendo en cuenta lo anterior, el factor por medio del cual se acerca más a 0.09° es 4, ya que 1024x4=4096 y empleando la fórmula (2), se obtiene una resolución de 0.08789°. El diagrama a bloques que describe la multiplicación dentro del FPGA se presenta en la **Figura III.20**. Este diagrama a bloques es una gran aportación por parte de Lygouras, contiene una combinación de compuertas para lograr el aumento de la resolución con el uso exclusivo de los canales del encoder y un reloj. Logrando una salida con el número de cuentas desarrolladas durante la base de tiempo pero con una resolución al cuádruple. En otros trabajos se pudieron encontrar algoritmos similares, sin embargo, la sencillez en cuanto a entradas y salidas del propuesto a continuación, fue factor determinante para su uso.

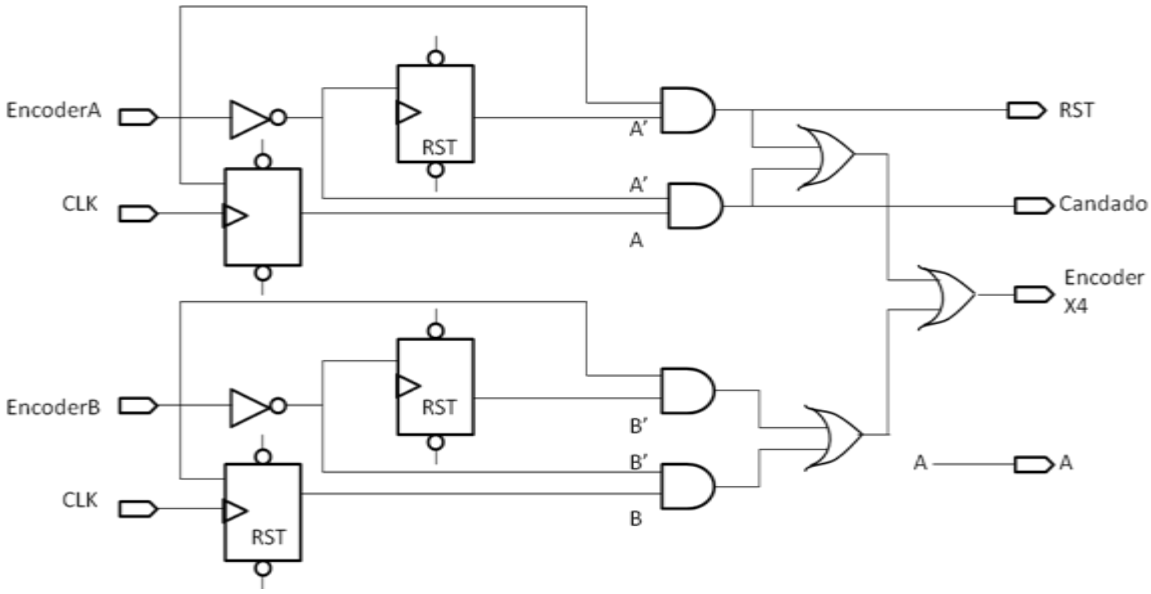


Figura III.19 Aumento al cuádruple la resolución del encoder (Lygouras, 1998)

El siguiente paso a seguir es realizar un algoritmo que identifique la dirección en la cual se está desarrollando el movimiento, esto es posible únicamente con ambos canales del encoder debido a su desfasamiento de 90°, así, tomando en cuenta cuál aparece primero se puede saber la dirección de giro.

Existen numerosos algoritmos para la detección de giro (Rairán, 2009), como puede ser el utilizar una señal cuasi triangular, en lugar de una onda cuadrada convencional (procedimiento paralelo, indicado sólo para ampliar el panorama del lector), sin embargo existen opciones más sencillas como es la siguiente (**Figura III.21**, **Tabla III.6**), este arreglo de compuertas exhibe el resultado con un uno en el bit Izquierda o Derecha, y las entradas simplemente están compuestas por ambos canales del encoder, a comparación con un encoder incremental, no es tan trivial el reconocimiento en la dirección de giro, sin embargo con ayuda de este arreglo de 6 compuertas es posible conocerla.

Tabla III.6 Tabla de verdad para arreglo lógico de detección de dirección

Canal A	Canal B	Dirección
0	0	X
0	1	Izquierda
1	0	Derecha
1	1	X

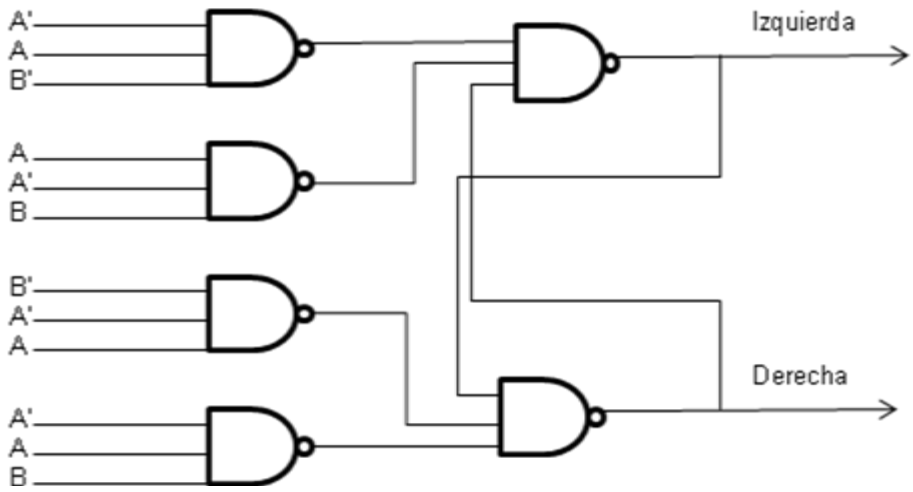


Figura III.20 Detección de giro dentro del algoritmo adaptivo

De tal manera que conjuntado los bloques anteriores y adicionando un contador podemos comenzar a medir la posición, claro está que al ser conocida la dirección de movimiento el contador debe ser ascendente/descendente para retroceder de manera natural al igual que la flecha del motor. De esta manera se puede formar un diagrama general (**Figura III.22**) del sistema adaptivo.

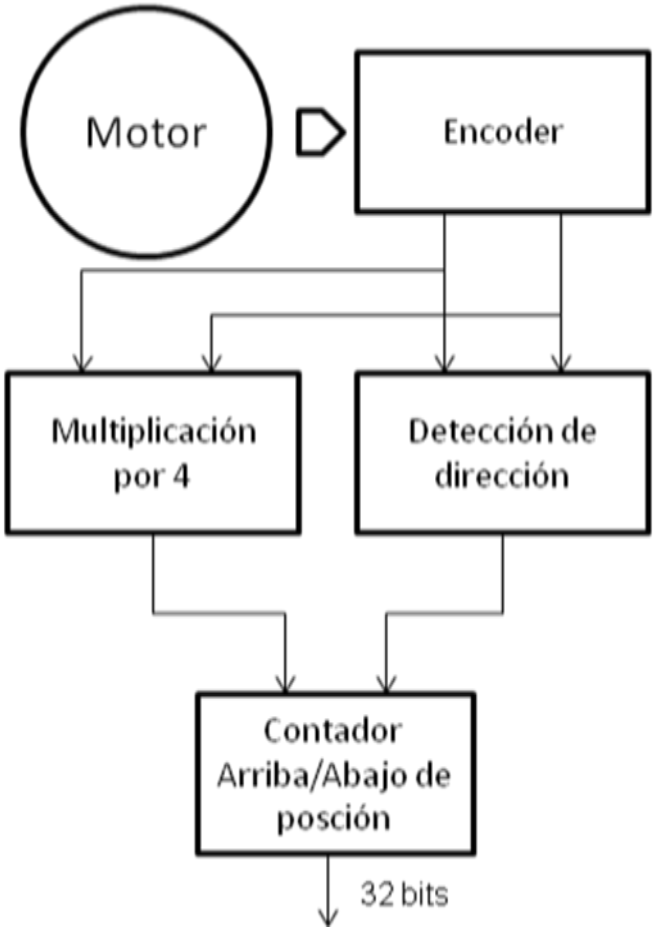


Figura III.21 Diagrama general a bloques del algoritmo adaptivo

Finalmente, se procede a diseñar un algoritmo capaz de medir la velocidad (**Figura III.23**), para esto se tomó una tabla de valores realizada por el mismo Lygouras (1998), en la cual se eligieron los rangos de velocidad por medio de experimentación y un acuerdo común, se indica un valor de tiempo específico del periodo de giro para distintas velocidades.

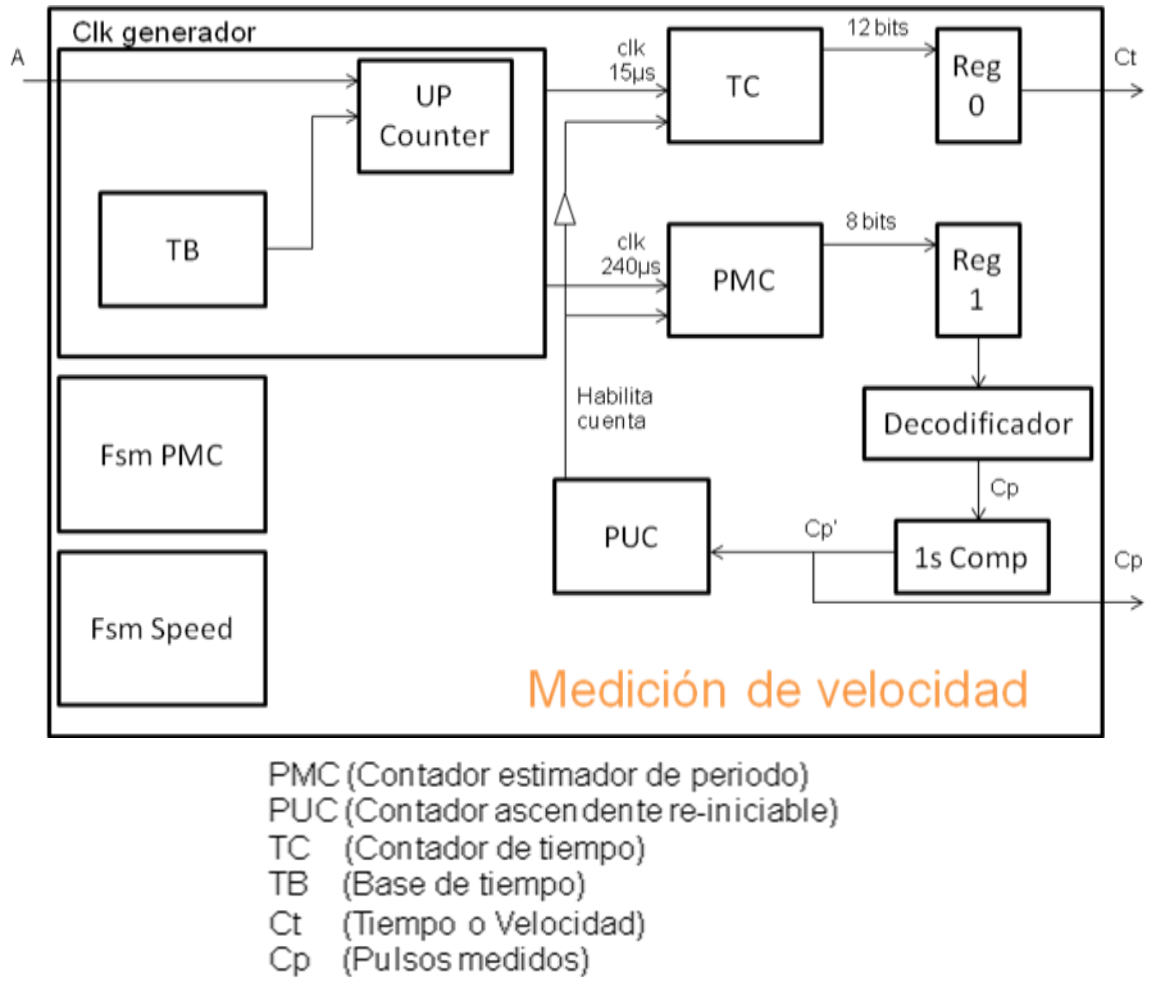


Figura III.22 Medición de la velocidad, diagrama a bloques

El algoritmo cuenta con una base de tiempo (TB) y un contador (UP Counter) con los cuales se mide el tamaño del periodo en un pulso del canal A del encoder (PMC), dependiendo de la medición, el valor es comparado en una base de datos previamente establecida (Decodificador, **Tabla III.3**). El encargado de medir el periodo del pulso es el PMC (Contador Estimador de Periodo, **Figura III.24**), esta medición se registra y es enviada al Decodificador, el cual su vez hace la comparación de la medición de periodo con una base de datos ya pre-establecida (también dentro del Decodificador). Una vez ubicado el periodo correspondiente, se asigna un valor de tiempo (programado en el PUC) para volver a ordenar al PMC que realice una nueva medición. Este nuevo valor es obtenido de la práctica, sabiendo que un motor tiene una velocidad estable por cierto periodo de tiempo dependiendo su velocidad.

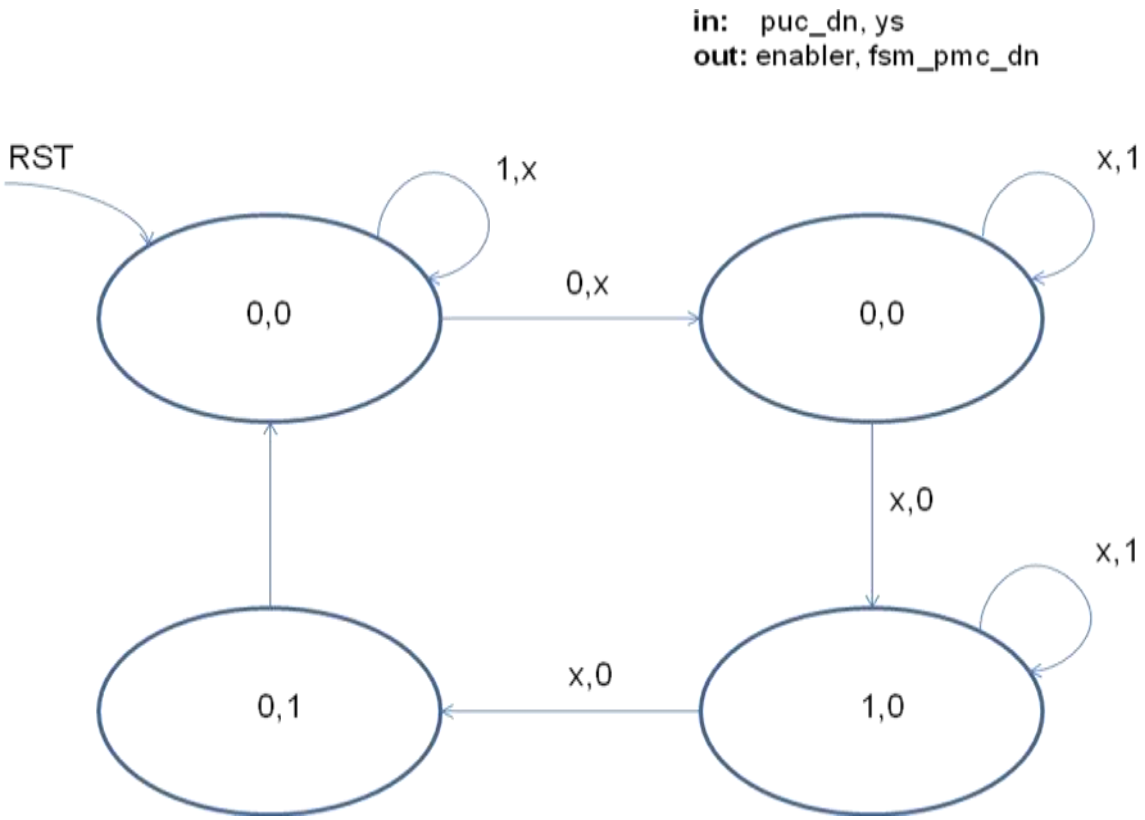


Figura III.23 FSM_PMC

La tabla a continuación muestra las comparaciones existentes a partir del periodo de un pulso del encoder (T), medido por medio del PMC, resultando una velocidad rotacional y un intervalo de muestreo, el cual corresponde al tiempo que tardará el PMC en volver a estimar T.

Cada medición tiene un error, el cual puede ser despreciable para la mayoría de las aplicaciones, a menos que se trate de una tarea de alta precisión como pueden ser algunas aplicaciones láser, sin embargo un robot de 6 GDL normalmente se emplea en el área automotriz, con usos como pintura, soldadura y ensamble. Así que el error encontrado en cada una de las escalas de velocidad no afecta de manera significativa en el trabajo final que desempeñan de los robots.

Tabla III.7 Tabla de valores adaptivos para la velocidad

Velocidad rotacional del motor (RPM)	Periodo T de un pulso del encoder (µs)	Contador estimador de periodo (8 bits)	Salida del decodificador (7 bits)	Complemento a 1 del Preset Cp	Intervalo de muestreo Ts(Ts=Cp*T(µs))	Error (=15µs/Ts)
2929.7	20	00000000	0000000	127	2540	0.0059
122.07	480(-)	00000001			60960	0.00024
122.07	480(+)	00000010	1000000	63	30240	0.00049
61.03	960(-)	00000011			60480	0.00024
61.03	960(+)	00000100	1100000	31	29760	0.0005
30.52	1920(-)	00000111			59520	0.00025
30.52	1920(+)	00001000	1110000	15	28800	0.00052
15.258	3840(-)	00001111			57600	0.00026
15.258	3840(+)	00010000	1111000	7	26880	0. 00055
7.629	7680(-)	00011111			53760	0. 00027
7.629	7680(+)	00100000	1111100	3	23040	0.00065
3.814	´15360(-)	00111111			46080	0.00032
3.814	15360(+)	01000000	1111110	1	15360	0.00097
0.957	61200(-)	11111111			61200	0.00024

Como parte final del diseño se necesita una FSM que controle todo el sistema de medición de velocidad**,** así se elaboró FSM_SPEED (**Figura III.25**), esta máquina de estados finitos tiene como principal tarea, coordinar el tiempo que debe esperar el PMC para volver a calcular la velocidad, esto se logra a partir de un contador ascendente reiniciable (PUC), para finalmente hacer la debida comparación del periodo dentro del Decodificador.

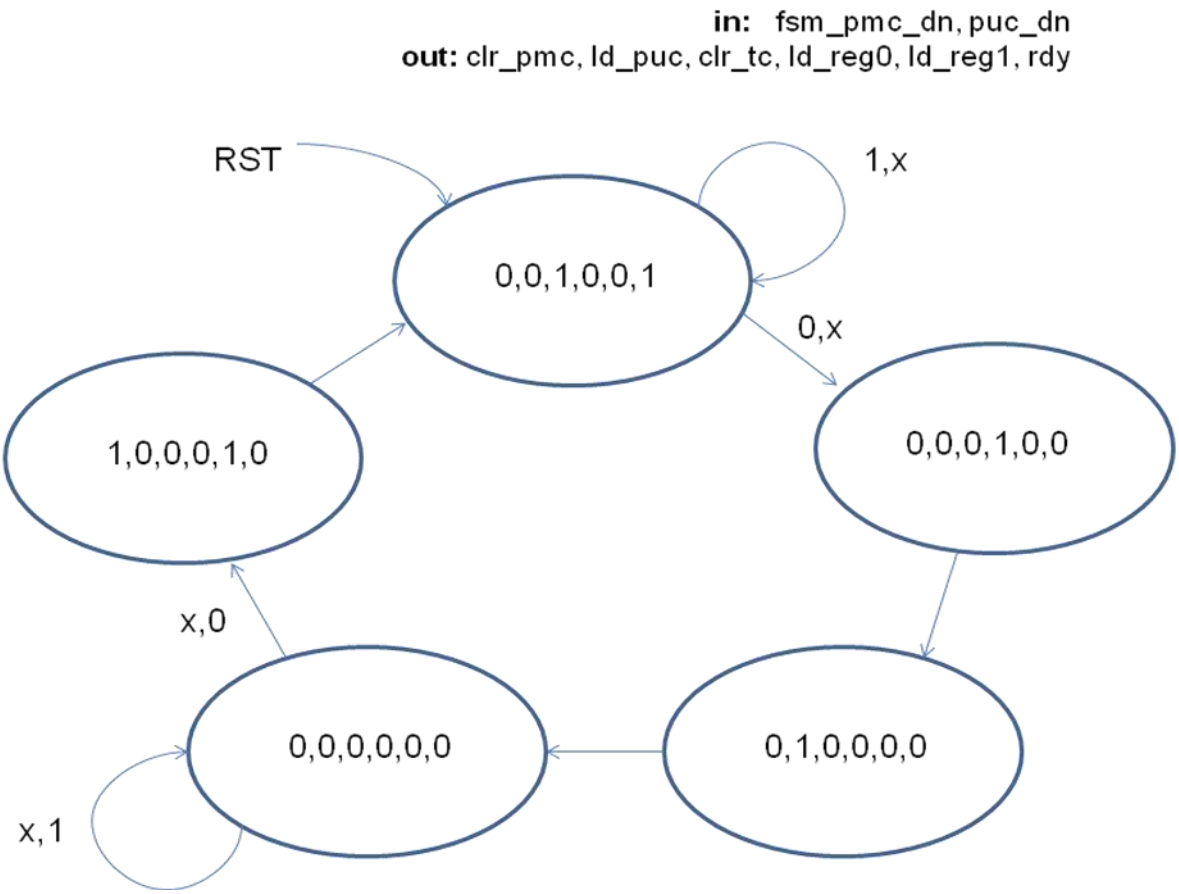


Figura III.24 FSM_SPEED

En la **Figura III.26** se muestra una fotografía del sistema para pruebas del algoritmo de medición de velocidad. Se acopló un motor de CD extraído de una de las articulaciones del robot y se acopló con el encoder para posteriormente energizarse, dentro del FPGA se halla programado el algoritmo adaptivo de medición de velocidad.

Figura III.25 Fotografía de validación del algoritmo adaptivo con motor de CD

III.2.1 Código en lenguaje C para control de motor

Para empezar, se hace hincapié en el motivo de programar el controlador PID dentro del DSP, y esto es debido al tiempo muy reducido de desarrollo que representa, respecto al necesario para programar un controlador de esta magnitud en VHDL, que es un lenguaje de más bajo nivel.

El código de este conjunto de algoritmos con el menú principal se adjunta dentro de los anexos ubicados al final de este trabajo (PID_Tesis.c). Y a continuación se hará una pequeña revisión acerca de su funcionamiento.

Como cualquier código en lenguaje C, hay que considerar el conjunto de librerías para poder emplear todas las funciones necesarias para la programación del PID, PWM, QEP, etc. Se comienza con DSP281x_Device.h, esta librería incluye otras necesarias para utilizar el TMS320F2812, tales como el McBSP, el SPI, ADC, GPIO, etc. Continuando nos encontramos

con otra librería madre importante, ésta es parameter.h, de la misma manera este archivo llama otros archivos de cabecera como son: PID_REG3.h, rmp_cntl.h, speed_est.h, f281xqep_no_index.h; además de establecer parámetros necesarios para la programación como lo es el número π, la frecuencia mecánica y la velocidad del motor.

Posteriormente se asignan los valores por defecto a las funciones principales: PWM, PID, QEP, RAMP, SPEED y DATALOG. Para el PID, es necesario el cálculo de las ganancias, colocar una velocidad de referencia y diferenciar la dirección de rotación en SPEED, un valor trigonométrico para DATALOG y las consideraciones del QEP, como es la resolución del encoder y el tamaño del prescalador.

Ahora hay que actualizar el valor de RawTheta, que es el portador de la posición del encoder, así mismo con las funciones speed.calc y pid1_pos.cal, se puede calcular la velocidad y la posición, respectivamente. De esta manera se prosigue, una vez conseguida la posición actual por medio del QEP_NO_INDEX_DRV se puede encontrar la velocidad (SPEED_EST) y de la salida de ambos se calcula un PID de posición y otro de velocidad, cada uno con su respectiva referencia, de esta manera podemos calcular una salida de ancho de pulso por medio del BDC_PWM_DRV y controlar nuestro motor desde el circuito de potencia.

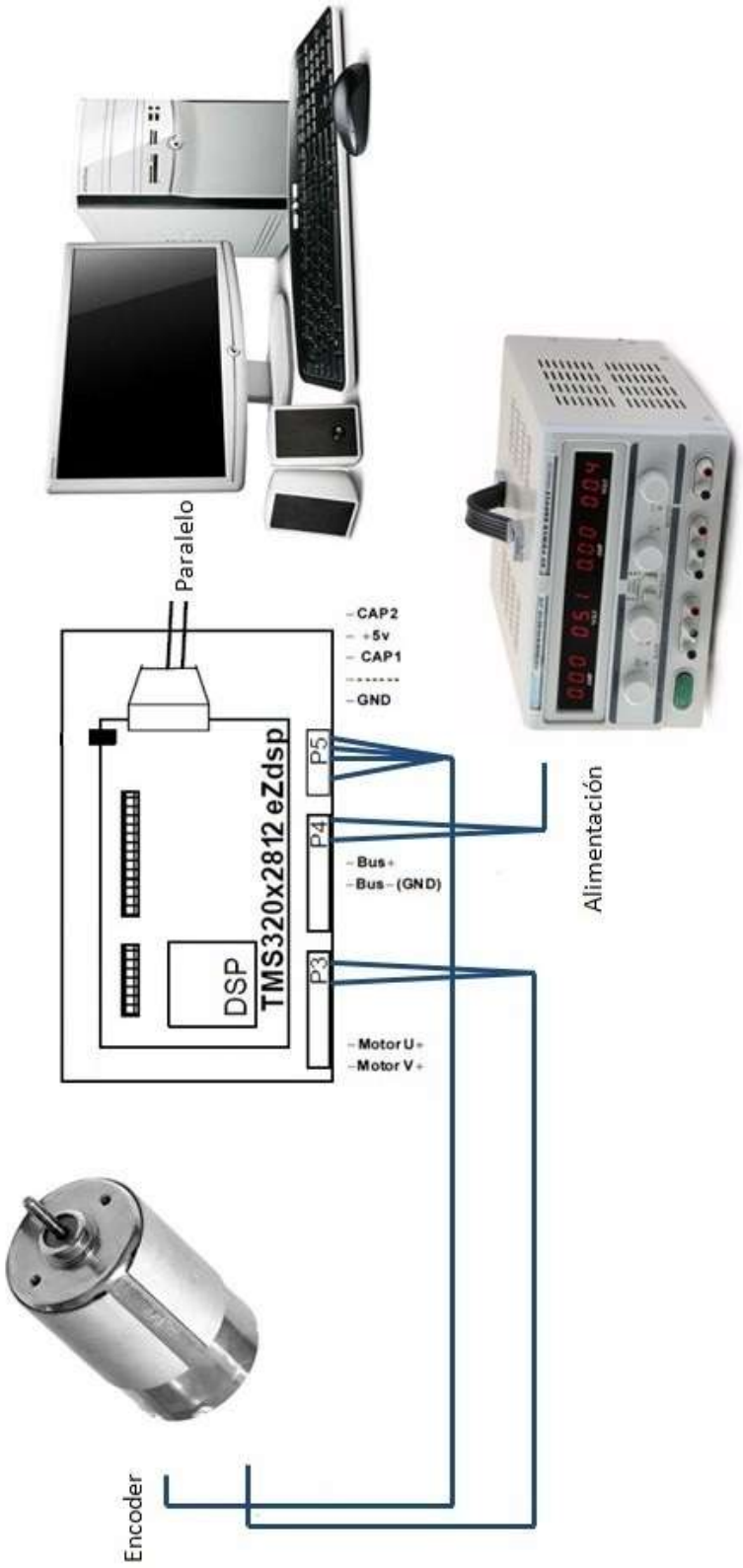


Figura III.26 Diagrama de conexión DSP/Motor/Encoder

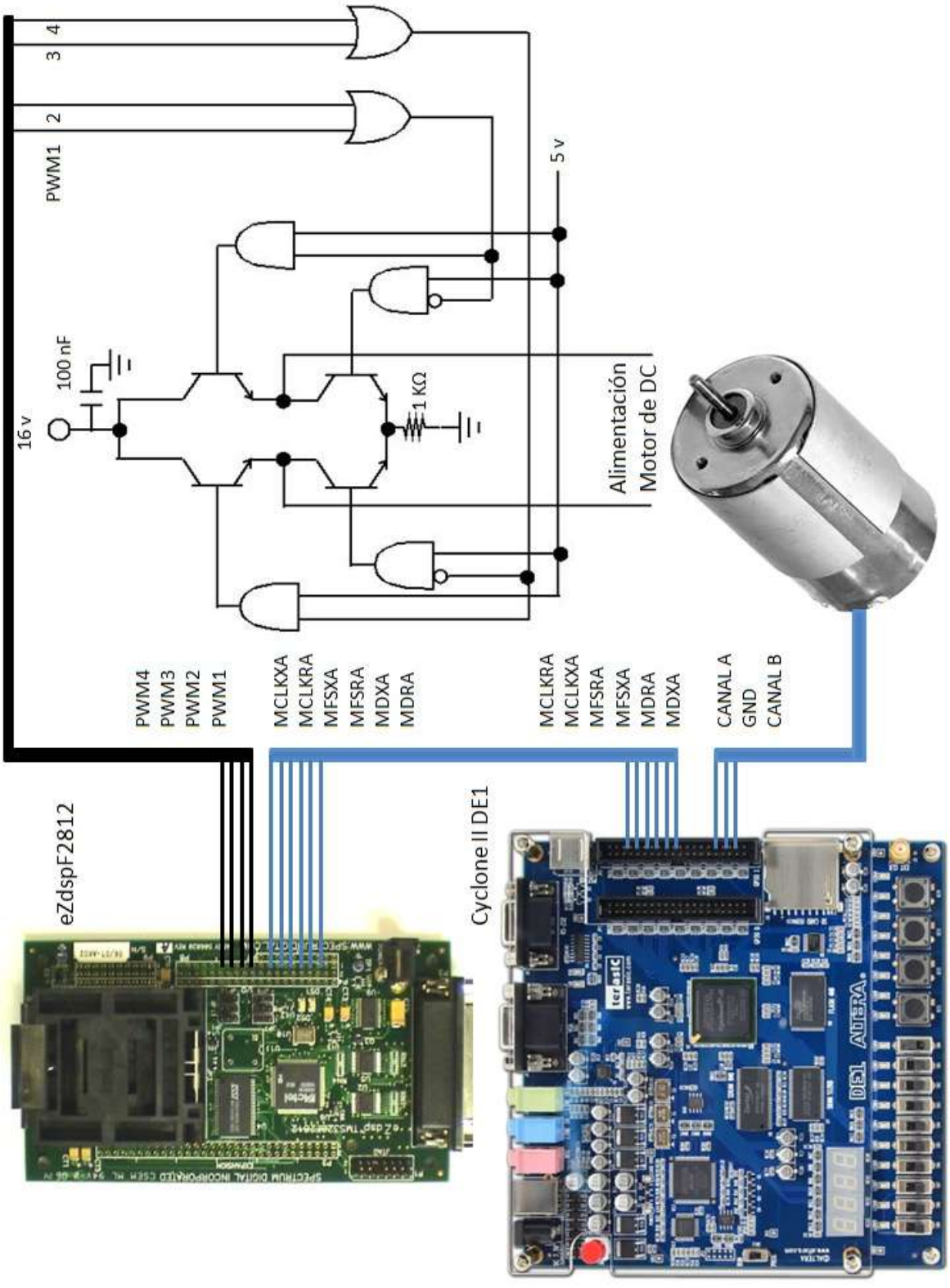


Figura III.27 Diagrama general del sistema

CAPÍTULO 4:

IV RESULTADOS Y DISCUSIÓN

IV.1 Resultados

El robot poliarticulado consta de un motor de CD por articulación, estos permiten el movimiento para formar las trayectorias trazadas por el actuador del robot, para facilitar el análisis de los resultados se presenta el trabajo con un motor por separado. Y la implementación completa estará formada por repeticiones de los códigos y conexiones a continuación mostradas.

IV.1.1 Algoritmo adaptivo de velocidad

Para conocer los coeficientes resultantes (Cp y Ct) independientemente del DSP, se desplegaron los resultados en los displays de 7 segmentos del FPGA. Así, con ayuda de una calculadora se hizo la medición manualmente y fue posible calcular la velocidad. Por otra parte en los leds verdes y rojos se programó el avance de las muescas del encoder, conociendo así la posición. En la **Figura IV.1** se pueden apreciar los valores Ct y Cp, los cuales fueron obtenidos por medio del analizador lógico, de esta manera se podía alimentar el motor con un voltaje constante y apreciar los niveles lógicos en el TLA5201B.

Tabla IV.1 Tabla de valores para Cp y Ct, obtenidos mediante praxis

	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
Cp						1	1	1	1	1	1	1
Ct	0	1	1	1	1	0	1	1	0	1	1	0
Cp						1	1	1	1	1	1	1
Ct	0	1	0	0	0	0	1	0	1	1	0	0
Cp						1	1	1	1	1	1	1
Ct	0	0	1	0	0	0	0	0	0	1	0	1

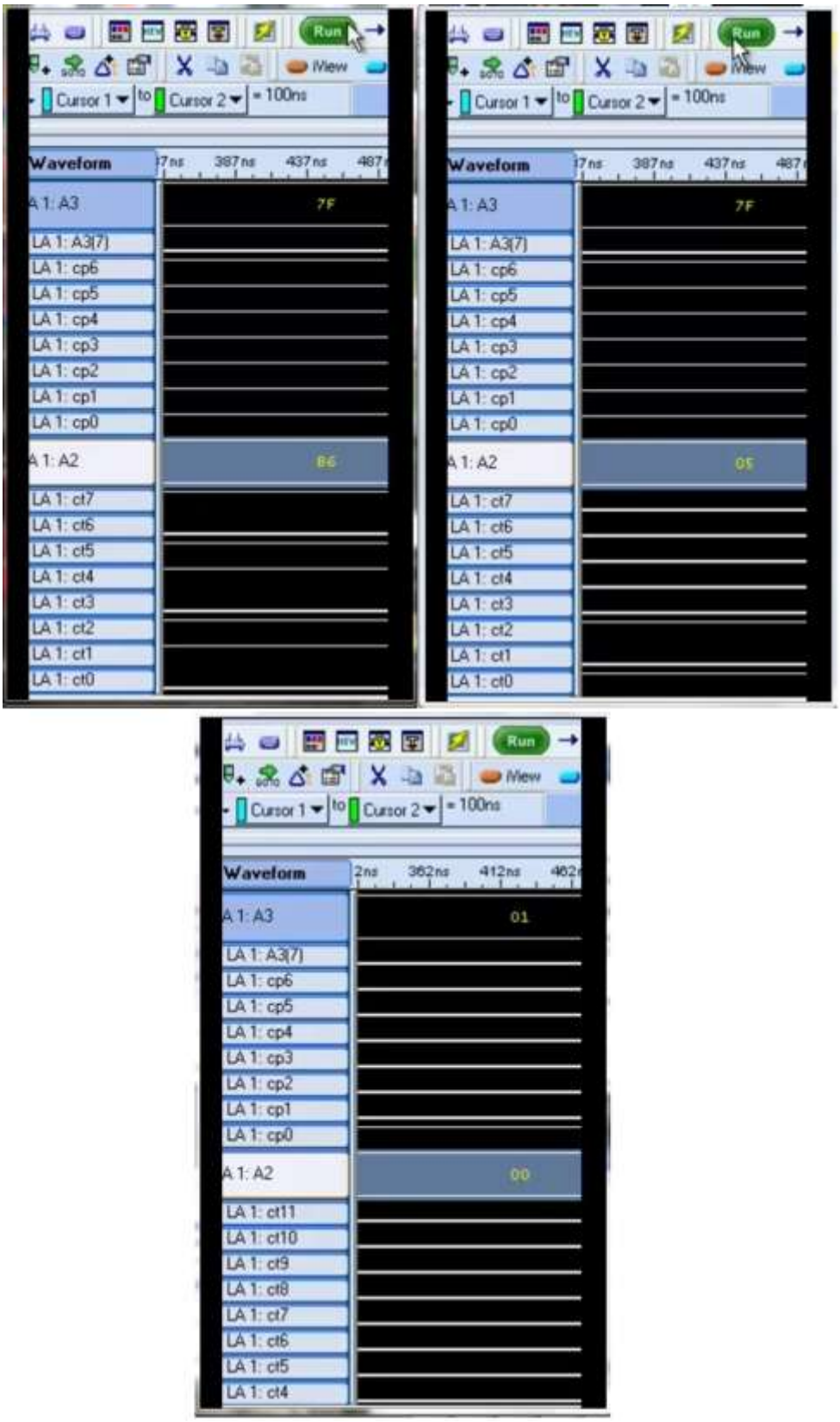


Figura IV.1 Coeficientes Ct y Cp, exhibidos en los Displays de 8 segmentos de la DE1

Los cuales corresponden a los siguientes números en base decimal:

Cp1=127	Cp2=127	Cp3=127
Ct1=1974	Ct2=1068	Ct3=517

Finalmente, gracias a esta representación decimal se puede aplicar la **Ecuación IV.1.**

$$N_{[rpm]} = \frac{C_p * 3906.26}{C_t} \qquad \text{(IV.1)}$$

En la **Tabla IV.2** se muestra la magnitud de la velocidad para valores constantes en la alimentación del motor.

Tabla IV.2 Tabla de velocidades dependiendo el voltaje de alimentación

$N_{[rpm]} = \frac{(Cp*8247.42)}{Ct}$

$\frac{(127*8247.42)}{1974}$	=530.6 rpm	en 5.33 V
$\frac{(127*8247.42)}{1068}$	=980.73 rpm	en 10.59 V
$\frac{(127*8247.42)}{517}$	=2025.96 rpm	en 18.93 V

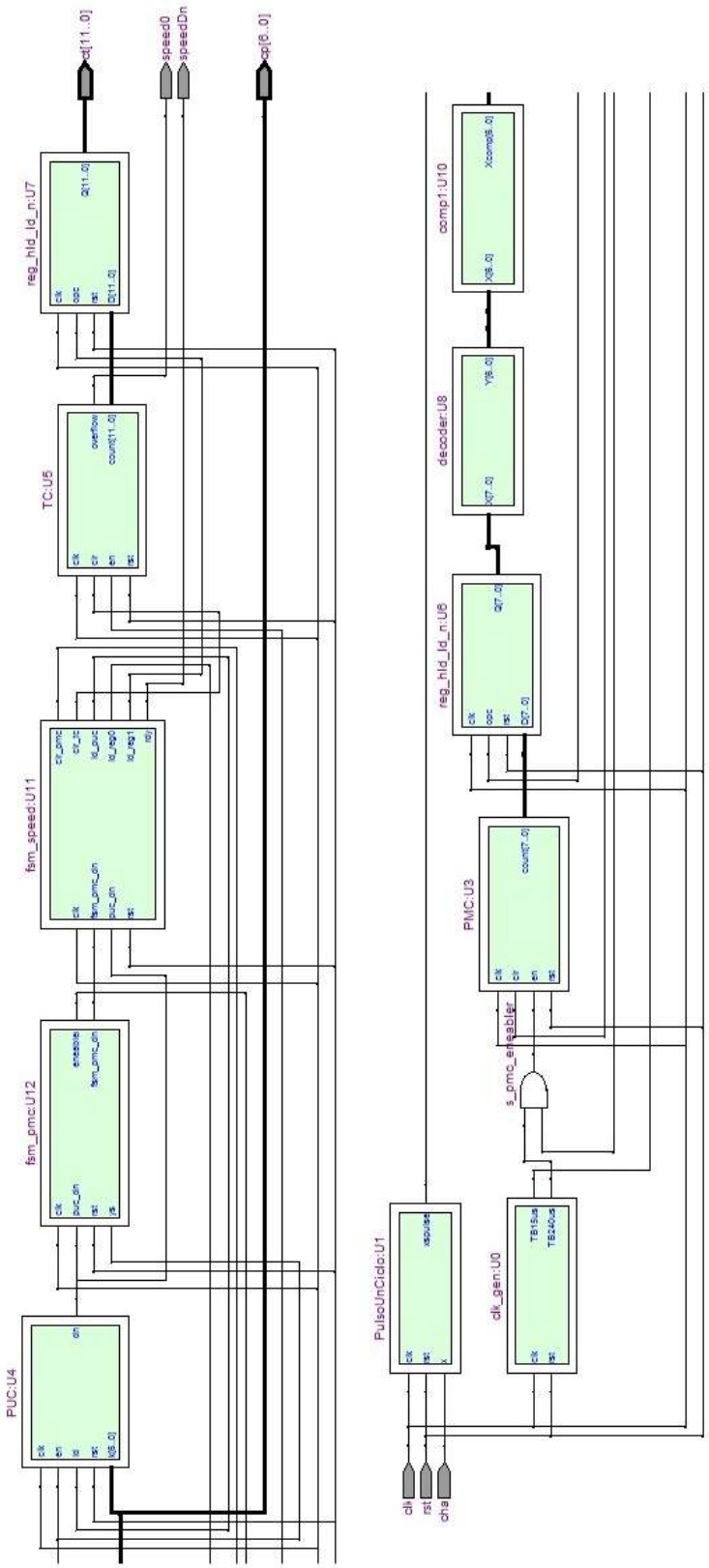


Figura IV.2 Diagrama RTL (Resistor Transistor Logic) de algoritmo adaptivo para medición de velocidad

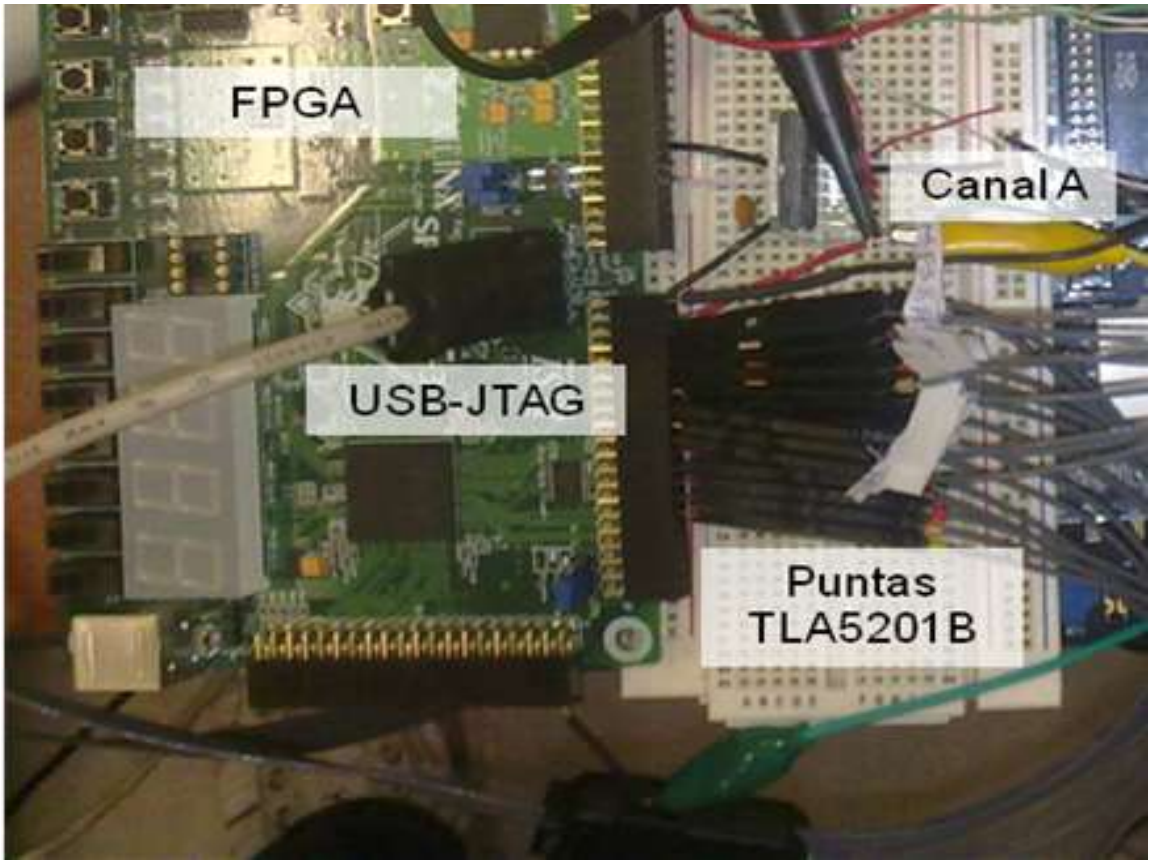


Figura IV.3 Fotografía de sistema implementado para medición de velocidad

IV.1.2 Interfaz McBSP

Para la visualización del flujo de datos entre el TMS320F2812 y la DE1, fue necesario emplear un Analizador Lógico (TLA5201B) de Tektronix, los primeros resultados se obtuvieron de programar el McBSP con un Loopback digital dentro del DSP.

La primer prueba del protocolo de comunicación fue empleando un tamaño de 8 bits para la dimensión de palabra, los cuales se pueden apreciar en la **Figura IV.4** con un total de 4 niveles en el FIFO. Además, en cada nivel se aprecia también el dato enviado, que corresponde a 0x0081. Éste fue el primer paso para realizar la conexión con el FPGA.

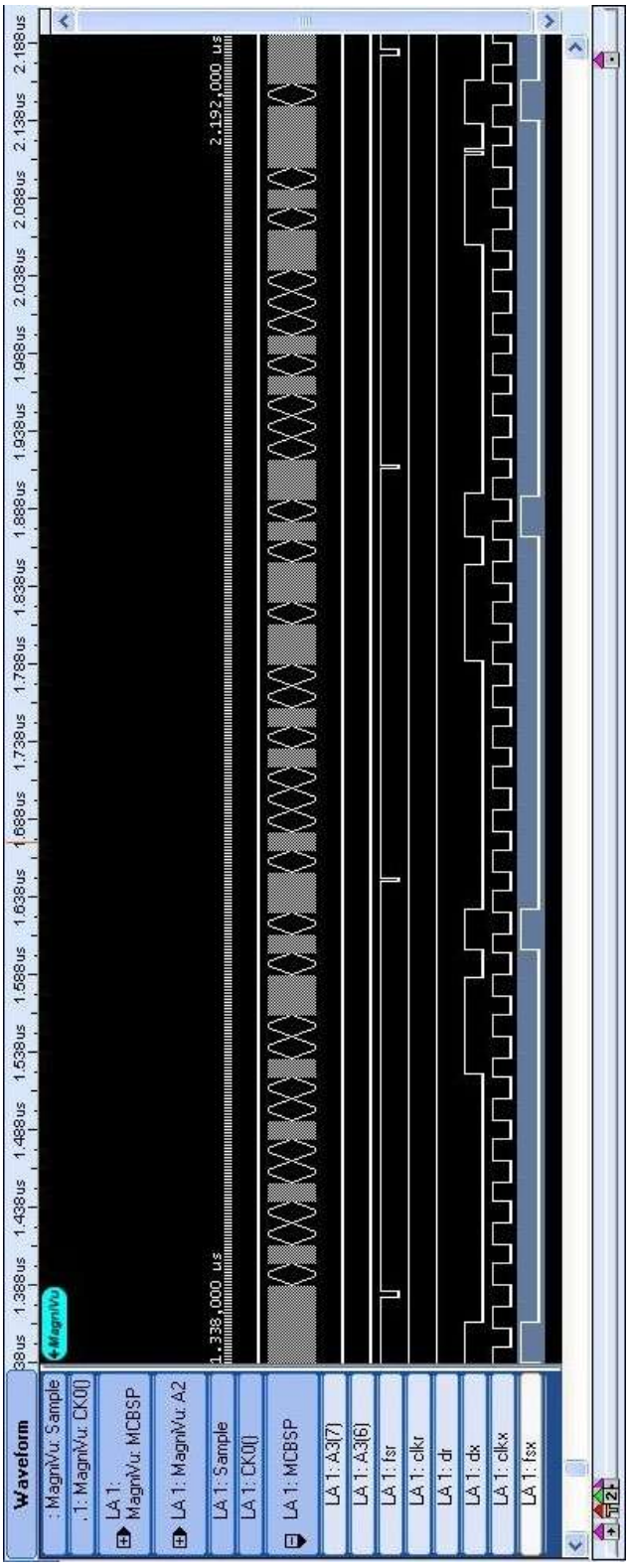


Figura IV.4 Captura de pantalla tomada en el TLA5201B que muestra 4 niveles en el FIFO

Para la adquisición de los datos se conectaron las salidas MCLKXA, MCLKRA, MFSXA, MFSRA, MDXA y MDRA (**Figura IV.5**) en el analizador lógico. Los periféricos que ofrece el DSP por medio del kit eZdsp se pueden encontrar en el documento *Technical Reference eZdsp F2812*, descargable en la página http://www.spectrumdigital.com.

Figura IV.5 Señales provenientes del DSP adquiridas por medio del TLA5201B

IV.1.3 Sintonización de controlador PID

Para validar la conexión del híbrido entre el DSP y el FPGA, se programó un controlador PID dentro del DSP, éste fue elaborado en C y el código corresponde a la siguiente ecuación:

$$PID = K_p e_f(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t) \tag{IV.2}$$

Partiendo del conocimiento que un motor de CD es un sistema de primer orden, el modelo que describe este sistema se representa con la siguiente ecuación:

$$\dot{y} + ay = KA \tag{IV.3}$$

Y aplicando la transformada de Laplace.

$$sY(s) + aY(s) = \frac{KA}{s} \tag{IV.4}$$

Donde la entrada U(s) es igual a 1/s, así, despejando Y/U obtenemos la siguiente planta con su respectivo diagrama de bloques.

$$\frac{Y(s)}{U(s)} = \frac{KA}{(s+a)} \tag{IV.5}$$

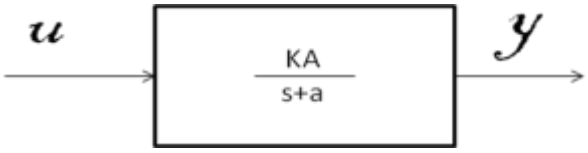


Figura IV.6 Modelo a bloques del comportamiento del motor

De modo que dentro de la **Figura IV.7** podemos apreciar el valor de la referencia (KA)/a para la respuesta del sistema.

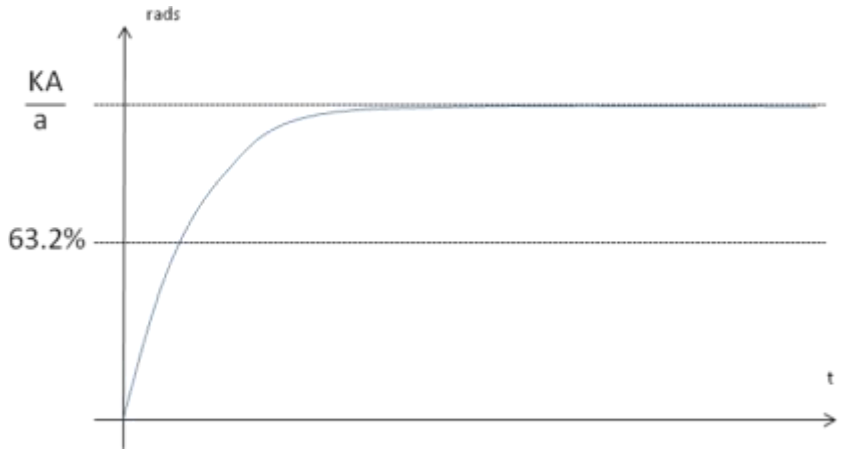


Figura IV.7 Cálculo de 1/a para la caracterización del motor acorde a un impulso de 4 V

Posteriormente en *simulink*, se realizó la programación a bloques del sistema (**Figura IV.8**), al cual se le introdujo una entrada impulso con valor de 4 V, seguida de nuestro controlador PID (con las ganancias a continuación calculadas) y la planta obtenida del modelado del sistema.

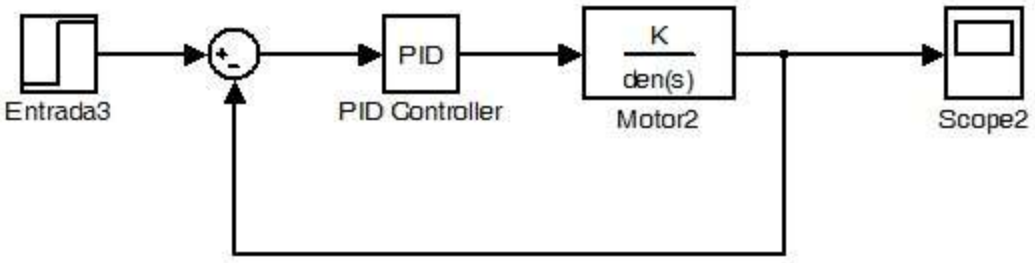


Figura IV.8 Diagrama a bloques en Simulink para graficar la respuesta del motor de CD

Y la siguiente **Figura IV.9** muestra la simulación anterior.

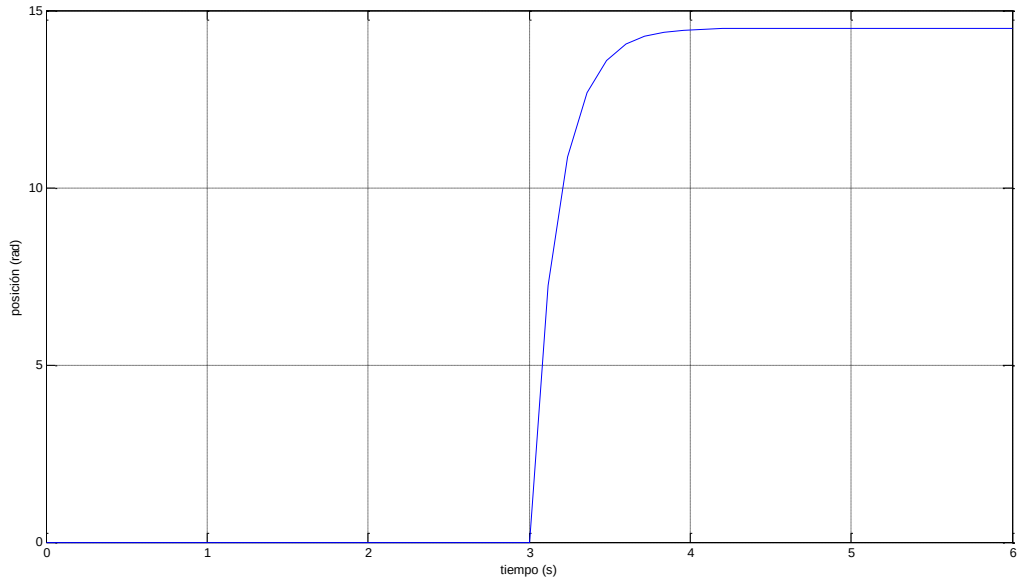


Figura IV.9 Gráfica de la simulación de la respuesta del motor, obtenida a partir de Simulink

Como se puede apreciar en la gráfica anterior, el valor de tendencia es 14.5 rad, es decir

$$\frac{KA}{a} = 14.5\,\text{rad} \qquad \text{(IV.6)}$$

La **Figura IV.10** es un acercamiento a la gráfica obtenida a partir de un programa en *Matlab* que grafica una serie de datos almacenados en dentro de un archivo de texto, tales datos fueron obtenidos por un PIC16F877A que calcula la respuesta del motor ante un impulso como entrada.

Este impulso A en realidad es un valor constante a lo largo de dos segundos dentro de la simulación, ya que en la realidad es imposible provocar una respuesta ante impulsos que duran un mínimo de tiempo. De esta manera para conocer el valor de la ganancia K, tenemos que calcular 1/a, el cual está dado por el 63.2% del valor de tendencia sobre la señal de respuesta.

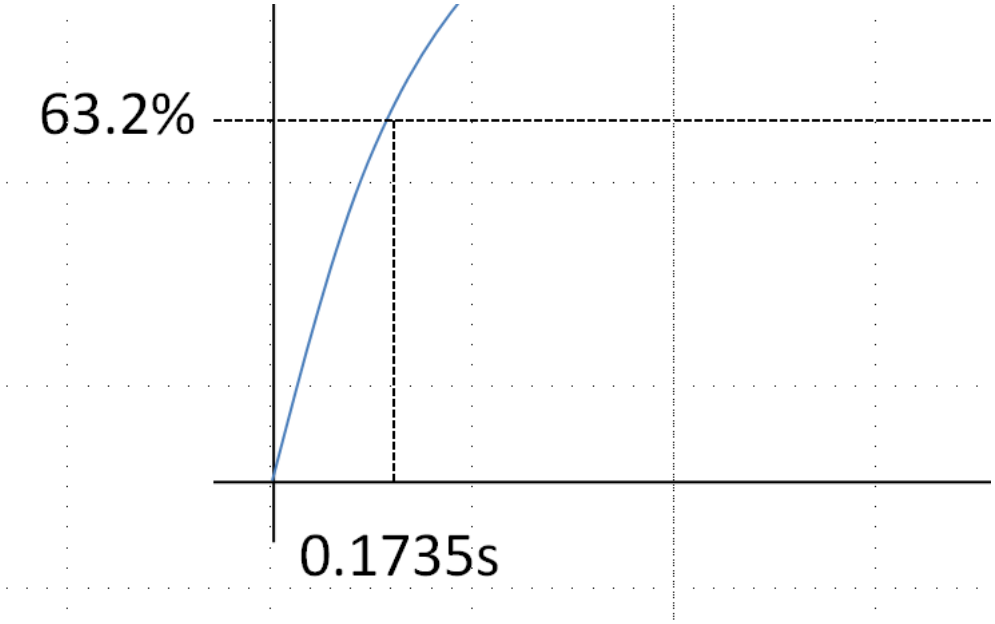


Figura IV.10 Acercamiento en imagen para obtener el valor exacto de 1/a

Y sustituyendo todos los valores.

$$K = \frac{(5.78)(19.5)}{4} = 20.95 \qquad \text{(IV.7)}$$

La **Figura IV.11** muestra la respuesta real del motor de CD, graficada a partir de los datos de posición con respecto al tiempo obtenidos con el PIC. Esta gráfica es obtenida por medio de un script en *Matlab* que obtiene los datos dentro de dos columnas en el archivo de texto (*.txt), la primera representa el tiempo y la segunda el avance de la flecha del motor acoplada a un encoder.

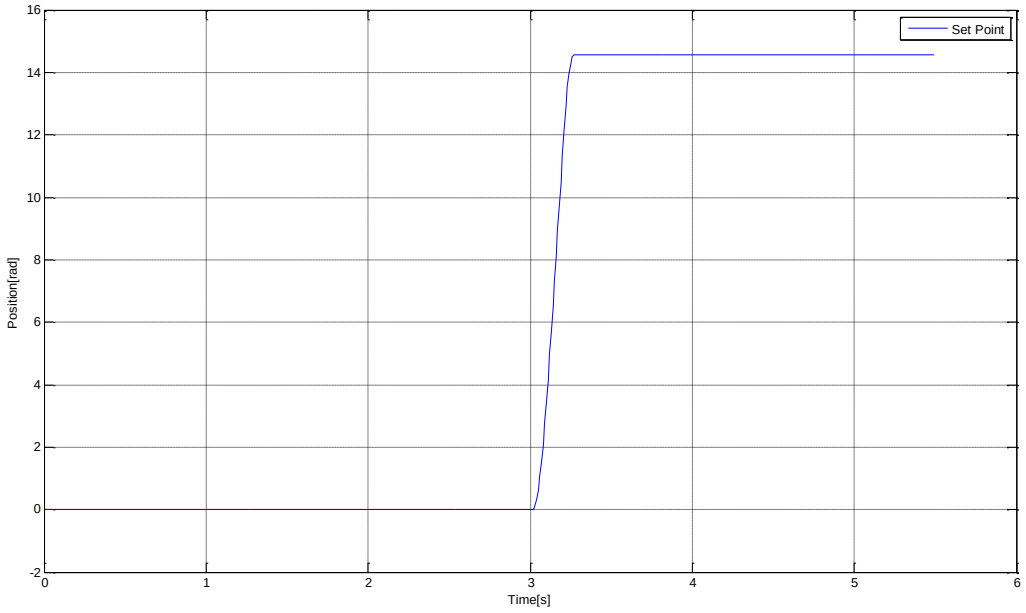


Figura IV.11 Señal real de respuesta del motor adquirida a partir de Borland C y RS-232

Ahora se pueden calcular las ganancias K_p, K_d y K_i, para esto debemos proponer un sobrepaso M_p y un tiempo de respuesta t_r. En nuestro caso elegimos un sobrepaso del 10% y un tiempo de respuesta de 0.15 s. Por último los parámetros que debemos encontrar son los siguientes:

donde

M_p Sobrepaso

ς Coeficiente de amortiguamiento

$$\zeta = \frac{\left|\ell n\left(\frac{M_p}{100}\right)\right|}{4\sqrt{\ell n^2\left(\frac{M_p}{100}\right)+\pi^2}}, \qquad \text{(IV.8)}$$

$$\omega_d = \frac{1}{t_r}\left(\pi - \tan^{-1}\left(\frac{\sqrt{1-\zeta^2}}{\zeta}\right)\right) \tag{IV.9}$$

y

$$\omega_n = \frac{\omega_d}{\sqrt{1-\zeta^2}} \tag{IV.10}$$

Si proponemos K_i=0, es decir un controlador PD, las fórmulas de K_p y K_d, se reducen a las siguientes.

$$K_p = \frac{\omega_n^2}{K} \tag{IV.12}$$

$$K_d = \frac{2\zeta\omega_n - a}{K} \tag{IV.13}$$

De esta manera se obtuvieron los valores K_p=89.9925 y K_d=3.1938.

IV.1.4 Etapa de potencia

Para la conexión del motor es necesario conocer las salidas que entran a la estructura BDC_PWM_DRV, para esto se impusieron dentro del *watch window* los valores mostrados en la siguiente gráfica (**Figura IV.12**).

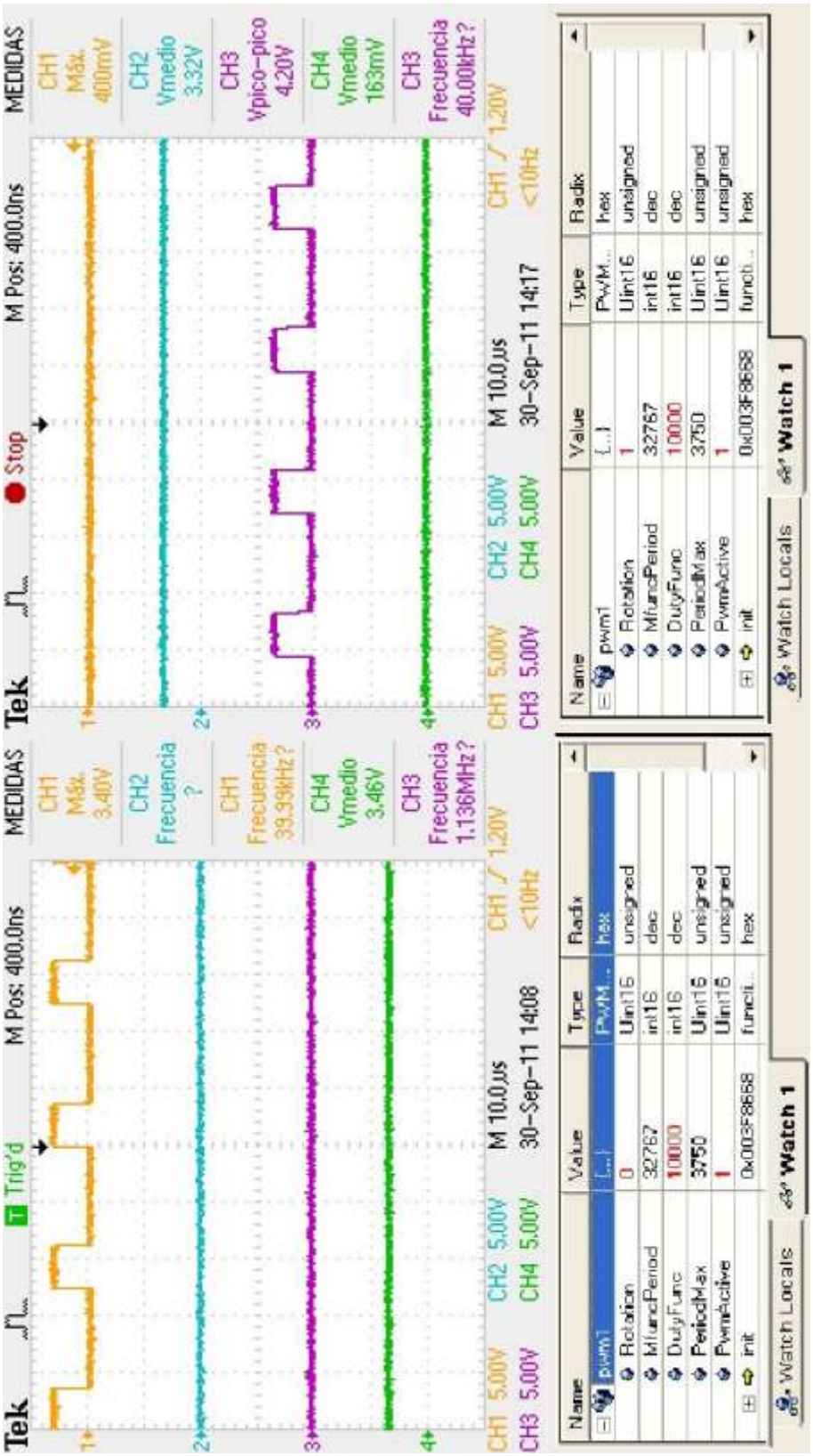


Figura IV.12 Impresión del osciloscopio TDS2024B con las señales de PWM correspondientes a los valores en el Watch Window

Debido al registro *Rotation* dentro de la estructura pwm1, sólo se utilizan las salidas PWM 1 y 4 cuando tiene valor 0, en el otro caso (cuando el valor es 1) se activan las salidas PWM 2 y 3.

Para utilizar estas señales PWM dentro del puente H (L298N, **Figura IV.13** y **Figura IV.14**) es necesario agregar una lógica de compuertas que mueva el motor a partir de las dos combinaciones de salidas (PWM1 y PWM4; PWM2 y PWM3).

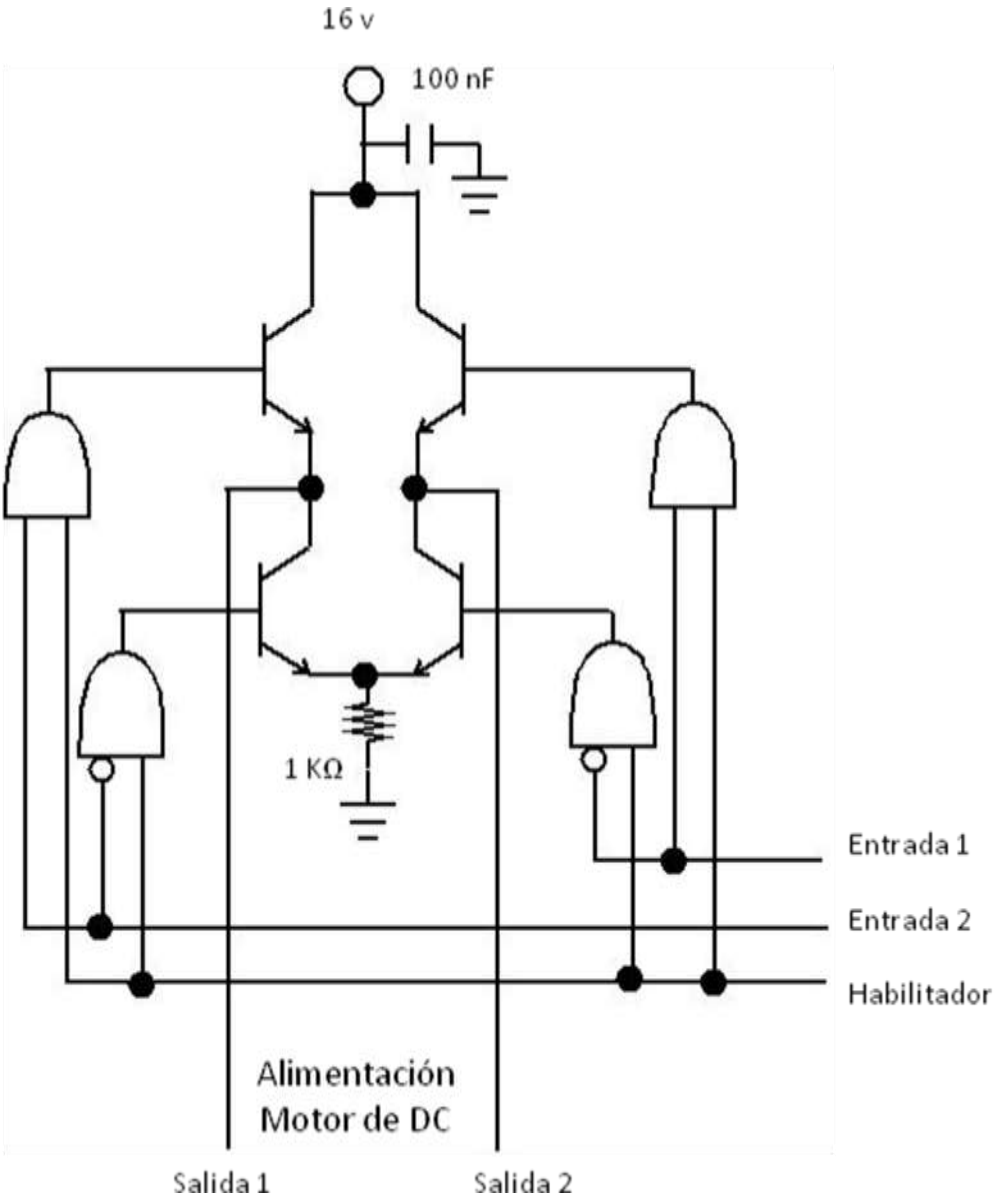


Figura IV.13 Circuito interno del L298N

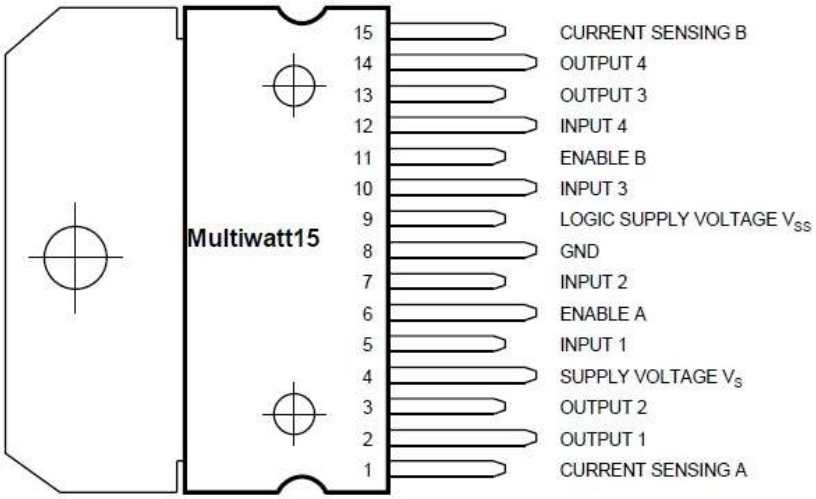


Figura IV.14 Fisiología del L298N (DataSheet, STMicroelectronics)

La solución para esta problemática está en el uso de una compuerta OR para cada una de las combinaciones, así se diseñó el siguiente arreglo lógico.

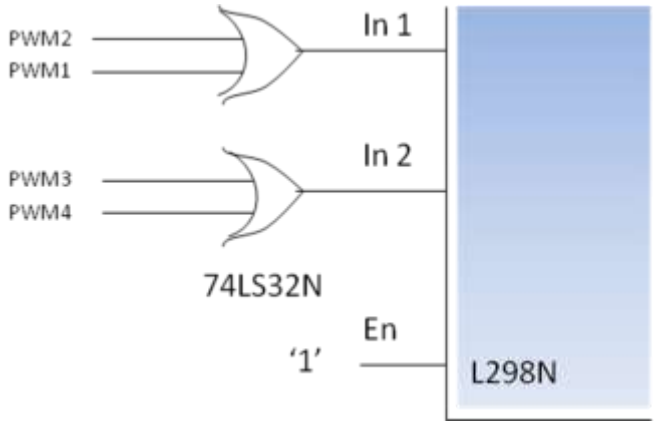


Figura IV.15 Uso de compuerta lógica OR para configuración de las salidas PWM

De esta manera se consigue la activación de un modo de giro con la amplitud que indique el PWM controlador (PWM1 y PWM3). Como se puede apreciar en la **Figura IV.12**, cuando se activan las salidas PWM 1 y 4, la salida número es la que contiene la señal cuadrática con el ancho de pulso necesaria, mientras que la salida 4 permanece en alto. De esta manera, con ayuda de la compuerta OR ordenamos las salidas de manera que cuando se active la salida cuatro y el

PWM 1 esté emitiendo pulsos, se genere un PWM en la entrada del motor que se active en bajo. De esta manera sólo hay que cambiar la función PwmActive en la estructura pwm1 para que sea activo en bajo.

IV.2 Conclusión

Como punto principal cabe destacar el ataque directo de los resultados hacia las premisas presentadas en hipótesis y objeticos, debido a que se desarrolló un sistema heterogéneo entre un DSP y un FPGA por medio de un modulo McBSP. Además, el FPGA cumplió como núcleo del sistema adquiriendo y facilitando acerca de la posición y velocidad del motor. Por otra parte, se implementó un algoritmo capaz de gestionar el valor de la velocidad con alta exactitud.

Los siguientes enunciados son afirmaciones concluidas a partir de la práctica que implicó el desarrollo del sistema presentado por este trabajo, sustentadas por la implementación y una repetitibilidad en los fenómenos.

- El controlador realizado dentro del DSP es capaz de monitorear la posición de la flecha del motor y llevarla hasta su referencia.
- A su vez, el SPEED_EST muestra una aproximación de la velocidad muy cercana a la obtenida por medio del algoritmo adaptivo y un tacómetro acoplado a la flecha.
- Las salidas PWM del DSP corresponden a la señal de salida obtenida por PID_REG3, llevando al motor hasta su referencia con ayuda del circuito L298N/74LS32N.
- Existe una íntegra compatibilidad entre DSP y FPGA para realizar interfaces McBSP,
- Es posible realizar una tarjeta híbrida por medio de la interfaz entre un DSP TMS320F2812 y un FPGA DE1, el desempeño mostrado es superior a la programación integral dentro del DSP y con un tiempo de desarrollo menor al necesario programando el controlador dentro del FPGA.

LITERATURA CITADA

Abdin Zain-ul, Svensson Bertil. 2008. Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing. Microprocessors and Microsystems 33 (2009), pp. 161-178.

Aguirre M. A., Tombs J.N., Baena-Lecuyer V., Mora J.L., Carrasco J.M., Torralba A., Franquelo L.G. 2004. Microprocessor and FPGA interfaces for in-system co-debugging in field programmable hybrid systems. Microprocessors and Microsystems 29, pp. 75–85.

Bhatti Pamela, Hannaford Blake. 1997. Single-chip velocity measurement system for incremental optical encoders. IEEE transactions on control systems technology 5. pp. 654-661.

Bolton W. 2001. Mecatrónica, Sistemas de control electrónico en ingeniería mecánica y electrónica. 2da. Edición. Alfa-Omega. pp. 292-300.

Buell D.A., Arnold J.M., Kleinfelder W.J. (Eds.), Splash 2: PFGAs in a custom computing machine. Wiley-IEEE Computer Society Press. 1996.

Chamberlain Roger D., Lancaster Joseph M., Cytron Ron K. 2008. Visions for application development on Hybrid computing systems. Parallel Computing 34, pp. 201-216.

Devrim Fidanci Osman, Poznanovic Dan, Gaj Kris, El-Ghazawi Tarek, Alexandridis Nikitas. 2003. Performance and overhead in a hybrid reconfigurable computer. Proceding of the international parallel and distributed processing sysmposium.0-7695-1926-1. pp. 1-8.

Dick Chris. 2008. CORDIC architectures for FPGA computing. Reconfigurable Computing 25 pp. 513-537.

El-Ghazawi T., Gaj K., Buell D., Kindratenko V., 2007. Reconfigurable Supercomputing. IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC07). Reno, NV. USA. Tutorial M04.

El-Ghazawi T., El-Araby E., Huang M., Gaj K., Kindratenko V., Buell D. 2008. The promise of high-performance reconfigurable computing. IEEE Computer 41. pp. 69-76.

Faes Philippe, Christiaens Mark, Stroobandt Dirk. 2007. Mobility of data in hybrid distributed computing systems. IEEE Computer Society. pp. 1-7.

Fang Wu, Yabin Wang, Liguang Chen, Jian Wang, Jinmei Lai, Yuan Wang, Jiarong Tong. 2009. Circuit design of a novel FPGA chip FDP2008. Journal of semiconductors 30. pp. 115009.1-115009.6.

Fernández Norberto, Fuentes Damaris, Sánchez Luis, Fisteus Jesús A. 2010. The NEWS ontology: Design and applications. Expert Systems with Applications 37 (12). pp. 8694-8704.

Franklin G.F, Powell D.J. and Workman M.L. 1990. Digital Control of Dynamic Systems. Addison-Wesley.

Galanis M. D., Milidonis A., Theodoridis G., Soudris D., Goutis C.E. 2006. Automated framework for partitioning DSP applications in hybrid reconfigurable platforms. Microprocessors and Microsystems 31. pp. 1-14.

Gao Shuli, Al-Khalili Dhamin, Chabini Noureddine. 2009. Efficient scheme for implementing large size signed multipliers using multigranular embedded DSP blocks in FPGA's. International Journal of Reconfigurable Computing 10.1155/2009/145130. pp. 1-11.

Gao Y. H., Tian X. L., Cheng P., Chang X., Dou W. J. 2006. Research of data acquisition and analysis systems for internal combustion engine based on DSP. Journal of Physics 48. pp. 671-645.

Götz Marcelo, Dittmann Florian, Xie Tao. 2008. Dynamic relocation of hybrid tasks: strategies and methodologies. Microprocessors and microsystems 33(2009). Pp. 81-89.

Guha Radha, Bagherzadeh Nader, Chou Pai. 2008. Resource management task partitioning and scheduling on a run-time reconfigurable embedded system. Computers and Electrical Engineering 35. pp. 258-285.

Hampel Volker, Sobe Peter, Maehle Erik. 2008. Experiences with a FPGA-based Reed/Solomon-encoding coprocessor. Microprocessor and Microsystems 32 (5). pp 313-320.

Hasan Ali T., Hamouda A. M. S., Ismail N., Al-Assadi H.M.A.A. 2006. An Adaptative-learning algorithm to solve the inverse kinematics problem of a 6 D.O.F. serial robot manipulator. Advances in engineering software 37. pp. 432-438.

Hartenstein R. 2001. A decade of reconfigurable computing: a visionary retrospective. Proceedings of ACM/IEEE design and test in Europe Conference (DATE). pp. 642-649.

Kastner R., Kaplan A., Memik S.O., Bozorgadeh E. 2002. Instruction generation for hybrid reconfigurable systems. ACM transaction on design automation for electronic systems 7 (4). pp. 605-627.

Koehler Seth, Curreri John, George Alan D. 2007. Performance analysis challenges and framework for high-performance reconfigurable computing. Parallel Computing. 24 (2008). pp. 217-230.

Kung Ying-Shieh, Shu Gua-Shieh. 2005. Development of a FPGA-based motion control IC for robot arm. International Conference on Industrial Technology. pp. 1-6.

Benkrid K, Benkrid A, Belkacemi S. 2007. Efficient FPGA hardware development: a multi-languaje approach. Journal of Systems Architecture 53 (4). pp. 184-209.

Lygouras J.N., Kodogiannis V., T Pachidis.P., Sirakoulis G.Ch. 2009. A new method for digital enconder adaptive velocity/aceleration evaluation using TDC with picosecond accuracy. Microprocessors and Microsystems 33. pp. 453-460.

Lygouras John N., Lalakos Konstantinos A., Tsalides Phillipos G. 1998. High-performance position detection and velocity adptive measurement for closed-loop position control. IEEE transactions on instrumentation and measurement 47. pp. 978-985.

Maguire L.P., McGinnity T.M., Glackin B., Ghani A., Belatreche A., Harkin J. 2007. Challenges for large-scale implementations of spiking neural networks on FPGAs. Neurocomputing 71. pp. 13-29.

McGrath Dylan. 2009. Mentor buys agility's C synthesis assets. EETimes. pp. 1.

Mignolet J.-Y., Nolllet V., Coene P., Verkest D., S Vernalde., Lauwareins R. 2003. Infrastructure for design and management of relocatable tasks is a heterogeneous reconfigurable system-on-chip. Proceedings of the conference on design, Automation and test in Europe (DATE).

Mignolet J.-Y., Vernalde S., Verkest D., Lauwereins R. 2002. Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances. International conference on engineering of reconfigurable systems and algorithms (ERSA). Las Vegas.

Ogata Katsuhiko. 2003. Ingeniería de Control Moderna. 4ta. Edición. Pearson/Prentice Hall. pp 681-745.

Ollero Baturone Aníbal. 2001. Robótica: manipuladores y robots móviles. 1ra. Edición. Marcombo. pp. 5-15.

Osornio-Ríos Roque Alfredo, Romero-Troncoso René de Jesús, Herrera-Ruiz Gilberto, Castañeda-Miranda Rodrigo. 2008. FPGA implementation of higher degree polynomial acceleration profiles for peak jerk reduction in servomotors. Robotics and Computer-Integrated Manufacturing 25. pp. 379-392.

Pellizzoni R., Caccamo M. 2005. Adaptive allocation of software and hardware real-time tasks for FPGA-based embedded systems. Proceedings of the 12thIEEE real-time and embedded technology and applications symposium (RTAS). IEEE Computer Society. Washington, DC. USA.

Pereira R. C., Combo A., Cruz N., Sousa Jorge, Correira C., Varandas C., Conroy S., Källne J. 2006. Enhanced neutron diagnostics data acquisition system based on a time digitizer and transient recorder hybrid module. Fusion Engineering and Design 81. pp. 1873-1877.

Pinto Alessandro, Carloni Luca P., Passerone Roberto, Sangiovanni-Vincentelli Alberto. 2006. Interchange Format for Hybrid Systems: Abstract Semantics. Computation and Control 3927. pp. 491-506.

Pinto Alessandro, Sangiovanni-Vincentelli Alberto, Carloni Luca P., Passerone Roberto. 2005. Interchange Formats for Hybrid Systems: Review and Proposal. Computation and Control 3414. pp. 526-541.

Plumlee Christopher L. 2008. Application development process for GNAT, a SOC networked system. Master's Thesis. University of New Hampshire. pp. 26-30.

Pranav Vaidya, Jaehwan. 2007. Simulation of hybrid computer architectures: simulators, methologies and recommendations. International conference on very large scale integration 978-1-4244-1710-0/07. pp. 157-162.

Proshanta Saha, Esam El-Araby, Miaoqing Huang, Mohamed Taher, Sergio Lopez-Buedo, Tarek El-Ghazawi, Chang Shu, Kris Gaj, Alan Michalski, Duncan Buell. 2007. Portable library development for reconfigurable computing systems: a case study. Parallel Computing 34(2008). pp. 245-260.

Rairán Danilo, Fonseca José. 2009. Algoritmos para la medición de posición y velocidad angular, a partir de un encoder incremental. II Congreso Internacional de Ingeniería Mecatrónica. Vol 1, No. 1. pp. 1-6.

Rauwerda G.K., Heyters P.M., Smit G.J.M. 2004. Mapping wireless communication algorithms onto a reconfigurable architecture. Journal of Supercomputing 30 (3). pp. 263-282.

Romero-Troncoso René de Jesús, Herrera-Ruiz Gilberto, Terol-Villalobos Iván, Jáuregui-Correa Juan Carlos. 2004. FPGA based on-line tool breakage detection system for CNC milling machines. MECHATRONICS 14. pp. 439-454.

Sancho-Pradel Dario L., Goodall Roger M. 2006. Targered processing of real-time embedded mechatronic systems. Control engineering practice 15. pp. 363-375.

Sawhney Divesh, Khera Priyanka, Keskar.A.G. 2008. Robotic arm with four degrees of freedom. First international conference on emerging trends in engineering and technology 10.1109/ICETET.200. pp. 802-806.

Schmidt Herman. 1963. An operational hybrid computing system provides analog-type computation with digital elements. IEEE transactions on electronic computers 12(6). pp. 715-732.

Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, Jarmo Takala. 2010. Handbook of Signal Processing Systems. 1st Edition. Springer. pp. 333-348.

Shuvra S. Battacharyya, Sundararajan Sriram, Edward A. Lee. 2000. Resynchronization for multiprocessors DSP systems. IEEE transactions on circuits and systems 47(11). pp. 1597-1609.

Simard Stéphane, Mailloux Jean-Gabriel, Beguenane Rachid. 2009. Prototyping advanced control systems on FPGA. Journal on embedded systems 10.1155/897023. pp. 1-12.

Takeo Miura, Junji Tsuda, Junzo Iwata. 1967. Hybrid computer solution of optimal control problems by the maximum principle. IEEE transactions on electronic computers 16(5). pp. 666-670.

Toliyat Hamid A., Campbell Steven. 2003. Dsp-based electromechanical motion control. Edit. CRC PRESS. pp. 147-162.

Upegui Andrés, Sánchez Eduardo. 2008. Envolvable FPGAs, Reconfigurable Computing (33). pp. 725-752.

Volder Jack E. 1959. The CORDIC trigonometric computing technique. Trans. Electron. Comput 8. pp. 330-334.

Wan M., Zhang H., George V., Benes M., Abnous A., Prabhu V., et al. 2001. Design methodology of a low-energy reconfigurable single-chip DSP system. Journal of very large scale integration signal processing 28 (1-2). pp. 47-61.

Yi J., Lilja D. 2006. Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations. IEEE trans. On computers 55 (3). pp. 268-280.

APÉNDICE

A. Artículo publicado en 7mo. Congreso Nacional de Ingeniería

Implementación en FPGA de algoritmo adaptativo para medición con alta precisión de velocidad en encoders.

FPGA implementation of adaptive algorithm for accurate measurement of speed in encoders.

Salvador Manuel Malagón Soldara, Edgar Alejandro Rivas Araiza

Posgrado de Ingeniería, Universidad Autónoma de Querétaro

chavamalagon@hotmail.com

RESUMEN. Este trabajo documenta la implementación de un algoritmo adaptativo que mide con gran precisión la velocidad rotacional en encoders. La implementación se efectuó dentro de un kit Cyclone II para aprovechar sus interfaces en la pruebas. A su vez contempla la interacción de algún procesador con el FPGA (Field Programmable Gate Array), desempeñando cada uno una parte del algoritmo. Éste se encuentra formado por dos bases de tiempo que se encargan de medir el periodo en la señal de cuadratura de un canal del encoder para ubicar el valor de la velocidad dentro de una tabla comparativa. Como resultado se obtiene un método de medición de velocidad con menor error y a la vez se aumenta la resolución del tranductor.

PALABRAS CLAVE. Medición de velocidad, FPGA, Encoder.

1. INTRODUCCIÓN.

La elección de este proyecto surge a través de la necesidad de tener técnicas precisas para la medición de velocidad en motores con encoders, particularmente en motores ubicados en articulaciones de brazos robóticos poli-articulados. Así, las operaciones de procesamiento serán designadas a la tarjeta de control del robot, por este motivo debemos visualizar que los procesadores empleados para el diseño de estas tarjetas son sistemas robustos tales como FPGAs, DSPs, FPOAs, GPUs, etc. De tal manera que se eligió un FPGA para aterrizar el algoritmo propuesto por Lygouras et al. (1998) hacia las actuales demandas tecnológicas en las que se exigen despreciables índices de error, altas velocidades de procesamiento y con transductores sencillos como los son los encoders.

El FPGA empleado fue un Cyclone II montado sobre su tarjeta de desarrollo, de esta manera se facilita la prueba, programación y obtención de resultados. Además para la flexibilidad y reproducibilidad del algoritmo se excluyeron operaciones tales como divisiones de la programación descrita dentro del FPGA. El encoder se eligió como transductor debido provee una fina resolución, omitiendo el problema que no brinda una posición absoluta, sin embargo con ayuda de este trabajo se podrá implementar sin el cuidado de obtener mediciones no aceptables o invariantes.

2. LECTURA DE POSICIÓN Y AMPLIACIÓN DE RESOLUCIÓN.

Un factor importante dentro de la instrumentación en motores se deriva de la precisión con la cual es medido el ángulo de rotación, la máxima resolución con un costo rasonable para un encoder, es de 0.09°. “Esta resolución es suficiente para muchas de las aplicaciones y movimiento en alta velocidad”, estable Lygouras, si tomamos en cuenta que la fórmula para calcular la resolución es P=360°/PPR del encoder y nuestro encoder tiene un total de cuentas de 1024, al hacer la operación obtenemos un resultado de 0.35156° lo cual representa poco más de un ¼ de la tasa de deseada.

Para aumentar la resolución del encoder se aplicará una sencilla solución, esta consitirá en multiplicar la lectura por cuatro, de esta manera se amplía el rango de trabajo con información que posteriormente podrá ser utilizada de acuerdo al rango de la tabla de valores propuesta por lygouras (Tabla 1).

Tabla 1. Relación Cuentas por periodo-Velocidad rotacional encontrada por Lygouras (1998).

Motor's Rotational Speed (rpm)	Encoder Pulse Period T (μs)	Period Measur. Counter's counts (8bits)	Decoder's output Preset Value (7bits)	Preset 1'sComplement Counted Encoder Pulses Cp	Sampling Interval Ts (Ts=Cp*T (μs))	Error (=15μs/Ts)
2929.7	20	00000000	0000000	127	2540	0.0059
..	..	..	..	..	..	..
122.07	$480_{(-)}$	00000001			60960	0.00024
122.07	$480_{(+)}$	00000010	1000000	63	30240	0.00049
..	..	..	..	..	..	..
61.03	$960_{(-)}$	00000011			60480	0.00024
61.03	$960_{(+)}$	00000100	1100000	31	29760	0.0005
..	..	..	..	..	..	..
30.52	$1920_{(-)}$	00000111			59520	0.00025
30.52	$1920_{(+)}$	00001000	1110000	15	28800	0.00052
..	..	..	..	..	..	..
15.258	$3840_{(-)}$	00001111			57600	0.00026
15.258	$3840_{(+)}$	00010000	1111000	7	26880	0.00055
..	..	..	..	..	..	..
7.629	$7680_{(-)}$	00011111			53760	0.00027
7.629	$7680_{(+)}$	00100000	1111100	3	23040	0.00065
..	..	..	..	..	..	..
3.814	$15360_{(-)}$	00111111			46080	0.00032
3.814	$15360_{(+)}$	01000000	1111110	1	15360	0.00097
..	..	..	..	..	..	..
0.957	61200	11111111			61200	0.00024

Adicionalmente se necesitará una forma de encontrar la dirección hacia la que va el eje del motor, de lo contrario sólo se podrá trabajar con la posición y la velocidad existente sólo hacia uno de los costados de la flecha del motor. De esta manera con el circuito de la figura 1 se puede conocer la dirección hacia la que se dirige el movimiento y así controlar un contador ascendente/descendente con ayuda de la máquina de estados correspondiente.

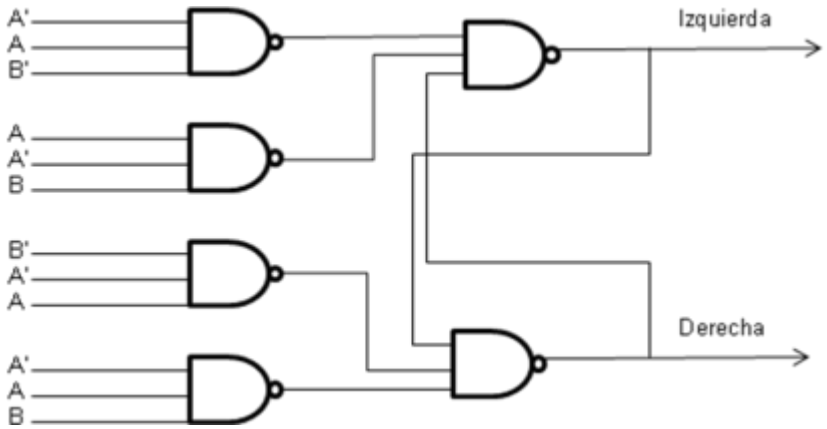


Figura 1. Circuito de detección de dirección.

A su vez obtenemos un vector a la salida con la información de la posición, para efectos de probar el motor a velocidades altas en este trabajo se almacena el valor de salida un vector de 32 bits (véase Figura 2).

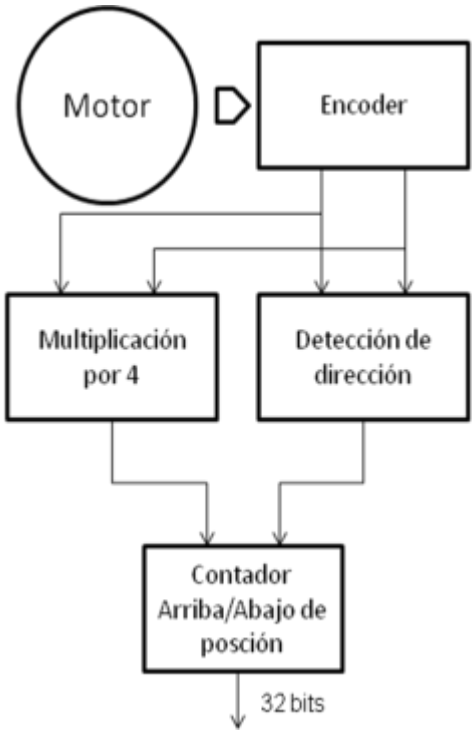


Figura 2. Diagrama a bloques del reconocimiento de dirección y posición.

3. ALGORITMO ADAPTIVO.

La solución al problema de la precisión en la lectura de encoders puede ser satisfacida de varias maneras, por ejemplo Hagiwara en 1992 establece que la velocidad se puede estimar por medio de la derivación de la posición actual y complementando a base de estimación, de esta manera queda a expensas de una simple diferenciación aritmética, sin embargo este método dispara el error al motor encontrarse rotando a velocidades muy bajas. El problema radica en que la resolución del encoder contiene saltos lo suficientemente separados para provocar un periodo muy largo entre cada pulso de cuadratura, es decir un punto ciego tan grande que la estimación de la velocidad se torna casi imposible de medir. Por otro lado tenemos propuestas como la de Bhatti et al. (2000) de medir con el periodo durante las velocidades bajas y cambiar el método en la velocidades altas procesando la freciencia. De aquí el conocimiento que existe una relación proporcional entre el periodo de la señal de cuadratura y el número de revoluciones para estimar la velocidad de manera adaptable.

Para el uso del algoritmo adaptativo implicó el tomar en cuenta el conocimiento desarrollado por John Lygouras, el cuál establece un conjunto de valores para los cuales se puede programar un contador ascendente y establecer un rango de velocidad rotacional dependiendo del periodo medido (Tabla 1).

El método consiste en el mesurado del periodo existente en la señal de cuadratura del encoder, de esta manera se puede estimar la velocidad del motor. Si además contamos el número de revoluciones Lygouras establece una relación entre estos dos datos (Ecuación 1) la cual nos puede entregar el valor de la velocidad rotacional.

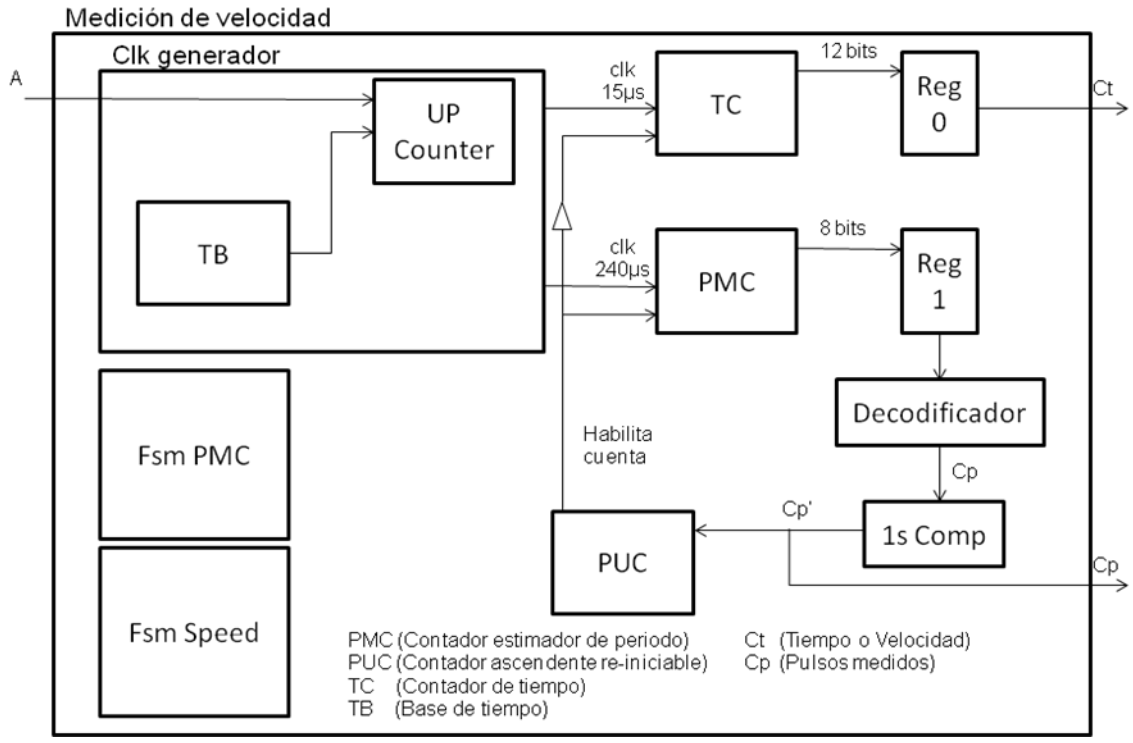


Figura. Diagrama a bloques de algoritmo adaptativo para medición de velocidad.

$$N_{[rpm]} = \frac{(Cp*3906.26)}{Ct} \quad (1)$$

4. RESULTADOS.

Como se puede observar en la tabla 1 existe un error diferente a ciertos rangos de velocidad, sin embargo se logra tener una tasa despreciable y además cumpliendo la expectativa de medir con exactitud semejante a velocidades bajas, sin importar la existencia de un encoder como transductor. Por otro lado la calidad de las mediciones puede ser mejorado aumentando la multiplicación en la resolución inicial de PPR del encoder.

Debido a la división contenida en la fórmula se deja a disposión el uso de Cp y Ct para ser procesados en algún dispositivo programable en C o alguna plataforma más amigable ya que una división en descripción de hardware implicaría el 80% del procesamiento total, y dejando un 20% de la programación al algoritmo adaptativo.

Para finalizar el algoritmo adaptativo provee características no existentes en el cálculo tradicional de la velocidad con encoders incrementales, entre las cuales destacan:

- Alta exactitud en la medición de la velocidad, aún en velocidades bajas.
- Gran velocidad procesamiento al encontrarse como descripción de hardware en un FPGA.
- Se gana flexibilidad en el sistema al proveer Cp, Ct y la posición en forma binaria disponible para cualquier dispositivo a conectar en el futuro.

5. REFERENCIAS.

N. Hagiwara, Y. Suzuki, H. Murase, “A method of improving the resolution and accuracy of rotary encoders using a code compensation technique”, IEEE transaction on insrumentation and measurement, 41:98-101

John N. Lygouras, Konstantinos A. Lalakos, Phillipos G. Tsalides, 1998, High-Performance position detection and velocity adaptative measurement for closed-loop position control, IEEE transactions on instrumentation and measurement, 47(4):978-985.

Pamela Bhatti, Blake Hannaford, 2000, Single chip velocity measurement system for incremental optical encoders, IEEE transactions on control systems technology, 5(6):654-662.

B. Póster publicado en 5to. Coloquio de Investigación (Postgrado, FI)

Universidad Autónoma de Querétaro
Facultad de Ingeniería
Divisón de Estudios de Posgrado
5to. Coloquio de Investigación

Sistema embebido de control de movimiento para robot poliarticulado

Embebed system of movement control for a poliarticulated robotic arm

S.M. Malagón-Soldara[a], E.A. Rivas-Araiza[b]
Facultad de Ingeniería de la Universidad Autónoma de Querétaro
[a]chavamalagon@hotmail.com, [b]erivas@uaq.mx

RESUMEN: El sistema híbrido de cómputo desarrollado en este trabajo contempla la interacción entre un DSP (Digital Signal Processor) y un FPGA (Field Programmable Gate Array) por medio de un protocolo de comunicación SPI (Serial Peripherial Interface), aprovechando las fortalezas de cada dispositivo como una unidad heterogénea. El sistema será implementado como una tarjeta de control con módulos de interacción externa para un brazo robótico tipo PUMA (Programmable Universal Machine for Assembly). Además, se aprovecha la velocidad de procesamiento y paralelismo del FPGA para implementar un algoritmo adaptable capaz de medir la velocidad con gran precisión obteniendo datos de un encoder incremental, a su vez el DSP quedará a disposición del usuario para implementar el control de posición del robot recibiendo los datos pertinentes del FPGA.

ABSTRACT: The hybrid computing system developed in this work involves the interaction between a DSP (Digital Signal Processor) and FPGA (Field Programmable Gate Array) using a protocol communication (SPI Serial Interface peripheral), exploiting the strengths of each device as heterogeneous unit. The system will be implemented as a control system with external interaction modules for a robot arm kind PUMA (Programmable Universal Machine for Assembly). In addition, it leverages the processing speed and parallelism of the FPGA to implement an adaptive algorithm capable of measuring the speed with great accuracy obtaining data from an incremental encoder, to turn the DSP will be available to the user to implement the robot position control receiving relevant data of the FPGA.

INTRODUCCIÓN

La elección de este proyecto surge a través de la necesidad de tener controladores fáciles de programar como lo son los DSPs para procesamiento de algoritmos de control de robots y un elemento que permita la interacción entre ellos (el FPGA), este último será utilizado como un servidor facilitando la información necesaria acerca del comportamiento del brazo robótico hacia todos los periféricos conectados, así, la unión entre dispositivos formará una tarjeta de control para el brazo robótico. Por otra parte a se implementará un método adaptativo en el FPGA desarrollado por Lygouras (1998) el cual permite la evaluación de la velocidad del motor con buena exactitud aún en velocidad muy bajas. El dispositivo propuesto además tiene la habilidad de conectar las 6 articulaciones del robot, cosa que un DSP es incapaz de hacer debido a su falta de módulos de cuadratura suficientes, sin embargo los DSPs posibilitan la adaptación del sistema a variadas aplicaciones, reduciendo su tiempo de programación debido a la simplicidad del lenguaje C.

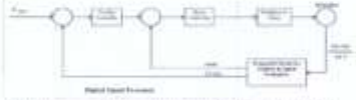
MATERIAL Y MÉTODOS

El DSP empleado es el TMS320F2812 el cual se encuentra en su presentación kit de desarrollo eZdsp2812 (Fig. 1), el FPGA utilizado es el FGG484AGQ0709 soldado sobre el kit Spartan-3A (Fig. 2). Este último será utilizado como un servidor facilitando la información necesaria acerca del comportamiento del brazo robótico hacia todos los periféricos conectados con él. Encima se aprovecha la cantidad de puertos que contiene un kit de desarrollo, ampliando el número de dispotivos que pueden acoplarse al sistema, incluyendo puertos importanes en la instrumentación como lo son el USB (Universal Serial Bus), el Ethernet, entre otros.

La conexión entre FPGA y el DSP se hará por medio del protocolo SPI debido a la velocidad que puede alcanzar (>10 Mhz), existe gran facilidad de programación debido a la delegación de tareas que efectúa el FPGA entre dispositivos más sencillos y con lenguajes de programción de nivel más alto (Fig. 3), adquiriendo la capacidad de conexión con dispositivos transitorios que pueden realizar tareas que contengan desde el controlador de movimiento hasta la instrumentación del robot.

El FPGA será dotado con la apritud de transformación entre señales de encoder incremental a señales de efecto Hall (Fig. 4.), esto a sabiendas del frecuente uso de este tipo de señales entre motores brushless.

El DSP lee la posición actual y la velocidad del eje del motor directamente en la interfaz del circuito, esto se compara con la posición deseada y velocidad, ejecuta el algoritmo de control y el controlador del motor (Fig. 5).



$$n = \frac{C_p}{C_t * m * T_c} \quad (1)$$

$$T_c = \frac{1}{f_{cl}}$$

n : número de marcas del encoders.
C_p: contador de pulsos.
C_t: número de pulsos de reloj medidos.
T_c : periodo

De (1) el contenido del contador de pulsos (C_p) es entonces el número de pulsos del encoder medidos, y el contenido de TC (C_t) es el número de pulsos de reloj medidos. El valor de C_p y C_t son usados para calcular la velocidad del motor, ya que C_p es la diferencia de posición angular, y C_t es el tiempo exacto de muestreo de cada medición. Así la velocidad rotación es:

$$n = \frac{\Delta\phi}{\Delta t} \quad \text{donde:} \quad \Delta\phi = \frac{C_p * 2 * \pi}{m}, \quad \Delta t = C_t * T_c$$

DISCUSIÓN Y TRABAJO FUTURO

El hardware de control híbrido propuesto en este trabajo ofrece grandes beneficios tales como flexibilidad, desempeño y mantenimiento. Por una parte el FPGA brinda una gran oportunidad en expansión de periféricos mientras que delega labores de procesamiento al DSP facilitando la programción al diseñador. Esta facilidad radica en el hecho de evitar descripción de hardware por una programación secuencial en C utilizada por el DSP. Otra aportación en flexibilidad es la adaptación en el sistema de medición en la posición angular de motores, transformando pulsos de encoder incremental a señales de efecto Hall y empleando un algoritmo adaptable para una evaluación más precisa de posición y velocidad aún en movimiento lentos. Para finalizar el mantenimiento y adaptación acorde a las diferentes aplicaciones se facilita debido a la manipulación de diferentes dispositivos sencillos formando un sistema heterogéneo.

REFERENCIAS

John N. Lygouras, A. Lalakos, Phillipos G. Tsalides, High-performance position detection and velocity adaptative measurement for closed-loop position control, IEEE transactinos on instrumentation and measurement, 47-4 (1998) pp. 978-985.

C. Código McBSP con loopback digital (McBSPLoopback.c)

```
#include "DSP281x_Device.h"
#include "DSP281x_SysCtrl.c"
#include "DSP281x_PieCtrl.c"
#include "DSP281x_PieVect.c"
#include "DSP281x_DefaultIsr.c"
#include "DSP281x_Gpio.c"
#include "DSP281x_MCbsp.c"
#include "DSP281x_Mcbsp.h"


//Define the frequency
//#define MCBSP_SRG_FREQ (150E8/4) //CPU/4

// Define the level of the FIFO 1-16
#define FIFO_LEVEL 16

// Choose a word size.  Uncomment one of the following lines
//#define WORD_SIZE    8    // Run a loopback test in 8-bit mode
#define WORD_SIZE 16      // Run a loopback test in 16-bit mode
//#define WORD_SIZE 32        // Run a loopback test in 32-bit mode


// Prototype statements for functions found within this file.

void mcbsp_xmit(int a, int b);
void mcbsp_fifo_init(void);
void error(void);

// Global data for this example
Uint16 sdata1 = 0x000;    // Sent Data
Uint16 rdata1 = 0x000;    // Recieved Data

Uint16 sdata2 = 0x000;    // Sent Data
Uint16 rdata2 = 0x000;    // Recieved Data

Uint16 rdata1_point;
Uint16 rdata2_point;


void main(void)
{

   Uint16 i;


// Step 1. Initialize System Control:
// PLL, WatchDog, enable Peripheral Clocks
// This example function is found in the DSP281x_SysCtrl.c file.
   InitSysCtrl();

// Step 2. Initalize GPIO:
// This example function is found in the DSP281x_Gpio.c file and
// illustrates how to set the GPIO to it's default state.
// InitGpio();  // Skipped for this example
// For this example, only enable the GPIO for McBSP

   InitMcbspGpio();                  // Select GPIOs to be McBSP pins
                              // Port F MUX - x011 1111 0000 0000

// Step 3. Clear all interrupts and initialize PIE vector table:
// Disable CPU interrupts
   DINT;

// Initialize PIE control registers to their default state.
// The default state is all PIE interrupts disabled and flags
// are cleared.
// This function is found in the DSP281x_PieCtrl.c file.
   InitPieCtrl();

// Disable CPU interrupts and clear all CPU interrupt flags:
   IER = 0x0000;
   IFR = 0x0000;

// Initialize the PIE vector table with pointers to the shell Interrupt
// Service Routines (ISR).
// This will populate the entire table, even if the interrupt
// is not used in this example.  This is useful for debug purposes.
```

```
// The shell ISR routines are found in DSP281x_DefaultIsr.c.
// This function is found in DSP281x_PieVect.c.
   InitPieVectTable();

// Step 4. Initialize all the Device Peripherals:
// This function is found in DSP281x_InitPeripherals.c
// InitPeripherals();     // Not required for this example
   mcbsp_fifo_init();     // Initialize the Mcbsp FIFO
   InitMcbsp();      // Initalize the Mcbsp in loopback test mode

// Step 5. User specific code, enable interrupts:

     if(WORD_SIZE == 8)            // Run a loopback test in 8-bit mode
     {
        InitMcbspa8bit();
        sdata2 = 0x0000;          // value is a don't care for 8-bit mode
        sdata1 = 0x0001;       //sdata1 = 0x0000; // 8-bit value to send
        rdata1_point = sdata1;
        while(1)
        {
          for(i = 1; i <= FIFO_LEVEL; i++)
          {
             mcbsp_xmit(sdata1,sdata2);
             sdata1++;
             sdata1 = sdata1 & 0x00FF;                    // Keep it to 8-bits
          }
          while(McbspaRegs.MFFRX.bit.RXFFST != FIFO_LEVEL ) { }  // Check for receive
          for(i = 1; i <= FIFO_LEVEL; i++)
          {
             rdata1 = McbspaRegs.DRR1.all;                // read DRR1
             if(rdata1 != rdata1_point) error();
             rdata1_point++;
             rdata1_point = rdata1_point & 0x00FF;        // Keep it to 8-bits
          }
          asm("    nop");                                 // Good place for a breakpoint
        }
      }


     if(WORD_SIZE == 16)           // Run a loopback test in 16-bit mode
     {
        InitMcbspa16bit();
        sdata2 = 0x0000;          // value is a don't care for 16-bit mode
        sdata1 = 0x0055;          // 16-bit value to send
        rdata1_point = sdata1;
        while(1)
        {
          for(i = 1; i <= FIFO_LEVEL; i++)
          {
             mcbsp_xmit(sdata1,sdata2);
             sdata1++;
          }
          while(McbspaRegs.MFFRX.bit.RXFFST != FIFO_LEVEL ) { }  // Check for receive
          for(i = 1; i <= FIFO_LEVEL; i++)
          {
             rdata1 = McbspaRegs.DRR1.all;                // read DRR1
             if(rdata1 != rdata1_point) error();
             rdata1_point++;
          }
          asm("    nop");                                 // Good place for a breakpoint
        }
      }

     /*
     if(WORD_SIZE == 32)           // Run a loopback test in 16-bit mode
     {
        InitMcbspa32bit();
        sdata1 = 0x0000;
        sdata2 = 0xFFFF;
        rdata1_point = sdata1;
        rdata2_point = sdata2;
        while(1)
        {
          for(i = 1; i <= FIFO_LEVEL; i++)
          {
             mcbsp_xmit(sdata1,sdata2);
             sdata1++;
             sdata2--;
          }
          while(McbspaRegs.MFFRX.bit.RXFFST != FIFO_LEVEL ) { }  // Check for receive
```

```
            for(i = 1; i <= FIFO_LEVEL; i++)
            {
               rdata2 = McbspaRegs.DRR2.all;
               rdata1 = McbspaRegs.DRR1.all;
               if(rdata1 != rdata1_point) error();
               if(rdata2 != rdata2_point) error();
               rdata1_point++;
               rdata2_point--;
            }
            asm("    nop");                          // Good place for a breakpoint

          }

      }
                */
}

// Some Useful local functions
void error(void)
{
   asm("     ESTOP0");  // test failed!! Stop!
   for (;;);
}


void mcbsp_xmit(int a, int b)
{
   McbspaRegs.DXR2.all=b;
   McbspaRegs.DXR1.all=a;
}

void mcbsp_fifo_init()
{

   McbspaRegs.MFFTX.all=0x0000;  //FIFO transmit register (pag 156)
       McbspaRegs.MFFRX.all=0x001F;  //FIFO recieve register (pag 157) valor                    McbspaRegs.MFFCT.all=0x0;     //FIFO ctrl
      register (pag 158) no cambia
   McbspaRegs.MFFINT.all=0x0; //FIFO interruput register (pag 158) ctrler interruptions, no here
   McbspaRegs.MFFST.all=0x0;     //FIFO status register. Detecta pulso de sincronia
   McbspaRegs.MFFTX.bit.MFFENA=1;         // Enable FIFO
   McbspaRegs.MFFTX.bit.TXFIFO_RESET=1;  // Enable Transmit channel
   McbspaRegs.MFFRX.bit.RXFIFO_RESET=1;  // Enable Receive channel


}
```

D. **Archivo de cabecera para McBSP con loopback digital (McBSPLoopback.h)**

```
#include "DSP281x_Device.h"     // DSP281x Headerfile Include File
#include "DSP281x_Examples.h"   // DSP281x Examples Include File


//---------------------------------------------------------------------------
// MCBSP_INIT_DELAY determines the amount of CPU cycles in the 2 sample rate
// generator (SRG) cycles required for the Mcbsp initialization routine.
// MCBSP_CLKG_DELAY determines the amount of CPU cycles in the 2 clock
// generator (CLKG) cycles required for the Mcbsp initialization routine.
// For the functions defined in Mcbsp.c, MCBSP_INIT_DELAY and MCBSP_CLKG_DELAY
// are based off of a 150 MHz SYSCLKOUT.
//---------------------------------------------------------------------------
#define CPU_SPD         150E6
#define MCBSP_SRG_FREQ   150E6/4    // SRG input is default LSPCLK for this example. (See note 1 at end)
#define MCBSP_SRG_PERIOD 1/MCBSP_SRG_FREQ

#define MCBSP_INIT_DELAY     2*(CPU_SPD/MCBSP_SRG_FREQ)                // # of CPU cycles in 2 SRG cycles-init delay
#define CLKGDV_VAL          0
#define MCBSP_CLKG_DELAY     2*(CPU_SPD/(MCBSP_SRG_FREQ/(1+CLKGDV_VAL))) // # of CPU cycles in 2 CLKG cycles-init delay

void delay_loop(void);                 // Delay function used for SRG initialization
void clkg_delay_loop(void);            // Delay function used for CLKG initialization
//---------------------------------------------------------------------------
```

```
// InitMcbsp: This function initializes the McBSP to a known state.
//---------------------------------------------------------------------------

void InitMcbspGpio(void)
{
   EALLOW;
   // Enable the GPIO pins for McBSP operation.
   GpioMuxRegs.GPFMUX.bit.MCLKRA_GPIOF9 = 1;
   GpioMuxRegs.GPFMUX.bit.MCLKXA_GPIOF8 = 1;
   GpioMuxRegs.GPFMUX.bit.MDRA_GPIOF13 = 1;
   GpioMuxRegs.GPFMUX.bit.MDXA_GPIOF12 = 1;
   GpioMuxRegs.GPFMUX.bit.MFSRA_GPIOF11 = 1;
   GpioMuxRegs.GPFMUX.bit.MFSXA_GPIOF10 = 1;
   EDIS;
}

void InitMcbsp(void)
{
   EALLOW;


// McBSP-A register settings

   McbspaRegs.SPCR2.all=0x0000;       // Reset FS generator, sample rate generator & transmitter
   McbspaRegs.SPCR1.all=0x0000;       // Reset Receiver, Right justify word
   McbspaRegs.SPCR1.bit.DLB = 1;      // Enable dig.loopback mode. Comment out for non-DLB, normal transfer mode.

   McbspaRegs.MFFINT.all=0x0;         // Disable all interrupts
   McbspaRegs.MFFST.all=0x0;          // Clear all status bits

   McbspaRegs.RCR2.all=0x0;           // Single-phase frame, 1 word/frame, No companding  (Receive)
   McbspaRegs.RCR1.all=0x0;

   McbspaRegs.XCR2.all=0x0;           // Single-phase frame, 1 word/frame, No companding  (Transmit)
   McbspaRegs.XCR1.all=0x0;

   McbspaRegs.SRGR2.bit.CLKSM = 1;    // CLKSM=1 (If SCLKME=0, i/p clock to SRG is LSPCLK)
   McbspaRegs.SRGR2.bit.FPER = 31;    // FPER = 32 CLKG periods

   McbspaRegs.SRGR1.bit.FWID = 0;     // Frame Width = 1 CLKG period
   McbspaRegs.SRGR1.bit.CLKGDV = CLKGDV_VAL;   // CLKG frequency = LSPCLK/(CLKGDV+1)

   McbspaRegs.PCR.bit.FSXM = 1;       // FSX generated internally, FSR derived from an external source
   McbspaRegs.PCR.bit.CLKXM = 1;      // CLKX generated internally, CLKR derived from an external source
   delay_loop();                      // Wait at least 2 SRG clock cycles
   McbspaRegs.SPCR2.bit.GRST=1;       // Enable the sample rate generator
   clkg_delay_loop();                 // Wait at least 2 CLKG cycles
   McbspaRegs.SPCR2.bit.XRST=1;       // Release TX from Reset
   McbspaRegs.SPCR1.bit.RRST=1;       // Release RX from Reset
   McbspaRegs.SPCR2.bit.FRST=1;       // Frame Sync Generator reset

}

void InitMcbspa8bit(void)
{
   McbspaRegs.RCR1.bit.RWDLEN1=0;     // 8-bit word
   McbspaRegs.XCR1.bit.XWDLEN1=0;     // 8-bit word
}

void InitMcbspa12bit(void)
{
   McbspaRegs.RCR1.bit.RWDLEN1=1;     // 12-bit word
   McbspaRegs.XCR1.bit.XWDLEN1=1;     // 12-bit word
}

void InitMcbspa16bit(void)
{
   McbspaRegs.RCR1.bit.RWDLEN1=2;     // 16-bit word
   McbspaRegs.XCR1.bit.XWDLEN1=2;     // 16-bit word
}

void InitMcbspa20bit(void)
{
   McbspaRegs.RCR1.bit.RWDLEN1=3;     // 20-bit word
   McbspaRegs.XCR1.bit.XWDLEN1=3;     // 20-bit word
}

void InitMcbspa24bit(void)
{
   McbspaRegs.RCR1.bit.RWDLEN1=4;     // 24-bit word
```

```
   McbspaRegs.XCR1.bit.XWDLEN1=4;     // 24-bit word
}

void InitMcbspa32bit(void)
{
   McbspaRegs.RCR1.bit.RWDLEN1=5;     // 32-bit word
   McbspaRegs.XCR1.bit.XWDLEN1=5;     // 32-bit word
}

void delay_loop(void)                 // Delay function used while initializing the registers
{
   long      i;
   for (i = 0; i < MCBSP_INIT_DELAY; i++) {}
}

void clkg_delay_loop(void)
{
   long      i;
   for (i = 0; i < MCBSP_CLKG_DELAY; i++) {} //delay in McBsp init. must be at least 2 SRG cycles
}
```

E. **Código McBSP en FPGA (McBSP.vhd)**

```
library IEEE;
use IEEE.std_logic_1164.all;

entity McBSP is
        port(                                   --                  _____________
        RST   : in  std_logic;                  --        St --|                  |-- EOT
        CLKin : in  std_logic;                  --             |   Rx   y   Tx   |
        Fs    : in  std_logic;                    --  Datin --|                  |-- Datout
        Rx    : in  std_logic;--                            |                   |
        Tx    : out std_logic;--                            |___________|
        Fsx   : out std_logic;
        CLKout: out std_logic;
        EOT   : out std_logic                   --End of Reception
        );
end entity;

architecture simple of McBSP is

component TransmisorMcBSP is  --A 115200 baudios
        port(
        RST   : in  std_logic;
        CLKit : in  std_logic;
        St    : in  std_logic;
        Dato  : in  std_logic_vector(15 downto 0);
        CLKot : out std_logic;
        Fsx   : out std_logic;
        Tx              : out std_logic;
        EOT   : out std_logic
        );
end component;

component ReceptorMcBSP is
        port(
        RST : in  std_logic;
        CLK : in  std_logic;
        Rx  : in  std_logic;
        Fs  : in  std_logic;
        EOR : out std_logic;
        Dato: out std_logic_vector(15 downto 0)
        );
end component;

--Declaracion de seniales
signal EOR  : std_logic;
signal Datin,Datout   : std_logic_vector(15 downto 0);

begin

        Datout <= Datin;
        U01: TransmisorMcBSP port map (RST,CLKin,EOR,Datin,CLKout,Fsx,Tx,EOT); --CLKout,Fsx,Tx,EOT);
        U02: ReceptorMcBSP    port map (RST,CLKin,Rx,Fs,EOR,Datin);
end architecture;
```

F. Código PID para control de posición de motor en DSP (PID_Tesis.c)

```
#include "target.h"
#include "DSP281x_Device.h"

#include "IQmathLib.h"
#include "dcmotor.h"
#include "parameter.h"
#include "build.h"
#include <math.h>

// Prototype statements for functions found within this file.
interrupt void MainISR(void);
interrupt void QepISR(void);

// Global variables used in this system
float32 SpeedRef = 0.2;          // speed reference (pu)
float32 SpeedRefRpm;             // speed reference (rpm)
float32 T = 0.001/ISR_FREQUENCY; // Samping period (sec), see parameter.h
float32 PositionRef = 0.5;       // Position reference (pu)

Uint16 IsrTicker = 0;
Uint16 BackTicker = 0;

int16 DlogCh1 = 0;
int16 DlogCh2 = 0;
int16 DlogCh3 = 0;
int16 DlogCh4 = 0;

volatile Uint16 EnableFlag = FALSE;

Uint16 SpeedLoopPrescaler = 50;      // speed loop prescaler
Uint16 SpeedLoopCount = 1;           // speed loop counter

Uint16 PositionLoopPrescaler = 1;    // Position loop prescaler
Uint16 PositionLoopCount = 1;        // Position loop counter

_iq SpeedRefIq;

// Instance PID regulators to regulate the speed and position
PIDREG3 pid1_spd = PIDREG3_DEFAULTS;
PIDREG3 pid1_pos = PIDREG3_DEFAULTS;

// Instance a PWM driver instance
PWMGEN pwm1 = PWMGEN_DEFAULTS;

// Instance a QEP driver instance
QEP qep1 = QEP_DEFAULTS;

// Instance a speed calculator based on QEP without index pulse
SPEED_ESTIMATION speed1 = SPEED_ESTIMATION_DEFAULTS;

// Instance a ramp controller to smoothly ramp the frequency
RMPCNTL rc1 = RMPCNTL_DEFAULTS;

// Create an instance of DATALOG Module
DLOG_4CH dlog = DLOG_4CH_DEFAULTS;


void main(void)
{


// ***********************************************
// Initialization code for DSP_TARGET = F2812
// ***********************************************
#if (DSP_TARGET==F2812)

// Initialize System Control registers, PLL, WatchDog, Clocks to default state:
      // This function is found in the DSP281x_SysCtrl.c file.
            InitSysCtrl();

// HISPCP prescale register settings, normally it will be set to default values
   EALLOW;   // This is needed to write to EALLOW protected registers
   SysCtrlRegs.HISPCP.all = 0x0000;     // SYSCLKOUT/1
```

```
   EDIS;   // This is needed to disable write to EALLOW protected registers

// Disable and clear all CPU interrupts:
                DINT;
                IER = 0x0000;
                IFR = 0x0000;

// Initialize Pie Control Registers To Default State:
        // This function is found in the DSP281x_PieCtrl.c file.
                InitPieCtrl();

// Initialize the PIE Vector Table To a Known State:
        // This function is found in DSP281x_PieVect.c.
                // This function populates the PIE vector table with pointers
        // to the shell ISR functions found in DSP281x_DefaultIsr.c.
                InitPieVectTable();

// User specific functions, Reassign vectors (optional), Enable Interrupts:

// Initialize EVA Timer 1:
    // Setup Timer 1 Registers (EV A)
    EvaRegs.GPTCONA.all = 0;

   // Waiting for enable flag set
   while (EnableFlag==FALSE)
    {
      BackTicker++;
    }

// Enable Underflow interrupt bits for GP timer 1
    EvaRegs.EVAIMRA.bit.T1UFINT = 1;
    EvaRegs.EVAIFRA.bit.T1UFINT = 1;

// Enable Period interrupt bits for GP timer 2
    EvaRegs.EVAIMRB.bit.T2PINT = 1;
    EvaRegs.EVAIMRB.bit.T2PINT = 1;

// Reassign ISRs.
        // Reassign the PIE vector for T1UFINT and T2PINT to point to a different
        // ISR then the shell routine found in DSP281x_DefaultIsr.c.
        // This is done if the user does not want to use the shell ISR routine
        // but instead wants to use their own ISR.

                EALLOW;  // This is needed to write to EALLOW protected registers
                PieVectTable.T1UFINT = &MainISR;
                PieVectTable.T2PINT = &QepISR;
                EDIS;   // This is needed to disable write to EALLOW protected registers

// Enable PIE group 2 interrupt 6 for T1UFINT
    PieCtrlRegs.PIEIER2.all = M_INT6;

// Enable PIE group 3 interrupt 1 for T2PINT
    PieCtrlRegs.PIEIER3.all = M_INT1;

// Enable CPU INT2 for T1UFINT and INT3 for T2PINT:
                IER |= (M_INT2 | M_INT3);

#endif

// Initialize PWM module
    pwm1.PeriodMax = (SYSTEM_FREQUENCY*1000000/1)*T;  // Perscaler X1 (T1), ISR period = T x 1, Asymmetric PWM
                pwm1.init(&pwm1);

// Initialize DATALOG module
    dlog.iptr1 = &DlogCh1;
    dlog.iptr2 = &DlogCh2;
    dlog.iptr3 = &DlogCh3;
    dlog.iptr4 = &DlogCh4;
    dlog.trig_value = 0x4000;
    dlog.size = 0x400;
    dlog.prescalar = 1000;
    dlog.init(&dlog);

// Initialize QEP module
    qep1.LineEncoder = 512;
    qep1.MechScaler = _IQ30(0.25/qep1.LineEncoder);
    qep1.PreScaler = 249;
    qep1.OutputMechScaler = _IQ30((0.25/qep1.PreScaler)*(1.0/qep1.LineEncoder));
    qep1.init(&qep1);
```

```
// Initialize RMP_CNTL module
   rc1.RampDelayMax = 2;

// Initialize the speed module QEP based speed calculation
   speed1.K1 = _IQ21(1/(BASE_FREQ*T));
   speed1.K2 = _IQ(1/(1+T*2*PI*30));  // Low-pass cut-off frequency
   speed1.K3 = _IQ(1)-speed1.K2;
   speed1.BaseRpm = BASE_SPEED;

// Initialize the PID_REG3 module for speed control
   pid1_spd.Kp = _IQ(1.5);
               pid1_spd.Ki = _IQ(T*SpeedLoopPrescaler/0.005);
               pid1_spd.Kd = _IQ(0/(T*SpeedLoopPrescaler));
               pid1_spd.Kc = _IQ(0.5);
   pid1_spd.OutMax = _IQ(0.8);
   pid1_spd.OutMin = _IQ(0.02);

// Initialize the PID_REG3 module for position control
   pid1_pos.Kp = _IQ(5);
               pid1_pos.Ki = _IQ(T*SpeedLoopPrescaler*PositionLoopPrescaler/2);
               pid1_pos.Kd = _IQ(0/(T*SpeedLoopPrescaler*PositionLoopPrescaler));
               pid1_pos.Kc = _IQ(0.5);
   pid1_pos.OutMax = _IQ(0.8);
   pid1_pos.OutMin = _IQ(-0.8);

// Enable global Interrupts and higher priority real-time debug events:
               EINT;   // Enable Global interrupt INTM
               ERTM;       // Enable Global realtime interrupt DBGM

// IDLE loop. Just sit and loop forever:
               for(;;) BackTicker++;

}

interrupt void MainISR(void)
{

// Verifying the ISR
   IsrTicker++;

// ***************** LEVEL1 *****************
#if (BUILDLEVEL==LEVEL1)

// ---------------------------------------------------------------------------
//    Call the PWM signal generation update function.
// ---------------------------------------------------------------------------
     pwm1.update(&pwm1);

// ---------------------------------------------------------------------------
//    Call the QEP calculation function.
// ---------------------------------------------------------------------------
     qep1.calc(&qep1);

// ---------------------------------------------------------------------------
//    Connect inputs of the speed module and call the speed
//    calculation function.
// ---------------------------------------------------------------------------
   #if (DSP_TARGET==F2812)
      speed1.EstimatedTheta = _IQ15toIQ((int32)qep1.MechTheta);
      speed1.calc(&speed1);
   #endif

// ---------------------------------------------------------------------------
//    Connect inputs of the DATALOG module
// ---------------------------------------------------------------------------
   #if (DSP_TARGET==F2812)
     DlogCh1 = qep1.MechTheta;
     DlogCh2 = qep1.OutputTheta;
   #endif
     DlogCh3 = (int16)_IQtoIQ15(speed1.EstimatedSpeed);
     DlogCh4 = pwm1.DutyFunc;

#endif // (BUILDLEVEL==LEVEL1)


// ***************** LEVEL2 *****************
#if (BUILDLEVEL==LEVEL2)

// ---------------------------------------------------------------------------
//    Connect inputs of the RMP_CNTL module and call the Ramp control
```

```
//    calculation function.
// ---------------------------------------------------------------------------
      rc1.TargetValue = _IQ(SpeedRef);
      rc1.calc(&rc1);

// ---------------------------------------------------------------------------
//    Connect input speed reference from ramp control output
// ---------------------------------------------------------------------------
      SpeedRefIq = rc1.SetpointValue;

// ---------------------------------------------------------------------------
//    Connect inputs of the PID_REG3 module and call the PID controller
//    calculation function.
// ---------------------------------------------------------------------------
     if (SpeedLoopCount==SpeedLoopPrescaler)
      {
        pid1_spd.Ref = _IQabs(SpeedRefIq);
        pid1_spd.Fdb = _IQabs(speed1.EstimatedSpeed);
                    pid1_spd.calc(&pid1_spd);

        SpeedLoopCount=1;
      }
      else SpeedLoopCount++;

// ---------------------------------------------------------------------------
//    Connect inputs of the PWM_DRV module and call the PWM signal generation
//    update function.
// ---------------------------------------------------------------------------
       if (SpeedRefIq > _IQ(0))
        pwm1.Rotation = 0;     // Positive speed
      else
        pwm1.Rotation = 1;     // Negative speed

      pwm1.DutyFunc = (int16)_IQtoIQ15(_IQabs(pid1_spd.Out));  // DutyFunc is in Q15
                  pwm1.update(&pwm1);

// ---------------------------------------------------------------------------
//    Call the QEP calculation function.
// ---------------------------------------------------------------------------
      qep1.calc(&qep1);

// ---------------------------------------------------------------------------
//    Connect inputs of the speed module and call the speed
//    calculation function.
// ---------------------------------------------------------------------------
    #if (DSP_TARGET==F2812)
       speed1.EstimatedTheta = _IQ15toIQ((int32)qep1.MechTheta);
       speed1.calc(&speed1);
    #endif

// ---------------------------------------------------------------------------
//    Calculate the speed reference (rpm) from the SpeedRefIq.
// ---------------------------------------------------------------------------
      SpeedRefRpm = (float32)(speed1.BaseRpm*_IQtoF(SpeedRefIq));

// ---------------------------------------------------------------------------
//    Connect inputs of the DATALOG module
// ---------------------------------------------------------------------------
    #if (DSP_TARGET==F2812)
      DlogCh1 = qep1.MechTheta;
      DlogCh2 = qep1.OutputTheta;
    #endif
      DlogCh3 = _IQtoIQ15(pid1_spd.Ref);
      DlogCh4 = _IQtoIQ15(pid1_spd.Fdb);

#endif // (BUILDLEVEL==LEVEL2)


// ***************** LEVEL3 *****************
#if (BUILDLEVEL==LEVEL3)

// ---------------------------------------------------------------------------
//    Connect inputs of the RMP_CNTL module and call the Ramp control
//    calculation function.
// ---------------------------------------------------------------------------
      rc1.TargetValue = _IQ(PositionRef);
      rc1.calc(&rc1);

// ---------------------------------------------------------------------------
//    Connect inputs of the PID_REG3 modules and call the PID controller
```

```
//    calculation function.
// ---------------------------------------------------------------------------
      if (SpeedLoopCount==SpeedLoopPrescaler)
      {

        if (PositionLoopCount==PositionLoopPrescaler)
        {
//          pid1_pos.Ref = _IQ(PositionRef);
          pid1_pos.Ref = rc1.SetpointValue;
        #if (DSP_TARGET==F2808)
          pid1_pos.Fdb = _IQ24toIQ((int32)qep1.OutputTheta);
        #endif
        #if (DSP_TARGET==F2812)
          pid1_pos.Fdb = _IQ15toIQ((int32)qep1.OutputTheta);
        #endif
                    pid1_pos.calc(&pid1_pos);

          PositionLoopCount=1;
        }
        else PositionLoopCount++;

        SpeedRefIq = pid1_pos.Out;
        pid1_spd.Ref = _IQabs(SpeedRefIq);
        pid1_spd.Fdb = _IQabs(speed1.EstimatedSpeed);
                  pid1_spd.calc(&pid1_spd);

        SpeedLoopCount=1;
      }
      else SpeedLoopCount++;

// ---------------------------------------------------------------------------
//    Connect inputs of the PWM_DRV module and call the PWM signal generation
//    update function.
// ---------------------------------------------------------------------------
      if (SpeedRefIq > _IQ(0))
        pwm1.Rotation = 0;     // Positive speed
      else
        pwm1.Rotation = 1;     // Negative speed

      pwm1.DutyFunc = (int16)_IQtoIQ15(_IQabs(pid1_spd.Out));  // DutyFunc is in Q15
                pwm1.update(&pwm1);

// ---------------------------------------------------------------------------
//    Call the QEP calculation function.
// ---------------------------------------------------------------------------
      qep1.calc(&qep1);

// ---------------------------------------------------------------------------
//    Connect inputs of the speed module and call the speed
//    calculation function.
// ---------------------------------------------------------------------------
    #if (DSP_TARGET==F2808)
       speed1.EstimatedTheta = _IQ24toIQ((int32)qep1.MechTheta);
       speed1.calc(&speed1);
    #endif
    #if (DSP_TARGET==F2812)
       speed1.EstimatedTheta = _IQ15toIQ((int32)qep1.MechTheta);
       speed1.calc(&speed1);
    #endif

// ---------------------------------------------------------------------------
//    Calculate the speed reference (pu and rpm) from the speedRefIq.
// ---------------------------------------------------------------------------
      SpeedRef = _IQtoF(SpeedRefIq);
      SpeedRefRpm = (float32)(speed1.BaseRpm*SpeedRef);

// ---------------------------------------------------------------------------
//    Connect inputs of the DATALOG module
// ---------------------------------------------------------------------------
    #if (DSP_TARGET==F2808)
      DlogCh1 = (int16)_IQtoIQ15(_IQ24toIQ(qep1.MechTheta));
      DlogCh2 = (int16)_IQtoIQ15(_IQ24toIQ(qep1.OutputTheta));
    #endif
    #if (DSP_TARGET==F2812)
      DlogCh1 = qep1.MechTheta;
      DlogCh2 = qep1.OutputTheta;
    #endif
      DlogCh3 = _IQtoIQ15(pid1_pos.Ref);
      DlogCh4 = _IQtoIQ15(pid1_pos.Fdb);
```

```
#endif // (BUILDLEVEL==LEVEL3)


// ---------------------------------------------------------------------------
//    Call the DATALOG update function.
// ---------------------------------------------------------------------------
   dlog.update(&dlog);


#if (DSP_TARGET==F2812)
interrupt void QepISR(void)
{

// ---------------------------------------------------------------------------
//    Call the QEP_DRV isr function.
// ---------------------------------------------------------------------------
  qep1.isr(&qep1);

// Enable more interrupts from this timer
        EvaRegs.EVAIMRB.bit.T2PINT = 1;

// Note: To be safe, use a mask value to write to the entire
        // EVAIFRB register.  Writing to one bit will cause a read-modify-write
        // operation that may have the result of writing 1's to clear
        // bits other then those intended.
   EvaRegs.EVAIFRB.all = BIT0;

// Acknowledge interrupt to recieve more interrupts from PIE group 3
        PieCtrlRegs.PIEACK.all |= PIEACK_GROUP3;

}
#endif
```

G. **Código de Algoritmo adaptivo de velocidad (Speed.vhd)**

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

entity speed is
        port(
                rst: in std_logic;
                clk: in std_logic;
                ----------------------
                cha: in std_logic;
                ----------------------
                cp:  out std_logic_vector(6 downto 0);
                ct:          out std_logic_vector(11 downto 0);
                ----------------------
                speed0:     out std_logic;
                speedDn: out std_logic
        );
end speed;

architecture estructural of speed is

component fsm_pmc
        port(
                rst:   in std_logic;
                clk:   in std_logic;
                ------------------------------------------
                ys:     in std_logic;
                puc_dn: in std_logic;
                ------------------------------------------
                eneabler:   out std_logic;
                fsm_pmc_dn: out std_logic
        );
end component;


component fsm_speed
        port(
                rst:   in std_logic;
                clk:   in std_logic;
                ----------------------
                fsm_pmc_dn: in std_logic;
                puc_dn:     in std_logic;
                ----------------------
```

```
                clr_pmc: out std_logic;
                ld_puc:  out std_logic;
                clr_tc:  out std_logic;
                ld_reg0: out std_logic;
                ld_reg1: out std_logic;
                rdy:     out std_logic
        );
end component;

component clk_gen
        port(
                rst: in std_logic;
                clk: in std_logic;
                ---------------------
                TB240us: out std_logic;
                TB15us:  out std_logic
        );
end component;

component PulsoUnCiclo
port(
        rst:    in  std_logic;
        clk:    in  std_logic;
        x:      in  std_logic;
        xspulse: out std_logic
);
end component;

component PMC
        port(
                rst: in  std_logic;
                clk: in  std_logic;
                ---------------------
                en:  in  std_logic;
                clr: in  std_logic;
                ---------------------
                count: out std_logic_vector( 7 downto 0)
        );
end component;

component PUC
        port(
                rst: in  std_logic;
                clk: in  std_logic;
                ---------------------
                k:   in  std_logic_vector(6 downto 0);
                ---------------------
                en:  in  std_logic;
                ld:  in  std_logic;
                ---------------------
                dn: out std_logic
        );
end component;

component TC
        port(
                rst: in  std_logic;
                clk: in  std_logic;
                ---------------------
                en:  in  std_logic;
                clr: in  std_logic;
                ---------------------
                count:    out std_logic_vector( 11 downto 0);
                overflow: out std_logic
        );
end component;

component reg_hld_ld_n
        generic     (
                n: integer :=8
        );
        port(
                rst:  in  std_logic;
                clk:  in  std_logic;
                opc:  in  std_logic;
                D:    in  std_logic_vector(n-1 downto 0);
                Q:    out std_logic_vector(n-1 downto 0)
        );
end component;
```

```
component reg_hld_ld_n_n
		generic	(
			n: integer :=8
		);
		port(
			rst:  in  std_logic;
			clk:  in  std_logic;
			opc:  in  std_logic;
			D:    in  std_logic_vector(n-1 downto 0);
			Q:    out std_logic_vector(n-1 downto 0)
		);
end component;

component decoder
		port(
			X: in  std_logic_vector(7 downto 0);
			Y: out std_logic_vector(6 downto 0)
		);
end component;

component comp1
		generic(
			n: integer :=8
		);
		port(
			X:     in  std_logic_vector(n-1 downto 0);
			Xcomp: out std_logic_vector(n-1 downto 0)
		);
end component;


--seniales

--xsp
signal s_ys: std_logic;

--clk_gen
signal s_240us,s_15us: std_logic;



--PMC
signal s_pmc_count: std_logic_vector(7 downto 0);

--PUC
signal s_puc_dn: std_logic;

--TC
signal s_tc_count: std_logic_vector(11 downto 0);

--decoder
signal s_decoder_y: std_logic_vector(6 downto 0);

--reg0
signal s_q_reg0: std_logic_vector(7 downto 0);

--reg1
signal s_q_reg1: std_logic_vector(11 downto 0);

--fsm
signal s_clr_pmc,s_ld_puc,s_clr_tc,s_ld_reg0,s_ld_reg1,s_rdy: std_logic;
--comp1
signal s_comp1: std_logic_vector(6 downto 0);

--fsm_pmc
signal s_eneabler,s_fsm_pmc_dn: std_logic;

--extras
signal s_pmc_eneabler: std_logic;


begin
		U0:  clk_gen                    port map(rst,clk,s_240us,s_15us);
		U1:  PulsoUnCiclo                   port map(rst,clk,cha,s_ys);
		U3:  pmc                      port map(rst,clk,s_pmc_eneabler,s_clr_pmc,s_pmc_count);
		U4:  puc                      port map(rst,clk,s_comp1,s_ys,s_ld_puc,s_puc_dn);
		U5:  tc                     port map(rst,clk,s_15us,s_clr_tc,s_tc_count,speed0);
		U6:  reg_hld_ld_n generic map(8) port map(rst,clk,s_ld_reg0,s_pmc_count,s_q_reg0);
		U7:  reg_hld_ld_n  generic map(12) port map(rst,clk,s_ld_reg1,s_tc_count,s_q_reg1);
		U8:  decoder                    port map(s_q_reg0,s_decoder_y);
```

```
        U10: comp1      generic map(7)   port map(s_decoder_y,s_comp1);
        U11: fsm_speed                   port map(rst,clk,s_fsm_pmc_dn,s_puc_dn,s_clr_pmc,s_ld_puc,s_clr_tc,s_ld_reg0,s_ld_reg1,s_rdy);
        U12: fsm_pmc                     port map(rst,clk,s_ys,s_puc_dn,s_eneabler,s_fsm_pmc_dn);

        ct     <= s_q_reg1;
        speedDn <= s_rdy;
        cp     <= s_comp1;
        s_pmc_eneabler <= s_240us and s_eneabler;


end estructural;
```

H. **Código Decodificador para Algoritmo adaptivo de velocidad (Decoder.vhd)**

```
library IEEE;
use IEEE.std_logic_1164.all;


entity decoder is
        port(
                X: in  std_logic_vector(7 downto 0);
                Y: out std_logic_vector(6 downto 0)
        );
end decoder;

architecture simple of decoder is
begin
        process(X)
        begin
                if ( X(7)= '1' )then
                        Y <="1111110";
                elsif( X(6)= '1' )then
                        Y <="1111110";
                elsif( X(5)= '1' )then
                        Y <="1111100";
                elsif( X(4)= '1' )then
                        Y <="1111000";
                elsif( X(3)= '1' )then
                        Y <="1110000";
                elsif( X(2)= '1' )then
                        Y <="1100000";
                elsif( X(1)= '1' )then
                        Y <="1000000";
                else
                        Y <="0000000";
                end if;
        end process;
end simple;
```