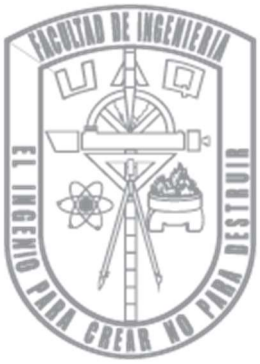
UNIVERSIDAD AUTONOMA DE QUERETARO

FACULTAD DE INGENIERÍA

INGENIERÍA EN AUTOMATIZACIÓN



Sistema Generador de Vibraciones

TESIS

Que como parte de los requisitos para obtener el título de Ingeniero en Automatización con línea terminal en Mecatrónica

Presenta:

Julio César Vargas Cano

Dirigido por:

Dr. José Marcelino Gutiérrez Villalobos

Querétaro, México 2019

Sistema Generador de Vibraciones

Julio César Vargas Cano

RESUMEN

El análisis sísmico de estructuras civiles tiene gran importancia en el diseño y la construcción gracias al avance tecnológico desarrollado a través de los años. Los nuevos sistemas de pruebas sísmicas permiten realizar ensayos dinámicos de las estructuras realizando movimientos controlados de posición y realizando pruebas con frecuencias y amplitudes específicas. El objetivo del presente trabajo es el diseño de un sistema capaz de controlar la posición de un motor de corriente directa sin escobillas (BLDC) con base en tecnología Field Programmable Gates Array (FPGA), el cual, mediante una transformación mecánica de movimiento rotacional en lineal, opera el movimiento de una plataforma con distintos modos de operación entre los que destaca la capacidad de generar señales de posición senoidales a diferentes frecuencias y amplitudes. Todo el sistema se controla por una interfaz gráfica donde el usuario puede preestablecer el movimiento senoidal además de observar la posición de la plataforma y los valores arrojados por los distintos acelerómetros.

Palabras Clave: Control de posición, motor BLDC, FPGA, señal senoidal, frecuencia y amplitud.

ABSTRACT

The seismic analysis of civil structures has taken a lot of importance thought the pass of years thanks to technology develop in construction and design. The new systems for seismic proofs allow make dynamic tests in structures making controlled movements of position and try with established frequency and amplitude values. The objective of this work is design a system capable of control the position of a brushless direct current motor (BLDC) based on Field Programmable Gates Array (FPGA) technology; which, through a mechanical transformation for rotational movement into lineal movement, manipulate the movement of a platform with some operation modes. One of those modes are the sinusoidal mode which is capable to generate sinusoidal position signals in different frequencies and amplitudes. All the system will be controlled by a Human-Machine Interface where the user could pre-establish the values of the sinusoidal movement besides could observe the actual position of the platform and the values acquired by the accelerometers in the structure.

Keywords: Position control, BLDC Motor, FPGA, sinusoidal signal, frequency and amplitude.

AGRADECIMIENTOS

A mis padres Rafaela y Julio que trabajaron mucho por lograr que yo alcanzara mis sueños, siempre me apoyaron y dieron lo mejor de sí por mí bienestar. Su cariño siempre me motivó a continuar trabajando y nunca detenerme.

A mis hermanos que con su gran apoyo y fuerza me guiaron y me empujaron a terminar, me enseñaron a jamás darme por vencido y a enfrentar los retos que la vida me ha puesto. Los quiero mucho.

A mis amigos y profesores que siempre me inspiraron y me demostraron que soy capaz de alcanzar grandes metas.

ÍNDICE DE CONTENIDO

ÍNDICE DE TABLAS

INDICE DE FIGURAS

CAPÍTULO 1: INTRODUCCIÓN

El jueves 19 de septiembre de 1985, en la Cuidad de México, un terremoto con magnitud 8.1 en la escala de Richter azoto el centro del país a las 7:19am, con dos minutos de duración el sismo se convirtió en uno de los desastres naturales más difíciles en la historia este país. El número de estructuras que quedaron destruidas debido al suceso se estiman cerca de 2800, los edificios de entre 5 y 7 pisos resultaron los más afectados debido a que las vibraban naturalmente en el rango de frecuencias en los que vibraban los suelos rocosos de la ciudad. Entre las estructuras que se consideran casos excepcionales se encuentran La Torre Latinoamericana y la Torre Ejecutiva de Pemex (Centro de instrumentación y registro sísmico A. C., 2017).

La Torre Latinoamericana es un edificio de 181m de alto soportado por 361 pilotes de concreto, el edificio se diseñó para soportar un sismo de una magnitud 9 en la escala de Richter (Fomperosa, 2017). ¿Cómo es que saben que la estructura soportará?, los ingenieros civiles han trabajado en modelos matemáticos que predicen el comportamiento de las estructuras ante excitaciones externas; sin embargo, estos modelos no toman en cuenta muchas variables aleatorias que suceden durante un terremoto por lo que se han desarrollado máquinas con las que se pueden probar los modelos a escala de las estructuras, y así conocer un comportamiento más real de la estructura frente a un sismo (Carrillo et al., 2013).

A estas máquinas se les conocen como Mesas Vibratorias, la construcción de estos equipos ha permitido que los ingenieros civiles alrededor del mundo puedan desarrollar y mejorar técnicas de construcción con la finalidad de evitar catástrofes. Las mesas vibratorias actuales cuentan con sofisticados sistemas de control para reproducir de manera exacta una señal constante o un sismo anterior, con estos nuevos estudios resulta más apropiado validar que una estructura pueda soportar ciertas magnitudes de aceleración y determinar la frecuencia natural de los edificios, que es la frecuencia en la que la estructura presenta una mayor oscilación y dónde las posibilidades de colapso aumentan. Con bajas frecuencias las estructuras más altas oscilan con mayor facilidad mientras que en frecuencias más grandes las estructuras de menor altura son las que presentan las oscilaciones (Carrillo et al., 2013).

Estos dispositivos se pueden hallar en distintos tamaños, existen desde mesas para probar estructuras de unos cuantos kilogramos, o modelos sencillos, hasta mesas

vibratorias dónde se construyen estructuras a escala real para revisar el movimiento exacto que tendrá la edificación. Existen mesas que se desplazan en los tres ejes, esto para asegurar que el movimiento telúrico se sienta en la estructura de una manera precisa. En la *Figura 1* se puede ver una mesa de gran tamaño para estructuras reales y en la *Figura 2* se aprecia una para modelos civiles a escala, como la que se pretende construir en este proyecto de tesis.

Figura 1. Mesa vibratoria a cargo de la Coordinación de Estructuras y Materiales de la UNAM.

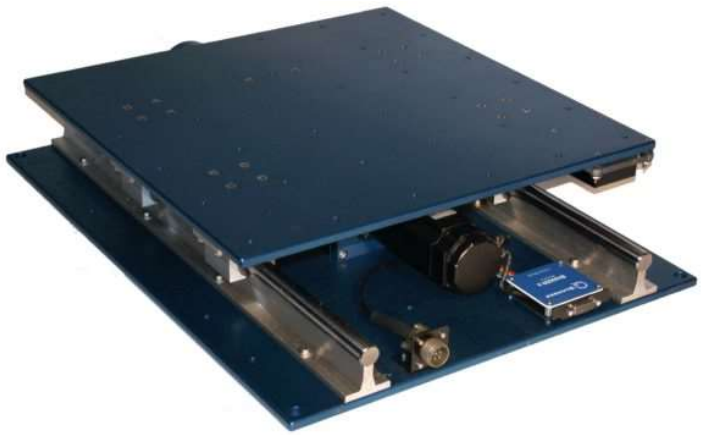


Figura 2. Mesa vibratoria comercial Qanser, capacidad 7.5kg.

Un estudio de movimiento resulta incompleto si solo se observa dicho desplazamiento, las mesas vibratorias de mayor calidad incorporan a su sistema la adquisición de datos, que comúnmente refiere a la aceleración que se está aplicando a la estructura en distintos niveles. También en muchas ocasiones se mide el desplazamiento de la base de la estructura, esto para comparar dicho movimiento del suelo con la magnitud de la aceleración registrada. Para fines de evaluación de la estructura todas las variables medidas se comparan en gráficas dentro de una interfaz usuario-máquina, donde los expertos pueden determinar con base a su conocimiento si la estructura es apta o no para su construcción.

1.1 Justificación

La necesidad de contar con una mesa vibratoria en la universidad reside en los beneficios que traerá para la docencia y el ámbito privado, pues estos dos tendrán la oportunidad de probar sus modelos y visualizar los efectos que sufren sus edificaciones en ensayos dinámicos, permitiendo a su vez el desarrollo e investigación de nuevas tecnologías útiles para evitar catástrofes.

El desarrollo de la mesa vibratoria dentro de la universidad asegura que la mesa cumplirá con los requisitos necesarios para llevar acabo las pruebas de laboratorio deseadas; contar con tecnología y desarrollo universitario nos permite disminuir sustancialmente los costos de fabricación.

Para lograr este cometido se diseñará un prototipo de una mesa vibradora controlada mediante un control en lazo cerrado de posición, esto nos asegurará un movimiento correcto en un eje para estresar los modelos que se deseen probar. Los ingenieros civiles podrán determinar, con una prueba de corta duración, la frecuencia natural de la estructura y su aceleración a dicha frecuencia en distintos niveles de ésta. Así los ellos determinarán si el modelo es apto para la construcción o requiere de un re-diseño que asegure que la estructura soportará en caso de un sismo.

1.2 Planteamiento del problema

El problema principal que busca solucionar el desarrollo de esta mesa vibratoria es la falta de este tipo de equipos en las universidades mexicanas. Sabemos que México es un

país altamente sísmico, los sismos son impredecibles por lo que se debe estar preparado ante un evento como estos. El desarrollo de estructuras sismo-resistentes es una solución a las catástrofes que trae consigo un fenómeno natural como este. Para que los ingenieros civiles puedan diseñar estructuras pensadas para esta clase de eventos se les debe proporcionar todas las herramientas posibles.

Los estudiantes de ingeniería civil de algunas universidades se ven en la necesidad de dedicar tiempo al desarrollo de mesas vibratorias manuales para validar algunos modelos estructurales que ellos realizan en la carrera, dichas mesas manuales no reproducen movimientos controlados, no reproducen patrones establecidos ni registran los datos de aceleración de los distintos niveles de la mesa vibratoria. Una mesa vibratoria que implementa un control de posición cubriría estos problemas pues los movimientos de la mesa se hacen controladamente y además ésta registra y despliega los datos en una interfaz para que el usuario pueda interpretarlos y así evaluar la estructura.

La falta de estos instrumentos de prueba en las universidades de debe al alto precio que estos equipos tienen en el mercado, pues es tecnología con gran tiempo de desarrollo y baja producción lo que los hace difíciles de conseguir. El desarrollo de un prototipo universitario permitirá, a los estudiantes de la UAQ, construir sus modelos y probarlos para comprender el comportamiento de las estructuras ante un sismo, además de probar distintos materiales y sus propiedades dinámicas ante los desplazamientos; en cuanto a los investigadores, les permitirá desarrollar nuevos métodos constructivos, y nuevas tecnologías sismo-resistentes.

El desarrollo de la mesa vibratoria en la Facultad de ingeniería posicionará a la Universidad Autónoma de Querétaro como una de las universidades líderes en el avance e investigación de estructuras sismo-resistentes en el país, pues pocas universidades mexicanas están de verdad preocupadas por el perfeccionamiento de nuevas tecnologías para evitar catástrofes urbanas.

1.3 Hipótesis y objetivos

1.3.1 Hipótesis general

Se puede diseñar y construir una mesa de vibraciones con un control de posición implementado en un motor BLDC para generar movimientos en una plataforma, esto nos permite realizar el estudio de aceleraciones presentadas en una estructura a partir del movimiento controlado de la base, para que los ingenieros civiles puedan evaluar daño

estructural en modelos a escala, identificar los problemas que la estructura puede presentar, probar nuevas tecnologías anti-sísmicas y validar diseños previos a su construcción.

1.3.2 Objetivo general

Desarrollar una mesa prototipo de vibraciones a través de un control de posición para motores de corriente directa sin escobillas (BLDC por sus siglas en inglés) y una interfaz de usuario que permite el control y el análisis de movimientos y aceleraciones en estructuras civiles.

1.3.3 Objetivos específicos

- Realizar en un FPGA un control de posición retroalimentado donde la señal real será adquirida mediante un encoder para asegurar que la plataforma se desplace la distancia requerida; el control permitirá trabajar empleando distintos modos de operación.

- Desarrollar un sistema de adquisición de datos y una interfaz usuario-máquina en el Software LabView para la visualización de los valores de aceleración que recopilen los sensores en distintas posiciones de la estructura civil; y desde donde se activarán los diferentes modos de operación del sistema y modificar sus características desde un formato amigable para el usuario.

- Diseñar y construir el prototipo de la mesa vibratoria empleando el software SolidWorks para tener documentado el diseño y establecer parámetros de la fabricación de la estructura de la mesa, permitiendo que esta sea replicable gracias a la documentación.

CAPÍTULO 2: REVISIÓN DE LITERATURA

2.1 Antecedentes

A continuación se describen los trabajos realizados anteriormente en temas de control de posición en motores BLDC, implementación de controladores en FPGA e interfaces gráficas en LabView en la Universidad Autónoma de Querétaro. También se describe un poco del desarrollo de mesas de vibraciones a lo largo de la historia, se citan trabajos relacionados con el control de plataformas similares que sirvieron de apoyo en la realización del presente trabajo.

2.1.1 Antecedentes UAQ

En cuanto a los trabajos realizados por los estudiantes de la Facultad de ingeniería de la Universidad Autónoma de Querétaro se pueden resaltar algunas tesis sobre el control de motores con FPGA como es el caso del documento presentado por García-Cortés (2014) que desarrolla un controlador de movimiento para un robot PUMA en FPGA, es este documento el compañero realiza un controlador PD similar al presentado aquí con la diferencia de que su controlador es implementado en la computadora en lugar de en la tarjeta de desarrollo. Por otra parte los compañeros Arana y Mendoza (2011) desarrollaron un controlador de movimiento para una bancada de dos ejes, ellos realizaron una interfaz gráfica en el entorno LabView que tiene similitudes a la presentada en el presente documento. Otro documento importante es el que desarrolló Mendoza-Mondragón (2012) como tesis de maestría en el cual usa un FPGA para el control de movimiento de múltiples motores de DC, en ese documento rescatamos la implementación discretizada del controlador dentro del FPGA y el seguimiento de trayectorias que propone, en ese caso para generar perfiles de velocidad.

Existe bastante bibliografía en el repositorio universitario sobre trabajos con los temas de control de posición, implementación en FPGA y el uso de motores BLDC, los mencionados anteriormente son los más importantes en relación directa al presente trabajo y además presentan un reto directo al desarrollo del proyecto. No es ajeno en la

universidad el desarrollo tecnológico que las tarjetas de desarrollo FPGA representan hoy en día en la industria, y por ello muchos trabajos de gran importancia basan su desarrollo en esta tecnología.

2.1.2 Antecedentes generales

A lo largo de los años los sismos han golpeado territorio Mexicano en distintas ocasiones, algunas de ellas de tal magnitud que han dejado daños irreversibles en las edificaciones a lo largo del país. Tan solo en el 2017 se produjeron alrededor de 26,364 sismos, según el Servicio Sismológico Nacional, pero tan solo 4 de ellos fueron de magnitud entre 6 y 8.9 (Servicio Sismológico Nacional, 2016). Estos rompimientos repentinos de las rocas del interior de la tierra son imposibles de predecir, su efecto libera energía acumulada por la resistencia de la roca propagándose en forma de ondas en todas direcciones provocando el movimiento del terreno, afectando a las construcciones.

La mesa vibratoria es un dispositivo que reproduce movimientos sísmicos reales debidamente registrados para investigar el comportamiento dinámico y la respuesta sísmica de modelos estructurales de manera precisa (Unidad de Puentes, 2013). Carrillo, Bernal Ruíz, & Porras (2013) mencionan que las mesas vibratorias son aparatos que permiten simular, médiate una plataforma móvil, el movimiento que produce un sismo, los ensayos son dinámicos y de corta duración. Encima de la mesa se escala un modelo de una edificación y se prueba para conocer su comportamiento en distintas frecuencias, la mesa tiene el fin de ensayar nuevos sistemas constructivos, nuevos materiales, sistemas aisladores de base, disipadores de energía y para aprobar la construcción de las edificaciones. Además se colocan sobre el modelo una serie de instrumentos para monitorear su respuesta.

Las primeras mesas vibratorias comenzaron a desarrollarse en Tokio en el año 1893, se movían mediante una rueda mecánica con una manivela conectada a una plataforma sobre rieles. Unas décadas más tarde comenzaron a sustituirse los sistemas manuales por motores eléctricos, estas mesas fueron pieza clave para la realización de numerosos avances en la construcción de estructuras sismo-resistentes (Carrillo et al., 2013).

Las mesas vibratorias actuales comenzaron a partir de los avances logrados en la segunda guerra mundial. En estos días las mesas usan sistemas de control avanzados para simular sismos reales. Existen alrededor del mundo mesas vibratorias de gran tamaño con capacidad de movimiento en dos ejes que han sido fundamentales para el análisis, diseño y rehabilitación de edificios, puentes, entre otras estructuras de gran importancia.

Las mesas se clasifican en 3 principales grupos según Lehmann et al. (2012) en mesas de gran tamaño que permiten realizar ensayos sobre estructuras de tamaño y peso real; mesas de mediano tamaño las cuales son útiles para hacer pruebas dinámicas sobre modelos a escala y de un peso reducido; finalmente, las mesas pequeñas que son las que permiten realizar ensayos sobre modelos sencillos y pequeños.

Algunas universidades alrededor del mundo se han inclinado por el desarrollo propio de mesas vibratorias debido a los altos costos que manejas las empresas en este ramo y a la falta de variedad en ellas. La empresa líder en mesas vibratorias Quanser maneja la mesa de pruebas Shake Table II, siendo esta una de las más vendidas no tiene las características específicas para muchas aplicaciones (Peralta et al., 2013).

Muchos de los desarrollos tecnológicos en cuestión de mesas de vibraciones sísmicas presentan estructuras similares para el movimiento que consisten en la plataforma rígida que en algunos casos es sujetada por pistones hidráulicos como lo presenta Conte, et al. (2000) que son dispositivos para estructuras muy pesadas o reales debido al poder hidráulico. En una cuestión más didáctica podemos encontrar mesas de menor tamaño con un sistema de movimiento electromecánico que consta de motores y elementos mecánicos como los que se ven en Baran et al. (2010) ya que estos sistemas tienden a ser más exactos y a tener un mejor control lo cual beneficia a las pruebas a realizar.

México siendo uno de los países más activos sísmicamente solo cuenta con una mesa vibratoria considerada de gran tamaño en la Universidad Nacional Autónoma de México, a cargo de la Coordinación de Estructuras y Materiales. Ha estado activa desde 1997 y fue donada a ésta universidad por el Kajima Technical Institute después de ser elegida entre 8 universidades del mundo. Esta mesa está a disposición de los investigadores de la UNAM, instituciones de investigación y docencia, así como para el servicio a la industria (Ciencia UNAM, 2013).

La necesidad de continuar con el crecimiento de las ciudades y el temor de presenciar un sismo han orillado a los ingenieros civiles a diseñar y construir estructuras sismo-resistentes. Existen herramientas basadas en modelos matemáticos que los ingenieros usan para determinar el comportamiento de las estructuras en caso de un terremoto, pero dichos modelos no tienen en cuenta los efectos aleatorios que ocurren durante un sismo real, así que se ha optado por agregar al estudio de las estructuras sismo-resistentes distintas pruebas que complementen los modelos, como por ejemplo las mesas vibratorias, ensayos dinámicos y estáticos, con el fin de que los ingenieros puedan construir estructuras más confiables ante los desastres naturales (Lehmann et al., 2012).

CAPÍTULO 3: METODOLOGÍA

3.1 Marco teórico

El control de motores es un campo ampliamente estudiado, en ellos se basa el movimiento de los sistemas robóticos de la actualidad, desarrollando sistemas integrados de funcionamiento general para aplicaciones en propósitos específicos, como el movimiento de un eje en un brazo robótico, la ubicación y movimiento de un cabezal en una impresora 3D o el posicionamiento de una bancada en una máquina de control numérico. Un sistema único de movimiento, como los anteriormente mencionados, se controlan de forma independiente, pero al ver trabajar distintos al mismo tiempo da la impresión de tener un movimiento en conjunto dentro de un dispositivo robótico.

Estas unidades independientes de movimiento se integran por ciertos componentes que convierten al conjunto en un llamado servosistema. Estos sistemas se componen de un servomotor, un dispositivo de detección de posición, un ensamble mecánico, un circuito amplificador y una unidad de control. El sistema generador de vibraciones funciona bajo el principio de un servosistema por el hecho de que estos dispositivos, que conjuntan lo electrónico y lo mecánico, trabajan con gran eficiencia y precisión.

3.1.1 El motor BLDC

Uno de los componentes anteriormente mencionados que componen al servosistema es el motor en sí. Existen diferentes tipos de motores con los que trabajan los servosistemas, éstos se pueden dividir en dos grandes grupos. Los motores de Corriente Alterna (CA) que son en los cuales la velocidad del motor es determinada por la frecuencia del voltaje aplicado y el número de polos magnéticos, éstos tienen una ramificación hacia la forma en la que operan, éstos puede ser síncronos o asíncronos.

Por otro lado se encuentran los motores de Corriente Directa (CD) en los cuales la velocidad es proporcional al voltaje suministrado a una carga constante, éstos se pueden

dividir en dos diferentes grupos los que trabajan con escobillas y los motores sin escobillas llamados motores BLDC (Brushless Direct Current).

Los motores con escobillas funcionan mediante el uso de un conmutador, que es un switch electrónico que periódicamente invierte la dirección de la corriente en el rotor. Éste es un cilindro con segmentos metálicos que tienen contacto con las escobillas que permiten la polarización del circuito en el rotor generando un campo magnético cambiante en éste. El motor con escobillas más usado es el de imanes permanentes que actúan con el campo magnético del rotor haciendo girar la flecha.

Por otra parte se encuentran los motores BLDC tienen imanes permanentes de distintos materiales en el rotor y podemos encontrar desde dos hasta ocho pares de polos alternados entre Norte (N) y Sur (S); mientras que en el estator tienen embobinados que son conmutados de acuerdo a la posición del rotor con respecto al estator (*Figura 3*). En este tipo de motores se pueden encontrar configuraciones para su funcionamiento con una, dos y tres fases, de acuerdo a esta configuración el estator presenta cierto número de embobinados, siendo los de tres fases los más populares y los más comúnmente usados.

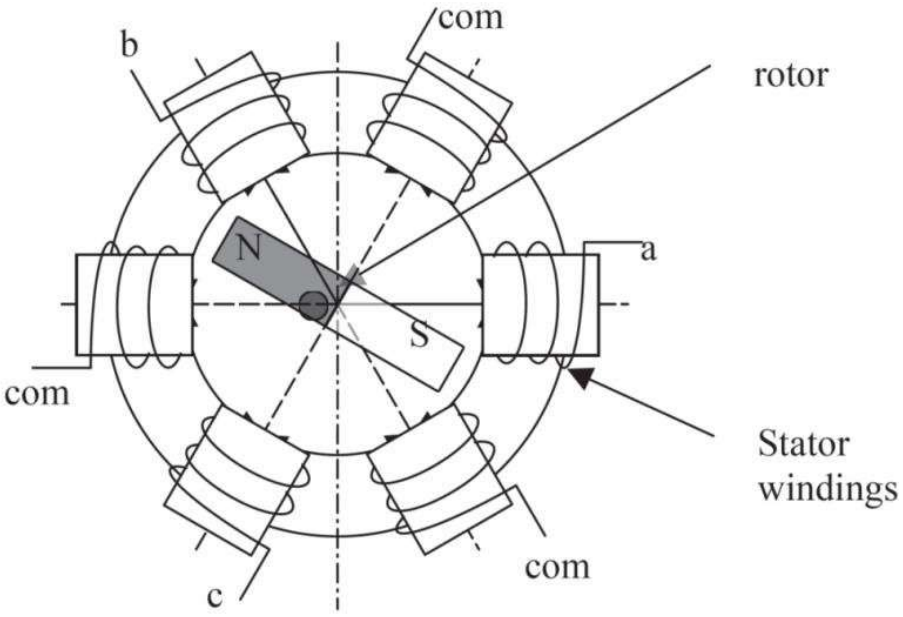


Figura 3. Diagrama interno de un motor BLDC.

Los BLDC son motores síncronos, esto significa que el campo magnético del estator y el campo magnético que genera el rotor giran a la misma frecuencia. A diferencia de los motores con escobillas, la conmutación de un motor BLDC se hace electrónicamente. Para hacer girar el rotor las bobinas del estator se deben de energizar con cierta secuencia. Los motores BLDC con frecuencia cuentan con sensores de efecto Hall unidos a la flecha del rotor (*Figura 4*). Cada vez que los polos magnéticos de rotor pasan a través del sensor Hall

entregan una señal en alto o en bajo indicando que el Norte o el Sur del polo está pasando a un costado del sensor. Usando la combinación de tres sensores de efecto Hall se determina con exactitud la secuencia de conmutación.

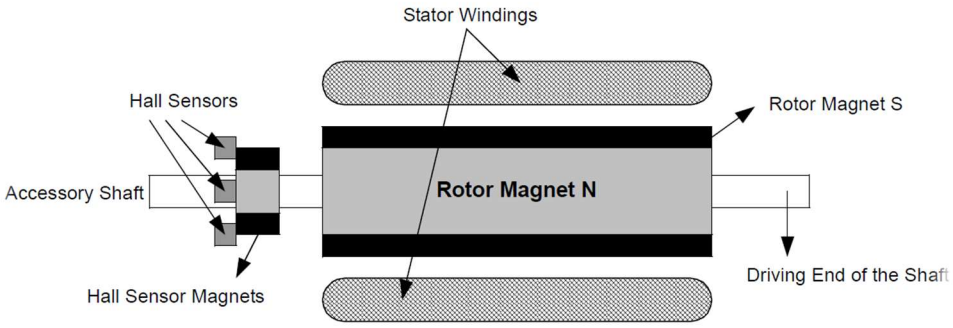


Figura 4. Partes del motor BLDC.

Cada secuencia de conmutación en el motor BLDC mantiene una bobina energizada positivamente (La corriente entra a la bobina), la segunda se polariza negativamente (La corriente sale de la bobina) y la tercera no está energizada, así el torque es generado de acuerdo a la interacción de los campos magnéticos formados en el estator junto a los imanes permanentes en el rotor. Para hacer que el rotor continúe moviéndose las bobinas deben conmutar en una secuencia de 6 pasos (*Figura 5*).

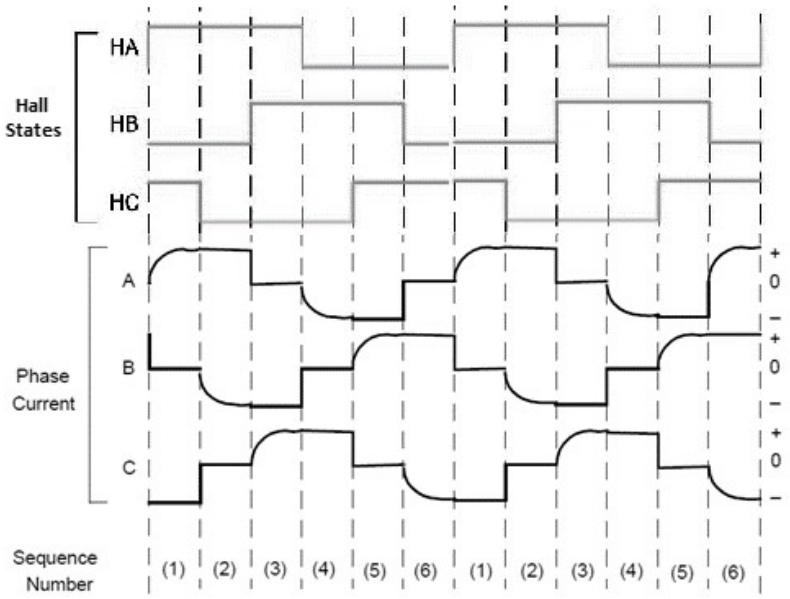


Figura 5. Comportamiento de las fases de un motor BLDC.

Este tipo de motores se usan principalmente en aplicaciones con constantes arranques y frenados, así como constantes cambios del sentido de giro, demandando mucho torque especialmente en los arranques debido a la inercia que la carga le genera al rotor. Las ventajas de este motor recaen en el bajo mantenimiento que requieren así como en su larga vida útil; los rangos de velocidad que manejan son muy amplios generando poco ruido con gran eficiencia.

3.1.2 El amplificador

Para conmutar de manera rápida y precisa se requiere de un dispositivo electrónico que energice las bobinas del rotor a medida que toma la lectura de la posición del rotor dada por los sensores de efecto Hall. Para eso se requiere de un amplificador, este dispositivo se encarga de activar las bobinas en la secuencia correcta, dada por los 6 pasos mostrados en la *Figura 5*. Esta energización requerida de las bobinas debe de crearse a través de un voltaje alto por lo que el amplificador debe de trabar con una tensión diferente a la usada por lógica que controla la activación y la lectura del estado que entregan los sensores Hall.

Los amplificadores internamente cuentan con un arreglo de transistores (Figura 6) que conmutan, con señales digitales, la activación de grandes voltajes de corriente directa. Es un punto a contrastar que un motor BLDC se tiene que energizar con una fuente de alimentación de frecuencia variable y tres fases de corriente alterna para su funcionamiento. El amplificador activa las líneas de suministro de la corriente directa con la secuencia y en la dirección que muestra la *Figura 5*, creando la señal de corriente alterna necesaria para el funcionamiento del motor; La frecuencia de esta señal dependerá de la velocidad a la que el sistema digital conmute sus salidas.

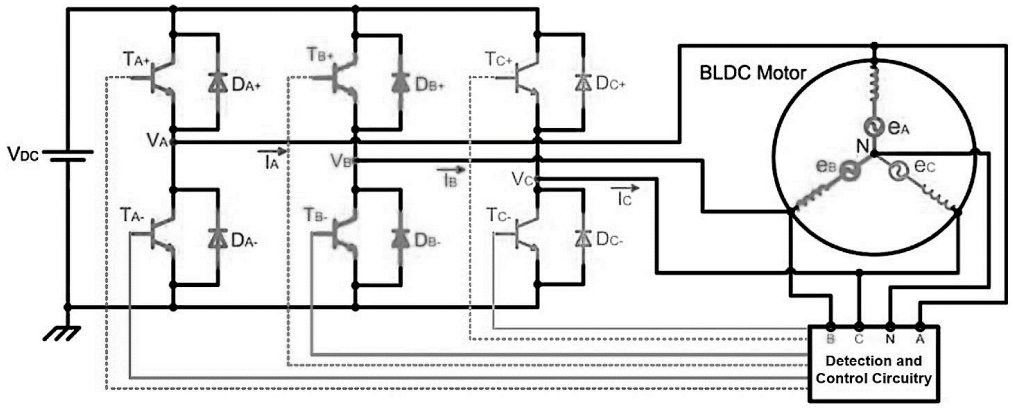


Figura 6. Arreglo interno para la conmutación de las fases.

Típicamente los amplificadores son suministrados con voltaje de CD y una señal de voltaje variable entre -10v y 10v, este voltaje funciona como referencia para la velocidad de salida del motor, el voltaje negativo significa un cambio de giro, mientras que el voltaje en cero equivale a que el motor se mantenga estático.

3.1.2.1 Acoplamiento de señales

Como se comentó, el sistema de amplificación típico recibe señales analógicas que van de los 10v hasta los -10v, mientras que los sistemas electrónicos, que típicamente hacen el procesamiento de datos y control, trabajan con señales lógicas de 5v por lo que en medio de ambos sistemas debe de haber un circuito que transforme la señal lógica en una análoga.

El primer requerimiento para acoplar las señales es lograr que es sistema de control envíe su señal de corrección (*Capítulo III.2.5 Unidad de control*) de manera analógica, esto el sistema de control no lo puede realizar debido a que sus señales son digitales, así que es necesario emplear un *Convertidor Digital-Analógico* (DAC, por sus siglas en inglés). El DAC es un dispositivo electrónico que típicamente trabaja con el mismo voltaje que el sistema electrónico, de 0 a 5v de DC, así que sus valores analógicos tendrán estos límites de voltaje; otro dato importante a conocer sobre los DAC es la resolución, y ésta está dada por la cantidad de bits que entran al dispositivo para controlar la salida.

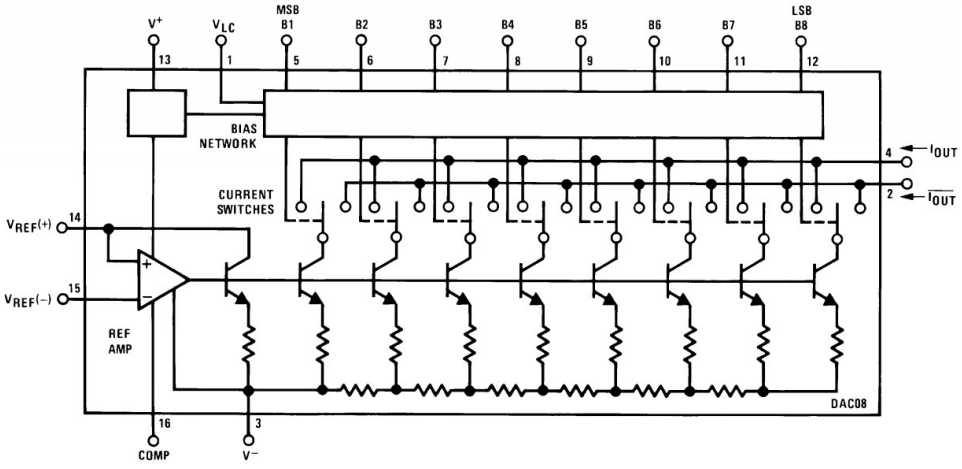


Figura 7. Diagrama interno del DAC0800.

La resolución de un DAC es simplemente la máxima diferencia de potencial que alcanza a la salida, dividido entre el número *decimal* máximo de entrada (*Ecuación 1*), donde podremos ver el voltaje por unidad, así calcularemos el número *binario* necesario para cierta cantidad de voltaje. Los convertidores reciben una señal de una longitud definida de

forma digital, por lo tanto, existen dos tipos de Convertidores Digital-Analógico, los que trabajan con un bus de datos en paralelo, lo cual significa que tiene tantas entradas como bits de resolución, en la *Figura 7* podemos observar el DAC0800 de 8 bits de resolución, como se puede observar tiene una gran cantidad de entradas lo que podemos traducir en la misma cantidad de salidas del sistema electrónico que lo controla; Por otra parte también podemos encontrar comercialmente los DAC que trabajan mediante algún protocolo de comunicación, éstos requieren de una programación más extensa pero ahorran puertos de entradas y salidas digitales al sistema electrónico de control.

$$Resolución = \frac{Diferencia\ Máxima\ de\ potencial}{2^{número\ de\ bits}} \quad (1)$$

Se pueden encontrar distintos tipos de protocolos de comunicación para estos dispositivos, pero uno de los más comunes es sin duda el protocolo I^2C (Se ve más a fondo en el *Capítulo III.2.6.1*), mediante este protocolo podemos controlar un DAC con tan solo dos cables de comunicación digital, los llamados SCL y SDA (Serial Clock y Serial Data). En la *Figura 8* se puede observar un diagrama de bloque que explica los componentes internos de un DAC I^2C.

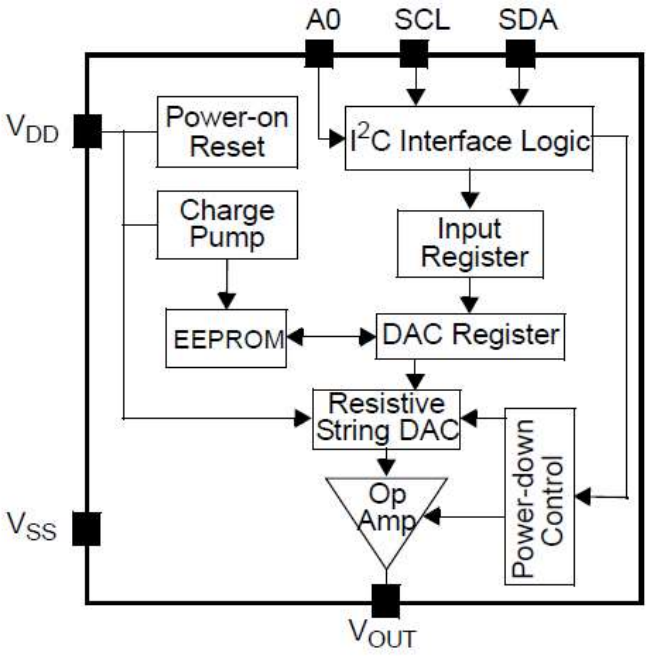


Figura 8. Diagrama de bloques de un DAC I^2C.

Una vez que el voltaje es controlado analógicamente se requiere que este trabaje tanto en valores de voltaje negativos como positivos, algo que un DAC como el anterior no puede realizar. Una de las opciones más comunes es usar otro Convertidor Analógico-Digital con su salida de voltaje invertida, y así controlar tanto los voltajes positivos como los negativos.

Existe otra solución a este problema y es el uso de *Amplificadores Operacionales* (OpAmp, por su acrónimo en inglés). Una configuración para OpAmp que nos serán muy útil para esta aplicación es usarlo como restador (*Figura 9*) nos permite hacer la diferencia entre dos tensiones en sus entradas, además de que si usamos ciertas resistencias podemos amplificar la señal a la salida. Para conseguir que el voltaje analógico de 0 a 5v se transforme en una señal de -10v a 10v seccionaremos en dos partes el proceso pues la configuración no nos permite lograrlo con un solo arreglo, por lo que se requiere de un arreglo en cascada de dos Amplificadores Operacionales.

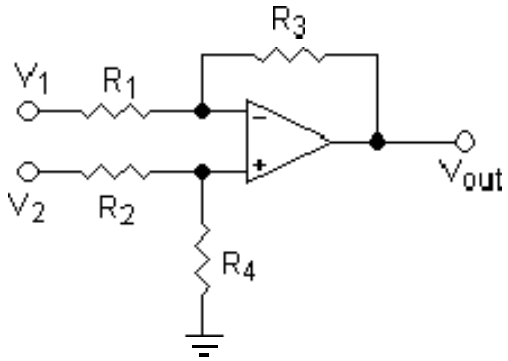


Figura 9. Circuito OpAmp restador.

Primero necesitamos conocer las ecuaciones que gobiernan el comportamiento de ésta configuración para así verificar que el proceso funciona. La *Ecuación 2* indica cómo es el voltaje a la salida del OpAmp. La configuración inicial tienen que lograr que la señal de voltaje trabaje desde valores negativos hasta valores positivos, para esto le restaremos un voltaje a la señal que sale del DAC, así obtendremos una señal de -5v a 5v; La segunda parte no restará nada a la señal, pero la amplificará dos veces para obtener a la salida la señal que necesitamos.

$$V_{out} = V_2 * \left(\frac{R_4}{R_2 + R_4}\right) * \left(1 + \frac{R_3}{R_1}\right) - \left(V_1 \frac{R_3}{R_1}\right) \qquad (2)$$

Consideremos para ambas configuraciones los siguientes valores en las resistencias:

$$R_1, R_3, R_4 = 10K\Omega \ , \qquad R_2 = 0\Omega$$

Con estos valores propuestos podemos armar el circuito de la *Figura 10*, analizándolo y aplicando la *Ecuación 2* podemos observar cómo se comporta el circuito, bajo una ecuación final. El Integrado propuesto para esta aplicación es el TL082, pues internamente tiene dos Amplificadores Operacionales.

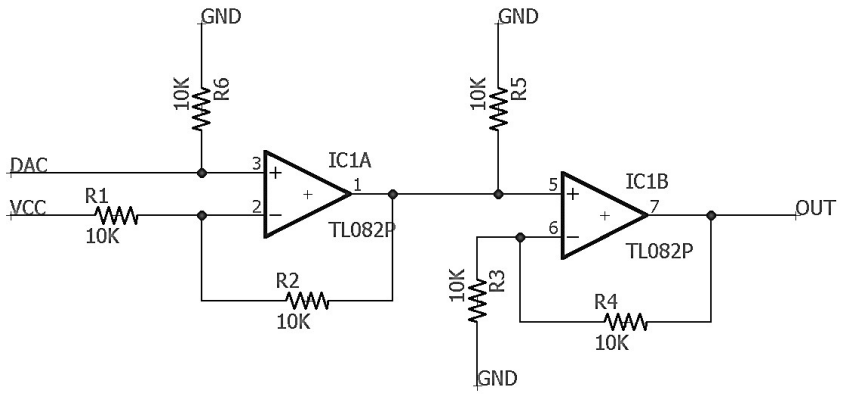


Figura 10. Circuito de acoplamiento DAC-Amplificador.

Aplicando los valores de las resistencias a la *Ecuación 2*, obtenemos:

$$V_{out} = V_2 * \left(\frac{10K\Omega}{0 + 10K\Omega}\right) * \left(1 + \frac{10K\Omega}{10K\Omega}\right) - \left(V_1 \frac{10K\Omega}{10K\Omega}\right)$$

$$V_{out} = V_2 * 2 - V_1 \quad (3)$$

La *Ecuación 3* es la ecuación para cada uno de los Amplificadores, a continuación sustituiremos el V_2 en una ecuación con el V_{out} de otra, justo como se observa el circuito de la *Figura 10*, también usaremos los valores de ambos V_1 que en la primera ecuación vale 5v y en la segunda ecuación vale 0v pues está conectada a GND. El primer circuito se denota con el sufijo "a" y el segundo circuito con el sufijo "b".

$$V_{out\,b} = (V_{2a} * 2 - V_{1a}) * 2 - V_{1b}$$

$$V_{out} = \Big(\big((V_{DAC} * 2) - 5v\big) * 2\Big) - 0v \quad (4)$$

Voltaje proveniente del DAC	Voltaje de salida al Amplificador
5v	10v
4v	6v
3v	2v
2.5v	0v
2v	-2v
1v	-6v
0v	-10v

Tabla 1. Comportamiento de los voltajes a la salida del circuito.

La *Ecuación 4* gobierna el comportamiento del circuito después del DAC y antes del amplificador, con ésta ecuación podemos obtener la *Tabla 1* que nos muestra el comportamiento de salida con diferentes entradas. Notemos que el valor de salida de 0v se encuentra con el valor de 2.5v proveniente del DAC, por lo que podemos concluir que idealmente el estado dónde el motor está estático, porque no recibe voltaje, es el valor medio de la resolución de nuestro DAC.

3.1.3 Encoder

Es necesario para controlar la posición de un motor tener un valor equivalente al ángulo al que se encuentra el rotor de nuestro motor, una forma típica y sencilla de conocer este parámetro es bajo el uso de un sensor de posicionamiento óptico conocido como encoder rotativo. Es posible también conocer la posición del rotor con el uso del sensor de efecto Hall integrado en un motor sin escobillas, pero para esto se requiere de un análisis extra de los datos entregados por el sensor por lo que un encoder es lo más útil y eficiente.

Se pueden encontrar encoder ópticos tanto Absolutos como incrementales (*Figura 11*). Los encoder ópticos operan con el uso de un disco con ranuras y elementos de reflexión, que dependen de un emisor y un receptor; El emisor pasan su haz de luz a través del disco, si el receptor detecta el haz significa un estado lógico, el caso contrario representará el estado lógico complementario.

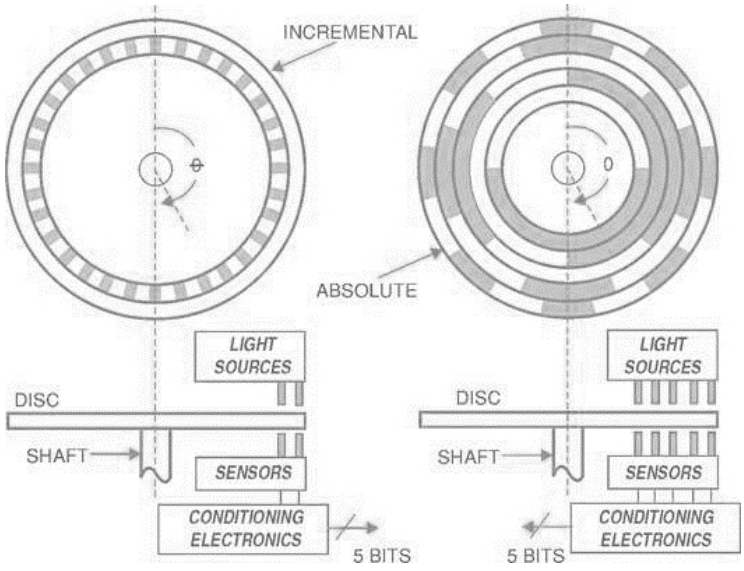


Figura 11. Comparación de encoder incremental y absoluto.

Los encoder Absolutos tienen un disco donde la combinación de sus muescas en una posición específica representa un ángulo único o una porción de circunferencia, así que un arreglo lógico es la salida que se obtiene de éste tipo de encoder. Los encoder Absolutos tienen la ventaja de conocer la posición en la que se encuentran incluso después de haber sido des-energizados pues la combinación de sus salidas les permiten reconocer ese dato al energizarse.

Por otra parte los encoder Incrementales tienen un disco con múltiples ranuras sobre una altura específica del disco y equidistantes entre ellas (un patrón periódico), a través del disco pasan únicamente dos haces de luz que están desfasados entre ellos, obteniendo a la salida dos señales cuadradas desfasadas en el tiempo (*Figura 12*). La secuencia en que un dispositivo lee esta señal determina la dirección que el disco tiene y por consecuencia lo que esté acoplado a él.

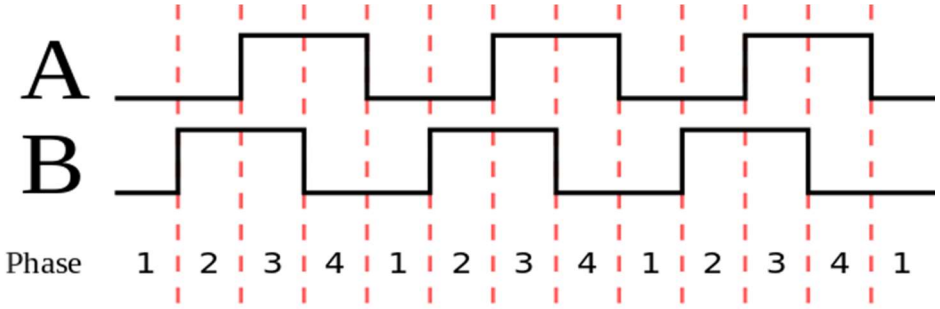


Figura 12. Secuencia del encoder.

Una ventaja de los encoder Incrementales es que pueden contar las vueltas completas que el rotor ha dado, las señales A y B anteriormente mostradas (*Figura 12*) son las salidas principales de nuestro encoder, muchos encoder presentan una tercera salida (Señal C) que indica con un pulso en alto cada que el encoder ha pasado por un punto en específico, o en otras palabras, cada que el rotor ha dado una vuelta completa.

Algunos parámetros importantes para trabajar con este tipo de encoder es la cantidad de ranuras que presenta el disco, pues esta será exactitud con la que podremos posicionar el disco, o sea, el ángulo al que se encuentra el rotor. Al conocer esta cantidad de ranuras podemos calcular cuánto es el ángulo mínimo de movimiento que nos permite el encoder, pues una precisión más grande será prácticamente imposible, el dato se conocerá dividiendo los 360° presentes en una circunferencia entre la cantidad de ranuras. Al conocer la cantidad de ranuras también podemos conocer cuando el encoder haya pasado en un punto específico, sin la necesidad de contar con la *Salida C,* solo hace falta que el dispositivo que recibe la señal del encoder cuente hasta el número de ranuras para saber que se dio una vuelta completa.

El conteo no es tan sencillo como parece pues no se trata solo de contar cuando la señal del encoder tiene un cambio en su estado, pues se trata de conocer en que paso está

de la señal recibida en las terminales A y B conociendo cual fue el estado anterior para compararlos y conocer la dirección. Esto ocasiona que cada ranura del encoder tenga cuatro estados, y por ende el conteo de las señales se multiplique por cuatro. Con esto podemos obtener las siguientes formulas.

$$Presición\ (°) = \frac{360°}{Número\ de\ ranuras * 4} \tag{5}$$

$$Cuentas\ por\ vuelta = Número\ de\ ranuras * 4 \tag{6}$$

3.1.4 Dispositivo mecánico

Cuando se controla la posición a la que gira el rotor del motor es útil conocer la aplicación del servomotor, pues ésta determinará una serie de elementos mecánicos que usarán el movimiento rotatorio del eje y lo transformarán para el movimiento de una estructura mecánica. Para poder generar una oscilación en cualquier cuerpo tal como lo hace un sismo se requiere que la estructura base del cuerpo tenga movimiento. El sistema requerido para una aplicación dónde la base mueve objetos sujetados a él, se asimila a una aplicación muy conocida cómo lo son las máquinas de control numérico.

El sistema base de movimiento de máquinas como esta es tan común que tiene una extensa aplicación en la industria, y en otras aplicaciones, las encontramos en maquinas CNC, impresoras 3D, cortadoras laser, entre otras. El movimiento de una plataforma metálica se tiene que hacer transformando el movimiento rotacional en un movimiento lineal, el elemento más comúnmente usado es el tornillo de bolas.

El tornillo de bola consiste en una barra metálica con un corte en espiral que funciona como guía para un bloque que se desliza linealmente a lo largo de la barra, el bloque logra avanzar debido a un arreglo de balines internos que giran dentro de los canales del tornillo empujando el bloque que tiene dentro un corte complementario similar al de la barra (*Figura 13*).

Figura 13. Interior del bloque de deslizamiento.

La cantidad de avance que el bloque tendrá siempre depende de la cantidad de espiras que contenga el total del largo del tornillo. Esto logra, según la aplicación, que podamos definir la cantidad de avance que se requiere linealmente por cada vuelta, con este dato inicial se puede inferir la cantidad de vueltas necesarias para completar una unidad de movimiento.

En el caso particular de la aplicación aquí desarrollada se requiere cierta cantidad de información mecánica para controlar la posición de la plataforma. Anteriormente se planteó la *Formula 6* con la que podemos obtener la cantidad de pulsos que recibiremos del encoder por cada vuelta del rotor, ahora es necesario calcular la cantidad de pulsos que obtendremos para mover la plataforma una unidad de longitud. A partir de ello y con la ayuda de un dato de diseño que es la cantidad total de movimiento requerida por la aplicación podemos inferir la cantidad de pulsos que obtendremos, como entrada en el sistema de control, por el desplazamiento entre los límites máximos de la plataforma.

$$Cuentas\ por\ unidad = Cuentas\ por\ vuelta * Vueltas\ por\ unidad\ de\ desplazamiento \quad (7)$$

$$Valor\ máximo = Cuentas\ por\ unidad * unidades\ de\ desplazamiento\ requeridas \quad (8)$$

3.1.5 Unidad de control

En general el control trata de lograr que un sistema reaccione y cumpla con los parámetros que el usuario requiere para la aplicación en la que se encuentra mediante

métodos matemáticos. En la aplicación que estamos desarrollando es necesario hacer un control de posición en el motor para obtener desplazamientos controlados de la plataforma. En general el control se basa en la diferencia que hay entre la variable medida y la variable deseada, por ejemplo, para la posición de un motor se obtiene el dato del encoder que equivale a la posición actual y se obtiene un dato suministrado por el usuario que equivale a la posición deseada; La diferencia de estas dos variables es el *Error*, dato principal de la unidad de control. El error es manipulado internamente para que a la salida de la unidad de control haya una señal de corrección que haga que el motor llegue a la posición deseada.

Como se explicó anteriormente, la plataforma obtiene su desplazamiento de la transformación de movimiento rotacional de un motor, manteniendo la premisa de que si el motor gira en sentido horario la placa se moverá a la derecha, caso contrario, si el motor gira en sentido anti horario la plataforma se moverá a la izquierda. El sentido de giro del motor está controlado por el amplificador que recibe un señal donde el voltaje positivo (hasta 10V) hace girar el motor en sentido horario, el voltaje negativo (hasta -10V) lo hace girar en sentido anti horario, por último, suministrarle una diferencia de tensión de 0V hace que el motor no se mueva.

La unidad de control también tiene la función de interpretar los datos que recibe de las salidas A y B del encoder, estas señales las transforma en un número entero para su fácil comprensión, donde el valor máximo de éste número es el calculado en la *Ecuación 8*, así la unidad sabe en todo instante la *posición real del motor* (θ) y recibe mediante algún método un valor asignado por el usuario que se denomina *posición deseada* (θ*d*). El control busca que θ = θ*d* en todo momento. Con esto podemos determinar que el error se calculará de la siguiente manera:

$$e = kp\ (\theta d - \theta) \tag{9}$$

Donde *Kp* es una constante positiva. Podemos observar que para la aplicación, tal como se describió anteriormente, el *error* es directamente proporcional al voltaje de salida, pues cumple con los requerimientos, el voltaje como el error pueden ser positivos o negativos, además de ser cero cuando la *posición real* alcanzó la *posición deseada*.

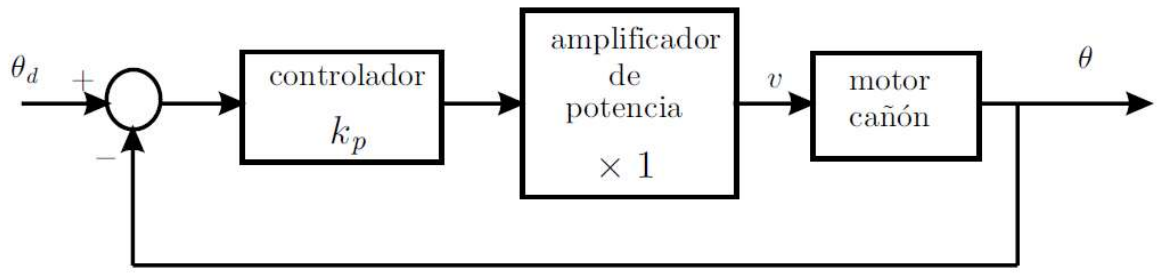


Figura 14. Diagrama a bloques del sistema de control de un cañón antiaéreo.

El diagrama a bloques de la *Figura 14* representa gráficamente el funcionamiento general del control de posición y sus partes a desarrollar, se observa una constante *Kp* que se define como la ganancia proporcional está multiplicando al valor de voltaje con el que se alimenta al motor, por lo tanto el motor girará más rápido o más lento dependiendo del valor de la constante. El tipo de respuesta de la posición con respecto del tiempo varía según sea el valor de *Kp*. Con un valor mayor la posición llegará más rápido al valor deseado pero el sobrepaso de la señal va a ser también grande pues la velocidad con la que el motor se acerca a θ*d* es grande y al sistema le cuesta frenar a tiempo para cambiar de sentido y estabilizarse; con una *Kp* baja, el sistema tiende a ser más lento, pero con un sobrepaso menor o incluso nulo.

Un punto importante del diseño de un sistema de control es la estabilidad, para afectar a un sistema de control se consideran perturbaciones externas a cualquier movimiento que altere la estabilidad del estado estacionario, en otras palabras, se requiere que la posición del eje se mantenga en la *posición deseada* aunque externamente algo esté alterando al sistema. Muchos sistemas mecánicos están pensados para trabajar en distintos ejes pero en algunos ejes existen fuerzas constantes que actúan sobre el sistema, la gravedad actúa sobre un eje que es perpendicular al eje de trabajo de la plataforma, por lo que en análisis estático no actúa sobre el movimiento de la placa de nuestro sistema alterándolo en el estado estacionario, se dice que el error en estado estacionario es cero para un sistema como este que trabaja sobre el eje horizontal.

Las señales de movimiento que se quieren estudiar son variantes con el tiempo, es posible que el controlador sea capaz que realizar cambios en su posición deseada para adaptarse a trayectorias de posición, una forma de ver cualquier señal en el tiempo es con muestras cada tiempo constante. La señal de las muestras tendrá una apariencia escalonada si el tiempo entre muestras es muy largo. Podemos de forma sencilla definir cuantas muestras nos son suficientes para conseguir los resultamos que deseamos.

La señal que se plantea, que cuenta con algunas características en común con los sismos, es la senoidal, pues nos ofrece dos rasgos importantes, la *frecuencia* y la *amplitud*. Para enviar la magnitud al sistema de control se requiere que la señal senoidal sea seccionada para que asemeje una entrada de una forma escalonada. El sistema de control tiene que ser lo suficientemente rápido, como mínimo, para que el tiempo de subida de la señal alcance la posición del escalón siguiente para poder realizar la trayectoria senoidal correctamente, manteniendo los valores de *frecuencia* y *amplitud* que se desean. Bajo este principio de movimiento es posible replicar casi cualquier señal variante en el tiempo, la limitante que tiene es la velocidad del motor. Una diferencia en la posición tan grande, que la velocidad máxima del motor no pueda alcanzar en el tiempo antes de que otro valor de posición sea establecido, será el caso en el que no se pueda seguir con exactitud la trayectoria deseada.

3.2 Descripción de hardware FPGA

El sistema que realiza el control, el monitoreo las señales, el envío de datos para graficar, el control del voltaje de salida, y los distintos protocolos de lectura de datos son actividades que se deben ejecutar al mismo tiempo para no perder un solo dato. Los microcontroladores dependen de la velocidad de un cristal para ejecutar cada instrucción de lo programado, como es de forma secuencial la ejecución de las instrucciones en el sistema puede ser más lenta. Un Field Programmable Gate Array o FPGA es un dispositivo capaz de modificar la configuración interna de su sistema para obtener, en un tic de reloj, múltiples tareas sencillas, así que la cantidad de estas compuertas determina la cantidad de procesos en paralelo que puedas tener. Esta característica permite que en un solo sistema de control pueda hacer el cálculo del error en cada instante ejecutando una acción correctiva, al mismo tiempo que realiza otras tareas como el envío de datos de posición y aceleración a la computadora para que se grafiquen; otra tarea que el sistema principal debe realizar es la lectura de la aceleración, que es un dato importante que puede ayudar a los ingenieros civiles a entender que sucede en la estructura al momento de un sismo.

3.2.1 Protocolo I^2C

Se necesitan programar algunos protocolos en la tarjeta FPGA para comunicarse con los dispositivos electrónicos que controlan la parte mecánica, como se comentó anteriormente (Capitulo III.1.2.1 Acoplamiento de señales) el amplificador necesita de una señal transmitida en el protocolo I^2C para poder ejecutar las tareas de conversión de datos digitales a una señal analógica.

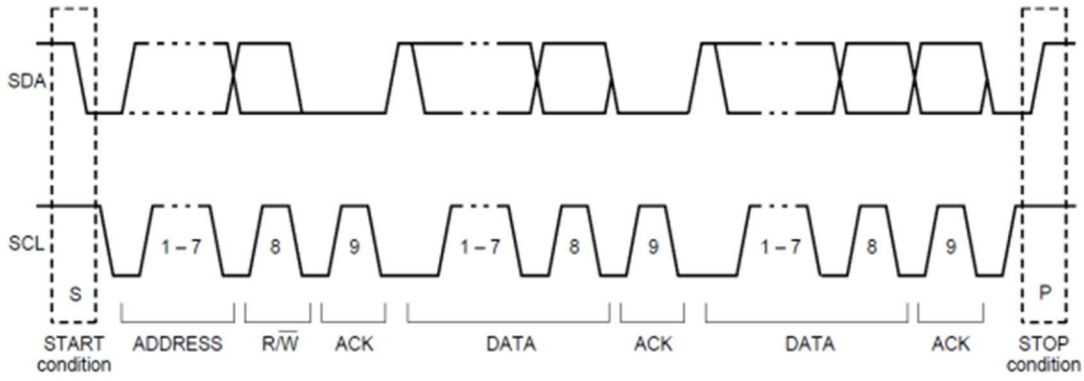


Figura 15. Trama básica del protocolo I^2C.

Este protocolo de transmisión de datos es síncrono por lo que se requiere de una señal de reloj constante entre el elemento maestro y el elemento esclavo. El dato que se quiere enviar se transmite mediante una sola línea de datos a la frecuencia que marca la señal de reloj para que el elemento que recibe la señal pueda interpretar la trama de datos enviada, este dato se requiere que se envíe con una trama específica. Cuando se definen los componentes que se utilizarán se puede consultar si existen tramas específicas para que la comunicación entre el dispositivo del fabricante y el FPGA sea la adecuada. El envío de datos se debe realizas con algunas reglas específicas del protocolo, como: las condiciones de START y STOP, el estado de ACK para la confirmación de lectura y estado lógico para lectura y escritura. Se puede observar una representación gráfica de cómo es el protocolo en la *Figura 15*.

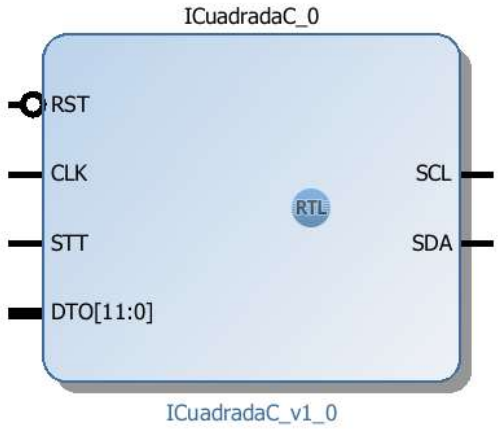


Figura 16. Bloque del protocolo I^2C.

El bloque que se desarrolló para la comunicación I^2C genérica en el FPGA es el mostrado en la *Figura 16,* dónde podemos observar un bit RST, equivalente a la señal que reinicia el bloque, es una señal que cuando detecta un cero realiza su función de reinicio, el bit CLK es la señal que se obtiene del cristal del FPGA, este bit gobernará la velocidad con la que se ejecuta cada instrucción, el bit STT es la señal que cuando se detecta en alto inicia el proceso de transmisión, aquí se usa un Timer para enviar pulsos a una frecuencia determinada para escribir los datos, y la señal DTO que son 12 bits en paralelo que representa el valor que se quiere enviar. Como salidas se tiene el SCL o Serial Clock, que envía una señal cuya función es sincronizar el proceso, y como caso especial tenemos la señal SDA o Serial Data que es una señal de entrada/salida, el dato se enviará y mediante esta vía también recibirá una respuesta del esclavo. El sistema principal pone la salida en alta impedancia, que es un estado donde el circuito maestro permite que otro sistema (El esclavo) utilice el canal para enviar su respuesta.

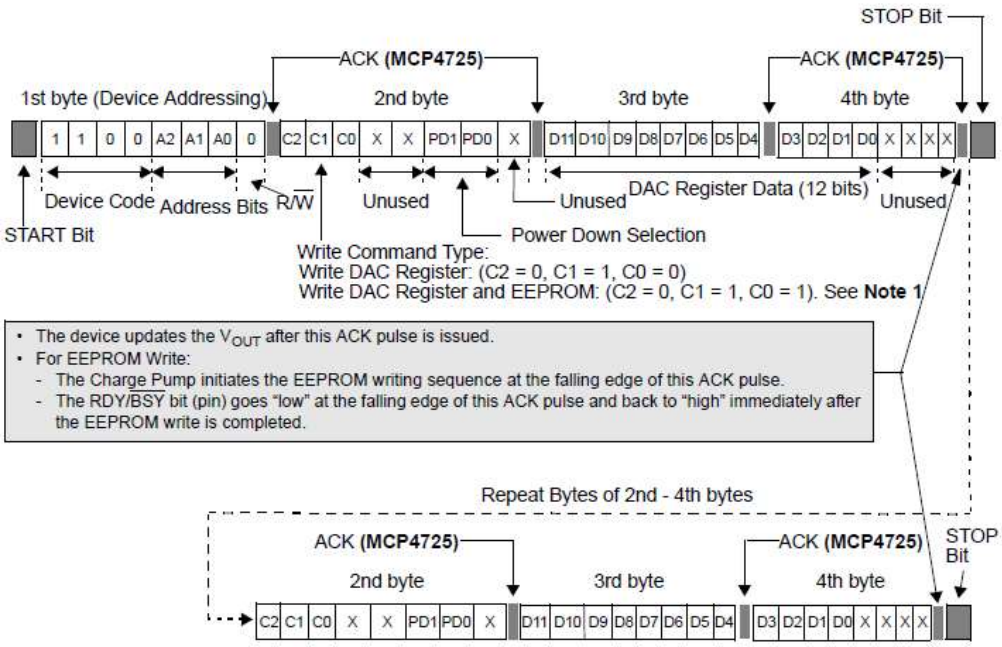


Figura 17. Trama del protocolo para el circuito MCP4725.

La *Figura 17* es la trama de datos que requiere en específico el circuito MCP4725 que es un DAC de 12 bits de resolución que entrega a la salida un valor máximo de 5v, pues este es el voltaje con el que se alimenta el circuito. La trama requiere de una dirección de esclavo la cual se puede conseguir en la hoja de datos del fabricante, remontándonos al Datasheet del MCP4725 obtenemos que A2 y A1 deben de estar en un estado lógico bajo ('0'), y A0 dependerá del estado lógico que nosotros deseemos, esto se hace con la finalidad de que si el usuario desea usar un segundo DAC de este tipo pueda controlar la salida de cada uno independiente del otro. Para este caso se ha decidido poner el bit A0 a un estado lógico bajo pues así lo entrega el fabricante, el último bit es el estado de lectura/escritura, como deseamos enviarle un dato para convertirlo en una salida analógica debemos escribir en el DAC, según la misma figura (*Figura 17*) el estado lógico para escribir es bajo, por lo tanto, el primer byte para enviar sería "11000000", seguido de una alta impedancia para que durante ese tiempo podamos recibir la respuesta del esclavo.

El segundo byte en la trama es de configuración para el DAC, la trama que a usar es "01000000", los primero 3 bits son el comando de escritura, como solo queremos escribir en el DAC un valor para que se refleje a la salida no hace falta guardarlo en la EEPROM, que es la memoria interna del DAC, por lo que únicamente se envía un "010". Los dos bits siguientes no importan en la trama por lo que se puede enviar cualquier valor, igual que el último bit; Los bits 2 y 3, de derecha a izquierda, son para activar el Power-Down mode, lo cual no necesitamos y por lo tanto van en cero. Después de enviar este byte se requiere que el canal SDA tenga un estado lógico de alta impedancia una vez más para recibir la respuesta del esclavo.

Los siguientes 12 bits equivalen al dato que queremos enviar como señal de control, estos se dividen en 2 partes, pos primeros 8 bits, los más significativos formarán un byte seguido de un lapso de alta impedancia, le continuarán los siguientes 4 bits menos significativos acompañados de 4 bits más, los cuales su valor no importa por lo que podemos enviar ceros. Para finalizar la transmisión se deja el canal en alta impedancia otra vez para recibir la respuesta del esclavo.

El esclavo responderá a cada bit con una señal en bajo si ha recibido correctamente la trama. Seguido de eso se enviará el estado de STOP que es una combinación del canal SCK y SDA.

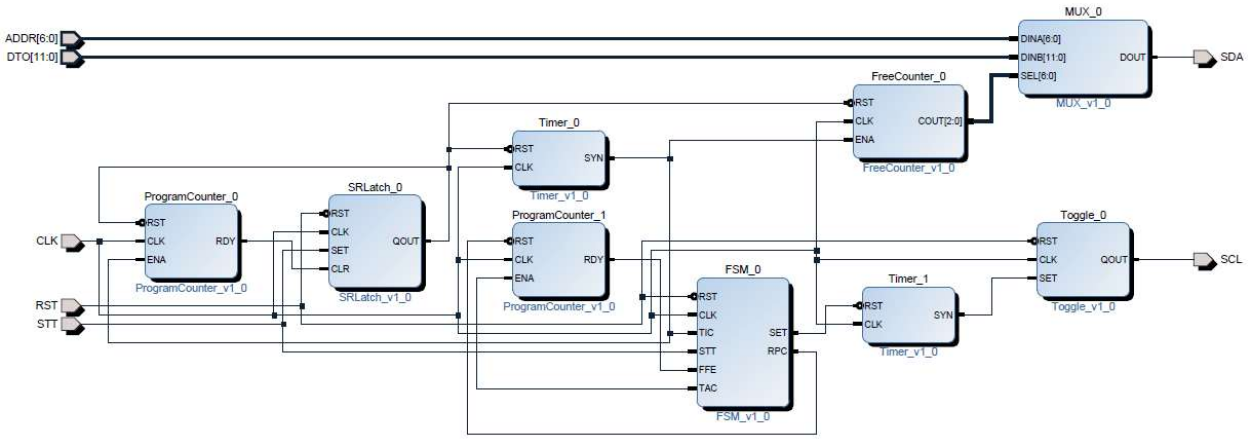


Figura 18. Diagrama a bloques del protocolo I^{2}C.

La máquina de estados (FSM_0) de la *Figura 18* es la que controla cómo funcionará el protocolo I^{2}C, ésta ejecuta una acción cada que recibe un pulso del Timer_0, envía un dato almacenado en el MUX_0 cada que se requiere para completar la trama antes mencionada. Hasta ahora el protocolo I^{2}C desarrollado solo escribe datos en el esclavo pero no puede leer datos, requerimos también que haga esta acción pues es necesario adquirir el valor de la aceleración que nos proporcionan los acelerómetros que instrumentan la edificación y con la cual los ingenieros civiles pueden realizar su estudio de manera adecuada.

Los acelerómetros elegidos para esta aplicación son los MPU-6050, los cuales pueden registrar la aceleración en los tres ejes con una resolución de 16bits por eje, con rangos de trabajo ajustables de ±2, 4, 6, 8 y 16g. La trama a enviar es la mostrada en la *Figura 19* según el Datasheet del instrumento.

Master	S	AD+W		RA		S	AD+R			ACK		NACK	P
Slave			ACK		ACK			ACK	DATA		DATA		

Figura 19. Trama del MPU-6050.

Como se puede notar, la trama tiene diferencias significativas incluso es más larga, pero trabaja con la misma base que tiene el MCP4725, por lo que se requiere de pequeñas modificaciones al diagrama a bloques de la *Figura 18* para poder funcionar en este otro dispositivo. La nueva trama comienza con la condición de START de la misma manera que en caso anterior, después, el primer byte será la dirección del esclavo seguida del bit de escritura, la dirección del acelerómetro es "110100X" siendo la X el estado al que se pone el pin A0 del circuito. Como en el caso anterior el último bit se pone en un estado lógico bajo que equivale a escritura, después el sistema debe poner el canal SDA en alta impedancia para permitirle al esclavo responder con un estado lógico en bajo (ACK). La trama continúa con un byte con la dirección del registro que se desea leer, para esto se requiere consultar los registros del acelerómetro para saber qué datos se desean obtener, en el caso particular de nuestra aplicación debemos leer la aceleración en un solo eje.

Recordemos que el acelerómetro que estamos usando nos entrega un valor de 16 bits, por lo que la lectura debe hacerse de dos registros, es complicado e innecesario realizar dos lecturas, una para cada registro, por lo que la configuración de la trama nos permite leer múltiples registros consecutivos. Por ejemplo, los registros de la aceleración en el eje Y son 0x3D para la parte alta y 0x3E para la parte baja. El byte del registro que usaremos es "00111101" que refiere a la parte alta del dato, una vez enviado el sistema debe esperar la respuesta del esclavo.

La trama de la *Figura 19* indica que se debe presentar, una vez más, la condición START, seguida de la dirección del esclavo pero acompañada del último bit en alto refiriendo a la lectura ("110100X1"), seguido de esto el sistema espera una respuesta del esclavo en bajo y comienza la lectura de los primeros 8 bits del dato, durante la lectura el canal SDA debe permanecer en alta impedancia para permitirle al esclavo usarlo para enviar los datos. Como ahora es el esclavo el que envía la información el sistema debe contestar al final de cada lectura con un estado en bajo, el esclavo continuará enviando los valores de los siguientes registros hasta que el sistema maestro responda con un valor en alto, o como la *Figura 19* lo llama, un ACK negado (NACK). Finalmente la trama termina con la condición STOP.

Toda esta nueva trama se debe configurar con la secuencia y valores anteriormente explicados en el bloque MUX_0 de la *Figura 18* así el sistema los enviará en el orden correcto. Una adición que se le hace al diagrama es un bloque llamado "*Registro Serial Paralelo*" que tiene la función de almacenar los datos que está leyendo el protocolo I^2C, los lee de forma serial pues la entrada es una sola línea dónde los valores cambian con el tiempo, para esto recibe una señal de un *Timer* que tiene la misma velocidad que el *Timer* que controla al protocolo. Una vez que almacenó la cantidad de valores marcada por un contador, el bloque presenta en su salida un arreglo de valores igual al recibido, ordenados desde el bit menos significativo hasta el más significativo, así podemos disponer de cada valor de aceleración para su manipulación y envío posterior a la interfaz gráfica.

3.2.2 Protocolo UART

Como se ha mencionado anteriormente, el proyecto consta de una interfaz gráfica en la cual el usuario puede controlar los movimientos del motor de distintas maneras al mismo tiempo que visualiza una gráfica de posición del motor y otra donde se despliega el valor de la aceleración que registran los sensores ubicados en distintas alturas de la estructura. El protocolo que se usa comúnmente para la comunicación entre la computadora y el FPGA es el UART, que es un protocolo muy sencillo capaz de enviar información binaria entre dos elementos a distintas velocidades configurables de transmisión.

UART (Universal Asynchronous Receiver-Transmitter) es un protocolo digital asíncrono, lo que significa que no requiere de una señal de reloj para transmitir la información, se basa en el cambio de estado de la línea de transmisión para poder comenzar a enviar y recibir datos. Solo cuenta con tres conexiones entre los elementos que se quieren comunicar, el primero es la referencia negativa o tierra (GND), a partir de la cual las otras dos líneas crearán una diferencia de potencial para poder transmitir valores en alto y bajo (binario), las otras dos líneas son de comunicación, una línea para transmitir y otra para recibir información (TX y RX, respectivamente).

El motivo por el cual el protocolo usa dos líneas de transmisión es por la sencillez de la trama, además de que la comunicación se puede hacer en cualquier instante en ambas vías por lo que la transmisión y la recepción pueden chocar si se encentran en la misma línea. El canal de Transmisión y recepción funcionan de la misma manera pero con diferente dirección, la trama se puede observar en la *Figura 20*, ambos tienen un estado de espera que es un valor alto en la línea, el bit de inicio es un valor bajo por lo que el cambio de estado representa el inicio de la transmisión o recepción. Como se mencionó antes, es posible configurar la velocidad de transmisión por lo que el tiempo que dura antes de enviar el siguiente valor se denomina *tiempo bit* y es inversamente proporcional al *Baud Rate*.

Cuando llega el cambio de estado que representa el inicio de la transmisión comienza el tiempo bit, cuando este se cumple la línea presenta el cambio al valor del primer bit, que es el bit menos significativo del byte a enviar, después de cada tiempo bit se envía un nuevo valor hasta completar los 8 bits, posteriormente se envía un bit de paridad. La paridad es una forma de detección de errores, la cual usa los datos enviados para calcular un valor de un bit representativo de la cantidad de valores en alto que se enviaron, esta puede ser par o impar, esto se configura previamente y tienen que coincidir en ambos elementos que se desean comunicar.

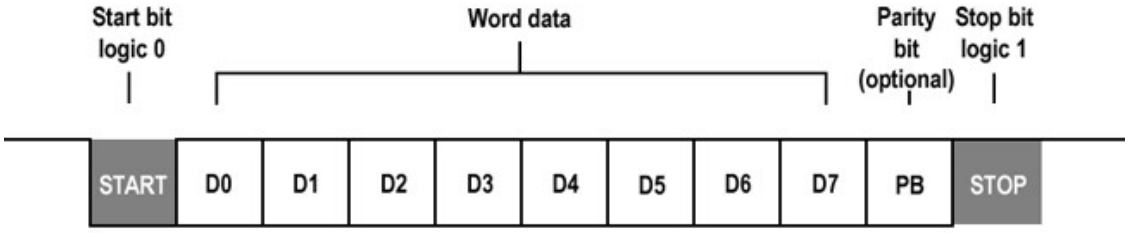


Figura 20. Trama del protocolo UART.

La recepción de un valor mediante el protocolo UART lleva una lógica muy sencilla, se necesita que ambos elementos involucrados en la comunicación estén configurados con la misma información: Baud Rate, número de bits y tipo de paridad, con esta información podemos recibir el dato enviado. El sistema de recepción debe estar atento al canal por el que recibe la información (RX), este canal presentará un cambio de estado de alto a bajo cuando la transmisión comience, en ese instante debemos dejar pasar la mitad del tiempo bit, esto permite situarnos a la mitad de un bit en un tiempo específico para poder muestrear el valor y no en una transición donde el valor puede ser incierto (*Figura 21*). El muestreo se hará a partir de ahí cada tiempo bit para obtener todos los valores de la trama, sabemos de antemano cuantos valores se están enviando por lo que detendremos el muestreo cuando obtengamos el número de valores que esperamos para después procesarlos, mientras que el UART regresa al estado inicial de espera para recibir otro byte de información.

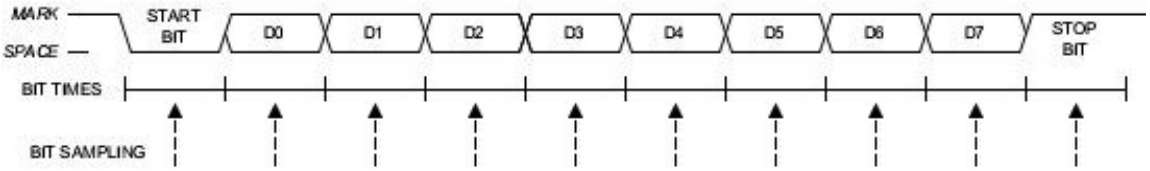


Figura 21. Recepción de datos.

La recepción de los datos es de forma serial, por lo que debemos, como en el protocolo anterior, usar un bloque llamado *Registro Serial Paralelo* para convertir el dato en un arreglo en paralelo del tamaño de la información enviada, típicamente 1 byte. El bit de la paridad se deja fuera del arreglo, pues debemos compararlo con un nuevo cálculo de la paridad para el byte recibido, si ambos valores coinciden significa que nuestro valor fue enviado correctamente y podemos hacer uso de el en el sistema, de lo contrario el valor se descarta.

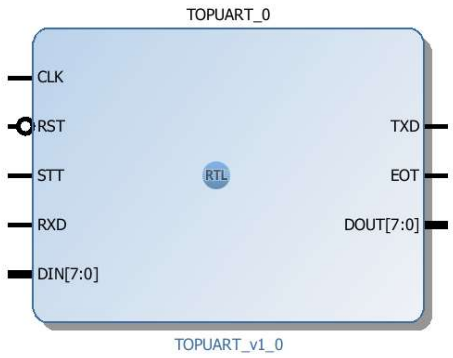


Figura 22. Bloque UART.

El bloque del protocolo UART implementado en la tarjeta de desarrollo FPGA se muestra en la *Figura 22* donde del lado izquierdo se observan las entradas y del derecho las salidas. CLK es la entrada del reloj de la tarjeta y RST es el reset maestro de la tarjeta, esto significa que el bloque funciona cuando la tarjeta está encendida y activa. La entrada STT se activa con un pulso en alto y es la instrucción que necesita para enviar el byte que esté a la entrada DIN que es un arreglo de 8 bits. RXD es el canal por el cual recibirá una trama, mientras que la salida TXD es el canal por el cual enviará la información. La salida EOT envía un pulso en alto cuando finaliza la recepción de un dato y éste se encuentre disponible en la salida DOUT que es un arreglo de 8 bits.

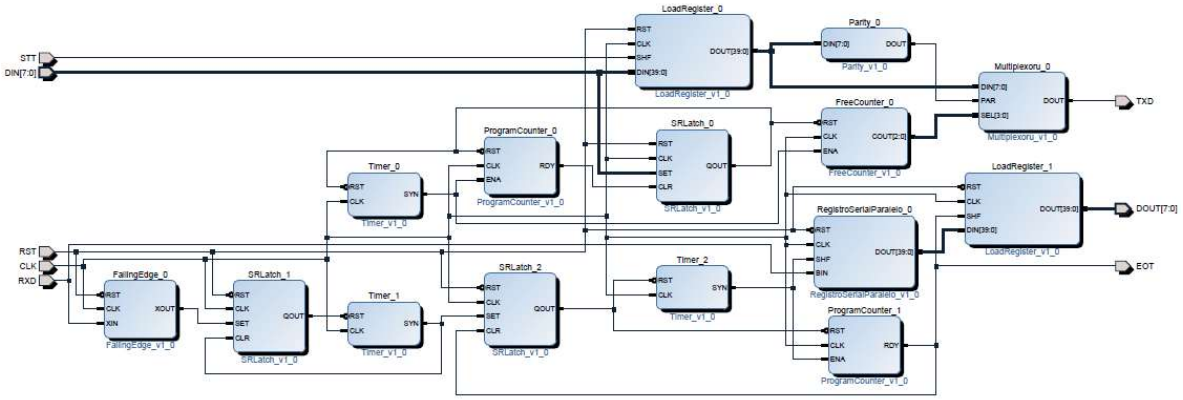


Figura 23. Composición del protocolo UART.

La composición interna del bloque UART se puede observar en la *Figura 23* dónde observamos que en su mayoría está compuesta de bloques genéricos. El *Timer_0* y el *Timer_2* deben de tener configurado el valor para cumplir el *Baud Rate*, mientras que el *Timer_1* tiene configurado la mitad del valor del *Baud Rate*, para poder leer los datos como se explicó anteriormente. El *Multiplexoru_0* se encarga de enviar los datos con la secuencia correcta y cambia de dato cada que el *FreeCounter_0* le indica que cambie, éste último bloque tiene configurado la cantidad máxima de cambios que se requieren para enviar por

completo la trama, y se reinicia terminando para estar listo para cuando se requiera enviar de nuevo otro valor.

Como se observa, el protocolo solo es capaz de enviar o recibir un dato de 8 bits por trama, pero es necesario enviar y recibir más datos. Para la transmisión lo único que se requiere es utilizar un multiplexor con la cantidad de datos que se requieren enviar junto a un contador que active el envío de datos y recorra los valores del multiplexor para enviar todos, esto es un arreglo externo al bloque, pues no afecta el uso del UART, más bien lo repite consecutivamente las veces necesarias. En cuanto a la recepción de los datos, lo que se tiene que hacer es utilizar un registro que almacene la cantidad de datos a recibir, y los divida en bytes para poder hacer uso de ellos, ya sabemos que el bloque UART tiene una salida (EOT) que indica cuando un dato ha llegado.

3.2.3 Cuadratura del encoder

Dentro de un sistema de *control de lazo cerrado* se encuentra la parte de la instrumentación del actuador que nos permite conocer el estado actual de éste, en el caso particular del proyecto del *Generador de Vibraciones* nuestro actuador es un motor, nuestro control es de posición por lo que la instrumentación debe tener como salida la posición angular del rotor, como se comentó en el *Capítulo III.1.3 El Encoder* conocimos un poco sobre el funcionamiento del encoder incremental, que es el encoder que se usará para este proyecto.

La unidad de control requiere conocer en tiempo real la posición en la que se encuentra el rotor de nuestro BLDCM por lo que tiene la tarea de interpretar las salidas A y B del encoder antes mencionadas. Sabemos que las salidas del encoder están desfasadas 90° con el objetivo de que se pueda diferenciar cuando gira en un sentido o en el otro.

En la *Figura 24* podemos observar gráficamente el comportamiento de las señales en ambas direcciones con el paso del tiempo. Para que la tarjeta pueda interpretar el encoder se diseñó un bloque llamado *QuadratureDecoder* el cual tiene como entradas las señales A y B provenientes del encoder y como salidas tiene un bit de dirección y uno más que indica el cambio de estado. El bloque, internamente, almacena el estado de las entradas en cuatro distintos lapsos de tiempo (Los lapsos que almacena es cuando la entrada ha cambiado), esto permite que el bloque tenga la tabla que se muestra en el lado derecho de la *Figura 24*.

Los estados son almacenados en un arreglo de 4 bits dónde el bit más significativo es el bit más antiguo. Ya que se han almacenado los cuatro estados, el bloque determina la dirección mediante una compuerta XNOR con entradas B[3] y A[2] (En la *Figura 24* se

pueden observar como B_{t1} y A_{t2}). Cuando el encoder cambia de estado los valores de las tablas se recorren, como se puede notar, cuando gira en dirección de las manecillas del reloj (Clockwise) la salida de la compuerta siempre será un “0”, mientras que en el caso contrario (Counter-Clockwise) la salida de la compuerta siempre será un “1”.

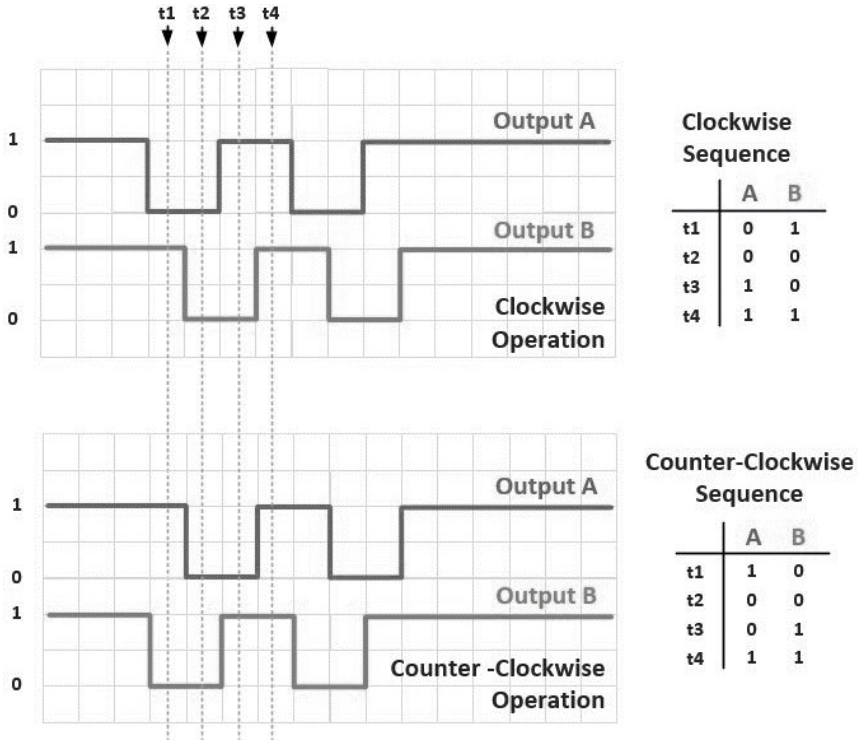


Figura 24. Secuencia del encoder.

Éste bit de dirección sale del *QuadratureDecoder* hacía un contador bidireccional el cual suma o resta un valor de acuerdo al sentido de giro. El bit que activa el conteo en el contador también sale del *QuadratureDecoder*, éste bit está en alto cada que los valores en las entradas A y B cambian, esto se logra con una compuerta XOR aplicada a los valores B[3], B[2], A[3] y A[2]; lo que se busca es comparar un valor con su consecutivo, siempre que sean diferentes la salida tendrá un “1”, pero si son iguales, significa que no cambió o que está cambando de sentido, por lo que a la salida se tendrá un “0”. Nótese que cambiará cuatro veces por ciclo, justo esto es la explicación del por qué el conteo del encoder se multiplica por 4.

3.2.4 Controlador

El controlador es la parte importante de un servomotor pues este toma el valor deseado y el valor actual de la posición para que a partir de su diferencia se pueda obtener, mediante métodos matemáticos, una señal de corrección de posición que, como ya se mencionó antes, arroja una señal que es transformada en voltaje para mover el motor.

En el *Capítulo III.1.5 Unidad de control* se habló de lo que se conoce como *control proporcional de posición* que, como se dijo, solo utiliza una constante que multiplica al error. En la *Figura 25* podemos observar la forma de la respuesta que podemos esperar. La forma con mayor oscilación representa un control con una *Kp* grande, mientras que la forma más amortiguada representa un control con una *Kp* pequeña. Mientras que la forma que se encuentra entre las dos sería una respuesta ideal para nuestro sistema. Sin embargo, no es posible que dicha forma se pueda conseguir con un controlador sencillo como es el *control proporcional de posición*.

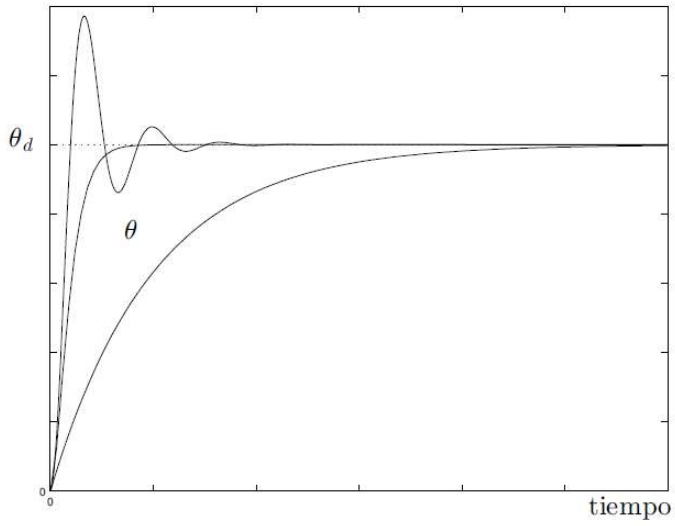


Figura 25. Formas de la posible respuesta transitoria de un control de posición.

Dicho lo anterior, se infiere que es necesario un controlador más robusto para obtener la respuesta ideal que se observa en la *Figura 25*. Existen otras dos variables comúnmente usadas en los controladores, que ayudan a tener respuestas más controladas. El controlador más común usado es el *control PID de posición (Controlador Proporcional-Integral-Derivativo de posición)*. Matemáticamente se puede ver el controlador en la forma que describe la *Ecuación 9.*

$$u = k_p e + k_d \frac{de}{dt} + k_i \int_0^t e(r)\, dr \qquad (9)$$

Donde $e = \theta d - \theta$. Rápidamente podemos ubicar donde se encuentra el *controlador P* que ya conocíamos, y podemos inferir como se adhieren las otras dos partes del controlador. Una explicación muy general de que hace cada parte del controlador sería la siguiente:

La constante *Kp*, equivalente a la parte proporcional de controlador, tiene como principal objetivo lograr que la respuesta a una entrada escalón sea más rápida o más lenta dependiendo de si valor. La constante *Kd*, equivalente a la parte derivativa del controlador, se encarga de amortiguar la respuesta del controlador para evitar oscilaciones. Por último, la constante *Ki*, representante de la parte integral del controlador, se encarga de lograr que el error de estado estacionario sea cero, el error de estado estacionario se presenta cuando se tiene una perturbación externa constante.

Dicho lo anterior y como se mencionó en el *Capítulo III.1.5 Unidad de control* el error de estado estacionario es cero en esta aplicación ya que se trabaja sobre el eje vertical, haciendo que la perturbación más común que es la gravedad no afecte al sistema, por lo que podemos concluir que no es necesario agregar la parte integral del controlador a nuestro sistema para lograr una respuesta como la que requerimos. De esta manera tenemos que el controlador necesario para esta aplicación es un *control de posición PD o Proporcional-Derivativo*.

El control PD tiene la siguiente forma:

$$u = k_p e + k_d \frac{de}{dt} \quad Donde\ e = \theta d - \theta \tag{10}$$

Usando la transformada de Laplace en la *Ecuación 10* obtenemos:

$$U(s) = \left(k_p + k_d s\right) E(s) \tag{11}$$

Esta forma es una función dependiente del tiempo por lo que no puede ser implementado directamente en el FPGA. Por lo tanto, se deben tomar en cuenta ciertos cambios basados en un tiempo de muestreo que llamaremos T_s por lo que vamos a considerar lo siguiente y lo sustituiremos en la *Ecuación 10* para obtener la *Ecuación 12*..

$$\frac{de(t)}{dt} \approx \frac{e(k) - e(k-1)}{Ts} \qquad k_d = k_p Td$$

$$u(k) = k_p \left\{ e(k) + \frac{Td}{Ts} [e(k) - e(k-1)] \right\} \tag{12}$$

Definimos el termino $\Delta u(k)$ que deberá ser calculado en lugar de $u(k)$ para ahorrar recursos.

$$\Delta u(k) = u(k) - u(k-1) \tag{13}$$

$$u(k-1) = k_p\left\{e(k-1) + \frac{Td}{Ts}[e(k-1) - e(k-2)]\right\} \quad (14)$$

Sustituyendo las *Ecuaciones 12* y *14* en la *Ecuación 13* obtenemos:

$$\Delta u(k) = k_p\left\{e(k) - e(k-1) + \frac{Td}{Ts}[e(k) - 2e(k-1) - e(k-2)]\right\} \quad (15)$$

De la *Ecuación 13* despejamos $u(k)$:

$$u(k) = \Delta u(k) + u(k-1) \quad (16)$$

Sustituimos en la *Ecuación 16* lo obtenido en la *Ecuación 15* y agrupamos un poco los valores para simplificar obteniendo:

$$u(k) = k_p\left\{\left(1 + \frac{Td}{Ts}\right)e(k) - \left(1 + 2\frac{Td}{Ts}\right)e(k-1) + \left(\frac{Td}{Ts}\right)e(k-2)\right\} + u(k-1) \quad (17)$$

Podemos ver la ecuación anterior como:

$$u(k) = q_0 e(k) + q_1 e(k-1) + q_2 e(k-2) + u(k-1) \quad (18)$$

$$Donde\ q_0 = k_p\left(1 + \frac{Td}{Ts}\right), q_1 = -k_p\left(1 + 2\frac{Td}{Ts}\right), q_2 = k_p\frac{Td}{Ts}$$

La *Ecuación 18* es mucho más fácil de interpretar e implementar en un FPGA pues de manera general se puede describir como una constante q_1 multiplicado por el valor del error en el instante actual, más una constante q_2 multiplicando al error en un instante anterior, más la constante q_3 multiplicada por el error de hace dos instantes, toso esto sumado a un cálculo similar anterior ($u(k-1)$). Las ganancias q_0, q_1 y q_2 se calculan a partir de los valores de las constantes k_p, k_d y T_s.

El diagrama a bloques desarrollado para la implementación de este controlador en el FPGA se puede observar en la *Figura 26*. Ahí podemos observar la entrada de las ganancias que previamente de debieron de haber calculado con un formato de 8.8, esto quiere decir que se usará 8 bits para la parte entera y otros 8 para la parte decimal. T_s es una entrada al controlador que viene de un *Timer* en el cual se programa el tiempo de muestreo requerido. X_{in} es la entrada del *Error* al controlador.

Nótese que entra un valor de 32 bits como *error* y sale una *señal de corrección* de 48 bits, esto pasa debido a la multiplicación del *error* con las *ganancias* de 16 bits.

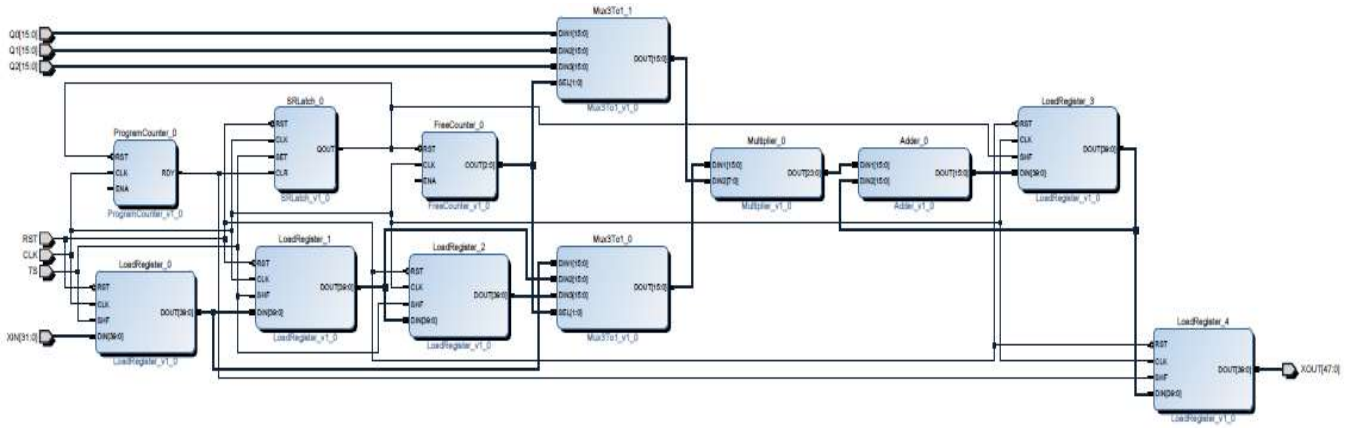


Figura 26. Diagrama a bloques del controlador PD.

3.2.5 Saturador

La señal de corrección, como salida del controlador, es de 48 bits. Como se señaló en el *Capítulo III.2.1 Protocolo I^2C* el *DAC MCP4725* recibe una señal de 12 bits, los cuales deben de salir de la señal de corrección. El bloque de *Saturador* implementado en el *FPGA* tiene la tarea de recortar el valor de 48bits a un valor equivalente de 12 bits.

Para poder recortar significativamente el valor de salida hasta los 12 bits necesarios es común que se analice primero como requerimos que sea el control de nuestro motor, para este caso es necesario que el motor responda rápidamente y la precisión milimétrica no es tan necesaria, por lo tanto, podemos comenzar con el hecho de que podemos descartar los valores decimales. Como se mencionó en el capítulo anterior, se trabaja con un formato 8.8 para las constantes, de esta manera podemos recortar los primeros 8 valores y trabajar únicamente con los valores enteros menos significativos.

El valor con el que se trabajará para la corrección son los 12 bits que comprenden del *bit 8* hasta el *bit 19*. En el *Capítulo III.1.2.1 Acoplamiento de señales* mencionamos que para que el motor esté estático debemos enviarle *2.5v* a la salida del *DAC*, si se envía el valor inmediatamente al *DAC* este registrará valores cercanos a cero, por lo que l motor siempre se estará moviendo a velocidad máxima en sentido anti horario, así que el *Saturador* nos puede ayudar a corregir eso sumándole a la salida un valor cercano a *0d2047* (decimal), que es la mitad del valor máximo alcanzado con 12 bits. Concluyendo, el valor a sumar a la salida del recorte de bits hechos por el *bloque de saturación* será *0b011111111111.* Aunque el número puede variar un poco debido al funcionamiento de la parte eléctrica.

3.3 Interfaz gráfica

El sistema de control de posición del motor requiere una interfaz en la cual el usuario pueda manipular la posición deseada del motor, que mediante el acoplamiento se transforma en la posición de la plataforma. Se requiere que la interfaz gráfica pueda comunicarse con la tarjeta de control mediante el protocolo de comunicación UART, y enviar el dato de la posición en un formato binario para que el sistema pueda procesarlo y generar una acción de control que lleve a la plataforma a la posición que se requiere.

Existen múltiples programas que permiten desarrollar interfaces usuario-máquina, una de las más comunes es el software LabView de National Instruments, el cual tiene un lenguaje de programación en bloques, muy sencillo y fácil de usar.

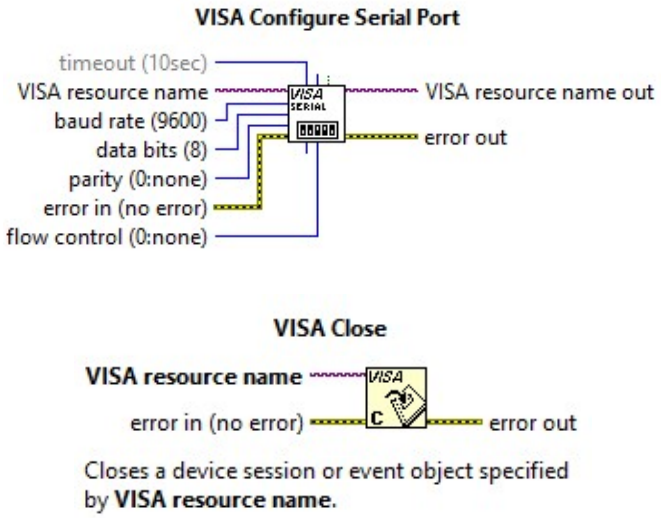


Figura 27. Bloque de configuración Serial y bloque de fin de comunicación.

Para comenzar con la programación de una interfaz gráfica en LabView para el *Sistema Generador de Vibraciones* se debe configurar el protocolo UART para comenzar a enviar y recibir datos con la tarjeta de control. Se requiere el uso del bloque *VISA Configure Serial Port* para hacer uso de uno de los puertos de nuestra computadora para la interfaz, en este bloque configuramos el *Baud Rate*, *la paridad*, el número de bits y un valor en tiempo el cual indica cuanto esperará el sistema a un dato antes de terminar con la conexión. Algunos de estos conceptos fueron mencionados en el *Capítulo III.2.2 Protocolo UART*. Al terminar de usar el puerto debemos liberarlo, para esto se tiene un bloque para terminar con la comunicación, este bloque también se puede observar en la *Figura 27* y se conecta al final del programa.

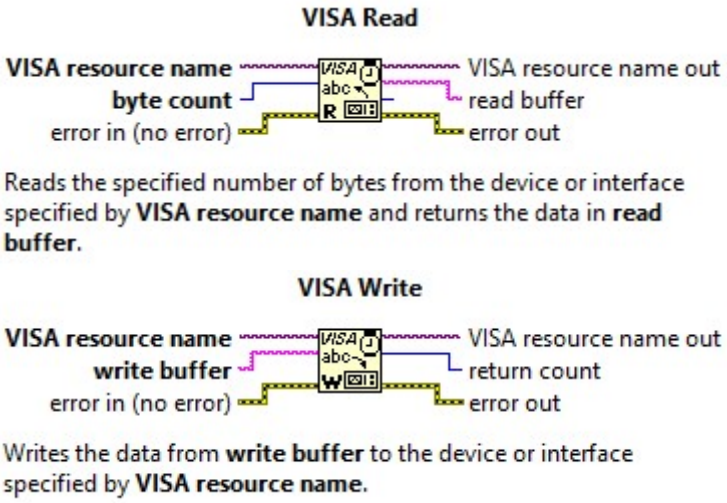


Figura 28. Bloques de la lectura y escritura.

La lectura de los datos se realiza mediante un bloque conectado en paralelo a los anteriormente mencionados. En este bloque debemos configurar la cantidad de bytes que va a recibir, esto para tenerlos en un solo arreglo de datos; cabe mencionar que el bloque está en uso constante, esto significa que siempre está atento a lo que tiene en la entrada. Por otra parte el bloque de envío de datos estará enviando constantemente el dato en su entrada, por lo que se necesita introducirlo dentro de una sentencia, así cuando la entrada booleana de la sentencia sea verdadera el dato se enviará, de lo contrario no se ejecuta ninguna acción. El detalle de los bloques se puede observar en la *Figura 28* mientras que la conexión completa del sistema está en la *Figura 29*.

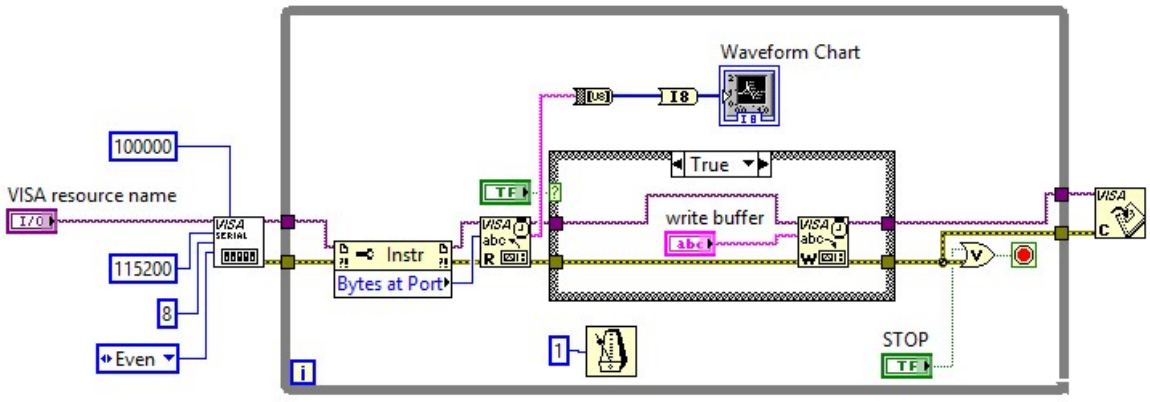


Figura 29. Programa a bloques de una interfaz de comunicación.

El programa, como se puede observar, se encuentra dentro de un loop infinito para asegurar que éste se ejecute indefinidamente hasta que el usuario decida detenerlo. El bloque de configuración del puerto se encuentra fuera de este loop debido a que solo se

configura una sola vez al inicio, de igual manera, el bloque de final se encuentra fuera pues solo se detiene una vez. Este programa toma el o los valores de la entrada y los grafica inmediatamente, existe un pequeño retraso entre el verdadero movimiento y la gráfica pero este se puede despreciar pues es muy pequeño. Se puede notar que el bloque de escritura está en una sentencia, como se mencionó antes, cuyo valor booleano es un botón en la interfaz gráfica (*Figura 30*); La entrada del bloque es un cuadro de escritura en la interfaz, éste está configurado para recibir un valor en hexadecimal debido a que si se recibe un valor decimal el programa toma los caracteres como ASCII.

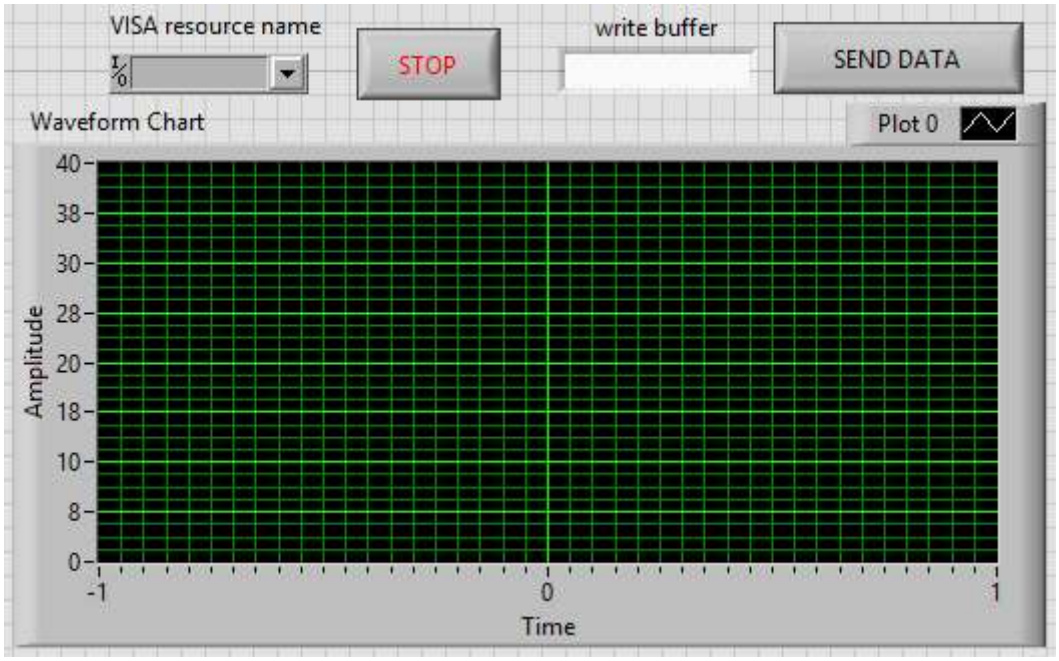


Figura 30. Interfaz gráfica del programa desarrollado.

Esta es una interfaz gráfica sencilla que representa la base para cualquiera que se desee desarrollar, a partir de esto se puede configurar para recibir más datos como lo serían las aceleraciones de los instrumentos en la estructura y poder enviar datos de configuración como la amplitud y frecuencia de una señal senoidal.

CAPÍTULO 4: RESULTADOS Y DISCUSIÓN

4.1 Resultados

Lo primero que se requiere hacer, para comenzar con el desarrollo de un prototipo funcional de una mesa vibratoria, es definir los componentes más importantes, estos son los componentes mecánicos, eléctricos y electrónicos. Algunos de los componentes ya fueron explicados a lo largo del trabajo por lo que solo se abordarán las características más importantes.

Componente	Características
Brushless DC Motor BLK42 (Anaheim Automation)	• Motor tamaño NEMA42 • Retroalimentación de Efecto Hall • Voltaje DC máximo: $170v$ • Velocidad Máxima: $3000rpm$ • Torque: $708oz.in$ • Peso: $15.21lbs$ • Longitud: $170.94mm$ • Trifásico
Servo Drive AB20A200 (Advanced Motion Control)	• Corriente pico: $20A$ • Corriente continua: $12A$ • Suministro de voltaje (DC): $40 - 175v$ • Diseñado para altas frecuencias de cambio • Modos de operación: Corriente, velocidad y lazo abierto • Suministro de control: $\pm 10v$
Guias de carro lineales SRS15M (THK)	• Peso: $0.047kg$ • Carga máxima: $6.6kN$

Tornillo de bolas BTK1605-2.6 (THK)	• $2\ vueltas/cm$ • Momento: $5.5kg * cm^2/mm$
Encoder HEDS5500 (AVAGO)	• 2 Canales de cuadratura • $500\ cuentas\ /\ rev$ • Suministro: $5v$ • Encoder incremental • Velocidad: $30000rpm$ • Aceleración: $250000rad/seg^2$ • Vibración: 5g, 5-1000Hz
DAC MCP4725 (Microchip)	• 12 bits de resolución • Memoria EEPROM • Pin de dirección A0 externo • Voltaje de operación: $2.7 - 5.5v$ • Interfaz I^2C
Acelerómetro MPU-6050 (InvenSense)	• Aceleròmetro de 3 ejes • Protocolo I^2C • ADC interno de 16bits • Rangos: $\pm 2g$, $\pm 4g$, $\pm 8g$ y $\pm 16g$ • Pin de dirección AD0 externo

Tabla 2. Listado de componentes.

Para comenzar con la construcción del prototipo de mesa vibratoria debemos definir que componentes generales se deben ir desarrollando, para que a partir de múltiples partes podamos lograr un prototipo funcional, por eso se desarrolló el diagrama a bloques de la *Figura 31* el cual tiene 5 bloques que conforman el proyecto, cada uno de ellos se irá desarrollando a continuación, y la unión de todos ellos comprenderá el prototipo del *Sistema Generador de Vibraciones.*

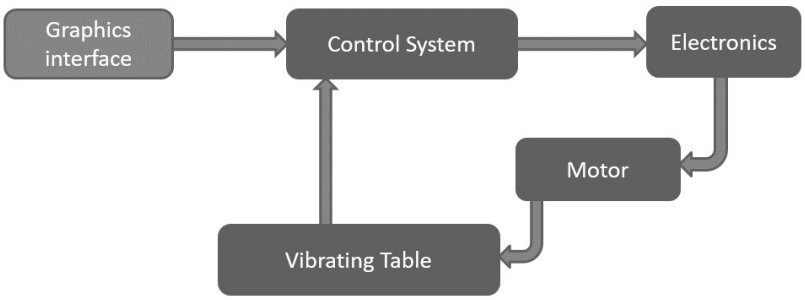


Figura 31. Diagrama a bloques del sistema generador de vibraciones.

Comenzando con la parte física o como la *Figura 31* lo llama "Vibrating Table" debemos construir la base del proyecto. A partir de los componentes anteriormente descritos podemos deducir las características a las que hacen referencia las *Ecuaciones 1, 5, 6* y *7.* Esta información es de utilidad en la programación del movimiento.

$$Resolución = \frac{5v}{2^{12}} = 1.22mv/bit \tag{19}$$

$$Presición = \frac{360°}{500*4} = 0.18° \tag{20}$$

$$Cuentas\ por\ vuelta = 500 * 4 = \frac{2000cuentas}{rev} \tag{21}$$

$$Cuentas\ por\ unidad = \frac{2000cuentas}{rev} * \frac{2rev}{1cm} = 4000cuentas/cm \tag{22}$$

También, podemos comenzar con el dimensionamiento del prototipo mediante el diseño asistido por computadora. Así que, a partir de los componentes antes mencionados se diseñó en el Software SolidWorks la estructura de la mesa junto a los componentes dimensionados para su previa construcción.

Figura 32. Diseño conceptual prototipo de la mesa vibratoria.

Figura 33. Vista superior del diseño conceptual (Vista de los componentes).

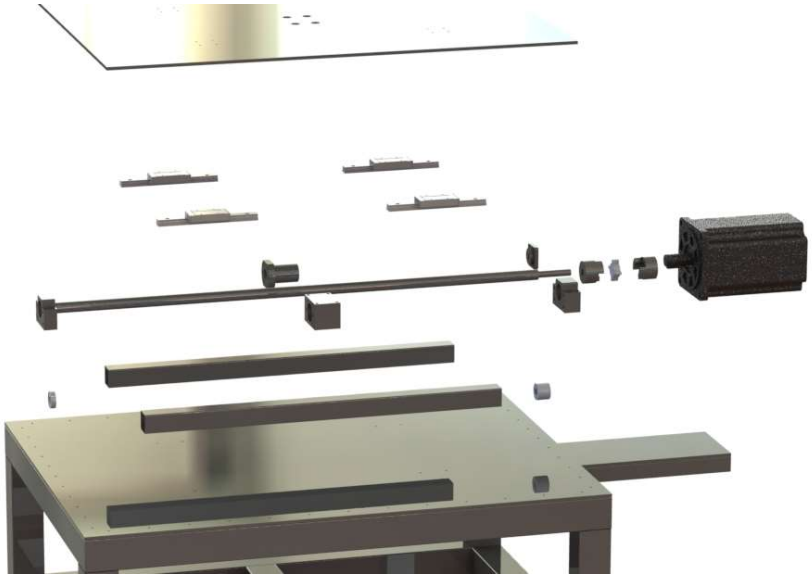


Figura 34. Vista explosionada de los componentes.

En la *Figura 32* se observa una vista general del diseño creativo de la mesa vibratoria, mientras que en la *Figura 35* podemos observar que el prototipo real quedó muy similar a lo planteado en el diseño. Las *Figuras 33* y *34* nos muestran una vista a detalle de los componentes mecánicos que soportan la plataforma y permiten el movimiento de ésta, en la construcción final hubo algunos de los cambios, los más significativos fueron el largo del tornillo de bolas, y la posición final del encoder, el cual se planeaba dentro del motor, pero al final quedó al final de la flecha.

Se podrán encontrar más imágenes, así como el diseño CAD acotado dentro de los anexos al final del trabajo. Esto con el objetivo de que su recreación sea lo más sencilla posible.

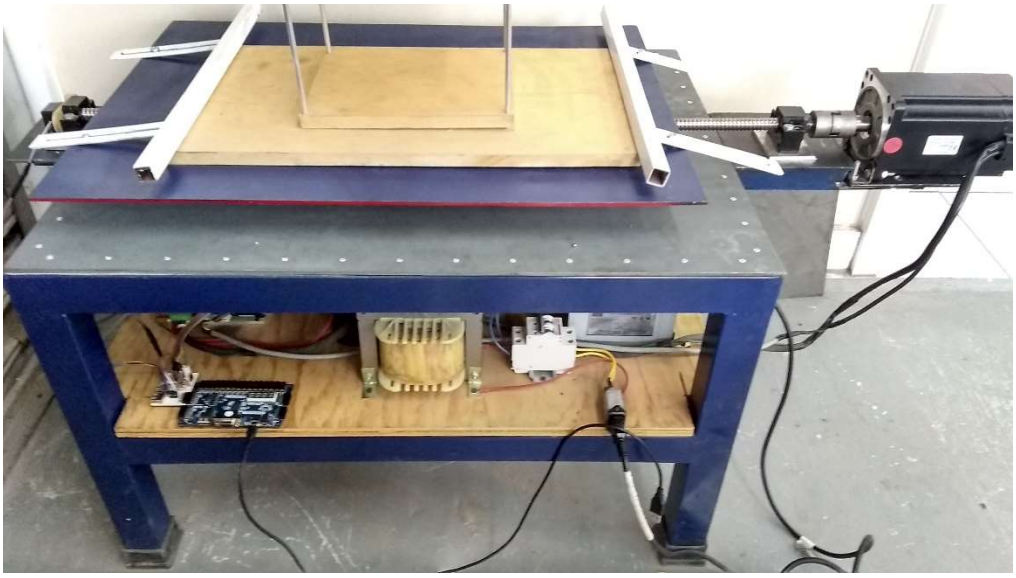


Figura 35. Vista general de la estructura del prototipo.

Al tener la parte física completamente funcional podemos hacer una prueba de movimiento para determinar los límites de desplazamiento de la mesa. Al analizar es posible obtener un movimiento de hasta *10cm*. Este es el valor deseado para la mesa, y así se diseñó y fabricó, pero por cuestiones de seguridad de los mecanismos recortaremos el desplazamiento hasta *7cm* para evitar los fallos mecánicos. Bajo este supuesto podemos calcular con la *Ecuación 8* el valor de cuentas que equivalen al total de nuestro desplazamiento. Así obtenemos que:

$$Cuentas\ Totales = \frac{4000 cuentas}{cm} * 7cm = 28000\ cuentas \qquad (23)$$

El siguiente paso, después de tener la parte mecánica ya ensamblada, es revisar la electrónica de potencia que le suministrará energía al Servo Drive para activar el Motor. Como se observó en la *Tabla 2*, se necesita una fuente de alimentación de voltaje de entre 40 y $175v$ de corriente directa, por lo que se implementó un circuito rectificador para que la línea de tensión general sea el suministro que el prototipo necesita.

En la *Figura 36* podemos observar el circuito de potencia que se utilizó, dónde teóricamente se tiene a la salida un voltaje de $89.8v$ de corriente directa, obtenidos de la línea general de suministro de voltaje de $127v$ de corriente alterna a una frecuencia de 60Hz. En la práctica, la salida del circuito no es tan distinta del valor teórico pues tenemos un voltaje de $83.4v$ *de corriente directa* con el cual estamos alimentando el Servo Drive que controla al motor.

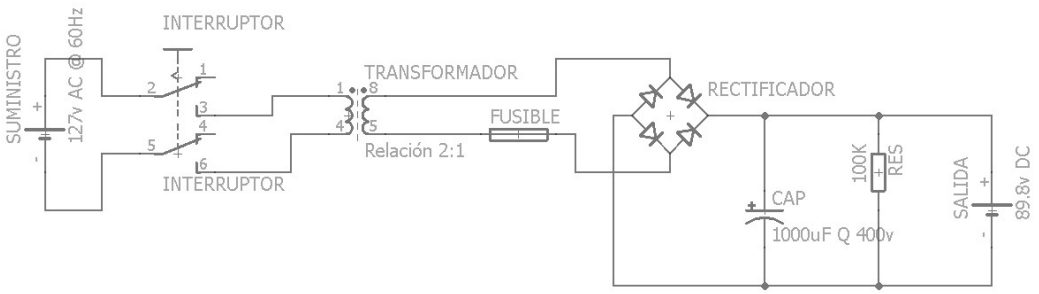


Figura 36. Circuito de potencia que suministra energía al Servo Drive.

Una vez que el motor está alimentado es necesario que elaboremos el circuito con el que la referencia de $\pm 10v$ será suministrada a la entrada del Servo Drive, como se comentó en el *Capítulo III.1.2.1 Acoplamiento de señales* se requiere la implementación del circuito de la *Figura 10*, con esto se podrá controlar la velocidad y la dirección en la que gira la flecha del motor y por ende la plataforma del equipo. Este circuito trabaja con *Amplificadores operacionales* por lo cual es necesario tener un suministro de voltaje de $\pm 15v$ para alimentar estos circuitos que vienen en un circuito integrado llamado *TL081*.

Así que se recurrió al uso de una fuente de alimentación como las que usan las computadoras. Esta fuente de voltaje independiente nos suministrará el voltaje necesario para alimentar nuestro circuito de acoplamiento además de ser capaz de entregar 5V para la alimentación del encoder, acelerómetros y DAC, quitándole esas cargas a la tarjeta Basys 3, lo único que tenemos que tomar en cuenta es que debemos referenciar ambas cosas, lo que significa que debemos tener una tierra en común para ambos dispositivos.

Comenzado con la implementación de código en el FPGA, debemos iniciar con la interfaz I^2C a través de la cual la tarjeta se comunica con el circuito integrado *MCP4725* que enviará el valor de voltaje al circuito acoplador antes visto. La trama que se tiene que seguir se puede observar en la *Figura 17* en el *Capítulo III.2.1 Protocolo I^2C*, a partir de eso comenzamos a desarrollar un diagrama a bloques que posteriormente se convertirá en un código implementado en el FPGA. La *Figura 18* muestra el diagrama a bloques del protocolo I^2C, mientas que la *Figura 36* muestra los componentes usados en el *TOP* del bloque *ICuadrdaC*.

```
begin
    SYN <= NOT SNN;
    SCL <= SYN;
    FE <= TAC AND SYN;
    U01 : FSM port map(RST, CLK, TIC, STT, FFE, TAC, SET, RPC);
    U02 : SRLatch port map(RST, CLK, STT, CLS, RTP);
    U03 : Timer generic map(500) port map(RTP, CLK, TIC);
    U04 : ProgramCounter generic map(83) port map(RTP, CLK, TIC, CLS);
    U05 : Timer generic map(500) port map(SET, CLK, TAC);
    U06 : Toggle port map(RST, CLK, TAC, SNN);
    U07 : ProgramCounter generic map(9) port map(RPC, CLK, FE, FFE);
    U08 : FreeCounter generic map(6) port map(RTP, CLK, TIC, SEL);
    U09 : MUX port map("1100000", DTO, SEL, SDA);
end TOP;
```

Figura 37. Componentes del bloque ICuadradaC.

El bloque *MUX_0* contiene el formato de la trama tal y como la requiere el circuito para funcionar, el hecho de que esté correctamente acomodada facilita la transmisión pues solo hay que variar la salida manipulando el selector a la velocidad que se necesita transmitir. Al bloque *MUX_0* lo complementan dos entradas que son la dirección del esclavo, y el valor a enviar.

La dirección del esclavo para el MCP4725 es "*110000X*", dónde la *X* representa el valor de A0 en el circuito, normalmente este bit viene por default a *GND* lo que representa un cero lógico. Mientras que el valor a enviar, el llamado *DTO*, es el valor que nuestro control enviará para la corrección de la posición

Una vez que podemos controlar el motor en lazo abierto es necesario que se desarrolle la cuadratura del encoder para visualizar el ángulo de la flecha, el desplazamiento máximo y asegurarnos que los datos entregados por los fabricantes de los materiales estén correctos. Así que se desarrolló un bloque llamado *Encoder* como se mencionó en el *Capítulo III.1.3 Encoder*. En la *Figura 38* se tienen los componentes del *TOP* del bloque, el código completo se encontrará en los *ANEXOS* al final del documento.

```
begin
    DOUT <= DAUX;
    BD1 <= DAUX(3 downto 0);
    BD2 <= DAUX(7 downto 4);
    BD3 <= DAUX(11 downto 8);
    BD4 <= DAUX(15 downto 12);
    U01 : BidirectionalCounter generic map(16) port map(RST, CLK, ENA, DIR, LIA, LIB, DAUX);
    U02 : QuadratureDecoder generic map(3) port map(RST, CLK, CHA, CHB, ENA, DIR);
    U03 : Segmentos port map(BD1, BC1);
    U04 : Segmentos port map(BD2, BC2);
    U05 : Segmentos port map(BD3, BC3);
    U06 : Segmentos port map(BD4, BC4);
    U07 : TimerSeg port map(RST, CLK, TIC, BC1, BC2, BC3, BC4, BC7, SEL);
    U08 : Timer generic map(1000000) port map(RST, CLK, TIC);
end Behavioral;
```

Figura 38. Componentes del bloque Encoder.

Una rápida explicación nos lleva a que el componente *QuadratureDecoder* es el más importante pues es, como ya se mencionó, el que determina la dirección del encoder así como indica cada que se cambió una posición. Estos datos llamados *DIR* y *ENA* entran en un contador bidireccional asigna un número entero a la posición del encoder, ya sabemos, según la *Ecuación 22*, que *4000 cuentas* equivalen a un centímetro en la plataforma. Este dato entero sale del bloque como *DOUT* para que otros bloques dispongan de esa información; también es seccionada en nibbles (conjunto de 4 bits en binario) para desplegarse de forma hexadecimal en los display de 7 segmentos con los que cuenta la tarjeta Basys 3.

Por ahora la salida del *BidirectionalCounter* es un dato de 16 bits, suficiente para almacenar el dato de posición que se requiere, sin embargo, es común que se agreguen bits de guarda, los cuales son bits extras que no afectan al desempeño del Encoder, pero ayudan a evitar que se consideren números negativos a aquellos valores que simplemente son muy grandes, además que facilitará la resta a la que posteriormente someteremos el dato, por lo que en el código final la salida del bloque *Encoder* es un dato de 32bits.

Con lo desarrollado hasta el momento es posible comenzar con las primeras pruebas sobre el motor en lazo cerrado, para esto se vinculó el valor de entrada del bloque *ICuadradaC* a los switches que tiene la tarjeta Basys 3, esto para probar los valores que recibirá el cómo comando el motor. Esta prueba se dividirá en dos partes, la primera es conocer como se ve afectada la velocidad en relación al voltaje suministrado por lo que el motor se debe desacoplar del eje para evitar colisiones en la plataforma, se requiere conectar todo y la prueba comenzará enviando el valor *"01111111"* que teóricamente es el valor donde el motor está estático.

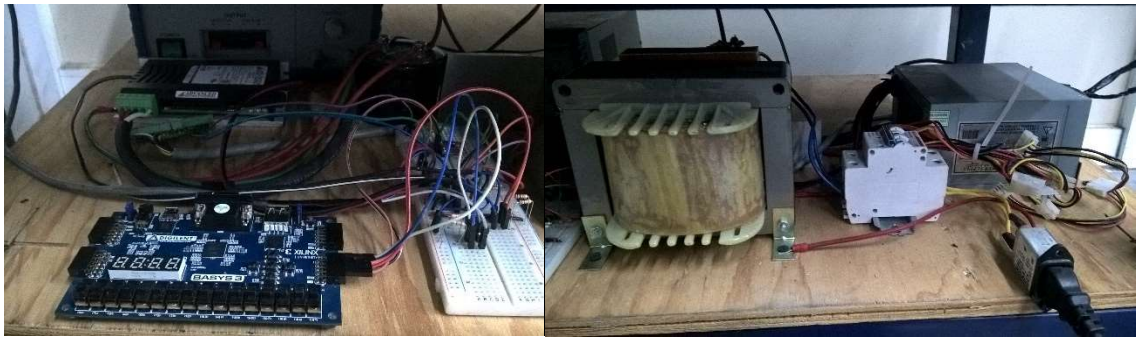


Figura 39.Vista general del circuito electrónico.

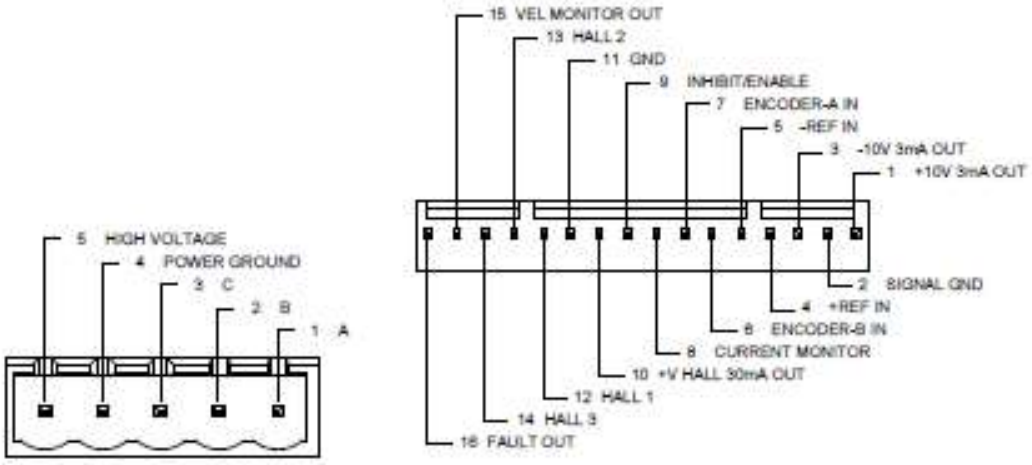


Figura 40. Conexiones en el Servo Drive.

Figura 41. Señal de control mediante el protocolo I^2C.

Para llevar a cabo la primera prueba debe conectarse el circuito de potencia como se mostró en la *Figura 36*, el circuito de acoplamiento y el DAC, todo puede verse en la *Figura 39* donde se muestra a grandes rasgos las conexiones. También debemos conectar el motor al Servo Drive y alimentar la referencia que saldrá de nuestro circuito de acoplamiento, para esto se siguieron las conexiones de la *Figura 40* que se halló en el *Datasheet* del componente.

Una vez que las conexiones se realizaron se verificó que el voltaje de alimentación sea el correcto en la entrada del Servo Drive y que este no marca ningún fallo, después alimentarle la referencia con una fuente externa para verificar que el motor gira correctamente. Con un multímetro verificar que la salida del voltaje sea la deseada, y que

concuerde con lo que se quiere enviar a la salida del DAC, y después del circuito de acoplamiento. Una vez que se realizaron estos pasos previos y funcionaron correctamente se continúa conectando la referencia del circuito de acoplamiento al Servo Drive. La *Figura 41* muestra el protocolo I^2C enviado al DAC desde el FPGA a través de un osciloscopio.

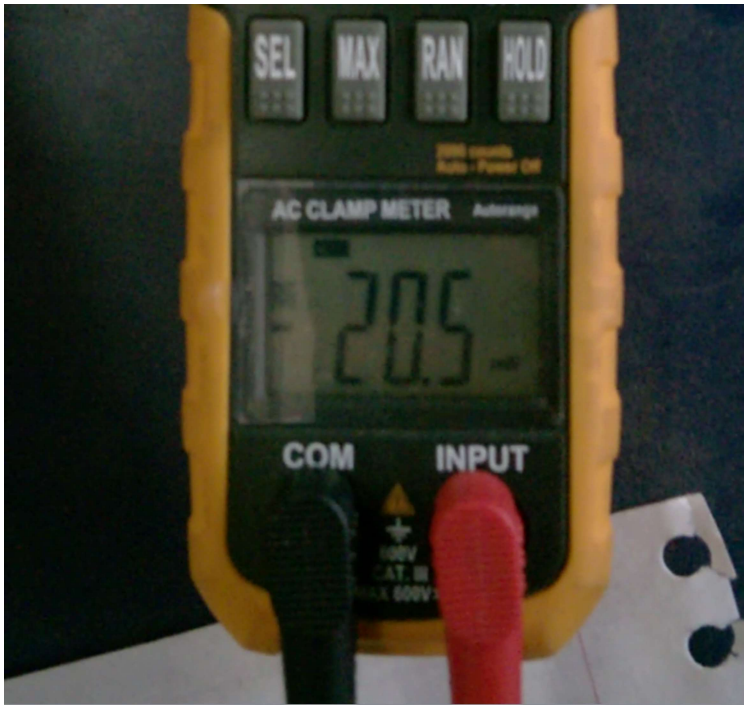


Figura 42. Valor de voltaje con el motor detenido.

Con esta prueba podemos observar como al variar el voltaje se mueve el motor a distintas velocidades e incluso cambia el sentido de giro. Es necesario en esta prueba verificar qué valor es el adecuado para dejar el motor estático, así que con la prueba se halló que el valor que detiene el motor es: *"011111100101"*, para este valor la salida es $20.5mv$ revisada con un multímetro (*Figura 42*). A partir de ese dato se obtienen dos valores importantes, el límite máximo y el límite mínimo que serán programados en el *Saturador* (Revisar en *Capítulo III.2.5 Saturador*).

Cuando el error sea cero significa que la posición de la plataforma es la correcta, suponiendo que el error directamente es nuestra salida de control debemos sumarle el valor *"011111100101"* continuamente para hacer que el motor no se mueva. Como sabemos que hay un valor continuo sumando, podemos inferir que nuestro valor máximo será: "*111111111111 - 011111100101*" (Valor máximo recibido por el DAC menos valor sumado continuamente) así definimos el valor máximo de nuestro error para este caso. De la misma manera el valor mínimo será: "*000000000000 - 011111100101*" (Valor mínimo recibido por el DAC menos valor sumado continuamente), nótese que este último valor es negativo por lo que la implementación de código deberá manejar valores con signo (usando

la línea *use IEEE.std_logic_signed.all;* al comienzo de cada bloque implementado que maneje valores con signo).

La segunda prueba básica será acoplar de nuevo el motor, juntando la implementación del *Encoder* y la implementación del *ICuadradaC* con el fin de verificar el funcionamiento del encoder, comenzando la prueba con el valor antes encontrado que equivale al *STOP* del motor, para esta prueba se necesita acoplar el motor a la flecha, para el movimiento se recomienda comenzar modificando los bits menos significativos para asegurar un movimiento controlado de la plataforma. Una buena prueba es mover la plataforma un a un determinado punto, medir la distancia desde el origen del movimiento, calcular la cantidad de cuentas que debió dar con el uso de una variante de la *Ecuación 23* y verificar que los displays de la *Basys 3* coincidan con el valor calculado. En la *Figura 43* se ve el valor de 36B0 que equivale en este caso 28000 *cuentas* que equivale a un desplazamiento de $3.5\ cm$.

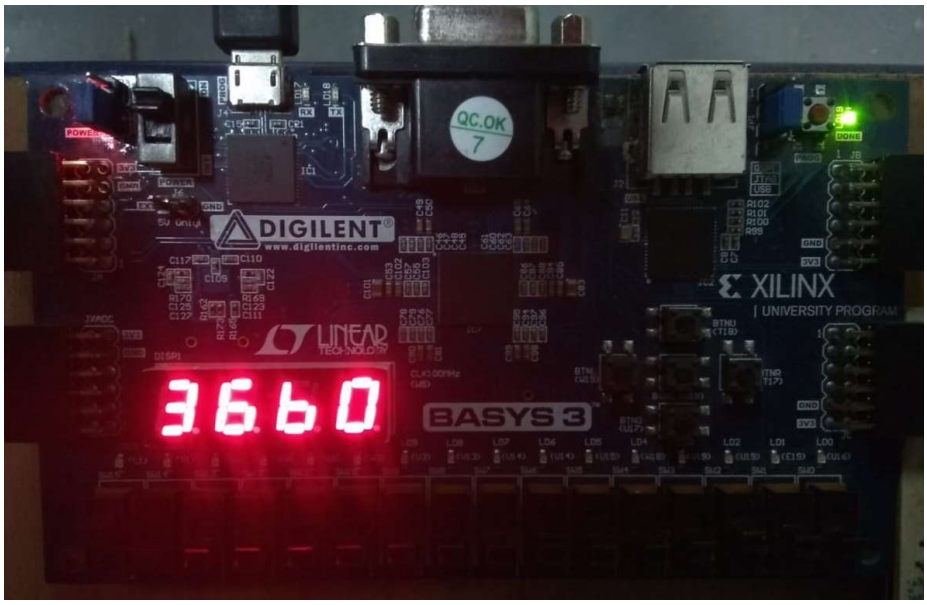


Figura 43. Visualización del valor del encoder en el FPGA.

Ya que podemos visualizar el valor que entrega el encoder después de mover la plataforma, podemos verificar la *Ecuación 23* dónde calculamos que el valor máximo que vamos a obtener para $7cm$ de movimiento es un valor de $24000\ cuentas$. Prácticamente se obtiene el valor calculado teóricamente.

Para siguiente paso es necesario que el valor del encoder se despliegue en una interfaz gráfica en la computadora, a través de la cual también se debería de controlar la posición del motor. Para este siguiente paso se implementó en el *FPGA* el bloque *TOPUART* tal y como se mencionó en el *Capítulo III.2.2 Protocolo UART*.

```
--Transmisión y recepción de datos
    U01 : TOPUART generic map(8) port map(CLK, RST, TIG, RXD, TXD, EOT, DIN, DATA);   --Protocolo UART
    U02 : Timer generic map(500000) port map(RST, CLK, PUT);                          --Velocidad de transmisión master 312500
    U03 : RisingEdge port map(RST, CLK, EOT, FIN);                                    --Final de recepción
    U04 : ShiftRegister port map(RST, CLK, FIN, DATA, DATO);                          --Toma los primero 8 bits de datos
    U05 : LoadRegister generic map(16) port map(RST, CLK, SYA, DATO, DOUQ);           --Toma datos de byte en byte para concatenar
    U06 : Counter generic map(2) port map(RST, CLK, FIN, SYA);                        --Número de datos a recibir
--Escritura a la interfaz gráfica
    U12 : SRLatch port map (RST, CLK, PUT, CLT, RTT);                                 --Enclava la transmisión
    U13 : Timer generic map(50000) port map(RTT, CLK, TIG);                           --Velocidad de transmisión slave 13020
    U14 : FreeCounter generic map(2) port map(RTT, CLK, TIG, SDE);                    --Array que entra como selector del dato al mux
    U15 : Counter generic map(2) port map(RTT, CLK, TIG, CLT);                        --Número de datos a enviar
    U16 : MuxData port map(DSAL(15 downto 8), DSAL(7 downto 0), SDE, DIN);            --Multiplexor con los datos a enviar
```

Figura 44. Implementación para enviar y recibir datos.

Como se dijo, el bloque *TOPUART* es un bloque genérico para enviar y recibir datos, debido a su simplicidad solo es capaz de enviar y recibir datos del tamaño de 1byte a la vez, por lo que es necesario usarlo múltiples veces. La *Figura 44* muestra la implementación de múltiples bloques genéricos que complementan al *TOPUART* con el objetivo de enviar 2 bytes equivalentes a la posición actual y recibir 2 bytes equivalentes a la posición deseada.

Para recibir la información se ocupa un *LoadRegister* que almacena un dato de 16 bits, y un *Counter* que define la cantidad de bytes a recibir. Si se desea recibir más datos se requiere que únicamente se modifiquen la cantidad de bits equivalente al valor que el *Counter* tenga.

Para escritura se usa un multiplexor llamado *MuxData* el cual almacena los bytes que se desean enviar, un *FreeCounter* le envía un selector que recorre la cantidad de datos que el *Counter* decida que tiene que enviar, de esta manera si se desean enviar más datos solo se acomodan en el *MuxData* y se cambia el valor del *Counter*. Nótese que hay un segundo *Timer*, el componente *U13* indica la frecuencia con la que se enviarán datos, el valor *50000* determina que se pueden enviar hasta 2000 datos por segundo, sin embargo el primer *Timer* pone un límite de 200 tramas por segundo, lo que al final resulta en tan solo enviar 400 datos por segundo (Ya que solo se envían dos datos de posición) por ahora.

Para poder visualizar los datos de la posición actual y enviar los datos de posición deseada debemos utilizar una interfaz gráfica desarrollada especialmente para atender las necesidades de este proyecto. Al principio del proyecto se planteó que podría desarrollarse la interfaz en dos entornos distintos, Matlab y LabView. LA interfaz gráfica desarrollada en Matlab presentó algunas dificultades para poder graficar en tiempo real, quiere decir que el valor mostrado presentaba un atraso significativo con respecto al valor real. La interfaz desarrollada en Matlab se puede observar en la *Figura 43*.

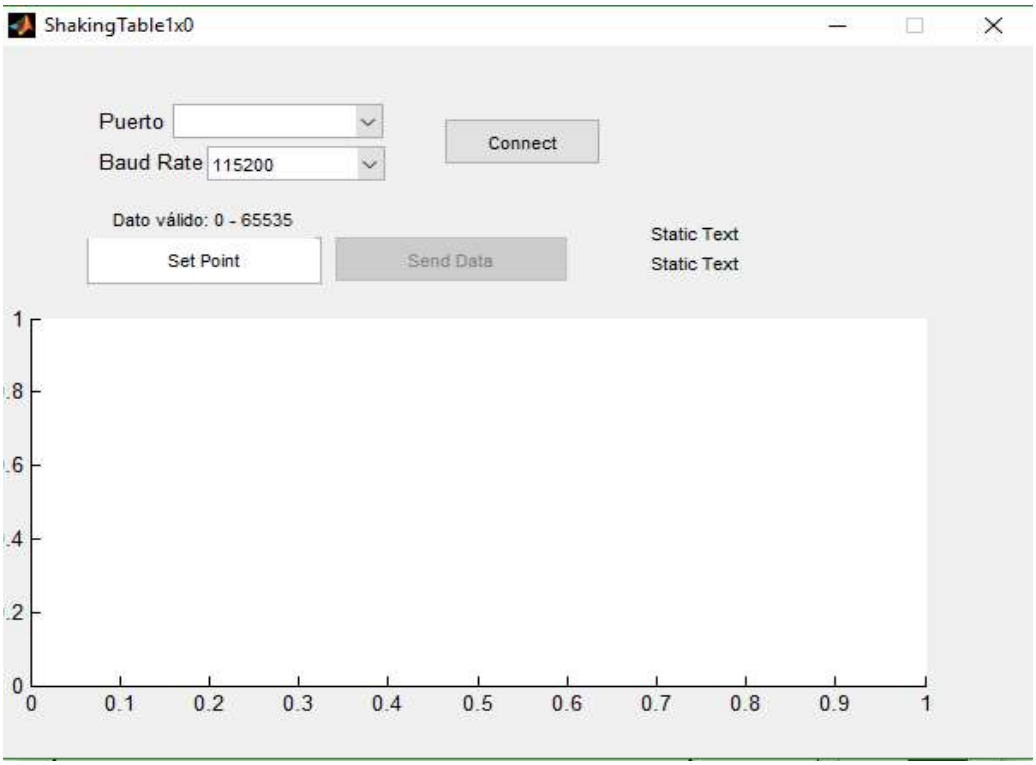


Figura 45. Interfaz gráfica en Matlab.

Por lo antes descrito se decidió realizar la interfaz gráfica en Labview, la primer interfaz gráfica desarrollada es la que se puede observar en las *Figuras 29* y *30*. En comparación con Matlab, este otro software hace un poco más complicado la manipulación de los datos, sin embargo, el entorno de programación es mucho más sencillo pues se programa en bloques.

Esta primera interfaz recibe el valor de posición de 8 bits (Solo los más significativos), la interfaz recibe la información como dato tipo *string*, por lo que se debe convertir a un entero de 8 bits para posteriormente graficarlo. Mientras que en la salida se utiliza una *caja de escritura* donde el usuario puede introducir el valor deseado, es importante modificar el tipo de dato aceptado, por lo general está programado como *string*, si este dato no se cambia los caracteres escritos se enviarán con base en el formato ASCII, así que debemos configurar la *caja de escritura* como dato tipo hexadecimal, así el dato se enviará en binario automáticamente.

El dato que se envía desde la interfaz debe ser forzosamente un dato de 4 *nibbles* (16 bits), así la FPGA recibirá este dato y tendrá que compararlo contra el dato que tiene en ese instante el *Encoder*. Para poder comparar los datos es necesario un bloque llamado *Substract* que no es más que una resta de dos números binarios. El código de este bloque se encontrará en los ANEXOS. Es importante notar que el valor de encoder es un arreglo de 32 bits pues le agregamos *bits de guarda*, es mucho más sencillo restar valores de la misma

longitud por lo que vamos a agregar 16 bits de guarda para el valor recibido como *posición deseada*, estos bits siempre serán ceros.

Ya que el valor es restado obtendremos un nuevo valor equivalente al error, recordemos que se vio en capítulos anteriores que el error es: $e = \theta - \theta d$. El error ahora almacenado en una variable de 32 bits llamada *ERR* deberá ser suficiente para ser la entrada del DAC que controla al motor, pero debemos recortar el valor de 32 a tan solo 12 bits.

Para esto se requiere del bloque llamado *Saturador* el cual se encargará de recortar el valor y también de enviar un valor máximo cuando el error es mayor y un valor mínimo en otro caso. Primero es importante definir por qué es posible que el valor entre directamente sin que exista un bloque de controlador. Si el valor enviado al DAC son los 12 bits menos significativos equivale a tener un *controlador proporcional* con una constante *Kp=1*, para todo caso vamos a suponer que la salida del *Saturador* será siempre un valor entero.

Es necesario definir límites máximos y mínimos porque el número máximo aceptado por el DAC es "*111111111111*", el error es capaz de llegar a ese número pero también es capaz de ser más grande, lo que se traduce a tener valores en cero y un bit más significativo en "*1*", esta acción haría que el motor cambiara de sentido de giro haciendo el error aún más grande provocando errores en el sistema. Anteriormente calculamos los valores máximos y mínimos entre los cuales se tiene que situar nuestro error para salir directamente al DAC, si éste se encuentra fuera de los límites, la salida hacia el DAC tomará el valor del límite que rebasó.

```
Architecture Behavioral of Saturador is
signal DMAX : std_logic_vector(31 downto 0) := "00000000000000000000100000011010";
signal DMIN : std_logic_vector(31 downto 0) := "11111111111111111111100000011011";
signal DATA : std_logic_vector(11 downto 0);
begin
    Combinational : process(DIN, DMAX, DMIN, DATA)
    begin
    DOUT <= DATA + "011111100101";
        if DIN > DMAX then
            DATA <= DMAX(11 downto 0);
        elsif DIN < DMIN then
            DATA <= DMIN(11 downto 0);
        else
            DATA <= DIN(11 downto 0);
        end if;
    end process Combinational;
end Behavioral;
```

Figura 46. Código implementado en el bloque de *Saturador*.

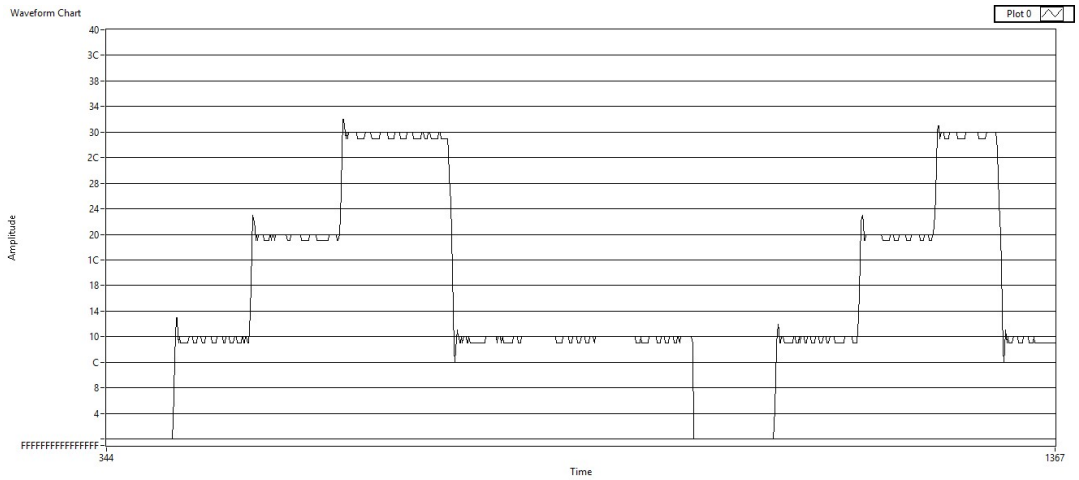


Figura 47. Respuesta del controlador con *Kp=1*.

La *Figura 46* muestra el código implementado del *Saturador* donde podemos observar que colocamos los valores antes calculados para los límites, y como a la salida se le suma el valor *STOP* que también se halló previamente. Con este valor de *Kp* se obtuvo la respuesta que se observa en la *Figura 47*, puede notarse que en estado estable se tiene una vibración no deseada, esto ocurre debido a que hasta con errores muy pequeños el motor tiene una respuesta, se podría decir también que se mantiene oscilando.

Para corregir esta oscilación es necesario bajar el valor de *Kp*, sin embargo, aún no se cuenta con un bloque de control. Es común que en ocasiones se requiera usar números decimales, así que es necesario conocer como como escribir un valor decimal fraccional en un binario. Para esto se define una cantidad de bits para la parte decimal, y otra para la fraccionaria. Debido a la falta de un controlador, por el momento se pueden usar únicamente los valores de la *Tabla 3.*

Valor decimal	Formato	Valor binario
1	**16.0**	**0000000000000001**
0.5	**15.1**	**000000000000000.1**
0.25	**14.2**	**00000000000000.01**
0.125	**13.3**	**0000000000000.001**

Tabla 3. Valores para *Kp*.

En la *Tabla 3* se puede observar un punto en los valores binarios, éste es únicamente representativo y no debe colocarse así en la programación. Hay que notar que al final el valor binario es el mismo solo se recorre el punto decimal, como mencionamos antes la salida del *Saturador* será siempre entera, por lo que si recorremos los 12 bits hacia valores más significativos obtendremos el efecto de cambiar el valor de *Kp*. En la *Figura 48* observamos el mismo código pero en esta ocasión con un valor de $Kp = 0.25$.

```
Architecture Behavioral of Saturador is
signal DMAX : std_logic_vector(31 downto 0) := "00000000000000100010000001101000";
signal DMIN : std_logic_vector(31 downto 0) := "11111111111111111110000001101100";
signal DATA : std_logic_vector(11 downto 0);
begin
    Combinational : process(DIN, DMAX, DMIN, DATA)
    begin
    DOUT <= DATA + "011111100101";
        if DIN > DMAX then
            DATA <= DMAX(13 downto 2);
        elsif DIN < DMIN then
            DATA <= DMIN(13 downto 2);
        else
            DATA <= DIN(13 downto 2);
        end if;
    end process Combinational;
end Behavioral;
```

Figura 48. Código implementado en el bloque *Saturador* con diferente valor de *Kp*.

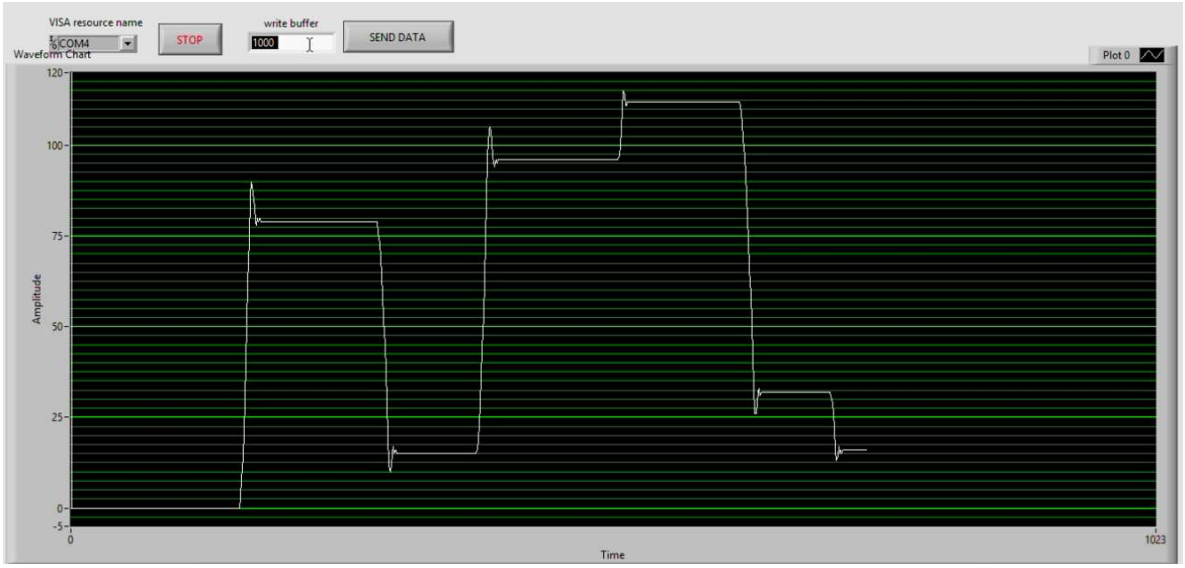


Figura 49. Respuesta al escalón con *Kp=0.25*.

En la *Figura 49* observamos que la respuesta al escalón ahora no tiene vibración en el estado estacionario y presenta oscilaciones, sin embargo la respuesta es rápida. Las oscilaciones podemos evitarlas bajando aún más el valor de *Kp*, así obtendremos una respuesta más amortiguada pero aún más lenta de lo que es ahora. Con esta respuesta podemos seguir trabajando por el momento. Hasta este punto ya tenemos un laso cerrado de control de posición para el motor, hemos logrado que trabaje bajo el primero de los cuatro modos de operación que llamaremos *Respuesta al Escalón*.

Ahora que podemos controlar el motor desde la interface y que tenemos listo el primer modo de operación es necesario continuar con uno de los modos de operación más importantes que es el *Senoidal* donde la señal de *Posición deseada* es variante en el tiempo.

En el *Capítulo III.1.5 Unidad de control* se vio de manera muy general como funcionaría este control.

Retomando, podemos discretizar cualquier señal variante en el tiempo mientras definamos un tiempo muestra, las señales replicadas se observarían como señales escalonadas, sin embargo, es así como podemos lograr que el controlador siga la señal, pues este recibirá escalones (Como en el modo anterior) a cierta frecuencia y con amplitudes específicas automáticamente.

Requerimos de una señal de tipo senoidal para probar los modelos de los ingenieros civiles, para esto vamos a definir una *Senoidal Unitaria* que es la señal base para nuestro sistema. Esta tendrá los valore de $Frecuencia = 1Hz$ y $Amplitud = 1cm$. Esta *Senoidal Unitaria* nos servirá para crear todo el espectro de trabajo que requiere la aplicación.

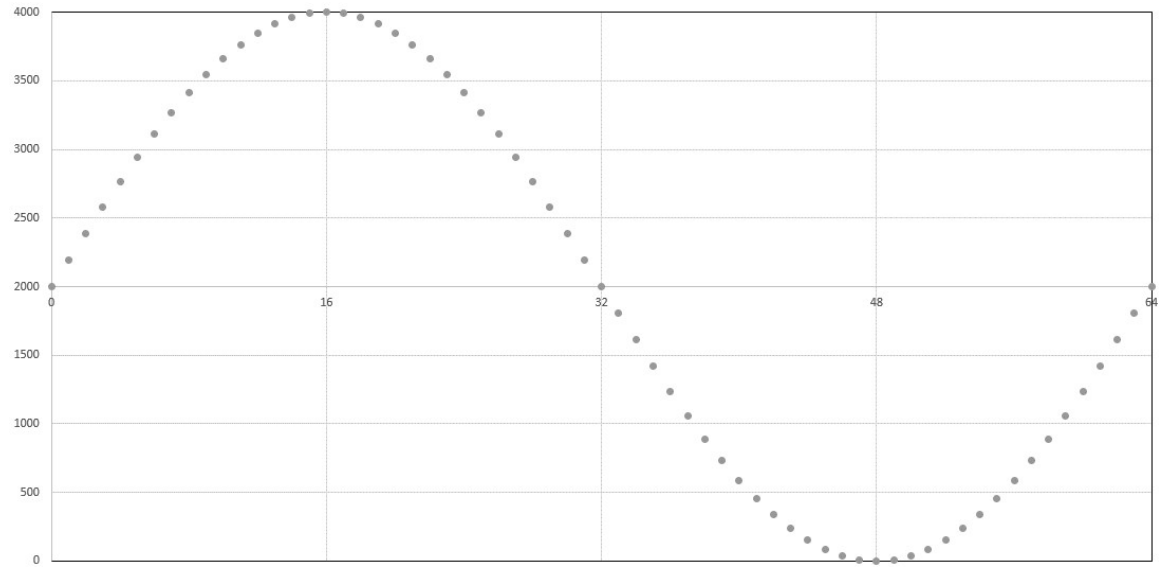


Figura 50. Discretización de la señal senoidal en 64 puntos para crear la Señal Unitaria.

Esta señal unitaria (*Figura 50*) tendrá todos sus valores guardados dentro de un *Multiplexor* (Bloque en el cual se tiene información ordenada y un selector determina que dato es el que se pone a disposición en la salida del bloque), estos valores pertenecen a una *Amplitud* de *1cm*, si requiero una amplitud más grande únicamente se debe multiplicar el valor de salida del multiplexor por la amplitud deseada, así cada escalón se separará lo necesario para tener a la salida una senoidal a diferentes amplitudes.

La señal de salida que será enviada como posición deseada para el control de posición seguirá la siguiente regla.

$$\theta d = \left(AMP * \frac{2n\pi}{N.Muestras}\right) + \left(14000 - (AMP * 2000)\right) \quad (24)$$

Donde *AMP* es un valor entero entre 1 y 7 (Valor definido anteriormente, equivalente al desplazamiento máximo de la plataforma), el *Número de muestras* es la cantidad de valores que conformarán un ciclo de la señal senoidal, para este trabajo se propuso que un ciclo estaría bien representado si utilizamos 64 muestras, también, *n* es un valor entero entre 1 y el *Número de muestras*. Es justo en la expresión $\frac{2n\pi}{64}$ donde se calculan los valores que se almacenarán en el *Multiplexor* de la senoidal unitaria. Los valores enteros 2000 y 14000 están en la ecuación debido a que el primero es equivalente a la mitad del desplazamiento mínimo (2000 cuentas equivalen a 0.5cm de movimiento) y el segundo número equivale a 3.5cm que es la mitad del desplazamiento total, esto para que la senoidal oscile a partir del centro de la mesa.

Si utilizamos 64 escalones para definir un ciclo de la señal senoidal tendremos que enviar 64 datos por segundo para obtener *1Hz* de frecuencia, bajo esta lógica necesitaremos enviar 168 datos en un segundo para *2Hz*, y así sucesivamente. El límite propuesto para este proyecto es alcanzar una frecuencia de hasta 15Hz. Los distintos valores para que el *Timer* cambie de dato a las distintas frecuencias se encuentran en un *MUXVel* cuyo valor de salida es elegido por una de dos posibles vías. Las dos vías llegan a través de un *Multiplexor2a1*, una de las vías es que el usuario elija la frecuencia a la que quiere correr la prueba, la segunda forma parte del tercer *Modo de operación* llamado *Barrido de frecuencias*, en este modo un *Timer* programado a *5 segundos* recorre todos los valores de frecuencia posibles.

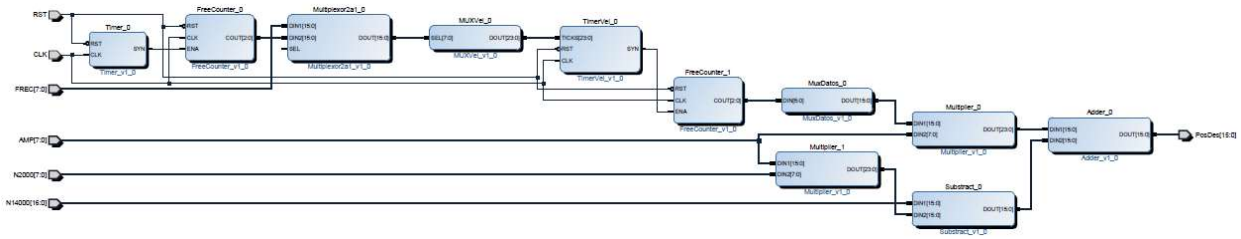


Figura 51. Diagrama de bloques de la posición deseada tipo senoidal.

La *Figura 51* muestra el diagrama a bloques de la salida de posición tipo senoidal, del lado izquierdo podemos ver la parte de selección de la *Frecuencia* a través de la selección de uno de los dos tipos de operación, del otro lado vemos implementada la *Ecuación 24* para obtener la salida deseada.

Ahora podemos controlar la posición deseada para que la referencia de la plataforma sea una senoidal a distintas frecuencias y amplitudes, pero cómo el usuario definirá esto desde una interface y cómo la implementación sabrá qué valor tiene que seguir el *Control de posición*. Pues para esto se ha definido que la entrada de valores desde la interfaz gráfica debería de tener un byte más, con el cual podamos definir el modo de operación.

Se definió que la trama recibida desde la interfaz, una trama de 24 bits, se dividiera en tres partes importantes: Del bit 1 al 8 será la parte de *Configuración*, del 9 al 16 será el valor *Frecuencia*, del 17 al 24 será el valor de *Amplitud*; en un caso especial, los bits del 9 al 24 serán el valor *SetPoint* o *Valor deseado de posición*.

Valor Binario	Uso	Señal	Asignado
00000001	SetPoint	16 bits de valor deseado	0100
00000010	Senoidal	8 bits Frecuencia y 8 Bits Amplitud	0010
00000011	STOP	No importa	1000
00000100	Barrido	No importa	0011
Otros	Ninguno	Caso diferente	1100

Tabla 4. Valor binario recibido para la *Configuración* y que significa.

En la *Tabla 4* vemos como se reciben únicamente 4 valores distintos para el byte recibido desde la interfaz y qué acción toma el control con cada uno de ellos. Un bloque importante es el llamado *Compara* donde entran estos cuatro bits menos significativos para asignar valores a variables que ayudarán a controlar las funcionalidades internas, en la misma tabla se ve en qué valor se convierte cada caso. A partir de que transformamos el valor de entrada definimos valores a las variables *VSE*, *DDO*, *DTR* y *DUN*.

La variable *VSE* toma el valor del bit menos significativo y sirve como selector en el *Multiplexor2a1_0* de la *Figura 51* que sirve para definir si la velocidad del *TimerVel* que define la frecuencia está dada por el usuario o es un barrido de frecuencias el que la controla; esta misma variable activa un *SRLatch* para que la frecuencia en el barrido siempre empiece en uno. La variable *DDO* recibe su valor del tercer bit, y sirve para activar el *TimerVel* antes mencionado, se puede notar que está en uno en los casos donde se necesita de la senoidal. La variable DTR toma el valor del cuarto bit o bit más significativo, y tiene la función de detener los *SRLatch* activados con las variables *VSE* y *DDO*.

```
Architecture DataFlow of MuxPosicion is
begin
    With SEL select DOUT <=
    DIN2 when "00",  --Valores de la senoidal
    DIN1 when "01",  --Valor de entrada DOUQ(23 downto 8)
    "0011011010110000" when "10",  --14000 equivale a 3.5cm
    "0000000000000000" when others;
end DataFlow;
```

Figura 52. Código implementado del *MuxPosición*.

La última variable llamada *DUN* toma su valor de conjunción de los bits 4 y 3 (En ese orden) y sirve como selector en el *MuxPosicion* el cual define el valor de posición deseado que se enviará al programa principal del *Control de posición* del *Sistema generador de vibraciones*. La *Figura 52* nos muestra el código implementado dentro del bloque dónde nos

podemos dar cuenta (Tomando también como referencia la *Tabla 4*) que para los modos de uso *Senoidal* y *Barrido de frecuencias* se tiene una sola salida, para el modo *Respuesta al escalón* existe una más y aun así existen otras. Cuando se presiona el botón de detener se ve un efecto parecido a cuando se envía un escalón con posición de *14000 cuentas* esto para que la mesa quede estática en la posición central, también funcionaría como botón para dejar la mesa al centro. La última salida se da cuando un valor cualquiera llega a la tarjeta como ruido eléctrico, esta salida hace que la mesa se desplace a posición de cero cuentas.

El paso final dentro de la tarjeta es alargar la recepción de los datos a 3 bytes, y la longitud de las variables que manipulan los datos, ya que se tiene esto, podemos pasar a los cambios necesarios en la interfaz gráfica.

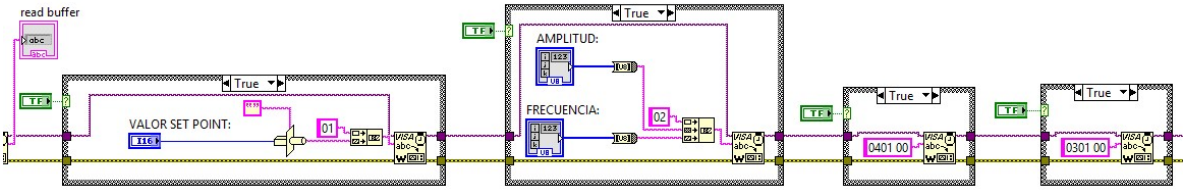


Figura 53. Programación en bloques para enviar datos para múltiples tareas.

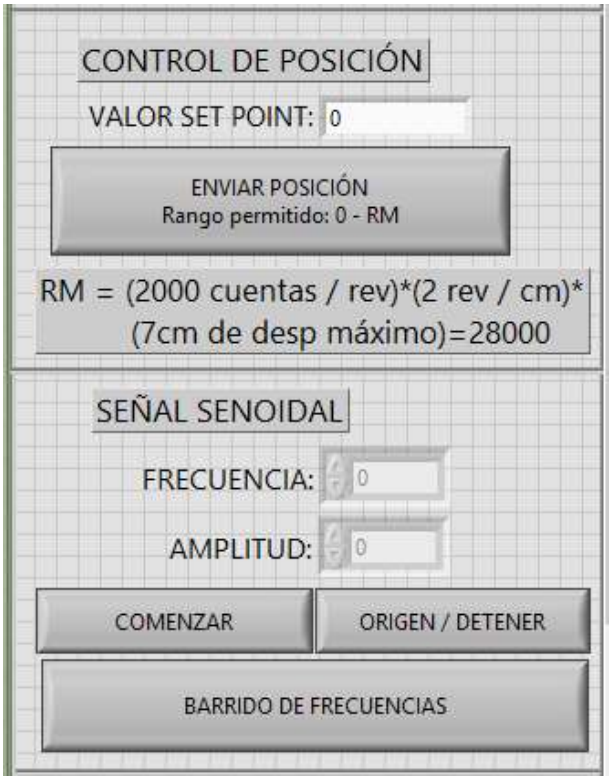


Figura 54. Parte de la interfaz que controla los *Modos* de operación.

Es muy sencillo poder enviar un dato con más nibbles pues al usar formatos hexadecimales en la *caja de escritura* solo se necesita concatenar 2 caracteres más a la transmisión, cuyo valor la *Tabla 4* ya definió. Para otros casos es necesario repetir el envío de datos, pues cuando se presiona un botón se envía lo que está dentro de la instancia, así que para enviar otros comandos se requieren otras instancias, así se ve en la *Figura 53* donde se ve que se repitió la instancia cuantas veces se necesitó para quedar visualmente como en la *Figura 54.*

Ahora que se tienen más *modos de operación* y una interfaz capaz de activarlos podemos hacer algunas pruebas de funcionalidad, principalmente a los modos que requieren de una señal variante en el tiempo. La *Figura 55* muestra el resultado de las pruebas para una referencia senoidal de valores unitarios, mientras en la *Figura 56* observamos la misma referencia con parámetros distintos.

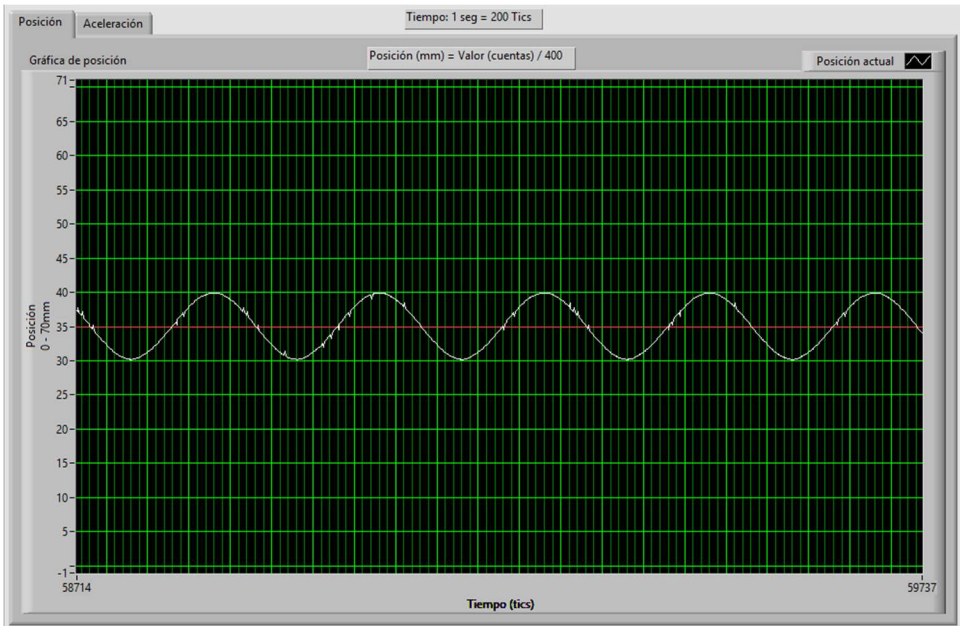


Figura 55. Funcionamiento de la respuesta senoidal con *Frec=1* y *Amp=1*.

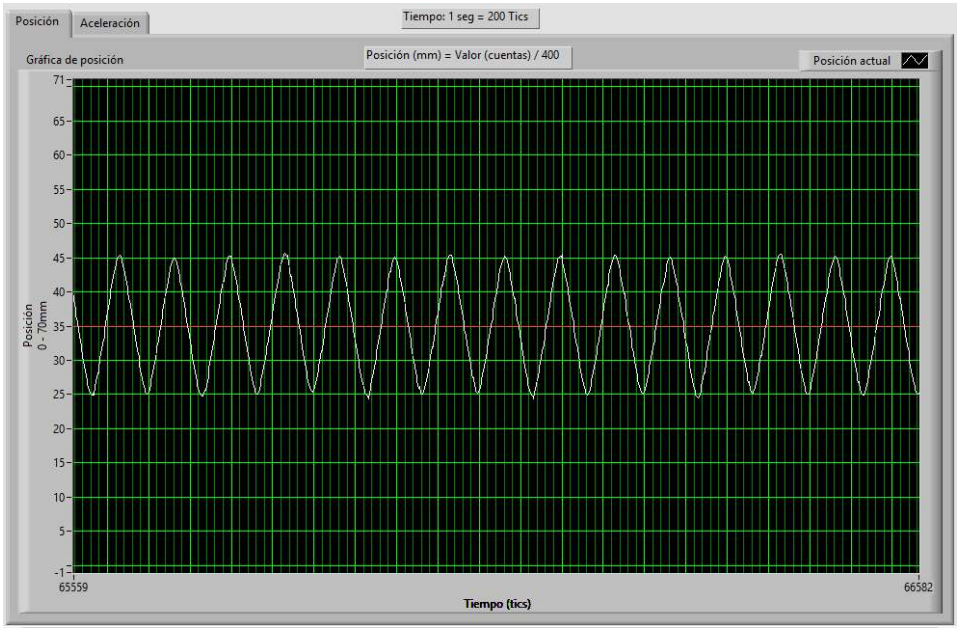


Figura 56. Respuesta senoidal con *Frec=3* y *Amp=5*.

Uno de los datos importantes a notar a partir de las pruebas antes realizadas es que como se darán cuenta para valores bajos de frecuencia la escala de los valores de amplitud se observan correctamente, pero a medida que crece la frecuencia las amplitudes grandes comienzan a atenuarse y no cumplirse los parámetros adecuados. Esto sucede debido a que el motor no puede responder tan rápido para cumplir frecuencias y amplitudes grandes al mismo tiempo, debido a que la velocidad necesaria sería muy grande aunado a una gran cantidad de energía. También afecta el valor del controlador pues con una constante *Kp* más grande el motor tiende a responder con mayor velocidad, pero como se vio anteriormente el control tiende a ser menos estable.

El cuarto y último *modo de operación* es una señal de un terremoto personalizado, este modo a final de cuentas es muy similar al *Modo Senoidal* pues los valores se tienen que guardar en un *Multiplexor* y únicamente se recorren los datos, así que este *modo de operación* podríamos no agregarlo por ahora pues repetiríamos el mismo proceso.

Para finalizar es necesario que instrumentemos cualquiera que sea la estructura a probar en el *Sistema Generador de Vibraciones*, la variable a medir para la instrumentación es la aceleración en distintos puntos de la estructura o en estructuras de diferentes alturas para compararlas. Ya en el *Capítulo III.2.1 Protocolo I^2C* se explicó que el protocolo se aplica de una manera similar tanto para el *DAC* y para los acelerómetros.

La trama a enviar para los acelerómetros está en la *Figura 18* dónde explicamos que cuando el esclavo, en este caso el acelerómetro, necesita responder el maestro, el FPGA, pone la línea de transmisión de datos *SDA* en alta impedancia. La nueva trama simplemente se agrega en el *MUX_0* y al mismo *SDA* se le agrega un *RegistroSerialParalelo* para la recepción de los datos. Una vez que se tienen los datos a disposición en una variable interna

es necesario enviarla a la interfaz gráfica para su visualización e interpretación por parte de los ingenieros civiles.

Para enviar los datos se desarrolló un sistema de envío secuencial, donde, el FPGA enviará 2 bytes correspondientes a la posición de la plataforma (Recordemos que anteriormente solo se enviaban los 8 bits más significativos), 2 bytes del *Acelerómetro 1* y 2 bytes del *Acelerómetro 2*. La aplicación no limita la capacidad de acelerómetros que podemos usar, pero para este primer prototipo comenzaremos con únicamente dos acelerómetros.

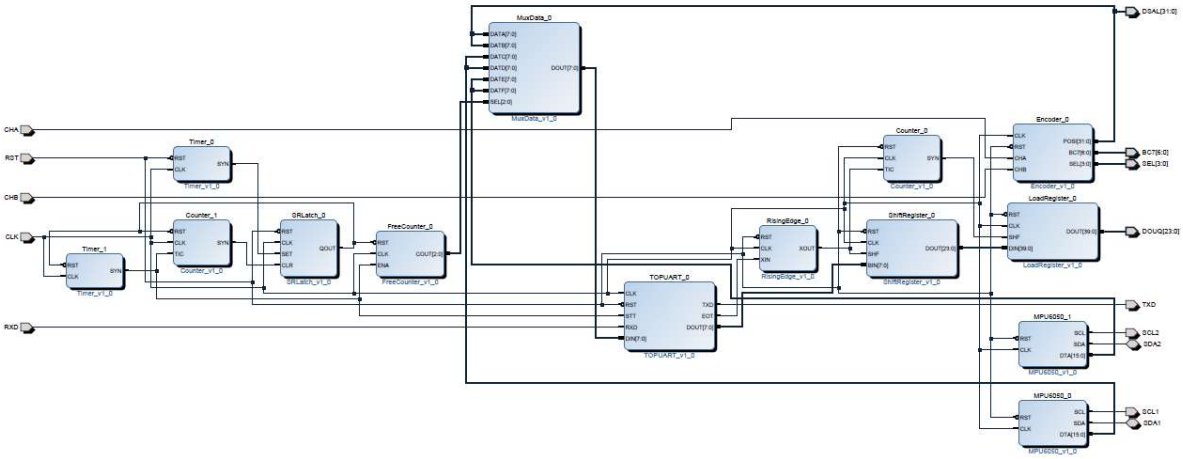


Figura 57. Transmisión UART de todos los datos.

En la *Figura 57* podemos observar como los datos que se envían están agrupados en un multiplexor llamado *MuxData_0* de donde salen uno a uno hacia el bloque *UART* para que éste pueda transmitirlos a la interfaz. Para que la transmisión de toda la trama se active se cuenta con el *Timer_0* que envía un pulso cada *5ms*, este pulso también activa un *SRLatch* que activa a su vez el *Timer_1* el cual se encarga de hacer cambiar los datos del *MuxData_0* a través de un *FreeCounter* cada 0.5ms.

Ahora que enviamos múltiples datos debemos también recibir en la interfaz múltiples datos y poder identificar cada uno de ellos para desplegarlos completamente, por ahora sabemos que la recepción será de 6 bytes y que están ordenados. En la *Figura 58* se observa como en el bloque de recepción ya no tiene una función para revisar el buffer en espera de un valor como se vio en la *Figura 29* ahora, por el contrario, se tiene directamente que recibirá 6 bytes, a la salida del bloque se tendrá un dato tipo *String* con los 6 valores.

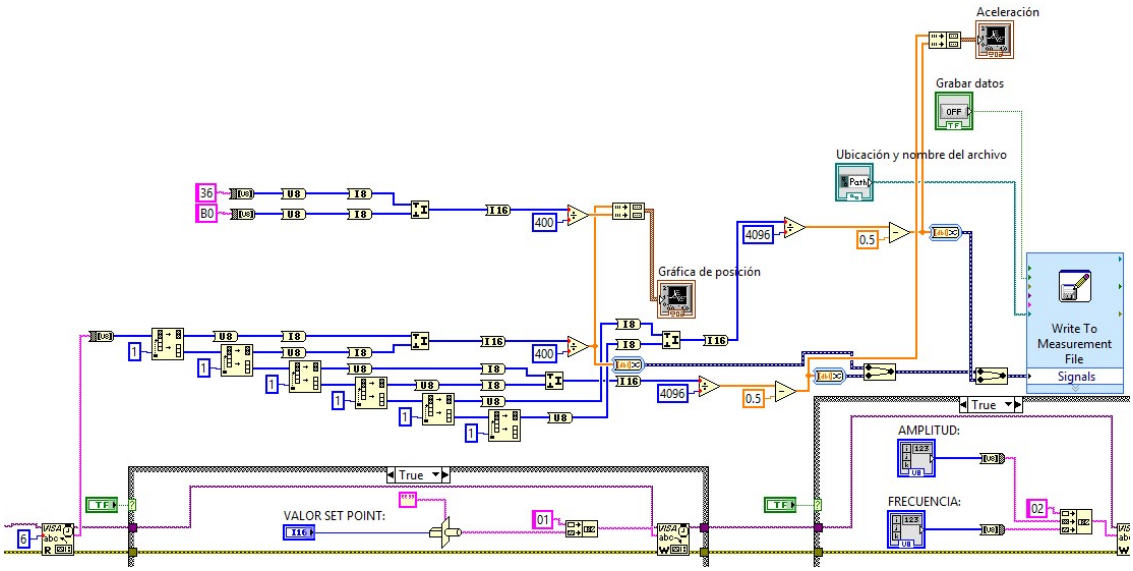


Figura 58. Código de bloques de la recepción de datos en la interfaz.

Como el dato es de un tipo diferente primero debemos convertirlo a un arreglo de bits, posteriormente los separaremos en datos independientes de 8 bits de longitud para poder convertirlos en datos tipo *entero*. Ahora tenemos 6 datos independientes equivalentes a 3 valores distintos por lo que se tienen que agrupar con su complemento, el valor equivalente a la posición se va a dividir entre *400* para tener a la salida *mm* en lugar de *cuentas*, como se observa en la parte de arriba se hace algo similar a lo explicado anteriormente con un valor fijo (0x36B0) para que al desplegarlo en la interfaz se tenga una línea recta sobre *35mm* para enunciar que ese es el valor medio de la plataforma.

Los datos de aceleración pasan por un proceso similar, estos se dividen entre 4096 para que a la salida tengamos un valor de aceleración con rangos máximos de ±8g, que es como se programaron los acelerómetros, incluso se hace un ajuste para tener los acelerómetros en cero cuando están perpendiculares al eje que están midiendo.

Los datos de posición se despliegan en una primer gráfica como se mostró en las *Figuras 55* y *56* mientras que las aceleraciones se desplegarán en otra gráfica en una pestaña detrás de la primer gráfica, como se muestra en la *Figura 59* donde se despliegan las aceleraciones registradas por los acelerómetros. Todos los datos procesados por la interfaz gráfica entran en un bloque, que se activa con un botón en la interfaz, dónde se almacenan en un archivo para *Excel*, así los ingenieros pueden hacer un mejor análisis de la información que se recopiló en la prueba, la gráfica en *Excel* de una señal de posición se muestra en la *Figura 60*.

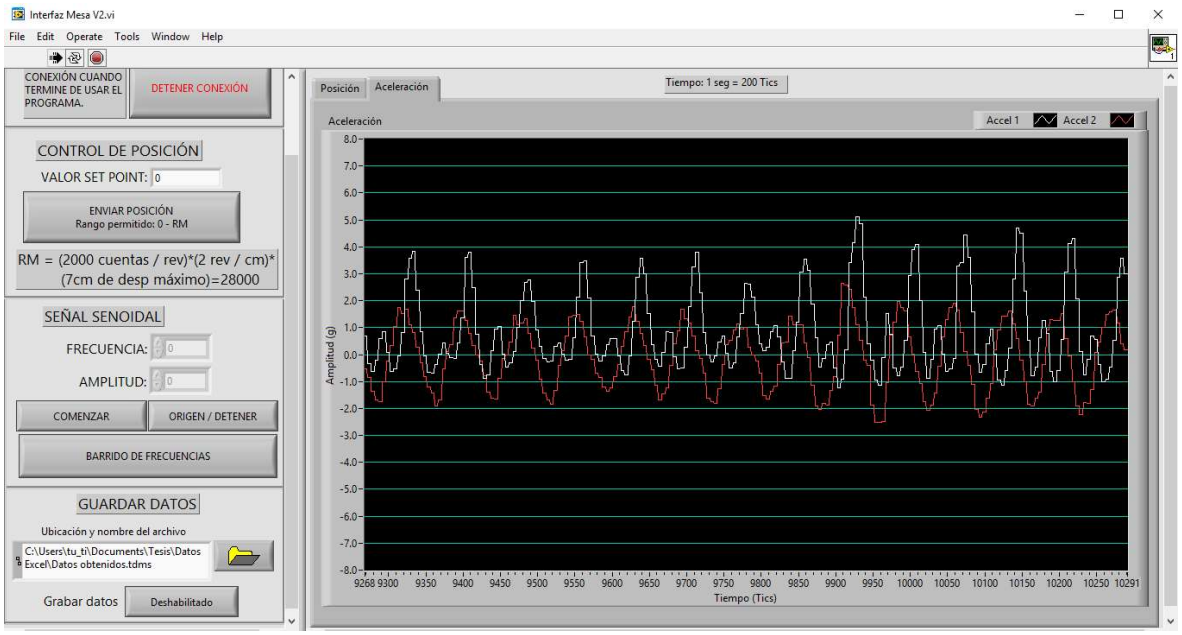


Figura 59. Gráfica de comparación de aceleraciones.

	Tiempo	Posicion	Aceleracion1	Aceleracion2
2	1	0	0.777587891	0.417480469
3	2	0	0.777587891	0.417480469
4	3	0	0.777587891	0.417480469
5	4	0	0.777587891	0.417480469
6	5	0	0.777587891	0.417480469
7	6	0	1.030517578	-0.007080078
8	7	0	1.030517578	-0.007080078
9	8	0	1.030517578	-0.007080078
10	9	0	1.030517578	-0.007080078
11	10	0	1.030517578	-0.007080078
12	11	0	0.914794922	0.19140625
13	12	0	0.914794922	0.19140625
14	13	0	0.914794922	0.19140625
15	14	0	0.914794922	0.19140625
16	15	0	0.914794922	0.19140625
17	16	0	0.491699219	0.329589844
18	17	0	0.491699219	0.329589844
19	18	0	0.491699219	0.329589844
20	19	0	0.491699219	0.329589844
21	20	0	0.491699219	0.329589844
22	21	0	0.682617188	0.231201172
23	22	0	0.682617188	0.231201172
24	23	0	0.682617188	0.231201172
25	24	0	0.682617188	0.231201172
26	25	0	0.682617188	0.231201172
27	26	0	0.543212891	-0.088134766
28	27	0	0.543212891	-0.088134766

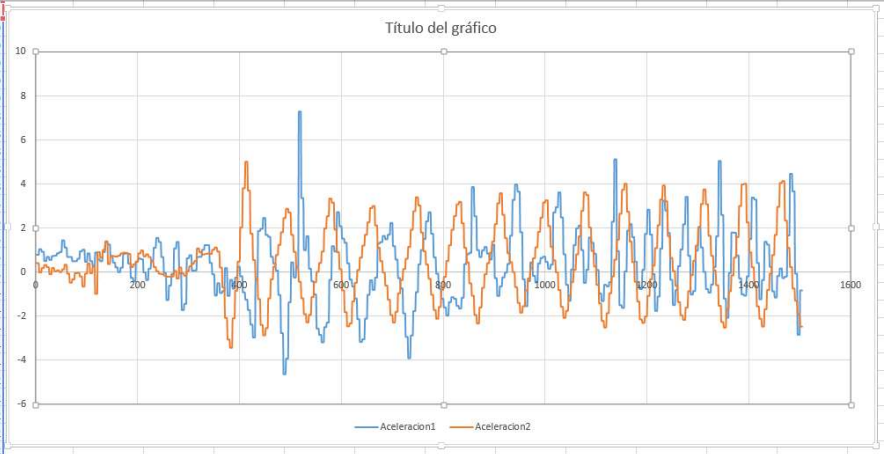


Figura 60. Gráfica a partir de los datos adquiridos.

En este punto podemos asegurar que todas las conexiones que deben hacerse ya las hemos probado de alguna forma por lo que el desarrollo de un circuito en un PCB es necesario para la presentación de un proyecto formal, así que con ayuda del software *Eagle* se desarrolló una tarjeta dónde pudiéramos tener: El circuito de acoplamiento con salida al *Servo Driver*, el *DAC* conectado al circuito, las conexiones hacia los *Acelerómetros*, las conexiones hacia el *Encoder* y la alimentación de la *Fuente* externa referenciada a la tierra del FPGA.

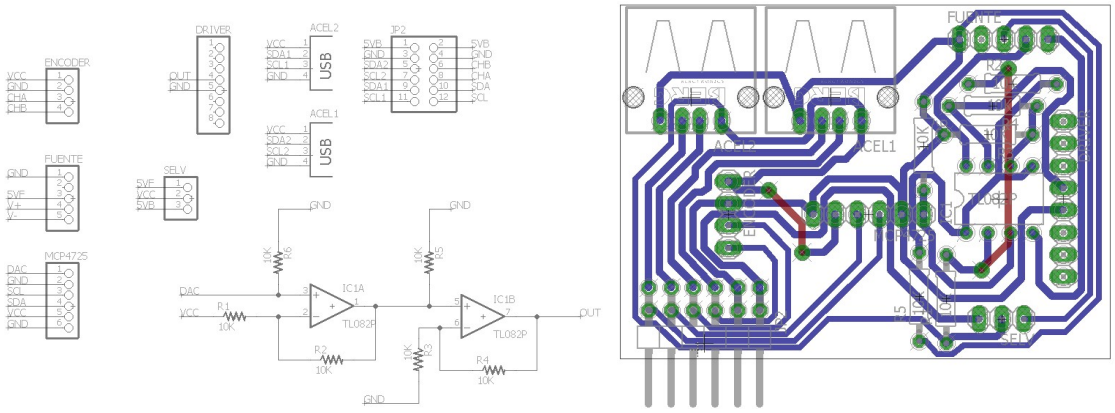


Figura 61. Esquemático de la placa del *SGV*.

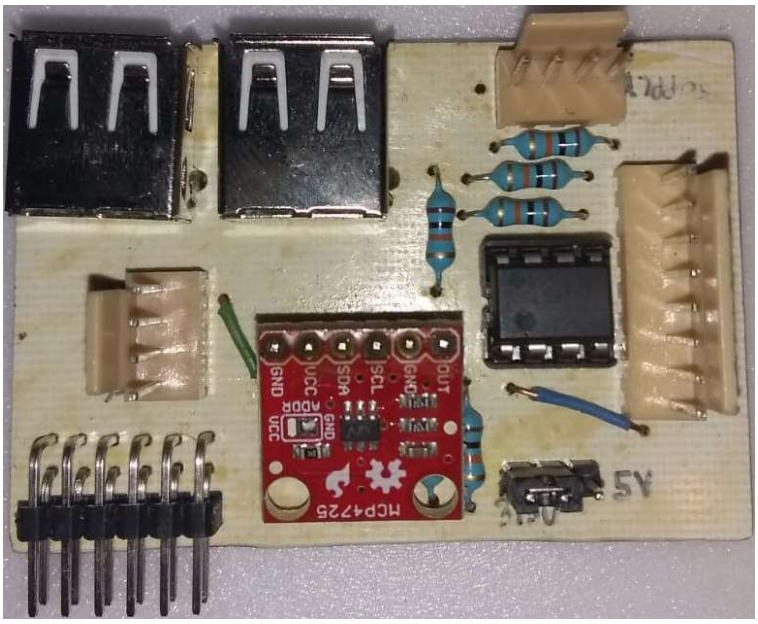


Figura 62. Placa del *SGV*.

El último paso en la implementación del *Sistema de Generación de Vibraciones* es la implementación de un controlador tal y como se ha mencionado desde el *Capítulo III.1.5 Unidad de control*. La implementación de este en el FPGA se hace como se vio en la *Figura 26* donde se observa cada bloque interno del bloque llamado *Controlador* que está conectado al resto del código como de observa en la *Figura 63*. Ahí podemos ver que la variable *TS* es de *1ms* según el *Timer* y también podemos ver que el controlador se activa con un switch desde la tarjeta, esto para dejar el controlador que ya funciona y poder probar el nuevo.

```
--El siguiente código es el controlador
    U34 : TopControl port map(RST, CLK, TS, ERR, CTL);
    U35 : Timer generic map(100000) port map(RST, CLK, TS);                --1ms
    U36 : SaturadorP port map(CTL, DSMI2);
    U37 : Multiplexor2a1 generic map(12) port map(DSMI1, DSMI2, SW4, DSMI);  --Saturador 1 y Saturador P
```

Figura 63. Conexión del controlador.

Las diferencias en los controladores las podemos ver para el *Modo de operación Escalón* en las *Figuras 64, 65 y 66*. Ahí vemos el tiempo de respuesta para los controladores y el sobrepaso en ellos.

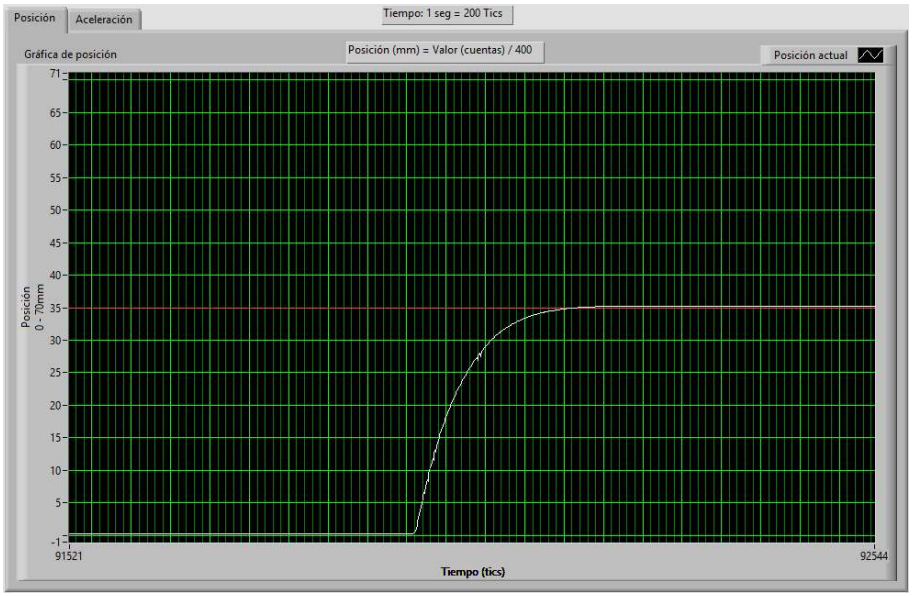


Figura 64. Controlador P con *Kp=0.0625*.

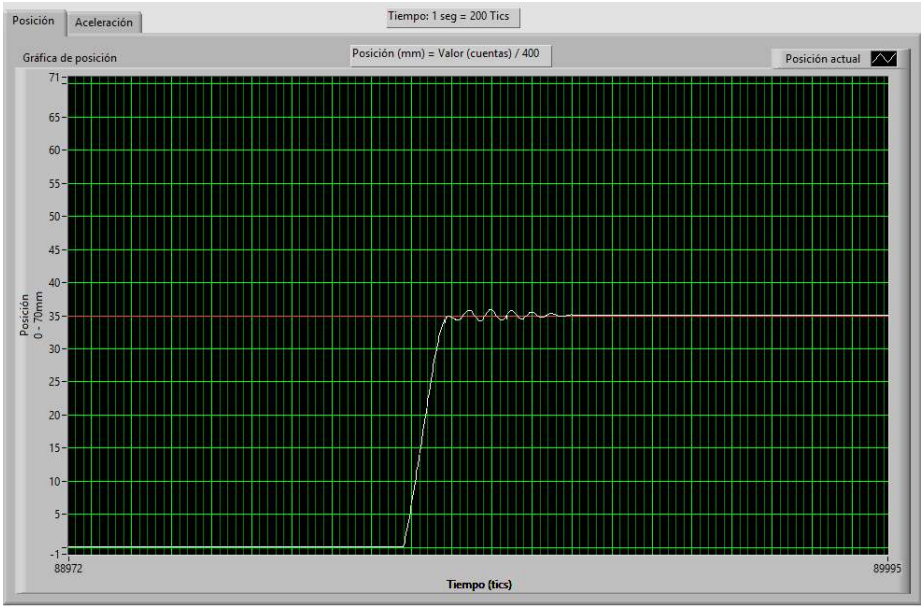


Figura 65. Controlador P con *Kp=0.25*.

4.2 Conclusiones

Al analizar detalladamente el funcionamiento del control para este proyecto, nos damos cuenta de que un controlador como el anteriormente desarrollado no es estrictamente necesario para el modo de operación de *Senoidal* y *Barrido de frecuencias* pues los escalones son tan pequeños que el control es imperceptible. De hecho la implementación de un controlador hace más lenta la respuesta del motor haciendo que la *amplitud/frecuencia* a la que el motor no responde correctamente disminuya. Podemos observar la comparación antes mencionada en las *Figuras 66* y *67* donde se tiene como referencia una señal senoidal con frecuencia de *1Hz* y amplitud de *3cm* (En color rojo), mientras que la respuesta de posición del sistema se observa en blanco.

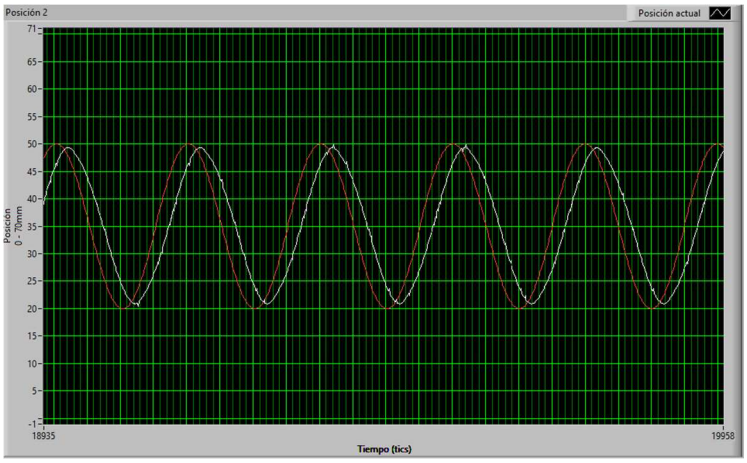


Figura 66. Señal senoidal con controlador P.

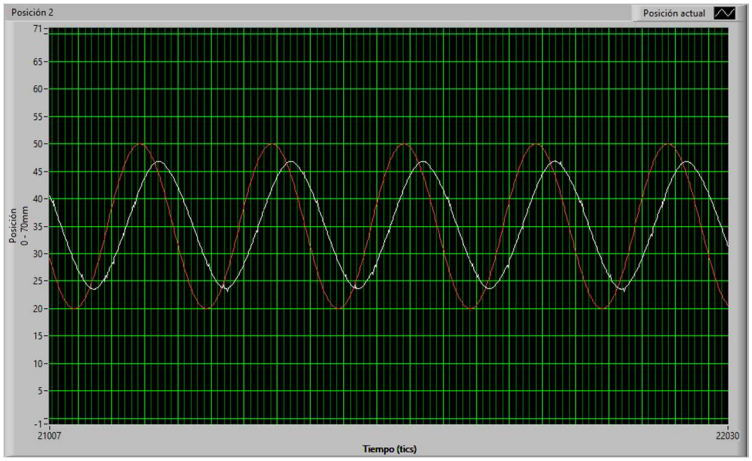


Figura 67. Señal senoidal con controlador PD.

El controlador PD busca que la respuesta sea más amortiguada por la que tarda un poco más en llegar al valor deseado, si queremos que siga una trayectoria esto se traduce en una respuesta general más lenta por lo que la atenuación de la señal se hace evidente. En el caso particular de la referencia vista en las figuras anteriores la atenuación es del 22.34% de la señal deseada mientras que si trabajamos únicamente con el controlador P la atenuación es de tan solo de 4.68%.

Sin embargo en el modo de operación *Escalón* sí que es importan el uso de un controlador, pues aquí si se nota la respuesta de éste al enviar un valor deseado de posición. Dicho lo anterior, el *controlador PD* se acciona como un extra desde la tarjeta de desarrollo *FPGA* para análisis y pruebas distintas, mientras que el funcionamiento general se deja con un simple *control P* con una ganancia $k_p = 0.25$.

La implementación del control de posición para el motor BLDC dentro de la tarjeta FPGA presenta múltiples ventajas, la principal es que tiene la funcionalidad de trabajar múltiples tareas en paralelo facilitando la adquisición de datos, en control y la escritura sin obtener retrasos significativos, además provee una estructura capaz de replicarse, esto significa que si se desea agregar otro eje a la plataforma no es necesario adquirir una nueva tarjeta, el mismo FPGA que controla esta mesa puede soportar uno o dos ejes más. Un caso similar pasaría con los acelerómetros, en esta ocasión comenzamos con el uso de dos acelerómetros pero fácilmente la tarjeta podría soportar el uso de dos más.

Por cuestiones como estas el desarrollo de controladores en FPGA para motores es un tema importante en el desarrollo tecnológico en la universidad y continuará dando importantes avances en el desarrollo de sistemas de movimiento. Esta tesis se ha desarrollado a manera que el prototipo aquí descrito pueda ser fácilmente replicable por los colegas ingenieros que deseen mejorar el modelo inicial o que requieren de un sistema similar en sus universidades para probar estructuras civiles y así mejorar la calidad constructiva en el país.

BIBLOGRAFÍA

1) Servicio Sismológico Nacional. (2018). Obtenido del Servicio Sismológico Nacional: http://www2.ssn.unam.mx:8080/estadisticas/

2) Fomperosa, M. (28 de Septiembre de 2017). Milenio. Obtenido de Milenio: http://www.milenio.com/tendencias/torre_latinoamericana-que_la_hace_resistente-terremotos-estructura-milenio-noticias_0_1038496144.html

3) Centro de instrumentación y registro sísmico A. C. (2017). Obtenido del Centro de instrumentación y registro sísmico A. C.: http://www.cires.org.mx/1985_es.php

4) Carrillo, J., Bernal Ruíz, N. M., & Porras, P. (2013). Evaluación del diseño de una pequeña mesa vibratoria para ensayos en ingeniería sismo-resistente. Bogotá, Colombia: Universidad Militar Nueva Granada.

5) Ciencia UNAM. (2013). Obtenido de Construcciones a prueba de sismos en la mesa vibradora: http//ciencia.unam.mx/leer/258/Construcciones_a_prueba_de_sismos_en_la_Mesa_Vibradora

6) Lehmann, A., Verri, A., Bertero, A., Muñoz, S. (2012). Consideraciones de diseño y construcción de una mesa vibradora para ejecución de ensayos dinámicos. Argentina: Laboratorio de Dinámica de Estructuras UBA.

7) Unidad de Puentes. (2013). Adquisición de mesas vibratorias para la docencia e investigación sobre el comportamiento dinámico de estructuras de puentes. San José, Costa Rica: Laboratorio Nacional de Materiales y Modelos Estructurales

8) Peralta, P., Castillo, R. (2013) Mesa vibratoria portátil para simular el efecto de sismos sobre estructuras de puentes y edificaciones. Boletín Técnico, volumen 4 No. 45.

9) Conte, J., Trombetti, T. (2000) Linear dynamic modeling of a uni-axial servo-hydraulic shaking table system, Earthquake Engineering and Structural Dynamics, Vol. 29, Pages: 1375-1404

10) Baran, T., Tanrikulu, A., Dundar, C., Tanrikulu, A. (2011) Construction and Performance Test of A Low-Cost Shake Table, Experimental Techniques, Vol. 35, No. 4, Pages 8-16

11) Rusia, P., Bhongade, S. (2014) Design and implementation of digital PID controller using fpga for precision temperature control. Power India International Conference (PIICON), In 2014 6th IEEE.

12) Martínez-Prado, M. (2014) Modelado dinámico de servomecanismos.

13) Hernández, V., Silva, R. & Carrillo, R. (2013) Control Automático: Teoría de Diseño, Construcción de Prototipos, Identificación y Pruebas Experimentales. México D.F.: Instituto Politécnico Nacional

14) Martinez-Prado, M., Rodríguez, J., Toledo, D., Torres, M., Herrera, G. (2017) Motion control with FPGA, Field - Programmable Gate Array, Prof. George Dekoulis (Ed.), InTech, DOI: 10.5772/67200. Available from: https://www.intechopen.com/books/field-programmable-gate-array/motion-control-with-fpga

15) García- Cortés, R. (2014) Análisis y desarrollo de un controlador de movimiento para un robot PUMA basado en FPGA. Tesis de licenciatura de la Facultad de ingeniería de la Universidad Autónoma de Querétaro.

16) Arana, D., Mendoza, R., (2011) Controlador de movimiento para bancada de dos ejes. Tesis de la Facultad de Ingeniería de la Universidad Autónoma de Querétaro.

17) Mendoza-Mondragón, F. (2012) Controlador estándar de movimiento multieje con base en FPGA. Tesis de maestría de la Facultad de Ingeniería de la Universidad Autónoma de Querétaro.

18) Estrella, R., García, M, (2014) Servo-controlador analógico-digital para motores de DC y BLDC. Tesis de la Facultad de ingeniería de la Universidad Autónoma de Querétaro.

ANEXOS

A1 Manual de Usuario

SISTEMA GENERADOR DE VIBRACIONES

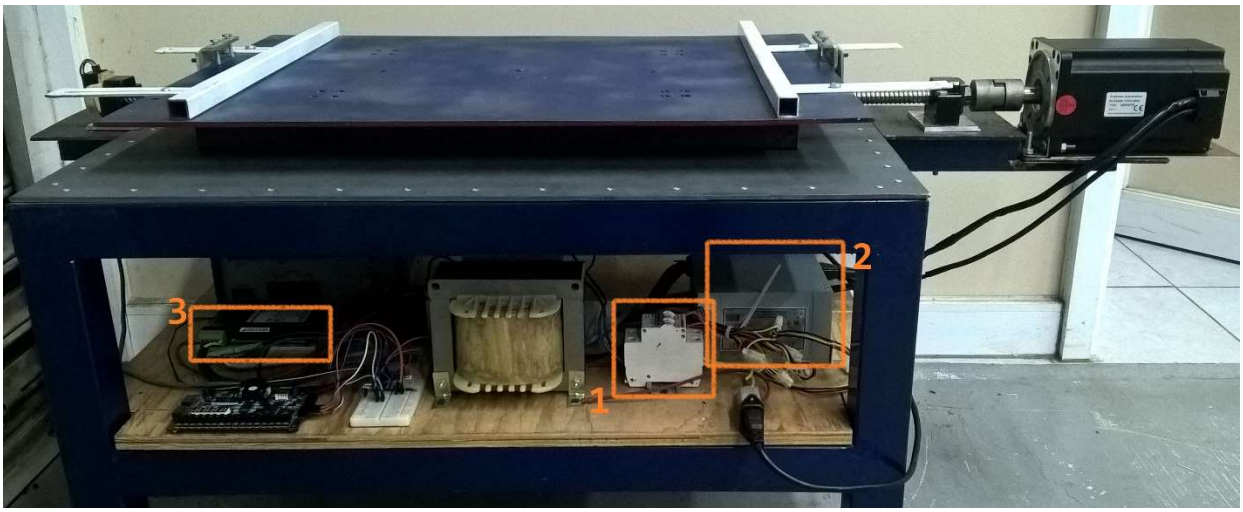


Figura A. Vista general del *Sistema Generador de Vibraciones.*

1. Interruptor principal de alto voltaje.
2. Fuente de alimentación.
3. Servo Driver.

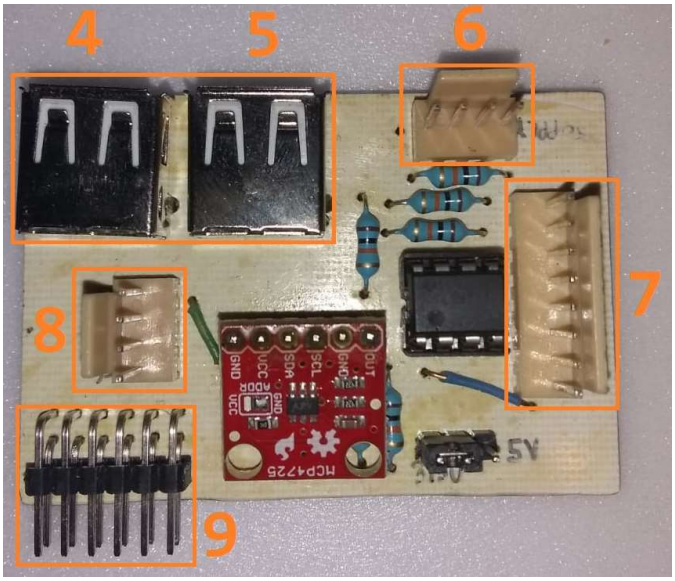


Figura B. Vista de la tarjeta impresa.

4. Conexión para Acelerómetro 1.
5. Conexión para Acelerómetro 2.
6. Conexión para alimentación de voltaje.
7. Conexión referencia para Servo Driver.
8. Conexión para Encoder.
9. Conexión para la tarjeta Basys.

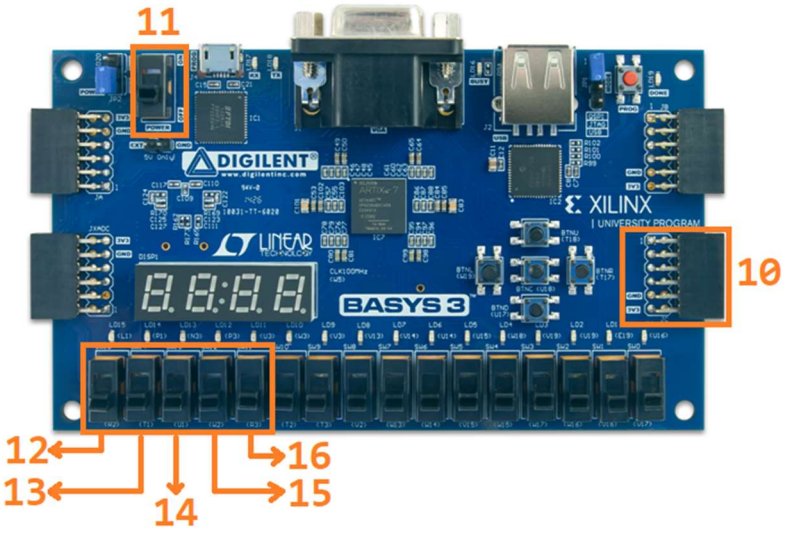


Figura C. Vista de la tarjeta FPGA *Basys 3*.

10. Puerto JC para el circuito impreso.
11. Interruptor principal de la tarjeta FPGA.
12. Interruptor Reset maestro.
13. Interruptor Eje del acelerómetro 1.
14. Interruptor Eje del acelerómetro 2.
15. Interruptor de configuración del acelerómetro para ±8g.
16. Interruptor para activar o desactivar el controlador.

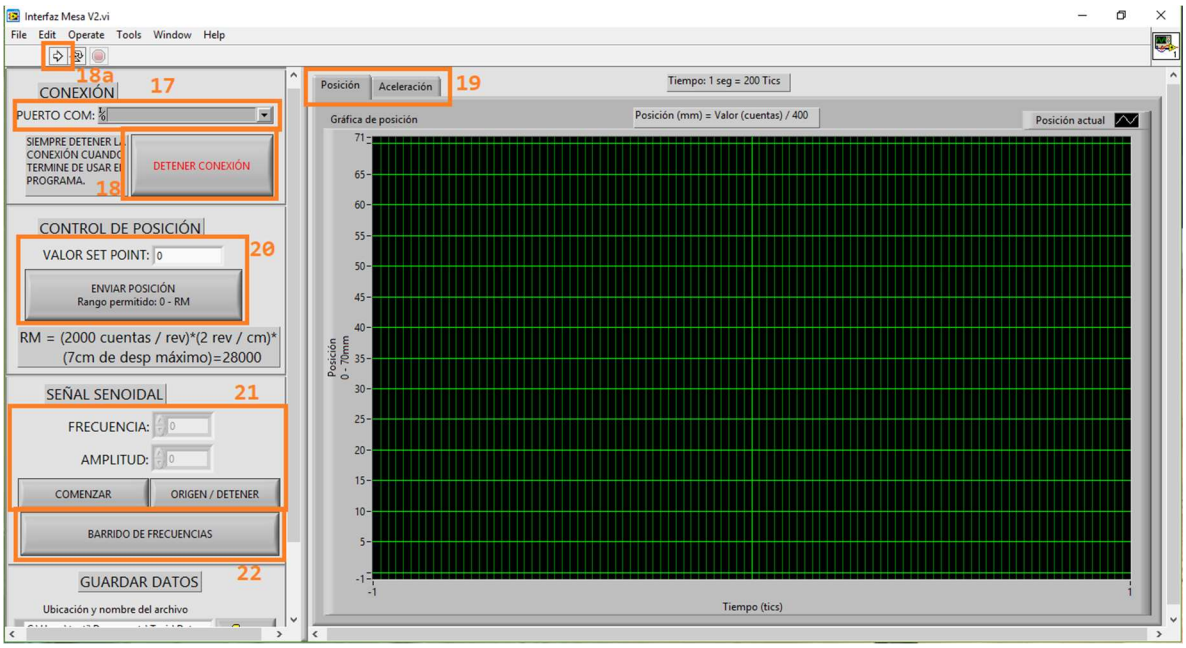


Figura D. Vista de la interfaz gráfica.

17. Selector del puerto para la tarjeta.
18. Botón para detener la conexión/ 18a. Botón para iniciar conexión.
19. Selector de visualización de posición y aceleración.
20. Modo de operación por escalón.
21. Modo de operación por referencia senoidal.
22. Modo de operación por barrido de frecuencias.

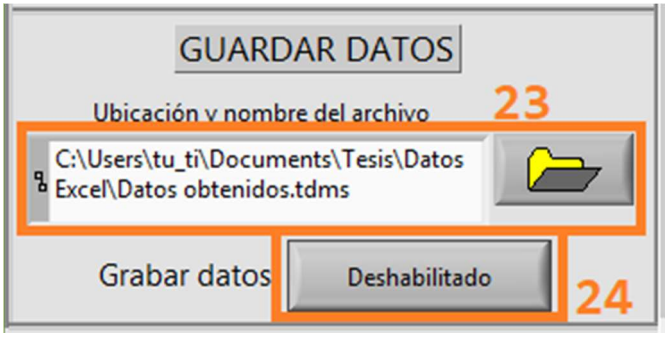


Figura E. Detalle de la interfaz gráfica para guardar datos.

23. Selector de ruta para guardar los datos.
24. Botón para inicio de recopilación de datos.

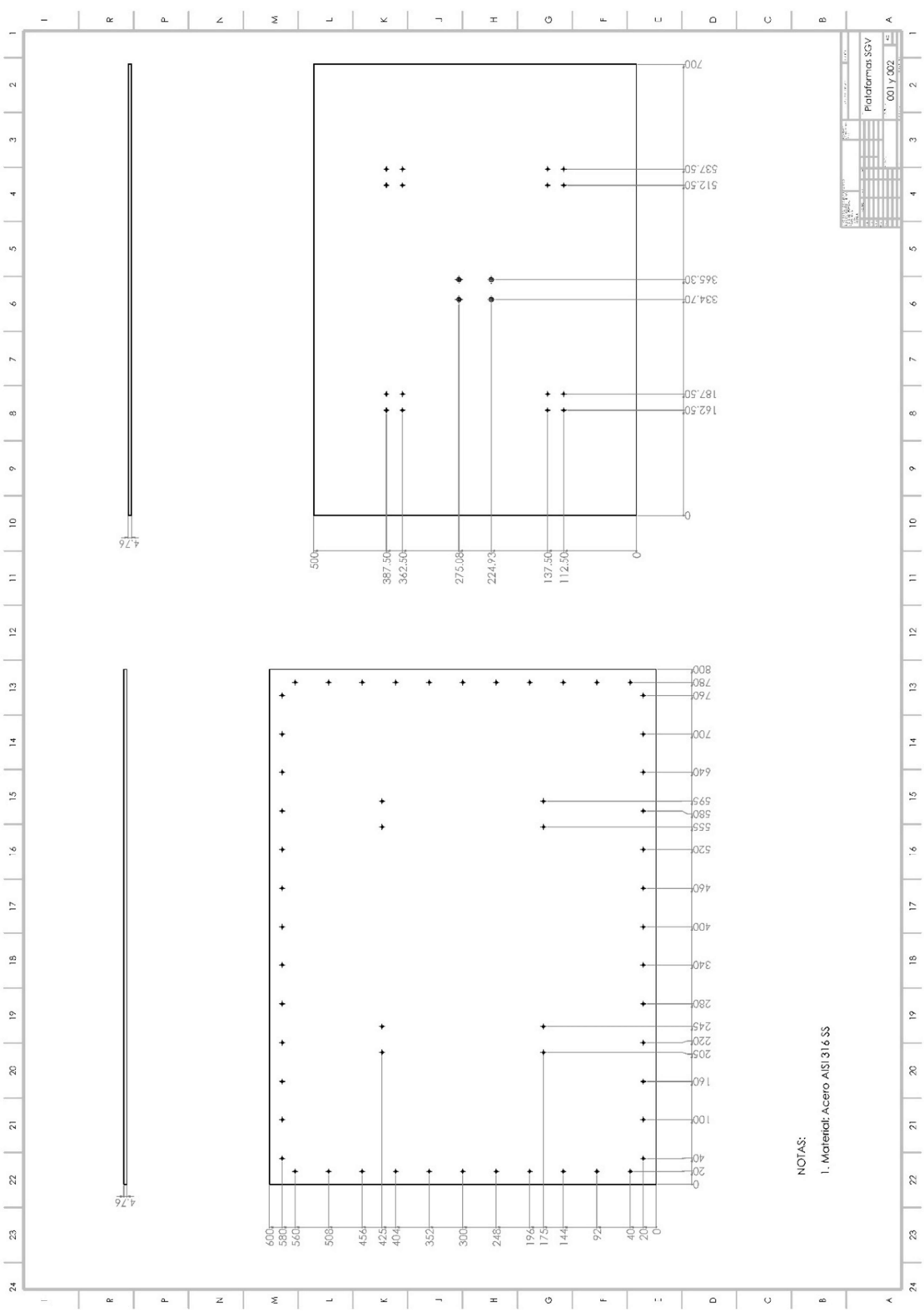
PASOS A SEGUIR

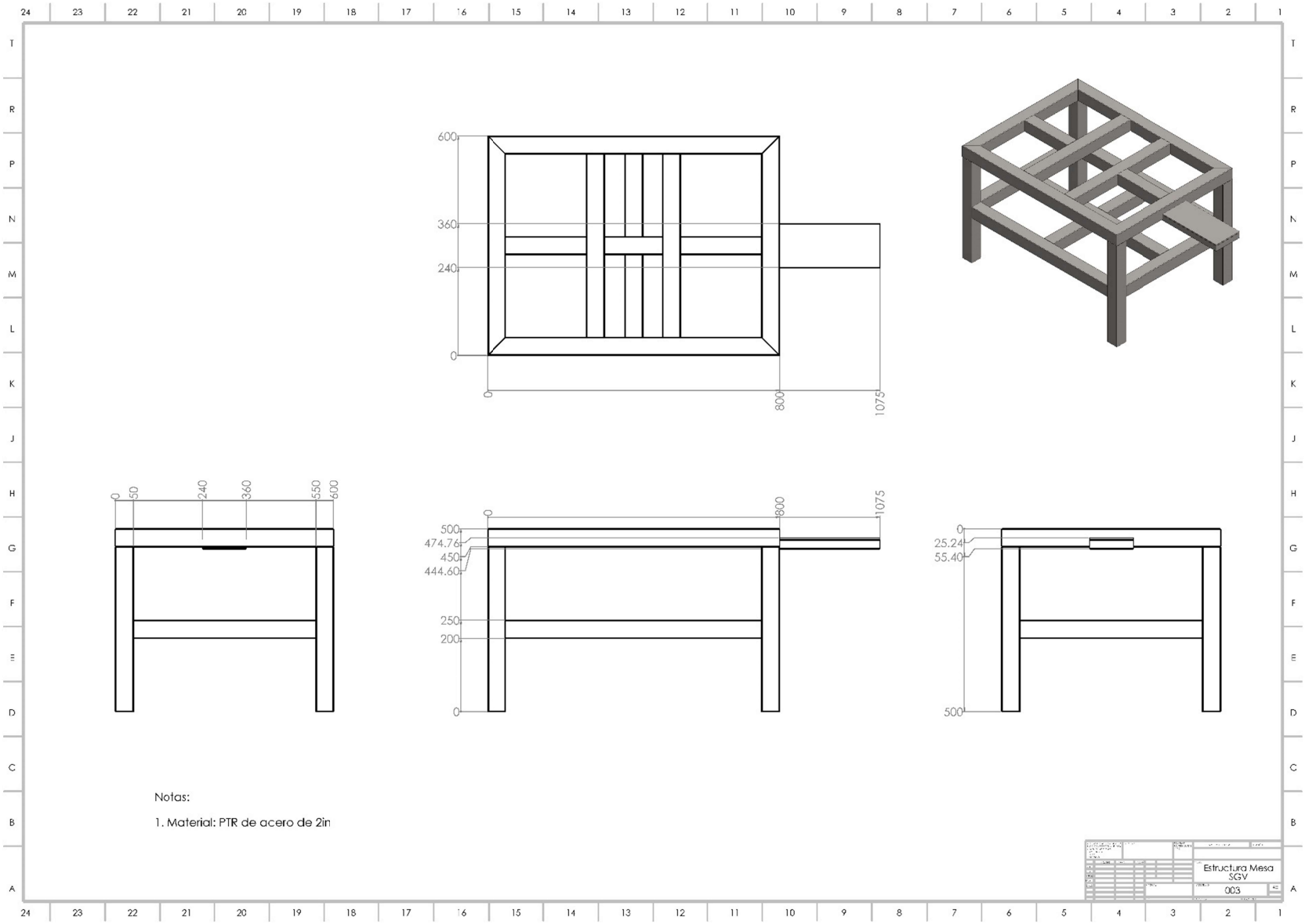
1) Conectar los pines del circuito impreso (**9**) con la tarjeta Basys 3 (**10**).

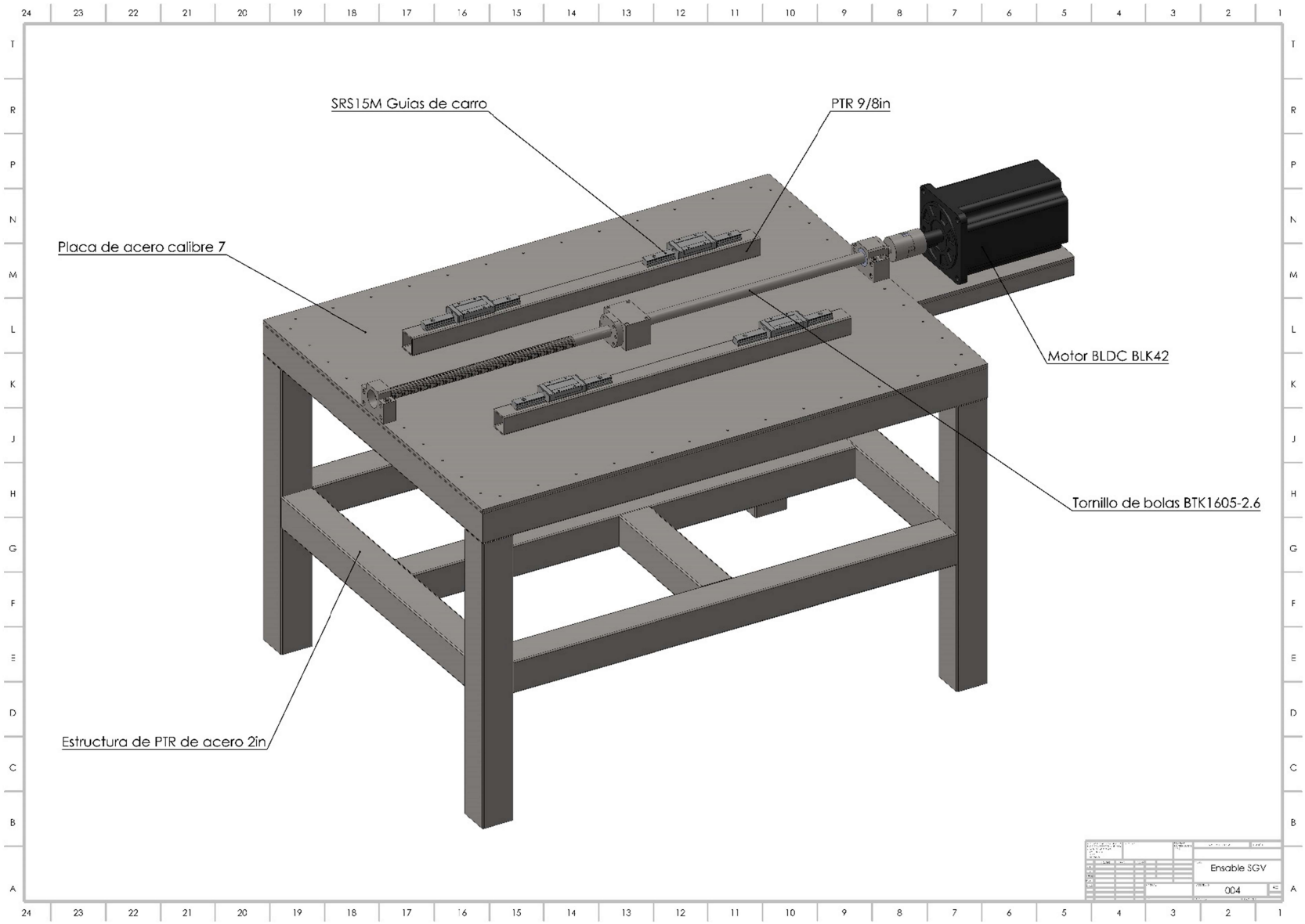
2) Conectar al circuito impreso la alimentación de voltaje proveniente de la fuente (**2** a **6**).
3) Conectar al circuito impreso el encoder (**8**).
4) Conectar al circuito impreso la referencia que se dirige al Servo Driver (**7** con **9**).
5) Conectar al circuito impreso loa acelerómetros en las posiciones **4** y **5**.
6) Conectar la tarjeta Basys 3 a la computadora mediante un cable USB.
7) Abrir en la computadora el programa *Interfaz Mesa V2*.
8) Encender la tarjeta Basys con el interruptor **11**. La Basys encenderá algunos leds y un solo dígito en los displays.
9) Asegurarse que el interruptor **15** esté apagado (Que esté como en la *Figura C*), después, encender el programa en la tarjeta para verificar que funciona (interruptor **12)**, si el programa funciona los displays se encienden completamente, en este paso llevar la plataforma a su punto inicial, el cual es en un extremo (El extremo depende de cómo se conectó el encoder, por default es la izquierda) y ver como se despliega el valor del encoder.
10) Volver a apagar el programa con el interruptor **12**, una vez apagado enciende el interruptor **15** para que la próxima vez que se encienda el programa se configuren los acelerómetros.
11) Encender la fuente de alimentación (**2**) y verificar que encienden los acelerómetros y en DAC.
12) Buscar el Puerto COM (**17**), normalmente la Basys presenta dos puertos, siempre es el segundo.
13) Iniciar conexión con el botón en la posición **18a**.
14) Cuando se inicia la conexión se tiene unos segundos para iniciar el programa en la Basys con el interruptor **12**. De lo contrario la interfaz presentará una falla.
15) Una vez que el programa inicie se verá como las gráficas se mueven, en este punto se puede encender el interruptor **1**, verificar que el Servo Driver (**3**) enciende un led en verde. El motor no debería de presentar movimiento, de lo contrario apague el interruptor **1** y verifique las salidas del circuito impreso.
16) En este punto la plataforma seguirá las instrucciones del usuario desde la interfaz, usted podrá manipular la plataforma usando los distintos *Modos de Operación* (**20**, **21** y **22**) y observando las aceleraciones en la pestaña trasera (**19**).
17) Por default, cuando los interruptores **13** y **14** están apagados los acelerómetros registran el *Eje X*, si se activan se registrará el *Eje Y*. El interruptor **13** controla el acelerómetro conectado en la posición **4**, por ende el interruptor **14** controla el acelerómetro en la posición **5**.
18) Para detener el uso del dispositivo se deberá primero apagar siempre el interruptor **1** seguido de presionar en la interfaz el botón **18** y finalmente el interruptor **12** de la Basys.

NOTA: Si en algún momento la Basys pierde energía asegurarse de apagar el interruptor **1** antes de cualquier otra cosa para evitar accidentes. Las pruebas de seguridad muestran que el motor no debería de moverse, pero de cualquier manera asegurarse de apagarlo.

A2 Modelo CAD




600
360
240
0
0
800
1075
0
50
240
360
550
600
500
474.76
450
444.60
250
200
0
0
800
1075
0
25.24
55.40
500
Notas:
1. Material: PTR de acero de 2in
Estructura Mesa SGV
003


SRS15M Guias de carro
PTR 9/8in
Placa de acero calibre 7
Motor BLDC BLK42
Tornillo de bolas BTK1605-2.6
Estructura de PTR de acero 2in
Ensable SGV
004

A3 Código implementado en el FPGA

```vhdl
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

entity MesaVibratoria is
    generic( n : integer := 8);
    port(
    CLK  : in std_logic;
    RST  : in std_logic;
    CHA  : in std_logic;
    CHB  : in std_logic;
    SW1  : in std_logic;
    SW2  : in std_logic;
    SW3  : in std_logic;
    SW4  : in std_logic;
    RXD  : in std_logic;
    TXD  : out std_logic;
    SCL  : out std_logic;
    SDA  : inout std_logic;
    SCL1 : out std_logic;
    SDA1 : inout std_logic;
    SCL2 : out std_logic;
    SDA2 : inout std_logic;
    DOUT : out std_logic_vector(15 downto 0);
    BC7  : out std_logic_vector(6 downto 0);
    SEL  : out std_logic_vector(3 downto 0)
    );
end MesaVibratoria;

architecture TOP of MesaVibratoria is
Component TOPUART is
    generic( n : integer := 8);
    port(
    CLK  : in std_logic;
    RST  : in std_logic;
    STT  : in std_logic;
    RXD  : in std_logic;
    TXD  : out std_logic;
    EOT  : out std_logic;
    DIN  : in std_logic_vector(n - 1 downto 0);
    DOUT : out std_logic_vector(n - 1 downto 0)
    );
end Component;
Component RisingEdge is
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    XIN  : in std_logic;
    XOUT : out std_logic
    );
end Component;
Component LoadRegister is
    generic(n : integer := 40);
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    SHF  : in std_logic;
    DIN  : in std_logic_vector(n - 1 downto 0);
    DOUT : out std_logic_vector(n - 1 downto 0)
    );
end Component;
Component ShiftRegister is
    generic(n : integer := 8);
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    SHF  : in std_logic;
    BIN  : in std_logic_vector(n - 1 downto 0);
    DOUT : out std_logic_vector((n + n + n) - 1 downto 0)
```

```
70      );
71  end Component;
72  Component Counter is
73      generic(TICKS : integer := 2);
74      port(
75      RST : in std_logic;
76      CLK : in std_logic;
77      TIC : in std_logic;
78      SYN : out std_logic
79      );
80  end Component;
81  Component Encoder is
82      port(
83      CLK  : in std_logic;
84      RST  : in std_logic;
85      CHA  : in std_logic;
86      CHB  : in std_logic;
87      POSI : out std_logic_vector(31 downto 0);
88      BC7  : out std_logic_vector(6 downto 0);
89      SEL  : out std_logic_vector(3 downto 0)
90      );
91  end Component;
92  Component Suma is
93      port(
94      DINA : in std_logic_vector(11 downto 0);
95      DINB : in std_logic_vector(11 downto 0);
96      DOUT : out std_logic_vector(11 downto 0)
97      );
98  end Component;
99  Component Timer is
100     generic(TICKS : integer := 10);
101     port(
102     RST : in std_logic;
103     CLK : in std_logic;
104     SYN : out std_logic
105     );
106 end Component;
107
108 Component Multiplexorx is
109     port(
110     DINA : in std_logic_vector(11 downto 0);
111     SEL  : in std_logic_vector(1 downto 0);
112     DOUT : out std_logic_vector(11 downto 0)
113     );
114 end Component;
115 Component Saturador is
116     port(
117     DIN  : in std_logic_vector(31 downto 0);
118     DOUT : out std_logic_vector(11 downto 0)
119     );
120 end Component;
121 Component Substract is
122     generic(n : integer := 16);
123     port(
124     DIN1 : in std_logic_vector(n - 1 downto 0);
125     DIN2 : in std_logic_vector(n - 1 downto 0);
126     DOUT : out std_logic_vector(n - 1 downto 0)
127     );
128 end Component;
129 Component ICuadradaC is
130     port(
131     RST : in std_logic;
132     CLK : in std_logic;
133     STT : in std_logic;
134     SCL : out std_logic;
135     SDA : inout std_logic;
136     DTO : in std_logic_vector(11 downto 0)
137     );
138 end Component;
```

```
139   Component TopControl is
140       port(
141       RST  : in std_logic;
142       CLK  : in std_logic;
143       TS   : in std_logic;
144       XIN  : in std_logic_vector(31 downto 0);
145       XOUT : out std_logic_vector(47 downto 0)
146       );
147   end Component;
148   Component MuxPosicion is
149       generic(n : integer := 16);
150       port(
151       DIN1 : in std_logic_vector(n - 1 downto 0);
152       DIN2 : in std_logic_vector(n - 1 downto 0);
153       SEL  : in std_logic_vector(1 downto 0);
154       DOUT : out std_logic_vector(n - 1 downto 0)
155       );
156   end Component;
157   Component FreeCounter is
158       generic ( n : integer := 2);
159       port(
160       RST  : in std_logic;
161       CLK  : in std_logic;
162       ENA  : in std_logic;
163       COUT : out std_logic_vector(n downto 0)
164       );
165   end Component;
166   Component SRLatch is
167       port(
168       RST  : in std_logic;
169       CLK  : in std_logic;
170       SET  : in std_logic;
171       CLR  : in std_logic;
172       QOUT : out std_logic
173       );
174   end Component;
175   Component ElMux is
176       port(
177       SEL  : in std_logic_vector(3 downto 0);
178       DOUT : out std_logic_vector(15 downto 0)
179       );
180   end Component;
181   Component MuxData is
182       port(
183       DATA : in std_logic_vector(7 downto 0);
184       DATB : in std_logic_vector(7 downto 0);
185       DATC : in std_logic_vector(7 downto 0);
186       DATD : in std_logic_vector(7 downto 0);
187       DATE : in std_logic_vector(7 downto 0);
188       DATF : in std_logic_vector(7 downto 0);
189       SEL  : in std_logic_vector(2 downto 0);
190       DOUT : out std_logic_vector(7 downto 0)
191       );
192   end Component;
193   Component MPU6050 is
194       port(
195       RST : in std_logic;
196       CLK : in std_logic;
197       SWI : in std_logic;
198       SW2 : in std_logic;
199       SCL : out std_logic;
200       SDA : inout std_logic;
201       DTA : out std_logic_vector(15 downto 0)
202       );
203   end Component;
204   Component Compara is
205       port(
206       COMP : in std_logic_vector(7 downto 0);
207       RES  : out std_logic_vector(3 downto 0)
```

```
208         );
209     end Component;
210     Component MUXVel is
211         port(
212         SEL  : in std_logic_vector(7 downto 0);
213         VER : out std_logic_vector(3 downto 0);
214         DOUT : out std_logic_vector(23 downto 0)
215         );
216     end Component;
217     Component TimerVel is
218         port(
219         TICKS : in std_logic_vector(23 downto 0);
220         RST : in std_logic;
221         CLK : in std_logic;
222         SYN : out std_logic
223         );
224     end Component;
225     Component Multiplier is
226         generic(n : integer := 16; m : integer := 8);
227         port(
228         DIN1 : in std_logic_vector(n - 1 downto 0);
229         DIN2 : in std_logic_vector(m - 1 downto 0);
230         DOUT : out std_logic_vector((n + m) - 1 downto 0)
231         );
232     end Component;
233     Component Multiplexor2a1 is
234         generic(n : integer := 16);
235         port(
236         DIN1 : in std_logic_vector(n - 1 downto 0);
237         DIN2 : in std_logic_vector(n - 1 downto 0);
238         SEL  : in std_logic;
239         DOUT : out std_logic_vector(n - 1 downto 0)
240         );
241     end Component;
242     Component Adder is
243         generic(n : integer := 16);
244         port(
245         DIN1 : in std_logic_vector(n - 1 downto 0);
246         DIN2 : in std_logic_vector(n - 1 downto 0);
247         DOUT : out std_logic_vector(n - 1 downto 0)
248         );
249     end Component;
250     Component SaturadorP is
251         port(
252         DIN  : in std_logic_vector(47 downto 0);
253         DOUT : out std_logic_vector(11 downto 0)
254         );
255     end Component;
256     Component MuxDatos is
257         port(
258         DIN : in std_logic_vector(5 downto 0);
259         DOUT : out std_logic_vector(15 downto 0)
260         );
261     end Component;
262     signal PUT, EOT, FIN, SYA, STT, TIC, RSR, TLI, CTO1, CTA1, CLT, RTT, TIG, ETT, DDO,
    DTR, VSE, VRE, CAM, TS : std_logic;
263     signal MUX, DUN : std_logic_vector(1 downto 0);
264     signal SDE : std_logic_vector(2 downto 0);
265     signal SES, VER, RES, VFR : std_logic_vector(3 downto 0);
266     signal DINT : std_logic_vector(5 downto 0);
267     signal DIN, DPOS, DATA, COMP, VFE, VVS : std_logic_vector(7 downto 0);
268     signal EROR, DSMI, DCPE, DSMIE, DSMI1, DSMI2 : std_logic_vector(11 downto 0);
269     signal DSET, DMUX, DAC1, DAC2, RMU, MU1, MU2, RRE : std_logic_vector(15 downto 0);
270     signal DATO, DOUQ, VEL : std_logic_vector(23 downto 0);
271     signal DSAL, ERR, DSETI : std_logic_vector(31 downto 0);
272     signal CTL : std_logic_vector(47 downto 0);
273
274     begin
275         DOUT <= DOUQ(23 downto 8);
```

```
276         DSETI <= "0000000000000000" & DSET;
277         COMP <= DOUQ(7 downto 0);
278         DUN <= RES(3) & RES(2);
279         DDO <= RES(1);
280         DTR <= RES(3);
281         VSE <= RES(0);
282         VFE <= "0000" & VFR;
283
284     --Transmisión y recepción de datos
285         U01 : TOPUART generic map(8) port map(CLK, RST, TIG, RXD, TXD, EOT, DIN, DATA);
            --Protocolo UART
286         U02 : Timer generic map(500000) port map(RST, CLK, PUT);
            --Velocidad de transmisión master 312500
287         U03 : RisingEdge port map(RST, CLK, EOT, FIN);
            --Final de recepción
288         U04 : ShiftRegister port map(RST, CLK, FIN, DATA, DATO);
            --Toma los primero 8 bits de datos
289         U05 : LoadRegister generic map(24) port map(RST, CLK, SYA, DATO, DOUQ);
            --Toma datos de byte en byte para concatenar
290         U06 : Counter generic map(3) port map(RST, CLK, FIN, SYA);
            --Número de datos a recibir
291
292     --Encoder y resta
293         U07 : Encoder port map(CLK, RST, CHA, CHB, DSAL, BC7, SEL);
            --Lectura del encoder
294         U08 : Substract generic map(32) port map(DSETI, DSAL, ERR);
            --Diferencia entre la posición y es set point
295
296     --Escritura al DAC
297         U09 : ICuadradaC port map(RST, CLK, STT, SCL, SDA, DSMI);
            --Escribe el dato en el DAC
298         U10 : Timer generic map(500000) port map(RST, CLK, STT);
            --Velocidad de escritura en el DAC
299         U11 : Saturador port map(ERR, DSMI1);
            --Satura la cantidad de bits a 12
300
301     --Escritura a la interfaz gráfica
302         U12 : SRLatch port map (RST, CLK, PUT, CLT, RTT);
            --Enclava la transmisión
303         U13 : Timer generic map(50000) port map(RTT, CLK, TIG);
            --Velocidad de transmisión slave 13020
304         U14 : FreeCounter generic map(2) port map(RTT, CLK, TIG, SDE);
            --Array que entra como selector del dato al mux
305         U15 : Counter generic map(6) port map(RTT, CLK, TIG, CLT);
            --Número de datos a enviar
306         U16 : MuxData port map(DSAL(15 downto 8), DSAL(7 downto 0), DAC1(7 downto 0),
            DAC1(15 downto 8), DAC2(7 downto 0), DAC2(15 downto 8), SDE, DIN);
307
308     --Lectura de acelerómetros
309         U17 : MPU6050 port map(RST, CLK, SW1, SW3, SCL1, SDA1,
            DAC1);                                  --Acelerometro 1
310         U18 : MPU6050 port map(RST, CLK, SW2, SW3, SCL2, SDA2,
            DAC2);                                  --Acelerometro 2
311
312     --Asignación de valores de entrada
313         U19 : Compara port map(COMP, RES);
            --Determina si es dato unico o parámetro
314         U20 : MuxPosicion generic map(16) port map(DOUQ(23 downto 8), RMU, DUN,
            DSET);          --Da paso al valor enviado o al banco de valores
315         U21 : SRLatch port map(RST, CLK, DDO, DTR, RSR);
            --Enclava la señal para enviar valores
316         U22 : TimerVel port map(VEL, RSR, CLK, TIC);
            --Velocidad de cambio o frecuencia (6250000 = 1Hz)
317         U23 : FreeCounter generic map(5) port map(RST, CLK, TIC, DINT);
318         U24 : MuxDatos port map(DINT, DMUX);                                    --Valores
            preestablecidos para senoidal
319     --      U25 : MUXVel port map(DOUQ(15 downto 8), VER, VEL);
320
321     --Codigo para variar frecuencia y amplitud
```

```
322       U25 : MUXVel port map(VVS, VER, VEL);                                    --Transforma el
          valor de velocidad en binario
323       U26 : Multiplier generic map(3, 13) port map(DOUQ(18 downto 16), DMUX(12 downto 0),
          MU1); --Para variar la amplitud
324       U27 : Multiplier generic map(3, 13) port map(DOUQ(18 downto 16), "0011111010000",
          MU2);
325       U28 : Substract generic map(16) port map("0011011010110000", MU2, RRE);
326       U29 : Adder generic map(16) port map(MU1, RRE, RMU);
327       U30 : SRLatch port map(RST, CLK, VSE, DTR, VRE);
          --Setear barrido de frecuencias
328       U31 : Timer generic map(500000000) port map(VRE, CLK, CAM);
          --Marca el cambio de frecuencia cada 5seg
329       U32 : FreeCounter generic map(3) port map(VRE, CLK, CAM, VFR);
          --Cuenta de 0 a 15hz (VFE es equvalente a DOUQ(15 downto 8))
330       U33 : Multiplexor2a1 generic map(8) port map(DOUQ(15 downto 8), VFE, VSE, VVS);
331
332   --El siguiente código es el controlador
333       U34 : TopControl port map(RST, CLK, TS, ERR, CTL);
334       U35 : Timer generic map(100000) port map(RST, CLK, TS);
335       U36 : SaturadorP port map(CTL, DSMI2);
336       U37 : Multiplexor2a1 generic map(12) port map(DSMI1, DSMI2, SW4, DSMI);
          --Saturador 1 y Saturador P
337
338   end TOP;
339
340   --Revisión 7.01 27 de Abril de 2019
341   --Funciona con la interfaz modificad
342   --Lee los dos acelerómetros y los grafica
343   --Los acelerometros cambian de eje con los switch 1 y 2
344   --Se requiere reinicio y activar el Switch 3 para cambiar a +-8g
345   --Se le agrega un controlador PID que se activa con el Switch No.4
346   --El control proporcional base tiene una Kp=0.25
```

```vhdl
Library IEEE;
use IEEE.std_logic_1164.all;

entity TOPUART is
    generic( n : integer := 8);
    port(
    CLK  : in std_logic;
    RST  : in std_logic;
    STT  : in std_logic;
    RXD  : in std_logic;
    TXD  : out std_logic;
    EOT  : out std_logic;
    DIN  : in std_logic_vector(n - 1 downto 0);
    DOUT : out std_logic_vector(n - 1 downto 0)
    );
end TOPUART;

architecture TOP of TOPUART is
Component LoadRegister is
    generic(n : integer := 40);
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    SHF  : in std_logic;
    DIN  : in std_logic_vector(n - 1 downto 0);
    DOUT : out std_logic_vector(n - 1 downto 0)
    );
end Component;
Component Parity is
    port(
    DIN : in std_logic_vector(7 downto 0);
    DOUT : out std_logic
    );
end Component;
Component Multiplexoru is
    port(
    DIN : in std_logic_vector(7 downto 0);
    PAR : in std_logic;
    SEL : in std_logic_vector(3 downto 0);
    DOUT : out std_logic
    );
end Component;
Component SRLatch is
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    SET  : in std_logic;
    CLR  : in std_logic;
    QOUT : out std_logic
    );
end Component;
Component Timer is
    generic(TICKS : integer := 10);
    port(
    RST : in std_logic;
    CLK : in std_logic;
    SYN : out std_logic
    );
end Component;
Component FreeCounter is
    generic ( n : integer := 2);
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    ENA  : in std_logic;
    COUT : out std_logic_vector(n downto 0)
    );
end Component;
Component ProgramCounter is
```

```
70      generic ( n : integer := 10);
71      port(
72      RST : in std_logic;
73      CLK : in std_logic;
74      ENA : in std_logic;
75      RDY : out std_logic
76      );
77  end Component;
78  Component FallingEdge is
79      port(
80      RST  : in std_logic;
81      CLK  : in std_logic;
82      XIN  : in std_logic;
83      XOUT : out std_logic
84      );
85  end Component;
86  Component RegistroSerialParalelo is
87      generic(n : integer := 40);
88      port(
89      RST  : in std_logic;
90      CLK  : in std_logic;
91      SHF  : in std_logic;
92      BIN  : in std_logic;
93      DOUT : out std_logic_vector(n - 1 downto 0)
94      );
95  end Component;
96
97  signal DPE : std_logic_vector(n - 1 downto 0);
98  signal SEL : std_logic_vector(3 downto 0);
99  signal DTA, DAUX : std_logic_vector(n downto 0);
100 signal PAR, CLL, RSS, TIC, SET, CLC, RET, CLS, RLD, TCC : std_logic;
101 begin
102     EOT <= NOT RLD;
103     DOUT <= DAUX(7 downto 0);
104     U01 : LoadRegister generic map(8) port map(RST, CLK, STT, DIN, DPE);
105     U02 : Parity port map(DPE, PAR);
106     U03 : Multiplexoru port map(DPE, PAR, SEL, TXD);
107     U04 : SRLatch port map(RST, CLK, STT, CLL, RSS);
108     U05 : Timer generic map(868) port map(RSS, CLK, TIC);
109     U06 : FreeCounter generic map(3) port map(RSS, CLK, TIC, SEL);
110     U07 : ProgramCounter generic map(11) port map(RSS, CLK, TIC, CLL);
111
112     U08 : FallingEdge port map(RST, CLK, RXD, SET);
113     U09 : SRLatch port map(RST, CLK, SET, CLC, RET);
114     U10 : Timer generic map(414) port map(RET, CLK, CLC);
115     U11 : SRLatch port map(RST, CLK, CLC, CLS, RLD);
116     U12 : Timer generic map(868) port map(RLD, CLK, TCC);
117     U13 : RegistroSerialParalelo generic map(9) port map(RST, CLK, TCC, RXD, DTA);
118     U14 : LoadRegister generic map(9) port map(RST, CLK, CLS, DTA, DAUX);
119     U15 : ProgramCounter generic map(10) port map(RLD, CLK, TCC, CLS);
120 end TOP;
```

```vhdl
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

Entity Encoder is
    port(
    CLK  : in std_logic;
    RST  : in std_logic;
    CHA  : in std_logic;
    CHB  : in std_logic;
    POSI : out std_logic_vector(31 downto 0);
    BC7  : out std_logic_vector(6 downto 0);
    SEL  : out std_logic_vector(3 downto 0)
    );
end Encoder;

Architecture Behavioral of Encoder is

Component BidirectionalCounter is
    generic( n : integer := 16);
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    ENA  : in std_logic;
    DIR  : in std_logic;
    COUT : out std_logic_vector(n - 1 downto 0)
    );
end Component;

Component QuadratureDecoder is
    generic( n : integer := 3);
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    CHA  : in std_logic;
    CHB  : in std_logic;
    ENA  : out std_logic;
    DIR  : out std_logic
    );
end Component;

Component Segmentos is
    port(
    DIN  : in std_logic_vector(3 downto 0);
    DOUT : out std_logic_vector(6 downto 0)
    );
end Component;

Component TimerSeg is
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    ENA  : in std_logic;
    SE1  : in std_logic_vector(6 downto 0);
    SE2  : in std_logic_vector(6 downto 0);
    SE3  : in std_logic_vector(6 downto 0);
    SE4  : in std_logic_vector(6 downto 0);
    SEG  : out std_logic_vector(6 downto 0);
    COUT : out std_logic_vector(3 downto 0)
    );
end Component;

Component Timer is
    generic(TICKS : integer := 10);
    port(
    RST : in std_logic;
    CLK : in std_logic;
    SYN : out std_logic
    );
```

```
70   end Component;
71   signal ENA, DIR, TIC : std_logic;
72   signal BC1, BC2, BC3, BC4 : std_logic_vector(6 downto 0);
73   signal BD1, BD2, BD3, BD4 : std_logic_vector(3 downto 0);
74   signal POST : std_logic_vector(31 downto 0);
75   begin
76       U01 : BidirectionalCounter generic map(32) port map(RST, CLK, ENA, DIR, POST);
77       U02 : QuadratureDecoder generic map(3) port map(RST, CLK, CHA, CHB, ENA, DIR);
78       POSI <= POST;
79       BD1 <= POST(3 downto 0);
80       BD2 <= POST(7 downto 4);
81       BD3 <= POST(11 downto 8);
82       BD4 <= POST(15 downto 12);
83       U03 : Segmentos port map(BD1, BC1);
84       U04 : Segmentos port map(BD2, BC2);
85       U05 : Segmentos port map(BD3, BC3);
86       U06 : Segmentos port map(BD4, BC4);
87       U07 : TimerSeg port map(RST, CLK, TIC, BC1, BC2, BC3, BC4, BC7, SEL);
88       U08 : Timer generic map(1000000) port map(RST, CLK, TIC);
89   end Behavioral;
```

```vhdl
Library IEEE;
use IEEE.std_logic_1164.all;

Entity ICuadradaC is
    port(
    RST : in std_logic;
    CLK : in std_logic;
    STT : in std_logic;
    SCL : out std_logic;
    SDA : inout std_logic;
    DTO : in std_logic_vector(11 downto 0)
    );
end ICuadradaC;

Architecture TOP of ICuadradaC is
Component FSM is
    port(
    RST : in std_logic;
    CLK : in std_logic;
    TIC : in std_logic;
    STT : in std_logic;
    FFE : in std_logic;
    TAC : in std_logic;
    SET : out std_logic;
    RPC : out std_logic
    );
end Component;
Component SRLatch is
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    SET  : in std_logic;
    CLR  : in std_logic;
    QOUT : out std_logic
    );
end Component;
Component Timer is
    generic(TICKS : integer := 10);
    port(
    RST : in std_logic;
    CLK : in std_logic;
    SYN : out std_logic
    );
end Component;
Component ProgramCounter is
    generic ( n : integer := 10);
    port(
    RST : in std_logic;
    CLK : in std_logic;
    ENA : in std_logic;
    RDY : out std_logic
    );
end Component;
Component Toggle is
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    SET  : in std_logic;
    QOUT : out std_logic
    );
end Component;
Component FreeCounter is
    generic ( n : integer := 12);
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    ENA  : in std_logic;
    COUT : out std_logic_vector(n downto 0)
    );
```

```
70   end Component;
71   Component MUX is
72       port(
73       DINA : in std_logic_vector(6 downto 0);
74       DINB : in std_logic_vector(11 downto 0);
75       SEL  : in std_logic_vector(6 downto 0);
76       DOUT : out std_logic
77       );
78   end Component;
79   Component RegistroSerialParalelo is
80       generic(n : integer := 40);
81       port(
82       RST  : in std_logic;
83       CLK  : in std_logic;
84       SHF  : in std_logic;
85       BIN  : in std_logic;
86       DOUT : out std_logic_vector(n - 1 downto 0)
87       );
88   end Component;
89   Component LoadRegister is
90       generic(n : integer := 40);
91       port(
92       RST  : in std_logic;
93       CLK  : in std_logic;
94       SHF  : in std_logic;
95       DIN  : in std_logic_vector(n - 1 downto 0);
96       DOUT : out std_logic_vector(n - 1 downto 0)
97       );
98   end Component;
99   signal TIC, FFE, TAC, SET, RPC, RTP, CLS, SNN, SYN, FE: std_logic;
100  Signal SEL : std_logic_vector(6 downto 0);
101  begin
102      SYN <= NOT SNN;
103      SCL <= SYN;
104      FE <= TAC AND SYN;
105      U01 : FSM port map(RST, CLK, TIC, STT, FFE, TAC, SET, RPC);
106      U02 : SRLatch port map(RST, CLK, STT, CLS, RTP);
107      U03 : Timer generic map(500) port map(RTP, CLK, TIC);
108      U04 : ProgramCounter generic map(83) port map(RTP, CLK, TIC, CLS);  --109
109      U05 : Timer generic map(500) port map(SET, CLK, TAC);
110      U06 : Toggle port map(RST, CLK, TAC, SNN);
111      U07 : ProgramCounter generic map(9) port map(RPC, CLK, FE, FFE);
112      U08 : FreeCounter generic map(6) port map(RTP, CLK, TIC, SEL);
113      U09 : MUX port map("1100000", DTO, SEL, SDA);
114  end TOP;
```

```vhdl
Library IEEE;
use IEEE.std_logic_1164.all;

Entity MPU6050 is
    port(
    RST : in std_logic;
    CLK : in std_logic;
    SWI : in std_logic;
    SW2 : in std_logic;
    SCL : out std_logic;
    SDA : inout std_logic;
    DTA : out std_logic_vector(15 downto 0)
    );
end MPU6050;

Architecture TOP of MPU6050 is
Component MPUFSM is
    port(
    RST : in std_logic;
    CLK : in std_logic;
    TIC : in std_logic;
    STT : in std_logic;
    FFE : in std_logic;
    TAC : in std_logic;
    CLS : in std_logic;
    PCR : in std_logic;
    NSE : out std_logic;
    NTI : in std_logic;
    SET : out std_logic;
    RPC : out std_logic
    );
end Component;
Component SRLatch is
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    SET  : in std_logic;
    CLR  : in std_logic;
    QOUT : out std_logic
    );
end Component;
Component Timer is
    generic(TICKS : integer := 10);
    port(
    RST : in std_logic;
    CLK : in std_logic;
    SYN : out std_logic
    );
end Component;
Component ProgramCounter is
    generic ( n : integer := 10);
    port(
    RST : in std_logic;
    CLK : in std_logic;
    ENA : in std_logic;
    RDY : out std_logic
    );
end Component;
Component Toggle is
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    SET  : in std_logic;
    QOUT : out std_logic
    );
end Component;
Component FreeCounter is
    generic ( n : integer := 12);
    port(
```

```
70        RST  : in std_logic;
71        CLK  : in std_logic;
72        ENA  : in std_logic;
73        COUT : out std_logic_vector(n downto 0)
74        );
75    end Component;
76    Component MPUMUX is
77        port(
78        DINA : in std_logic_vector(6 downto 0);
79        DINB : in std_logic_vector(7 downto 0);
80        DINC : in std_logic_vector(15 downto 0);
81        SEL1  : in std_logic_vector(6 downto 0);
82        SEL2  : in std_logic_vector(6 downto 0);
83        DOUT : out std_logic
84        );
85    end Component;
86    Component RegistroSerialParalelo is
87        generic(n : integer := 40);
88        port(
89        RST  : in std_logic;
90        CLK  : in std_logic;
91        SHF  : in std_logic;
92        BIN  : in std_logic;
93        DOUT : out std_logic_vector(n - 1 downto 0)
94        );
95    end Component;
96    Component LoadRegister is
97        generic(n : integer := 40);
98        port(
99        RST  : in std_logic;
100       CLK  : in std_logic;
101       SHF  : in std_logic;
102       DIN  : in std_logic_vector(n - 1 downto 0);
103       DOUT : out std_logic_vector(n - 1 downto 0)
104       );
105   end Component;
106   Component SelDir is
107       port(
108       SEL  : in std_logic;
109       DOUT : out std_logic_vector(7 downto 0)
110       );
111   end Component;
112   Component Multiplexor2a1 is
113       generic(n : integer := 16);
114       port(
115       DIN1 : in std_logic_vector(n - 1 downto 0);  --Valores de setpoint
116       DIN2 : in std_logic_vector(n - 1 downto 0);  --Valores de senoidal
117       SEL  : in std_logic;
118       DOUT : out std_logic_vector(n - 1 downto 0)
119       );
120   end Component;
121   signal TIC, STT, FFE, TAC, CLS, PCR, NTS, NTI, SET, RPC, RTP, NCL, RNE, SNN, SYN, FE,
      SSL, NOW, RLE, RE : std_logic;
122   signal SEL1, SEL2 : std_logic_vector(6 downto 0);
123   signal ADDR : std_logic_vector(6 downto 0) := "1101001";
124   signal REGI : std_logic_vector(7 downto 0);-- := "00111101";
125   signal DTO, DTE : std_logic_vector(16 downto 0);
126   signal DINC : std_logic_vector(15 downto 0);
127   begin
128       SYN <= NOT SNN;
129       FE <= TAC AND SYN;
130       RE <= TAC AND SNN;
131       SCL <= SYN;
132       DTA <= DTE(9) & DTE(10) & DTE(11) & DTE(12) & DTE(13) & DTE(14) & DTE(15) & DTE(16)
      & DTE(0) & DTE(1) & DTE(2) &DTE(3) & DTE(4) & DTE(5) & DTE(6) & DTE(7);
133       U01 : MPUFSM port map(RST, CLK, TIC, STT, FFE, TAC, CLS, PCR, NTS, NTI, SET, RPC);
134       U02 : SRLatch port map(RST, CLK, STT, CLS, RTP);
135       U03 : Timer generic map(500) port map(RTP, CLK, TIC);
136       U04 : ProgramCounter generic map(109) port map(RTP, CLK, TIC, CLS);
```

```
137       U05 : SRLatch port map(RST, CLK, NTS, NCL, RNE);
138       U06 : Timer generic map(500) port map(RNE, CLK, NTI);
139       U07 : ProgramCounter generic map(65) port map(RNE, CLK, NTI, NCL);
140       U08 : Timer generic map(500) port map(SET, CLK, TAC);
141       U09 : Toggle port map(RST, CLK, TAC, SNN);
142       U10 : ProgramCounter generic map(9) port map(RPC, CLK, FE, FFE);
143       U11 : ProgramCounter generic map(86) port map(RTP, CLK, TIC, PCR);
144       U12 : FreeCounter generic map(6) port map(RTP, CLK, TIC, SEL2);
145       U13 : FreeCounter generic map(6) port map(RNE, CLK, NTI, SEL1);
146       U14 : MPUMUX port map(ADDR, REGI, DINC, SEL1, SEL2, SDA);
147       U15 : ProgramCounter generic map(67) port map(RTP, CLK, TIC, SSL);
148       U16 : SRLatch port map(RST, CLK, SSL, NOW, RLE);
149       U17 : ProgramCounter generic map(18) port map(RLE, CLK, RE, NOW);
150       U18 : RegistroSerialParalelo generic map(17) port map(RLE, CLK, RE, SDA, DTO);
151       U19 : LoadRegister generic map(17) port map(RST, CLK, NOW, DTO, DTE);
152       U20 : Timer generic map(2500000) port map(RST, CLK, STT);
153       U21 : SelDir port map(SWI, REGI);
154       U22 : Multiplexor2a1 generic map (16) port
          map("0110101100000000","0001110011110000", SW2, DINC);
155   end TOP;
```

```vhdl
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

entity TopControl is
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    TS   : in std_logic;
    XIN  : in std_logic_vector(31 downto 0);
    XOUT : out std_logic_vector(47 downto 0)
    );
end TopControl;

architecture TOP of TopControl is
Component LoadRegister is
    generic(n : integer := 16);
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    SHF  : in std_logic;
    DIN  : in std_logic_vector(n - 1 downto 0);
    DOUT : out std_logic_vector(n - 1 downto 0)
    );
end Component;
Component SRLatch is
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    SET  : in std_logic;
    CLR  : in std_logic;
    QOUT : out std_logic
    );
end Component;
Component ProgramCounter is
    generic ( n : integer := 10);
    port(
    RST : in std_logic;
    CLK : in std_logic;
    ENA : in std_logic;
    RDY : out std_logic
    );
end Component;
Component FreeCounter is
    generic ( n : integer := 5);
    port(
    RST  : in std_logic;
    CLK  : in std_logic;
    ENA  : in std_logic;
    COUT : out std_logic_vector(n downto 0)
    );
end Component;
Component Multiplier is
    generic(n : integer := 16; m : integer := 8);
    port(
    DIN1 : in std_logic_vector(n - 1 downto 0);
    DIN2 : in std_logic_vector(m - 1 downto 0);
    DOUT : out std_logic_vector((n + m) - 1 downto 0)
    );
end Component;
Component Adder is
    generic(n : integer := 16);
    port(
    DIN1 : in std_logic_vector(n - 1 downto 0);
    DIN2 : in std_logic_vector(n - 1 downto 0);
    DOUT : out std_logic_vector(n - 1 downto 0)
    );
end Component;
Component Mux3To1 is
```

```
70      generic(n : integer := 16);
71      port(
72      DIN1 : in std_logic_vector(n - 1 downto 0);
73      DIN2 : in std_logic_vector(n - 1 downto 0);
74      DIN3 : in std_logic_vector(n - 1 downto 0);
75      SEL  : in std_logic_vector(1 downto 0);
76      DOUT : out std_logic_vector(n - 1 downto 0)
77      );
78  end Component;
79  signal DTO1, DTO2, DTO3, DTO : std_logic_vector(31 downto 0);
80  signal DMU, DSU, DRE : std_logic_vector(47 downto 0);
81  signal SEL : std_logic_vector(1 downto 0);
82  signal QTO : std_logic_vector(15 downto 0);
83  signal Q1 : std_logic_vector(15 downto 0) := "0000000001000000";
84  signal Q2 : std_logic_vector(15 downto 0) := "1111111110000000";
85  signal Q3 : std_logic_vector(15 downto 0) := "0000000001000000";
86  signal RTE, CLR: std_logic;
87  begin
88      U01 : LoadRegister generic map(32) port map(RST, CLK, TS, XIN, DTO1);
89      U02 : LoadRegister generic map(32) port map(RST, CLK, TS, DTO1, DTO2);
90      U03 : LoadRegister generic map(32) port map(RST, CLK, TS, DTO2, DTO3);
91      U04 : Mux3To1 generic map(32) port map(DTO1, DTO2, DTO3, SEL, DTO);
92      U05 : Mux3To1 generic map(16) port map(Q1, Q2, Q3, SEL, QTO);
93      U06 : Multiplier generic map(32,16) port map(DTO, QTO, DMU);
94      U07 : Adder generic map(48) port map(DMU, DRE, DSU);
95      U08 : LoadRegister generic map(48) port map(RST, CLK, RTE, DSU, DRE);
96      U09 : SRLatch port map(RST, CLK, TS, CLR, RTE);
97      U10 : ProgramCounter generic map(5) port map(RTE, CLK, '1', CLR);
98      U11 : LoadRegister generic map(48) port map(RST, CLK, CLR, DRE, XOUT);
99      U12 : FreeCounter generic map(1) port map(RTE, CLK, '1', SEL);
100 end TOP;
```