UNIVERSIDAD DE MÁLAGA

ESCUELA DE INGENIERÍAS INDUSTRIALES

ESCUELA DE INGENIERÍAS INDUSTRIALES

Departamento: Ingeniería Electrónica

Área de Conocimiento: Electrónica

PROYECTO FIN DE GRADO

DISEÑO DE UN PLC CON LÓGICA PROGRAMABLE PARA CONTROL INDUSTRIAL

PLC DESIGN WITH PROGRAMMABLE LOGIC FOR INDUSTRIAL CONTROL

Autor: Marcos Guerrero Luque

Tutor: Jorge Romero Sánchez

Titulación: Grado en Ingeniería en Tecnologías Industriales

Málaga, 3 de Septiembre de 2023

Agradecimiento

Me gustaría expresar mi agradecimiento a todas las personas que me han apoyado durante mi crecimiento personal y académico.

En primer lugar, quiero mostrar mi gratitud a mi tutor, Jorge Romero, por su valiosa guía a lo largo de mi carrera. No solo durante el Proyecto de Fin de Grado, sino por enseñarme e ilustrarme el campo de la electrónica.

Principalmente, deseo expresar mi mayor agradecimiento a mi abuelo Leopoldo Guerrero, Perito Industrial. Que ha sido un referente en mi vida, guiándome y compartiendo su conocimiento en el campo de la ingeniería. Su inspiración siempre ha sido un factor determinante en mi desarrollo profesional y personal.

RESUMEN

El Proyecto de Fin de grado se centra en el diseño de un *PLC* con lógica programable para el control industrial. Si bien en esta área los autómatas son la opción predominante, existen alternativas de sistemas embebidos como la *FPGA*, que son altamente versátiles y ofrecen la posibilidad de implementar múltiples automatismos siempre que los recursos físicos lo permitan.

El objetivo es desarrollar un bloque mediante el entorno *Vivado*, capaz de ejecutar automatismos a partir de tablas de verdad. Dicho bloque, podrá ser reutilizado para diversos proyectos, siendo posible implementar varios en el mismo.

La validación real del proyecto es mediante la implementación de un sistema real, compuesto por una mesa giratoria. Su montaje pone a prueba la capacidad de diseño electrónico y la viabilidad de un hardware reprogramable como la *FPGA*.

Las secciones adicionales del trabajo como el Estado del arte y tecnologías ayudan a poner en situación al lector sobre el proyecto.

En última instancia, el proyecto busca contribuir a la automatización industrial, enfrentando retos y desafíos de la ingeniería industrial. El trabajo abre puertas tanto para sistemas industriales como una herramienta didáctica a futuros estudios.

Palabras Clave: FPGA, Vivado,PLC, Automatización Industrial.

ABSTRACT

The Final Degree Project focuses on the design of a *PLC* with programmable logic for industrial control. Although *PLCs* are the predominant option in this area, there are alternatives of embedded systems such as *FPGA*, which are highly versatile and offer the possibility of implementing multiple automatisms. Highly versatile and offer the possibility of implementing multiple automatisms as long as the physical resources allow it.

The objective is to develop a block using the Vivado environment, capable of executing automatism from truth tables. This block can be reused for different projects, being possible to implement several of them in the same one.

The actual validation of the project is through the implementation of a real system, consisting of a rotary table. Its assembly tests the electronic design capacity and the feasibility of reprogrammable hardware such as the *FPGA*.

Additional sections of the paper such as the State of the Art and Technologies help to situate the reader on the project.

Ultimately, the project seeks to contribute to industrial automation, facing challenges and challenges of industrial engineering. The work opens doors to both industrial situations and as a didactic tool for future studies.

Keywords: FPGA,Vivado,PLC, Industrial Automation.

Índice

Índice de figuras

Índice de tablas

1. Introducción

El control industrial es un área de la ingeniería que se encarga de diseñar y desarrollar sistemas y procesos automatizados para la producción de bienes y servicios. En este sentido, el ***PLC*** (Programmable Logic Controller) es un dispositivo electrónico utilizado en la automatización industrial, que se encarga de controlar y monitorear el funcionamiento de los sistemas y procesos productivos. El *PLC* se ha convertido en una herramienta indispensable en la industria moderna, ya que permite optimizar la producción, mejorar la calidad de los productos, reducir costos y mejorar la seguridad en los procesos.

La lógica programable, por su parte, es una tecnología que se utiliza para diseñar circuitos electrónicos digitales mediante la programación de dispositivos como las ***FPGA*** (Field Programmable Gate Arrays). Estas tecnologías ofrecen una gran flexibilidad y permiten diseñar sistemas con alta capacidad de procesamiento y control en tiempo real.

La combinación de la **lógica programable** y el *PLC* puede proporcionar una solución eficiente y flexible para el control industrial. Sin embargo, existen dificultades en la integración de estas tecnologías, como la complejidad del diseño y la programación, la compatibilidad de los dispositivos y la fiabilidad del sistema.

Por lo tanto, un Proyecto de fin de grado enfocado en el diseño de un *PLC* con lógica programable para **control industrial** puede ser una oportunidad para abordar estos retos y contribuir al avance de la automatización industrial. El objetivo de este trabajo sería diseñar un sistema que combine las capacidades de la lógica programable y el *PLC* para lograr un control eficiente y flexible de los sistemas. Esto incluye la programación de la FPGA, el diseño de circuitos electrónicos, la integración de los dispositivos y la validación del sistema.

En resumen, el diseño de un *PLC* mediante lógica programable para control industrial es un tema relevante y actual en la ingeniería industrial, que puede proporcionar soluciones eficientes y flexibles para la automatización de los procesos productivos. Un Proyecto enfocado en este tema puede contribuir a la industria, abrir oportunidades de investigación y desarrollo en este campo.

1.1. Motivación

Durante mi trayectoria académica en el Grado en Tecnologías Industriales, he adquirido un amplio conocimiento y habilidades en ingeniería, con un enfoque específico en la electrónica. Dentro de este campo, he adquirido experiencia en el diseño de automatismos, programación en lenguajes de bajo y alto nivel, así como en el montaje e implementación de hardware, incluyendo el uso de dispositivos como *PLCs*, *FPGAs* y microcontroladores.

En consecuencia, considero que la programación de una *FPGA* para que se comporte como un *PLC* puede ser una solución interesante para el control y manejo industrial, debido a que las *FPGAs* son altamente personalizables, tienen una capacidad de procesamiento muy alta, pueden procesar señales analógicas y digitales, y son dispositivos de bajo costo. Por lo tanto, el desarrollo de un proyecto de este tipo podría tener aplicaciones prácticas en el control de procesos industriales, lo que podría tener un impacto significativo en la eficiencia y la productividad.

1.2. Objetivos

El principal objetivo del Proyecto de fin de grado, se basa, en diseñar una *FPGA* (Lógica programable) para generar un autómata de estados finitos y de esta forma se pueda comportar como un *PLC*, así podremos ante cualquier diagrama de estados realizar un manejo y control de mesas industriales.

1.3. Estructura del trabajo

La estructura del trabajo se plantea como 4 Fases fundamentales para diseñar un *PLC* mediante Lógica Programable, con un uso de control industrial.

Se dividen en: un estudio teórico, la programación en el entorno *Vivado* del proyecto, La implementación y montaje al sistema real y Testeo y validación del sistema.

Para el correcto desarrollo del Proyecto se llevará a cabo un estudio teórico sobre los autómatas industriales, ya que es el sistema de control a “imitar” y la *FPGA* que implica, a su vez, el aprendizaje de nuevos lenguajes de programación como *VHDL*.

Además, se requiere un uso avanzado del *software Vivado*, que incluye la programación en *VHDL*, el diseño de bloques y el uso de cronogramas. Para llevar a cabo esta tarea, se necesitan buenos conocimientos de electrónica tanto para el diseño de *software* como para *hardware* y control de automatismos, así como el diseño de los Diagramas de Estados.

Luego, para la resolución del proyecto se necesita la adquisición de habilidades técnicas y conocimientos específicos para poder desarrollar con éxito una máquina de estados finitos en una *FPGA*, que pueda comportarse como un *PLC*. Se trata de

una tarea compleja y requiere una sólida formación en electrónica y programación, así como una buena comprensión de los sistemas empotrados y la automatización.

Una vez que se ha comprendido los diferentes fundamentos teóricos mencionados, será necesario el diseño del *software* a implementar. Para llevar a cabo esta tarea, se deberá programar el diseño de la *FPGA* en *Vivado* con el fin de desarrollar el correcto funcionamiento del *PLC*. De esta forma se reprogramará la *FPGA* para que se pueda implementar cualquier autómata de estados finitos.

El programa se utilizará para el control de una mesa industrial, Esto implica el manejo de los *pmod* que son dispositivos utilizados para la lectura de matrices de entrada al sistema, tales como sensores, pulsadores o láseres, además será necesario el uso de botones,*switches*, etc. Pertenecientes a la propia *FPGA*.

Tras el diseño del programa, se realiza el montaje y pruebas en una Planta Real(Mesa Giratoria): Una vez configurado y diseñado el programa que se subirá a la *FPGA*, habrá que realizar el montaje del Sistema de control(conexionado de los sensores, pulsadores, motor, actuadores con la *FPGA* y mesa Industrial...).

Finalmente, se deberá realizar un conjunto de pruebas y simulaciones para poder validar el sistema industrial, corrigiendo los posibles fallos que surjan durante el mismo.

2. Estado del Arte

2.1. Autómatas Industriales

En la actualidad se ha ido imponiendo en todo ámbito Industrial, la utilización de autómatas Programables(PLC), para controlar a tiempo real procesos industriales. El ***PLC*** supone grandes ventajas frente a otros sistemas como el uso de lógica cableada(relés, electroneumática...). Ya que puede ser reprogramado, dando cabida a nuevos diagramas de control, y puede ser conectado directamente a sensores y relés. Por lo descrito, evitas el uso de grandes aparamentas con el uso de lógica cableada, minimizando gastos y permitiendo diseñar control de complejos industriales.

Su utilización es especialmente recomendable en instalaciones donde es necesario realizar procesos de maniobra, control, señalización, etc. por tanto, su aplicación abarca desde procesos de fabricación industrial, de cualquier tipo de transformaciones industriales, control de instalaciones [1].

Los **automatismos** son sistemas que posibilitan que las máquinas y equipos operen de manera autónoma, sin la intervención directa del ser humano.
Un automatismo bien diseñado:

- Reducen significativamente la carga de trabajo del operador, liberándolo de la obligación de permanecer constantemente frente a la máquina y permitiéndole enfocarse en tareas más importantes y valiosas.
- Elimina las tareas complejas, peligrosas, pesadas o indeseadas, siendo ejecutadas por la máquina.
- Facilitan cambios en los procesos de producción, permitiendo la transición de una cantidad o tipo de producción a otra de manera más sencilla y eficiente.
- mejoran la calidad de los productos al permitir que la máquina supervise los criterios de fabricación y tolerancias necesarias, asegurando que se cumplan de manera constante y precisa a lo largo del tiempo.
- incrementa la producción y la productividad.
- permite economizar material y energía.
- aumenta la seguridad del personal y controla y protege las instalaciones y las máquinas.

Un sistema a automatizar se puede descomponer fundamentalmente en 2 partes, una parte de control encargada de elaborar órdenes necesarias para la ejecución y una parte operativa que efectúa dichas operaciones[2].

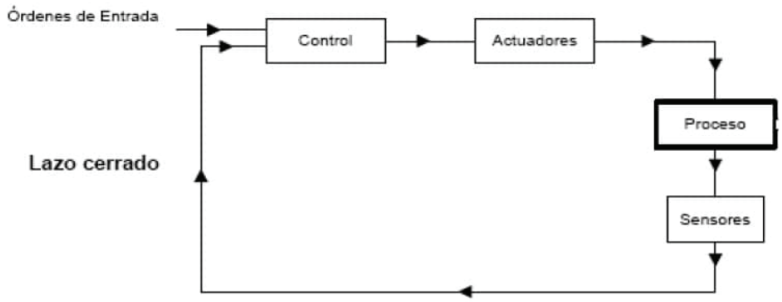


Figura 1: Esquema de automatismo

Los *PLCs* disponen de distintos sistemas de programación, estos lenguajes nos sirve como canal de comunicación entre el sistema operativo que interpreta el lenguaje, y el usuario que tiene al manejo del programa. La finalidad es diseñar instrucciones secuenciales que el procesador del autómata se encargará de traducir en señales digitales que controlan los procesos y máquinas asignadas.

Los Tipos de lenguaje se pueden fundamentar respecto a su jerarquía en 2, los lenguajes de nivel **bajo** o **alto**, aunque también se puede dividir respecto a lenguajes **visuales** y **escritos**.

Lenguajes de Bajo Nivel

Lista de Instrucciones (*STL*) es un lenguaje utilizado para pequeñas aplicaciones que tiene mucha similitud con lenguaje ensamblador.

Texto estructurado (*ST*) es un lenguaje escrito que tiene una sintaxis parecida a Pascal, este lenguaje abarca muchas posibilidades, siendo el más usado a bajo nivel para programar *PLC*, en programas tales como *TwinCat*.

Lenguaje de Alto Nivel

Diagrama escalera (*LD*) es mediante interfaz gráfica donde su estructura se asemeja a una escalera donde corren 2 relés verticales, siendo uno el flujo de entrada y otro su salida.

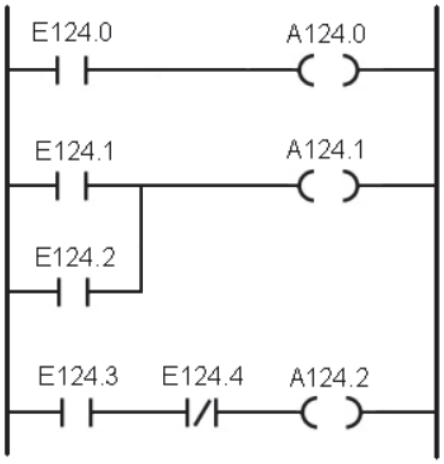


Figura 2: Diagrama Escalera

Diagrama de Funciones secuenciales (*SFC*) es una representación de las secuencias de control en un programa, el proceso fluye a lo largo del diagrama con forma va cumpliendo las condiciones establecidas, dicho lenguaje proviene del estándar *Grafcet* siendo uno de los más conocidos y aplicados en los Autómatas.

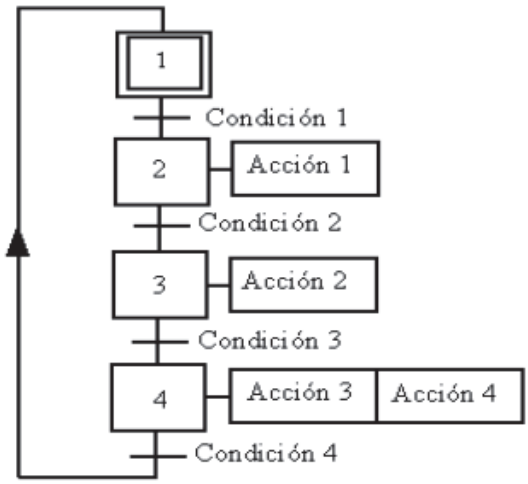


Figura 3: Diagrama de Bloques en *SFC*

Existen más lenguajes para programar los *PLC*, aunque la mayoría están en desuso, siendo los mencionados los más relevantes [3].

2.2. *FPGA*

Existen otros tipos de Lógica programada a estudiar, tales como la ***FPGA*** siendo una matriz de puertas lógicas reprogramables. También existen otros casos como los **microcontroladores** que se fundamentan en conjuntos de puertas lógicas destinadas a aplicaciones concretas.

Las *FPGA* pertenecen a la familia de Lógicas programada, Una *FPGA* se puede definir como una matriz de bloques lógicos configurables(de manera combinatoria y/o secuencial), que está unida por una red de interconexión totalmente reprogramable. Luego nos permite rediseñar el Hardware, algo que no nos permite otros sistemas de lógica programada como los autómatas o microcontroladores.

Las más recientes se producen utilizando un proceso de cobre de 65 nm, donde su densidad puede alcanzar más de 10 millones de puertas equivalentes por chip con frecuencias de más de 500 MHz. Los dos principales fabricantes de *FPGA* son *Altera* y *Xilinx*, este último proporciona software muy relevante para la configuración, nombrado ***Xilinx Vivado*** [4].

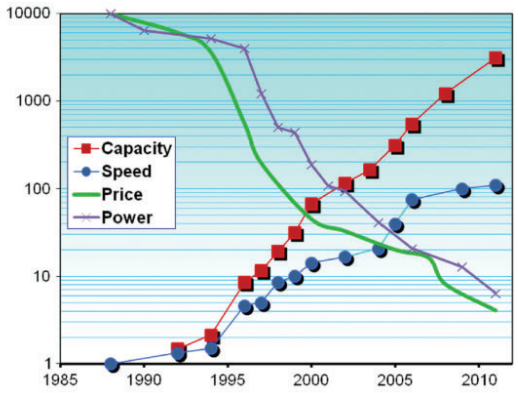


Figura 4: Evolución de *FPGA*

La Figura 4 representa la evolución de la *FPGA*, con un aumento continuado de la capacidad y velocidad del mismo, ligado a una reducción continua del precio y potencia necesaria. Dicha evolución del sector ha sido impulsada por la tecnología de proceso y la Ley de *Moore*, los cuales han originado cambios cualitativos en la *FPGA*.

Una *FPGA* es un chip que contiene componentes lógicos programables e interconexiones programables entre ellos. Estos componentes pueden ser programados como puertas lógicas(AND,OR,XOR,etc) y a su vez se pueden implementar en combinaciones más complejas como decodificadores. Además, también incluye elementos de memoria, como el uso de *flips-flops*.

En la arquitectura de una *FPGA* se utilizan bloques lógicos configurables (*CLBs*) para realizar funciones lógicas y se emplean interconexiones programables para construir los circuitos. El *bitstream* es el encargado de activar las interconexiones y establecer el estado de los cables. Los bloques lógicos pueden ser interconectados mediante una jerarquía de interconexiones programables, lo que permite una mayor flexibilidad en el diseño.

La programación de una *FPGA* se realiza mediante un software que utiliza un lenguaje de descripción de hardware, y después se sintetiza, emplaza y enruta el diseño para finalmente cargar un *bitstream* en la *FPGA*.

El ***bitstream*** no es más que un tramo de datos que se transporta por un bus serial al chip de la FPGA. Para generar el*bitstream* se pueden utilizar distintos entornos de trabajo tales como *Xilinx Vivado*, siendo un software que nos permite reprogramar y configurar el hardware de la FPGA [5].

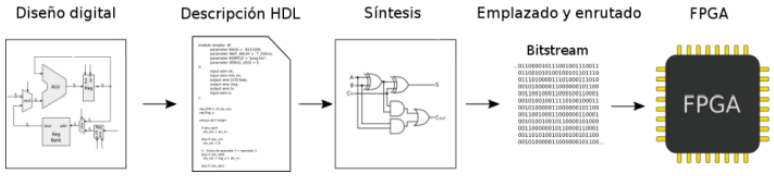


Figura 5: Generación del *Bitstream*

La Figura 5 muestra las etapas necesarias para programar la *FPGA*.

- Tras plantear el diseño a realizar será necesario utilizar un lenguaje de descripción de hardware, siendo los más populares *Verilog* y *VHDL*.
- La herramienta de síntesis (*vivado*) deberá interpretar dicho código y sintetizar el proceso que describirá el esquema con las puertas lógicas necesarias para desarrollar el diseño digital propuesto.
- Con la síntesis realizada, deberá realizar el emplazado y enrutado para adaptar toda la información al formato *bitstream* que será enviado a la placa.

  [5].

2.3. Lenguajes de Programación

Las *FPGAs* no se basan en un procesamiento secuencial de la programación, esto es debido a que se basa en hardware programable, y, por tanto, será necesario utilizar un lenguaje que trabaje de distinta forma.

Un lenguaje de descripción de hardware (*HDL*) es una herramienta especializada en la descripción de estructuras, diseño y comportamiento de hardware. Se pueden utilizar para representar diagramas lógicos y circuitos de diferentes niveles de

complejidad, desde expresiones booleanas hasta sistemas más complejos. Estos lenguajes son útiles para representar sistemas digitales de manera legible tanto para las máquinas como para las personas.

ABEL es un lenguaje *HDL* que fue inventado para permitir a diseñadores especificar funciones lógicas. Un programa *ABEL* es un archivo de texto que contiene varios elementos:

- Documentación, incluyendo el nombre del programa y comentarios.
- Declaración de entradas y salidas de las funciones lógicas.
- Instrucciones que especifican las funciones lógicas.

  [6].

Durante la década de 1980, el Departamento de Defensa de Estados Unidos y el *IEEE* patrocinaron el desarrollo de un lenguaje de descripción de hardware de gran capacidad denominado *VHDL* con las siguientes características:

- Diseños con la posibilidad de descomponerse jerárquicamente.
- Cada elemento de diseño con una interfaz bien definida con la opción de conectarla a otros elementos.
- Las especificaciones de comportamiento se pueden utilizar ya sea para un algoritmo o una estructura *Hardware*.
- La concurrencia, señales de reloj pueden ser modeladas, y se pueden manejar circuitos secuenciales tanto asíncronos como síncronos.
- Las operaciones lógicas y comportamiento pueden simularse mediante cronogramas.

  [6].

En la actualidad, se usan principalmente 2 tipos de lenguajes *HDL*, en la *FPGA* los más presentes son ***Verilog*** y ***VHDL***. *VHDL* significa “Very High Speed Integrated Circuit”, es un estándar de dominio público y, por tanto, no depende de ningún fabricante o dispositivo. Se basa en un diseño jerárquico, en el que se mantiene un orden. su procesamiento es procedural, donde Los procedimientos simplemente contienen una serie de pasos a realizar y Cualquier procedimiento puede ser llamado en cualquier momento durante la ejecución de un programa. Este diseño de lenguaje se asemeja a *Java*.

Por otro lado, *Verilog* es también de dominio público y similar al *VHDL*, pero se diseñó basándose en el lenguaje de programación ***C***, con el propósito de ser familiar para los diseñadores, ya que *C* es uno de los lenguajes mundialmente más conocido y usado.

Los Lenguajes ***HDL*** tienen una gran importancia sobre el concepto del tiempo, ya que se basan en descripción de *hardware* [5].

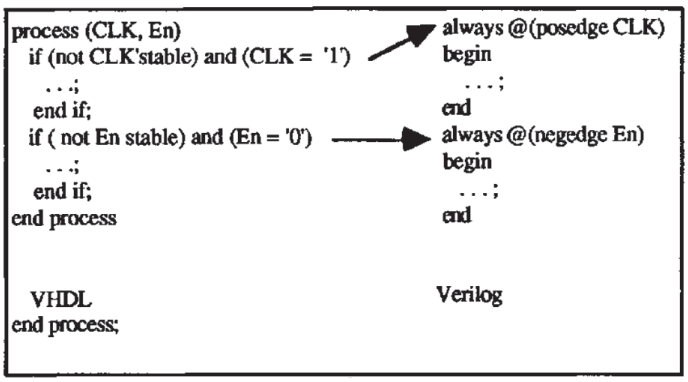


Figura 6: Ejemplo de código en *VHDL* frente a *Verilog*

2.4. Microcontroladores

Un **microcontrolador** es un circuito integrado digital que incorpora todos los elementos de un procesador digital síncrono secuencial programable de arquitectura Harvard o *Princeton* (Von Neumann). También se conoce como un computador integrado o embebido, y está diseñado específicamente para tareas de control y comunicaciones. Debido a su tamaño compacto, los microcontroladores permiten integrar un procesador programable en muchos productos industriales, y su bajo costo, consumo de energía y velocidad adaptable los hacen apropiados para una amplia variedad de aplicaciones. Los microcontroladores son necesarios para la realización de sistemas eléctricos empotrados en otros sistemas.

Los microcontroladores se utilizan para múltiples sectores como computación, domótica, comunicaciones, industria, automóvil, etc.

Se pueden categorizar en función de su gama y bits a los que trabaja siendo:

- Gama baja: 4,6,8 y 16 bits. Sirven únicamente para tareas sencillas de control.
- Gama media: 16 y 32 bits. Tareas con mayor grado de complejidad como para control de automóvil.
- Gama alta: 32,64 y 128 bits. Destinado a procesamientos complejos como ordenadores.

Nº	***FPGA***	**Microcontrolador**
1	La *FPGA* trabaja en paralelo, pudiendo realizar varias tareas a la vez	Solo puede realizar una acción a la vez, al trabajar en serie
2	Son peores para la comunicación en serie	Son buenos para las comunicaciones en serie
3	La *FPGA* incorpora procesador *ARM*, pudiendo, por tanto, trabajar como microcontrolador	No es posible hacer que trabaje como una *FPGA*
4	Es un circuito electrónico con hardware reprogramable, configurando las interconexiones de puertas lógicas	Únicamente ejecuta las instrucciones del código, no siendo posible rediseñar el hardware
5	Consume mayor cantidad de energía	Menor consumo de energía
6	Requiere mayor estudio, al ser mas complejo	La curva de aprendizaje es sencilla
7	El uso de punto flotante presenta grandes Complicaciones	Se puede utilizar puntos fijos y flotantes
8	El cambio de código requiere bastante tiempo,debido a que tiene mas etapas para generar el *bitstream*	Fácil cambio de código
9	Precio de adquisición elevada	Fácil adquisición

Tabla 1: Comparación entre *FPGA* y Microcontrolador

En resumen de la Tabla 1 dependiendo del tipo de proyecto que se quiera realizar, será mejor una opción u otra, dando la *FPGA* la posibilidad de tanto diseñar el circuito electrónico como el uso de sistemas empotrados [5].

3. Tecnologías

En la presente sección, se proporciona una descripción detallada de las herramientas, metodologías y tecnologías utilizadas en el proyecto, con el objetivo de brindar al lector una comprensión completa de los recursos necesarios para llevar a cabo el trabajo.

3.1. *FPGA*

EL elemento principal del proyecto es la *FPGA*, siendo el modelo ***Zybo*** *Zynq*^{TM} ***-7000***, mostrado en la Figura 7. Con el fin de justificar su elección, se proporcionará una descripción detallada de la arquitectura del dispositivo, explicando sus especificaciones y arquitectura que satisfacen los requisitos y objetivos del proyecto.

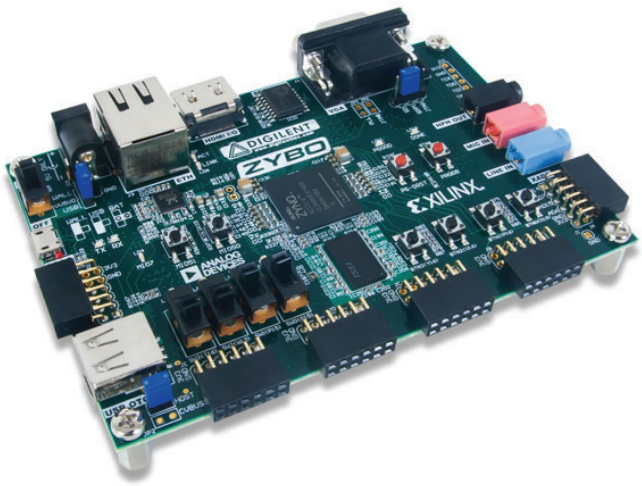


Figura 7: Placa *Zybo Zynq*^{TM}*-7000*

Las especificaciones técnicas principales de la *Zybo* son:

- Procesador *dual-core Cortex-A9* de 650 MHz
- Controlador de memoria *DDR3* con 8 canales *DMA*
- Controladores de periféricos de alta velocidad: Ethernet de 1G, USB 2.0, *SDIO*
- Lógica reprogramable equivalente a *FPGA Artix-7*
- *Zynq-7000 AP Soc XC7Z010-1CLG400C*
- 512MB x32 *DDR3* con un ancho de banda de 1050Mbps

- Puerto *HDMI* de doble función (fuente/receptor)
- Puerto de origen *VGA* de 16 bits por píxel
- *PHY Ethernet* (1Gbit/100Mbit/10Mbit)
- *PHY USB 2.0 OTG* (compatible con host y dispositivo)
- *EEPROM* externa (programada con un identificador compatible con EUI-48/64™ de 48 bits único en todo el mundo)
- Flash serie de 128Mb con interfaz *QSPI*
- *GPIO*: 6 botones pulsadores, 4 interruptores deslizantes, 5 LED
- Seis puertos *Pmod* (1 dedicado al procesador, 1 analógico/digital dual, 3 diferenciales de alta velocidad, 1 dedicado a lógica)

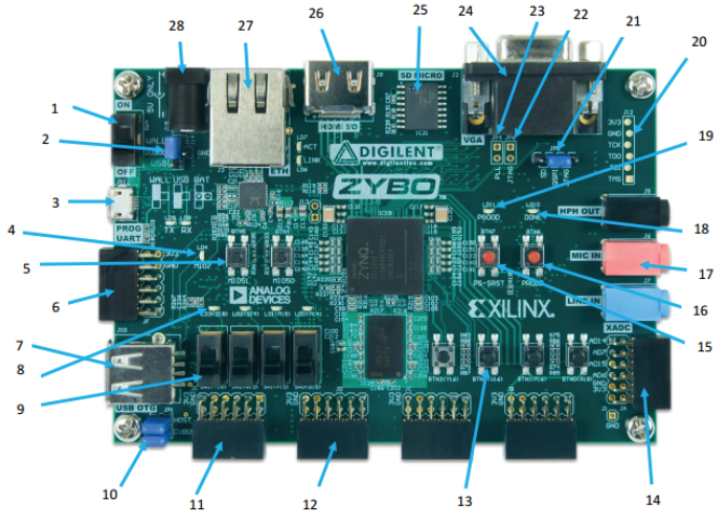


Figura 8: Diagrama *Zybo Zynq™-7000*

Nº	Componentes	Nº	Componentes
1	Botón de Encendido	**15**	Pulsador de reinicio del procesador
2	Seleccionador de alimentación y bateria	**16**	Pulsador de reinicio para la configuración Lógica
3	Puerto USB compartido UART/JTAG	**17**	Conectores del codec de audio
4	LED MIO	**18**	LED de finalización de configuración Lógica
5	Pulsadores MIO (2)	**19**	LED de correcta alimentación a la Placa
6	Pmod MIO	**20**	Puerto JTAG para cable externo opcional
7	Conectores USB OTG	**21**	Puente para el modo de programación
8	LEDs Lógicos (4)	**22**	Puente de habilitación de modo JTAG independiente
9	switches (4)	**23**	Puente de PLL Bypass
10	Puente de selección de host/dispositivo USB OTG	**24**	Conector VGA
11	Pmod estándar	**25**	Conector microSD (lado inverso)
12	Pmods de alta velocidad (3)	**26**	Conector de HDMI
13	Botones (4)	**27**	Conector Ethernet RJ45
14	Pmod XADC	**28**	Conector de alimentación

Tabla 2: Elementos de la Zybo *Zynq™-7000*

En la Figura 8 y su respectiva tabla 2 se ilustra los distintos elementos de la Placa *Zybo Zynq^{TM}-7000*, junto con una descripción de cada uno de ellos.

El ***Zynq AP SoC*** se divide en dos subsistemas distintos: el sistema de procesamiento (PS) y la lógica programable (PL). La Figura 9 muestra una descripción general de la arquitectura *Zynq AP SoC*, con el *PS* en verde claro y el *PL* en amarillo.

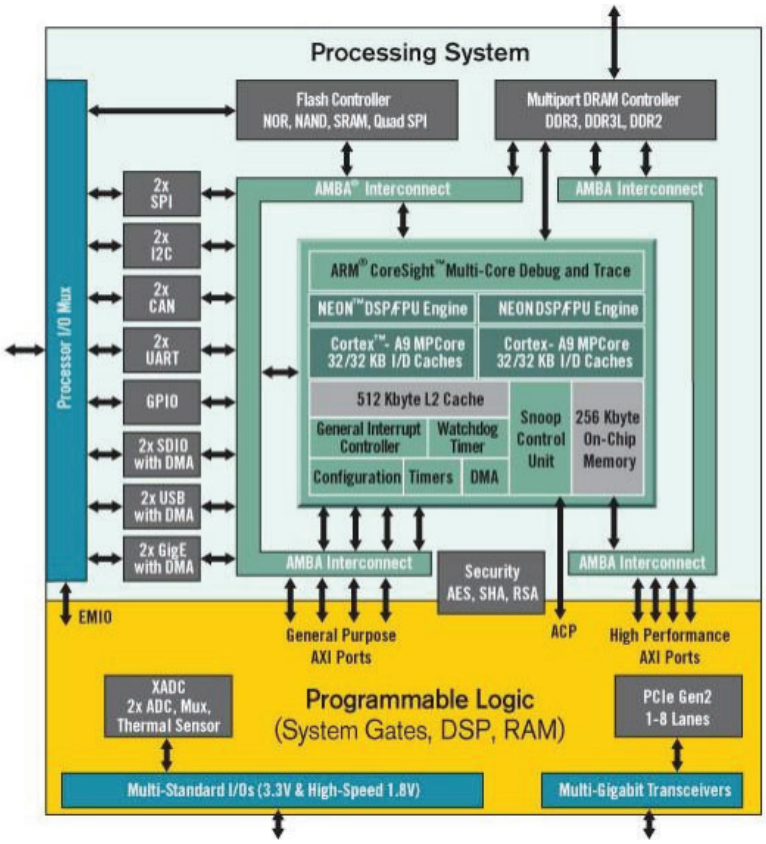


Figura 9: Arquitectura *Zynq AP SoC*

El ***PL*** es muy similar a una *FPGA Artix* serie 7 de *Xilinx*, pero con puertos y buses dedicados para conectarse al *PS*. El ***PS*** consta de muchos elementos, incluyendo la unidad de procesamiento, interconexión de arquitectura de bus de microcontrolador avanzada (AMBA), controlador de memoria DDR3 y varios controladores periféricos pines. Los controladores periféricos que no tienen sus entradas y salidas conectadas a los pines *MIO* pueden enrutar sus E/S a través del *PL*, a través de la interfaz *Extended-MIO* (EMIO). Los controladores periféricos están conectados a los procesadores como esclavos a través de la interconexión *AMBA*. La lógica programable también está conectada a la interconexión como esclavo, y los diseños pueden implementar múltiples núcleos en la estructura FPGA que también contienen registros de control direccionables.

En Resumen, el Sistema de procesamiento lleva como elementos principales 2 procesadores ***ARM***, controladores de memoria externa y diferentes periféricos. Por otro lado, la lógica programable lleva puertas, módulos DSP y de memoria totalmente reconfigurables.

Toda la información sobre las especificaciones técnicas y arquitectura de la *Zybo Zynq^{TM}-7000* se puede consultar en la ficha técnica [7].

3.2. Entorno *Xilinx Vivado*

Para la implementación de cualquier diseño es necesario definir el entorno de software, se ha optado por el diseño ***Vivado Design Suite***.

El entorno de diseño *Vivado Design Suite* permite la posibilidad de seguir el flujo de diseño tradicional de *FPGA*. De esta forma, la metodología de diseño puede partir de una descripción *RTL* para llegar al *bitstream* que finalmente se implementará en la Placa.

Xilinx Vivado es un software de diseño de sistemas digitales que permite diseñar, simular y sintetizar circuitos digitales en la *FPGA*. *Vivado Suite* es una herramienta integral que permite crear diseños de alta complejidad mediante herramientas de diseño y verificación, todo ello desde la misma plataforma.

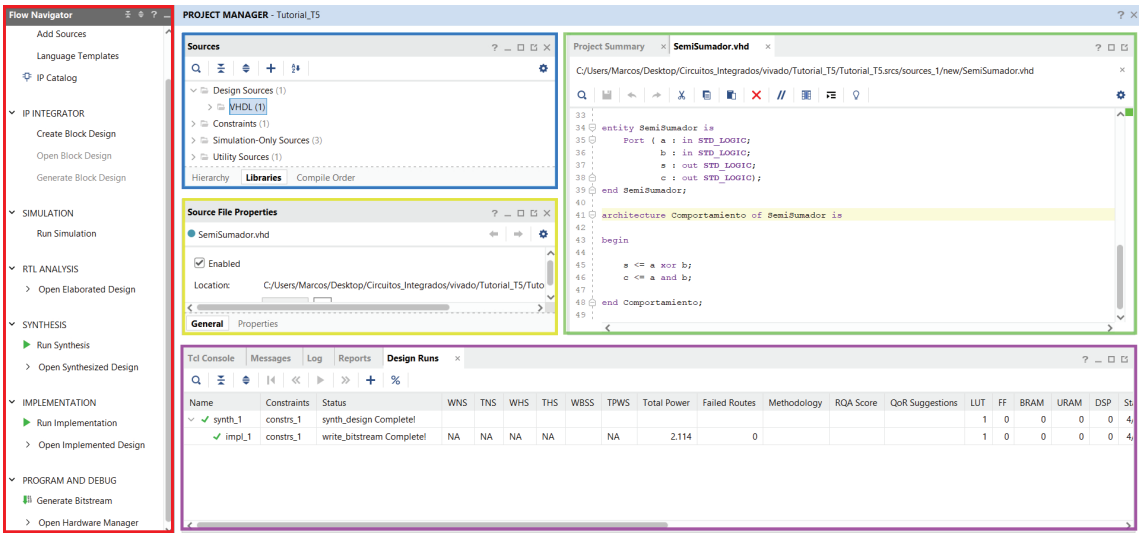


Figura 10: Interfaz de *Xilinx Vivado*

La Figura 10 muestra la Interfaz de Vivado,En la ventana azul se encuentran los *Sources*, que contiene todos los ficheros generados, que pueden ser diseños, ficheros de restricciones, tests de simulación, biblioteca, etc.

La Ventana verde muestra el fichero seleccionado, siendo en este caso el diseño Semisumador en formato de programación para *VHDL*, que será el lenguaje a desarrollar durante el proyecto.

La ventana amarilla presenta las propiedades del archivo seleccionado. La morada presenta un cuadro con los mensajes por Consola, reportes y estado en el que se encuentra cada fase de diseño.

Por último, el recuadro rojo es el flujo de navegación, donde se encuentra cada una de las fases posibles.

Fases

- ***IP Integrator***: es una herramienta que nos permite la creación y configuración de sistemas digitales mediante bloques de diseño IP predifinidos. Procesadores, interfaces controladores, y otros bloques.

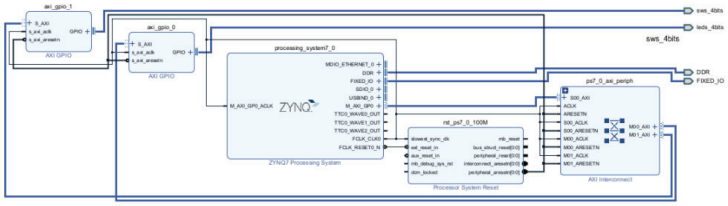


Figura 11: Ejemplo de bloque Integrador

- ***Simulation***: Fase de simulación donde se puede comprobar mediante cronogramas el diseño propuesto a partir de test de simulación.

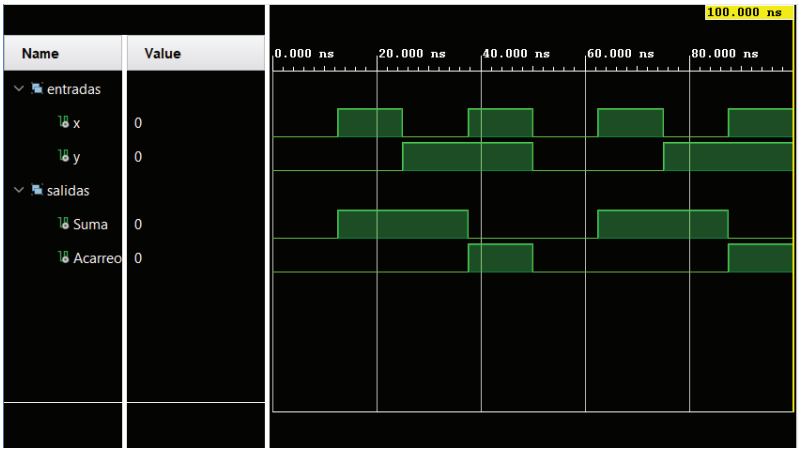


Figura 12: Cronograma de Semisumador

- ***RTL Analysis***: Genera el esquemático para el circuito electrónico que cumple diseño propuesto en VHDL

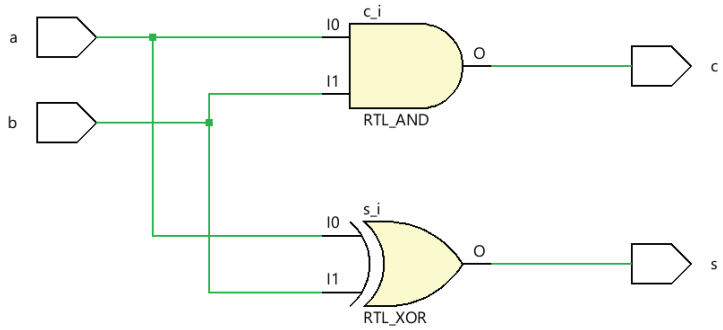


Figura 13: Esquemático de Semisumador

- ***Synthesis***:Se encarga de sintetizar el esquema *RTL* en una estructura física específica para el dispositivo.

- ***Implementation***: Fase donde se configura y personaliza los recursos a utilizar de la *FPGA*.

- ***Bitstream***: Se genera finalmente el tramo de datos que se transporta al chip, para conseguir la estructura deseada.

Estos pasos cumplen el diseño mostrado en la Figura 5.

3.3. Lenguaje *VHDL*

VHDL es un lenguaje de Descripción de Hardware para circuitos Integrados de Muy Alta Velocidad. Permite describir y modelar diseños, en una forma que la máquina pueda entender, organizando sistemas de hardware digitales. Tiene una síntesis amplia y flexible para el modelado estructural.

Existen fundamentalmente 3 tipos de secciones en *VHDL*, que son las Bibliotecas, entidades y arquitecturas [8].

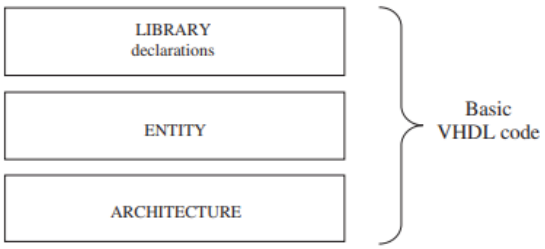


Figura 14: Secciones fundamentales de *VHDL*

Library

Consiste en la declaración de **paquetes** a añadir al diseño *VHDL*, desde la biblioteca especificada. Para incluirlos se utiliza esta palabra clave. El primer campo es el nombre de la biblioteca, el segundo campo es el nombre del paquete. El último campo es la funcionalidad específica del paquete que se incluirá [9].

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

Figura 15: *Library*

En la Figura 15 se muestra un ejemplo donde primero se declara la biblioteca estándar *IEEE*, para que se incluya en el proyecto, y se introduce los paquetes a usar.

Generalmente, se suelen incluir estos 3 paquetes al diseño:

- iee.std_logic_1164 (de la IEEE)
- standard (de la std)
- work

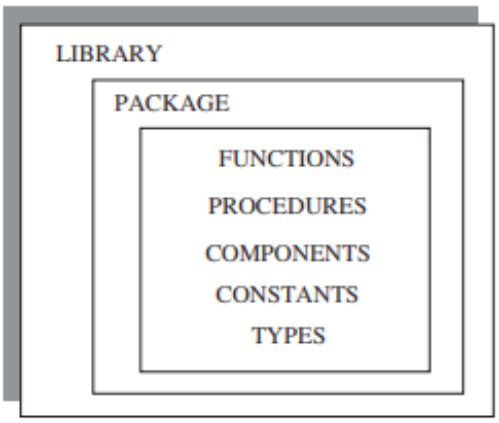


Figura 16: Bloques fundamentales de la biblioteca

La Figura 16 especifica las partes en las que se divide fundamentalmente la biblioteca.

Entity

La Entidad es una lista con especificaciones de todos los pines de entrada y salida del circuito.

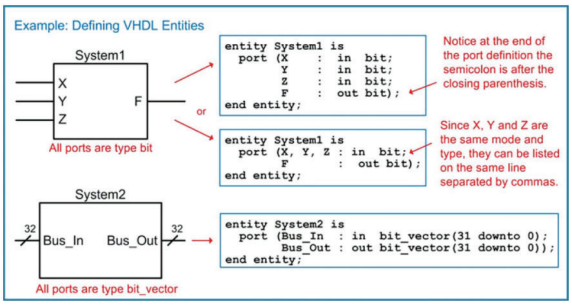


Figura 17: Ejemplo de Entidad

La Figura 18 muestra la estructura para generar una entidad, de distintas formas.

Architecture

La **Arquitectura** es el cuerpo del diseño, siendo el modo fundamental en el que VHDL describe el componente a modelar. La descripción del comportamiento se puede hacer a niveles concurrentes (entre procesos) o secuencias (dentro de cada proceso).

```
ARCHITECTURE architecture_name OF entity_name IS
    [declarations]
BEGIN
    (code)
END architecture_name;
```

Figura 18: Ejemplo de Arquitectura

Es importante recalcar que la Arquitectura trabaja de manera **Concurrente**, y, por tanto, los procesos se ejecutaran y desarrollan en la simulación simultáneamente. Pero dentro de un mismo proceso el código sí se ejecutará de manera secuencial.

3.3.1. Conceptos de *VHDL*

Es necesario para su uso entender algunos conceptos básicos del lenguaje.

Proceso: Es una construcción que se utiliza para describir el comportamiento secuencial o concurrente de un circuito digital. Los procesos pueden estar asociados a una señal de reloj y se utilizan para actualizar el estado de las señales de salida en función del estado actual de las señales de entrada.

Un proceso puede contener declaraciones de variables, señales, constantes y otros elementos de diseño, y se puede ejecutar de forma secuencial o concurrente con otros procesos. Los procesos se utilizan para modelar el comportamiento de un sistema digital y son una parte fundamental del diseño en *VHDL*.

Señales: es el modo fundamental en que *VHDL* modela una conexión eléctrica tipo línea de transmisión (wire, transmission line) y su asignación nunca es inmediata tras la ejecución de la misma.

Variables: únicamente se puede utilizar en procesos y subprogramas, se les puede asignar valores en caso de ser necesario y al no tener equivalencia física como las señales, se actualizan de forma inmediata.

Constantes: solo se puede asignar un valor inicial y se utiliza como parámetro.

Subprogramas: es una forma de agrupar código para aclarar la programación en VHDL. Las declaraciones son locales.

Parte de la información sobre el lenguaje *VHDL* utilizada ha sido influenciada de la documentación y los apuntes de la asignatura de Circuitos Integrados.

3.4. Sistema Industrial

Dentro del marco de las tecnologías que serán empleadas en el desarrollo del proyecto, se requiere una explicación detallada de los componentes que integran el sistema industrial. Estos componentes son esenciales para garantizar el funcionamiento de la **mesa giratoria** en cuestión.

Los elementos que se encuentran en la mesa giratoria son el motor inductivo, un variador de frecuencia para controlar dicho motor y 2 sensores fotocélulas con sus reflectores.

Gracias a los avances en tecnología de microprocesadores, ahora es posible controlar motores asíncronos de corriente alterna (c.a.) con la precisión necesaria para aplicaciones de servo. Estos motores son robustos y comunes en la industria manufacturera, siendo utilizados en la mayoría de las aplicaciones de accionamiento industrial. Aunque son conocidos por su uso en aplicaciones de velocidad fija, también son útiles en aplicaciones variables [10].

3.4.1. Variador de Frecuencia

Un **inversor de frecuencia**, también conocido como variador de frecuencia o drive de velocidad variable, es un dispositivo electrónico diseñado para controlar la velocidad, el par y otros parámetros de operación de motores eléctricos, principal-

mente en motores de inducción, como es el caso. Su objetivo principal es modificar la frecuencia de la corriente alterna suministrada al motor. Lo que permite controlar su velocidad de rotación.

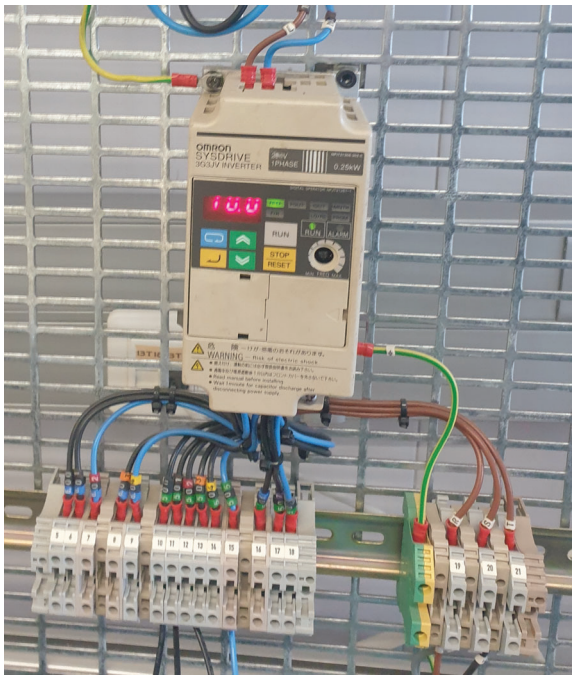


Figura 19: Inversor 3G3JV

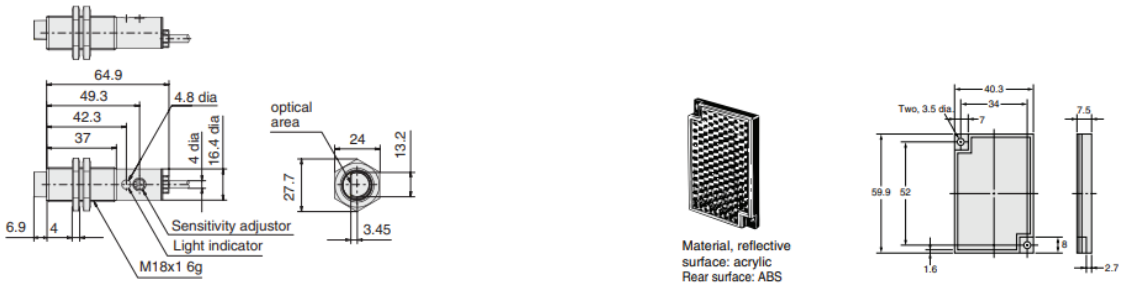
El variador de frecuencia a utilizar es el modelo ***3G3JV*** que se muestra en la Imagen 19, destaca por su alta eficiencia y un equilibrado rendimiento en relación con el costo. La sección de potencia generosa en dimensiones es una característica clave que asegura un arranque de alto torque y una reducida sensibilidad a las sobrecargas. Esto resulta en una mayor fiabilidad del sistema y una mejor protección tanto para el inversor como para el motor. También ofrece una serie de funciones avanzadas y características que aumentan su versatilidad y flexibilidad. Por ejemplo, dispone de múltiples entradas y salidas programables que permiten la interacción con otros dispositivos y sistemas. La inclusión de un potenciómetro integrado para el control de velocidad brinda una forma manual de ajustar la velocidad del motor [11].

3.4.2. Fotocélula

Una Fotocélula o fotoeléctrico es un **sensor** que convierte la luz en corriente eléctrica. es decir, si la luz incide sobre ella, genera una corriente proporcional. Si se bloquea la luz, la corriente disminuye. Para mejora dicha detección y poder saber realmente cuando hay un objeto o no, se usan reflectores para así dirigir más luz hacia la fotocélula. Estos sensores son esenciales en aplicaciones de automatización y control industrial.

Figura 20: Fotocélula E3F2

El sensor fotoeléctrico a utilizar es el **E3F2-R2C4** con carcasa cilíndrica M18A y amplificador incorporado, está diseñado para usarse como un interruptor de proximidad óptico. Tiene una clasificación IP67, ofrece distancias de detección de 0-1-2m. Cuenta con protección contra cortocircuitos y conexiones inversas para tipos de conmutación de corriente continua. Es apta para entornos industriales exigentes y está aprobada por *UL* y *CSA*. La salida de control es en *NPN* [12].



(a) *E32F* (b) Reflector

Figura 21: Dibujo Técnico Fotocélula

4. Desarrollo del trabajo

El Desarrollo del Proyecto consistirá en Diseñar todo un esquema analógico mediante *Vivado*, para generar un *PLC*, dicho bloque podrá ser reusado en el diseño tantas veces como la arquitectura de la Placa permita.

Una vez diseñado el Bloque *PLC*, se implementará un sistema de prueba con una mesa giratoria. El sistema tendrá un interruptor para cambiar entre el modo manual y remoto. En el modo manual, se podrá controlar el motor inductivo y la mesa directamente. En el modo remoto, el sistema realizará una vuelta completa, contará las piezas en ambos sentidos y verificará si el conteo es correcto mediante un *LED* en la placa *Zybo*. Además, se implementará 2 pulsadores en el sistema, el reset y un Pulsador de emergencia que parará la mesa.

Para ello, se integrarán dos *PLCs* en una entidad "TOP" que controlará la mesa industrial y coordinará las acciones de los *PLCs*. Uno de los *PLCs* determinará el sentido de giro, mientras que el otro será un contador de piezas.

El sistema de prueba permitirá verificar el funcionamiento del Bloque *PLC* en una aplicación práctica con la mesa giratoria, demostrando su capacidad para controlar dicha mesa en ambos modos y realizar el conteo preciso de las piezas.

4.1. Descripción de bloques

Se va a argumentar y exponer los distintos bloques necesarios para el diseño de nuestro programa en Vivado.

Figura 22: Jerarquía

La Figura 42 muestra la jerarquía de bloques necesarias para el trabajo, que se explica de forma detallada en esta sección.

4.1.1. Sincronizadores

Las entradas asíncronas al sistema, tales como los pulsadores, generan señales ruidosas y difíciles de interpretar para un sistema digital. Además, la señal puede provocar falsas conmutaciones o 'rebotes' que podrían disparar de nuevo el sistema. Para evitar dichos rebotes es necesario un filtro.

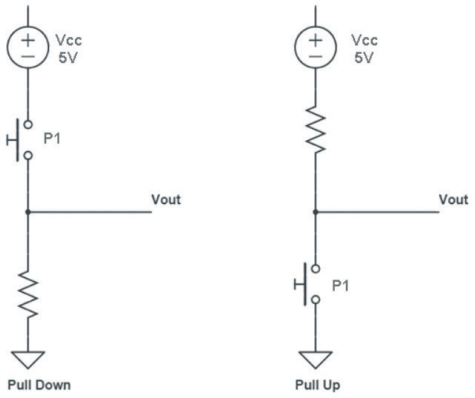


Figura 23: Esquema *Pull Down* y *Pull Up*

El sincronizador se debe diseñar en función del esquema de los pulsadores como se muestra en la Figura 23.

Pull Down

En la configuración *Pull Down* si el circuito se encuentra en reposo, la caída de tensión es prácticamente 0 (LOW), sería un '0' débil, que en *vivado* se interpretaría como ('L' en *stdlogic*). Cuando se acciona el pulsador tendremos 'HIGH'.

Pull Up

En la configuración *Pull Up* ocurre lo contrario, cuando se encuentra el circuito sin pulsar, la caída de tensión es de unos 5V(HIGH), que representaría un nivel parecido al '1' lógico('H' en std_logic).

Para el diseño del sincronizador es necesario un filtro antirrebotes digital, la captura del suceso con independencia de la duración de pulsación, reduciéndolo a un único ciclo de reloj.

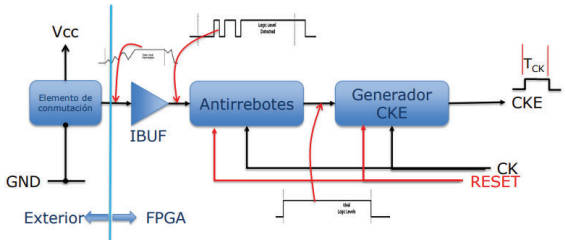


Figura 24: Esquema de Filtrado

El sincronizador ante una entrada de señal Ruidosa, con esquema *Pull down*, se filtrará para que no tenga rebote, y dure un único ciclo de reloj.

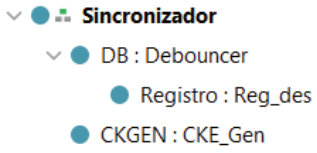


Figura 25: Jerarquía del Sincronizador

Debouncer

Para la Generación del *Debouncer* es necesario un Registro de Desplazamiento.

Los Registros de Desplazamientos son circuitos secuenciales formados por biestables, para controlar la manera de cargar y acceder a los datos que se almacenan.

El registro de Desplazamiento, por tanto, va desplazando cada ciclo de reloj los datos en cada una de las líneas del bus de salida.

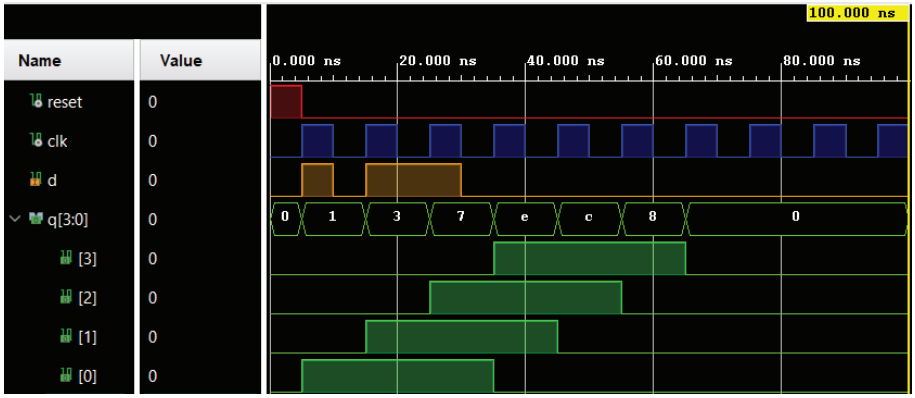


Figura 26: Cronograma del Registro de Desplazamiento

Generador *CKE*

Es necesario diseñar un modelo secuencial, para generar el Pulso de señal filtrado, con duración 1 ciclo de reloj. Para ello se implementa un diagrama de Estados, mediante la codificación tradicional en *VHDL*, así como generar un Proceso Secuencial, para el cambio de Estado cada ciclo de reloj, Y un proceso combinacional donde se tiene en cuenta todos los posibles casos de cada estado. Importante:Si no se tiene en cuenta todos los casos se genera en el esquema *Latch*, lo que acarrearía problemas y dejaría de ser dicho proceso combinacional.

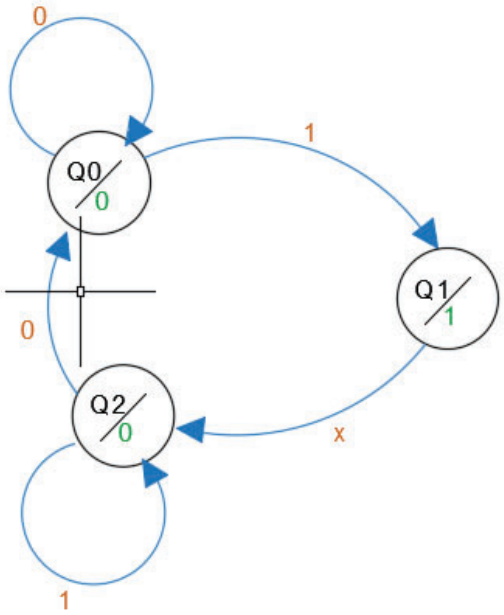


Figura 27: Diagrama de Estados del CKE GEN

El Diagrama de Estados de la Figura 27 se utiliza para que ante la activación de pulsador, únicamente dure 1 ciclo re reloj, evitando rebotes y pulsaciones largas.

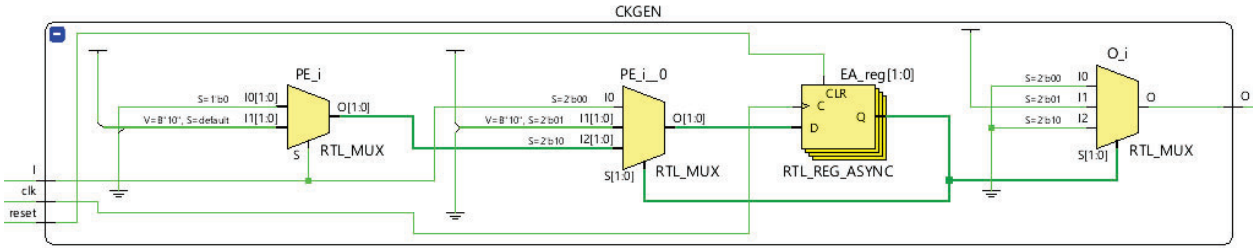


Figura 28: Esquema del CKE GEN

El Esquema de la Figura 28 es la configuración de las distintas conexiones, multiplexores y registros necesarios, generados por *Vivado* a partir del código en *VHDL*.

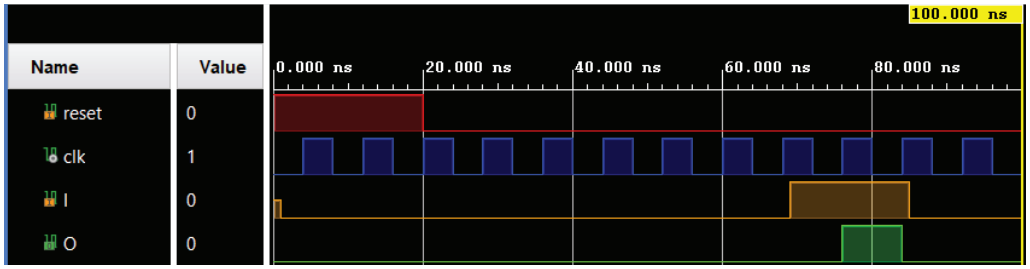


Figura 29: Cronograma del Generador *CKE*

Tras diseñar el *Debouncer* y el Generador *CKE*, es necesario implementarlos en un bloque superior diseñando el Sincronizador.

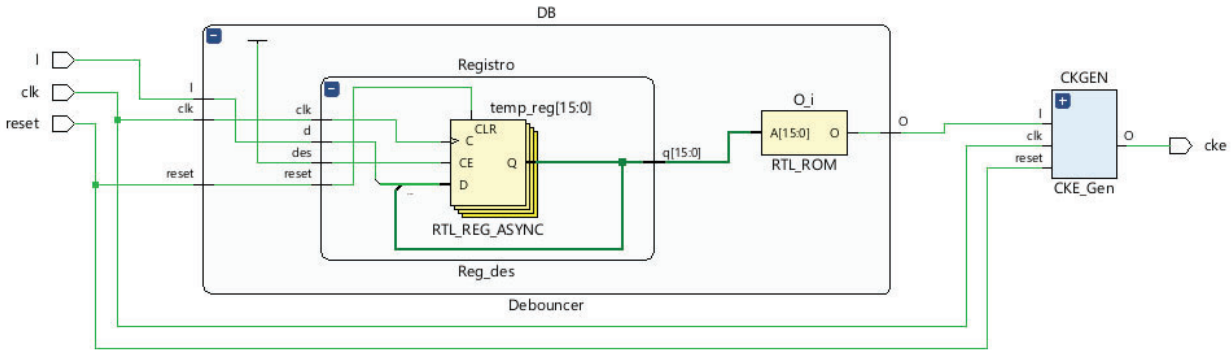


Figura 30: Esquema del Sincronizador

Al agregar las etapas de *Debouncer* y generador de *CKE* al sincronizador, como se muestra en la Figura 30, se garantiza que las señales de entrada se procesen de manera estable y sincronizada con el reloj del sistema, evitando problemas relacionados con el rebote del pulsador y asegurando que la activación sea registrada adecuadamente durante un único ciclo de reloj.

Esta mejora es especialmente útil cuando se trata de señales de entrada que provienen de componentes físicos, como botones o pulsadores, que pueden tener ruido o comportamiento errático en el tiempo. Al implementar un sincronizador con estas características, se mejora la fiabilidad y robustez del sistema digital en general.

El código en *VHDL* necesario para implementar este diseño se encuentra adjuntado en 7.1.1.

4.1.2. FSM

Un *PLC*(Programmable-Logic Controll) es un sistema multipropósito para la industria, con el que se puede implementar de manera digital cualquier FSM, limitado a un número máximo de entradas, salidas y hasta 2^m estados, es decir, en el sistema hardware se puede implementar cualquier autómata que se encuentre por debajo de sus restricciones máximas.

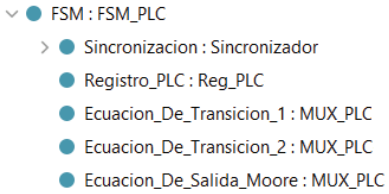


Figura 31: Jerarquía del FSM

Mis Tipos PLC

Para el diseño del proyecto se va a establecer en el sistema unos parámetros fijos de la FSM, siendo kmax=3 entradas,Pmax=4 salidas, M-max=4 biestables, lo que permite un total de 16 posibles estados. Se genera, por tanto, un paquete donde se define los tipos de datos constantes y señales globales, subprogramas, etc.

Se diseña un tipo de Tabla con *arrays*, para generar una *ROM*. La *ROM* no es más que un multiplexor donde las entradas son una memoria de solo lectura que almacena datos permanentes.

Todas las restricciones de la *FSM* estarán presente en una librería como se muestra en el código 7.1.2.

La *FSM* que se va a diseñar para nuestro *PLC* estará compuesto de 3 bloques nuevos:

- Un Registro *PLC*, con el que se actualizará el Estado actual y Próximo Estado del Diagrama de forma síncrona.
- Un multiplexor nombrado *MUX PLC*, encargado de seleccionar una salida a partir de una tabla de valores, dicho bloque nos servirá como Ecuaciones de Transición.
- El Bloque principal FSM que se diseñará de forma que contemple la posible utilización de un Diagrama *Moore* y uno *Mealy*.

Registro *PLC*

Se trata de un Registro implementado mediante un biestable, que actualiza el estado. Toma la señal de entrada "D" como próximo estado y la retiene hasta el próximo flanco de subida del reloj. El biestable se encuentra conectado al reset global.

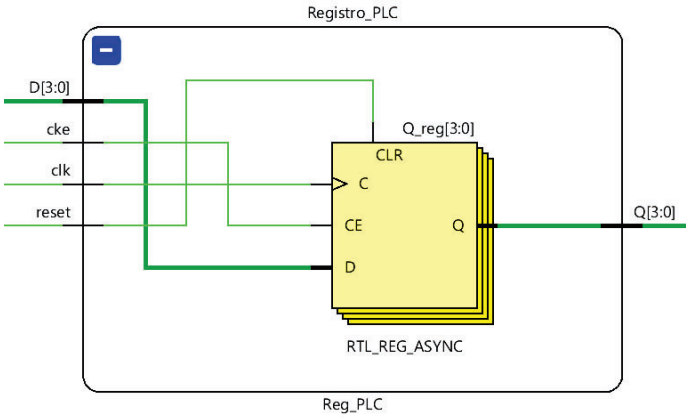


Figura 32: Esquema del Registro

Por tanto, el diseño es esencialmente un elemento de almacenamiento que captura y retiene el próximo estado de la *FSM*. Está implementado en *VHDL* asegurando que sea cambiado de manera controlada y sincronizada con el sistema.

MUX PLC

Es necesario un bloque para crear un Multiplexor que contiene las ecuaciones de transición de nuestra FSM. Se encargará, por tanto, de seleccionar a partir de unas tablas de entrada y la dirección, la salida correcta.

Como las entradas del Multiplexor son tablas constantes, el propio entorno de *vivado* lo interpreta y lo diseña como una *ROM*.

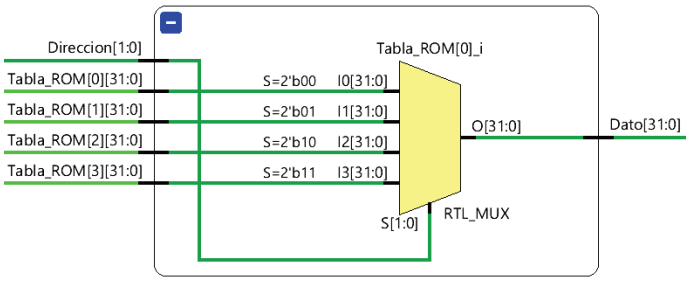


Figura 33: Esquema del *MUX PLC*

El esquema 33 es un ejemplo de una de las ecuaciones de transición necesarias para el bloque principal de la *FSM*, donde como se aprecia, se ha catalogado como una ROM.

FSM_PLC

Es el bloque esencial del Proyecto, mediante el y los sub bloques mencionados, se pueden implementar a cualquier diseño todos los autómatas que se necesiten siempre que cumplan los límites definidos.

El reloj se habilita mediante el *cke* y *Trigger*, utilizando una puerta or.

Para asegurar los correctos tiempos en un diseño *VHDL*, es necesario verificarlos. Estos tiempos son críticos para garantizar que las señales se establezcan y mantengan adecuada en los flancos de reloj.

Es necesario verificar los tiempos de *Setup* y *Hold*:

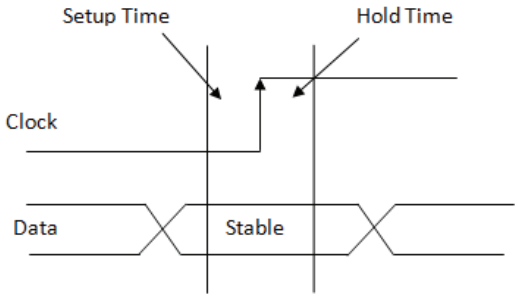


Figura 34: Tiempo *Setup* y *Hold*

- **Tiempo de Setup(T_SU)**: Es el tiempo mínimo que debe pasar antes del flanco de subida del reloj, para que los datos de entrada puedan ser reconocidos por el circuito como válidos, es decir, se verifica el tiempo para asegurar que las señales de entrada están establecidas antes de que el reloj llegue.

- **Tiempo de Hold(T_H)**:es el tiempo mínimo que debe mantenerse los datos de entrada después del flanco de subida de reloj. Para garantizar que se pueden capturar correctamente.

Además, también se comprobará la anchura de pulso y que se cumplen las restricciones de tamaño. Para ello se utiliza las directivas *assert* que son procesos pasivos en VHDL.

Un proceso pasivo es una construcción utilizada para monitorear, verificar o dar información adicional, sin llegar a realizar cambios directos de comportamiento. Por tanto, no son **sintetizables**.

Para utilizar las tablas de verdad que se proporcionan en el bloque superior, son necesarias ecuaciones de transición mediante el bloque *MUX PLC*.

- **Ecuación de Transición 1**: Se encarga de seleccionar, a partir del estado actual, los posibles próximos estados, mediante las tablas de verdad como entrada.

- **Ecuación de Transición 2**:Selecciona a partir de la entrada el próximo estado, siendo su salida finalmente el próximo estado a elegir.

Con esas dos ecuaciones se asignará cuál es el Próximo Estado en el siguiente ciclo de reloj mediante un proceso de asignación.

El Bloque principal FSM se quiere diseñar de forma que contemple la posible utilización de un Diagrama *Moore* y uno *Mealy*, codificando de manera que únicamente será necesario descomentar el método elegido.

Moore

En el caso de *moore* únicamente será necesario implementar una ecuación de transición con el bloque MUX_PLC, para asignar la salida únicamente en función del Estado actual.

Ya que en el diagrama *Moore* las salidas dependen exclusivamente del estado actual, no siendo necesario tener en cuenta para las tablas de verdad las entradas.

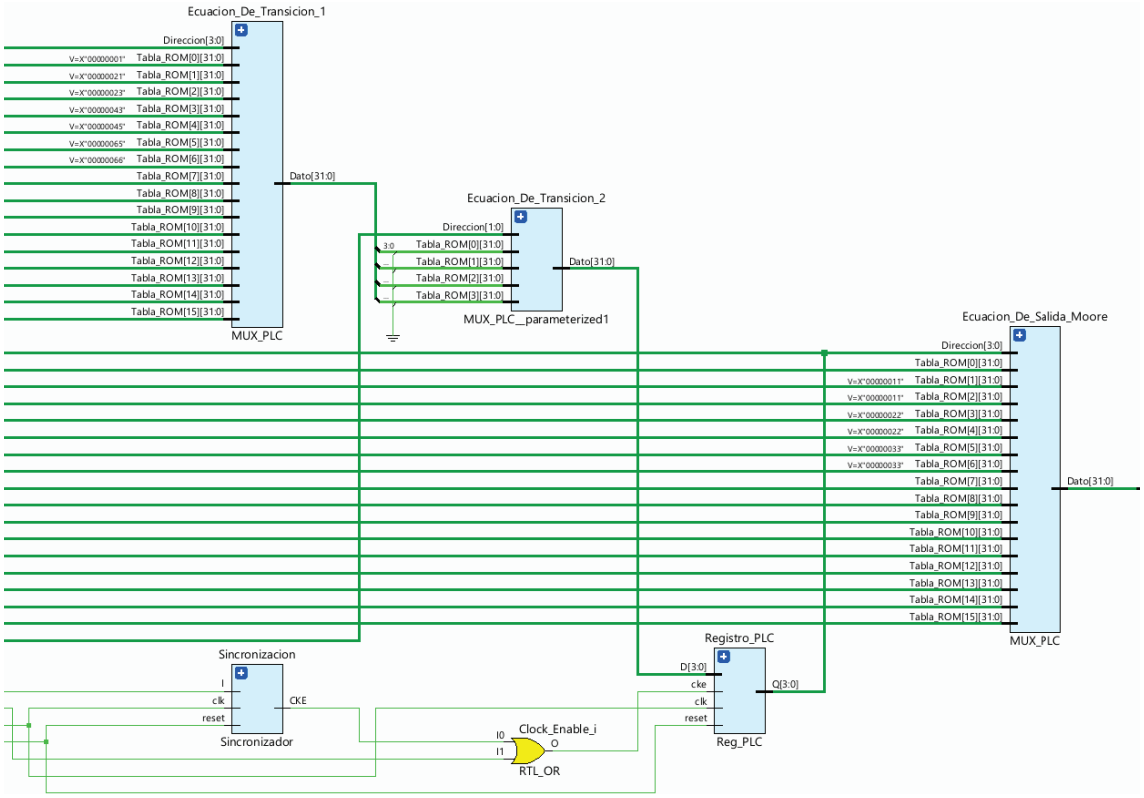


Figura 35: Esquema FSM en *Moore*

Mealy

En *Mealy* la salida sí que depende de la entrada y el estado actual. Por tanto, es necesario realizar el mismo proceso que para asignar el estado con las ecuaciones de transición. Se implementa una ecuación para seleccionar las posibles salidas a partir del estado actual, y una ecuación para seleccionar la salida según la entrada actual.

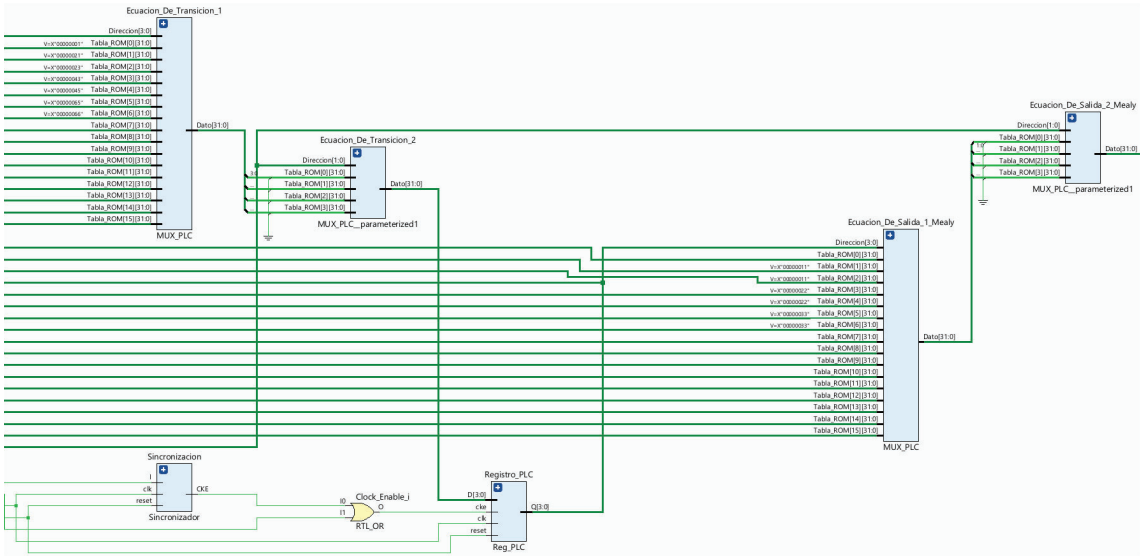


Figura 36: Esquema FSM en *Mealy*

En la Figura 36 se muestra el esquemático diseñado para el caso *mealy*, donde a diferencia del esquema *moore* 35, es necesario implementar 1 bloque ROM más.

El código en *VHDL* necesario para implementar la *FSM* se encuentra adjuntado en los Anexos 7.1.2.

4.1.3. PLC's

En este punto se ha desarrollado el código para utilizar una *FPGA*, como un autómata mediante el bloque *FSM*, siempre que cumpla los parámetros máximos elegidos.

Para poner a prueba el diseño, se ha planteado el control de una Mesa giratoria, realizando una verificación de piezas.

Para demostrar la gran versatilidad de una *FPGA*, se implementará un proyecto con la utilización de 2 FSM, ya que la reconfiguración de la lógica es tan flexible que nos permite implementar todos los *PLC's* que los recursos lógicos y E/S permitan.

Se diseñará un bloque superior que recogerá el bloque FSM y le proporcionará las Tablas de verdad.

4.1.3.1. Contador de Piezas Esta FSM, consistirá en un contador de piezas. El sistema cuenta con 2 sensores fotocélula que se encuentran a nivel alto cuando no hay pieza, en caso contrario se pondrá a '0' lógico.

Como solo es necesario saber si se detecta pieza y no el sentido, la entrada del diagrama de estados será una puerta or de los 2 sensores.

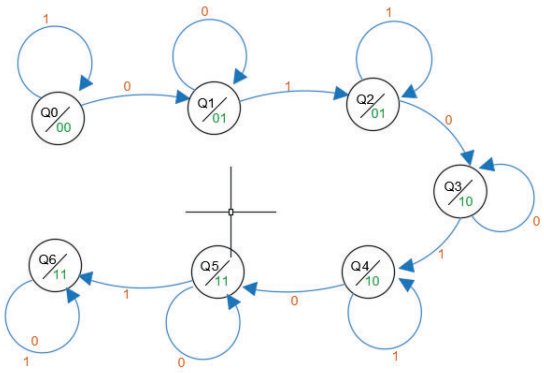


Figura 37: Diagrama de Estados:Contador

El Diagrama de Estados tiene como salida un bus de datos con la cuenta de piezas hasta un máximo de 3. Para no utilizar más recursos. Se trata de un diseño en Moore, por lo que la salida no depende de los estados y se descomenta del código dicho método.

Para contar correctamente las piezas es necesario implementar un estado de retención por cada estado que detecte el objeto, ya que si no al estar un tiempo físico muy grande en comparación al reloj, se contaría sucesivamente, añadiendo a su vez un estado inicial queda un total de 7 estados para un contador de 3 piezas.

$Q\backslash X_1X_0$	1	0
000	000	001
001	010	001
010	010	011
011	100	011
100	100	101
101	110	101
110	110	110

(a) Tabla de Estados

$Q\backslash X_1X_0$	1	0
000	00	00
001	01	01
010	01	01
011	10	10
100	10	10
101	11	11
110	11	11

(b) Tabla de Salidas

Tabla 3: Tablas de Verdad: Contador

Se añaden las tablas de verdad en el código del proyecto como tablas constantes. Los parámetros máximos, como se menciona antes, son Kmax=3 (entradas),Pmax=4 (salidas), Mmax=4(biestables). Clasificando cada valor de las tablas en hexadecimal, al tener 3 entradas (8 posibles casos) y 4 biestables (16 estados), queda una tabla de 16x8. Como no es necesario utilizar todos los recursos, el resto de valores se pondrán a 0.

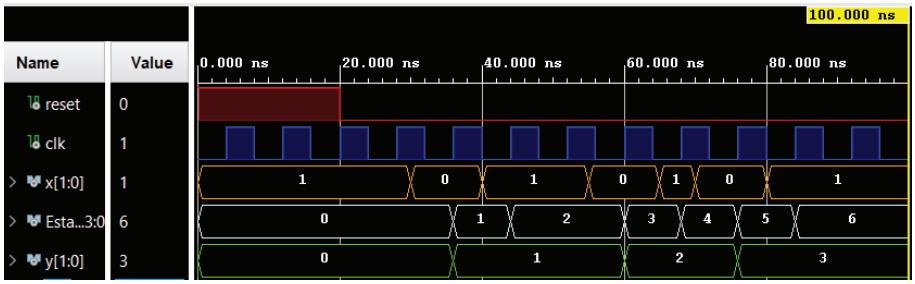


Figura 38: Cronograma del Contador

El cronograma de la Figura 38 es una simulación del bloque *PLC* para el contador, donde el bus de salida muestra el conteo de piezas, y se muestran los estados de retención, donde el contador no avanza. Su esquemático no es más que un bloque superior a la FSM con los buses de entrada y salidas necesarios.

4.1.3.2. Sentido Esta *FSM*, consistirá en una verificación y muestreo del sentido actual. El sistema cuenta con 2 sensores fotocélula como se ha mencionado antes.

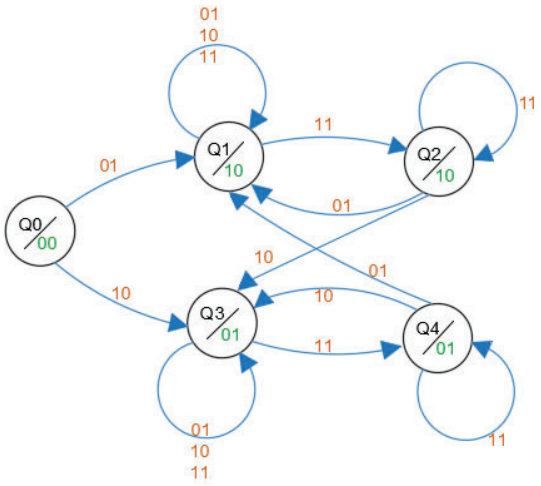


Figura 39: Diagrama de Estados:Sentido

El diagrama de estados ante los 2 sensores, comprobará cuál se ha iniciado antes y de esta manera, se podrá asignar en la salida el sentido de la mesa. La salida será un bus de tamaño 2. en el caso 10 será sentido horario, si es 01 sentido antihorario, 00 no se ha comprobado el sentido todavía y el 11 no puede darse.

Con este diseño, tras pasar una pieza se podrá verificar que el sentido mostrado es el correcto, conectando el bus de salida a 2 leds de nuestra FPGA.

Se añaden las tablas de verdad en el código del proyecto como tablas constantes de la misma forma que se ha realizado para el Contador.

$Q\backslash X_1X_0$	11	10	01	00
000	000	011	001	000
001	010	001	001	001
010	010	011	001	000
011	100	011	011	011
100	100	011	001	000

(a) Tabla de Estados

$Q\backslash X_1X_0$	11	10	01	00
000	00	00	00	00
001	10	10	10	10
010	10	10	10	10
011	01	01	01	01
100	01	01	01	01

(b) Tabla de Salidas

Tabla 4: Tablas de Verdad:Sentido

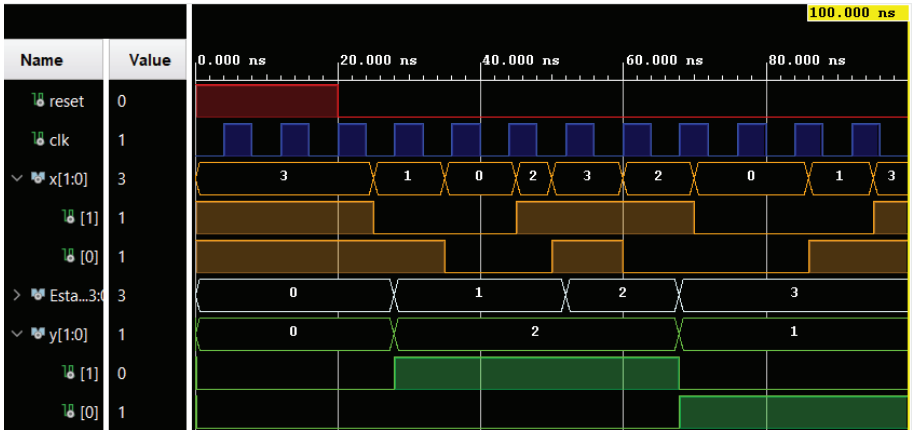


Figura 40: Cronograma del verificador de Sentido

El cronograma 40 contiene una simulación donde se ha ajustado los valores de entrada para verificar el bloque. De esta forma se comprueba que se asigna correctamente el sentido en función de las entradas y los estados que se encuentra.

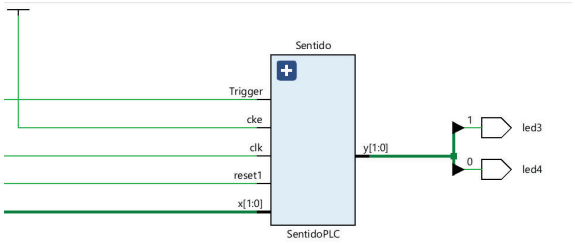


Figura 41: esquema del PLC Sentido

El esquemático del PLC Sentido 41,consiste en un bloque donde se encuentra implementado la FSM, y la salida se asigna a 2 leds de nuestra FPGA, en todo momento del sistema se comprobará el sentido cada vez que pase una pieza por los sensores, mostrándolo por dichos leds.

El reloj para ambos PLC siempre se encuentra activado, colocando el *cke* a '1' lógico.

4.1.4. Unidad de Control

Tras el diseño de las FSM a implementar en el proyecto. Se debe generar el bloque principal encargado de controlar todo el sistema industrial.

4.1.4.1. Planteamiento El planteo del sistema industrial consistirá en, el control de la mesa a través de 2 modos, manual y remoto. En caso de seleccionar el modo manual, se controlará el motor a partir de los *switches* de la *Zybo*. Si se establece el modo automático, el sistema deja de darle el control a los *switches* y realiza un proceso de verificación de piezas.

El proceso de verificación de piezas consistirá en dar una vuelta en 1 sentido, contar las piezas en cuestión y realizar lo mismo en sentido contrario para comprobar que da la misma cuenta en ambos sentidos.

Este proceso conlleva varios procesos específicos. Como es necesario dar 1 vuelta exacta, es necesario contar el tiempo. Para ello se puede implementar un proceso síncrono que cuenta cada ciclo de reloj. La cuenta durante este proceso se realizará mediante el *FSM* diseñado para el conteo de piezas.

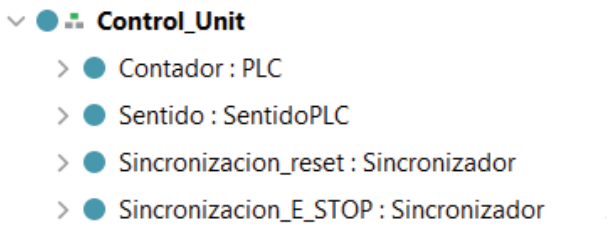


Figura 42: Jerarquía de Unidad de Control

La Figura 42 muestra la jerarquia de bloques para la Unidad de control. Donde se necesitan los 2 PLC's diseñados y 2 sincronizadores, tanto para el pulsador reset como la parada de emergencia.

4.1.4.2. Contador del Tiempo Para trabajar con el tiempo en *VHDL*, es algo más complejo que con otros entornos de programación, los cuales tienen funciones propias para lo mismo. Aunque es posible realizar funciones en *vivado* para esperar tiempos concretos, únicamente son simulables , pero queremos poder contar el tiempo de manera Sintetizable.

Para ello es necesario contar cada ciclo de reloj y de esta forma poder controlar el tiempo. Es necesario un proceso síncrono donde la señal a modificar es un contador que aumenta en el flanco de subida, en caso contrario se mantiene el valor por lo que se trata de un biestable. En cuanto al reset es algo distinto al los procesos síncronos habituales, ya que se ha implementado varios resets intermedios para reiniciar la cuenta del tiempo en caso de ser necesario, además de estar conectado al reset global del sistema.

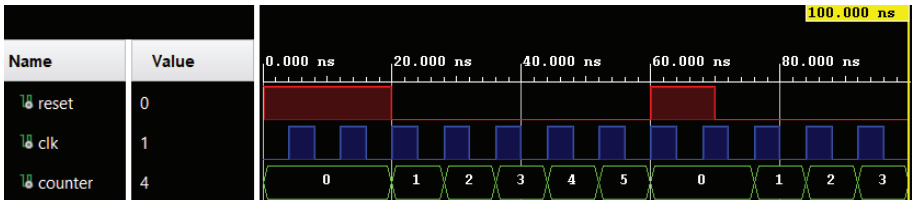


Figura 43: Cronograma del Contador de Tiempo

El cronograma de la Figura 43 muestra la simulación de nuestro contador del tiempo, donde va sumando cada ciclo de reloj. Para saber el tiempo real que ha pasado, será necesario multiplicar la cuenta por el Periodo de reloj, de esta forma sabremos el tiempo real transcurrido.

4.1.4.3. Verificación de Piezas El proceso principal del bloque será un selector entre el modo manual y automático.

En modo automático se realizará un algoritmo cerrado que consiste en la verificación de piezas en ambos sentidos. El objetivo es comprobar que el número de piezas contadas es igual en sentido horario y antihorario.

Esto conlleva la necesidad de poder dar una vuelta exacta, pero se entrará en detalle en la Implementación Física 4.2.

Para comprobar las piezas en ambos sentidos, nos apoyaremos del PLC Contador 4.1.3.1 diseñado, como son necesarios 2 contadores, se utilizan 2 señales intermedias.

Importante: Ambos contadores deben retener su valor en memoria, para realizarlo correctamente y evitar *Latch*, es necesario implementarlo con procesos síncronos, actualizando su valor con los flancos de subida del reloj.

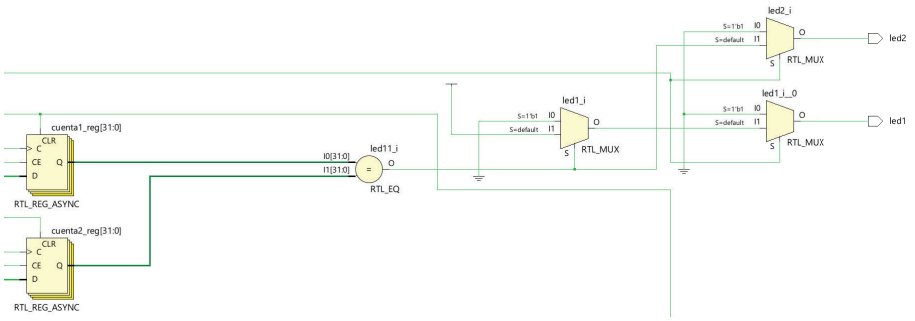


Figura 44: Esquema la Verificación de Piezas

Tras terminar el algoritmo, se comprueba si las cuentas son iguales, y dependiendo de la respuesta se enciende un led u otro, como se muestra en el Esquema 44.

En todo momento el *PLC* del sentido estará en funcionamiento, determinando el

sentido actual cada vez que pase una pieza. A diferencia del PLC del Contador que únicamente actúa para la verificación de piezas.

Además de Todo lo mencionado, se implementará un Pulsador de emergencia que, en caso de su activación, para el motor y lo bloquea, hasta que se resetee el sistema.

Los esquemas RTL y Síntesis del proyecto se encuentran adjuntados en el Anexo 7.2.

4.2. Implementación y Montaje Físico

El montaje físico y su implementación se puede separar en 2 grupos principales:

- La mesa industrial, con los módulos, el variador de frecuencia, motor inductivo, sensores, piezas, etc.
- El sistema de control del sistema compuesto por la FPGA, los relés, convertidores de potencia, etc.

4.2.1. Mesa Giratoria

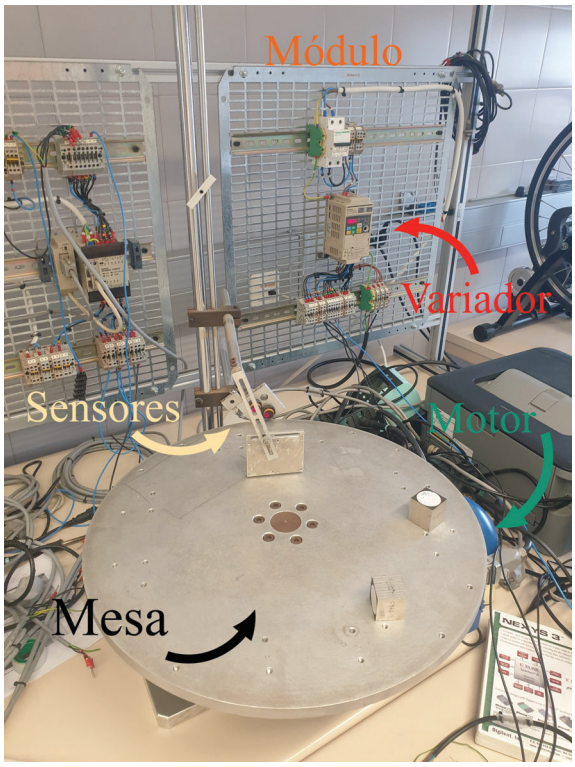
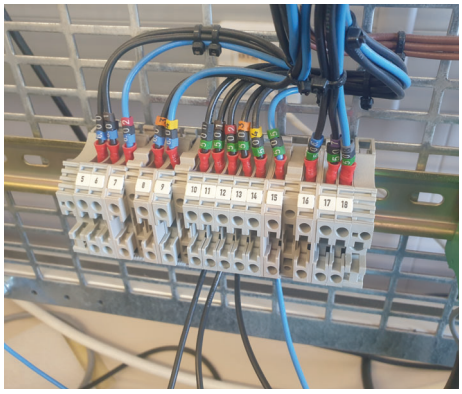


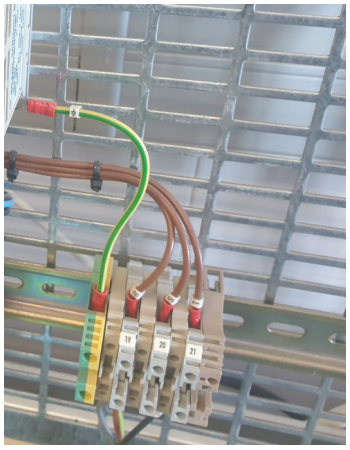
Figura 45: Mesa Giratoria

En la Imagen de la Figura 45 se puede apreciar los distintos elementos de la Mesa. Primero vamos a trabajar con el conexionado entre el motor trifásico y el variador.

Conexión Inverso-Motor



(a) Módulo del variador



(b) Conexionado del motor inductivo

Figura 46: Conexionado de la Interfaz de control

La Figura 46b es la alimentación del motor mediante sus terminales de conexión. Esta fuente se destina a un motor trifásico asíncrono, lo que implica la necesidad de tres líneas de alimentación identificadas como R, S y T. Además se incluye una línea destinada a la tierra de protección.

De esta forma se encuentra listo el esquema entre el Inversor y el motor trifásico.

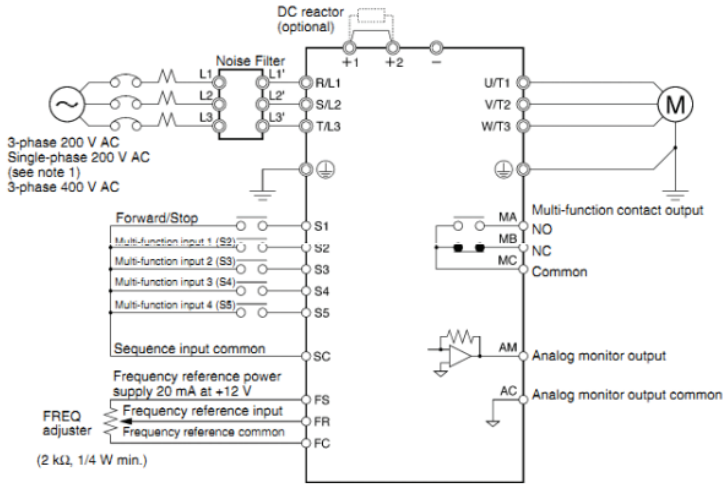


Figura 47: Esquema Control del motor

En el esquema 47 se puede apreciar como queda representado el diseño. Nos interesa utilizar los *Multi-function*, siendo las señales del sistema que nos permite controlar el motor, las señales trabajan a 24 V/DC.

Conexión Inversor-*FPGA*

Origen		Destino	
Cable	Borna	Relé	Señal
500	10	1	Run
501	11	2	Bloqueo
502	12	3	Sentido
505	15	1,2,3,4	Común

Tabla 5: Asignación de Cables

Las señales de salida del inversor para controlar la mesa son 4, una de avance que cuando está a '0' lógico permite el paso de los 24 V y avanza el motor inductivo a la frecuencia establecida desde el inversor. Una señal de Bloqueo, que cuando se encuentra a '1' lógico impide iniciar el movimiento del motor, pero en caso de estar ya en movimiento no actúa. Una señal que determina el sentido del motor, siendo horario para el '0' lógico y antihorario en '1' lógico. Una última señal común que va al resto de relés.

4.2.2. Sistema de Control

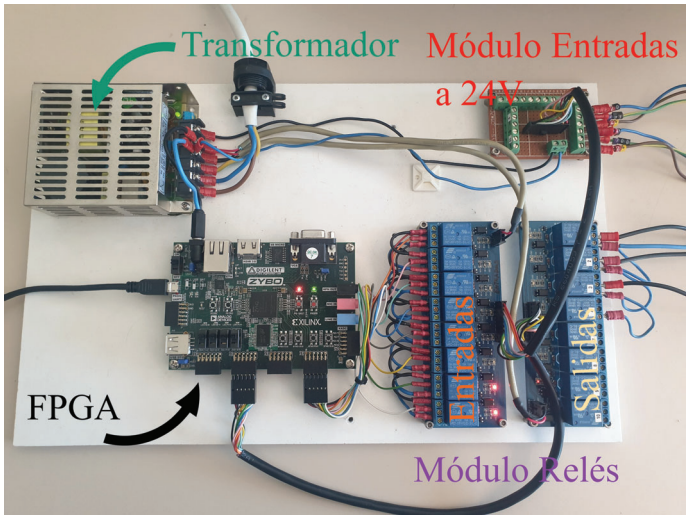


Figura 48: Sistema de control

La figura 48 muestra el sistema de control con la *FPGA*, transformador, relés y un módulo con bornas para enlazar conexiones a 24V. Los relés se encuentran agrupados en 2 módulos de 8 relés cada uno, uno para las entradas de la *FPGA* y otra para las salidas.

(a) Sensores fotocélula (b) Módulo sensores

Figura 49: Conexionado de los sensores

Los sensores fotocélula están compuestos por 3 cables, alimentación, tierra y la señal. El cable marrón es la alimentación de 24 V, el cable azul la tierra y el cable Verde/amarillo es la señal de salida como queda reflejado en el esquema 50.

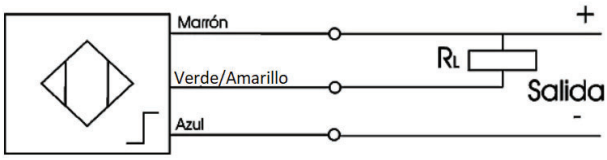


Figura 50: Esquema del Sensor

El módulo mostrado en la figura 49b son el conjunto de entradas que mediante un cable principal se conecta al módulo de relés izquierdo.

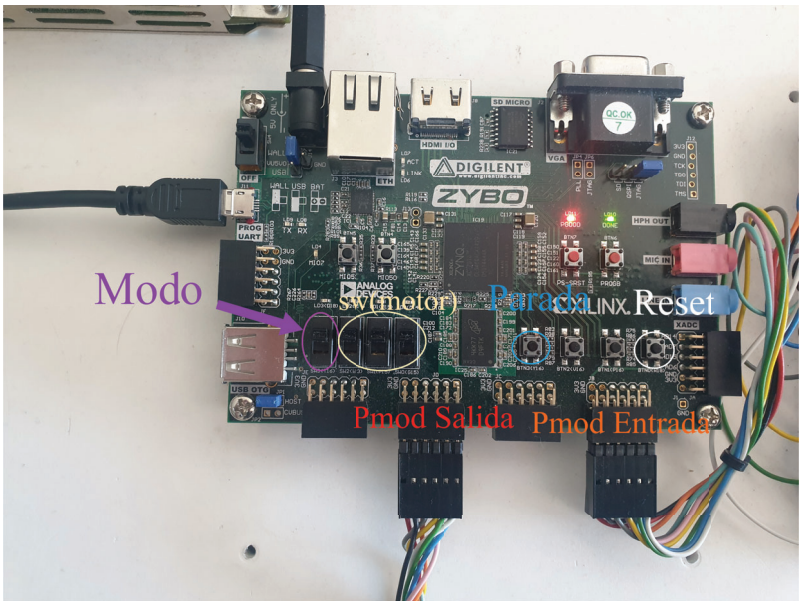


Figura 51: Zybo

En la Figura 51 se muestra el elemento principal del proyecto, encargado de controlar todo el sistema. Los switches se han asignado 1 para seleccionar el modo del sistema, y los 3 restantes para controlar el motor en el modo manual. Respecto a los botones, únicamente se han conectado 2, uno para el reset general y otro como parada de emergencia de la mesa. Los *pmods* tenemos el JD asociado a la salida, para controlar las señales del inversor y, por tanto, el motor. El *pmod* JB es para las entradas, siendo únicamente los 2 sensores fotoeléctricos.

Una vez concluido el proceso integral de ensamblaje del proyecto y sustentado la implementación del diseño en la placa 4.1, solo quedaría verificar el correcto funcionamiento del sistema y comprobar que, para diversas situaciones, actúa correctamente.

Las simulaciones y verificaciones del sistema encuentran de manera pública y adjuntadas en la bibliografía [13].

5. Conclusión y Líneas Futuras

El desarrollo del proyecto sobre el Diseño de un *PLC* con lógica programable para control industrial, ha sido un proceso esclarecedor en el ámbito de la ingeniería, específicamente en el contexto de un Trabajo de Fin de Grado. Se ha demostrado de manera efectiva la profunda utilidad que una *FPGA* puede proporcionar. Durante el proceso de exploración, hemos realizado comparaciones significativas entre el empleo de *FPGA* y otros sistemas embebidos.

Dentro de este trayecto, se han identificado obstáculos y desafíos que requerían soluciones cuidadosas, no solo en el ámbito del diseño de bloques, sino en el Montaje físico del sistema. Siendo necesario una gran habilidad para manejarse adecuadamente con el lenguaje *VHDL* y el entorno de *vivado*. La implementación práctica del entorno industrial ha subrayado la necesidad de poner en práctica habilidades con el ensamblaje y conexionado de componentes, para realizar su correcto esquema, permitiendo una evaluación del programa implementado.

Este proyecto ha logrado demostrar la versatilidad sobresaliente de las *FPGA*, las cuales pueden desempeñarse de una manera similar a otros sistemas embebidos, estando sujeta únicamente a las limitaciones de recursos físicos y la experiencia del individuo que las realiza.

A líneas futuras, este proyecto sirve como una base no solo para su aplicación en situaciones industriales donde se requiera la implementación de autómatas, sino como un recurso educativo. Puede ser una herramienta didáctica en futuros estudios, permitiendo a los estudiantes explorar de manera práctica el control de diversas configuraciones de mesas industriales en la Escuela de Ingenieras Industriales.

6. Bibliografía

Referencias

[1] R. Ferreiro García, "Nociones Sobre Aplicación de PLC's Al Control de Procesos Industriales,"La Coruña, Spain: Universidade da Coruña, Servicio de Publicaciones, 1995. [Fecha consulta: 18/04/2023].

[2] A. Simon, .Autómatas programables: programación, automatismo y lógica programada,"Madrid, Spain: Paraninfo, 1988. [Fecha consulta: 20/04/2023].

[3] SEIKA Automation, "5 Lenguajes de Programación para PLC — SEIKA Automation,"[En línea]. Disponible en:https://www.seika.com.mx/5-lenguajes-de-programacion-para-plc/. [Fecha consulta: 22 de abril de 2023].

[4] E. Monmasson y M.N. Cirstea, "FPGA Design Methodology for Industrial Control Systems - A Review,ÏEEE Transactions on Industrial Electronics, vol. 54, no. 4, pp. 1824-1842, julio de 2007. [Fecha consulta: 23/04/2023]

[5] D. Daroch Montoya y S. Antiñiré Monje, .Estudio del mundo de las FPGAs comparativas e implementación,"2018. [Fecha consulta: 23/04/2023]

[6] Wakerly, John F. Digital Design: Principles and Practices. 3rd ed. Upper Saddle River, New Jersey: Prentice Hall, 2001. Print.[Fecha consulta: 25/05/2023]

[7] Digilent.(2017). Zybo Reference Manual.https://digilent.com/reference/programmable-logic/zybo/reference-manual.[Fecha consulta:29/04/2023]

[8] Pedroni, V. A. (2020). Circuit Design with VHDL, Third Edition. The MIT Press. [Fecha consulta: 30/04/2023]

[9] B. J. LaMeres, Ïntroduction to Logic Circuits & Logic Design with VHDL,"2nd ed. [Fecha consulta: 30/04/2023]

[10] Crowder, Richard M. Electric Drives and Electromechanical Systems: Applications and Control. Second edition. Kidlington, Oxford, England; Butterworth-Heinemann, 2020. [Fecha de consulta: 24/06/2023].

[11] SYSDRIVE 3G3JV - User's Manual, OMRON. [En línea]. Disponible en: https://assets.omron.eu/downloads/manual/en/i528_3g3jv_users_manual_en.pdf. [Fecha de consulta: 10/06/2023].

[12] "Photoelectric sensor E3F2 Datashee". [En línea]. Disponible en: http://www.edata.omron.com.au/eData/PEs/E58E-EN-01.pdf. [Fecha de consulta: 18/06/2023].

Recursos adicionales

[13] Carpeta de Simulaciones en Google Drive, "Simulaciones del TFG"[En línea]. Disponible en: https://drive.google.com/drive/folders/1gVnPJHEBIaqraZz6N_7X30Xq0Y6UgHwx?usp=drive_link [Acceso: 20 de agosto de 2023].

7. Anexos

7.1. Código

7.1.1. Sincronizador

Generador CKE

```
1
2
3  library IEEE;
4  use IEEE.STD_LOGIC_1164.ALL;
5
6  -- Uncomment the following library declaration if using
7  -- arithmetic functions with Signed or Unsigned values
8  --use IEEE.NUMERIC_STD.ALL;
9
10 -- Uncomment the following library declaration if instantiating
11 -- any Xilinx leaf cells in this code.
12 --library UNISIM;
13 --use UNISIM.VComponents.all;
14
15
16 --Bloque para generar un pulso de reloj al activar un pulsador
17 entity CKE_Gen is
18     Port ( I : in STD_LOGIC;
19            O : out STD_LOGIC;
20            reset : in STD_LOGIC;
21            clk : in STD_LOGIC);
22 type Estado is ( E_1, E_2, E_3); --3 Estados.
23 end CKE_Gen;
24
25 architecture FSM_Simple of CKE_Gen is
26 signal EA, PE : Estado; -- EA es el estado actual y PE es el
       pr ximo estado.
27 begin
28
29
30 Secuencial: process(clk,reset)
31                          begin
32                                  if reset='1' then
33                                      EA<= E_1;
34                                  elsif rising_edge(clk) then
35                                      EA<= PE;
36                                  end if;
37                          end process Secuencial;
38
39
40 Combinacional: process(I,EA)
41                                  begin
```

```
                                        case EA is
                                        when E_1 =>
                                                O <= '0';
                                                if I='0' then
                                                  PE <= E_1;
                                                else
                                                  PE <= E_2;
                                                end if;
                        when E_2 =>
                                        O  <= '1';
                                        PE <= E_3;
                                when E_3 =>
                                        O <= '0';
                                        if I='0' then
                                            PE <= E_1;
                                        else
                                            PE <= E_3;
                                        end if;
                                end case;
                    end process Combinacional;
end FSM_Simple;
```

Registro Desplazamiento

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Reg_Des is

    generic( n     : integer := 8);
    Port    ( d     : in STD_LOGIC;
              q     : out STD_LOGIC_VECTOR(n-1 downto 0);
              reset : in STD_LOGIC;
              des   : in STD_LOGIC;
              clk   : in STD_LOGIC);
```

```
end Reg_Des;

architecture Simple of Reg_Des is

signal temp : std_logic_vector(n-1 downto 0);

begin

    process(clk,reset)
    begin
        if reset='1' then

                    temp <= (others => '0');

                elsif rising_edge(clk) then

            if des='1' then

                                for i in temp'high downto 1 loop

                                            temp(i) <= temp(i-1);
                                end loop;
                                temp(0) <= d;
                        end if;
             end if;
    end process;
    q <= temp;
end Simple;
```

Debouncer

```


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Debouncer is
```

```
17      generic ( n      : natural := 4 );
18      Port    ( I      : in STD_LOGIC;
19                O      : out STD_LOGIC;
20                reset  : in STD_LOGIC;
21                clk    : in STD_LOGIC );
22  end Debouncer;
23
24  architecture Simple of Debouncer is
25
26  signal   s         : STD_LOGIC_VECTOR(n-1 downto 0);
27  signal   Resultado : STD_LOGIC; -- Contiene la AND de todos los
        bits de s.
28  component Reg_Des is
29      generic( n      : integer);
30      port    ( d      : in STD_LOGIC;
31                q      : out STD_LOGIC_VECTOR(n-1 downto 0);
32                reset  : in STD_LOGIC;
33                des    : in STD_LOGIC;
34                clk    : in STD_LOGIC);
35  end component Reg_Des;
36
37  begin
38
39  Registro: Reg_Des   generic map(n)
40                      port map(d=>I,q=>s,reset=>reset, des=>'1',
                            clk=>clk);
41  AND_n : process(s)
42      variable Parcial : STD_LOGIC;
43      begin
44          parcial := '1';
45          for i in 0 to n - 1 loop
46              Parcial := Parcial and s( i );
47          end loop;
48          Resultado <= Parcial;
49      end process AND_n;
50
51      O <= transport '1' after 1 ns when Resultado = '1' else '0'
            after 1 ns;
52
53  end Simple;
```

Sincronizador

```
1
2
3
4
```

```
5  library IEEE;
6  use IEEE.STD_LOGIC_1164.ALL;
7
8  -- Uncomment the following library declaration if using
9  -- arithmetic functions with Signed or Unsigned values
10 --use IEEE.NUMERIC_STD.ALL;
11
12 -- Uncomment the following library declaration if instantiating
13 -- any Xilinx leaf cells in this code.
14 --library UNISIM;
15 --use UNISIM.VComponents.all;
16
17 entity Sincronizador is
18     generic ( n     : natural := 4 );
19     Port    ( I     : in STD_LOGIC;
20               CKE   : out STD_LOGIC;
21               reset : in STD_LOGIC;
22               clk   : in STD_LOGIC);
23 end Sincronizador;
24
25 architecture Estructura of Sincronizador is
26
27 signal    s1, s2 : STD_LOGIC;         -- se ales internas de la
      estructura.
28
29 component Debouncer is generic ( n     : natural := 4 );
30                         port    ( I     : in std_logic;
31                                   O     : out std_logic;
32                                   reset : in std_logic;
33                                   clk   : in std_logic);
34 end component Debouncer;
35 component CKE_Gen is port( I : in std_logic; O : out std_logic;
      reset: in std_logic; clk: in std_logic);
36 end component CKE_Gen;
37
38 begin
39 DB:     Debouncer generic map( n )
40                   port    map (I=>I, O=>s2, reset=>reset,clk=>
                        clk);
41 CKGEN:  CKE_Gen   port    map (I=>s2,O=>CKE,reset=>reset,clk=>
      clk);
42 end Estructura;
```

7.1.2. FSM

Mis Tipos PLC

```
1
2  library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

Package Tipos_FSM_PLC is    -- En este paquete defino mis tipos
                            -- de datos constantes y se ales
                               globales, subprogramas, etc.
                            -- El tipo Tabla es com n para
                               generar una ROM    un Multiplexor
                               .
    constant N_Bits_Dato : Natural := 32; -- Anchura del dato de
        la tabla.
    type Tabla_FSM is array( Natural range <> ) of
       STD_LOGIC_VECTOR( N_Bits_Dato - 1 downto 0 );
                            -- Requerimientos de E/S del
                               enunciado:
    constant        k_Max : natural := 3;   -- 3 entradas como
       m ximo.
    constant        p_Max : natural := 4;   -- p salidas como
       m ximo.
    constant        m_Max : natural := 4;   -- m biestables como
       m ximo. (Hasta 16 estados)
end Tipos_FSM_PLC;
```

Registro PLC

```


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

--Registro que actualiza el Estado del Diagrama

```

```
17 entity Reg_PLC is
18    generic( N_Bits_Reg : integer := 8;
19             T_D        : time := 10 ps ); -- Tiempo de retardo
                 desde el flanco activo del reloj hasta la
                 actualizaci n de la salida Q.
20    Port    ( D : in  STD_LOGIC_VECTOR(N_Bits_Reg-1 downto 0);
21              Q : out STD_LOGIC_VECTOR(N_Bits_Reg-1 downto 0);
22           reset : in STD_LOGIC;
23             cke : in STD_LOGIC;
24             clk : in STD_LOGIC);
25 end Reg_PLC;
26
27 architecture Comportamiento of Reg_PLC is
28 begin
29 Registro: process(clk,reset)
30     begin
31         if reset = '1' then
32             Q <= (others => '0');
33         elsif rising_edge(clk) then
34             if cke = '1' then
35                 Q <= D after T_D;
36             end if;
37         end if;
38     end process Registro;
39 end Comportamiento;
```

MUX PLC

```
1
2
3  library IEEE;
4  use IEEE.STD_LOGIC_1164.ALL;
5  use work.Tipos_FSM_PLC.all;
6  -- Uncomment the following library declaration if using
7  -- arithmetic functions with Signed or Unsigned values
8  use IEEE.NUMERIC_STD.ALL;
9
10 -- Uncomment the following library declaration if instantiating
11 -- any Xilinx leaf cells in this code.
12 --library UNISIM;
13 --use UNISIM.VComponents.all;
14
15 entity MUX_PLC is
16     generic( N_Bits_Dir : Natural := 3;
17              T_D        : time    := 10 ps ); -- Tiempo de
                 retardo desde el cambio de direcci n hasta la
                 actualizaci n de la salida Q.
18     Port    ( Direccion  : in  STD_LOGIC_VECTOR ( N_Bits_Dir  - 1
         downto 0 );
```

```
19              Dato       : out STD_LOGIC_VECTOR ( N_Bits_Dato - 1
                    downto 0 );
20              Tabla_ROM  : in  Tabla_FSM( 0 to 2**N_Bits_Dir - 1
                   ) );
21  end MUX_PLC;
22
23  architecture Comportamiento of MUX_PLC is
24
25  begin
26      Dato <= transport Tabla_ROM( to_integer( unsigned( Direccion
            ) ) ) after T_D;
27  end Comportamiento;
```

FSM

```
1
2
3
4   library IEEE;
5   use IEEE.STD_LOGIC_1164.ALL;
6
7
8   -- Uncomment the following library declaration if using
9   -- arithmetic functions with Signed or Unsigned values
10  use IEEE.NUMERIC_STD.ALL;
11
12  -- Uncomment the following library declaration if instantiating
13  -- any Xilinx leaf cells in this code.
14  --library UNISIM;
15  --use UNISIM.VComponents.all;
16
17  use work.Tipos_FSM_PLC.all;
18
19  entity FSM_PLC is
20      generic( k    : natural := 32;     -- k entradas.
21               p    : natural := 32;     -- p salidas.
22               m    : natural := 32;     -- m biestables. (Hasta 16
                     estados)
23               T_DM : time    := 10 ps; -- Tiempo de retardo desde
                     el cambio de direcci n del MUX hasta la
                    actualizaci n de la salida Q.
24               T_D  : time    := 10 ps; -- Tiempo de retardo desde
                     el flanco activo del reloj hasta la
                    actualizaci n de la salida Q.
25               T_SU : time    := 10 ps; -- Tiempo de Setup.
26               T_H  : time    := 10 ps; -- Tiempo de Hold.
27               T_W  : time    := 10 ps); -- Anchura de pulso.
28       port    (   x : in  STD_LOGIC_VECTOR( k - 1 downto 0 );
            -- x es el bus de entrada.
```

```
29                  y : out STD_LOGIC_VECTOR( p - 1 downto 0 );
                        -- y es el bus de salida.
30                Tabla_De_Estado : in Tabla_FSM( 0 to 2**m - 1 );
                    -- Contiene la Tabla de Estado estilo Moore: Z(
                    n+1)=T1(Z(n),x(n))
31                Tabla_De_Salida : in Tabla_FSM( 0 to 2**m - 1 );
                    -- Contiene la Tabla de Salida estilo Moore: Y(
                    n  )=T2(Z(n))
32                clk      : in STD_LOGIC;   -- La se al de reloj.
33                cke      : in STD_LOGIC;   -- La se al de
                    habilitaci n de avance: si vale '1' el
                    aut mata avanza a ritmo de clk y si vale '0'
                    manda Trigger.
34                reset    : in STD_LOGIC;   -- La se al de
                    inicializaci n.
35
36                Trigger : in STD_LOGIC ); -- La se al de disparo
                    (single shot) as ncrono y pos blemente con
                    rebotes para hacer un avance  nico . Ha de
                    llevar un sincronizador.
37  end FSM_PLC;
38
39  architecture Estructura of FSM_PLC is
40
41  component MUX_PLC is    -- Implementa las ecuaciones de
        transici n y salida estilos Moore    Mealy.
42      generic( N_Bits_Dir : Natural := 3;
43               T_D        : time    := 10 ps ); -- Tiempo de
                    retardo desde el cambio de direcci n hasta la
                    actualizaci n de la salida Q.
44      Port    ( Direccion  : in  STD_LOGIC_VECTOR ( N_Bits_Dir  - 1
            downto 0 );
45                Dato       : out STD_LOGIC_VECTOR ( N_Bits_Dato - 1
                      downto 0 );
46                Tabla_ROM  : in  Tabla_FSM( 0 to 2**N_Bits_Dir - 1
                    ) );
47  end component MUX_PLC;
48
49  component Reg_PLC is    -- Registro de estado del aut mata
        estilos Moore    Mealy.
50      generic( N_Bits_Reg : integer := 8;
51               T_D        : time := 10 ps ); -- Tiempo de retardo
                    desde el flanco activo del reloj hasta la
                    actualizaci n de la salida Q.
52      Port    ( D          :  in STD_LOGIC_VECTOR( N_Bits_Reg - 1
          downto 0 );
53                Q          : out STD_LOGIC_VECTOR( N_Bits_Reg - 1
                    downto 0 );
```

```
54              reset       :  in STD_LOGIC;
55              cke         :  in STD_LOGIC;
56              clk         :  in STD_LOGIC);
57  end component Reg_PLC;
58
59  component Sincronizador is -- Genera el CKE libre de rebotes y
        de duraci n 1 ciclo de reloj clk.
60      generic ( n      : natural := 4 );
61      Port     ( I      : in STD_LOGIC;
62                 CKE    : out STD_LOGIC;
63                 reset  : in STD_LOGIC;
64                 clk    : in STD_LOGIC);
65  end component Sincronizador;
66
67  signal Estado_Actual       : STD_LOGIC_VECTOR( m - 1 downto 0 );
68  signal Proximo_Estado      : STD_LOGIC_VECTOR( m - 1 downto 0 );
69  signal Estados_Con_Formato : Tabla_FSM( 0 to 2**k - 1 );
70  signal Salidas_Con_Formato : Tabla_FSM( 0 to 2**k - 1 ); --
        S lo sirve para Mealy.
71  signal Salida_Mux          : STD_LOGIC_VECTOR( N_Bits_Dato - 1
        downto 0 ); -- Pr ximo estado justificado a la derecha con
        ceros.
72  signal Salida              : STD_LOGIC_VECTOR( N_Bits_Dato - 1
        downto 0 ); -- Salida justificada a la derecha con ceros.
73
74  signal Dato_Intermedio_1 : STD_LOGIC_VECTOR( N_Bits_Dato - 1
        downto 0 );
75  signal Dato_Intermedio_2 : STD_LOGIC_VECTOR( N_Bits_Dato - 1
        downto 0 ); -- S lo sirve para Mealy.
76
77  signal Pos_Trigger  : STD_LOGIC; -- Trigger filtrado y sin
        rebotes, con la anchura de un ciclo de reloj.
78  signal Clock_Enable : STD_LOGIC; -- Se al de trigger y cke
        procesada con una OR.
79
80
81  begin
82  --
83  -- Verificaciones b sica de rangos y tiempos:  NO  SON
        SINTETIZABLES!
84  --
85
86  Comprueba_2kxm : process
87                   begin
88                       assert( 2**k * m <= N_Bits_Dato )    report
                            " No cumple la restricci n de tama o!"
                            severity failure;
89                       wait;
```

```
90                  end process Comprueba_2kxm;
91
92  Comprueba_2kxp : process
93                  begin
94                      assert( 2**k * p <= N_Bits_Dato )    report
                           " No cumple la restricci n de tama o!"
                           severity failure;
95                      wait;
96                  end process Comprueba_2kxp;
97
98  Comprueba_TSUH: process -- Proceso Pasivo para la verificaci n
       del Setup y Hold.
99                  begin
100                     wait until rising_edge(clk);
101                     assert (Proximo_Estado'Stable( T_SU ) )
                           report " No cumple el tiempo de Setup!"
                           severity failure;
102                         wait for T_H;
103                     assert (Proximo_Estado'Stable( T_H  ) )
                           report " No cumple el tiempo de Hold!"
                           severity failure;
104 end process Comprueba_TSUH;
105
106 Comprueba_TW:   process -- Proceso Pasivo para la verificaci n
       de la anchura de un pulso (Width).
107                 begin
108                     wait until rising_edge(clk); -- IMPORANTE !
                           ESTA L NEA SUSTITUYE A LA SIGUIENTE PARA
                            SINTESIS.(ES REALMENTE EQUIVALENTE)
109 --                    wait until Proximo_Estado'Event; -- D
       representa la se al a medir (std_Logic_Vector).
110                     assert (Proximo_Estado'Delayed'Stable( T_W )
                            ) report " No cumple la anchura de pulso
                           !" severity Error;
111                 end process Comprueba_TW;
112
113 -- Procesos del FSM.
114 Sincronizacion: Sincronizador
115                 generic map( 4 ) -- El sincronizador tiene 32
                       etapas en el filtro FIR (Finite Impulse
                       Response).
116                 port    map( I     => Trigger,
117                              CKE   => Pos_Trigger,
118                              reset => reset,
119                              clk   => clk );
120
121 Generacion_CKE: Clock_Enable <= Pos_Trigger or cke;
122
```

```
123 Registro_PLC:   Reg_PLC -- Registro de estado del aut mata.
124                 generic map( m, 10 ps )
125                 Port    map( D     => Proximo_Estado,
126                              Q     => Estado_Actual,
127                              reset => reset,
128                              cke   => Clock_Enable,
129                              clk   => clk);
130
131
132 --
133 -- Ecuaci n de Transici n de estado en los dos estilos: Mealy
    y Moore.
134 --
135
136 -- ajusta a la derecha con ceros de relleno, m bits del Estado a
     N_Bits_Dato bits del dato del MUX.
137 Asignacion_MUX : process( Dato_Intermedio_1 )
138     begin
139         for i in 0 to 2**k - 1 loop
140             Estados_Con_Formato( i ) <= std_logic_vector( resize
                ( unsigned( Dato_Intermedio_1( m * ( i + 1 ) - 1
                downto m * i ) ), N_Bits_Dato ) );
141         end loop;
142     end process Asignacion_MUX;
143
144 Ecuacion_De_Transicion_1: MUX_PLC generic map( m, T_DM ) --
    Selecciona a partir del estado actual todos lo posibles
    pr ximos estados.
145                                 Port    map( Direccion =>
                                       Estado_Actual,
146                                          Dato      =>
                                             Dato_Intermedio_1
                                             ,
147                                          Tabla_ROM =>
                                             Tabla_De_Estado);
148 Ecuacion_De_Transicion_2: MUX_PLC generic map( k, T_DM ) --
    Selecciona a partir de la entrada actual el pr ximo estado.
149                                 Port    map( Direccion => x,
150                                          Dato      =>
                                             Salida_Mux,
151                                          Tabla_ROM =>
                                             Estados_Con_Formato
                                             );
152
153 Asignacion_Proximo_Estado: Proximo_Estado <= Salida_MUX( m - 1
    downto 0 );  -- ajusta de N_Bits_Dato bits del dato del MUX a
     m bits del estado.
154
```

```
155 --
156 -- Ecuaci n de salida estilo Moore:
157 -- Descomentar en caso de Moore comentar en caso de Mealy:
158
159
160
161
162 Ecuacion_De_Salida_Moore: MUX_PLC generic map( m, T_DM ) --
       Selecciona a partir del estado actual la salida.
163                                   Port    map( Direccion =>
                                         Estado_Actual,
164                                              Dato      =>
                                                 Salida,
165                                              Tabla_ROM =>
                                                 Tabla_De_Salida
                                                 );
166 Salida_De_Moore: y <= Salida( p - 1 downto 0 );
167
168
169 --
170 --Fin de Descomentar en caso de Moore comentar en caso de Mealy
171 --
172
173 --
174 -- Ecuaci n de salida estilo Mealy:
175 -- Descomentar en caso de Mealy comentar en caso de Moore:
176 --
177
178 -- ajusta a la derecha con ceros de relleno, p bits de la salida
        a N_Bits_Dato bits del dato del MUX.
179
180
181 --Asignacion_MUX_Mealy : process( Dato_Intermedio_2 )
182 --    begin
183 --        for i in 0 to 2**k - 1 loop
184 --            Salidas_Con_Formato( i ) <= std_logic_vector(
       resize( unsigned( Dato_Intermedio_2( p * ( i + 1 ) - 1 downto
        p * i ) ), N_Bits_Dato ) );
185 --        end loop;
186 --    end process Asignacion_MUX_Mealy;
187
188 --Ecuacion_De_Salida_1_Mealy: MUX_PLC generic map( m, T_DM ) --
       Selecciona a partir del estado actual todas las posibles
       salidas.
189 --                                    Port    map( Direccion =>
       Estado_Actual,
190 --                                                 Dato      =>
       Dato_Intermedio_2,
```

```
--                                                  Tabla_ROM =>
   Tabla_De_Salida);
--Ecuacion_De_Salida_2_Mealy: MUX_PLC generic map( k, T_DM ) --
   Selecciona a partir de la entrada actual la salida.
--                                  Port    map( Direccion => x,
--                                               Dato      =>
   Salida,
--                                               Tabla_ROM =>
   Salidas_Con_Formato);
--Salida_De_Mealy: y <= Salida( p - 1 downto 0 );



--
--Fin de Descomentar en caso de Mealy comentar en caso de Moore
--



end Estructura;
```

7.1.3. PLC's

PLC:Contador

```


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;
use work.Tipos_FSM_PLC.all;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity PLC is
    generic(   k    : natural := 32;    -- k entradas.
               p    : natural := 32;    -- p salidas.
               m    : natural := 32);   --2^m estados
    Port (

        x:in std_logic_vector(k-1 downto 0);
        y:out std_logic_vector(p-1 downto 0);
```

```
25          Trigger : in STD_LOGIC;
26          clk     : in STD_LOGIC;
27          cke     : in STD_LOGIC;
28          reset   : in std_logic);
29
30  --        sensores:in std_logic_vector(1 downto 0);
31  --        modo: in STD_LOGIC;
32  --        motor : out std_logic_vector(2 downto 0);
33  --        sw: in std_logic_vector(2 downto 0));
34
35  end PLC;
36
37  architecture Comportamiento of PLC is
38
39
40  --se ales intermedias
41
42
43
44
45
46      component FSM_PLC is
47
48          generic(k     : natural := 32;     -- k entradas.
49                  p     : natural := 32;     -- p salidas.
50                  m     : natural := 32;     -- m biestables. (Hasta
                        16 estados)
51                  T_DM : time     := 10 ps; -- Tiempo de retardo
                        desde el cambio de direcci n del MUX hasta
                        la actualizaci n de la salida Q.
52                  T_D  : time     := 10 ps; -- Tiempo de retardo
                        desde el flanco activo del reloj hasta la
                        actualizaci n de la salida Q.
53                  T_SU : time     := 10 ps; -- Tiempo de Setup.
54                  T_H  : time     := 10 ps; -- Tiempo de Hold.
55                  T_W  : time     := 10 ps); -- Anchura de pulso.
56          port    (   x : in  STD_LOGIC_VECTOR( k - 1 downto 0 );
                      -- x es el bus de entrada.
57                y : out STD_LOGIC_VECTOR( p - 1 downto 0 );     --
                      y es el bus de salida.
58                Tabla_De_Estado : in Tabla_FSM( 0 to 2**m - 1 );
                      -- Contiene la Tabla de Estado estilo Moore: Z(n
                      +1)=T1(Z(n),x(n))
59                Tabla_De_Salida : in Tabla_FSM( 0 to 2**m - 1 );
                      -- Contiene la Tabla de Salida estilo Moore: Y(n
                        )=T2(Z(n))
60                clk     : in STD_LOGIC;   -- La se al de reloj.
61                cke     : in STD_LOGIC;   -- La se al de
```

```
                habilitaci n de avance: si vale '1' el
                aut mata avanza a ritmo de clk y si vale '0'
                manda Trigger.
62              reset    : in STD_LOGIC;   -- La se al de
                inicializaci n.
63              Trigger : in STD_LOGIC ); -- La se al de disparo (
                single shot) as ncrono y pos blemente con
                rebotes para hacer un avance  nico . Ha de
                llevar un sincronizador.
64      end component FSM_PLC;
65
66
67
68      constant TablaE :Tabla_FSM(0 to 2**m-1):=
69      (x"00000001",
70                                                          x"00000021"
                                                              ,
71                                                          x"00000023"
                                                              ,
72                                                          x"00000043"
                                                              ,
73                                                          x"00000045"
                                                              ,
74                                                          x"00000065"
                                                              ,
75                                                          x"00000066"
                                                              ,
76                                                          (others =>
                                                              '0'),
77                                                          (others =>
                                                              '0'),
78                                                          (others =>
                                                              '0'),
79                                                          (others =>
                                                              '0'),
80                                                          (others =>
                                                              '0'),
81                                                          (others =>
                                                              '0'),
82                                                          (others =>
                                                              '0'),
83                                                          (others =>
                                                              '0'),
84                                                          (others =>
                                                              '0'));
85
86      constant TablaS :Tabla_FSM(0 to 2**m-1):=
87  (x"00000000",
```

```
                                                    x"00000011"
                                                       ,
                                                    x"00000011"
                                                       ,
                                                    x"00000022"
                                                       ,
                                                    x"00000022"
                                                       ,
                                                    x"00000033"
                                                       ,
                                                    x"00000033"
                                                       ,
                                                    (others =>
                                                       '0'),
                                                    (others =>
                                                       '0'),
                                                    (others =>
                                                       '0'),
                                                    (others =>
                                                       '0'),
                                                    (others =>
                                                       '0'),
                                                    (others =>
                                                       '0'),
                                                    (others =>
                                                       '0'),
                                                    (others =>
                                                       '0'),
                                                    (others =>
                                                       '0'));


begin


    FSM:FSM_PLC
        generic map(k=>k,p=>p,m=>m)
        port  map(x=>x,y=>y,Tabla_de_estado=>TablaE,
           Tabla_de_Salida=>TablaS,clk=>clk,cke=>'1',reset=>
           reset,Trigger=>Trigger);



end Comportamiento;
```

PLC:Sentido

```

```

```
3
4  library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;
6
7  -- Uncomment the following library declaration if using
8  -- arithmetic functions with Signed or Unsigned values
9  use IEEE.NUMERIC_STD.ALL;
10 use work.Tipos_FSM_PLC.all;
11
12 -- Uncomment the following library declaration if instantiating
13 -- any Xilinx leaf cells in this code.
14 --library UNISIM;
15 --use UNISIM.VComponents.all;
16
17 entity SentidoPLC is
18     generic(   k    : natural := 32;    -- k entradas.
19             p    : natural := 32;    -- p salidas.
20             m    : natural := 32);   --2^m estados
21     Port (
22
23         x:in std_logic_vector(k-1 downto 0);
24         y:out std_logic_vector(p-1 downto 0);
25         Trigger : in STD_LOGIC;
26         clk     : in STD_LOGIC;
27         cke     : in STD_LOGIC;
28         reset1   : in std_logic);
29
30     --         sensores:in std_logic_vector(1 downto 0);
31     --         modo: in STD_LOGIC;
32     --         motor : out std_logic_vector(2 downto 0);
33     --         sw: in std_logic_vector(2 downto 0));
34
35 end SentidoPLC;
36
37 architecture Comportamiento of SentidoPLC is
38
39
40     --se ales intermedias
41
42
43
44
45
46     component FSM_PLC is
47
48         generic(k    : natural := 32;    -- k entradas.
49                 p    : natural := 32;    -- p salidas.
50                 m    : natural := 32;    -- m biestables. (Hasta
```

```
                    16 estados)
51                T_DM : time    := 10 ps; -- Tiempo de retardo
                     desde el cambio de direcci n del MUX hasta
                     la actualizaci n de la salida Q.
52                T_D  : time    := 10 ps; -- Tiempo de retardo
                     desde el flanco activo del reloj hasta la
                     actualizaci n de la salida Q.
53                T_SU : time    := 10 ps; -- Tiempo de Setup.
54                T_H  : time    := 10 ps; -- Tiempo de Hold.
55                T_W  : time    := 10 ps); -- Anchura de pulso.
56
57          port   (   x : in  STD_LOGIC_VECTOR( k - 1 downto 0 );
                    -- x es el bus de entrada.
58                y : out STD_LOGIC_VECTOR( p - 1 downto 0 );     --
                     y es el bus de salida.
59                Tabla_De_Estado : in Tabla_FSM( 0 to 2**m - 1 );
                     -- Contiene la Tabla de Estado estilo Moore: Z(n
                     +1)=T1(Z(n),x(n))
60                Tabla_De_Salida : in Tabla_FSM( 0 to 2**m - 1 );
                     -- Contiene la Tabla de Salida estilo Moore: Y(n
                       )=T2(Z(n))
61                clk     : in STD_LOGIC;   -- La se al de reloj.
62                cke     : in STD_LOGIC;   -- La se al de
                     habilitaci n de avance: si vale '1' el
                     aut mata avanza a ritmo de clk y si vale '0'
                     manda Trigger.
63                reset : in STD_LOGIC;   -- La se al de
                     inicializaci n.
64                Trigger : in STD_LOGIC ); -- La se al de disparo (
                     single shot) as ncrono y pos blemente con
                     rebotes para hacer un avance  nico . Ha de
                     llevar un sincronizador.
65      end component FSM_PLC;
66
67
68
69      constant TablaE :Tabla_FSM(0 to 2**m-1):=
70    (x"00000310",
71                                                        x"00002111"
                                                             ,
72                                                        x"00002310"
                                                             ,
73                                                        x"00004333"
                                                             ,
74                                                        x"00004310"
                                                             ,
75                                                        (others =>
                                                             '0'),
```

```
76                                                    (others =>
                                                        '0'),
77                                                    (others =>
                                                        '0'),
78                                                    (others =>
                                                        '0'),
79                                                    (others =>
                                                        '0'),
80                                                    (others =>
                                                        '0'),
81                                                    (others =>
                                                        '0'),
82                                                    (others =>
                                                        '0'),
83                                                    (others =>
                                                        '0'),
84                                                    (others =>
                                                        '0'),
85                                                    (others =>
                                                        '0'));
86
87      constant TablaS :Tabla_FSM(0 to 2**m-1):=
88    (x"00000000",
89                                                    x"00002222"
                                                        ,
90                                                    x"00002222"
                                                        ,
91                                                    x"00001111"
                                                        ,
92                                                    x"00001111"
                                                        ,
93                                                    (others =>
                                                        '0'),
94                                                    (others =>
                                                        '0'),
95                                                    (others =>
                                                        '0'),
96                                                    (others =>
                                                        '0'),
97                                                    (others =>
                                                        '0'),
98                                                    (others =>
                                                        '0'),
99                                                    (others =>
                                                        '0'),
100                                                   (others =>
                                                        '0'),
101                                                   (others =>
```

```
                                                          '0'),
102                                                      (others =>
                                                          '0'),
103                                                      (others =>
                                                          '0'));
104
105 begin
106
107     FSM:FSM_PLC
108         generic map(k=>k,p=>p,m=>m)
109         port  map(x=>x,y=>y,Tabla_de_estado=>TablaE,
               Tabla_de_Salida=>TablaS,clk=>clk,cke=>'1',reset=>
               reset1,Trigger=>Trigger);
110
111
112
113 end Comportamiento;
```

7.1.4. Unidad de Control

```
1
2
3  library IEEE;
4  use IEEE.STD_LOGIC_1164.ALL;
5
6  -- Uncomment the following library declaration if using
7  -- arithmetic functions with Signed or Unsigned values
8  use IEEE.NUMERIC_STD.ALL;
9
10 -- Uncomment the following library declaration if instantiating
11 -- any Xilinx leaf cells in this code.
12 --library UNISIM;
13 --use UNISIM.VComponents.all;
14
15 entity Control_Unit is
16     generic(   k    : natural := 32;     -- k entradas.
17             p    : natural := 32;     -- p salidas.
18             m    : natural := 32);    --2^m estados
19     Port(
20
21
22         Trigger : in STD_LOGIC;--se utiliza para activar pulsos
               de se al,pero el reloj siempre esta activo
23         clk     : in STD_LOGIC;--Reloj establecido a un Periodo
               de 8ns (125MHz)
24         --cke     : in STD_LOGIC; --  el reloj siempre esta
               activo
```

```
25          sensor :in std_logic_vector(1 downto 0);--2 sensores
               fotoc lula de la mesa
26          motor: out std_logic_vector(2 downto 0);--se ales del
               variador de frecuencia para el motor [0]bloqueo,[1]
               movimiento,[2]sentido
27          modo: in std_logic;--switch para seleccionar entre modo
               manual o el dise o FSM
28          sw:in std_logic_vector(2 downto 0);--swtiches para
               manejar la mesa de forma manual
29
30          E_Stop:in std_logic;--Boton de Parada de Emergencia
31
32          --leds de la Zybo
33          led1:out std_logic;
34          led2:out std_logic;
35          led3:out std_logic;
36          led4:out std_logic;
37
38          --Pulsador del reset
39          resetP    : in std_logic);
40
41
42  end Control_Unit;
43
44
45  architecture Behavioral of Control_Unit is
46      --entradas y salidas de las FSM PLC y SentidoPLC
47      signal x1: std_logic_vector(k-1 downto 0);
48      signal x2:std_logic_vector(k-1 downto 0);
49      signal y1: std_logic_vector(p-1 downto 0);
50      signal y2: std_logic_vector(p-1 downto 0);
51
52      --resets necesarios para el sistema
53      signal reset2:std_logic;
54      signal reset:std_logic;
55      signal Pos_resetP :  STD_LOGIC;
56
57      --Signals para la Parada de Emergencia
58      signal Pos_E_Stop:std_logic;
59      signal Parada:std_logic;
60
61      --cuentas del numero de pieza
62      signal cuenta1:integer:=0;
63      signal cuenta2:integer:=0;
64
65      --contador de pulsos de reloj para el Tiempo
66      signal counter : integer:=0;
67      signal reset_counter:natural:=0;
```

```
68
69      component Sincronizador is -- Genera el CKE libre de rebotes
            y de duracion 1 ciclo de reloj clk.
70          generic ( n      : natural := 4 );
71          Port     ( I      : in STD_LOGIC;
72               CKE    : out STD_LOGIC;
73               reset  : in STD_LOGIC;
74               clk    : in STD_LOGIC);
75      end component Sincronizador;
76
77      --FSM con el Diagrama de estados para contar las Piezas
78      component PLC is
79          generic(   k     : natural := 32;     -- k entradas.
80                   p     : natural := 32;     -- p salidas.
81                   m     : natural := 32);    --2^m estados
82          Port (
83              x:in std_logic_vector(k-1 downto 0);
84              y:out std_logic_vector(p-1 downto 0);
85              Trigger : in STD_LOGIC;
86              clk     : in STD_LOGIC;
87              cke     : in STD_LOGIC;
88              reset   : in std_logic);
89
90      end component PLC;
91
92      --FSM para asignar el sentido mediante los sensores
93      component SentidoPLC
94          generic(   k     : natural := 32;     -- k entradas.
95                   p     : natural := 32;     -- p salidas.
96                   m     : natural := 32);    --2^m estados
97          Port (
98
99              x:in std_logic_vector(k-1 downto 0);
100             y:out std_logic_vector(p-1 downto 0);
101             Trigger : in STD_LOGIC;
102             clk     : in STD_LOGIC;
103             cke     : in STD_LOGIC;
104             reset1   : in std_logic);
105     end component SentidoPLC;
106
107 begin
108     --Asignacion de los PLC
109     Contador:PLC
110         generic map(k=>2,p=>2,m=>4)
111         port  map(x=>x1,y=>y1,Trigger=>Trigger,clk=>clk,cke=>'1'
                ,reset=>reset);
112
113     Sentido:SentidoPLC
```

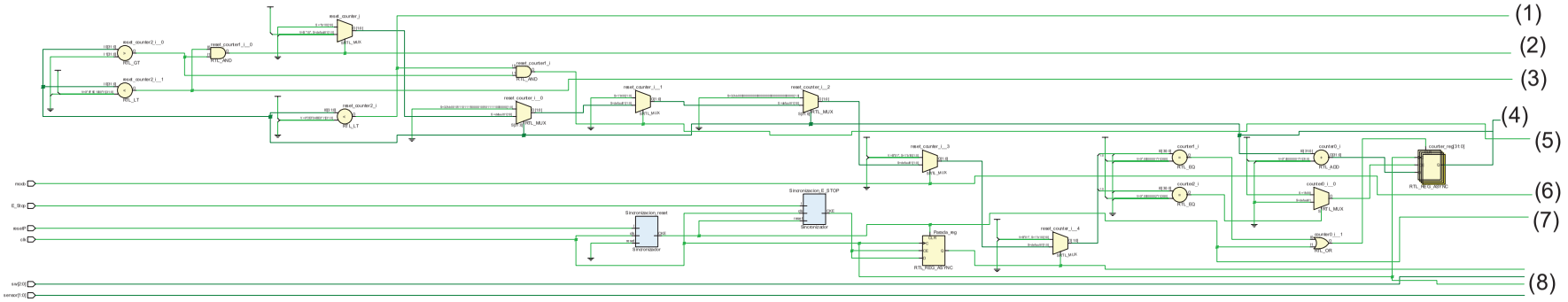
```
114         generic map(k=>2,p=>2,m=>4)
115         port map(x=>x2,y=>y2,Trigger=>Trigger,clk=>clk,cke=>'1',
               reset1=>Pos_resetP);
116
117
118     --Asignaci n Sincronizadores para evitar rebotes
119     Sincronizacion_reset: Sincronizador
120         generic map( 4 ) -- El sincronizador tiene 32 etapas en
               el filtro FIR (Finite Impulse Response).
121         port    map( I     => resetP,
122                  CKE   => Pos_resetP,
123                  reset => '0',
124                  clk   => clk );
125
126     Sincronizacion_E_STOP: Sincronizador
127         generic map( 4 ) -- El sincronizador tiene 32 etapas en
               el filtro FIR (Finite Impulse Response).
128         port    map( I     => E_Stop,
129                  CKE   => Pos_E_Stop,
130                  reset => Pos_resetP,
131                  clk   => clk );
132
133
134
135
136     x2<=sensor;
137
138     --la Entrada de la FSM Contador de Piezas es una Puerta or
           de los sensores
139     x1(0)<=sensor(0) and sensor(1);
140
141
142     --En caso de accionar la Parada de Emergencia se
143     --para la mesa hasta la activacion del reset
144     Asignacion_E_STOP:process
145     begin
146         if Pos_resetP='1' then
147             Parada<='0';
148         elsif rising_edge(clk) and Pos_E_Stop='1'  then
149
150             Parada<='1';
151
152         else
153             Parada<=Parada;
154         end if;
155
156     end process;
157
```

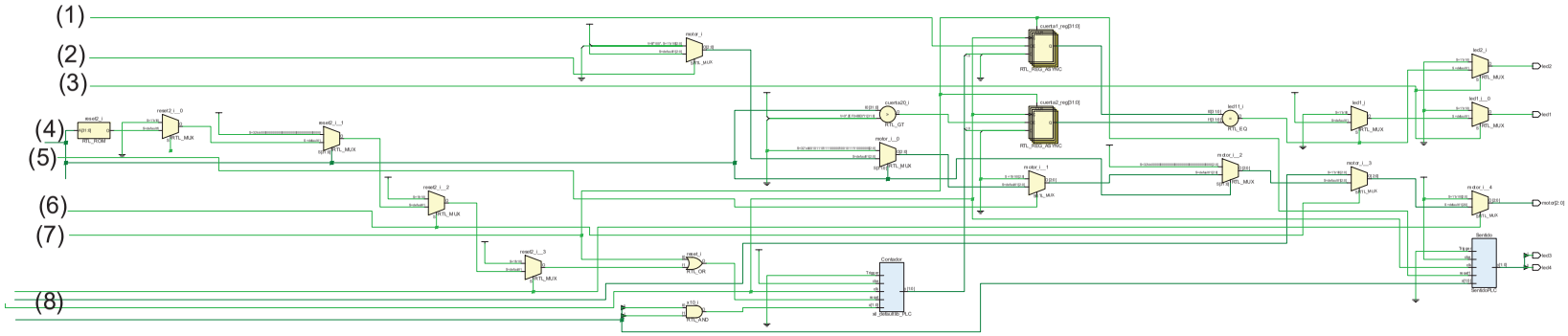
```
158
159
160     --Proceso para contar el Tiempo Transcurrido
161     Secuencial:process(clk,reset_counter)
162     begin
163         if reset_counter=1 or Pos_ResetP='1' then
164
165             counter<=0;
166
167         elsif reset_counter=2 then
168
169             counter<=counter;
170
171         elsif rising_edge(clk) then
172
173             counter<=counter+1;
174
175         else
176
177             counter<=counter;
178
179         end if;
180     end process Secuencial;
181
182
183     --Proceso Principal donde se asigna entre los 2 modos
184     Seleccion:process
185
186     begin
187
188         if Parada='1' then
189             motor<="111";
190             reset_counter<=1;
191             reset2<='1';
192         else
193             if modo='1' then    --Modo Manual
194                 motor<=sw;
195                 reset_counter<=1;
196                 reset2<='1';
197             else      --Modo automatico
198
199                 if counter=0 then
200                     motor<="111";
201                     reset_counter<=0;
202                     reset2<='1';
203
204                 elsif counter<6300*125000 and counter>0  then
205                     motor<="000";
```

```
206                     reset_counter<=0;
207                     reset2<='0';
208 
209                 elsif counter=6300*125000 then
210                     motor<="000";
211                     reset_counter<=0;
212                     reset2<='1';
213 
214 
215                 elsif counter<2*6400*125000 and counter>0  then
216                     motor<="100";
217                     reset_counter<=0;
218                     reset2<='0';
219                 else
220                     motor<="111";
221                     reset_counter<=2;
222                     reset2<='0';
223                 end if;
224 
225             end if;
226         end if;
227     end process Seleccion;
228 
229     --Para las cuentas es necesario el uso de biestables para
        mantener sus valores
230     --cuenta1 es en sentido horario,Cuenta2 en sentido
        antihorario
231     Asignacion_Cuenta1:process
232     begin
233         if Pos_resetP='1' then
234             cuenta1<=0;
235         elsif rising_edge(clk) and counter<6300*125000  then
236 
237             cuenta1<=to_integer(unsigned(y1));
238         else
239             cuenta1<=cuenta1;
240         end if;
241 
242     end process;
243 
244 
245     Asignacion_Cuenta2:process
246     begin
247         if Pos_resetP='1' then
248             cuenta2<=0;
249         elsif rising_edge(clk) and counter>6300*125000  then
250 
251             cuenta2<=to_integer(unsigned(y1));
```

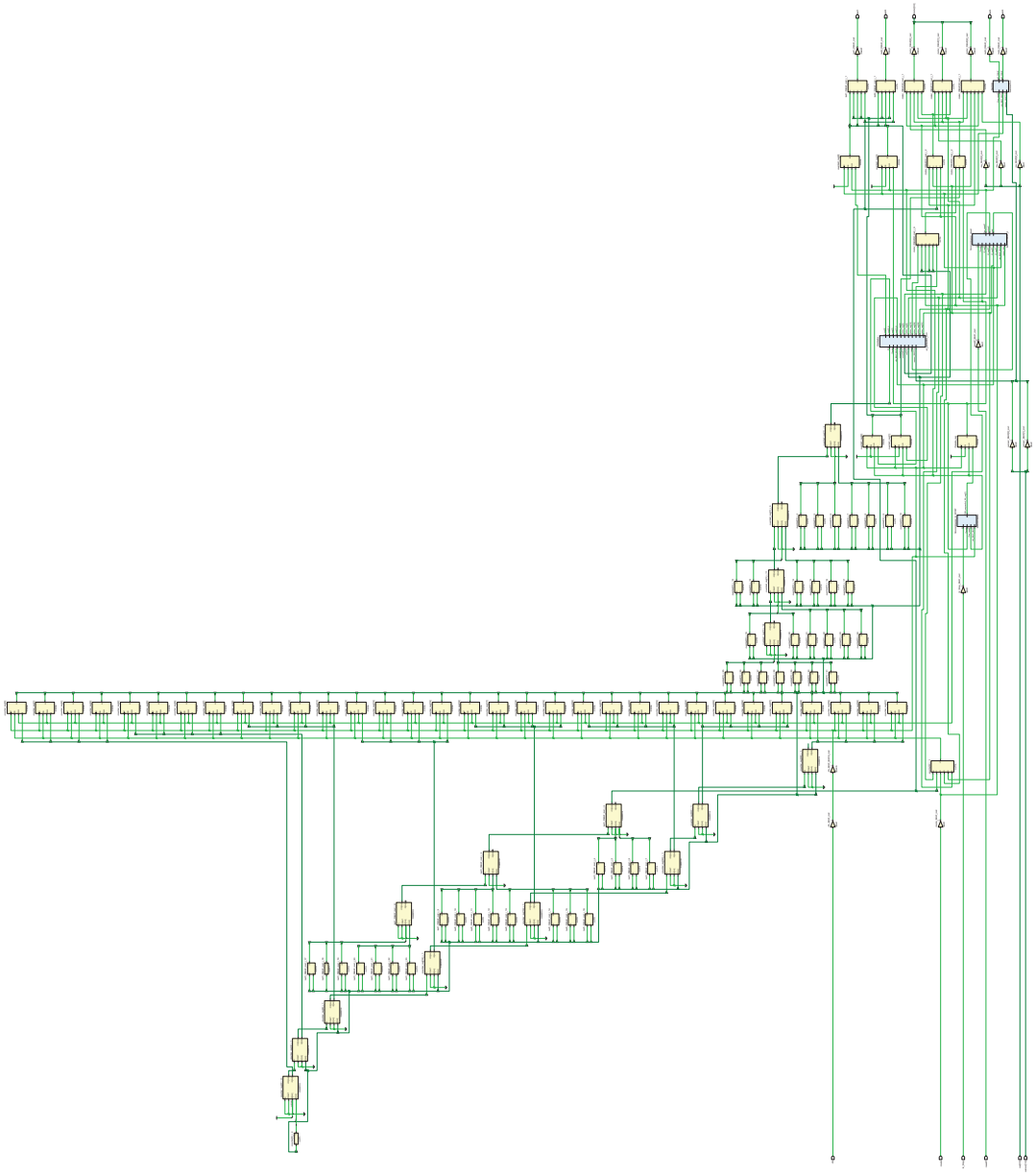
```
252         else
253             cuenta2<=cuenta2;
254         end if;
255 
256     end process;
257 
258 
259     --Se verifica la cuenta tras el proceso del aut mata
260     --Si la cuenta es correcta se enciende el led2,en caso
        contrario el led1
261     Verificacion_Cuenta: process(counter)
262     begin
263         if counter<2*6400*125000 then
264 
265             led1<='0';
266             led2<='0';
267         else
268 
269             if cuenta1=cuenta2 then
270 
271                 led1<='0';
272                 led2<='1';
273 
274             else
275                 led1<='1';
276                 led2<='0';
277             end if;
278         end if;
279     end process  Verificacion_Cuenta;
280 
281 
282 
283 
284     --Aignacion de leds para el sentido
285     led3<=y2(1);
286     led4<=y2(0);
287 
288 
289     --El reset del PLC se activar  tanto si se pulsa de manera
        manual,
290     --como si es necesario reiniciar el diagrama de Estados
291     reset<=Pos_resetP or reset2;
292 
293 
294 
295 end Behavioral;
```

7.2. Esquemas





UNIVERSIDAD DE MÁLAGA	
ESCUELA DE INGENIERÍAS INDUSTRIALES	TÍTULO: DISEÑO DE UN PLC CON LÓGICA PROGRAMABLE PARA CONTROL INDUSTRIAL
AUTOR: MARCOS GUERRERO LUQUE	
PLANO Nº: 1	ESQUEMA RTL
TITULACIÓN: GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES	FECHA: AGOSTO/2023
FIRMA: Fdo.: MARCOS GUERRERO LUQUE	



UNIVERSIDAD DE MÁLAGA

ESCUELA DE INGENIERÍAS INDUSTRIALES

TÍTULO:
DISEÑO DE UN PLC CON LÓGICA PROGRAMABLE PARA CONTROL INDUSTRIAL

AUTOR: MARCOS GUERRERO LUQUE

PLANO Nº: 2

ESQUEMA SINTETIZADO

TITULACIÓN: GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

FECHA: AGOSTO/2023

FIRMA:

Fdo.:MARCOS GUERRERO LUQUE