INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

ESCOM

Trabajo Terminal

"Sistema Embebido para Procesamiento Digital de Señales (SEPRODIS)"

2013-A045

Presentan

Chacon Arenas Luis Antonio

Directores

M. en C. Víctor Hugo García Ortega ***Dr. Julio César Sosa Savedra***

Mayo de 2014

INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
SUBDIRECCIÓN ACADÉMICA

No. de TT: 2013-A045 28 de Mayo de 2014

Documento Técnico

"Sistema Embebido para Procesamiento Digital de Señales (SEPRODIS)"

Presentan

Chacon Arenas Luis Antonio[1]

Directores

M. en C. Víctor Hugo García Ortega ***Dr. Julio César Sosa Savedra***

RESUMEN

En este trabajo de tesis se presenta un Sistema Embebido que permite el procesamiento de señales electrocardiográficas (ECG). Dichas señales son tomadas de la base de datos QT, proporcionadas por el Instituto Tecnológico de Massachuset (MIT). Las señales ECG pueden ser analizadas y consultadas desde un servidor en forma remota mediante internet. Para la implementación del Sistema Embebido se usa un FPGA (Field Programmable Gate Array), donde se diseñó una arquitectura específica usando recursos dedicados para el aumento de la frecuencia de reloj y un funcionamiento eficiente del sistema. Esta arquitectura específica se usa para detectar el complejo QRS mediante el algoritmo de correlación cruzada. Además, se utiliza el procesador soft-core Microblaze de Xilinx para la gestión de la arquitectura junto con los periféricos que realizan la comunicación remota.

Palabras clave: Señal ECG, Soft-core, FPGA, Procesamiento de señales digitales, Sistemas Embebidos.

1 **chacon12345@hotmail.com**

ESCUELA SUPERIOR DE CÓMPUTO
SUBDIRECCIÓN ACADÉMICA

DEPARTAMENTO DE FORMACIÓN INTEGRAL E INSTITUCIONAL

COMISIÓN ACADÉMICA DE TRABAJO TERMINAL

México, D.F. a 28 de mayo de 2014.

DR. FLAVIO ARTURO SÁNCHEZ GARFIAS
PRESIDENTE DE LA COMISIÓN ACADÉMICA
DE TRABAJO TERMINAL
P R E S E N T E

Por medio del presente, se informa que el alumno que integra el **TRABAJO TERMINAL**: 2013-A045, titulado "**Sistema Embebido para Procesamiento Digital de Señales (SEPRODIS)**" concluyó satisfactoriamente su trabajo.

Los discos (DVDs) fueron revisados ampliamente por sus servidores y corregidos, cubriendo el alcance y el objetivo planteados en el protocolo original y de acuerdo a los requisitos establecidos por la Comisión que Usted preside.

ATENTAMENTE

M. en C. Víctor Hugo García Ortega

Dr. Julio César Sosa Savedra

Advertencia

"Este documento contiene información desarrollada por la Escuela Superior de Cómputo del Instituto Politécnico Nacional, a partir de datos y documentos con derecho de propiedad y por lo tanto, su uso quedará restringido a las aplicaciones que explícitamente se convengan."

La aplicación no convenida exime a la escuela su responsabilidad técnica y da lugar a las consecuencias legales que para tal efecto se determinen.

Información adicional sobre este reporte técnico podrá obtenerse en:

La Subdirección Académica de la Escuela Superior de Cómputo del Instituto Politécnico Nacional, situada en Av. Juan de Dios Bátiz s/n Teléfono: 57296000, extensión 52000.

Agradecimientos

Al profesor Víctor Hugo García Ortega, por apoyarme ampliamente en el desarrollo de este Trabajo Terminal, por la paciencia que me brindó en las complicaciones que se presentarón a lo largo del desarrollo, por tomar en cuenta mis opiniones acerca de varios aspectos y detalles presentes en cada una de las etapas, por la confianza y la credibilidad de que el trabajo se terminaría obteniendo los resultados esperados, por todo el conocimiento tanto técnico como pedagógico que tuve la oportunidad de aprender en los dos años y medio que fue mi profesor en el salón de clases.

Al profesor Julio César Sosa Savedra, por brindarme sus conocimientos y experiencia en áreas nuevas para mí, por sus observaciones y correcciones que ayudaron a crear un mejor proyecto, por sus ideas que complementaban el trabajo físico en el sistema, por sus recomendaciones para desarrollar de mejor manera el Trabajo Terminal.

Al profesor Miguel Olvera Aldana, a quien le guardo un profundo respeto y admiración , ya que no sólo se limitó a ser mi profesor en el aula y mi tutor fuera de ésta, sino que me brindó su apoyo siempre que lo necesitaba, desde que comencé mi camino en esta escuela llamada ESCOM. La confianza en mis capacidades y la casi infinita paciencia que tuvo siempre hacia mí, fueron los elementos clave que me hicieron ser no sólo un mejor estudiante, sino una mejor persona, que ahora puedes sentirse orgulloso y muy agradecido de haber sido su alumno. Por acompañarme todo el camino, de principio a fin, durante el cual pude aprender de sus consejos, sus experiencias, sus habilidades, inclusive de sus regaños, que me ganaba a veces. Por todo lo que compartió conmigo, es que yo lo considero mi segundo padre.

A mi papá, mi mamá, mi abuela y mi hermano menor, todos ellos parte de mi familia, que me dio todos los recursos para poder prepararme y convertirme en un profesionista, dándome ánimos algunas veces, creyendo en mí, convenciéndome de que lo lograría si me esforzaba lo suficiente. He aquí el resultado.

A mi gran amigo y hermano Edgar (Edgar-san), que conocí en el segundo semestre, con el que compartí varias clases y siempre tratando de apoyarnos uno al otro, tanto en las clases como cuando se terminaban, ayudándonos inclusive cuando se nos presentaban problemas fuera de la vida en la escuela.

A mi otro gran amigo y hermano Sergio (Serch), que conocí ya casi al final de la carrera, pero que en un tiempo muy corto, logramos estrechar nuestra amistad, teniendo como principal motivador en común, terminar nuestros respectivos Trabajos Terminales de la mejor manera posible.

A Diana Karen (Karen Diana), a quien también conoci ya en la recta final de la carrera en ESCOM, por ayudarme a reir un poco en esos días que me estresaba el no poder resolver algún problema del Trabajo Terminal.

A mis demás compañeros y amigos Gandhi, Miguel, Hugo M, Hugo A (Ramirez[2]), Hugo E (Quic), Roa, Cinthya, Karen, Iván, Aura, Joshio, yo todos aquellos que, aunque no los mencioné, no dejan de ser igualmente importantes.

A todos los profesores que tuve en cada una de las Unidades de Aprendizaje, ya que siempre trataron de hacer su mejor esfuerzo para lograr trasmitirnos su conocimiento y a su vez, colaborar en nuestra formación como profesionales en el área de las ciencias de la computación.

Al Instituto Politécnico Nacional, por darme las bases desde mi formación en el nivel medio superior, momento en el cual formé parte del mismo. Asi mismo también un profundo agradecimiento a la Escuela Superior de Cómputo, la cual siembre está dispuesta a brindar la mejor formación a cada uno de los estudiantes que logran ingresar, desde su inicio, hasta su salida como ingenieros.

Luis Antonio Chacon Arenas

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1
Introducción

Desde sus orígenes, la humanidad ha padecido numerosas enfermedades que han provocado incontables muertes de seres humanos. Las causas por las que se desarrollan las enfermedades son muy variadas y muy diversas, pero todas ellas afectando alguna parte del cuerpo. Cada enfermedad afecta el organismo de manera distinta provocando una serie de síntomas. Algunas veces, los síntomas son detectables a simple vista, lo que facilita su diagnóstico. Sin embargo, en otras ocasiones, los síntomas son muy sutiles y no es posible identificarlos de manera sencilla.

Para facilitar el diagnostico de las enfermedades, el hombre se ha apoyado de instrumentos que le permitan conocer el estado físico de una persona. Cada uno de estos instrumentos está íntimamente ligado con la tecnología existente en ese momento. La relación entre ambos hace notar que los instrumentos tienden a mejorar y ser más sofisticados con el desarrollo de nuevas tecnologías que pueden ser adquiridas y utilizadas por casi cualquier persona.

Al mismo tiempo, a través de la historia, los componentes electrónicos han visto reducido su tamaño a niveles impresionantes. La reducción del tamaño ha permitido diseñar y construir sistemas de bajo costo y fáciles de transportar en casi cualquier campo de aplicación. Sumado a las pequeñas dimensiones de los componentes, también se ha incrementado la capacidad para realizar numerosas tareas dentro de un solo dispositivo. Cada nueva generación de componentes electrónicos abre nuevas posibilidades de diseño e implementación de sistemas.

Un área de la computación que aprovecha las ventajas que proporciona la tecnología en los componentes electrónicos, son los Sistemas Embebidos. Estos sistemas han cobrado fuerza gracias a su flexibilidad para diseñar e implementar sistemas que realicen tareas específicas con solamente los recursos necesarios. Las tareas en las que se puede hacer uso de un sistema embebido son múltiples como son adquisición de datos, procesamiento de información, envió y recepción de datos, entre otras.

Del gran número de elementos a considerar en la tecnología, dos de ellos son particularmente considerados en la construcción de instrumentos de apoyo médico: el tamaño y el costo. Un bajo costo permite adquirir un mayor número de unidades para atender a un mayor número de personas. Un menor tamaño implica menor dificultad en la movilidad del dispositivo en caso de ser necesario.

Para poder saber la condición física de un paciente, es necesario conocer la forma en la que está funcionando el cuerpo internamente. Cada función del cuerpo, es controlada mediante impulsos eléctricos, que también pueden llamarse señales. Existen varios tipos de señales en el cuerpo humano como son encefalográficas y electrocardiográficas por mencionar algunos ejemplos. Como cualquier tipo de señal, es posible medirlas y procesarlas para obtener información que ayude a determinar la condición de un paciente a cada momento.

La capacidad de los Sistemas Embebidos para adquirir y procesar información, permite crear sistemas que lleven a cabo esta tarea. El grado de complejidad del sistema está condicionado por las necesidades y requerimientos específicos, pudiendo realizar únicamente una sola tarea o sub-tareas derivadas de una principal.

1.1. Problemática

En México, existen muchas comunidades donde no se tiene acceso a recursos de salud, ya sea por falta de medios económicos o de infraestructura. Generalmente las comunidades apartadas sólo cuentan con una clínica de medicina general. En las clínicas de medicina general muchas veces no existen especialistas que puedan dar diagnósticos precisos sobre ciertas enfermedades cardiacas o detectar anomalías en pacientes con factores de riesgo. Asimismo, el traslado de pacientes a una clínica de especialidad, no es una alternativa factible debido a varios factores. Los factores más comunes son el tiempo de traslado entre el paciente y la clínica y el costo que puede llegar a generar. Esto casi siempre no se encuentra dentro de las posibilidades de las familias. Sin embargo, el creciente uso de las comunicaciones ha incrementado la posibilidad de que se cuente al menos con una conexión a internet.

De acuerdo con los datos estadísticos del Instituto Nacional de Estadística y Geografía (INEGI), la población registrada en el último censo realizado durante el año 2010, fue de 112,336,538 habitantes [1].

De la misma forma, dentro del censo del mismo año, se registró un total de 160,175 médicos, entre médicos generales, médicos especialistas y médicos en entrenamiento [2]. Sin embargo, la distribución de médicos a lo largo de todo el territorio nacional no es igual.

La gráfica de la figura 1.1 muestra la concentración de médicos en cada uno de los estados, con lo que se puede observar dónde se encuentra el mayor número de médicos [3]. Esto provoca que la disponibilidad de un médico o especialista esté condicionada a la región donde se encuentre el paciente que lo requiere.

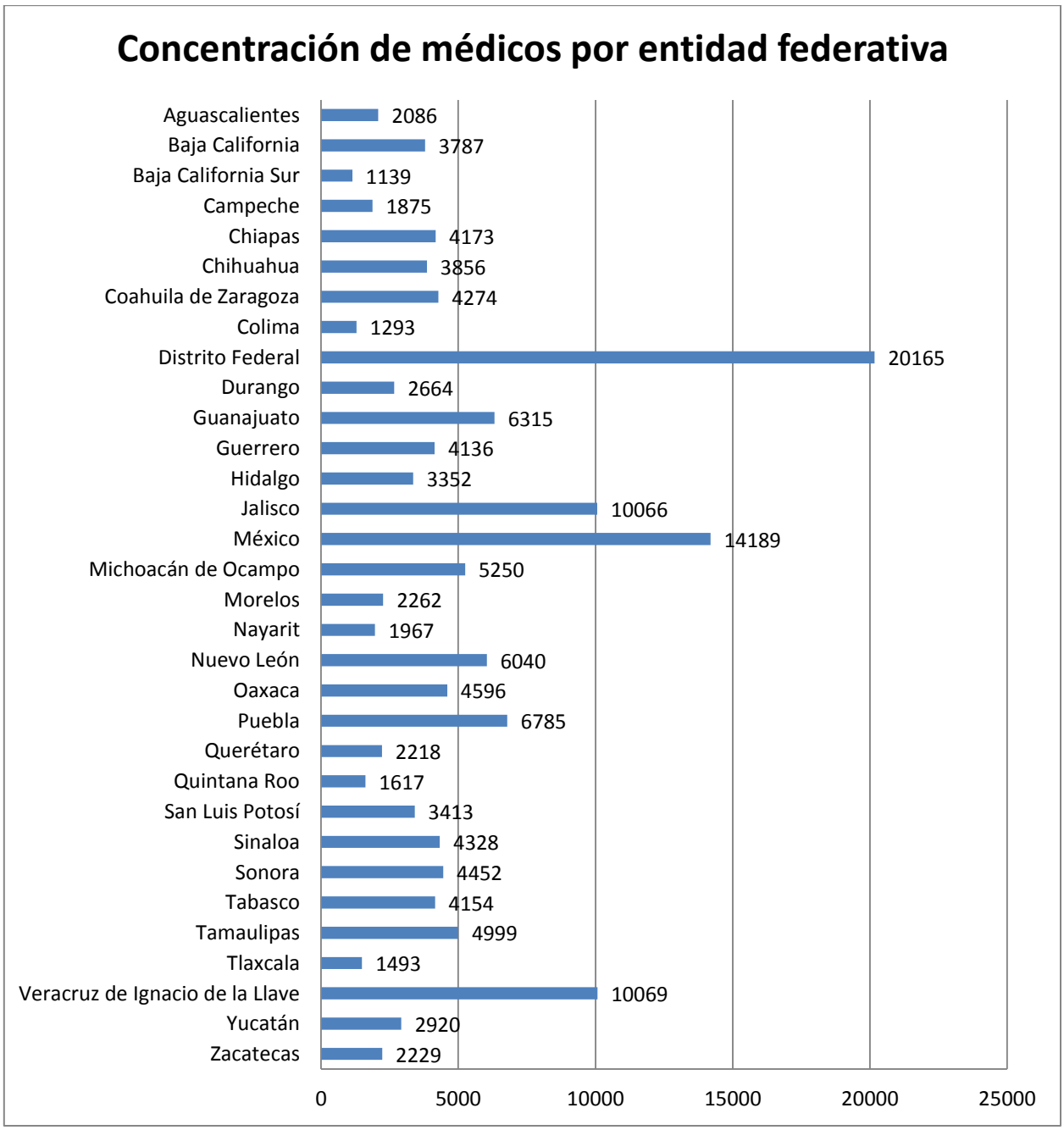


Figura 1.1 Gráfica de distribución de médicos en la República Mexicana.

De acuerdo con la gráfica anterior, el mayor número de médicos se registra en el Distrito Federal con 20,165 médicos, el Estado de México con 14,184 médicos, y los estados de Jalisco y Veracruz, teniendo 10,066 y 10,069 médicos respectivamente. Cabe resaltar que algunos son grandes ciudades con una gran concentración de habitantes, por lo que se cuenta con mayores recursos de salud, tanto en tecnología como en recursos humanos.

En cuanto a los recursos humanos disponibles, se puede hacer una relación entre el número total de habitantes junto con el número total de médicos para obtener el factor de correspondencia entre ambos como se muestra a continuación:

$$\frac{Número\ total\ de\ habitantes}{Número\ total\ de\ médicos} = \frac{112,336,538}{160{,}175} = 701.336 \approx 701$$

Este cálculo indica que a cada médico le corresponden aproximadamente 701 pacientes, lo que hace resaltar una falta de personal médico para poder atender de manera apropiada a toda la población que lo requiera.

Por el lado de la infraestructura, el factor de mayor peso, son la disponibilidad de instalaciones donde se puedan hacer pruebas y estudios a los pacientes para elaborar los diagnósticos correspondientes.

En el año 2009 se contabilizaron el número de laboratorios existentes en ese momento y de acuerdo con los datos estadísticos del INEGI de ese año, el número de laboratorios de análisis clínicos es de 2,499 [4]; lo que resulta insuficiente para atender a toda la población que necesite atención médica.

1.2. Propuesta de solución

Se diseñó un Sistema Embebido que permite analizar señales ECG de una base de datos de pacientes reales para identificar los complejos QRS presentes en la señal y mandar dichos complejos almacenados desde un servidor remoto por medio de internet. En el lugar donde se envía la información se encuentra un especialista que pueda examinar los datos enviados. Con la información proporcionada se puede realizar un diagnóstico más preciso y determinar el mejor tratamiento acorde a la enfermedad del paciente y al juicio del especialista.

Para la implementación del Sistema Embebido se usó un FPGA, ya que este dispositivo soporta el uso de microprocesadores de tipo soft-core como Microblaze. Así mismo cuenta con diversos recursos dedicados que permiten implementar una arquitectura eficiente y de gran velocidad de reloj, lo que reduce el tamaño del sistema al implementarlo dentro del FPGA.

También el uso de un FPGA permite la integración de diferentes módulos en un solo dispositivo reprogramable, lo que facilita la escalabilidad del sistema.

1.3. Justificación

El análisis y diseño del Sistema Embebido se enfoca en el procesamiento de señales del corazón, ya que la implementación de un sistema de este tipo, puede ayudar a combatir uno de los grandes problemas de salud que están presentes en el territorio mexicano.

De acuerdo con el censo realizado por parte del INEGI, en el año 2012 las enfermedades del corazón ocuparon el primer lugar entre las causas de muerte de la población en México [5].

En la figura 1.2 se muestra una gráfica donde se puede apreciar la parte proporcional que le corresponde a cada enfermedad registrada. La información del censo incluye las muertes tanto por enfermedades como por accidentes. Sin embargo, se excluyen las causas de muerte accidentales, ya que estas causas contienen factores que no son medibles y son ajenos a padecimientos o patologías físicas.

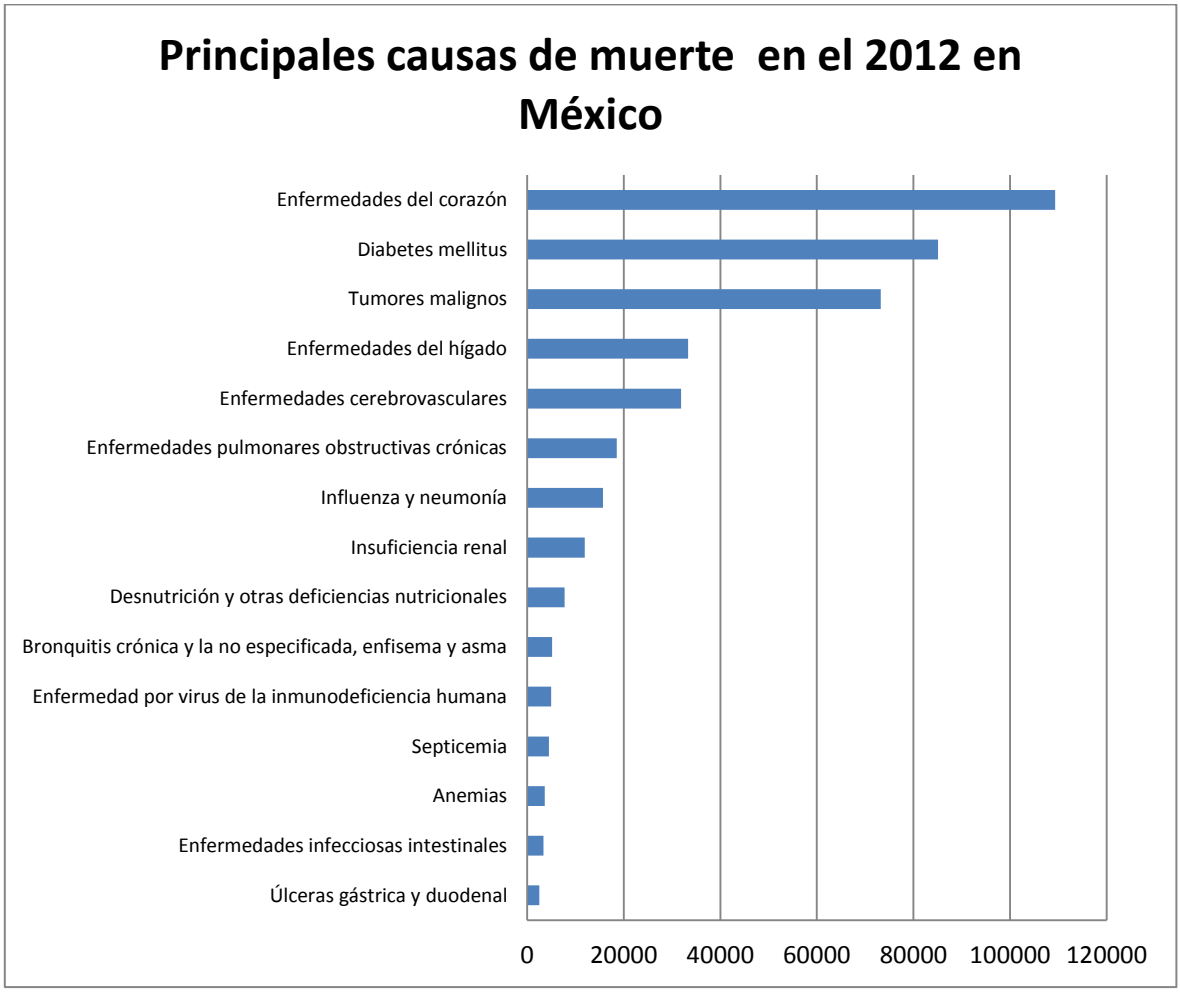


Figura 1.2 Gráfica de las principales causas de muerte registradas en el año 2012.

Como se observa en la imagen, la principal causa de muerte entre los habitantes son las enfermedades de corazón, superando incluso a otros padecimientos igualmente graves como son la diabetes mellitus o los tumores malignos, generalmente relacionados con la presencia de algún tipo de cáncer.

Los factores que provocan que el número de incidentes del corazón sean tan alto son muy variados y en una cantidad grande. Es por ello que se deja de lado el análisis de las causas y se proporciona una herramienta que puede apoyar en el diagnóstico de algunos padecimientos, lo que podría ayudar a reducir la tasa de mortalidad en la población.

1.4. Objetivos

Diseñar un sistema que permita detectar los complejos QRS de una señal electrocardiográfica, usando un Sistema Embebido implementado en un FPGA, para su posterior uso y/o análisis de forma remota por un especialista médico.

1.4.1. Objetivos específicos

- Usar una señal proveniente de alguna fuente simulando una señal real, obteniendo los datos de algún dispositivo electrónico, base de datos o software especial, para comprobar el correcto funcionamiento del sistema.

- Crear un core dedicado para identificar los componentes QRS de una señal ECG usando el algoritmo de correlación cruzada, para aislarlos del resto de los componentes.

- Enviar la señal procesada por medio de internet usando una arquitectura cliente-servidor mediante comunicación entre sockets para que puedan ser evaluadas por un especialista a distancia.

Capítulo 2

Estado del Arte

2.1. Sistemas Embebidos

Un Sistema Embebido se define como un sistema computacional hibrido de hardware y software especialmente diseñado para realizar una tarea específica o un pequeño conjunto de las mismas.

Los Sistemas Embebidos poseen otros nombres para poder referirse a ellos como son Sistemas Empotrados, Sistemas Inmersos, Sistemas Incrustados, Sistemas Alojados.

2.1.1. Características

Los Sistemas Embebidos deben cumplir con ciertas características que los diferencian de otros tipos de sistemas. En la tabla 2.1 se muestran las principales características que poseen así como una breve descripción de cada una de ellas.

Característica	Descripción
Bajo consumo de energía	Ya que solo posee los recursos necesarios para ejecutar su tarea, se tienen pocos componentes, lo que reduce el consumo de energía total del sistema.
Bajo costo	El costo es reducido porque sólo requieren los componentes necesarios para ejecutar su tarea.
Capacidad de cómputo	Su capacidad de procesamiento es mayor en comparación con un sistema de propósito general.
Confiabilidad	El sistema debe trabajar correctamente en todo momento.
Diseño especializado	Son diseñados para cumplir un propósito específico o tareas muy particulares.
Especificaciones de tiempo real	Debe reaccionar a los cambios en las condiciones donde se encuentra, dentro de un intervalo de tiempo definido.

Tabla 2.1 Características más relevantes de los Sistemas Embebidos.

2.1.2. Evolución en los niveles de integración

Los Sistemas Embebidos están conformados por hardware y software. Ambos han pasado por muchos cambios a lo largo del tiempo. En consecuencia, los Sistemas Embebidos también han sufrido muchos cambios tanto en su diseño como en su implementación. Un punto a favor en los continuos cambios, es el desarrollo de dispositivos electrónicos cada vez más pequeños pero con mayor capacidad de procesamiento en un solo dispositivo.

La implementación de los Sistemas Embebidos se apoya de dos tipos de herramientas para su construcción, las herramientas de software embebido y las herramientas de lógica de diseño. El software embebido generalmente se centra en el núcleo del sistema, es decir, el procesador. La lógica de diseño toma en cuenta todos los periféricos asociados al procesador y sus configuraciones.

Ambos conceptos han cambiado en cuanto a la forma de emplear cada uno en su correspondiente área. Al aumentar el nivel de integración de los dispositivos, la separación entre ambos se ha reducido casi al punto de unirse en un solo.

Tomando en consideración las dos partes descritas anteriormente, se pueden distinguir tres generaciones en los sistemas embebidos. Cada generación marca una nueva perspectiva de diseño como se especifica a continuación.

2.1.2.1. Primera generación

En la primera generación se tiene un nivel de integración muy bajo. Los componentes son físicamente independientes unos de otros y la comunicación entre ellos es generalmente a través de buses de datos externos. En cuanto al tiempo de envío y recepción de información entre los componentes, es condicionado por la velocidad de los buses de conexión.

En esta generación, el CPU es generalmente un “hard-core”, independiente del dispositivo programable FPGA. El FPGA es usado para implementar cores personalizados que realicen únicamente la tarea de procesar los datos de manera específica. Es considerado otro periférico que forma parte del sistema más que el componente principal donde se aloja el Sistema Embebido. En esta etapa el CPU sobresale más que el FPGA.

En la figura 2.1 se muestra un ejemplo de los componentes principales que contiene un Sistema Embebido de primera generación y su organización. Es importante señalar que los componentes incluidos en la imagen no son los únicos elementos que integran el sistema completo. Tampoco se muestran las conexiones entre los mismos ni los tipos de buses.

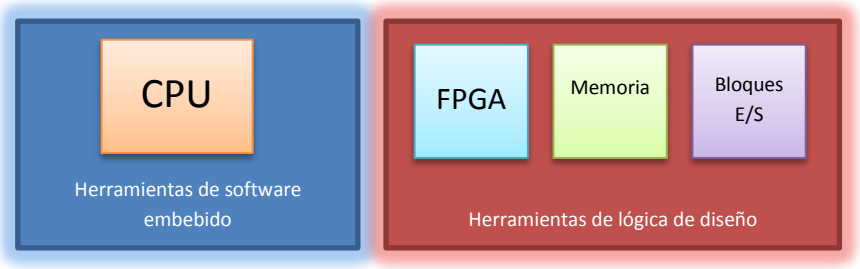


Figura 2.1 Primer nivel de integración de los sistemas embebidos.

2.1.2.2. Segunda generación

En la segunda generación se da un aumento en el nivel de integración de los componentes. Aún se mantiene una separación entre la parte que utiliza las herramientas de software embebido de la parte que utiliza las herramientas de lógica de diseño.

El FPGA comienza a tomar fuerza convirtiéndose en el elemento contenedor de los demás dispositivos periféricos del sistema. El CPU sigue manteniéndose separado físicamente del resto de los componentes. Del lado de las herramientas de lógica de diseño, el FPGA se convierte en el elemento principal, y los demás periféricos se integran dentro del mismo

La figura 2.2 muestra los cambios ocurridos en la parte de las herramientas de lógica de diseño. Se pueden apreciar que los bloques de E/S ahora son de alta velocidad por ser ahora buses internos del FPGA.

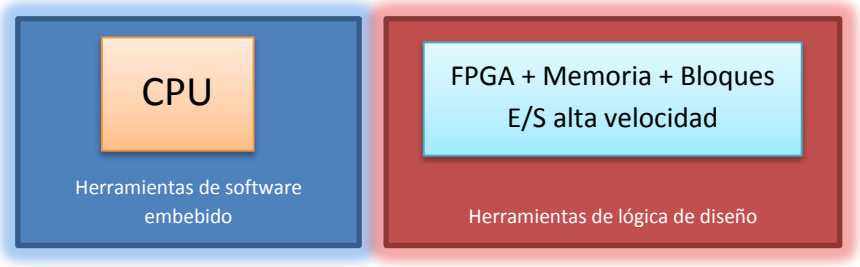


Figura 2.2 Segundo nivel de integración de los sistemas embebidos.

2.1.2.3. Tercera generación

Actualmente existe una tercera generación, donde el nivel de integración es casi total. Se siguen empleando tanto las herramientas de software embebido como las herramientas de lógica de diseño pero desde una perspectiva diferente.

A nivel físico, todos los componentes se encuentran ahora alojados dentro el FPGA. El CPU ahora puede ser de tipo “soft-core” al poder incrustarse como un elemento más. Los periféricos como la memoria y los bloques de E/S siguen siendo elementos programables igual que en la segunda generación.

En cuanto a la comunicación de los periféricos con el CPU, ahora es por completo mediante buses internos del FPGA. Todos los buses son internos, pero existen dos tipos de buses internos, los buses para comunicación con el CPU y los buses para comunicación con los periféricos.

Los buses de comunicación con el procesador son más rápidos que los de comunicación con periféricos, ya que la velocidad de trabajo del procesador siempre es mayor comparada con las velocidades de trabajo de los periféricos.

En la figura 2.3 se observa cómo el CPU ahora se encuentra dentro del FPGA junto con el resto de los elementos del sistema. También se añaden un nuevo elemento, los IP cores. Los IP cores son bloques personalizados que realizan funciones muy específicas. Algunos ejemplos pueden ser el filtrado de una señal o realizar operaciones matemáticas de alto nivel de procesamiento.

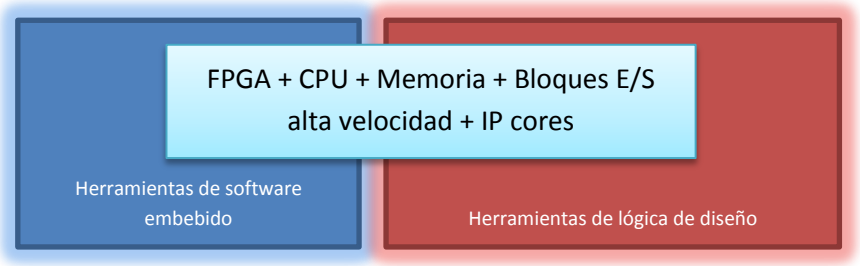


Figura 2.3 Tercer nivel de integración de los sistemas embebidos.

2.1.3. Sistemas SoC y sistemas SoPC

Actualmente existen sistemas que integran en un solo circuito integrado todos los componentes de un sistema electrónico, los cuales son conocidos como sistemas en un chip (SoC por System on a Chip). También es posible trasladar un sistema de tipo SoC a un dispositivo programable como un FPGA. A esta nueva tecnología de implementación se le denomina sistema en un chip programable (SoPC por System on a Programmable Chip). A diferencia de los SoC, los SoPC pueden ser reconfigurados.

Tanto los SoC como los SoPC pueden contener elementos digitales, analógicos o combinaciones de ambos, e incluso pueden contener funciones de radio frecuencia. Una aplicación típica de ambos sistemas es en la construcción de Sistemas Embebidos.

2.1.3.1. System on Chip (SoC)

En dichos sistemas se integran diferentes módulos de hardware en un solo chip. El surgimiento de este tipo de sistemas resulta importante porque la cantidad de circuitos que se utilizaban anteriormente se reduce al uso de un solo chip, logrando con esto reducción en el espacio que ocupan, eficiencia en el consumo de energía, adaptabilidad en las bandas de comunicación y la reducción del costo por componente [6].

Las arquitecturas estructuralmente programables pueden proporcionar una solución de compromiso entre el rendimiento de la lógica cableada y la flexibilidad de los procesadores, gracias a la gran variedad de unidades de lógica reutilizable de propiedad intelectual, llamadas comúnmente IP cores, disponibles para diseñadores de SoCs, donde resalta un cambio importante en la concepción tradicional de los circuitos VLSI, dando un enfoque más moderno hacia lo que se ha denominado SoC [7][8], gracias al rápido diseño del prototipo y a la reconfiguración de los elementos del sistema.

Con la introducción de FPGAs que contienen varios millones de compuertas, ha sido recurrente y práctico agregar un núcleo de procesador al chip FPGA para aplicaciones SoC. La lógica adicional necesaria para construir un procesador embebido puede costar algunos centavos de dólar [9]. Los vendedores de FPGA se refieren a los diseños SoC basados en FPGA como sistemas en un chip programable, (SoPC por System on a Programmable Chip), que son arquitecturas de cómputo reconfigurable y que a diferencia de los SoC se pueden programar de acuerdo a los requerimientos de la aplicación, lo cual permite que el sistema pueda ser como su nombre indica reconfigurable y reprogramable, esta última cualidad es muy apreciada en la enseñanza.

2.1.3.2. System on a Programmable Chip (SoPC)

Debido a la flexibilidad de los SoPC surge un gran interés en el desarrollo de Sistemas Embebidos basados en los sistemas en un chip programable, lo cual permite al diseñador emplear FPGAs muy grandes que contienen elementos lógicos y de memoria [10], además de disponer de núcleos de procesador disponibles para implementar en algún sistema que requiera cumplir una tarea específica.

Un SoPC integra, en principio, un procesador, módulos de memoria, periféricos de entrada y salida, y aceleradores de hardware personalizados en un solo dispositivo FPGA. Además del software personalizado, el hardware personalizado puede ser desarrollado e incorporado en un Sistema Embebido. Con el co-diseño hardware software es posible configurar un procesador, descrito en un lenguaje de descripción de hardware o también denominado soft-core, crear interfaces de entrada/salida adaptadas a la aplicación, desarrollar módulos aceleradores de hardware especializados para tareas que requieran de alto nivel de procesamiento.

2.1.4. Hard-core y Soft-core

Un core es una unidad de lógica reutilizable. Los núcleos de procesador usados en diversos SoPC se dividen en hard-cores y soft-cores.

2.1.4.1. Hard-core

Un procesador hard-core es un microprocesador que no puede ser modificado por el diseñador. Los procesadores hard-core son embebidos en un FPGA en una zona dedicada del mismo y son independientes del resto de la lógica del FPGA.

2.1.4.2. Soft-core

Un procesador soft-core es un microprocesador totalmente descrito por software, por lo general en un lenguaje de descripción de hardware (como VHDL o Verilog), que se puede sintetizar en hardware programable, tales como FPGA. Un procesador soft-core implementado en FPGA es flexible, ya que sus parámetros se pueden cambiar en cualquier momento mediante la reprogramación del dispositivo.

En la figura 2.4 se muestra la clasificación de algunos procesadores hard-core y soft-core comerciales.

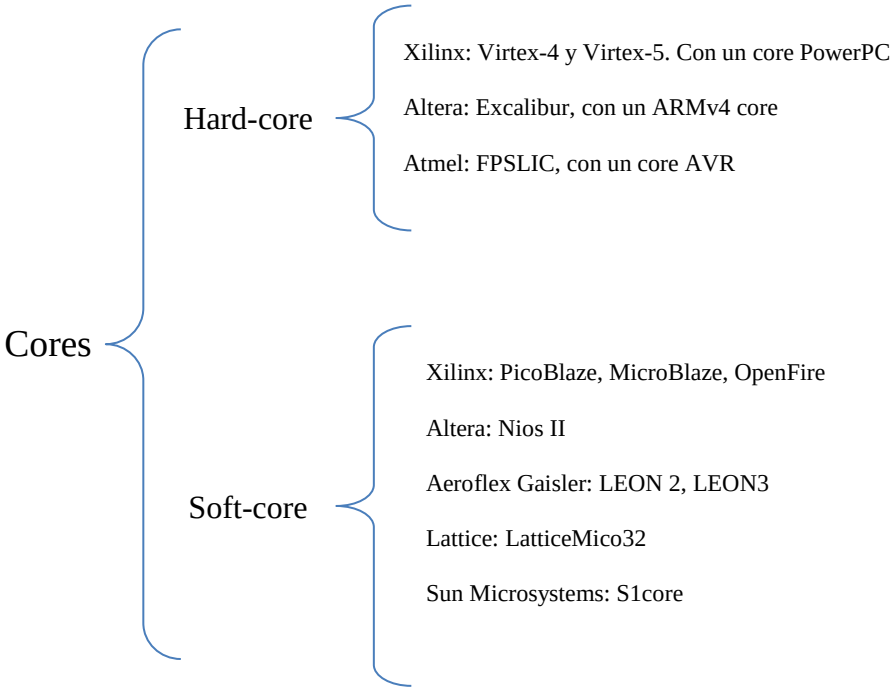


Figura 2.4 Procesadores Hard-core y Soft-core disponibles en el mercado.

Los procesadores hard-core no se reconfiguran pero tienen mejor rendimiento que los procesadores soft-core.

Los soft-cores, como el Nios II de Altera o el Microblaze de Xilinx, usan elementos lógicos programables de la FPGA para implementar el procesador [11], y permiten al diseñador especificar el datapath, el funcionamiento de la unidad aritmética lógica (ALU), el número y tipo de periféricos y el espacio de memoria, pero tienen algunas desventajas como: frecuencias de reloj más lentas y consumen mayor potencia que los hard-cores [9].

2.2. Procesamiento digital de señales

El procesamiento digital de señales es un proceso en el cual se toma una señal de entrada, se convierte en una serie de datos digitales, se realizan procesos sobre dichos datos y se obtiene un resultado. Cada una de estas etapas puede ser simple o compleja, dependiendo de la información que se desea obtener de la señal.

En la mayoría de los Sistemas Embebidos que involucra el procesamiento digital de señales, existen tres etapas principales, la adquisición, el tratamiento de los datos y la salida.

2.2.1. Adquisición

En la actualidad, las señales que se analizan con mayor frecuencia para obtener información, son variables físicas del ambiente o señales biológicas de seres vivos. Este tipo de señales tiene una naturaleza analógica, es decir, tienen un conjunto muy grande de valores, donde cada uno está asociado a un instante en el tiempo.

A pesar del gran poder de cómputo que poseen los sistemas actuales, no es posible procesar una señal analógica directamente. El problema que se tiene es la dificultad de procesar valores diferentes dentro de un sistema que trabaja con valores binarios.

Para solucionar este problema, se requiere hacer una digitalización de la señal. La digitalización es el proceso donde a una porción de la señal se le asocia valor binario. El conjunto de valores que se obtienen puede ser manipulado de una manera más sencilla y con mayor rapidez en un hardware especializado en dicha tarea.

Una de las ventajas de las señales digitales, es que pueden ser regresadas a señales analógicas por medio de la reconstrucción de la señal. La reconstrucción es el proceso inverso a la digitalización, donde a cada valor binario de la señal, se le asocia un valor analógico.

2.2.2. Tratamiento de los datos

En este punto, es donde se procede a manipular la información obtenida en la etapa de adquisición. Para que los datos se puedan procesar, la información debe haber pasado por una digitalización de manera previa. En el caso particular de las señales provenientes de seres vivos, existen procesos muy comunes que se aplican a las señales. Un ejemplo ampliamente utilizado en la actualidad dentro de los Sistemas Embebidos es el filtrado digital.

Los filtros digitales tienen una gran variedad de usos y aplicaciones como son obtener información específica, quitar información innecesaria, modificar la información contenida, entre otras. La forma más común de implementar filtros es con ayuda funciones matemáticas como la Transformada de Fourier. La Transformada de Fourier permite diseñar filtros especiales que se conocen como filtros FIR e IIR. En la práctica, únicamente se implementan los filtros FIR, ya que sólo estos tienen un resultado significativo a nivel práctico.

De manera muy general, el funcionamiento de los filtros FIR es mediante una serie de sumas y productos entre arreglos de valores. En su mayoría, se necesitan solamente dos arreglos para la implementación del filtro. Un primer arreglo contiene los valores digitales o discretos de la señal que se va a procesar. El segundo arreglo contiene una serie de coeficientes que se operan con el primer arreglo para obtener como salida el resultado deseado.

2.2.3. Salida

Después que la señal es procesada, en la salida se obtiene la señal resultante. Este resultado puede utilizarse de múltiples maneras según sea necesario o dependiendo del diseño del Sistema Embebido.

Un ejemplo de posibles aplicaciones de la señal es usar los elementos resultantes como variables de entrada para controlar algún otro proceso dentro o fuera del sistema. Un segundo ejemplo consiste en tomar la señal resultante que se encuentra en formato digital o discreto, y convertirlo a una señal analógica. La conversión a señal analógica puede hacerse mediante hardware o vía software. Una tercera opción consiste en enviar la información a un medio de almacenamiento persistente como puede ser un servidor en internet o una unidad de disco local.

En cada uno de los tres ejemplos, se extrae una parte de la información de la señal, que es la parte de interés que se busca obtener desde la etapa de adquisición hasta la salida.

2.3. ECG

Un electrocardiograma o ECG es un registro de la actividad eléctrica producida por el corazón. El corazón funciona de manera autónoma, mediante pequeñas ondas eléctricas que se generan por diferencias de voltaje entre las células del corazón. Cada parte del corazón está asociada con un tipo de onda en particular con características específicas de tiempo y magnitud.

2.3.1. Estructura del corazón

El corazón está compuesto de diferentes tipos de tejido cardiaco organizados en capas. Dichas capas son endocardio, miocardio, epicardio y pericardio donde cada una posee una función específica.

Endocardio: Es una capa delgada que recubre la parte interna del corazón. Se compone de fibras elásticas de colágeno, vasos sanguíneos y fibras musculares especializadas llamadas Fibras de Purkinje.

Miocardio: Es el músculo cardíaco encargado de impulsar la sangre por todo el cuerpo mediante su contracción. También cuenta con tejido conectivo, capilares sanguíneos, capilares linfáticos y fibras nerviosas.

Epicardio: Es una membrana viscosa que forma parte del pericardio, formando una bolsa que cubre el corazón.

Pericardio: Es una membrana con dos partes, el pericardio seroso y el pericardio fibroso. Envuelve completamente al corazón y los grandes vasos formando un saco.

La figura 2.5 muestra una representación del orden en el que se encuentran ordenadas las capas anteriormente descritas.

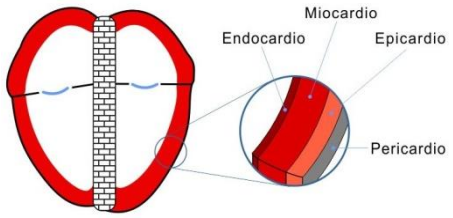


Figura 2.5 Estructura del corazón por capas.

Al mismo tiempo, el corazón está dividido por secciones o cavidades cardiacas. La parte superior está compuesta por dos aurículas, derecha e izquierda. La parte inferior consta de dos ventrículos, derecho e izquierdo.

La aurícula derecha, junto con el ventrículo derecho, pertenecen al lado derecho del corazón. Se encarga de circular la sangre hacia los pulmones para su oxigenación. La aurícula izquierda y el ventrículo izquierdo forman el lado izquierdo del corazón. Su función es enviar la sangre con oxígeno y nutrientes a todo el organismo.

En la figura 2.6 se indica la disposición tanto de las aurículas, como de los ventrículos [12].

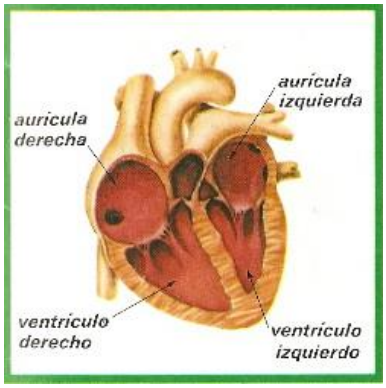


Figura 2.6 Cavidades del corazón.

2.3.2. Generación de un ECG

El ECG representa la actividad eléctrica de las células de un corazón. Los impulsos eléctricos generados por el músculo cardíaco (el miocardio) estimulan el latido (contracción) del corazón. Esta señal eléctrica se origina en el nódulo sinoauricular (SA) ubicado en la parte superior de la aurícula derecha.

El nódulo SA también se denomina el «marcapasos natural» del corazón. Cuando este marcapasos natural genera un impulso eléctrico, estimula la contracción de las aurículas.

A continuación, la señal pasa por el nódulo auriculoventricular (AV). El nódulo AV detiene la señal un breve instante y la envía por las fibras musculares de los ventrículos, estimulando su contracción [13].

Este proceso se repite una y otra vez en un ciclo conocido como ciclo cardiaco. Cada evento dentro del corazón se produce de manera secuencial uno detrás del otro. El registro de cada uno de los eventos permite descubrir anormalidades en el ciclo cardiaco.

La figura 2.7 muestra los pulsos producidos por cada parte del corazón. El ECG es el resultado de la combinación de cada pulso para formar una sola señal continua.

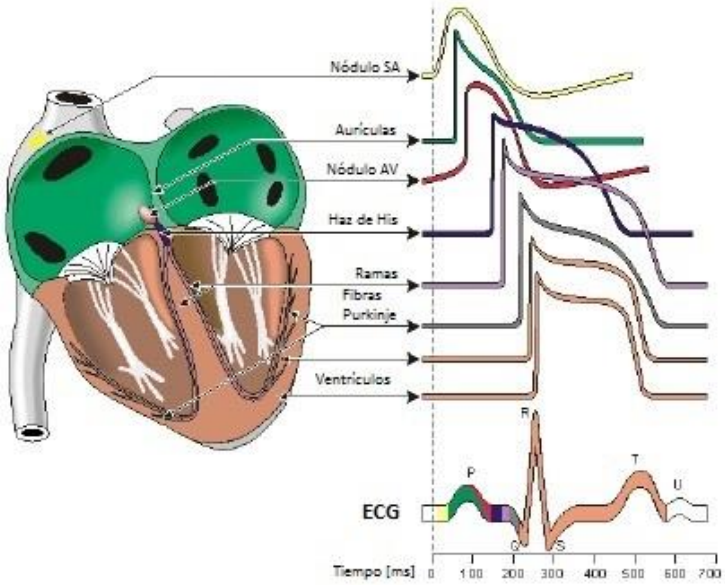


Figura 2.7 Actividad eléctrica del corazón y el ECG generado.

2.3.3. Componentes del ECG

Para su análisis, el ECG se divide en distintos tipos de ondas que se diferencian mediante letras. Las letras usadas van desde la letra P hasta la letra U. El orden que siguen es onda P, complejo QRS, onda T y onda U. A su vez, entre cada onda existen segmentos que se nombran con las mismas letras de las ondas que conectan. Los segmentos son PQ, QT y ST.

En la gráfica de un ECG normal se pueden visualizar la onda P, el complejo QRS y la onda T. La onda U es pequeña y normalmente es invisible. Estos son los eventos eléctricos correspondientes con los eventos mecánicos de la contracción y relajación de las cámaras del corazón.

La sístole o contracción ventricular comienza justo después del inicio del complejo QRS y finaliza justo antes de terminar la onda T. La diástole, que es la relajación y rellenado ventricular, comienza después que culmina la sístole correspondiendo con la contracción de las aurículas, justo después de iniciarse la onda P.

En la figura 2.8 se muestra la forma de un ECG con los componentes respectivos y algunos datos relevantes de cada componente, como son su duración en tiempo y su valor de amplitud aproximado [14].

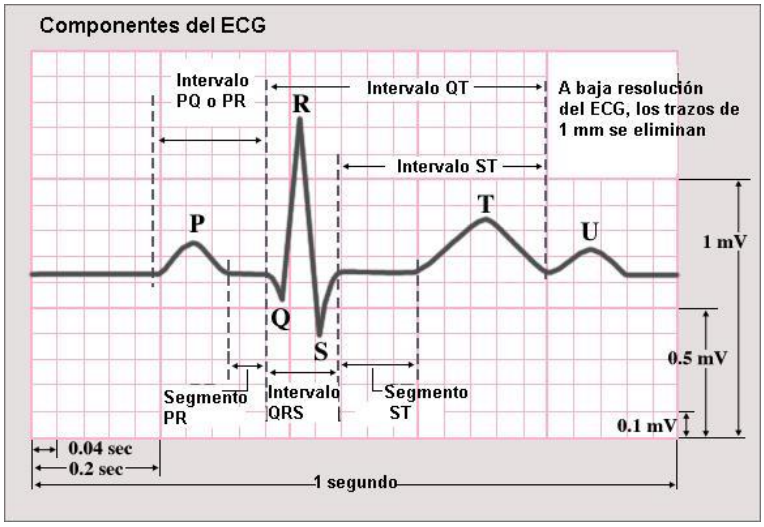


Figura 2.8 Componentes y segmentos del ECG.

Onda P

La onda P es la señal eléctrica que corresponde a la despolarización auricular. Resulta de la superposición de la despolarización de la aurícula derecha (parte inicial de la onda P) y de la aurícula izquierda (final de la onda P). Debe reunir ciertas características:

- No debe superar los 0.25 mV. Si lo supera, estamos en presencia de un agrandamiento auricular derecho.

- Su duración no debe superar los 110 milisegundos en el adulto y 70-90 milisegundos en los niños. Si está aumentado posee un agrandamiento auricular izquierdo y derecho.

- Precede al complejo QRS ventricular.

Complejo QRS

La despolarización ventricular ocurre mediante la sucesión de tres vectores: septal, pared libre y basal. El primer vector tiene una duración de 10 milisegundos, es de poca magnitud, está dirigido a la derecha y adelante, y puede orientarse hacia arriba o abajo dependiendo de la posición eléctrica del corazón. El segundo vector tiene una duración de 40 milisegundos y una magnitud diez veces mayor que el primero. Se dirige hacia la izquierda, atrás, y abajo o arriba según la posición del corazón. El tercer vector tiene una duración de 20 milisegundos y se dirige hacia la derecha, arriba, y atrás (en adultos) o adelante (en la infancia y juventud). De la sumatoria de los tres vectores individuales se origina un vector resultante que constituye el promedio de la activación ventricular y se representa por el eje eléctrico [15].

- Tiene una duración de 60 a 100 milisegundos.
- La amplitud del complejo QRS es muy variable, ubicándose en los 3.5mV aproximadamente.
- Precede a la onda T

Onda T

La onda T representa la re polarización de los ventrículos. En la mayoría de las ocasiones, la onda T es positiva. Las ondas T negativas pueden ser síntomas de enfermedad.

- Su duración aproximadamente es de 200 milisegundos.
- Posee una amplitud de 0.5 mV.

Onda U

Existe controversia sobre el origen de la onda U. Se considera que en condiciones normales, la onda U es el resultado del potencial de acción prolongado de las células M. Su polaridad es la misma que la onda T que le precede y su amplitud no suele rebasar el 15 % de la misma. Es importante no confundir a la onda U con una onda T bimodal. Esta diferenciación es particularmente importante cuando se valora la duración del intervalo QT (que no debe incluir a la onda U) [15].

Intervalo PQ o PR

El segmento PQ abarca desde el comienzo de la onda P hasta el comienzo de las ondas Q o R (cuando la onda Q no está presente). Se produce por la actividad eléctrica del nodo sinusal, aurículas, nodo AV, His, rama derecha e izquierda y red de Purkinje.

- La duración del intervalo PQ o PR es de 120 a 200 milisegundos

Intervalo QT

El intervalo QT corresponde a la despolarización y re polarización ventricular. Se mide desde el principio del complejo QRS hasta el final de la onda T. El intervalo QT y el QT corregido son importantes en el diagnóstico del síndrome de QT largo y síndrome de QT corto. Su duración varía de forma inversa con respecto a la frecuencia cardíaca, por lo que se debe aplicar un método de corrección.

El método más utilizado es el formulado por Bazett, publicado en 1920. La fórmula de Bazett es:

$$QTc = \frac{QT}{\sqrt{RR}}$$

donde:

QTc es el intervalo QT corregido para la frecuencia cardíaca

QT es el intervalo original,

RR es el intervalo desde el comienzo de un complejo QRS hasta el siguiente, medido en segundos.

Sin embargo, esta fórmula tiende a ser inexacta; sobre-corrige en frecuencias cardíacas altas e infra-corrige en las bajas.

Un método mucho más exacto fue desarrollado por el Dr. Pentti Rautaharju, que creó la fórmula:

$$QTp = \frac{656}{1 + \frac{frecuencia\ cardiaca}{100}}$$

- El límite inferior normal de duración es de 360-370 milisegundos.

- El punto de corte superior varía dependiendo de la edad y el género: 440 milisegundos hasta los quince años, 460 milisegundos en mujeres adultas y 450 milisegundos en hombres adultos[15].

Intervalo ST

El segmento ST abarca desde el final de la onda S o la onda R (cuando la onda S no está presente) hasta el comienzo de la onda T. Muestra el comienzo de la re polarización ventricular. El segmento ST es isoeléctrico (sin carga positiva o negativa) debido a que todas las células ventriculares se encuentran des-polarizadas.

2.4. Otros proyectos

La necesidad de poder monitorear la actividad del corazón de una persona es de vital importancia para el diagnóstico de enfermedades cardiacas. Al mismo tiempo, la forma de realizar ese monitoreo ha podido evolucionar al punto de poder realizarse remotamente.

La tabla 2.2 contiene algunas características de tres proyectos similares desarrollados dentro del Instituto Politécnico Nacional en diferentes unidades académicas. También se incluyen las características del presente trabajo a manera de comparativo para poder visualizar las diferencias más relevantes entre los trabajos.

Característica	Equipo biomédico con telemetría diseñado para las áreas rurales[16]	Diseño de un electrocardiógrafo para el laboratorio de bioacústica de la ESIME Zacatenco[17]	Electrocardiógrafo para aplicaciones en medicina[18]	TT 2013-A045
Tipo de tesis	N/E	Licenciatura	Maestría	Licenciatura
Número de integrantes	2	2	1	1
Año de desarrollo	2005	2012	2010	2013
Monitoreo remoto (Telemetría)	SI	NO	SI	SI
Tecnología utilizada para el procesamiento	Circuitos integrados y microcontroladores	Circuitos integrados	Circuitos integrados y microcontroladores	FPGA
Posibilidad de modificar los módulos	NO	NO	NO	SI
Posibilidad de expansión	NO	NO	NO	SI
Tecnología utilizada para el monitoreo remoto	Red de telefonía móvil	N/A	Red WiMax Red IP	Internet
Nivel de integración del hardware (Sistemas Embebidos)	2ª generación	1ª generación	2ª generación	3ª generación
Separación de la señal en componentes	NO	NO	NO	SI
Requiere fabricar el hardware	SI	SI	SI	NO
Etapas incluidas en el desarrollo	• Adquisición • Acondicionamiento • Limitación ancho de banda • Transmisión	• Adquisición • Acondicionamiento • Limitación de ancho de banda	• Adquisición • Acondicionamiento • Limitación ancho de banda • Transmisión	• Adquisición (señal simulada) • Tratamiento • Transmisión

Tabla 2.2 Comparativo con diferentes proyectos ya desarrollados.

Capítulo 3
Metodología

3.1. Introducción

En la actualidad, todos los productos y proyectos deben ser desarrollados bajo un esquema de trabajo bien definido y estructurado. Estos esquemas se denominan metodologías, y ayudan a asegurar, en la medida de sus posibilidades, un trabajo que cumpla con estándares que garanticen su correcto funcionamiento.

Dentro del área de los sistemas computacionales, las metodologías se aplican en el desarrollo de sistemas de software, hardware o híbridos. Las metodologías contienen una serie de estándares o normas que especifican una serie de rubros como son la manera en que se debe dividir y organizar cada una de las fases del proyecto, su duración, los detalles mínimos que se deben incluir en la documentación, entre otros.

Para los productos donde el sistema a desarrollar es enteramente en software, existen muchas metodologías de trabajo. Algunas de las metodologías tradicionales más utilizadas por los desarrolladores son:

- Cascada
- Espiral
- Incremental
- Prototipos

Además se han desarrollado nuevas metodologías conocidas como metodologías ágiles, que se utilizan comúnmente para obtener desarrollos en un menor tiempo. Ejemplos de algunas de ellas son:

- Kanban
- Krystal
- Scrum
- XP

En contraparte, no ocurre lo mismo para el caso de los sistemas de hardware e híbridos. En el caso donde los productos deben ser desarrollados total o parcialmente en hardware, no se cuenta con metodologías definidas y estandarizadas que indiquen la forma de construir el sistema.

Sin embargo, existen métodos no estandarizados que ayudan a organizar las fases de desarrollo y especifican algunos requisitos mínimos que se deben cubrir durante todo el tiempo que dura terminar el sistema por completo. Este tipo de metodologías no estándares, están mayormente enfocadas al desarrollo de sistemas híbridos, que pueden ser tomadas y adaptadas para los Sistemas Embebidos, donde se requiere de hardware y software.

3.2. Metodología propuesta

En esta metodología, se sugiere tomar todo el sistema y dividirlo en dos partes principales: hardware y software. La parte de hardware corresponde a los componentes físicos del sistema. En este caso en particular corresponde al FPGA. La parte de software corresponde a los programas que se incrustan dentro del dispositivo programable y la mayoría del tiempo, controlan el flujo de información.

En la figura 3.1 se muestra la metodología propuesta para Sistemas Embebidos desarrollados bajo un entorno EDK (Embedded Development Kit o Kit de Desarrollo de Embebidos). En ella se muestra el flujo de diseño de ambas partes del Sistema Embebido [19].

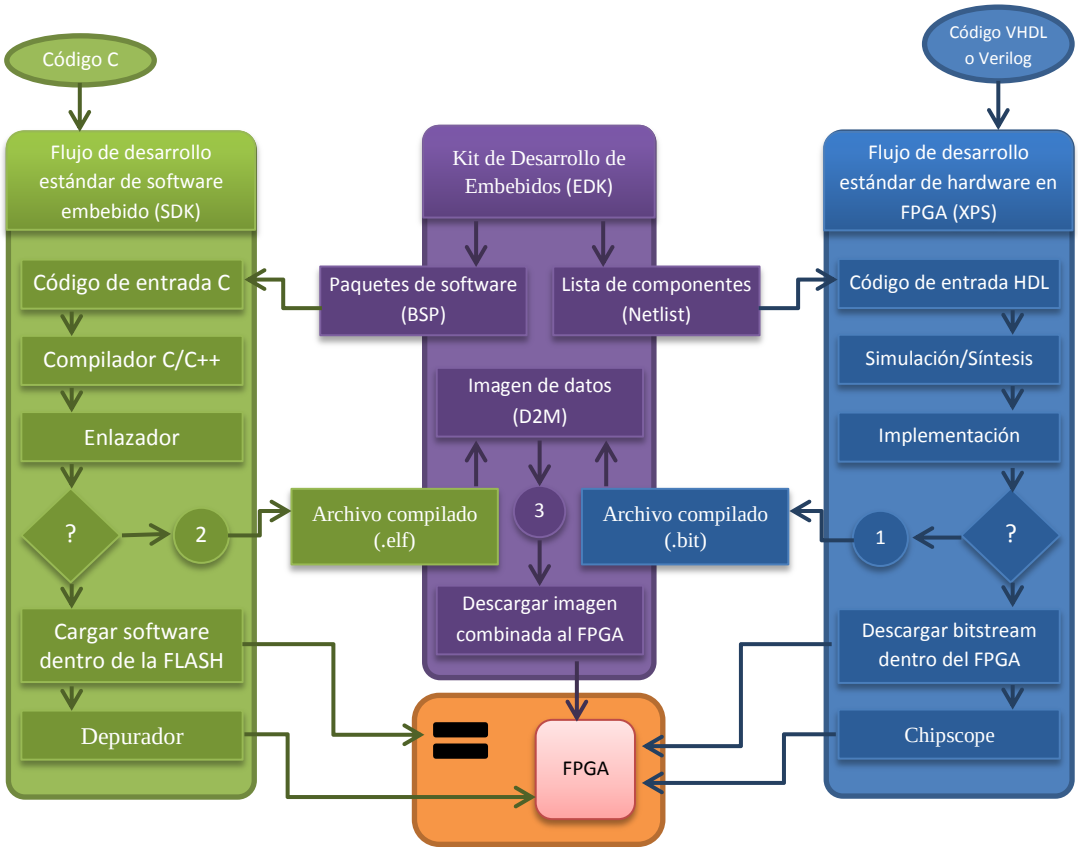


Figura 3.1 Flujo de diseño para sistemas embebidos desarrollados en EDK.

En la imagen se observan tres etapas numeradas y separadas unas de otras. Cada una de las tres etapas corresponde a la parte de desarrollo de hardware, la parte de desarrollo de software y una fase de unión de ambas.

En la etapa 1 se muestran las actividades correspondientes al diseño de la parte de hardware. El lenguaje de programación para esta etapa puede ser VHDL o Verilog. Tiene la opción de generar un archivo de programación completo que puede ser directamente descargado en el FPGA. También es posible generar los archivos necesarios que, más adelante, se unirán a los archivos de software para formar el sistema completo.

La etapa 2 corresponde al flujo de diseño de la parte del software. En los Sistemas Embebidos, es muy común que se realice la implementación en el lenguaje de programación C. De manera similar a como ocurre en la etapa de hardware, también es posible crear o modificar la parte del software e implementarla de manera directa en el FPGA. Sin embargo, existe una diferencia notable entre ambas partes.

A diferencia de la implementación del hardware, donde se puede realizar el diseño completo o parcial y colocarlo dentro del FPGA directamente, no es posible hacerlo de manera directa con la parte del software sin haber depurado previamente el código. Una alternativa para ello, es cargar el programa compilado sin depurar en una memoria externa. El programa es entonces extraído de la memoria para ser usado en conjunto con el hardware implementado en el PFGA.

Para la etapa 3 se considera el desarrollo conjunto de ambas partes del sistema embebido. Aquí el EDK requiere de los componentes de ambas partes para poder crear el sistema. Las partes mínimas que requiere son: un archivo con las especificaciones de conexiones entre los componentes del FPGA así como otro archivo con la plataforma sobre la cual se programa el código del software.

Posteriormente, para crear el archivo de imagen final con el que se programa el FPGA, es necesario contar con dos archivos. El archivo de configuración de hardware (.bit) que contiene todas las especificaciones de conexiones, mapeo de componentes, síntesis de módulos, mapeo de puertos de E/S por mencionar algunos ejemplos. También se requiere del archivo generado al compilar el código en C del software (.elf) que es el que contiene las librerías de comunicación con los módulos del hardware, las funciones creadas por el usuario, datos de variables y constantes, flujo y control de datos, entre otros más.

Ambos archivos son combinados en uno solo por el EDK. Una vez combinados, el archivo resultante se carga dentro del FPGA para poder comprobar el funcionamiento del Sistema Embebido.

De acuerdo con la metodología propuesta para Sistemas Embebidos, el desarrollo de este trabajo debe descomponerse en tres fases: diseño del hardware, diseño del software y unión de ambas partes para su implementación en el FPGA.

En la figura 3.2 se muestra, de manera muy general, la forma en la que interactúan las fases de hardware y software para lograr la comunicación entre el usuario y el Sistema Embebido.

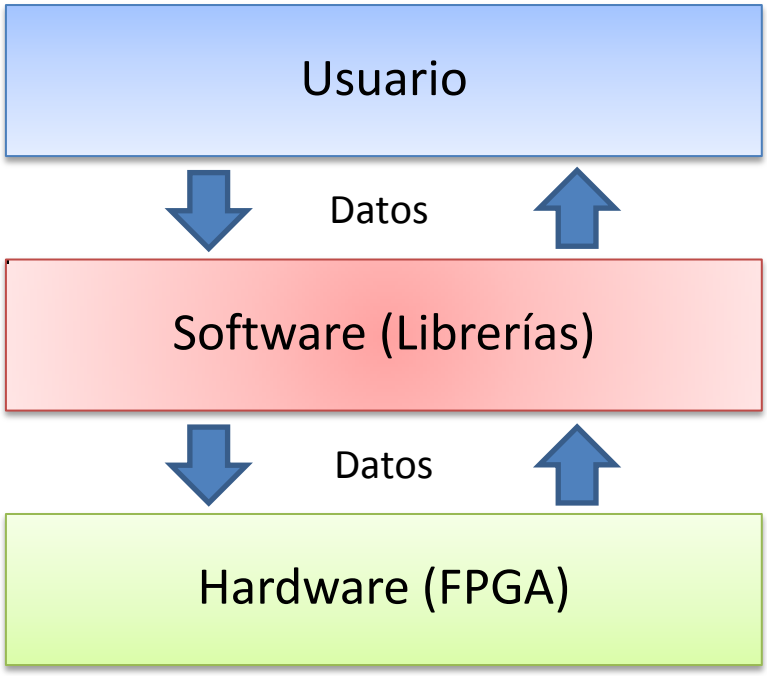


Figura 3.2 Diagrama de interacción de la parte de hardware con la parte de software.

En este caso, la parte del software corresponde a las librerías, las cuales contienen las funciones necesarias para poder manipular los datos dentro del hardware. Las librerías necesarias para cada módulo del hardware son generadas automáticamente por el EDK como archivos de cabeceras .h. La razón por la que se emplea este formato, es porque se utiliza el lenguaje de programación C, donde las librerías poseen formato .h.

Capítulo 4
Análisis

4.1. Análisis de requerimientos

Cuando se realiza el desarrollo de algún proyecto, el primer paso es determinar todas las características o requerimientos con las que debe cumplir. Los requerimientos pueden ser determinados ya sea por el solicitante, el usuario final, o las restricciones físicas del mismo sistema. En el caso de los Sistemas Embebidos, que son sistemas híbridos, se tienen dos tipos de requerimientos: requerimientos de hardware y requerimientos de software.

En la parte de requerimientos de hardware, se especifican todos los recursos físicos a utilizar. En algunos casos, también se determinan aspectos relacionados con las características del hardware como son: capacidad de almacenamiento, frecuencia de reloj, interfaces de comunicación, entre otros.

En cuanto a los requerimientos de software, se deben incluir todas las restricciones que afectan las características del programa embebido. Los principales puntos que se especifican son: comunicación con el hardware, presentación de resultados al usuario, intercambio de información con otros sistemas, por mencionar algunos ejemplos.

En la figura 4.1 se identifican los bloques necesarios para poder realizar, desde la adquisición de la señal real del corazón, hasta la etapa de visualización y análisis de la información obtenida.

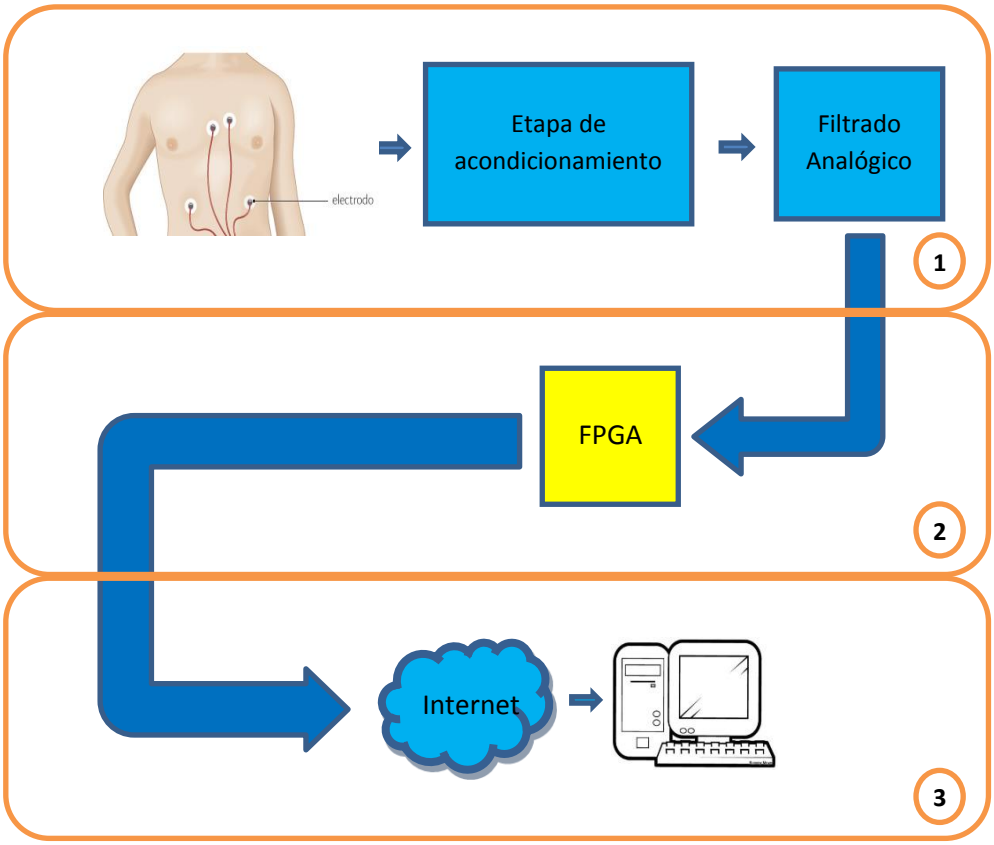


Figura 4.1 Diagrama general de los bloques identificados para el sistema completo.

Como se puede observar, cada etapa del sistema completo puede ser agrupado en tres diferentes fases: adquisición, procesamiento y salida.

En la fase 1 se adquiere la señal del paciente por medio de electrodos. Después se pasa por una etapa de acondicionamiento para poder ser procesada por el hardware. Opcionalmente también se puede usar una etapa de filtrado para obtener una señal más limpia. En el caso del presente trabajo, esta primera fase es sustituida por una fuente que proporciona una señal cardiaca simulada.

En la fase 2 es donde se realiza todo el procesamiento de la señal. Aquí se implementa el algoritmo para poder identificar el complejo QRS de la señal. También se realiza el envío de los datos a internet para la siguiente fase.

En la fase 3 se recupera la información obtenida en la fase anterior para poder reconstruir la señal y visualizar el resultado del procesamiento. Opcionalmente se puede hacer un análisis de forma automática, pero en esta ocasión, el análisis automático se sustituye por un especialista médico que interprete los resultados.

En éste caso particular, solamente se obtienen los requerimientos que contiene la parte de procesamiento en el FPGA. Las otras fases se dejan como trabajo a futuro.

En la tabla 4.1 se especifican los requerimientos básicos o generales de la fase 2 del sistema. Cada uno de estos requerimientos puede ser un requerimiento de hardware o software o ambos, por lo que hay que diferenciar los correspondientes a cada uno.

ID	Descripción
RB1	Adquirir una señal simulada en sustitución de la señal cardiaca en tiempo real.
RB2	Se debe contar con un módulo que controle la adquisición de los datos.
RB3	Identificar los complejos QRS presentes en la señal.
RB4	Almacenar temporalmente los datos dentro del sistema antes del envío.
RB5	Enviar la información de la señal procesada a internet.
RB6	El sistema debe contener una unidad central de control (microprocesador)

Tabla 4.1 Requerimientos básicos del sistema.

4.1.1. Requerimientos de procesamiento

Muchas veces, cuando se desarrollan proyectos que involucran el procesamiento digital de señales, es altamente recomendable realizar un análisis de las características de la señal a procesar. Las características propias de la señal pueden ayudar a seleccionar el hardware y software más adecuado para obtener mejores resultados.

Cada señal posee características propias que pueden ser analizadas o comparadas con otras señales. Usualmente, las características más importantes a considerar de una señal son: su frecuencia, el periodo, los componentes de la señal, el ancho de banda y algunos otros. Los puntos a considerar, así como su descripción son:

Ancho de banda

De acuerdo a la American Heart Association (AHA), la banda de frecuencias recomendada es de los 0.05 a los 100 hertz (Hz)[20].

Frecuencia de muestreo

La frecuencia de muestreo debe ser de al menos 500 Hz[20].

Resolución de la digitalización

La señal debe ser digitalizada con un mínimo de 12 bits de resolución[20].

Frecuencia de la señal

La frecuencia de la señal está directamente relacionada con la frecuencia cardiaca. En un adulto sano, la frecuencia cardiaca se considera normal cuando se encuentra en un rango entre 60 a 100 latidos por minuto.

Para obtener la frecuencia de la señal, hay que calcular el número de latidos por segundo. Para hacerlo, se divide el número de latidos por minuto entre 60.

$$latidos\ x\ segundo = \frac{latidos\ x\ minuto}{60}$$

Aplicando la fórmula a los límites superior en inferior de un ritmo cardiaco normal se obtiene lo siguiente.

$$\frac{60\ latidos\ x\ minuto}{60} = 1\ latido\ x\ segundo$$

$$\frac{100\ latidos\ x\ minuto}{60} = 1.66\ latidos\ x\ segundo$$

Por lo tanto, la frecuencia de un ECG normal se encuentra en el intervalo de 1 a 1.66 Hz.

4.1.2. Requerimientos de hardware

En la parte de requerimientos de hardware es necesarios considerar la función que realizara el Sistema Embebido. Cada etapa requiere de un hardware específico, que puede encontrarse separado físicamente por módulos, o incluido en un solo dispositivo.

En la tabla 4.2 se muestran los requerimientos de la parte de hardware del sistema embebido.

ID	Origen	Descripción
RH1	**RB1**	El sistema debe permitir leer la señal de entrada.
RH2	**RB3**	El sistema debe contar con una unidad especializada para el procesamiento digital de señales (DSP).
RH3	**RB3**	Diseñar un IP core que obtenga el complejo QRS de la señal.
RH4	**RB4**	El sistema debe contar con un bloque de memoria donde se almacenen los resultados del filtrado.
RH5	**RB5**	El sistema debe contar con interfaces que permitan la conectividad a internet o a una computadora.
RH6	**RB6**	El sistema debe contener un módulo principal de control de todo el sistema (microprocesador).
RH8	-	El sistema debe ser embebido dentro de un FPGA.

Tabla 4.2 Descripción de los requerimientos de hardware.

4.1.3. Requerimientos de software

El software es la parte complementaria del sistema que ayuda a realizar la comunicación con el hardware para controlar la información. Como en cualquier sistema, tanto hibrido como sólo de software, también está sujeto a las restricciones propias del sistema donde se aloja.

La tabla 4.3 muestra los requerimientos de software que son necesarios para poder hacer uso de los componentes del hardware. Como el sistema se aloja dentro de un FPGA, el software consta principalmente de librería que contienen funciones para controlar el flujo de información a través de los módulos del hardware.

ID	Origen	Descripción
RS1	**RB2**	Se debe generar la librería y el driver para utilizar el IP core de adquisición.
RS2	**RB3**	Se debe generar la librería y el driver para el uso del IP core de filtrado.
RS3	**RB4**	Se deben generar la librería y el driver para el uso de los bloques de memoria RAM.
RS4	**RB5**	El software debe poder gestionar el envío a internet usando algún protocolo de comunicación.
RS5	**RB6**	Se debe generar la librería y el driver para usar el microprocesador.

Tabla 4.3 Descripción de los requerimientos de software.

4.2. Análisis del hardware

4.2.1. FPGA

Un FPGA (Fiel Programmable Gate Array o Arreglo de Compuertas de Campos Programables) es un dispositivo electrónico que contiene bloques lógicos programables interconectados entre sí. La programación del FPGA se lleva a cabo mediante un lenguaje de descripción de hardware, como pueden ser VHDL o Verilog.

La arquitectura de los FPGA es muy variada de un fabricante a otro. Sim embargo, todos siguen una arquitectura básica en su diseño. Cada uno contiene tres elementos básicos que permiten implementar los diseños dentro de ellos. Los componentes base son los bloques de lógica, los bloques de E/S y las líneas de interconexión.

Los bloques lógicos son la parte donde se implementan de manera física las funciones del sistema. Se pueden implementar muchos tipos de lógica, desde lógica simple de compuertas, hasta circuitos combinatorios y secuenciales o combinaciones de ambos.

Los bloques de E/S son otra parte del FPGA que permiten llevar la información contenida dentro de los bloques lógicos hacia el exterior. Sirven como conexión entre los bloques lógicos internos y las terminales físicas del dispositivo (pines).

Las líneas de interconexión son la vía de comunicación entre los bloques lógicos y los bloques de E/S, así como entre los bloques mismos. Dependiendo del diseño y del fabricante, pueden existir diferencias en las líneas de interconexión, existiendo de baja y alta velocidad.

En la figura 4.2 se observa un diagrama genérico de la organización de un FPGA. Cada fabricante puede agregar componentes adicionales a su dispositivo como pueden ser bloques de memoria RAM, unidades DSP, lógica de acarreo, flip-flop's, controladores físicos de memoria RAM, entre otros.

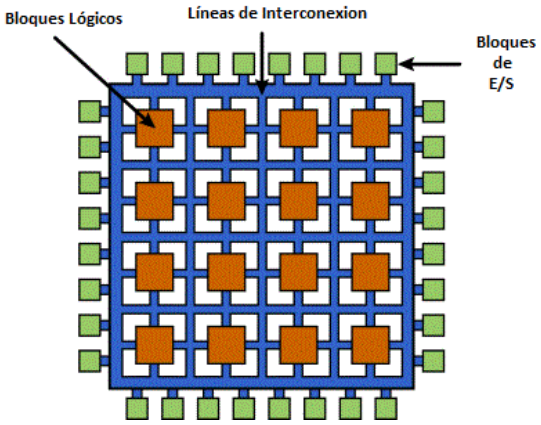


Figura 4.2 Diagrama general de la arquitectura de un FPGA.

4.2.1.1. Comparación

Existen varias compañías que han incursionado en la fabricación de dispositivos FPGA. En la actualidad, dos compañías se han colocado como las líderes en el mercado de estos dispositivos: Xilinx y Altera.

En la tabla 4.4 se muestra una comparación entre las características que ofrecen ambas compañías en las diferentes familias de FPGA que existen en el mercado.

Característica	Xilinx[21]	Altera[22]
Numero de bloques lógicos máximos	2,000,000	952,000
Pines E/S máximos	1,200	1,170
Componentes dedicados	• Bloques RAM • Unidades DSP • Controladores RAM • Transceiver	• Bloques RAM • Unidades DSP • Controladores RAM • Transceiver
Familias	• Spartan • Artix • Kintex • Virtex	• Stratix • Arria • Cyclone
Entorno de desarrollo (IDE)	• Vivado Design Suite • ISE Design Suite	• Quartus II
Microprocesadores	• Picoblaze • Microoblaze • Power PC • ARM Cortex-9	• Nios II

Tabla 4.4 Comparación entre algunas características de los FPGA de Xilinx y Altera.

Como se puede apreciar, ambas compañías ofrecen características muy similares entre sí. Para efectos de desarrollo e implementación del Sistema Embebido, se hace uso de un FPGA de la compañía Xilinx.

En la tabla 4.5 se realiza un comparativo de las principales características de tres tarjetas seleccionadas para su análisis.

Característica	Spartan 3A FPGA Starter Kit[23]	Avnet Spartan-6 LX9 MicroBoard[24]	Nexys3[24]
FPGA	Spartan 3A XC3S700A	Spartan-6 XC6LX9-2CSG324	Spartan-6 XC6LX16-CS324
Número de CLB's	1,472	9,152	14,579
Número de Slices	5,888	1,430	2,278
Bloques de RAM (Kb)	360	576	576
RAM Distribuida (Kb)	92	90	136
Slices DSP	-	16	32
Número de pines E/S	372	200	232
Recursos dedicados integrados dentro del FPGA	• Bloques memoria RAM • Registro de corrimiento • Multiplicadores dedicados (18 bits) con pipeline opcional • Lógica de acarreo rápido anticipado • DCM	• Bloques memoria RAM • Bloques de controlador de memoria integrados • Registros Duales • Slices DSP contienen multiplicadores 18 x 18 y un acumulador de 48 bits • CMT = DCM + PLL	• Bloques memoria RAM • Registros Duales • Slices DSP contienen multiplicadores 18 x 18 y un acumulador de 48 bits • CMT = DCM + PLL
Procesadores SOFT CORE soportados	• Microblaze • Picoblaze • Personalizados (Open Source)	• Microblaze • Picoblaze • Personalizados (Open Source)	• Microblaze • Picoblaze • Personalizados (Open Source)
Interfaces de Comunicación	• SPI • BPI • USB-JTAG	• Broad third-party SPI • NOR flash • BPI • USB-UART • USB-JTAG	• Broad third-party SPI • NOR flash • BPI • USB-UART • USB-JTAG
Capacidad Ethernet	SI	SI	SI
ADC incluido en la tarjeta	SI	NO	NO
Tipo de Conectores de Expansión para Pmods	6 pines 12 pines	6 pines 12 pines	6 pines 12 pines
Conectores de Expansión Pmods	4	2	4
Precio de venta con los fabricantes	189 dólares	89 dólares	119 dólares

Tabla 4.5 Comparativo de tres modelos de tarjetas de desarrollo con FPGA de la familia Spartan de Xilinx.

Se toman en consideración algunos puntos relevantes para el desarrollo del sistema embebido como son recursos dedicados integrados en el FPGA, herramientas y entornos de desarrollo, microprocesadores soportados y costos.

4.2.2. Simulación del ECG

Ya que no se cuenta con el módulo para realizar la adquisición de una señal real del corazón de un paciente, se debe optar por una fuente alterna que permita simular una señal real. Existen algunas opciones disponibles que permiten obtener simulaciones de señales cardiacas para realizar pruebas de dispositivos de monitoreo y para propósitos generales.

4.2.2.1. Comparación

Algunas características que se deben considerar para el análisis son el costo, experiencia de uso, documentación, acceso, por mencionar algunos. La generación se puede realizar mediante hardware especial como un generador de funciones, mediante software especial en una computadora o con una base de datos previamente construida.

Generador de funciones

Un generador de funciones es un dispositivo electrónico que es capaz de generar tanto señales analógicas como digitales. Normalmente se emplea en los laboratorios de electrónica para realizar pruebas de funcionamiento de dispositivos relacionados con uso de señales.

Actualmente se encuentran a disposición del público una enorme cantidad de modelos de generadores de funciones con una amplia variedad de características diferentes. Cada fabricante añade o restringe ciertas funcionalidades de acuerdo a las necesidades y el modelo del dispositivo. Casi todos los generadores de hoy en día, permiten una completa configuración de las funciones y de las señales que puede generar.

Existen generadores básicos que permiten generar solamente algunas señales de uso común como son de tipo senoidal, triangular, y cuadrada. En contraparte, también es posible encontrar generadores más avanzados, que son capaces de generar más tipos de señales, como son rampas, pulsos de PWM, señales arbitrarias, y personalizadas por el usuario. Inclusive cuentan con una interfaz de comunicación con la computadora para controlar su funcionamiento de manera más precisa.

El costo de los generadores está directamente relacionado con las capacidades y opciones que se brindan al usuario, así como la compañía que los proporciona, por lo que es muy variable de un generador a otro, pudiendo ir desde un precio que no supera los 10,000 pesos, hasta algunos que pueden alcanzar los 50,000 pesos.

Existen varias compañías que fabrican generadores de funciones con diferentes características. Una de ellas es la compañía Tektronix. Tektronix es una compañía que fabrica distintos tipos de instrumentos de medición y dispositivos electrónicos, entre los cuales están incluidos los generadores de funciones.

En la mayoría de las compañías es común que los distintos modelos sean agrupados en familias o series. Para el caso de los generadores de funciones, existe la familia AFG3000C.

En la tabla 4.6 se colocan las principales características proporcionadas por el sitio web de Tektronix. Las características son mostradas en rangos mínimo y máximo, del generador más básico al más completo de la serie [25].

Serie	AFG3000C
Canales analógicos	1 – 2
Velocidad de muestras	250 MS/s – 2 GS/s
Ancho de banda	10 MHz – 240 MHz
Longitud de grabaciones	128, 000 puntos
Resolución vertical	14 bits
Frecuencia salida	10 MHz – 240 MHz
Precio	1, 890 dólares

Tabla 4.6 Características principales de los generadores de funciones de la familia AFG3000C de Tektronix.

De las características que se mencionan, las más relevantes a considerar para saber si son de utilidad al proyecto son la velocidad de las muestras, la frecuencia de salida y el ancho de banda. Es necesario verificar que cubre los requerimientos obtenidos de las características de un ECG en cuanto al ancho de banda principalmente para asegurar que sea capaz de simular la señal.

Software especial

Otra alternativa para simular un ECG es con la ayuda de un software que, mediante funciones matemáticas, produzca la señal requerida. El software puede ser propiedad de alguna empresa, o elaborado de manera personalizada por un programador.

Algunas veces, los programas comerciales no cuentan con las funciones específicas para generar ciertas señales. Sin embargo, es posible crear funciones personalizadas que logren realizar la tarea en base a funciones más elementales que sí estén incluidas dentro el software.

Un ejemplo de este tipo de programas, es el conjunto de aplicaciones MATLAB. A pesar de que MATLAB no es un software propiamente dedicado a la generación de señales, es posible crear funciones que permitan hacerlo.

MATLAB (MATrix LABoratory) es un lenguaje de alto nivel y entorno interactivo para cálculos numéricos, visualizaciones y programación. Se puede usar MATLAB para una serie de aplicaciones incluyendo procesamiento de señales y comunicaciones, procesamiento de imagen y video, control de sistemas, pruebas y mediciones, finanzas computacionales y biología computacional[26].

De forma predeterminada, MATLAB no incluye funciones que permitan realizar la simulación de un ECG. Para poder hacerlo, es necesario crear de manera específica, funciones o métodos en el lenguaje propio de la herramienta.

Una opción que se encuentra disponible para ser usada en conjunto con el software de MATLAB, es "ECG waveform generator for Matlab/Octave"[27], que se encuentra disponible en el sitio web PhysioNet. Fue desarrollado por Floyd Harriott de Stellate Systems. El recurso consta de dos archivos. El archivo ECGwaveGen.m genera una señal ECG artificial con parámetros configurables como duración de la señal, frecuencia de muestreo, amplitud. El archivo QRSpulse.m crea cambios prematuros seguidos de pausas de compensación.

La aplicación no está diseñada para ser altamente realista. Su función principal es probar la fidelidad de procesamiento de señales analógicas de componentes de monitores cardiacos e instrumentos similares.

Base de datos pre-construida

Una tercera opción para obtener una simulación de un ECG es mediante una base de datos existente que contenga registros almacenados de una señal real o artificial. Los datos contenidos dentro de la base de datos pueden ser muy variados, dependiendo de las especificaciones sobre la cual haya sido construida la base de datos.

En el caso particular de las bases de datos, es necesario considerar los detales de la señal que esta almacenada como son: la frecuencia a la que fue muestreada, la resolución de bits, los componentes visibles en la señal, factores que modifican la señal (ruido de la señal), alteraciones patológicas (arritmia, taquicardia, QT largo), y algunos otros.

En el presente trabajo de tesis, se requiere un ECG de características normales, es decir, de un paciente sano. Adicionalmente, es necesario que contenga al menos el complejo QRS para poder realizar el proceso de identificación y aislamiento del mismo.

Una de las fuentes más usadas para obtener bases de datos de muchos tipos es el MIT (Massachusetts Institute of Technology o Instituto Tecnológico de Massachusetts) en los Estados Unidos de América. El instituto cuenta con un gran conjunto de bases de datos con registros de ECG tanto reales como simulados.

Las bases de datos que proporciona se encuentran clasificadas de acuerdo a patologías, usuarios, duración de los registros y trabajos relacionados con las mismas. Los archivos que contienen la información de las bases de datos pueden ser encontrados en el sitio web de PhysioNet [28].

Los archivos se encuentran en formato de datos (.dat), lo que dificulta la visualización de su contenido si no se cuenta con el software apropiado. Sin embargo, dentro del mismo sitio web, se proporciona un conjunto de herramientas que ayudan a la visualización, graficación y conversión de los archivos.

Cada base de datos (o la gran mayoría de ellas) contiene documentación relacionada con la creación de la base de datos, la forma en la que se almacenó la información, posibles errores, y algunos posibles usos.

Capítulo 5

Diseño

5.1. Diseño del sistema

Una vez establecidos todos los requerimientos, y habiendo comparado los dispositivos que cumplen con dichos requisitos, se procede a realizar el diseño de la arquitectura y la funcionalidad. En el diseño deben estar incluidos los diagramas generales y detallados del sistema embebido.

Como primer paso, se definen los diagramas de bloques generales del sistema, donde se pueda apreciar tanto la entrada como la salida de la información. Posteriormente se deben elaborar diagramas detallados de cada módulo si es necesario.

5.1.1. Diseño general del sistema

El primer paso consiste en establecer las etapas que forman parte del Sistema Embebido para saber la ruta por la cual fluyen los datos, desde la adquisición, hasta el envío de la información. En la figura 5.1 se muestran un diagrama general a bloques.

Figura 5.1 Diagrama general a bloques de las etapas del sistema.

A continuación se describe a detalle las características y la forma de desarrollar cada una de las tres etapas que comprenden el procesamiento de la información dentro del Sistema Embebido

5.1.1.1. Simulación de la señal ECG

Para la primera etapa, se retoman las posibles fuentes de datos analizados con anterioridad. Cada una de ellas tiene como propósito, proporcionar la señal ECG que permita verificar que el Sistema Embebido puede reconocer y aislar los complejos QRS mediante el algoritmo de correlación cruzada propuesto para dicho fin.

5.1.1.1.1. Selección

Considerando las tres alternativas propuestas como medio de obtención de la señal, se debe hacer la selección de la mejor alternativa para el proyecto. Para poder determinarlo, hay que comparar las características e información proporcionada por cada fuente, con las características de una señal ideal.

En la tabla 5.1 se realiza un comparativo entre las tres alternativas de solución. Se incluyen los puntos considerados como relevantes en cuanto a las características propias de cada fuente.

Característica	Generador de funciones	Software especial	Base de datos pre-construida
Capacidad para simular un ECG	SI	SI	SI
Modificación de parámetros	SI	SI	NO
Requiere hardware adicional	SI	SI	NO
Documentación disponible	SI	SI	SI
Compatible con el sistema	SI	SI	SI
Precio	1, 890 dólares	Gratuito	Gratuito

Tabla 5.1 Comparación entre tres diferentes métodos de simulación de ECG.

Como se puede observar, los tres métodos que aparecen en la tabla cumplen en casi todos los aspectos considerados para realizar la selección. Sin embargo, hay una diferencia muy marcada en el rubro del precio. Mientras que el generador de funciones tiene un costo relativamente alto, las otras dos opciones son gratuitas. De éstas dos últimas alternativas, ambas son elegibles para ser usadas con el sistema.

Una de las principales ventajas de utilizar una base de datos, es poder obtener directamente los datos en un formato digital. Las herramientas proporcionadas en el mismo sitio web donde se encuentran las bases de datos, permiten obtener los datos en el formato que se requiere de acuerdo a las necesidades del sistema.

Dada su facilidad de convertir los datos al formato deseado, se opta por utilizar una base de datos pre-construida. Esto es porque, auxiliándose de las herramientas proporcionadas, es posible obtener los datos de la base de datos directamente a un formato digital, lo que elimina el uso de hardware adicional.

La base de datos seleccionada para obtener los datos del ECG es la base de datos llamada QT, ya que esta base de datos cuenta con registros que incluyen el complejo QRS dentro de los ECG que contiene.

5.1.1.1.2. Base de Datos QT

La base de datos QT contiene un total de 105 fragmentos de quince minutos de dos canales de ECG, seleccionados para evitar fluctuaciones significativas en la línea base u otros artefactos.

La tabla 5.2 muestra los orígenes de los datos así como la clasificación dentro de la base de datos[29].

MIT-BIH	MIT-BIH	MIT-BIH	MIT-BIH	ESC	MIT-BIH	Sudden
Arrhyt.	ST DB	Sup. Vent.	Long Term	STT	NSR DB	Death
15	6	13	4	33	10	24

Tabla 5.2 Distribución de los 105 registros de acuerdo con la base de datos original.

De los 7 conjuntos disponibles, la más adecuada para este trabajo es el NSR DB (Normal Sinus Rhythm Database o Base de Datos de Ritmo Sinusal Normal) ya que sus registros no presentan anomalías y la señal es estable.

En la tabla 5.3 se muestra los registros correspondientes al subconjunto NSR DB el cual contiene 10 registros en total de ECG [29].

Registros de la Base de Datos MIT-BIH Ritmo Sinusal Normal			
Registro	Intervalo original tiempo	notas	Patrón forma de onda
		marca	(p) (QRS) (t) (u)
sel16265	10:37:30 → 10:52:30	30	(p) (N) t)
sel16272	11:13:10 → 11:28:10	30	(p) (N) t)
sel16273	11:22:36.5 → 11:37:36.5	30	(p) (N) t)
sel16420	14:24:00 → 14:39:00	30	(p) (N) t)
sel16483	13:32:00 → 13:47:00	30	(p) (N) t)
sel16539	20:56:00 → 21:11:00	30	(p) (N) (t)
sel16773	10:23:40 → 10:38:40	30	(p) (N) t) u)
sel16786	18:22:00 → 18:37:00	30	(p) (N) (t)
sel16795	15:00:00 → 15:15:00	30	(p) (N) (t)
sel17453	14:15:00 → 14:30:00	30	(p) (N) (t) u)
10	2:30:00	300	

Tabla 5.3 Descripción de los 10 registros contenidos en la sección NSR DB de la base de datos QT

En cuanto a especificaciones técnicas, cada señal ECG almacenada en cada uno de los 10 registros se encuentra muestreada a 250Hz a 12 bits de resolución. La duración de todos ellos es de 15 minutos aproximadamente con ritmo constante. De los 10 registros contenidos, se elige el número sel16539, ya que su forma de onda se ajusta muy bien a las necesidades del proyecto.

En la figura 5.2 se muestra un fragmento de los primeros 10 segundos de la señal correspondiente a dicho registro, donde se puede apreciar el ECG generado con los datos almacenados.

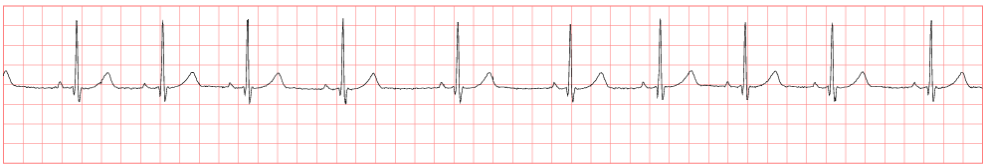


Figura 5.2 Grafica de la señal ECG del registro sel16539.

Para obtener el número de muestras en 10 segundos de la señal, se multiplica la frecuencia de muestreo por el número de segundos, es decir:

$$Número\ total\ de\ muestras = (250\ Hz)(10\ segundos)$$

$$Número\ total\ de\ muestras = 2,500\ muestras$$

Es decir, se requiere que la memoria tenga capacidad para almacenar al menos 2, 500 muestras. Como las memorias tienen capacidades preestablecidas, en este caso en particular, es necesaria una memoria ROM de 4KB que puede almacenar hasta un total de 4096 datos de muestras.

Dado que la memoria no es muy grande en cuanto a capacidad de almacenamiento, puede ser implementada dentro del FPGA como un módulo interno y conectarlo con los demás módulos cómo es el core de procesamiento o el módulo de comunicación UART.

5.1.1.2. Identificación de los complejos QRS

Ya habiendo elegido la fuente de la cual se obtendrán las señales ECG, el siguiente paso es comenzar a diseñar la arquitectura del Sistema Embebido, para lo cual es necesario revisar los recursos disponibles con los que cuenta el FPGA a usar para alojar el sistema.

El FPGA es el contenedor del Sistema Embebido, donde se encuentran todos los módulos que realizan la tarea de procesamiento y envío de datos a internet. El FPGA debe incluir los requerimientos básicos de un Sistema Embebido como son el microprocesador, buses de conexión, bloque de memoria y periféricos.

Como se trata de un Sistema Embebido de tipo SoPC, la implementación se puede realizar parcial o enteramente mediante un lenguaje de descripción de hardware como puede ser VHDL o Verilog.

En la figura 5.3 se muestra el interior del FPGA con los módulos que son necesarios implementar. Cabe destacar que sigue siendo una vista general del sistema, así que aún no se incluyen detalles específicos de cada componente.

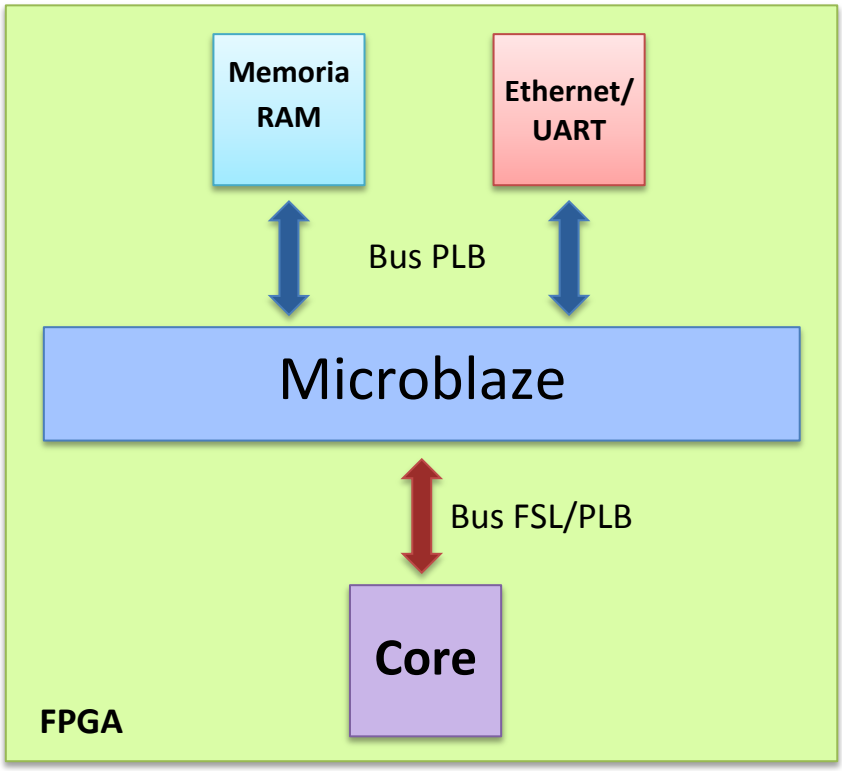


Figura 5.3 Diagrama general de los módulos dentro del FPGA.

Como se puede observar, Microblaze es el encargado de gestionar todo el control de los periféricos en el FPGA. Es por este motivo que no existe una conexión directa entre los dispositivos, y la comunicación por medio de los buses debe ser bidireccional. Sin embargo, el flujo de la información sí existe de un módulo a otro.

De la misma manera, también se pueden distinguir dos tipos de buses diferentes, el bus FSL y los buses PLB. El bus FSL es un bus de alta velocidad que tiene como característica distintiva, un retardo de propagación de apenas unos cuantos picosegundos. Es común su uso en IP cores de procesamiento de datos que requieren alta velocidad de comunicación.

Microblaze

Microblaze es un procesador de 32 bits de tipo soft-core con una arquitectura RISC Harvard. Al ser un procesador de tipo soft-core esta implementado enteramente con un lenguaje de descripción de hardware. Microblaze es el encargado de controlar todos los periféricos asociados a él, así como la transmisión de la información entre los mismos.

Microblaze posee muchas características y opciones configurables. Algunas de ellas se enlistan a continuación[30]:

- Cuenta con unidad de punto flotante (FPU)
- Precisión simple
- Unidad de administración de memoria (MMU)
- Caché de instrucciones y datos
- Tamaño de caché configurable (2kB – 64kB)
- Excepciones de división por cero, punto flotante, MMU y FSL

Microblaze soporta buses de alta y baja velocidad. Los tipos de buses que se pueden usar en conjunto con Microblaze son PLB, FSL y Axis.

Core

El IP Core es el segundo bloque de mayor importancia en el Sistema Embebido. Es el encargado de identificar y aislar los complejos QRS presentes en la señal ECG. Para este sistema en particular, los datos son leídos desde una memoria, donde después de pasar al core, se procesan para obtener el componente QRS de la señal. El resultado se almacena en otro bloque de memoria para después ser enviado de forma remota mediante alguna de las interfaces de comunicación disponibles.

Dentro de ese modulo se encuentra implementado el algoritmo de procesamiento, en este caso se utiliza un algoritmo de correlación cruzada para obtener el complejo QRS de la señal ECG.

Memoria RAM

El bloque de memoria RAM es utilizado para sintetizar tanto el bloque de memoria de datos como el bloque de memoria de instrucciones. Se colocan de manera externa, ya que los bloques de memoria dedicados que se encuentran dentro del FPGA son limitados, y no son suficientes para almacenar todos los datos que se requieren.

Ethernet / UART

El modulo Ethernet / UART se encarga de gestionar el envío de los resultados obtenidos del core por medio de alguna de las dos interfaces. Para el caso de la interfaz Ethernet se requiere que la tarjeta tenga un conector de red como puede ser un RJ-45, además de soportar algún protocolo de comunicación para internet, como por ejemplo TCP/IP.

Si se desea hacer uso de la interfaz UART, la tarjeta debe poseer, ya sea un conector de tipo serial, o un conector que permita usar un cable adaptador serie-USB. También debe poder soportar un protocolo de comunicación serial, como puede ser el RS-232.

Bus PLB

El bus PLB (Processor Local Bus) es un bus de 128 bits que proporciona la infraestructura para la conexión de un número opcional de maestros o esclavos dentro del sistema. Cosiste de una unidad de control del bus, un timer tipo watchdog, unidades de vías de lectura y escritura separadas,y un DCR (Device Control Register) esclavo para obtener acceso a los registros de estado de error. Cuenta con las siguientes características[31]:

- Soporte 128, 64 y 32 bits para maestros y esclavos
- Pipeline (en configuración de bus compartido o punto a punto)
- Arquitectura PLB compatible
- Circuito de reinicio PLB

Bus FSL

El bus FSL (Fast Simplex Link) es un bus unidireccional con canal de comunicación punto a punto entre dos elementos del FPGA implementados con una interfaz FSL. La interfaz transfiere datos de y hacia el archivo de registros del procesador al hardware ejecutándose en el FPGA. sus principales características son[32]:

- Implementa una comunicación unidireccional punto a punto basada en FIFO
- Provee mecanismos para comunicación no compartida y sin arbitraje para rápida comunicación entre maestro y esclavo.
- Provee un bit extra de control para comentar datos antes de comenzar.
- La profundidad de la FIFO puede ir desde 1 hasta 8K.
- Soporta modo FIFO síncrono y asíncrono.

Envío y visualización de resultados

Cuando se termina de procesar la señal, los resultados obtenidos son transferidos a una computadora de manera serial, por medio de la interfaz UART y el cable adaptador serie-USB. Los datos son almacenados en la computadora, donde se tiene implementado un servidor que, por medio de comunicación tipo socket, se encuentra a la espera de conexión de clientes para enviar la información de forma remota.

Cuando el cliente establece la comunicación, el servidor envía los datos hacia el cliente. Una vez recibidos, los datos son graficados para que puedan ser visualizados de manera más clara y sean interpretados por el especialista médico.

5.1.2. Arquitectura del sistema

Después de colocar los módulos necesarios, se procede a crear la estructura del sistema. En la arquitectura se incluyen los detalles específicos de los componentes, así como la conexión y flujo de datos a lo largo de todo el sistema.

5.1.2.1. Diagrama de flujo de datos

Después de haber escogido el algoritmo más adecuado para realizar el procesamiento de la información, el primer paso es crear los diagramas que ayuden a implementar e interconectar cada uno de los módulos propuestos de manera correcta.

En la figura 5.4 se muestra el diagrama de flujo de datos del algoritmo de correlación cruzada, que permite procesar los datos para poder hacer la detección y separación del complejo QRS de la señal digital de prueba que se tiene almacenada en una de las memorias ROM.

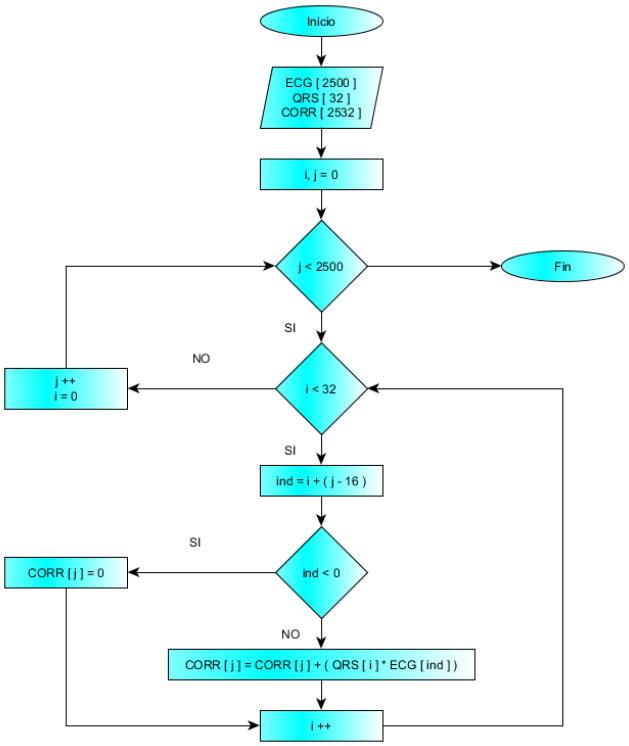


Figura 5.4 Diagrama de flujo del procesamiento de la señal mediante correlación cruzada.

El diagrama de flujo muestra la representación del algoritmo de correlación cruzada aplicado al caso particular de detectar la presencia del complejo QRS en la señal ECG. A continuación se describen las señales involucradas, así como los procesos que permiten realizar los cálculos necesarios para el algoritmo.

Datos de entrada

- Vector con los valores del ECG de prueba (2500 muestras): ECG
- Vector con el patrón del complejo QRS (32 muestras): QRS
- Variables enteras i, j, ind

Datos de salida

- Vector con los resultados de la correlación (2532 resultados): CORR

Proceso

1. Inicio
2. Inicializar i , j a 0
3. Mientras j es menor a 2500 hacer,
4. Mientras i es menor a 32 hacer,
5. Asignar valor a ind con la fórmula i + (j – 16)
6. Si ind es menor a 0 entonces,
7. Colocar el valor del vector CORR en la posición j a 0,
8. Incrementar i
9. En caso contrario
10. Calcular los valores de CORR en la posición j multiplicando el valor del vector QRS en la posición i, con el valor del vector ECG en la posición ind, y sumarlo al valor del vector CORR en la posición j
11. Incrementar i
12. Fin-mientras
13. Incrementar j
14. Inicializar i a 0
15. Fin-mientras
16. Fin

5.1.2.2. Carta ASM

Una vez obtenido el diagrama de flujo del algoritmo, el siguiente paso es obtener la carta ASM que permita comenzar a implementar cada uno de los módulos necesarios dentro del FPGA.

Las cartas ASM (Algorithmic State Machine o Máquinas de Estado Algorítmicas) es un tipo especial de diagrama de flujo que puede emplearse para representar las transiciones de estado y las salidas generadas por una máquina de estados. Están especialmente enfocadas a representar algoritmos de tipo secuencial.

A diferencia de los diagramas de flujo comunes, las cartas ASM tienen características especiales para poder representar y desarrollar algoritmos en hardware. Dichas características son:

- Poseen una señal de control de inicio para comenzar con la ejecución del algoritmo
- El cambio entre los estado es controlado por una señal de reloj
- Están formadas por 3 elementos básicos: cajas de estado, donde se establecen las acciones que se pueden realizar durante un mismo ciclo de reloj; las cajas de decisión, donde se pregunta si cierta condición de entrada es verdadera o falsa; y las cajas de acción condicional, que contiene las acciones a realizar dependiendo si cierta condición se cumple o no y siempre se encuentran a continuación de cajas de decisión.

Agregando la señal de inicio, y transformando los estados normales en el tipo correspondiente de elemento básico, se puede formar la carta ASM equivalente con el diagrama de flujo original.

La figura 5.5 muestra la carta ASM obtenida a partir del diagrama de flujo original. Contiene el mismo algoritmo, pero al ser una carta ASM, permite realizar la implementación en un sistema de hardware que cuente con los requisitos básicos de una señal de reloj y una señal de inicio que pueda ser activada.

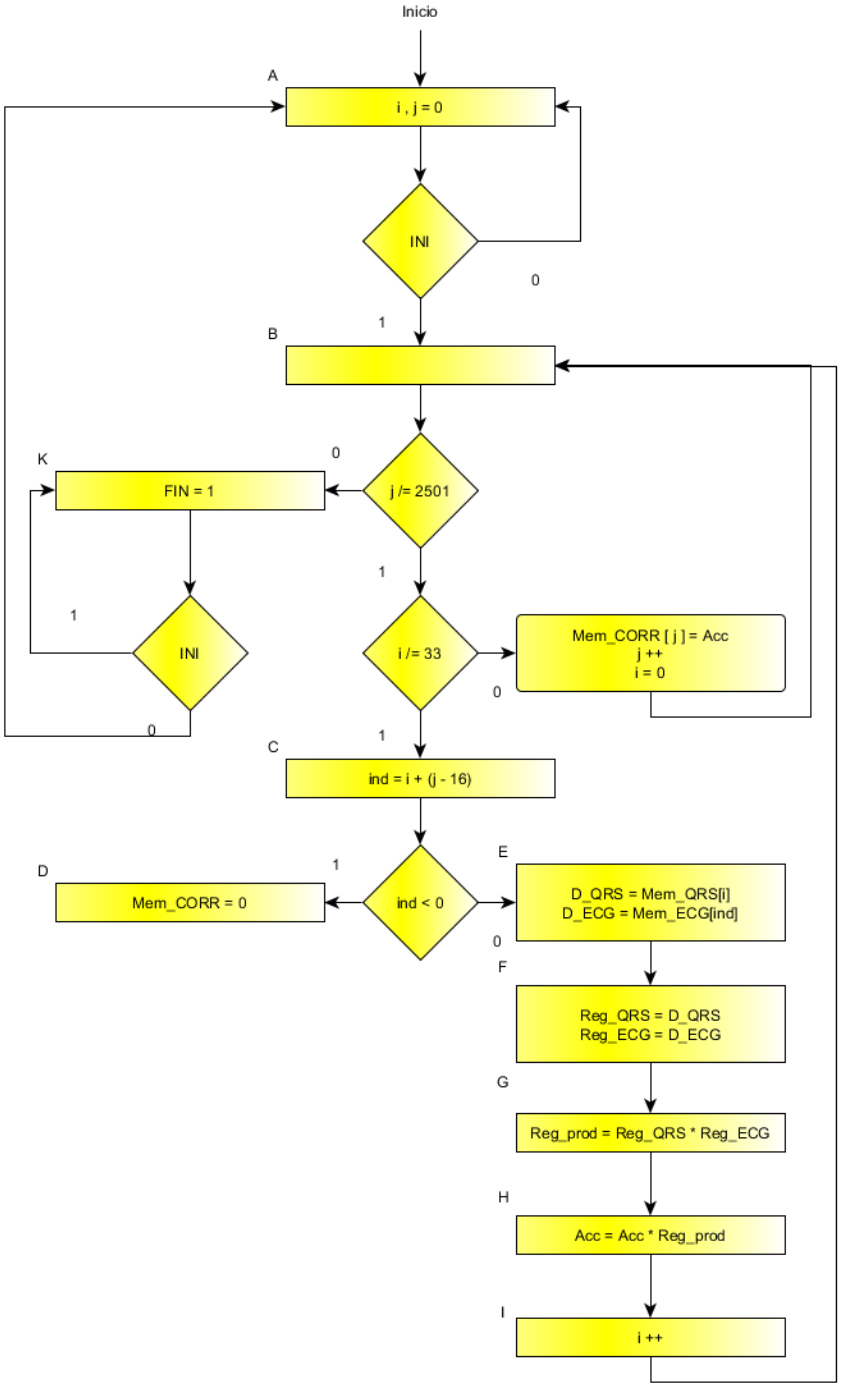


Figura 5.5 Carta ASM obtenida a partir del diagrama de flujo del algoritmo anterior.

Las diferencias apreciables a primera vista son principalmente dos: la inclusión de la señal de inicio para comenzar el procesamiento, y el desglose de todos los módulos involucrados en el procesamiento propio del algoritmo de correlación cruzada.

A continuación se describen los estados contenidos en la carta ASM. De manera implícita, existe una señal de reloj que permite controlar el cambio entre los estados. Asimismo cada bloque de proceso ahora se convierte en un estado.

Inicio

Estado A:

Se inicializan las variables i, j a 0

Se comprueba la señal INI

Si INI es igual a 0 entonces, se conserva el estado A

Si INI es igual a 1 entonces, se pasa al estado B

Estado B:

Si j es diferente a 2500 entonces, se comprueba i

Si j es igual a 2500 entonces, se pasa al estado K

Si i es diferente de 32 entonces, se pasa al estado C

Si i es igual a 32 entonces,

Se almacena el valor de la correlación en la memoria de resultados en la posición dada por la variable j

Se incrementa la variable j

Se inicializa la variable i a 0

Estado C:

Se asigna valor a la variable ind con la formula i + (j - 16)

Si ind es menor a 0 entonces, se pasa al estado D

Si ind es mayor a 0 entonces, se pasa al estado E

Estado D:

El valor en la memoria Mem_CORR, en la posición apuntada por la variable j, se asigna a 0

Estado E:

Se lee el valor contenido en las memorias Mem_ECG y Mem_QRS en las posiciones dadas por las variables ind , i respectivamente

Estado F:

Se cargan los registros Reg_ECG y Reg_QRS con los valores obtenidos de las memorias Mem_ECG y Mem_QRS

Estado G:

Se carga el registro Reg_Prod con la multiplicaion de los valores contenidos en los registros Reg_ECG y Reg_QRS

Estado H:

Cargar el registro ACC con el valor ya presente en el acumulador, mas, el valor contenido en el registro Reg_Prod

Estado I:

Se incrementa la variable i

Estado K:

Se coloca la señal FIN a un valor de 1

Si la señal INI es igual a 1 entonces, se conserva el estado K

Si la señal INI es igual a 0 entonces, se pasa al estado A

.

5.1.2.3. Diagrama de ruta de datos

Posterior al diseño y obtención de la carta ASM, se procede con el diseño del diagrama de ruta de datos. El diagrama de ruta de datos muestra de manera gráfica cada uno de los bloques que conforman el sistema, la interconexión que existe entre cada uno de ellos, la cantidad de información a la entrada y salida de los mismos, la dirección en la que fluye la información y la forma de controlar dicho flujo mediante señales de control.

Las señales de control, como su nombre lo indica, permiten controlar el flujo de información dentro del sistema. Entre las principales funciones de control se encuentran la carga de datos, habilitación de los módulos, selectores, lectura/escritura, entre otras.

El orden en el que se activan o desactivan estas señales, es establecido en un bloque especial dentro del diseño llamado Unidad de Control. Dentro de la Unidad de Control, se establece el mecanismo para lograr que la información que entra al sistema sea procesada y se obtenga el resultado a la salida del mismo.

La figura 5.6 muestra el diagrama con los bloques que conforman el sistema de procesamiento del ECG. En el diseño se pueden observar tanto bloques secuenciales para carga y retención de datos, como bloques combinatorios para realizar operaciones lógicas y aritméticas.

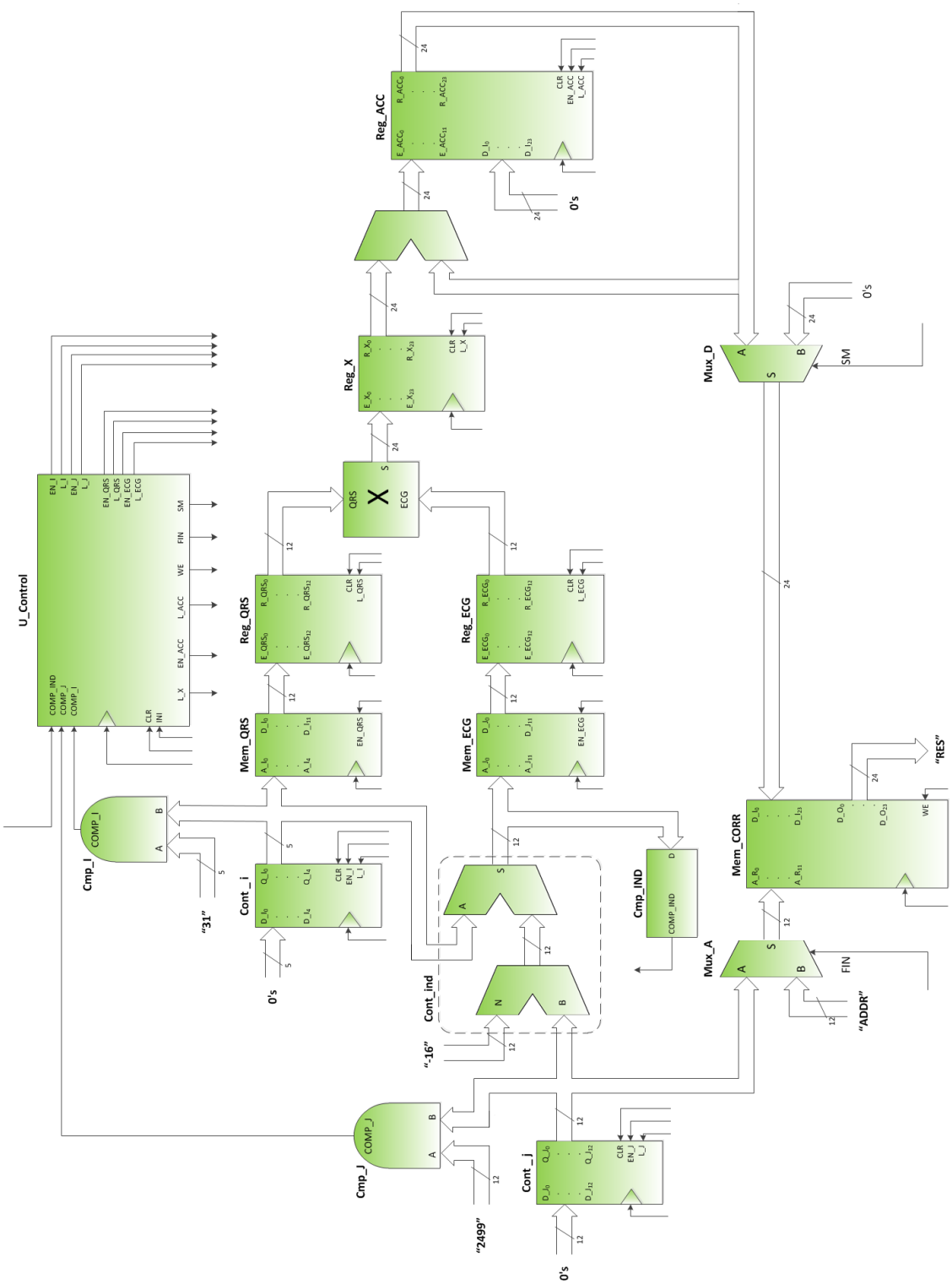


Figura 5.6 Diagrama de flujo de datos del sistema en bloques.

En el diagrama se observan bloques tales como contadores, memorias, registros, multiplexores, ente otros. La información fluye de izquierda a derecha a lo largo de todo el sistema, donde es procesada en los diferentes bloques. El origen de los datos proviene de las memorias MEM_QRS y MEM_ECG, donde se encuentran las muestras a procesar.

La secuencia en el que los datos son procesados desde el origen hasta la salida es:

1. Los datos salen de las memorias MEM_QRS y MEM_ECG de la posición o índice dado por el contador CONT_I, y por el bloque auxiliar CONT_IND respectivamente.

2. Los datos obtenidos de las memorias son almacenados en los registros REG_QRS Y REG_ECG correspondientes.

3. Los datos salen de ambos registros hacia el bloque aritmético de multiplicación, donde una vez realizada la operación, el resultado es enviado hacia el registro de resultado de multiplicación REG_X.

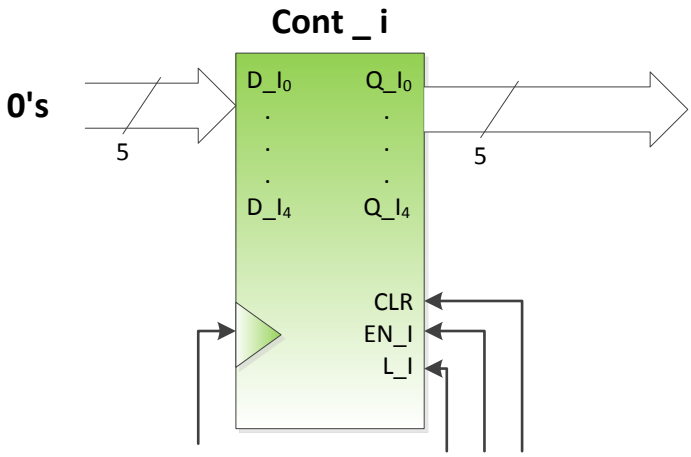
4. En el siguiente bloque se comprueba la señal COMP_I de una de las compuertas AND que están contenidas en el diseño, con el fin de indicarle al sistema, si el resultado debe acumularse (sumar el valor de REG_X con el valor del registro acumulador REG_ACC), o si debe escribirse en memoria como resultado final de la operación.

5. Una vez calculado el resultado final, éste se escribe en la memoria MEM_CORR, en la posición de memoria obtenida del contador CONT_J.

El proceso se repite hasta que se procesan todos los datos contenidos en la memoria MEM_ECG y los resultados se encuentren almacenados en la memoria MEM_CORR.

A continuación se describe de manera más detallada cada uno de los bloques que conforman el sistema, especificando las señales que lo conforman, la información que entra y sale, así como la función u operación que realizan para procesar la información.

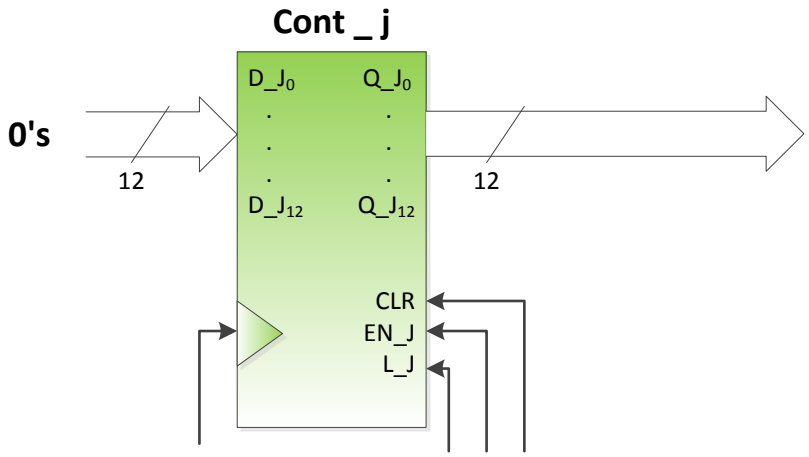
5.1.2.3.1. CONT_I



Nombre	CONT_I
Tipo de bloque	Contador ascendente de 5 bits
Funcionamiento	Secuencial
Señal de datos de entrada	D
Bloque del que provienen los datos	-
Señal de datos de salida	Q
Bloque al que se envían los datos	MEM_QRS
Señales de control	CLK: Proporciona la frecuencia de trabajo del bloque.
	CLR: Coloca el valor del bloque a su valor inicial, generalmente a cero.
	EN_I: Activa el bloque para que el valor pueda salir por el bus de salida.
	L_I: permite cargar un valor al bloque mediante el bus de entrada.
Descripción	La función del CONT_I es generar un conteo de 0 a 32, los cuales son utilizados como posiciones de memoria en el bloque MEM_QRS.

Tabla 5.4 Descripción del bloque CONT_I.

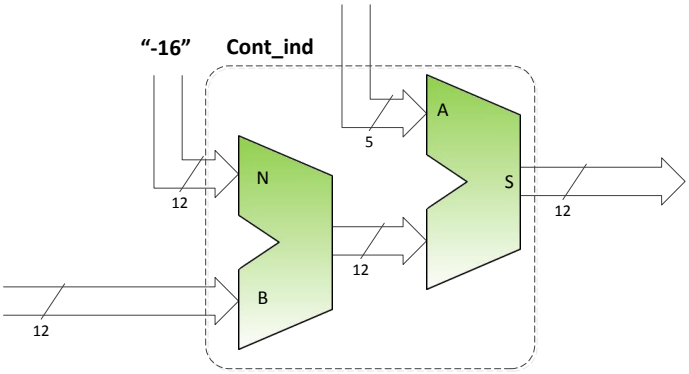
5.1.2.3.2. CONT_J



Nombre	CONT_J
Tipo de bloque	Contador ascendente de 12 bits
Funcionamiento	Secuencial
Señal de datos de entrada	D
Bloque del que provienen los datos	-
Señal de datos de salida	Q
Bloque al que se envían los datos	CONT_IND
Señales de control	CLK: Proporciona la frecuencia de trabajo del bloque.
	CLR: Coloca el valor del bloque a su valor inicial, generalmente a cero.
	EN_J: Activa el bloque para que el valor pueda salir por el bus de salida.
	L_J: permite cargar un valor al bloque mediante el bus de entrada.
Descripción	La función del CONT_J es generar un conteo de 0 a 4096, donde los primeros 2500 valores son utilizados como posiciones de memoria en el bloque MEM_QRS.

Tabla 5.5 Descripción del bloque CONT_J.

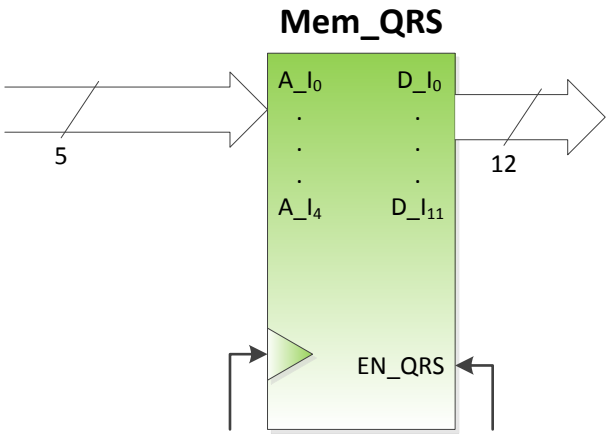
5.1.2.3.3. CONT_IND



Nombre	CONT_IND
Tipo de bloque	Generador de valores consecutivos
Funcionamiento	Combinatorio
Señal de datos de entrada	A
	B
	N
Bloque del que provienen los datos	Para A: CONT_I
	Para B: CONT_J
	Para N: -
Señal de datos de salida	S
Bloque al que se envían los datos	MEM_ECG
Señales de control	-
Descripción	El bloque contiene internamente de dos ALU's que generan los números por medio de dos pasos: En la primera ALU se hace una operación de suma del valor proveniente de la señal de entrada B con un valor constante de -16. El resultado es enviado a la segunda ALU, donde se realiza una segunda operación de suma del resultado con el valor de la señal de entrada A.

Tabla 5.6 Descripción del bloque CONT_IND.

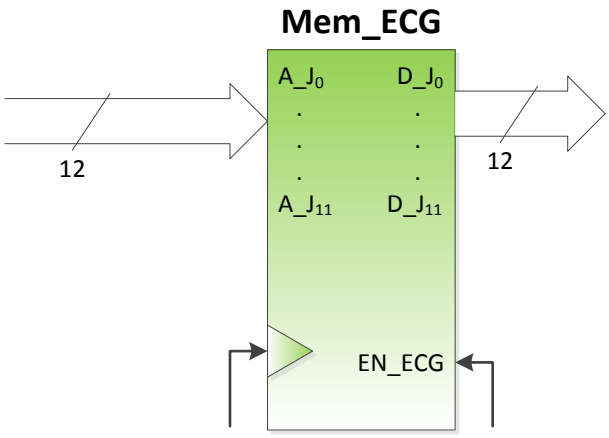
5.1.2.3.4. MEM_QRS



Nombre	MEM_QRS
Tipo de bloque	Memoria ROM de 32x12 (48 bytes)
Funcionamiento	Síncrono
Señal de datos de entrada	A
Bloque del que provienen los datos	CONT_I
Señal de datos de salida	D
Bloque al que se envían los datos	REG_QRS
Señales de control	CLK: Proporciona la frecuencia de trabajo del bloque.
	EN_QRS: Activa el bloque para que el valor pueda salir por el bus de salida.
Descripción	En el bloque de memoria MEM_QRS se encuentran almacenados los 32 valores de muestras que conforman el patrón QRS a buscar en la señal ECG. Asimismo, cada valor tiene una longitud de 12 bits.

Tabla 5.7 Descripción del bloque MEM_QRS.

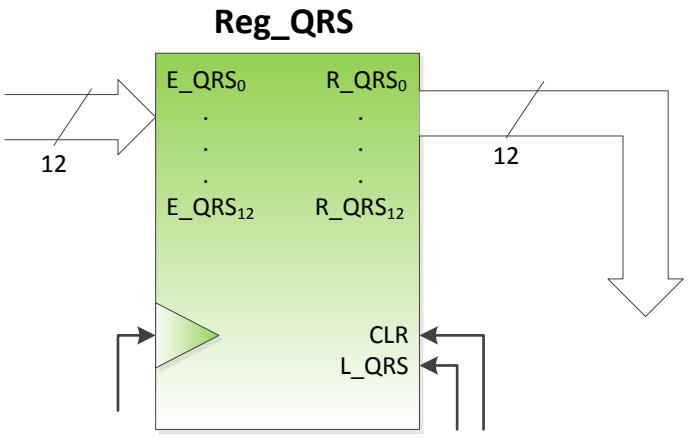
5.1.2.3.5. MEM_ECG



Nombre	MEM_ECG
Tipo de bloque	Memoria ROM de 4096x12 (6KB)
Funcionamiento	Síncrono
Señal de datos de entrada	A
Bloque del que provienen los datos	CONT_IND
Señal de datos de salida	D
Bloque al que se envían los datos	REG_ECG
Señales de control	CLK: Proporciona la frecuencia de trabajo del bloque.
	EN_ECG: Activa el bloque para que el valor pueda salir por el bus de salida.
Descripción	En el bloque de memoria MEM_ECG se encuentran almacenados los 2500 valores de muestras que conforman la señal ECG a analizar. Los valores se almacenan de la posición 0 a la 2499; las posiciones de 2500 a 4095 se rellenan con valores de ceros.

Tabla 5.8 Descripción del bloque MEM_ECG.

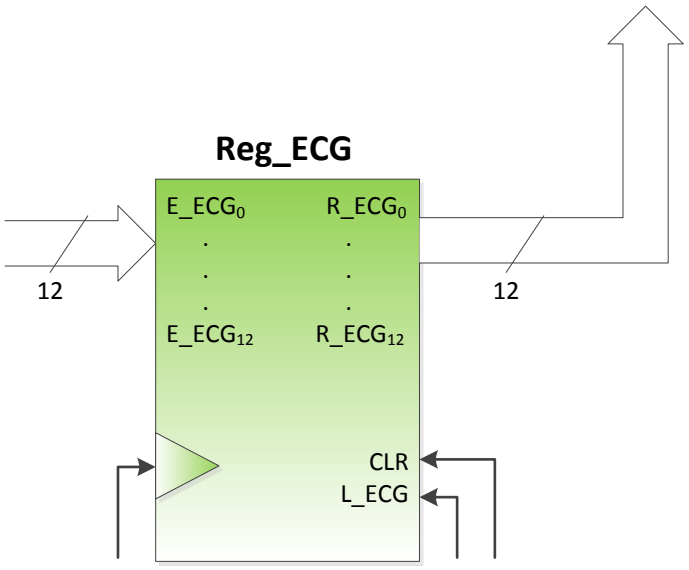
5.1.2.3.6. REG_QRS



Nombre	REG_QRS
Tipo de bloque	Registro de 12 bits
Funcionamiento	Secuencial
Señal de datos de entrada	E_QRS
Bloque del que provienen los datos	MEM_QRS
Señal de datos de salida	R_QRS
Bloque al que se envían los datos	X
Señales de control	CLK: Proporciona la frecuencia de trabajo del bloque.
	CLR: Coloca el valor del bloque a su valor inicial, generalmente a cero.
	L_QRS: permite cargar un valor al bloque mediante el bus de entrada.
Descripción	El registro REG_QRS almacena el valor proveniente de la memoria MEM_QRS.

Tabla 5.9 Descripción del bloque REG_QRS.

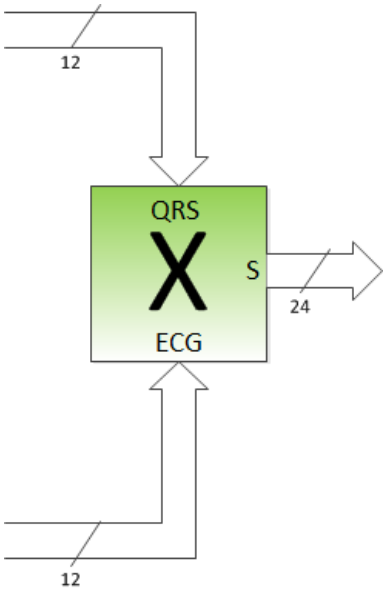
5.1.2.3.7. REG_ECG



Nombre	REG_ECG
Tipo de bloque	Registro de 12 bits
Funcionamiento	Secuencial
Señal de datos de entrada	E_ECG
Bloque del que provienen los datos	MEM_ECG
Señal de datos de salida	R_ECG
Bloque al que se envían los datos	X
Señales de control	CLK: Proporciona la frecuencia de trabajo del bloque.
	CLR: Coloca el valor del bloque a su valor inicial, generalmente a cero.
	L_ECG: permite cargar un valor al bloque mediante el bus de entrada.
Descripción	El registro REG_ECG almacena el valor proveniente de la memoria MEM_ECG.

Tabla 5.10 Descripción del bloque REG_ECG.

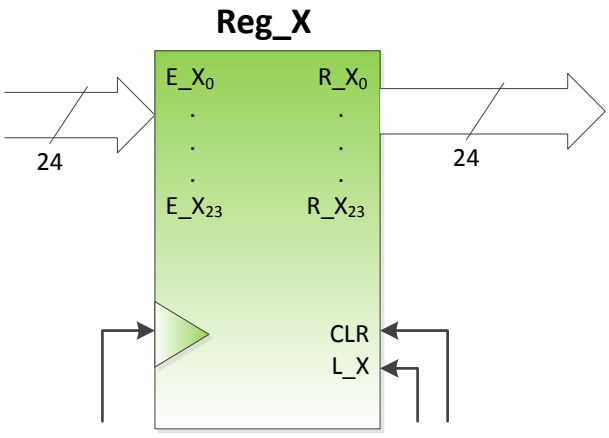
5.1.2.3.8. X



Nombre	X
Tipo de bloque	Unidad multiplicadora de 12x12
Funcionamiento	Combinatorio
Señal de datos de entrada	QRS
	ECG
Bloque del que provienen los datos	Para QRS: REG_QRS
	Para ECG: REG_ECG
Señal de datos de salida	S
Bloque al que se envían los datos	REG_X
Señales de control	-
Descripción	El bloque X realiza una operación de multiplicación sobre los valores obtenidos de los registros REG_QRS y REG_ECG de 12 bits cada uno. El resultado de la operación es un valor con longitud de 24 bits.

Tabla 5.11 Descripción del bloque X.

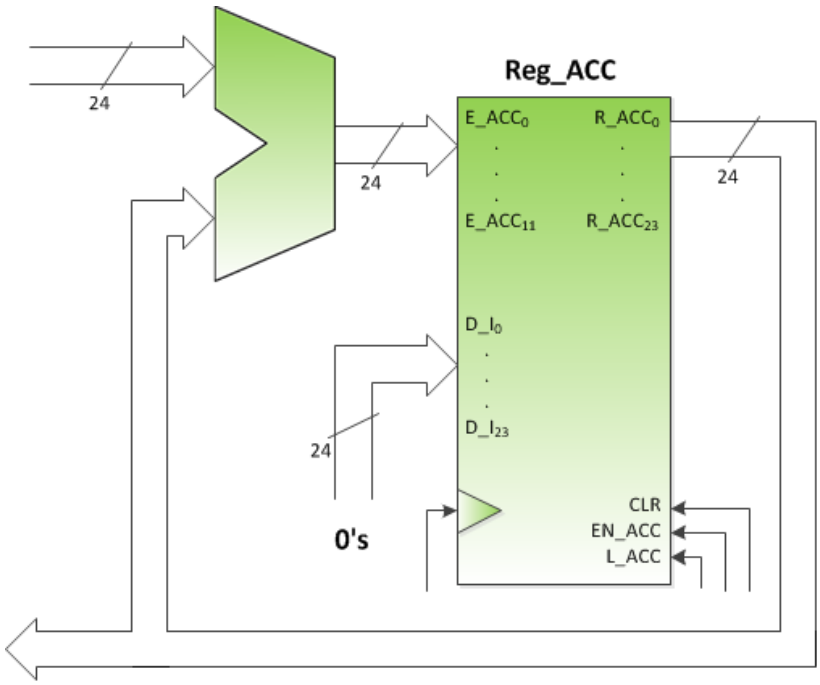
5.1.2.3.9. REG_X



Nombre	REG_X
Tipo de bloque	Registro de 24 bits
Funcionamiento	Secuencial
Señal de datos de entrada	E_X
Bloque del que provienen los datos	X
Señal de datos de salida	R_X
Bloque al que se envían los datos	REG_ACC
Señales de control	CLK: Proporciona la frecuencia de trabajo del bloque.
	CLR: Coloca el valor del bloque a su valor inicial, generalmente a cero.
	L_X: permite cargar un valor al bloque mediante el bus de entrada.
Descripción	El REG_X se encarga de almacenar el valor proveniente del bloque X

Tabla 5.12 Descripción del bloque REG_X.

5.1.2.3.10. REG_ACC

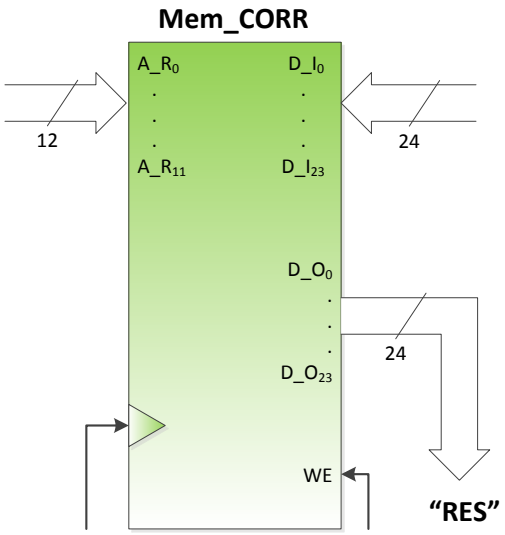


Nombre	REG_ACC
Tipo de bloque	Registro acumulador de 24 bits
Funcionamiento	Secuencial
Señal de datos de entrada	E_ACC
	D
Bloque del que provienen los datos	Para E_ACC: REG_X/ REG_ACC
	Para D: -
Señal de datos de salida	R_ACC
Bloque al que se envían los datos	REG_ACC/ MEM_CORR
Señales de control	CLK: Proporciona la frecuencia de trabajo del bloque.
	CLR: Coloca el valor del bloque a su valor inicial, generalmente a cero.
	EN_ACC: Activa el bloque para que el valor pueda salir por el bus de salida.
	L_ACC: permite cargar un valor al bloque mediante el bus de entrada.

Descripción	El REG_ACC es un tipo de registro especial cuya función es sumar los valores de resultado que se generan. La ruta que toman los datos de salida dependen del valor almacenado en el bloque CONT_I como sigue: Si el valor de CONT_I es menor o igual a 30, el valor proveniente del bloque REG_X se suma al valor contenido en REG_ACC y el resultado se guarda nuevamente en REG_ACC. Si el valor de CONT_I es igual a 31 el valor contenido en REG_ACC se envía al bus de salida hacia el bloque MEM_CORR.

Tabla 5.13 Descripción del bloque REG_ACC.

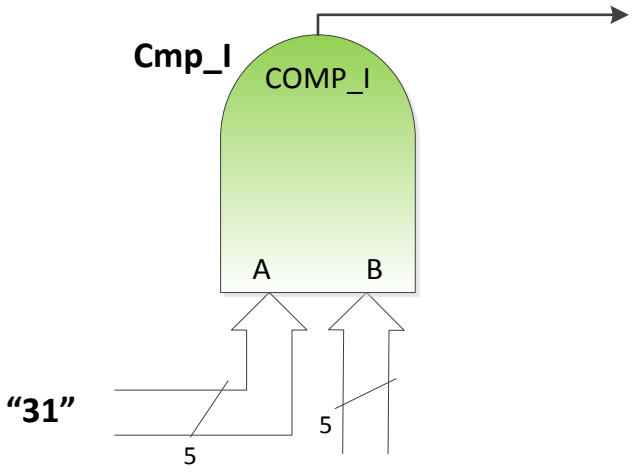
5.1.2.3.11. MEM_CORR



Nombre	MEM_CORR
Tipo de bloque	Memoria RAM de 4096x24 (12KB)
Funcionamiento	Síncrono
Señal de datos de entrada	A
	D_I
Bloque del que provienen los datos	Para A: CONT_J
	Para D_I: REG_ACC
Señal de datos de salida	D_O
Bloque al que se envían los datos	-
Señales de control	CLK: Proporciona la frecuencia de trabajo del bloque.
	WE: permite controlar tanto la escritura como la lectura de datos mediante los buses de entrada y salida respectivamente.
Descripción	El bloque de memoria MEM_CORR es el encargado de almacenar los valores finales de la operación de correlación, los cuales se utilizaran en una etapa posterior.

Tabla 5.14 Descripción del bloque MEM_CORR.

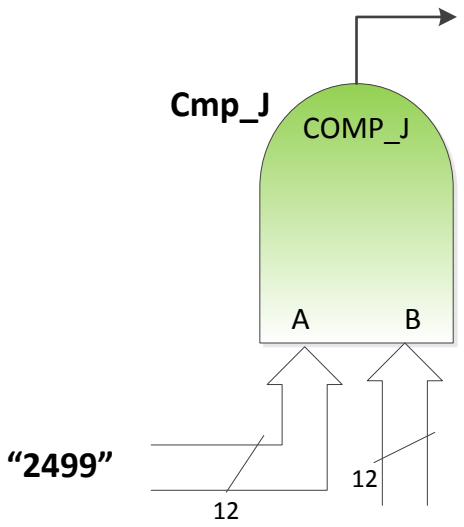
5.1.2.3.12. CMP_I



Nombre	CMP_I
Tipo de bloque	Compuerta lógica AND
Funcionamiento	Combinatorio
Señal de datos de entrada	A
	B
Bloque del que provienen los datos	Para A: -
	Para B: CONT_I
Señal de datos de salida	COMP_I
Bloque al que se envían los datos	U_CONTROL
Señales de control	-
Descripción	El bloque CMP_I es una compuerta lógica de tipo AND que activa su bit de resultado cuando el valor del bus de entrada B es igual al valor del bus de entrada A, el cual tiene un valor constante de 31.

Tabla 5.15 Descripción del bloque CMP_I.

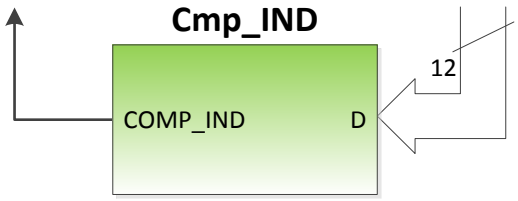
5.1.2.3.13. CMP_J



Nombre	CMP_J
Tipo de bloque	Compuerta lógica AND
Funcionamiento	Combinatorio
Señal de datos de entrada	A
	B
Bloque del que provienen los datos	Para A: -
	Para B: CONT_J
Señal de datos de salida	COMP_J
Bloque al que se envían los datos	U_CONTROL
Señales de control	-
Descripción	El bloque CMP_J es una compuerta lógica de tipo AND que activa su bit de resultado cuando el valor del bus de entrada B es igual al valor del bus de entrada A, el cual tiene un valor constante de 2499.

Tabla 5.16 Descripción del bloque CMP_J.

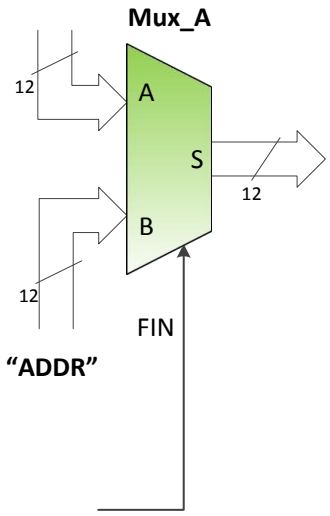
5.1.2.3.14. CMP_IND



Nombre	CMP_IND
Tipo de bloque	Comparador lógico
Funcionamiento	Combinatorio
Señal de datos de entrada	D
Bloque del que provienen los datos	CONT_IND
Señal de datos de salida	COMP_IND
Bloque al que se envían los datos	U_CONTROL
Señales de control	-
Descripción	El bloque CMP_IND es un comparador lógico que activa su bit de resultado cuando el valor del bus de entrada D es menor a 0 o mayor a 2486. Ya que los valores son utilizados como direcciones de memoria, este comparador indica cuando se generan direcciones negativas o direcciones con contenido basura.

Tabla 5.17 Descripción del bloque CMP_IND.

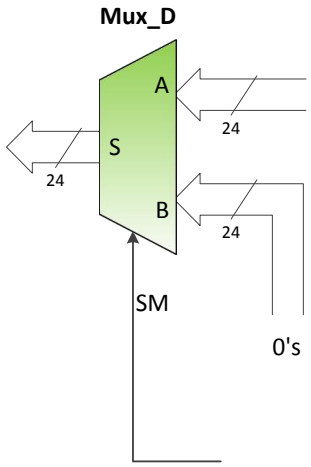
5.1.2.3.15. MUX_A



Nombre	MUX_A
Tipo de bloque	Multiplexor
Funcionamiento	Combinatorio
Señal de datos de entrada	A
	B
Bloque del que provienen los datos	Para A: CONT_J
	Para B: -
Señal de datos de salida	S
Bloque al que se envían los datos	MEM_CORR
Señales de control	FIN
Descripción	El MUX_A es un selector de ruta de datos, que permite elegir la fuente de la cual, el bloque MEM_CORR obtendrá los datos mediante la señal de control FIN. Si el valor de FIN es 0, la información en el bus de salida provendrá del bloque CONT_J. Si el valor de FIN es 1 la información en el bus de salida provendrá del exterior.

Tabla 5.18 Descripción del bloque MUX_A.

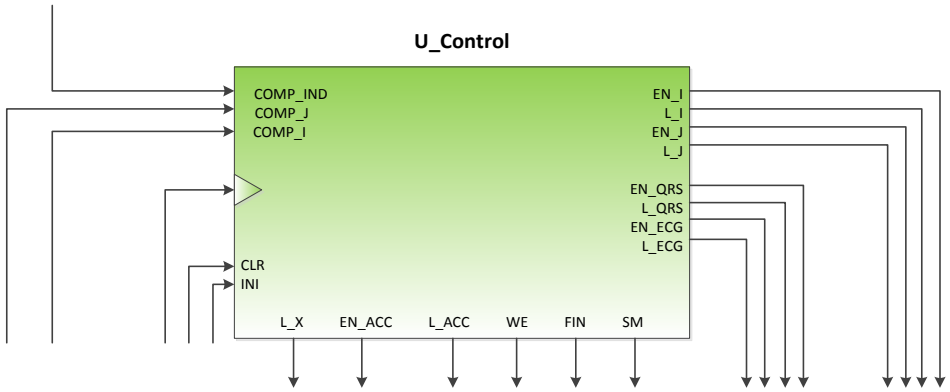
5.1.2.3.16. MUX_D



Nombre	MUX_D
Tipo de bloque	Multiplexor
Funcionamiento	Combinatorio
Señal de datos de entrada	A
	B
Bloque del que provienen los datos	Para A: REG_ACC
	Para B: -
Señal de datos de salida	S
Bloque al que se envían los datos	MEM_CORR
Señales de control	SM
Descripción	El MUX_D es un selector de ruta de datos, que permite elegir la fuente de la cual, el bloque MEM_CORR obtendrá los datos mediante la señal de control SM. Si el valor de SM es 0, la información en el bus de salida provendrá del bloque REG_CORR. Si el valor de SM es 1 la información del bloque de salida provendrá de un valor constante de 0.

Tabla 5.19 Descripción del bloque MUX_D

5.1.2.3.17. U_CONTROL



Nombre	U_CONTROL
Tipo de bloque	Unidad de control
Funcionamiento	Secuencial
Señal de datos de entrada	COMP_IND
	COMP_J
	COMP_I
Bloque del que provienen los datos	Para COMP_IND: CMP_IND
	Para COMP_J: CMP_J
	Para COMP_I: CMP_I
Señal de datos de salida	EN_I
	L_I
	EN_J
	L_J
	EN_QRS
	L_QRS
	EN_ECG
	L_ECG
	L_X
	EN_ACC
	L_ACC

	WE
	FIN
	SM
Bloque al que se envían los datos	Para EN_I: CONT_I
	Para L_I: CONT_I
	Para EN_J: CONT_J
	Para L_J: CONT_J
	Para EN_QRS: MEM_QRS
	Para L_QRS: REG_QRS
	Para EN_ECG: MEM_ECG
	Para L_ECG: REG_ECG
	Para L_X: REG_X
	Para EN_ACC: REG_ACC
	Para L_ACC: REG_ACC
	Para WE: MEM_CORR
	Para FIN: MUX_A
	Para SM: MUX_D
Señales de control	CLK: Proporciona la frecuencia de trabajo del bloque.
	CLR: Coloca el valor del bloque a su valor inicial, generalmente a cero.
	INI: Indica el comienzo de procesamiento de la información en el sistema.
Descripción	El bloque U_CONTROL es el encargado de gestionar los demás bloques que conforman el sistema. Su construcción se basa en una máquina de estados, donde en cada estado se especifica el orden y el momento en el que cada señal de control se debe activar. Las señales COMP_IND, COMP_I, COMP_J, son valores de entrada condicionales que permiten identificar cuando se ha alcanzado cierto valor en algún bloque para poder hacer la transición al siguiente estado.

Tabla 5.20 Descripción del bloque U_CONTROL.

5.1.2.4. Autómata de control

El siguiente paso después de haber establecido el diagrama de ruta de datos, es diseñar el autómata de control, que se encarga de gestionar el comportamiento de todos los módulos que conforman el sistema.

El autómata de control es la representación gráfica del funcionamiento de uno de los módulos del sistema llamado Unidad de Control. La Unidad de Control es un bloque dentro del sistema, al cual se conectan todas las señales de habilitación, carga, selección, entre otras, de los demás bloques del sistema; logrando una coordinación entre todos ellos.

El diseño del autómata de control es idéntico a la carta ASM, ya que también cuenta con estados, los cuales también son controlados por una señal de reloj, lo que implica que la ejecución del autómata se realiza de forma secuencial.

La principal diferencia con la carta ASM es que, en los estados, se colocan las señales de control que serán activadas para controlar los respectivos módulos especificados en la carta ASM. Estas señales de control son generalmente señales booleanas, es decir, sólo tienen dos posibles valores, 1 o 0, ya que su función es activar o desactivar una función específica de algún modulo.

En la figura 5.7 se tiene el autómata de control obtenido de la carta ASM. El número de estados se sigue conservando igual, sin embargo, las variables, módulos, señales y otros, son cambiados por las respectivas señales de control de cada uno.

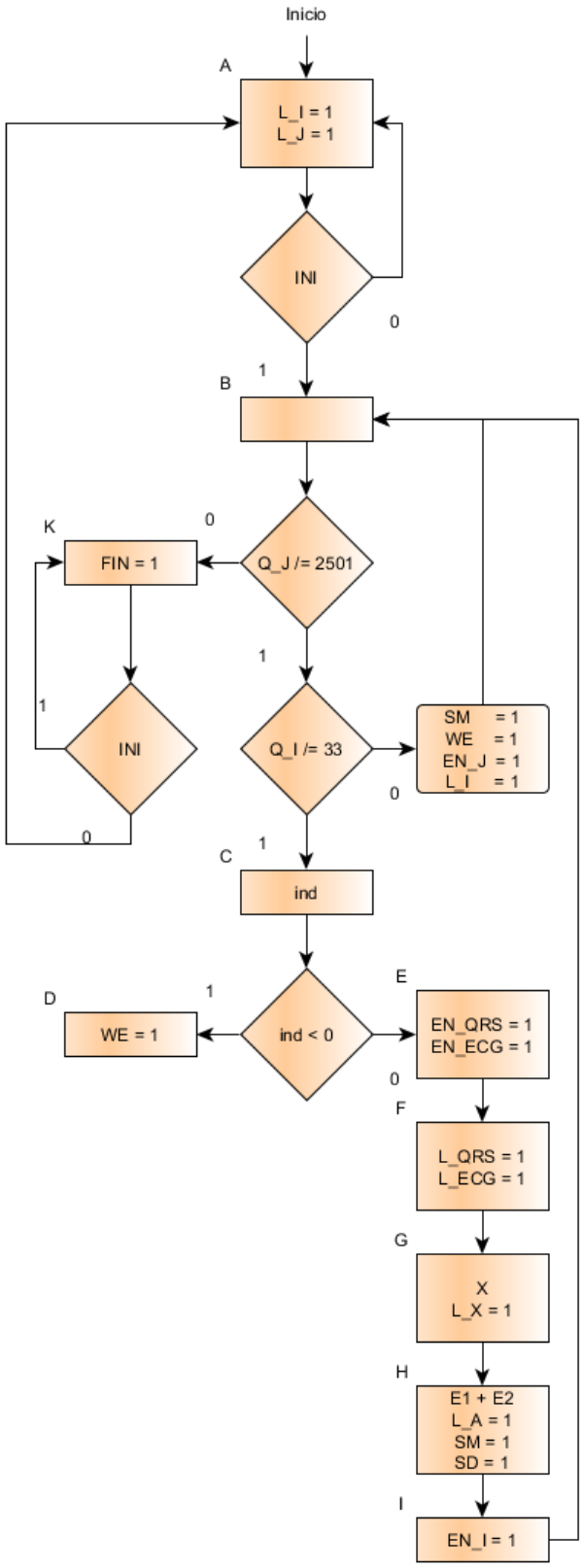


Figura 5.7 Autómata de Control obtenido a partir de la Carta ASM.

Capítulo 6
Implementación y Pruebas

6.1. Dispositivo de hardware

Del análisis y diseño desarrollado anteriormente, se pueden obtener los requerimientos específicos del sistema, así como las capacidades y características físicas que debe cumplir el hardware a utilizar. De acuerdo con las definiciones dadas en el diagrama de ruta de datos, el dispositivo debe poder ser capaz de cubrir los siguientes requerimientos:

- El dispositivo debe poderse programar en el leguaje de descripción de hardware VHDL.

- Debe ofrecer la posibilidad de agregar o quitar bloques funcionales según se requiera, así como modificar los ya implementados.

- Contar con unidades me memoria dedicadas o CLB's suficientes para poder implementar las memorias ROM que contienen el patrón de muestra y la señal ECG a procesar, las cuales requieren 48 bytes y 6KB de capacidad respectivamente.

- Contar con unidades me memoria dedicadas o CLB's suficientes para poder implementar la memoria RAM donde se almacenan los resultados del procesamiento, la cual requiere 12KB de capacidad.

- Incluir recursos dedicados para realizar operaciones especiales como acumulación y multiplicación de valores. Estas operaciones generalmente pueden ser realizadas por módulos llamados DSP.

- Soportar protocolos de comunicación para leer y escribir datos desde y hacia el exterior (RS-232, TCP/IP).

- Poseer interfaces de comunicación para conectarse con otros dispositivos e intercambiar información (serie-USB, conectores de red RJ-45, puertos de expansión para Pmod's, JTAG-USB).

- Adicionalmente, deben existir dispositivos para poder comprobar el correcto funcionamiento del sistema (LED's push buttons, switches).

Previamente, en la sección 2 de este documento, se propusieron tres diferentes tipos de tarjetas de desarrollo, todas ellas con un FPGA de la compañía Xilinx. El FPGA de cada una de estas tarjetas pertenece a la familia Spartan, por lo que poseen características similares, sin embargo, existen diferencias marcadas en la subfamilia de cada FPGA.

Tomando en cuenta los requerimientos antes mencionados, así como la información contenida en la tabla 4.7 respecto a las características y recursos de las tarjetas de desarrollo, se elige la que cubra de mejor manera las necesidades de recursos, así como algunos otros factores que se mencionarán en la siguiente sección.

6.1.1. Selección

Una vez listadas las características de las tarjetas candidatas a utilizar, se debe seleccionar la tarjeta sobre la cual se alojará el sistema embebido. Para elegir la más adecuada, es necesario verificar que las características del hardware cubran las necesidades de los requerimientos del sistema. También es necesario considerar algunos factores que no forman parte de los requerimientos del sistema como puede ser el costo y las herramientas de desarrollo.

En la tabla 6.1 se encuentran algunas de las características principales a considerar para elegir una de las tres tarjetas analizadas. Se incluyen tanto características de los requerimientos del sistema, como puntos de las características propias de las tarjetas.

Característica	Spartan 3A FPGA Starter Kit	Avnet Spartan-6 LX9 MicroBoard	Nexys3
Módulo de adquisición (ADC)	SI	NO	NO
Capacidad de agregar Pmods	SI	SI	SI
Unidades DSP	NO	SI	SI
Controladores físicos de RAM	NO	SI	NO
Dimensiones aproximadas	25cm x 25cm x 3cm	10cm x 5cm x 2cm	20cm x 20cm x 3cm
Precio	189 dólares	89 dólares	119 dólares

Tabla 6.1 Tabla de cotejo de características a cubrir por la tarjeta a usar.

De ésta manera, la tarjeta de desarrollo Spartan 3A FPGA Starter Kit cumple con dos de los primeros 4 puntos. La tarjeta Avnet Spartan-6 LX9 MicroBoard cumple con 3 de los primeros 4 puntos. La tarjeta Nexys3 cumple con 2 de los primeros 4 puntos.

En los dos últimos puntos referentes al tamaño y costo de la tarjeta, la Avnet Spartan-6 LX9 MicroBoard es la que posee el menor tamaño y el menor costo de las tres. La Nexys3 se encuentra en la posición intermedia en costo y dimensiones. La tarjeta Spartan 3A FPGA Starter Kit es la de mayor costo y tamaño de las tres tarjetas.

6.1.2. Avnet Spartan-6 LX9 MicroBoard

La tarjeta de desarrollo Avnet Spartan-6 LX9 MicroBoard, elegida para desarrollar el sistema, contiene todos los elementos necesarios para cubrir las demandas de recursos, al mismo tiempo que su costo es el más económico de las tres.

La tarjeta fue desarrollada por la compañía Avnet, utilizando como núcleo principal un FPGA de la familia Spartan6, la cual pertenece a la compañía Xilinx. La tarjeta cuenta con varios recursos de hardware adicionales integrados dentro de la propia tarjeta como leds, un dip switch, un conector de red RJ-45, bloques generadores de frecuencias de reloj, y una pequeña memoria externa.

Las características y recursos con los que cuenta la tarjeta de desarrollo son [33]:

- FPGA, Familia: Spartan-6, Modelo: XC6SLX9-2CSG324C
- Memoria externa de 64 MB, Tipo: LPDDR, Tecnología: SDRAM
- Memoria de 128 Mb Tipo: Multi-E/S, Tecnología: SPI Flash
- Chip PHY Ethernet compatible con 10/100
- Interfaz de comunicación micro USB-to-UART
- Circuito de programación USB JTAG integrado en la tarjeta
- 2 conectores tipo head de 2x6, compatibles con Pmod de Digilent
- Chip único, 3 fuentes de alimentación con indicador de buen funcionamiento
- Chip de reloj programable
- Protección contra sobrecarga y ESD en USB
- 4 LEDs
- DIP switch de 4-bit
- Push-buttons de Reset y PROG

Como se puede apreciar, la tarjeta cuenta con todos los recursos periféricos necesarios para implementar el proyecto. Aunado a esto, su factor de forma compacto tipo pendrive, así como su bajo costo, permiten un mayor grado de trasportación y adquisición respectivamente.

En la figura 6.1 se puede observar una fotografía de la tarjeta desde una vista superior, observándose los componentes periféricos principales [34].

Figura 6.1 Tarjeta de desarrollo Avnet Spartan-6 LX9 MicroBoard.

Ésta tarjeta no cuenta con un módulo de adquisición y conversión de datos integrado. Sin embargo, es posible agregar un módulo externo mediante los conectores de expansión.

Actualmente, existen varias empresas que se dedican a la fabricación de módulos externos para ser integrados a las tarjetas de desarrollo. A estos módulos se les conoce como Pmods.

Para cubrir el requerimiento de adquisición y digitalización de la señal, es necesario agregar un Pmod que cuente con un convertidor analógico-digital para procesar la señal dentro del FPGA. Al igual que con las tarjetas de desarrollo, también se debe hacer un análisis de los Pmods disponibles para elegir el adecuado.

6.2. Herramientas de desarrollo de software

La implementación del sistema embebido requiere de dos herramientas de desarrollo, ambas proporcionadas por la compañía Xilinx.

El primer paso consiste en crear los archivos de código que conforman el core de procesamiento, para ellos se utiliza la herramienta ISE Design Suite.

Una vez codificado y probado, el siguiente paso consiste en integrar el core, ya como un bloque periférico junto con los módulos adicionales que controlaran las interfaces de comunicación y almacenamiento de datos. Para esto se utiliza la herramienta EDK.

El EDK a su vez está divido en dos partes, la parte que se encarga de sintetizar tanto el core como los bloques adicionales dentro del FPGA, el XPS. Después se encuentra la herramienta que permite generar las librerías y drivers necesarios para la comunicación con otros dispositivos externos, el SDK.

Finalmente, para poder visualizar los resultados del procesamiento de una manera más cómoda, se requiere una herramienta que pueda generar una representación gráfica de los datos resultantes, para lo cual se hace uso del entorno de desarrollo de Java conocido como Netbeans, en conjunto con una librería especial para la generación de graficas llamada JFreeChart, que a su vez se apoya en otra librería llamada JCommon.

6.2.1. ISE Design Suite

El ISE Design Suite es un conjunto de aplicaciones que proporcionan un entorno grafico de desarrollo de proyectos basado en lenguajes de descripción de hardware. El software permite realizar varias tareas como crear archivos personalizados, agregar archivos previamente elaborados, exportar el diseño a un dispositivo, realizar simulaciones, visualizar diagramas de bloques, entre otras.

Sus principales características se listan a continuación [35]:

El ISE Design Suite es la solución probada en la industria para todos los dispositivos programables de Xilinx, incluyendo las series 7 (y dispositivos anteriores a la serie 7) y SoC Zynq-7000 All Programmable. Está disponible en tres ediciones:

- **ISE Design Suite: Embedded Edition**: El ISE Design Suite: Embedded Edition incluye Xilinx Platform Studio (XPS), Software Development Kit (SDK), un gran repositorio de IP plug and play incluyendo el Procesador Soft MicroBlaze y periféricos, y un completo flujo de diseño RTL to bit stream. Embedded Edition proporciona las herramientas fundamentales, tecnologías y un flujo de diseño familiar para alcanzar óptimos resultados de diseño. Estos incluyen compuertas de reloj inteligente para reducción dinámica de poder, diseño en equipo para equipos de diseño multi-sitio, preservación del diseño para repeticiones de tiempos, y una opción de reconfiguración parcial para mayor flexibilidad del sistema, tamaño, energía, y reducción de costos.

- **ISE Design Suite: System Edition:** El ISE Design Suite: System Edition está construido sobre el Embedded Edition agregando el Sistema Generador para DSP. El Sistema Generador para DSP es la herramienta de alto nivel líder de la industria para diseñar sistemas DSP de alto rendimiento usando dispositivos Xilinx All Programmale, proporcionando el modelado del sistema y generación automática de código de Simulink y MATLAB.

- **ISE Design Suite: WebPACK Edition:** El ISE WebPACK ofrece un completo, flujo de diseño front-to-back proporcionando acceso instantáneo a las características y funcionalidad del ISE sin costo.

6.2.1.1. ISE WebPACK Design Software

ISE WebPACK Design Software es el único gratuito de la industria, una solución de diseño front-to-back en FPGA totalmente equipada para Linux, Windows XP, y Windows 7. ISE WebPACK es la solución descargable ideal para diseños en FPGA y CPLD, ofreciendo síntesis y simulación en HDL, implementación, montaje del dispositivo, y programación mediante JTAG. ISE WebPACK ofrece un completo flujo de diseño front-to-back, dando acceso instantáneo a las opciones y funcionalidad del ISE sin costo. Xilinx ha creado una solución que permite una conveniente productividad proporcionando una solución de diseño que está siempre actualizada, sin errores de descarga y un único archivo de instalación.

Características principales[36]:

- Un gratuito ambiente descargable de diseño para PLDA en Microsoft Windows and Linux
- Soporte de diseño de procesamiento embebido para la familia Zynq-7000 All Programmable SoC, el Z-7010, Z-7020, y Z-7030
- El tiempo de cierre más rápido de la industria con tecnología Xilinx SmartCompile
- Completo entorno de diseño front-to-back, incluyendo el sistema Xilinx CORE Generator y el diseño completo de PlanAhead y herramientas de análisis — con el nuevo flujo de diseño RTL to Bitstream para diseñadores lógicos
- Verificación Integrada de HDL con la versión Lite del ISE Simulator (ISim)
- La forma más fácil y barata para iniciarse con el líder de la industria en productividad, rendimiento y poder.
- Capacidad de actualizarse fácilmente en cualquiera de las ediciones ISE Design Suite de la Xilinx Online Store.

Para poder usar un entorno gráfico, se incluye el ambiente de trabajo llamado Project Navigator. Este es similar a los IDE de desarrollo de software para lenguajes como C/C++ o Java, ya que también es posible crear proyectos y agregar archivos de código fuente. En la figura 6.2 se muestra una imagen de referencia del entorno inicial de la aplicación.

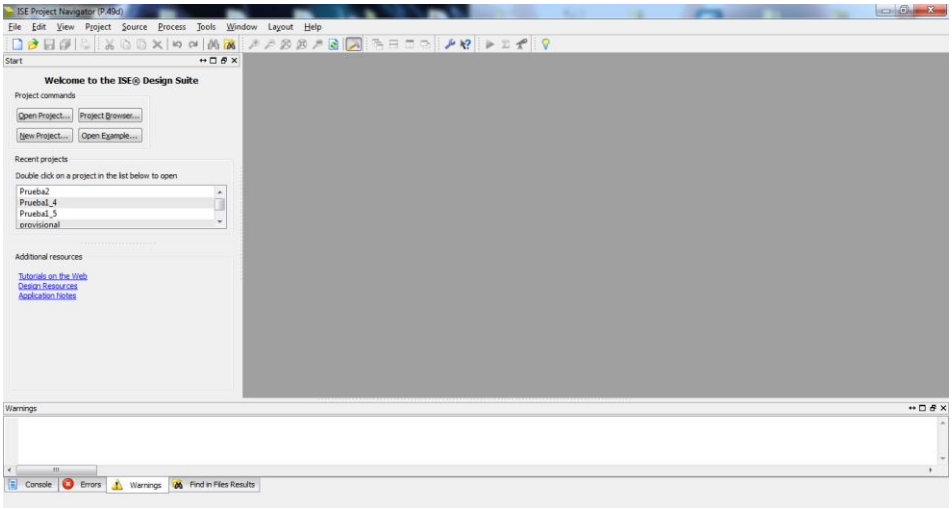


Figura 6.2 Imagen de referencia de la ventana de Project Navigator.

6.2.1.2. Embedded Development Kit (EDK)

El Embedded Development Kit (EDK) es un entorno de desarrollo integrado para diseñar sistemas de procesamiento embebidos. Este kit pre-configurado incluye el Xilinx Platform Studio y el Software Development Kit, así como toda la documentación e IP que se requieren para diseños en FPGA en la plataforma Xilinx con procesadores hard cores embebidos PowerPC y/o procesadores soft cores MicroBlaze.

El Embedded Development Kit proporciona[37]:

- **Xilinx Platform Studio (XPS) Tool Suite** - Incluye: IDE gráfico y soporte para línea de comandos para desarrollar plataformas de hardware para aplicaciones embebidas. El asistente Base System Builder permite la creación de un sistema embebido funcional en minutos. XPS también incluye otros asistentes de diseño inteligentes para configurar rápidamente la arquitectura del sistema embebido, buses y periféricos.

- **Software Development Kit (SDK) para MicroBlaze y PowerPC** - Incluye: GNU C/C++ compilador y depurador; servidor de destino Xilinx Microprocessor Debug (XMD); utilidad Data2MEM para cargar y actualizar el bitstream. SDK es el entorno de diseño de software centralizado recomendado basado en el IDE Eclipse.

- **Sistema Operativo de Tiempo Real y Suporte para SO Embebido** - Proporciona soporte de diseño y la generación de Board Support Package (BSP) para numerosos proveedores de terceros en el mundo de Xilinx, incluyendo proveedores como Wind River, Green Hills, Mentor, LynuxWorks y otros líderes en la industria de embebidos.

- **IP de Procesamiento y el Procesador Soft Core MicroBlaze** – Catálogo pre comprobado de IP, incluye una amplia variedad de cores periféricos de procesamiento para personalizar el sistema embebido así como la flexibilidad del procesador soft core MicroBlaze de 32-bit. El procesador MicroBlaze ofrece gestión de memoria y opciones de configuración de FPU habilitando el soporte RTOS de grado comercial, único para los procesadores soft.

Como este sistema procesa información que posteriormente se mostrará de manera gráfica con ayuda de una computadora, es necesario utilizar tanto el software de configuración de hardware XPS para implementar el sistema de procesamiento dentro del FPGA, así como el software SDK que contiene el compilar de código que permite generar las librerías necesarias para establecer la comunicación con la computadora.

6.2.1.2.1. Xilinx Platform Studio (XPS)

Xilinx Platform Studio (XPS) es un componente clave del ISE Embedded Edition Design Suite, ayudando al diseñador de hardware a construirlo fácilmente, conectar y configurar sistemas basados en procesadores embebidos; desde simples máquinas de estado, hasta robustos sistemas con microprocesadores RISC de 32-bit.

XPS emplea vistas de diseño gráfico y sofisticados asistentes de corrección a través del diseño para guiar a los desarrolladores a través de los pasos necesarios para crear sistemas de procesador personalizado en minutos.

El verdadero potencial de XPS surge con su habilidad para configurar e integrar IP cores plug and play del catálogo Xilinx Embedded IP, con diseños de Verilog y VHDL personalizados o de terceros. Ahora, procesadores altamente personalizados pueden ser diseñados acorde con las necesidades específicas del proyecto incluyendo; periféricos y requerimientos de E/S, respuesta en tiempo real, poder de procesamiento de propósito general, rendimiento de punto flotante, memoria en chip o no, mínimo consumo de energía y mucho más.

Los desarrolladores de firmware y software se benefician de la integración de XPS con Xilinx SDK que permite la generación automática de un sistema crítico de software como gestores de arranque, BSP en crudo, y BSP Linux. Esta función asegura que la adaptación del sistema operativo y el desarrollo de aplicaciones pueden comenzar sin retrasos causados por el desarrollo del firmware [38].

En la figura 6.3 se observa una ventana de ejemplo de un proyecto desarrollado en el entorno XPS.

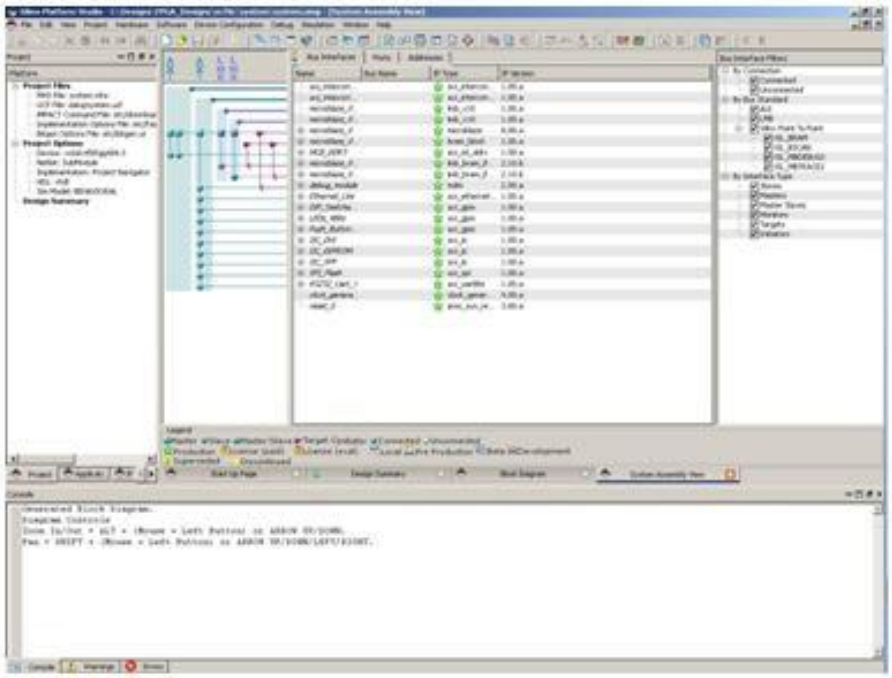


Figura 6.3 Imagen de referencia del entorno de trabajo XPS versión 14.4.

6.2.1.2.2. Software Development Kit (SDK)

El Software Development Kit (SDK) es el entono de diseño integrado de Xilinx para crear aplicaciones embebidas sobre cualquiera de los microprocesadores Xilinx Zynq-7000 All Programmable SoCs, y el líder de la industria MicroBlaze ™. El SDK es el primer IDE de aplicación para desarrollar un verdadero multiprocesador homogéneo y heterogéneo en diseño y depuración[39].

- Soporte para Zynq-7000 AP SoCs, y MicroBlaze
- Incluido con la Vivado Design Suite o disponible como una descarga gratuita por separado para desarrolladores de software embebido
- Basado en Eclipse 4.3.2 y CDT 8.3.0 (liberado como 2014.4)
- Integrated Design Environment (IDE) completo que hace interfaz directamente con el entorno de diseño de hardware embebido Vivado.
- Diseño completo de software y flujo de depuración soportado, incluyendo multiprocesador y capacidades de sub depuración de hardware/software
- Editor, compiladores, herramientas de construcción, gestión de memoria flash, e integración de depuración JTAG/GDB

- Soportado por la edición de Xilinx Mentor Sourcery CodeBench Lite (versión 2014.05)

- Librerías personalizadas y drivers de dispositivos

En la figura 6.4 que se muestra a continuación, se presenta una imagen de referencia del ambiente de trabajo SDK.

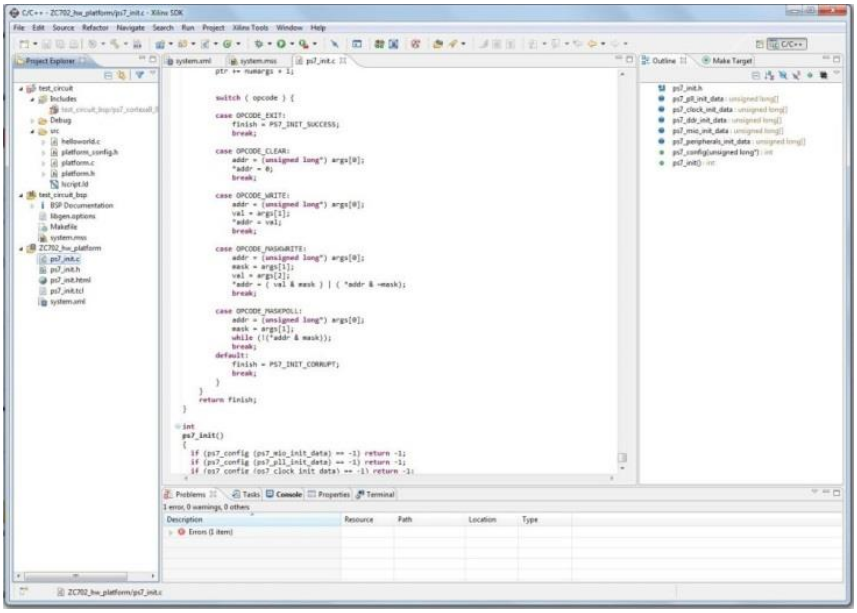


Figura 6.4 Imagen de referencia del entorno de trabajo SDK versión 14.4.

6.2.2. Netbeans

El IDE NetBeans es una herramienta de diseño modular para un amplio rango de aplicaciones de desarrollo de tecnologías. El IDE base incluye un avanzado editor multilenguaje, Depurador y Perfilador, así como herramientas de control de versión y desarrollo colaborativo.

Plantillas y Aplicaciones de Ejemplo: El IDE de NetBeans brinda estructuras de aplicaciones en forma de plantillas de proyecto para todas las tecnologías que soporta. Además, proporciona un conjunto de aplicaciones de ejemplo, algunas de las cuales pueden ser recreadas paso a paso siguiendo un tutorial relacionado.

El IDE proporciona plantillas de proyecto y proyectos de ejemplo que ayudan a crear aplicaciones Java SE, aplicaciones Java EE, aplicaciones Java ME, aplicaciones HTML5, aplicaciones en Plataforma NetBeans, aplicaciones PHP, y aplicaciones C/C++[40].

Bases de Datos y Servicios: La ventana de Servicios da acceso a muchos recursos auxiliares, como bases de datos, servidores, servicios web, y control de incidencias. Pueden iniciarse y detenerse bases de datos y servidores directamente en el IDE. Cuando se trabaja con bases de datos, se puede agregar, quitar y modificar datos en el IDE. Cuando se ha desplegado una aplicación a un servidor, se pueden administrar los recursos desplegados porque son mostrados en el nodo Servidor.

Administrador de Plugins: Mientras se usa el IDE, siempre se puede ir al Administrador de Plugins del menú Herramientas para agregar, quitar o actualizar el conjunto de opciones instaladas. Una gran variedad de plugins están disponibles para todo tipo de desarrollos, desde Java SE, Java EE, Java ME, HTML5, Groovy, y PHP hasta desarrollo en C/C++.

en la figura 6.5 se muestra la ventana principal del IDE de NetBeans, cuando se abre la aplicación.

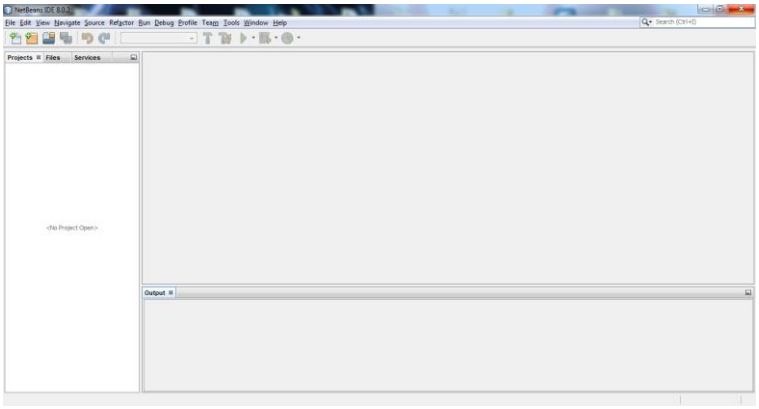


Figura 6.5 Imagen de referencia del entorno de trabajo Netbeans versión 8.0.

6.2.2.1. JFreeChart

JFreeChart es una librería de graficas en Java 100% gratis que hace fácil para los desarrolladores mostrar gráficos de calidad profesional en sus aplicaciones. El amplio conjunto de funciones de JFreeChart incluye[41]:

- Una consistente y bien documentada API, soportando un amplio rango de tipos de gráficas.

- Un diseño flexible que es fácil de extender, y aplicaciones objetivo tanto del lado del servidor como del lado del cliente

- Soporte para muchos tipos de salida, incluyendo componentes Swing y JavaFX, archivos de imagen (incluyendo PNG y JPEG), y formatos de archivo de vectores gráficos (incluyendo PDF, EPS y SVG)

- JFreeChart es de código abierto o, más específicamente, software libre. Es distribuido bajo los términos del GNU Lesser General Public Licence (LGPL), la cual permite usarlo en aplicaciones propietarias.

La figura 6.6 muestra algunas graficas de ejemplo que se pueden generar con ayuda de esta librería.

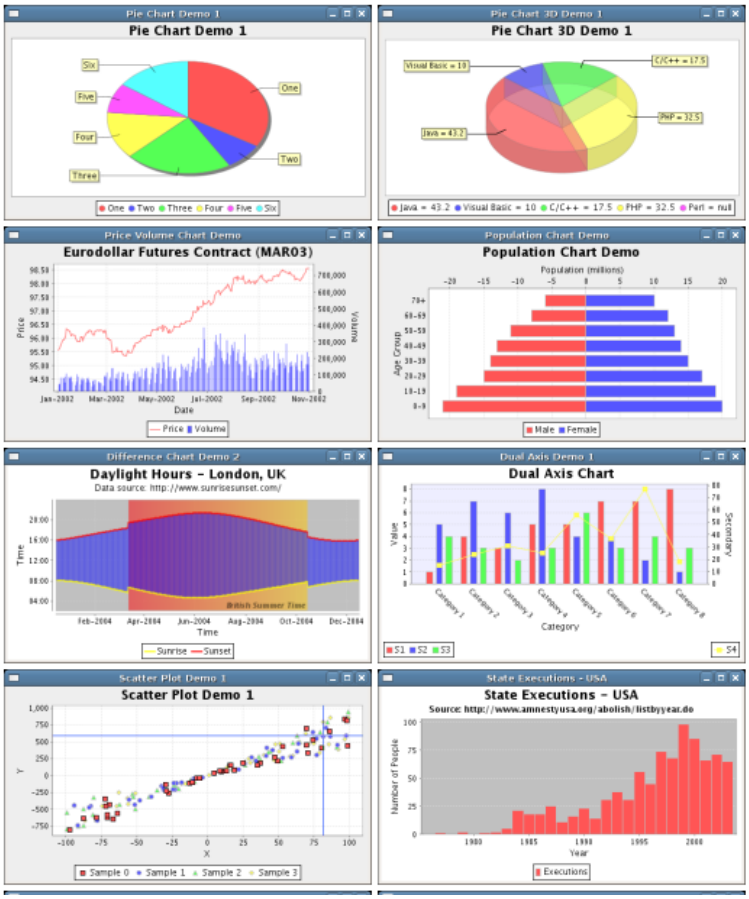


Figura 6.6 Diferentes tipos de gráficas de ejemplo generadas con la librería JFreeChart.

6.2.2.2. JCommon

Jcommon es una biblioteca de clases de Java que es usada por JFreeChart, Pentaho Reporting y algunos otros proyectos. La librería contiene diversas clases que soportan[42]:

- Configuración y gestión de dependencias de código
- Un marco de registro general
- Clases de interfaz de usuario para mostrar información de las aplicaciones
- Administradores de capas personalizadas
- Un panel selector de fechas
- Utilidades de serialización

Jcommon tiene licencia bajo los términos de GNU Lesser General Public Licence (LGPL) versión 2.1 o posterior.

6.3. Implementación del core en ISE WebPACK

Una vez descritas las herramientas necesarias para crear el sistema embebido, el primer paso es crear el core, que es el encargado de procesar toda la información para obtener los resultados.

Para crear y comprobar el funcionamiento del core, la primera herramienta a utilizar es el ISE WebPACK Design Software, que permite crear los archivos fuente de los bloques que conforman el core de procesamiento del sistema embebido, así como acceder a funciones de simulación y vistas de diseño.

Dentro de la herramienta se encuentran dos opciones que permiten visualizar la forma en la que se implementa el core dentro del FPGA, tanto de manera física como de manera lógica.

6.3.1. Diagrama RTL

Una de las opciones disponibles en la herramienta es poder generar y visualizar el diagrama RTL del diseño. Un diagrama RTL (Register Transfer Logic o Lógica de Transferencia de Registros en español) muestra los distintos bloques que conforman el módulo general, las interconexiones que existen entre ellos y las distintas señales de E/S presentes en cada uno.

El diagrama RTL es muy similar al diagrama de ruta de datos analizado en la sección anterior, ya que ambos representan la lógica de funcionamiento del core. Por esta razón, varios de los bloques son comunes a ambos diagramas, con lo que se puede comprobar en cierto grado, que la implementación es correcta.

En la figura 6.7 se muestra el diagrama RTL generado con la herramienta que forma parte del ISE WebPACK.

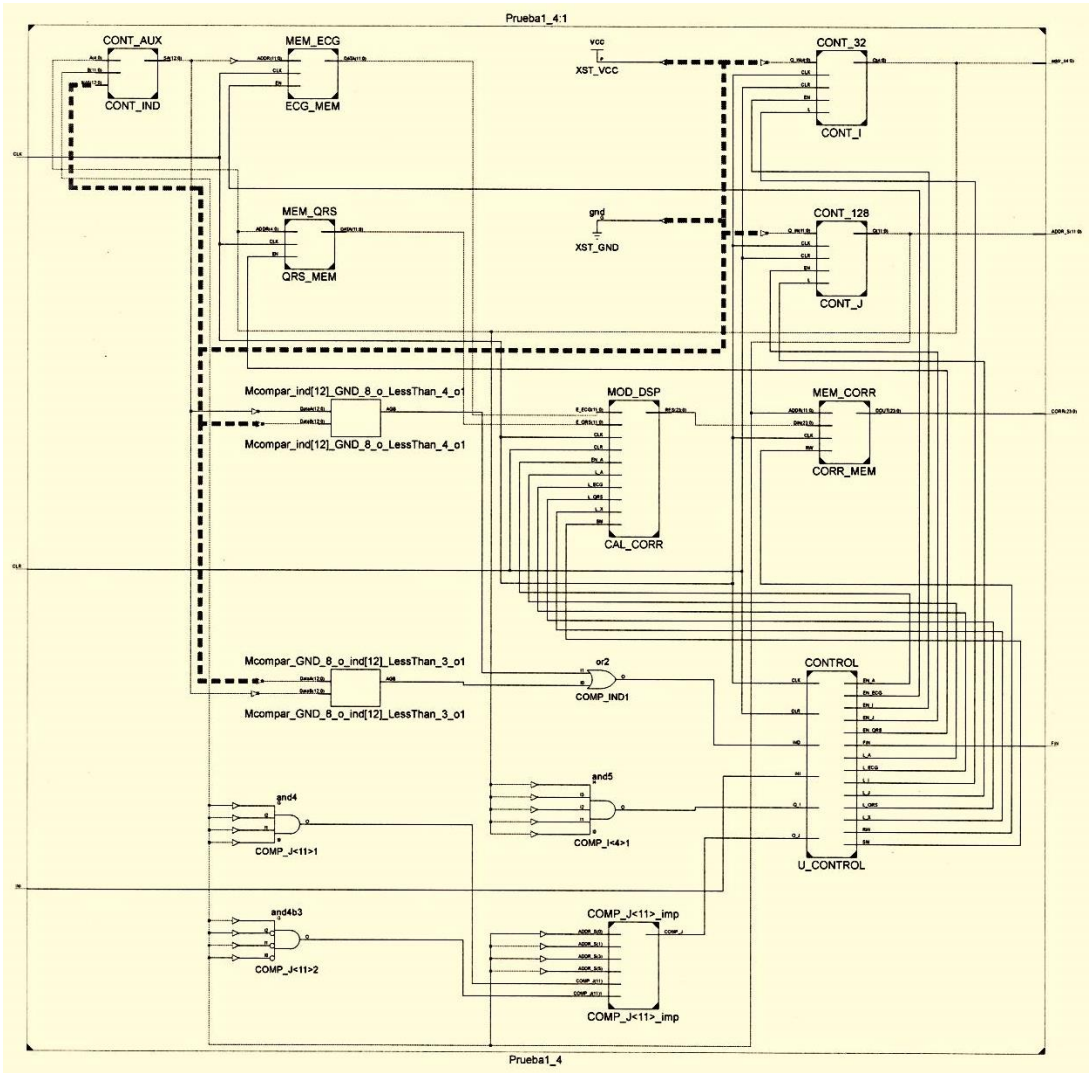


Figura 6.7 Diagrama RTL del core generado por la herramienta ISE WebPACK.

En la figura se pueden apreciar los bloques CONT_32, CONT_128 y CONT_AUX, que corresponden a los bloques CONT_I, CONT_J y CONT_IND del diagrama de ruta de datos respectivamente. De la misma manera, los bloques MEM_QRS, MEM_ECG y MEM_CORR corresponden a los bloques de memoria del mismo nombre en el diagrama de ruta de datos ya mencionado.

Adicionalmente se encuentra el modulo CONTROL que tiene su equivalente con el módulo U_CONTROL, así como las compuertas que tienen su equivalente con los comparadores representados en el diagrama de ruta de datos también como compuertas lógicas.

El único bloque que es diferente a ambos diagramas es el bloque MOD_DSP. Este bloque contiene dentro de él, los bloques X, REG_X y REG_ACC. Estos tres bloques se implementan dentro de uno solo, ya que esto permite hacer uso de los recursos dedicados que se encuentran dentro del FPGA, en este caso en particular, las unidades DSP.

6.3.2. Vista Diseño/Edición en FPGA

La siguiente vista disponible dentro del ISE WebPACK es la vista Diseño/Edición en FPGA. Esta vista es una opción que muestra los compontes físicos que se están utilizando en la síntesis del sistema.

La figura 6.8 muestra el espacio total del FPGA y remarcado en color azul claro, las conexiones entre los componentes que emplea el diseño para funcionar.

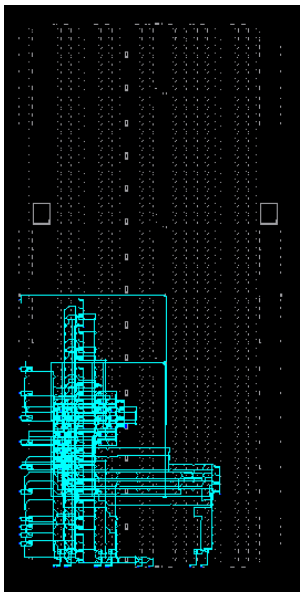


Figura 6.8 Vista Diseño/Edición en FPGA mostrando el espacio ocupado por el core.

En la figura se observa que el diseño completo ocupa solo una porción de la totalidad del espacio disponible del FPGA, lo que da la posibilidad de escalar el sistema, es decir, agregar más módulos al diseño sin cambiar el dispositivito a uno de mayor capacidad.

La figura 6.9 es una vista ampliada a una sección del diseño para poder ver los recursos dedicados que se emplean, tales como las unidades DSP o bloques de memoria RAM dedicada.

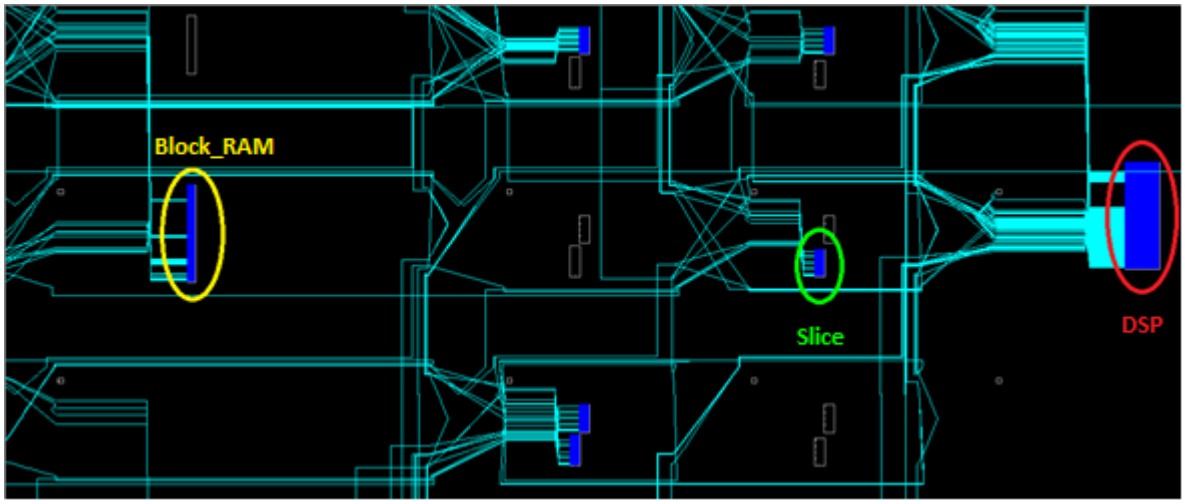


Figura 6.9 Ampliación de la vista Diseño/Edición en FPGA mostrando algunos recursos dedicados.

En esta ampliación se muestran 3 de los principales recursos dedicados empleados en el diseño. Dentro de la elipse de color rojo se encuentra una unidad de tipo DSP, cuya funciona particular es realizar operaciones para procesamiento digital de señales, esta unidad cuenta con un multiplicador y un acumulador, que son la parte fundamental en el procesamiento de la información, ya que la velocidad es un factor a considerar en el procesamiento de señales digitales.

En la elipse de color amarillo se encuentra una unidad de memoria llamado Block_RAM, esta memoria especial permite almacenar información sin utilizar la memoria RAM distribuida que se encuentra dentro de los slice, como se haría de manera habitual en este tipo de diseños, lo que permite reducir el tamaño del diseño de manera considerable.

Finalmente, dentro de la elipse de color verde se encuentran los slice, que son las unidades lógicas con las que cuenta cualquier FPGA. En esta modelo de FPGA existen dos tipos de slice, tipo X y tipo L, dentro de los cuales se encuentran recursos como bloques de memoria RAM distribuida, lógica de acarreo rápido, registros de corrimiento, entre algunos otros.

Cada uno de los módulos que conforman el core de procesamiento se sintetizan de manera física dentro del FPGA usando alguno de los recursos dedicados descritos anteriormente. Dependiendo de la función que realiza cada uno, se utiliza un recurso diferente para que pueda desempeñar de manera más eficiente du función.

En la tabla 6.2 se muestra cada uno de los módulos que conforman el core de procesamiento, así como el recurso dedicado utilizado para su síntesis.

Módulo del core	Recurso dedicado del FPGA
Cont_I	Slice_X
Cont_J	Slice_X
Cont_IND	Slice_X
Mem_QRS	Slice_X
Mem_ECG	Block_RAM
Reg_QRS	Slice_X
Reg_ECG	Slice_X
X	DSP
Reg_X	DSP
Reg_ACC	DSP
Mem_CORR	Block_RAM
Cmp_I	Slice_L
Cmp_J	Slice_L
Cmp_IND	Slice_L
Mux_A	Slice_X
Mux_D	Slice_X
U_Control	Slice_X, Slice_L

Tabla 6.2 Módulos del core de procesamiento y recursos dedicados utilizados del FPGA.

6.3.3. Pruebas del core de manera independiente

Ya depurado el core, se procede a realizar pruebas para asegurarse que funciona correctamente. Para esto se requiere un punto de comparación, es decir, resultados previos para comprobar los obtenidos del core.

Para esto, se elaboró un programa en lenguaje C con ayuda del compilador DEV-C++[43], que lee los valores contenidos en dos archivos. Estos archivos contienen los mismos valores que se encuentran dentro de las memorias QRS y ECG del diseño. En el código se implementa el mismo algoritmo de procesamiento utilizado en el core, es decir, la correlación cruzada, pero usando la sintaxis propia del lenguaje de programación. Los resultados de salida se escriben tanto en pantalla como a archivo para poder visualizarlos y poder usarlos en aplicaciones de graficación.

De manera similar ocurre con el procesamiento de los datos en hardware. Como únicamente se desea comprobar el core de procesamiento se debe realizar una simulación utilizando la herramienta incluida para dicho fin, además de crear un código que realice la escritura de los datos obtenidos a un archivo como en el programa escrito en lenguaje C, para que, de las misma manera, puedan ser usarlos en alguna aplicación de graficación.

En la figura 6.10, 6.11 y 6.12 se ilustran tres gráficas diferentes, correspondientes a una de las señales ECG de la base de datos seleccionada previamente, el resultado de la correlación obtenida mediante el programa en lenguaje C y el resultado de la correlación obtenida en la simulación del core en lenguaje VHDL.

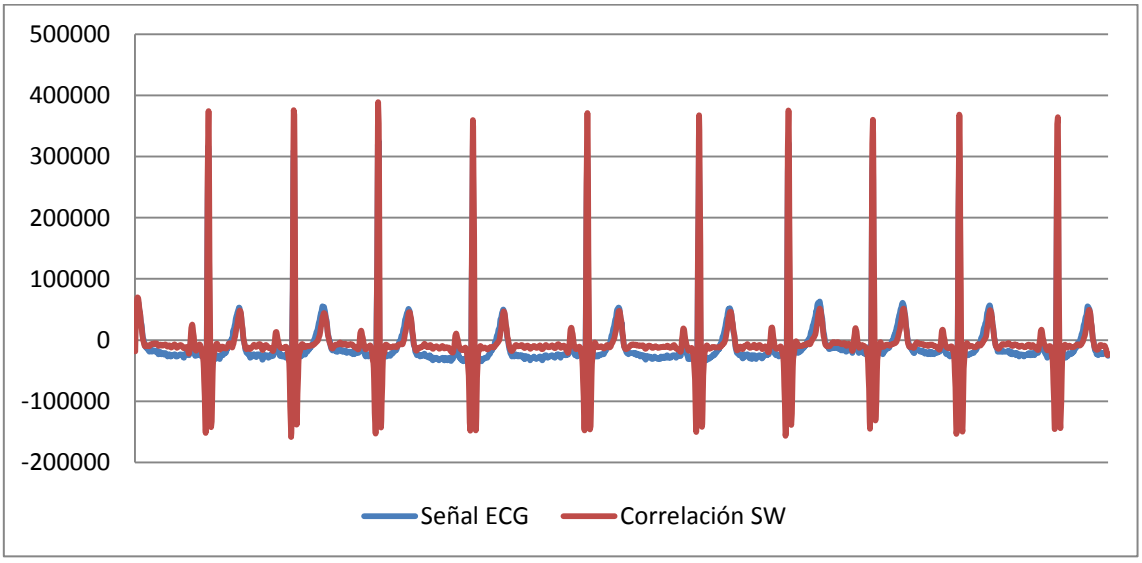


Figura 6.10 Gráficas de la señal ECG y la correlación obtenida por software.

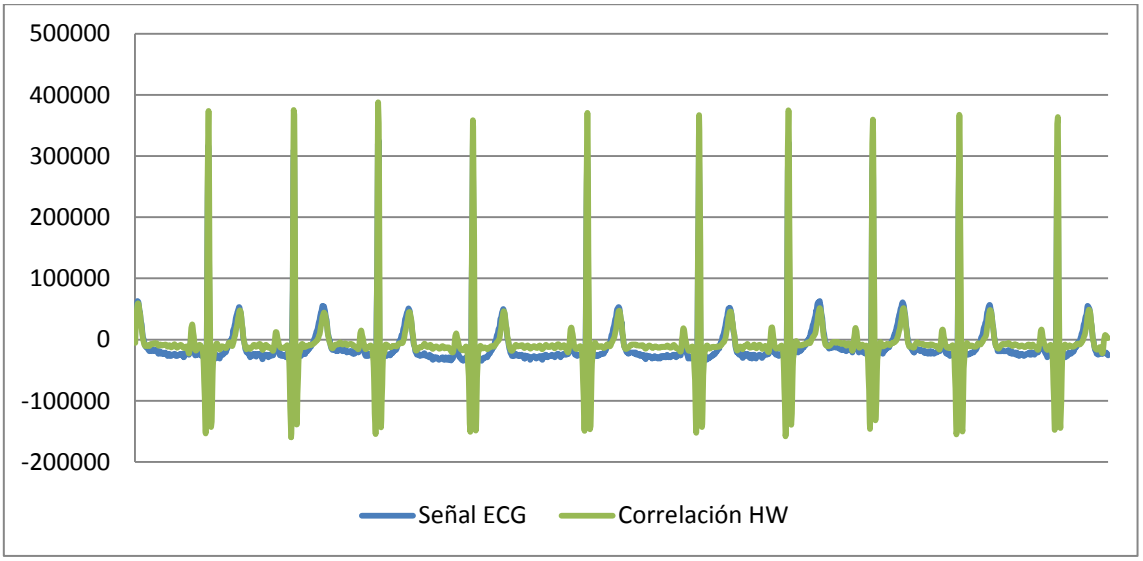


Figura 6.11 Gráficas de la señal ECG y la correlación obtenida por hardware.

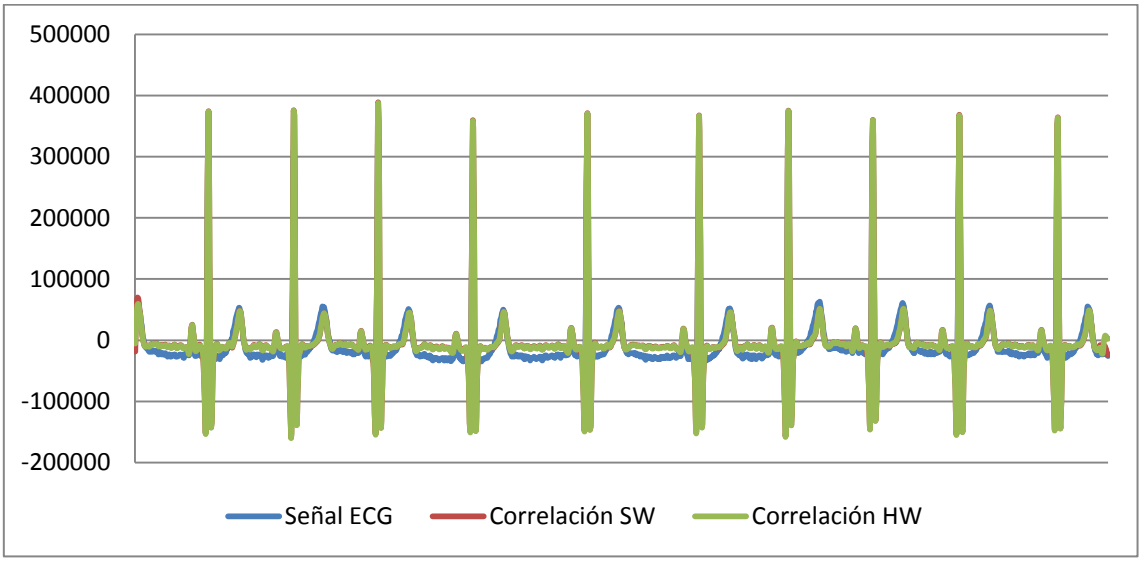


Figura 6.12 Gráficas de la señal ECG y los resultados de ambas correlaciones.

De las figuras anteriores hay que hacer resaltar puntos muy importantes.

En la tercera gráfica donde se sobreponen ambas correlaciones, las mismas se encuentran una sobre otra en muchos puntos, sobre todo en las partes donde se localizan los componentes QRS del ECG. Esto indica que el algoritmo seleccionado de correlación cruzada es capaz de detectar los componentes de interés con gran precisión.

Por tal motivo, no es posible distinguir la gráfica de correlación mediante software, de la gráfica de correlación del core implementado. El motivo de esto es que los valores obtenidos por el core son casi idénticos a los valores obtenidos por el software, lo que indica que el core de procesamiento funciona adecuadamente y los resultados son confiables.

Tanto el código del programa en lenguaje C como el código en lenguaje VHDL para la simulación del core, se incluyen en la sección de Anexos del presente documento. Además se incluye el contenido de los archivos de resultados de cada una de las gráficas de la imagen, en caso de requerir los valores numéricos.

6.4. Inclusión del core como periférico en el sistema embebido

Después de comprobar que el core funciona adecuadamente y que los resultados obtenidos son los correctos, se procede a integrarlo al Sistema Embebido completo junto con otros periféricos que permitan la comunicación, entre la tarjeta de desarrollo donde se encuentra implementado este sistema y la computadora donde se hará el despliegue de la información obtenida.

Para implementar la parte de configuración de los elementos de hardware de la tarjeta se utiliza la herramienta dedicada a la construcción de Sistemas Embebidos, el Enviroment Development Kit, el cual incluye el Xilinx Platform Studio, que es el IDE que proporciona el ambiente de trabajo, IP cores prediseñados, compatibilidad con procesadores como Microblaze y la posibilidad de exportar el diseño para agregar la parte complementaria de software.

En el ambiente de trabajo de XPS, los Sistemas Embebidos son vistos como diseños compuestos de las siguientes partes principales: un microprocesador principal, al que se interconectan los periféricos, mediante un bus de comunicación. Ya que el sistema trabaja bajo este esquema, el core creado previamente, se debe acoplar al sistema como un periférico más, que también se comunicara por medio de un bus de datos con el microprocesador.

En la figura 6.13 se muestra el sistema embebido completo con el core ya integrado dentro del diseño.

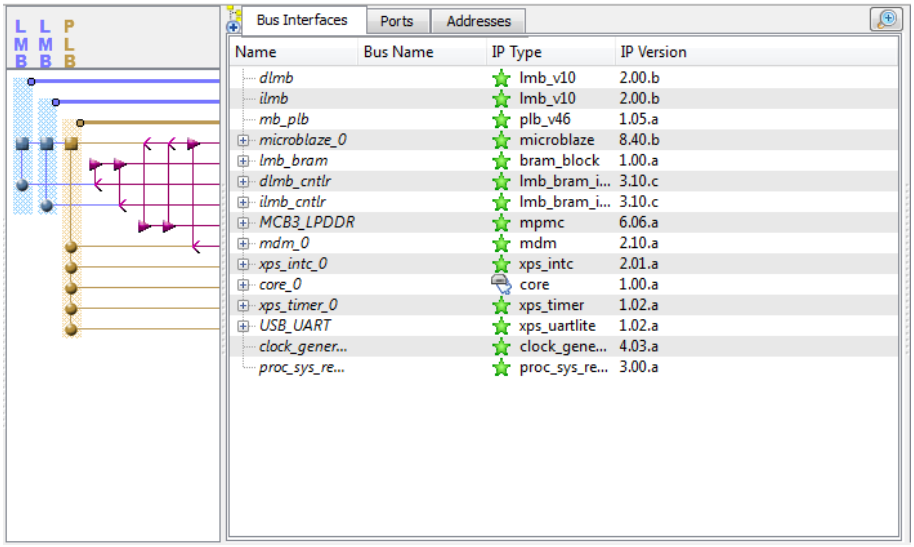


Figura 6.13 Vista del sistema embebido dentro en la herramienta XPS.

Del lado derecho de la imagen se encuentran los tipos de buses presentes en el diseño, así como la interconexión entre cada uno de los periféricos con el microprocesador.

En color marrón se representa el bus de comunicación PLB, que comunica el microprocesador con cada uno de los periféricos del sistema, mientras que en color azul se muestra una interconexión especial, entre el microprocesador y dos bloques especiales de memoria, dlmb y ilmb. Ambos bloques sirven para gestionar el almacenamiento tanto de datos como instrucciones respectivamente, como ocurre en una arquitectura de tipo Harvard.

Las líneas en color rosa, muestran las dependencias entre los bloque especiales de memoria mencionados anteriormente con el microprocesador, ya que toda la información es almacenada en dichos bloques y su gestión debe ser controlada de marera independiente a la del resto de periféricos.

Del lado derecho de la imagen se muestra un listado de los periféricos añadidos al sistema. Asimismo se puede visualizar el nombre el periférico o IP core, el tipo al que pertenecen, y la versión del mismo.

De las 3 opciones mencionadas, la más importante a resaltar es el tipo de periférico que se emplea, ya que existen de diversos tipos. Los tipos de core que se pueden utilizar en XPS son gratuitos, de licencia, de licencia de prueba, en desarrollo o fase de prueba y los personalizados o creados por el usuario. La manera de identificar cada uno es mediante la simbología que se especifica en la misma herramienta.

En la imagen se puede apreciar dos tipos de símbolos que identifican a los IP core del sistema. Los que tienen una estrella en color verde representan IP cores previamente diseñados por el fabricante e incluidos dentro de la herramienta los cuales son de libre uso, es decir, son de licencia gratuita.

Por otro lado, solamente uno de ellos tiene un símbolo diferente al resto de los demás, el cual es una especie de caja gris con un pequeño documento de color blanco en la base. Este símbolo sirve para indicar un IP core creado por el usuario, el cual es el core que se creó previamente y que es el encargado de realizar el procesamiento de la señal.

Para visualizar de una manera más clara la organización del Sistema Embebido, en la figura 6.14, se muestra una representación a modo de diagrama de bloques del mismo sistema.

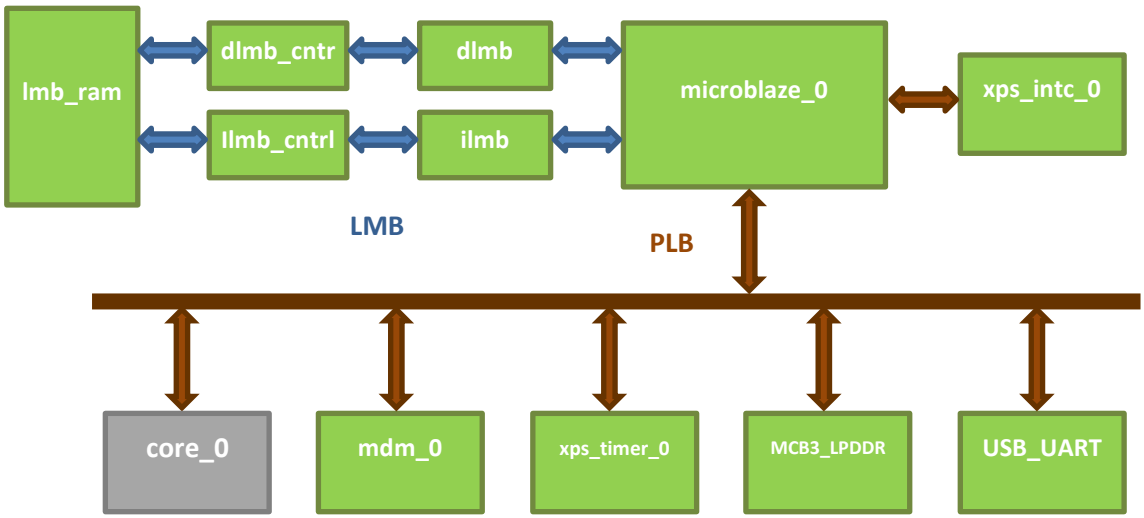


Figura 6.14 Diagrama a bloques del sistema implementado en XPS.

Los bloques en color verde representan los IP cores proporcionados por el fabricante, los cuales tienen licencia de uso gratuita. Uno de ellos es la parte central del sistema, el microprocesador Microblaze, que es el encargado de gestionar todos los módulos restantes mediante el bus de comunicación PLB mostrado en color marrón.

Los módulos especiales de memoria solamente poseen comunicación con el microprocesador mediante el bus LMB, que es un bus dedicado a la comunicación del procesador estos bloques de memoria.

Del mismo modo, el bloque xps_intc_0 posee comunicación especial con el microprocesador, debido a que este módulo en particular se encarga de gestionar lo que se conoce como interrupciones, las cuales son operaciones que solamente el microprocesador puede realizar. Por esta razón se requiere que la comunicación sea lo más rápida y directa posible entre ambos bloques.

Finalmente, el modulo en color gris representa el core personalizado que se diseñó e implemento en la herramienta ISE WebPACK, que ahora se añade al sistema como un periférico más, comunicado igualmente mediante el bus PLB.

6.5. Desarrollo del código fuente para el control del Sistema Embebido

Una vez configuradas las opciones de hardware en XPS, se exporta el diseño completo a la siguiente herramienta del EDK correspondiente al desarrollo del programa de software que ayudara a controlar los módulos previamente agregados, el SDK.

El SDK es el encargado de generar las librerías apropiadas de cada periférico, a fin de poder hacer uso de los mismos. Las librerías que se generan son archivos de cabecera de tipo .h, lo que implica que estas librerías son creadas para ser usadas con lenguajes de programación como C/C++.

Para que el código pueda ejecutarse dentro de la configuración del FPGA, debe existir un sistema operativo que gestione las instrucciones del código elaborado en lenguaje C/C++. El SDK cuenta con dos alternativas de sistema operativo ya precargadas en el software, standalone y xilkernel.

Para este sistema se elige xilkernel, ya que de los dos sistemas disponibles, éste permite el uso de interrupciones, mismas que son necesarias al haber incluido el periférico de gestión de memoria RAM DDR externo, ya que este controlador ocupa una gran cantidad de espacio y funciones especiales que requieren el uso de las interrupciones.

Una característica particular de xilkernel es que, para poder utilizar las interrupciones se requiere que dentro de la configuración de los periféricos, se incluyan los bloques xps_intc_0 y xps_timer, ambos bloques poseen configuraciones especiales que ayudan al microprocesador a reconocer adecuadamente las interrupciones. Por este motivo fue necesario añadir previamente estos dos bloques adicionales cuando se configuro el diseño del sistema en la herramienta XPS.

En la figura 6.15 se muestra una representación de la función que realiza el SDK, tanto en la generación de las librerías, como en la generación del código de control principal.

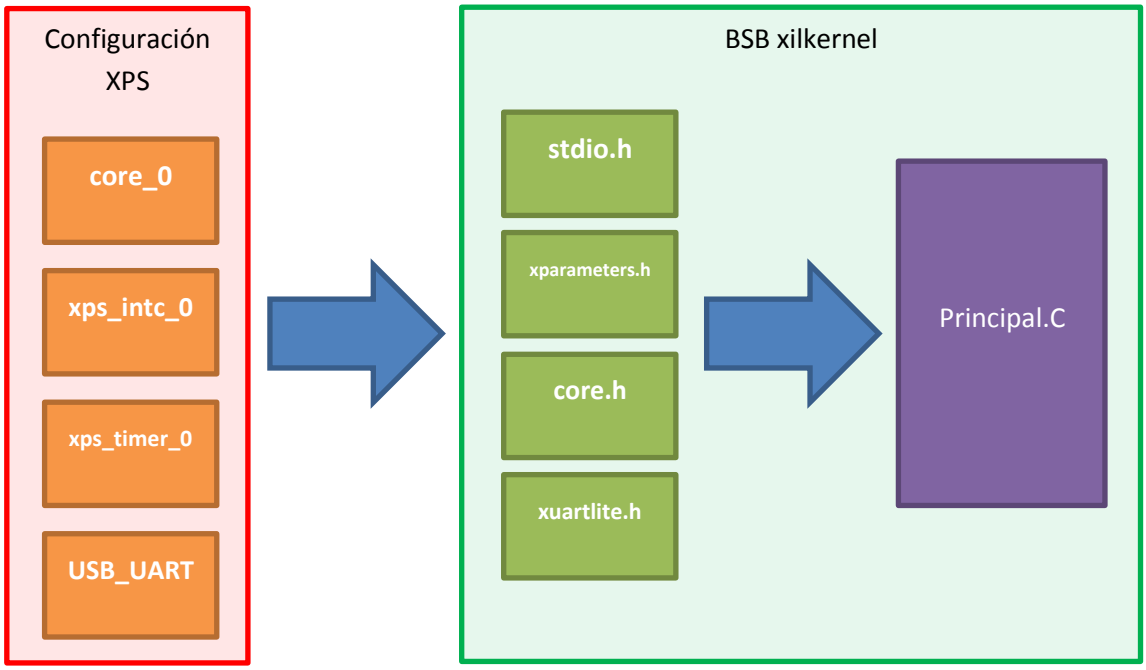


Figura 6.15 Representación de la configuración en cada herramienta del EDK.

De todas las librerías que se generan al definir el BSB, cuatro de ellas son las que se emplean en el código de control principal del sistema: stdio.h, xparameters.h, xuartlite.h y core.h. A continuación se da una breve descripción de la función que cumple cada una de ellas.

stdio.h: Librería estándar de entrada y salida. Dentro de esta librería, se encuentran las mismas funciones que se usan en cualquier compilador para C/C++ como son imprimir datos en pantalla, leer datos del teclado, imprimir saltos de línea, colocar estructuras de control condicionales y de ciclos, entre otras.

xparameters.h: Librería de parámetros. Aquí se encuentra toda la información referente a la configuración de los periféricos en hardware. La información se encuentra definida por medio de macros para dar mayor claridad y facilidad de uso. Algunos de los datos que contiene son, direcciones base de los periféricos, identificadores de dispositivo, frecuencia de trabajo, tipo de acceso, entre otros.

xuartlite.h: Librería asociada al periférico USB_UART. Aquí se definen las funciones de comunicación vía UART, permitiendo hacer lecturas y escrituras de datos desde y hacia el exterior.

core.h: Librería asociada al periférico core_0. Esta librería en particular permite acceder a los registros del core, permitiendo leer los resultados generados durante el procesamiento de la señal para poder ser utilizados posteriormente en la etapa de graficación.

Partiendo de los resultados obtenidos del algoritmo de correlación, se escribe la lógica para separar el complejo QRS del resto de ondas de la señal ECG. Esto se logra usando el valor máximo de la onda R en cada pulso de la señal para definir un umbral y crear un rango o ventana en el cual se leerán los valores contenidos en la señal ECG original.

En la figura 6.16 se muestra una captura de pantalla con un fragmento del código principal, donde se ha definido el umbral y la ventana particular para el ECG analizado.

```
for(i=0;i<2500;i++){
    CORE_mWriteSlaveReg2(CORE, 0, i);
    res = CORE_mReadSlaveReg3(CORE, 0);

    if(res>355000){
        CORE_mWriteSlaveReg2(CORE, 0, i+1);
        res2 = CORE_mReadSlaveReg3(CORE, 0);
        if(res2<355000){
            for(j=i;j<(i+32);j++){

                CORE_mWriteSlaveReg4(CORE, 0, (j-16));
                res = CORE_mReadSlaveReg5(CORE, 0);

                muestras[j-16]=res*1000;

            }
        }
            else
                continue;
    }
}
```

Figura 6.16 Fragmento de código del archivo Principal.c.

El ciclo for mas en el exterior utiliza la función CORE_mReadSlaveReg3 para leer cada valor del core correspondiente al resultado de la correlación en la posición dada por el índice i; donde posteriormente dicho valor es asignado a la variable res.

Luego, se hace una primera comparación del valor contenido en res. Si el valor es menor a 355000 se procede a abrir una ventana de 32 posiciones, donde dicha ventana contiene los valores del complejo QRS que se busca en la señal ECG.

Para abrir esta ventana en la posición indicada, se usa un ciclo for, donde la variable de control j se coloca 16 posiciones antes del valor i. Esto se debe a que, el valor identificado por el índice i es el valor máximo del complejo QRS, que se encuentra justo a la mitad del mismo, y dado que el complejo QRS se define con 32 muestras, implica que el resultado completo abarca tanto las 16 muestras anteriores, como las 16 muestras siguientes.

Los 32 valores se almacenan en un arreglo previamente definido de 2500 posiciones, en la misma posición en la que se encuentran en la señal ECG original. Adicionalmente, cada valor es multiplicado por un valor constante de 1000; esto con la finalidad de que la señal pueda ser apreciada y graficada de mejor manera.

6.5.1. Pruebas del core de manera independiente

El proceso se repite hasta terminar de comparar todos los resultados de la correlación con los valores de la señal ECG. Al final, en el arreglo muestras se obtienen únicamente los valores correspondientes al complejo QRS, habiendo eliminado el resto de los valores de las demás ondas.

En la figura 6.17 se muestra una parte del resultado que se obtiene en la consola de salida de la misma herramienta, donde se pueden apreciar valores de 0's, y valores distintos a 0, que corresponden a un complejo QRS identificado en la señal.

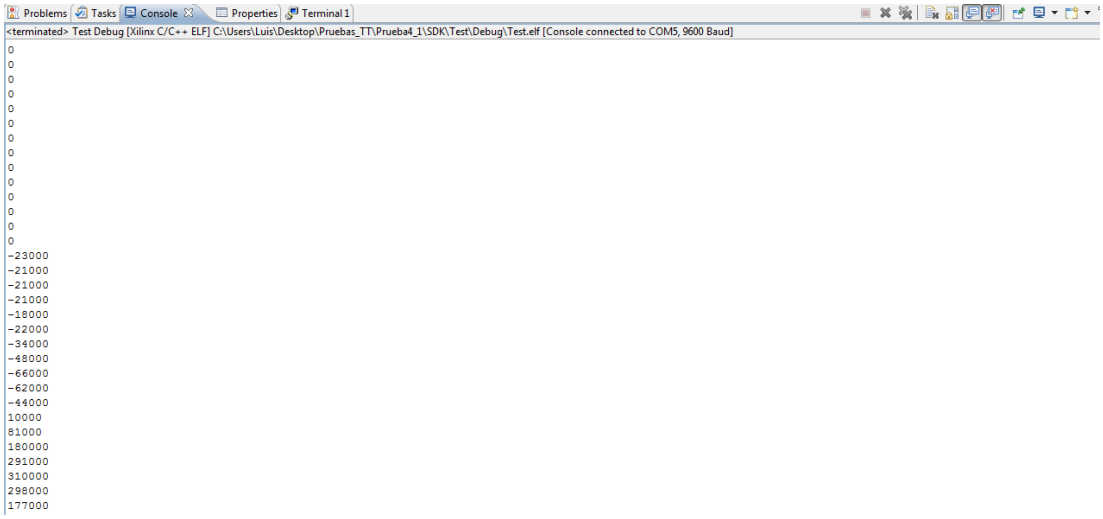


Figura 6.17 Valores contenidos en el arreglo muestras, desplegados en consola.

Es muy importante destacar que los valores mostrados son los valores correspondientes a los complejos QRS de la señal ECG que fue procesada en el core, y no los valores obtenidos de la correlación. Los valores de correlación, tienen la función de ayudar a crear el umbral para identificar los complejos QRS y definir el rango de muestras en las cuales se encuentran; pero los valores que finalmente se muestran deben ser estrictamente los valores de la señal original.

En la figura 6.18 se muestra la gráfica que se obtiene al usar todos los valores obtenidos al probar el sistema completo, con todos los periféricos correspondientes, incluyendo el core de procesamiento. También se incluye la gráfica del ECG procesado para poder apreciar el resultado.

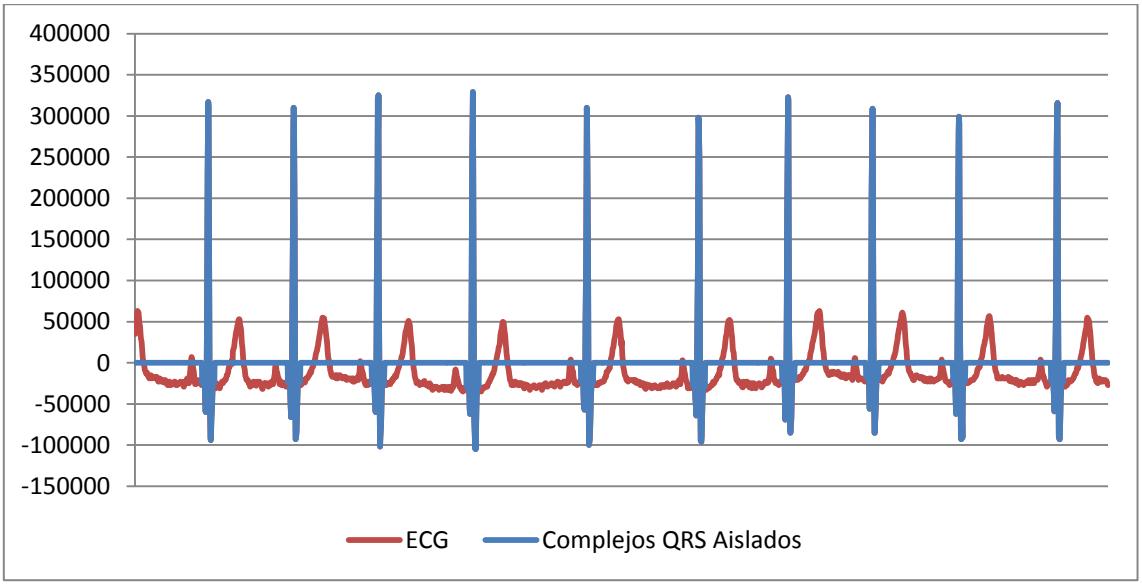


Figura 6.18 Grafica del ECG procesado y grafica de los complejos QRS de la misma.

En la imagen se pueden apreciar líneas de color rojo y azul, donde las líneas rojas representan el ECG procesado por el core creado anteriormente; y las líneas azules representan los fragmentos de la misma señal, pero mostrando únicamente los complejos QRS.

Con esta grafica se comprueba el éxito del sistema embebido, ya que en la gráfica en rojo, no se aprecian los componentes QRS. Esto es debido a que se encuentran cubiertos totalmente por la gráfica en color azul, lo que demuestra el correcto funcionamiento del algoritmo de procesamiento y la identificación de los complejos.

Ya escrito y depurado el código Principal.c, el paso restante es enviar los mismos datos obtenidos mediante consola hacia una computadora para graficarlos mediante una aplicación de graficación propuesta, desarrollada en lenguaje Java, para simular un envío remoto de datos de un punto a otro.

6.6. Pruebas de envío y gráfica en la aplicación Java

El paso final para concluir el proyecto, es enviar los datos de manera remota, usando la interfaz UART de la tarjeta para realizar una conexión con la interfaz USB de la computadora, la que posteriormente transferirá la información a una segunda computadora simulando un envío remoto de información.

En la figura 6.19 se muestra un diagrama de cómo se comunican entre sí los diferentes dispositivos.

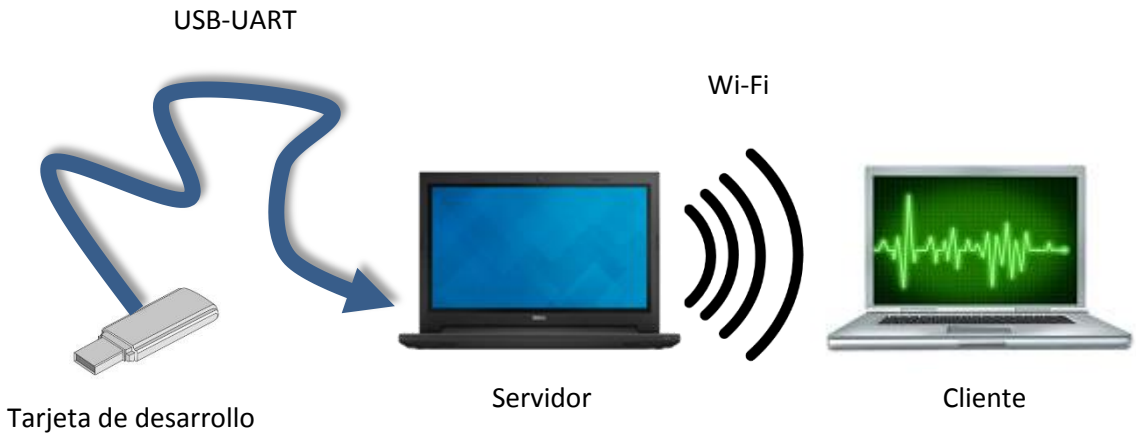


Figura 6.19 Diagrama de comunicación entre los diferentes dispositivos.

En el diagrama se observan dos computadoras, una de tipo servidor, y la otra de tipo cliente. Esta arquitectura en particular está presente, ya que la comunicación entre ambas se lleva a cabo mediante sockets.

Se utiliza la comunicación con sockets, ya que tienen la capacidad de conectarse, tanto de manera local como a través de Internet, lo que brinda un alcance muy grande de comunicación en un entorno abierto y al mismo tiempo permite comprobar su funcionamiento en una red de solo dos computadoras.

La primera parte de la comunicación consiste en leer la información a través de la interfaz UART que posee la tarjeta, mediante un cable adaptador micro USB a USB.

En la figura 6.20 se observa otra porción del código fuente del archivo Principal.c generado con anterioridad en la herramienta SDK.

```
for(i=0;i<2500;i++){

    dato = muestras[i];

    XUartLite_Send(&SERIAL,p,4);
    for(l=0;l<10000;l++){
    }

}
```

Figura 6.20 Porción de código del archivo Principal.c para el envío serial de datos.

Para realizar el envío, se usa un ciclo for que se repite 2500 veces, el mismo número de muestras contenidas en el arreglo muestras. Cada valor del arreglo es asignado a la variable dato, donde posteriormente se envía seriamente mediante la función XUartLite_Send(); que pertenece a la librería xuartlite.h.

Entre cada dato enviado existe un pequeño retardo generado por un ciclo for adicional dentro el primero. Este retraso es necesario, ya que la comunicación de tipo serial es más lenta en comparación con la lectura de los datos; y para evitar que la información se corrompa, se le da un margen de tiempo pequeño para permitirle terminar el envío y realizar el siguiente. Este retardo no debe ser muy grande para evitar que el sistema se vuelva lento.

Los datos enviados, son recibidos por la computadora servidor, donde se encuentra funcionando una aplicación de tipo socket escrita en lenguaje Java, que actúa como la parte de servidor del socket para la comunicación.

La aplicación se compone de dos partes, la lectura de los datos a través del puerto serie junto con su almacenamiento en un arreglo de enteros de 2500 posiciones, y la creación del socket servidor que queda a la espera de petición de conexión de un socket cliente.

Cuando se hace la lectura de los datos de la tarjeta se presenta un pequeño problema relacionado con la longitud de los datos. La función que se encarga de leer la información del puerto serie solamente puede leer datos de 1 byte de longitud, es decir 8 bits; sin embargo la longitud de los datos en la tarjeta son de 24 bits, así que hay que acomodar la información de manera que el dato se pueda reconstruir a su forma original.

En la figura 6.21 se muestra la parte del código que se encarga de reconstruir el dato conforme la función lee cada uno de los 3 bytes que conforman el dato completo. El código fuente de la aplicación se encuentra por completo en la sección de Anexos del presente documento.

```
// Cuando haya datos disponibles se leen y luego se
    // imprime lo recibido en la consola
    case SerialPortEvent.DATA_AVAILABLE:

        try {
            while (i >= 0) {
                int nByte = entrada.read();
                temp = nByte << (i * 8);
                dato = dato | temp;
                i--;
            }
            i = 3;
            leer.setInt(dato);
            System.out.println(dato);
        } catch (IOException e) {
            System.out.println(e);
        }
        break;
```

Figura 6.21 Porción de código del archivo Comunicación.java para la lectura serial de datos.

El procedimiento consiste en un ciclo de 3 iteraciones que realizara operaciones de corrimiento y operaciones lógicas OR. Primero se declara un índice igual a 3, que son el número de iteraciones a realizar. Después, mientras este índice sea mayor a 0 se lee el primer byte recibido por la función entrada.read().

Este valor corresponde a los 8 bits de la parte alta del primer dato, pero el programa los coloca en los 8 bits de la parte baja de la variable temp, así que se debe realizar primero una operación de corrimiento, igual al número de bits multiplicado por el valor del índice del ciclo.

Después se realiza una operación lógica OR entre la variable temp y la variable, dato para dejar intactos los bits restantes y no alterar el valor final. Una vez que se hacen las 3 iteraciones la variable dato, contiene el valor que se encontraba almacenado originalmente en la tarjeta. Dicho valor se almacena en un arreglo de enteros para posteriormente ser enviados al cliente cuando este se conecte al servidor. Este proceso se realiza hasta terminar las 2500 muestras que conforman el ECG.

Cuando se terminan de leer los 2500 valores de la tarjeta, el siguiente paso es crear el socket servidor para que el cliente pueda conectarse y leer la información.

En la figura 6.22 se muestra la forma de establecer los parámetros para la configuración de la conexión del socket. En lenguaje Java sólo es necesario especificar el puerto mediante el cual los clientes podrán acceder al servidor.

```
public void initServer() {
    try {
        ServerSocket skServidor = new ServerSocket(PUERTO);
        System.out.println("Escucho el puerto " + PUERTO);

            Socket skCliente = skServidor.accept(); // Crea objeto
            System.out.println("Sirvo al cliente ");
            OutputStream aux = skCliente.getOutputStream();
            DataOutputStream flujo = new DataOutputStream(aux);

           for(int f=0;f<2500;f++){
            flujo.writeInt(leer.getInt());
           }
           skServidor.close();

    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

Figura 6.22 Porción de código del archivo Comunicación.java para la creación del socket servidor.

En Java, el resto de los parámetros se configuran automáticamente, así que únicamente se debe estar a la espera de petición de conexión de un cliente. Este tipo de socket servidor en particular, es un servidor de tipo bloqueante, ya que sólo puede atender a un solo cliente a la vez.

Una vez establecida la conexión con el cliente mediante la función skServidor.accept(), se crea un flujo de salida de datos para enviar la información hacia el cliente. Para ello se utiliza un ciclo for que lee cada valor del arreglo donde se encuentran almacenadas las muestras y las envía mediante el flujo de salida que se creó anteriormente. Esto se repite para los 2500 valores que contiene el arreglo, y al finalizar se termina la conexión con el cliente.

Ahora del lado del cliente sucede una situación similar, ya que la aplicación del cliente también se compone de dos partes, establecer la conexión del socket tipo cliente, y la generación de la gráfica con los valores obtenidos del servidor

El primer paso es crear el socket tipo cliente en la segunda computadora denominada cliente, y realizar la petición de conexión con el servidor que se encuentra en la primera computadora denominada servidor.

En la figura 6.23 se muestra la parte del código encargada de crear el socket tipo cliente y establecer la conexión con el socket servidor que se encuentra esperando la petición de parte del cliente. De nueva cuenta se hace mención que la conexión entre ambas computadoras es mediante una pequeña red inalámbrica creada únicamente con las dos computadoras que, en este caso, intercambiaran información.

```
        try {
            Socket skCliente = new Socket(HOST, PUERTO);
            InputStream aux = skCliente.getInputStream();
            DataInputStream flujo = new DataInputStream(aux);

            for (int f = 0; f < 2500; f++) {
                line_chart_dataset.addValue(flujo.readInt(), "valor", f);
                System.out.println(flujo.readInt());
            }
            skCliente.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

        return line_chart_dataset;
    }
```

Figura 6.23 Porción de código del archivo Ventana.java en la computadora cliente.

Como se puede observar en la imagen, la creación del socket tipo cliente es muy similar a la del socket tipo servidor, salvo algunas diferencias que es importante remarcar.

A diferencia con el servidor, en el cliente es necesario conocer la dirección IP que tiene asignada el servidor, ya que es necesario saber a dónde se enviara la solicitud de conexión, y es necesario especificarla dentro de los parámetros de la función Socket(), donde después de especificar los parámetros de conexión, se crea un flujo de entrada de datos, que reciba la información proveniente del servidor.

Posteriormente se utiliza un ciclo for para leer cada uno de los valores que se reciben y agregarlos a un arreglo especial de tipo Dataset, donde dichos valores son usados para crear la gráfica de resultados final.

Finalmente, una vez que toda la información llega al cliente, el paso final consiste en generar la gráfica para poder ver los datos de una mejor manera. Debido a que en el servidor se hizo uso de un hilo para comprobar continuamente cuando los datos están disponibles, por lo que para que la gráfica se muestre en pantalla es necesario detener el proceso que se está ejecutando en la computadora servidor. De esta manera se dibuja con los resultados obtenidos.

En la figura 6.24 se muestra la gráfica final que es generada por la aplicación de Java de la computadora cliente. La gráfica es igual a la que fue generada con los datos que se obtuvieron de la consola en la herramienta SDK, con lo que se garantiza que la información que se recibe es la misma que se obtuvo al procesar la información en el core.

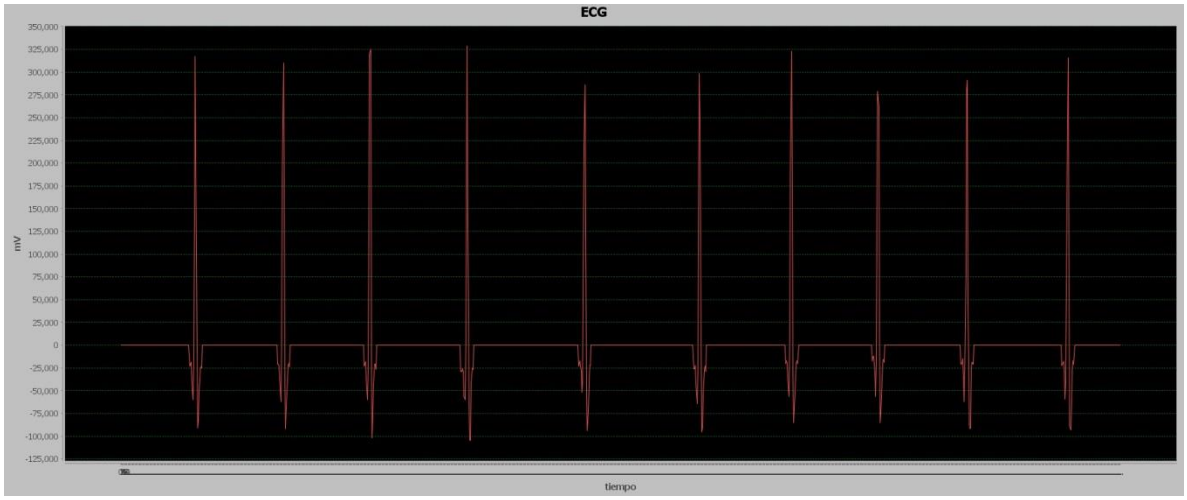


Figura 6.24 Gráfica obtenida en la aplicación de la computadora cliente.

Capítulo 7
Conclusiones

Del desarrollo del presente Trabajo Terminal se obtuvieron las siguientes conclusiones:

- Al usar FPGA para realizar la implementación, se tuvieron muchas ventajas en cuanto a flexibilidad, ya que es un dispositivo totalmente configurable; posee escalabilidad al poder agregar más bloques funcionales al diseño y modificarlos acorde a las necesidades.

- El uso de un microprocesador de tipo soft-core como es Microblaze permite una gran compatibilidad con diversos sistemas, gracias a su característica de ser un microprocesador con una arquitectura de 32 bits, tal y como los procesadores de tipo hard-core comerciales que poseen dispositivos como computadoras, laptops, teléfonos inteligentes, etc.

- Al realizar pruebas en software y en hardware, se comprobó la correcta funcionalidad del core, ya que se obtuvieron resultados muy similares en ambas pruebas.

- Gracias al uso de los recursos dedicados que posee el FPGA, se pudo sintetizar un diseño más pequeño, con lo que se reduce el tamaño del diseño sin tener que hacer uso de un dispositivo de mayor capacidad.

- Se añadieron etapas de segmentación al diseño, lo que permite tener una mayor frecuencia de reloj, lo que da como resultado, una mayor velocidad de procesamiento del sistema.

- Para poder desarrollar esta tesis, fue necesario contar con conocimientos de varias áreas de la computación como son, diseño de sistemas digitales, arquitectura de computadoras, diseño de sistemas embebidos, electrónica analógica, análisis de algoritmos, redes de computadoras y comunicaciones en red.

Capítulo 8

Trabajo a Futuro

Dadas las dimensiones del proyecto, es posible expandir aún más el proyecto, desarrollando las etapas que no fueron incluidas en la implementación.

Algunos posibles puntos a desarrollar son los siguientes:

- Diseñar e implementar un módulo de adquisición de datos. El modulo debe ser diseñado para poder monitorear la actividad del corazón de un paciente en tiempo real. A su vez la señal adquirida debe ser acondicionada para que pueda ser enviada al Sistema Embebido aquí desarrollado y procesarse para obtener los resultados deseados.

- Implementar algún algoritmo de reconocimiento de patrones o de inteligencia artificial, que permita detectar anomalías en los resultados obtenidos después de procesar la señal ECG. Esto permitiría detectar alguna posible enfermedad y si se requiere de atención especializada de urgencia.

- Implementar un sistema de comunicación en red que cuente con mayores opciones como pueden ser consulta de resultados desde cualquier tipo de dispositivo, agregar seguridad en la transmisión de información a través de la red, el cual puede incluir tipos de usuarios que cuenten con claves de acceso.

- Diseñar e implementar otros tipos de core similares que puedan procesar otros tipos de señales como pueden ser señales encefalografícas (cerebro), trasmisión de voz, procesamiento de video, edición de imágenes (aplicación de filtros), entre otros.

Capítulo 9

Referencias

[1] INEGI. (2013/Oct/12). [Online]. Disponible: http://www.inegi.org.mx.

[2] INEGI. (2013/Oct/12). [Online]. Disponible: http://www3.inegi.org.mx/sistemas/sisept/default.aspx?t=msal22&s=est&c=22482.

[3] INEGI. (2013/Oct/12). [Online]. Disponible: http://www3.inegi.org.mx/sistemas/sisept/default.aspx?t=msal23&s=est&c=22483.

[4] INEGI. (2013/Oct/12). [Online]. Disponible: http://www3.inegi.org.mx/sistemas/sisept/default.aspx?t=msal49&s=est&c=22529.

[5] INEGI. (2013/Oct/12). [Online]. Disponible: http://www3.inegi.org.mx/sistemas/sisept/Default.aspx?t=mdemo107&s=est&c=23587.

[6] P. Rickert and W. Krenik, "Cell phone integration: Sip, soc, and pop," IEEE Design & Test of Computers, pp. 188–195, 2006.

[7] M. Nakkar, "Design based tutorials for system on chip teachings," International Conference on Engineering Education (ICEED 2009), pp. 117–121, 2009.

[8] A. Alsolaim, J. Becker, M. Glesner, and J. Starzyk, "Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems," Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Machines, pp. 205–214, 2000.

[9] J. O. Hamblen, "Using second generation sopc boards for student desingn projects," International Conference on Microelectronic Systems Education (MSE'05, pp. 69–70, 2005.

[10] L. Fei-yu, J. Xi-xiang, G. Yu-Hui, Z. Jian-chuan, Q. Wei-ming, J. Lan, W. Yan-Yu, and M. Yun.hai, "System on programmable chip development system," Second International Workshop on Education Technology and Computer Science, pp. 471–474, 2010.

[11] J. G. Thong, I. D. L. Anderson, and M. A. S. Khalid, "Soft core processors for embedded systems," International Conference on Microelectronics, pp. 170–173, 2006.

[12]Arturo Ramo García. (2013/Sep/02). [Online]. Disponible: http://www.aplicaciones.info/naturales/natura18.htm.

[13] Gretchen Espinoza, Jimena Palma. Plantilla Ethereal. (2013/Sep/03). [Online]. Disponible: http://udlasistemabiologico.blogspot.mx/2010/09/electrocardiograma-ecg.html.

[14] FREE SCIENCES. (2013/Sep/06). [Online]. Disponible: http://elmercaderdelasalud.blogspot.mx/2012/02/el-corazon-ix.html

[15] Gerardo Pozas Garza. . (2013/Sep/08). [Online]. Disponible: http://www.hsj.com.mx/media/44215/el_electrocardiograma_normal._parte2.pdf

[16] David-Arturo Gutiérrez-Begovich, Raúl Ruiz-Meza, "Equipo biomédico con telemetría diseñado para las áreas rurales" Científica Vol. 10 Núm. ESIME-IPN. ISSN 1665-0654. México. 2006. pp.185-190

[17] F. J. Barrera Hernández, J. Castañón García. "Diseño de un electrocardiógrafo para el laboratorio de bioacústica de la ESIME Zacatenco". Tesis Licenciatura. ESIME Zacatenco. Cuidad México. México. Abril. 2012.

[18] J. H. Puebla Lomas. "Electrocardiógrafo para aplicaciones en medicina". Tesis Maestría. ESIME Zacatenco. Cuidad México. México. Septiembre. 2010.

[19] J. C. Moctezuma Eugenio. "Diseño de sistemas empotrados en FPGAs". Instituto Nacional de Astrofísica, Óptica y Electrónica. Departamento de Ciencias Computacionales. Mayo. 2012.

[20] *Guía tecnológica 17: Electrocardiógrafo.* México. Junio. 2006, pp. 4.

[21] Xilinx. (2013/Sep/12). [Online]. Disponible: http://www.xilinx.com/products/silicon-devices/fpga/index.htm.

[22] Altera. (2013/Sep/12). [Online]. Disponible: http://www.altera.com/products/selector/psg-index.html.

[23] *Extended Spartan-3A Family Overview*. Febrero. 2011. pp. 1.

[24] *Extended Spartan-6 Family Overview*. Octubre. 2011. pp. 1-2.

[25] Tektronix. (2013/Sep/28). [Online]. Disponible: http://www.tek.com/signal-generator/afg3000-function-generator

[26] MathWorks. (2013/Sep/28). [Online]. Disponible: http://www.mathworks.com/products/matlab/

[27] Floyd Harriott. ECG waveform generator for Matlab/Octave. PhysioNet. (2013/Sep/28). [Online]. Disponible: http://physionet.org/physiotools/matlab/ECGwaveGen/

[28] PhysioNet. (2013/Sep/28). [Online]. Disponible: http://physionet.org/physiobank/database/#ecg

[29] Laguna P, Mark RG, Goldberger AL, Moody GB. A Database for Evaluation of Algorithms for Measurement of QT and Other Waveform Intervals in the ECG. Computers in Cardiology. 1997. pp. 673-676.

[30] Xilinx. (2013/Oct/12). [Online]. Disponible: http://www.xilinx.com/tools/microblaze.htm

[31] Xilinx. LogiCORE IP Processor Local Bus (PLB) v4.6 (v1.05a). Septiembre. 2010. pp. 1

[32] Xilinx. LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c). Abril. 2010. pp. 1

[33] Avnet. (2014/Feb/12). [Online]. Disponible: http://www.em.avnet.com/en-us/design/drc/Pages/Xilinx-Spartan-6-FPGA-LX9-MicroBoard.aspx

[34] Xilinx. (2013/Sep/12). [Online]. Disponible: http://www.xilinx.com/products/boards-and-kits/AES-S6MB-LX9-image.htm.

[35] Xilinx. (2014/Feb/12). [Online]. Disponible: http://www.xilinx.com/products/design-tools/ise-design-suite.html

[36] Xilinx. (2014/Feb/12). [Online]. Disponible: http://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.html

[37] Xilinx. (2014/Feb/12). [Online]. Disponible: http://www.xilinx.com/tools/platform.htm

[38] Xilinx. (2014/Feb/12). [Online]. Disponible: http://www.xilinx.com/tools/xps.htm

[39] Xilinx. (2014/Feb/12). [Online]. Disponible: http://www.xilinx.com/tools/sdk.htm

[40] Netbeans. (2014/Feb/12). [Online]. Disponible: https://netbeans.org/features/ide/index.html

[41] JFree. (2014/Feb/12). [Online]. Disponible: http://www.jfree.org/jfreechart/

[42] JFree. (2014/Feb/12). [Online]. Disponible: http://www.jfree.org/jcommon/

[43] Bloodshed. (2014/Feb/12). [Online]. Disponible: http://www.bloodshed.net/devcpp.html

Capítulo 10

Glosario

ADC: Analog-to-Digital Converter o Convertidor Analógico-Digital.

CMT: Clock Management Tile o Mosaico de Gestión de Reloj

Core: Núcleo o parte central que controla una tarea o zona específica del sistema.

CPU: Central Processing Unit o Unidad Central de Procesamiento.

DAC: Digital-to-Analog Converter o Convertidor Digital-Analógico.

DCM: Distribution Clock Management o Gestor de Distribucion de Reloj

DSP: Digital Signal Process o Procesamiento Digital de Señales.

E/S: Abreviación de Entrada/Salida.

ECG: Registro gráfico de la actividad eléctrica del corazón.

EDK: Embedded Development Kit o Kit de Desarrollo de Embebidos.

Ethernet: Estándar o norma de redes de área local para computadoras.

FIR: Finite Impulse Response o Respuesta Finita al Impulso.

FPB: Filtro Pasa Banda.

FPGA: Dispositivo lógico programable compuesto por bloques de celdas lógicas.

FSL: Fast Simplex Link o Enlace Simple Rápido.

GHz: Abreviación de Giga-Hertz.

GS/s: Giga-Muestras por Segundo.

Hz: Abreviación de Hertz.

I2C: Inter-Integrated Circuit o Inter-Circuitos Integrados.

IDE: Integrated Development Environment o Entorno de Desarrollo Integrado.

IIR: Infinite Impulse Response o Respuesta Infinita al Impulso.

IP: Intelectual Propietary o Propiedad Intelectual.

KB: Abreviación de Kilo-Byte.

KHz: Abreviación de Kilo-Hertz.

MATLAB: MATrix LABoratory o Laboratorio de Matrices.

MHz: Abreviación de Mega-Hertz.

MS/s: Mega-Muestras por Segundo.

PLB: Processor Local Bus o Bus Local del Procesador.

PLL: Phase-Locked Loops o Ciclo de Bloqueo de Fase

Pmod: Peripherical Module o Módulo Periférico

RAM: Random Access Memory o Memoria de Acceso Aleatorio.

ROM: Read Only Memory o Memoria de Sólo Lectura.

SMA: SubMiniature version A

SoC: System on Chip o Sistemas en un Chip.

SoPC: System on Programable Chip o Sistemas en un Chip Programable.

SPI: Serial Peripheral Interface o Interfaz Periférica Serial.

UART: Universal Asynchronous Receiver-Transmitter o Transmisor-Receptor Asíncrono Universal.

USB: Universal Serial Bus o Bus Serial Universal.

VHDL: VHSIC Hardware Description Language o VHSIC Lenguaje de Descripción de Hardware.

VHSIC: Very High Speed Integrated Circuit o Cuicuito Integrado de Muy Alta Velocidad.

Anexos

Códigos en VHDL del Core del Sistema Embebido

Cont_32.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CONT_32 is
   Port ( CLK  : in  STD_LOGIC;
          CLR  : in  STD_LOGIC;
          EN   : in  STD_LOGIC;
          L    : in  STD_LOGIC;
          Q_IN : in  STD_LOGIC_VECTOR (4 downto 0);
          Q    : out  STD_LOGIC_VECTOR (4 downto 0));
end CONT_32;

architecture DESIGN of CONT_32 is
SIGNAL CON : STD_LOGIC_VECTOR (4 downto 0);
begin

CONTAR : PROCESS (CLK, CLR)
BEGIN
    IF (CLR = '1')THEN
        CON <= (OTHERS => '0');
  ELSIF (RISING_EDGE(CLK))THEN
       IF (EN = '1')THEN
          CON <= CON + 1;
       ELSIF(L = '1')THEN
          CON <= Q_IN;
       END IF;
             END IF;
END PROCESS CONTAR;
    Q <= CON;
end DESIGN;
```

Cont_128.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CONT_128 is
   Port ( CLK   : in  STD_LOGIC;
          CLR   : in  STD_LOGIC;
          EN    : in  STD_LOGIC;
          L     : in  STD_LOGIC;
          Q_IN :  in  STD_LOGIC_VECTOR (11 downto 0);
          Q     : out  STD_LOGIC_VECTOR (11 downto 0));
end CONT_128;

architecture DESIGN of CONT_128 is
SIGNAL CON : STD_LOGIC_VECTOR (11 downto 0);
begin

CONTAR : PROCESS (CLK, CLR)
BEGIN
    IF (CLR = '1')THEN
       CON <= (OTHERS => '0');
    ELSIF (RISING_EDGE(CLK))THEN
       IF (EN = '1')THEN
             CON <= CON + 1;
       ELSIF(L = '1')THEN
             CON <= Q_IN;
       END IF;
             END IF;
END PROCESS CONTAR;
    Q <= CON;
end DESIGN;
```

Cont_AUX.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CONT_AUX is
   Port (A    : in STD_LOGIC_VECTOR (4 downto 0);
         B    : in STD_LOGIC_VECTOR (11 downto 0);
         N16  : in STD_LOGIC_VECTOR (12 downto 0);
         SA   : out STD_LOGIC_VECTOR (12 downto 0));
end CONT_AUX;

architecture DESIGN of CONT_AUX is
begin

       SA <=  "00000000"&A + ('0'&B + N16);
end DESIGN;
```

Mem_ECG.vhd

```
library IEEE;
LIBRARY WORK;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use WORK.PAQUETE.ALL;

entity MEM_ECG is
  GENERIC(
       NBITS_ADDR : INTEGER := 12;
       NBITS_DATA : INTEGER := 12
       );
   Port (CLK  : in  STD_LOGIC;
         EN    : in  STD_LOGIC;
         ADDR : in  STD_LOGIC_VECTOR (NBITS_ADDR-1 downto 0);
         DATA : out STD_LOGIC_VECTOR (NBITS_DATA-1 downto 0));
end MEM_ECG;

architecture DESIGN of MEM_ECG is
SIGNAL MEM : M_ECG := LLENAR_ECG("ecg1.txt");
SIGNAL SAL : STD_LOGIC_VECTOR (NBITS_DATA-1 downto 0);
begin
    SAL <= MEM(conv_integer(ADDR));
    -- LECTURA DE MEMORIA
    PMEM : PROCESS( CLK )
    BEGIN
       IF( RISING_EDGE(CLK) )THEN
             IF( EN = '1' )THEN
                DATA <= SAL;
             END IF;
       END IF;
END PROCESS PMEM;
end DESIGN;
```

Mem_QRS.vhd

```
library IEEE;
LIBRARY WORK;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use WORK.PAQUETE.ALL;

entity MEM_QRS is
    GENERIC(
             NBITS_ADDR : INTEGER := 5;
             NBITS_DATA : INTEGER := 12
             );
   Port ( CLK : in  STD_LOGIC;
          EN      : in  STD_LOGIC;
          ADDR : in  STD_LOGIC_VECTOR (NBITS_ADDR-1 downto 0);
          DATA : out STD_LOGIC_VECTOR (NBITS_DATA-1 downto 0));
end MEM_QRS;

architecture DESIGN of MEM_QRS is
SIGNAL MEM : M_QRS := LLENAR_QRS("patron_qrs.txt");
SIGNAL SAL : STD_LOGIC_VECTOR (NBITS_DATA-1 downto 0);
begin
  SAL <= MEM(conv_integer(ADDR));
             -- LECTURA DE MEMORIA
  PMEM : PROCESS( CLK )
  BEGIN
       IF( RISING_EDGE(CLK) )THEN
             IF( EN = '1' )THEN
                DATA <= SAL;
             END IF;
       END IF;
 END PROCESS PMEM;
end DESIGN;
```

Mem_CORR.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity MEM_CORR is
    GENERIC(
            NBITS_ADDR : INTEGER := 12;
            NBITS_DATA : INTEGER := 24
            );
 Port (CLK : in  STD_LOGIC;
        RW    : in  STD_LOGIC;
        ADDR: in  STD_LOGIC_VECTOR (NBITS_ADDR-1 downto 0);
        DIN    : in  STD_LOGIC_VECTOR (NBITS_DATA-1 downto 0);
        DOUT: out STD_LOGIC_VECTOR (NBITS_DATA-1 downto 0));
end MEM_CORR;

architecture DESIGN of MEM_CORR is
TYPE MEM_TYPE IS ARRAY (0 TO (2**NBITS_ADDR-1)) OF STD_LOGIC_VECTOR(DIN'RANGE);
SIGNAL MEM : MEM_TYPE;
begin
    PMEM : PROCESS( CLK )
    BEGIN
        IF( RISING_EDGE(CLK) )THEN
            IF( RW = '1' )THEN
                MEM(CONV_INTEGER(ADDR)) <= DIN;   -- ESCRITURA DE MEMORIA
            ELSE
                DOUT <= MEM(CONV_INTEGER(ADDR));  -- LECTURA DE MEMORIA
            END IF;
        END IF;
    END PROCESS PMEM;
end DESIGN;
```

Mod_DSP.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity MOD_DSP is
    Port ( CLK :     in  STD_LOGIC;
           CLR :     in  STD_LOGIC;
           E_QRS : in  STD_LOGIC_VECTOR (11 downto 0);
           E_ECG : in  STD_LOGIC_VECTOR (11 downto 0);
           L_QRS : in  STD_LOGIC;
           L_ECG : in  STD_LOGIC;
           L_X :     in  STD_LOGIC;
           EN_A :  in  STD_LOGIC;
           L_A :    in  STD_LOGIC;
           SM :     in  STD_LOGIC;
           RES :    out  STD_LOGIC_VECTOR (23 downto 0));
end MOD_DSP;

architecture DESIGN of MOD_DSP is
SIGNAL R_QRS : STD_LOGIC_VECTOR (11 downto 0);
SIGNAL R_ECG : STD_LOGIC_VECTOR (11 downto 0);
SIGNAL E_X   :   STD_LOGIC_VECTOR (23 downto 0);
SIGNAL R_X   :   STD_LOGIC_VECTOR (23 downto 0);
SIGNAL R_ACC : STD_LOGIC_VECTOR (23 downto 0);
SIGNAL MUX   :  STD_LOGIC_VECTOR (23 downto 0);
CONSTANT INI : STD_LOGIC_VECTOR (23 downto 0) := (OTHERS=>'0');  --CTE DE 24 0's PARA EL MUX Y EL ACUMULADOR
```

```
begin
-------------------------------------------------------------------
--              REGISTRO QRS
-------------------------------------------------------------------
REG_QRS : PROCESS(CLK, CLR)
BEGIN
    IF(CLR = '1')THEN
       R_QRS <= (OTHERS=>'0');
    ELSIF(RISING_EDGE(CLK))THEN
              IF(L_QRS = '1')THEN
                 R_QRS <= E_QRS;
              END IF;
    END IF;
END PROCESS REG_QRS;
-------------------------------------------------------------------
--              REGISTRO QRS
-------------------------------------------------------------------
REG_ECG : PROCESS(CLK, CLR)
BEGIN
   IF(CLR = '1')THEN
       R_ECG <= (OTHERS=>'0');
   ELSIF(RISING_EDGE(CLK))THEN
              IF(L_ECG = '1')THEN
                 R_ECG <= E_ECG;
              END IF;
   END IF;
END PROCESS REG_ECG;
-------------------------------------------------------------------
--              BLOQUE X
-------------------------------------------------------------------
       E_X <= R_QRS * R_ECG;     ---------MULTIPLICACIÓN DE VALORES
-------------------------------------------------------------------
--              REGISTRO X
-------------------------------------------------------------------
REG_P : PROCESS(CLK, CLR)
BEGIN
   IF(CLR = '1')THEN
       R_X <= (OTHERS=>'0');
   ELSIF(RISING_EDGE(CLK))THEN
              IF(L_X = '1')THEN
                 R_X <= E_X;
              END IF;
   END IF;
END PROCESS REG_P;
-------------------------------------------------------------------
--              REGISTRO ACC
-------------------------------------------------------------------
ACC : PROCESS(CLK, CLR)
BEGIN
   IF(CLR = '1')THEN
       R_ACC <= (OTHERS=>'0');
   ELSIF(RISING_EDGE(CLK))THEN
              IF(EN_A = '1')THEN
                 R_ACC <= R_ACC + R_X;
              ELSIF(L_A = '1')THEN
                 R_ACC <= INI;
              END IF;
   END IF;
END PROCESS ACC;
-------------------------------------------------------------------
--              MULTIPLEXOR D
-------------------------------------------------------------------
              MUX <= INI WHEN (SM = '0') ELSE R_ACC;  --MULTIPLEXOR
              RES <= MUX;
end DESIGN;
```

Control.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CONTROL is
   Port ( CLK : in  STD_LOGIC;
          CLR : in  STD_LOGIC;
          INI : in  STD_LOGIC;
          Q_I : in  STD_LOGIC;
          Q_J : in  STD_LOGIC;
          IND : in  STD_LOGIC;
          EN_I : out  STD_LOGIC;
          L_I : out  STD_LOGIC;
          EN_J : out  STD_LOGIC;
          L_J : out  STD_LOGIC;
          EN_QRS : out  STD_LOGIC;
          EN_ECG : out  STD_LOGIC;
          RW : out  STD_LOGIC;
          L_QRS : out  STD_LOGIC;
          L_ECG : out  STD_LOGIC;
          L_X : out  STD_LOGIC;
          EN_A : out  STD_LOGIC;
          L_A : out  STD_LOGIC;
          SM : out  STD_LOGIC;
          FIN : out  STD_LOGIC);
end CONTROL;

architecture DESIGN of CONTROL is
TYPE ESTADOS is (      A, B, C, D, E, F, G, H, I, J, K);
SIGNAL ACT, SIG : ESTADOS;
begin

MAQUINA : PROCESS(ACT, INI, Q_I, Q_J, IND)
BEGIN
    EN_I <= '0';
    L_I <= '0';
    EN_J <= '0';
    L_J <= '0';
    EN_QRS <= '0';
    EN_ECG <= '0';
    RW <= '0';
    L_QRS <= '0';
    L_ECG <= '0';
    L_X <= '0';
    EN_A <= '0';
    L_A <= '0';
    SM <= '0';
    FIN <= '0';

CASE (ACT) IS

            WHEN A =>
               L_I <= '0';
               L_J <= '0';
                  IF(INI='1')THEN
                        SIG <= B;
                  ELSE
                        SIG <= A;
                  END IF;

WHEN B =>

      IF(Q_J = '0')THEN
            IF(Q_I = '0')THEN
               SIG <= C;
            ELSE
               SM  <= '1';
               RW  <= '1';
               SIG <= J;
            END IF;
      ELSE
               SIG <= K;
      END IF;


WHEN C =>
      IF(IND = '1')THEN
            SIG <= D;
      ELSE
            SIG <= E;
      END IF;


WHEN D =>
       RW <= '1';
       SIG <= I;


WHEN E =>
       EN_QRS <= '1';
       EN_ECG <= '1';
       SIG <= F;


WHEN F =>
       L_QRS <= '1';
       L_ECG <= '1';
       SIG <= G;


WHEN G =>
       L_X <= '1';
       SIG <= H;


WHEN H =>
       EN_A <= '1';
       SIG <= I;


WHEN I =>
       EN_I <= '1';
       SIG <= B;


WHEN J =>
       EN_J <= '1';
       L_I  <= '1';
       L_A  <= '1';
       SIG <= B;
```

```
            WHEN K => FIN <= '1';
                  L_J <= '1';
                  EN_ECG <= '1';
                        IF(INI = '1')THEN
                           SIG <= K;
                        ELSE
                           SIG <= A;
                        END IF;
END CASE;
END PROCESS MAQUINA;
CAMBIOS : PROCESS(CLR, CLK)
BEGIN
   IF(CLR='1')THEN
      ACT <= A;
   ELSIF(CLK'EVENT AND CLK='1')THEN
      ACT <= SIG;
   END IF;
END PROCESS CAMBIOS;
end DESIGN;
```

Paquete.vhd

```
library IEEE;
library STD;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use STD.TEXTIO.ALL;

package PAQUETE is
            CONSTANT NBITS_ADDR_QRS : INTEGER := 5;
            CONSTANT NBITS_DATA_QRS : INTEGER := 12;
            CONSTANT NBITS_ADDR_ECG : INTEGER := 12;
            CONSTANT NBITS_DATA_ECG : INTEGER := 12;
            CONSTANT NBITS_ADDR_COR : INTEGER := 12;
            CONSTANT NBITS_DATA_COR : INTEGER := 24;

TYPE M_QRS IS ARRAY (0 TO (2**NBITS_ADDR_QRS -1)) OF STD_LOGIC_VECTOR(NBITS_DATA_QRS-1 DOWNTO 0);
TYPE M_ECG IS ARRAY (0 TO (2**NBITS_ADDR_ECG -1)) OF STD_LOGIC_VECTOR(NBITS_DATA_ECG-1 DOWNTO 0);

IMPURE FUNCTION LLENAR_ECG(ARCHIVO : IN STRING) RETURN M_ECG;
IMPURE FUNCTION LLENAR_QRS(ARCHIVO : IN STRING) RETURN M_QRS;

component MEM_QRS is
   Port ( CLK          : in  STD_LOGIC;
          EN           : in  STD_LOGIC;
          ADDR         : in  STD_LOGIC_VECTOR (NBITS_ADDR_QRS-1 downto 0);
          DATA         : out STD_LOGIC_VECTOR (NBITS_DATA_QRS-1 downto 0)
          );
end component;


component MEM_ECG is
   Port ( CLK          : in  STD_LOGIC;
          EN           : in  STD_LOGIC;
          ADDR         : in  STD_LOGIC_VECTOR (NBITS_ADDR_ECG-1 downto 0);
          DATA         : out STD_LOGIC_VECTOR (NBITS_DATA_ECG-1 downto 0)
          );
end component;


component MEM_CORR is
   Port ( CLK          : in  STD_LOGIC;
          RW           : in  STD_LOGIC;
          ADDR         : in  STD_LOGIC_VECTOR (NBITS_ADDR_COR-1 downto 0);
          DIN  : in  STD_LOGIC_VECTOR (NBITS_DATA_COR-1 downto 0);
          DOUT         : out STD_LOGIC_VECTOR (NBITS_DATA_COR-1 downto 0)
          );
end component;
```

```
component CONT_128 is
   Port ( CLK  : in  STD_LOGIC;
          CLR    : in  STD_LOGIC;
          EN     : in  STD_LOGIC;
          L      : in  STD_LOGIC;
          Q_IN  : in  STD_LOGIC_VECTOR (11 downto 0);
          Q      : out  STD_LOGIC_VECTOR (11 downto 0));
end component;


component CONT_32 is
   Port ( CLK  : in  STD_LOGIC;
          CLR    : in  STD_LOGIC;
          EN     : in  STD_LOGIC;
          L      : in  STD_LOGIC;
          Q_IN  : in  STD_LOGIC_VECTOR (4 downto 0);
          Q      : out  STD_LOGIC_VECTOR (4 downto 0));
end component;


component CONT_AUX is
   Port ( A   : in STD_LOGIC_VECTOR (4 downto 0);
          B   : in STD_LOGIC_VECTOR (11 downto 0);
          N16 : in STD_LOGIC_VECTOR (12 downto 0);
          SA  : out STD_LOGIC_VECTOR (12 downto 0));
end component;


component MOD_DSP is
   Port ( CLK : in  STD_LOGIC;
          CLR : in  STD_LOGIC;
          E_QRS : in  STD_LOGIC_VECTOR (11 downto 0);
          E_ECG : in  STD_LOGIC_VECTOR (11 downto 0);
          L_QRS : in  STD_LOGIC;
          L_ECG : in  STD_LOGIC;
          L_X : in  STD_LOGIC;
          EN_A : in  STD_LOGIC;
          L_A : in  STD_LOGIC;
          SM : in  STD_LOGIC;
          RES : out  STD_LOGIC_VECTOR (23 downto 0));
end component;


component CONTROL is
   Port ( CLK : in  STD_LOGIC;
          CLR : in  STD_LOGIC;
          INI : in  STD_LOGIC;
          Q_I : in  STD_LOGIC;
          Q_J : in  STD_LOGIC;
          IND : in  STD_LOGIC;
          EN_I : out  STD_LOGIC;
          L_I : out  STD_LOGIC;
          EN_J : out  STD_LOGIC;
          L_J : out  STD_LOGIC;
          EN_QRS : out  STD_LOGIC;
          EN_ECG : out  STD_LOGIC;
          RW : out  STD_LOGIC;
          L_QRS : out  STD_LOGIC;
          L_ECG : out  STD_LOGIC;
          L_X : out  STD_LOGIC;
          EN_A : out  STD_LOGIC;
          L_A : out  STD_LOGIC;
          SM : out  STD_LOGIC;
          FIN : out  STD_LOGIC);
end component;
end PAQUETE;
```

```
package body PAQUETE is

------------ FUNCION PARA LLENAR LA ROM ECG ---------------------------
IMPURE FUNCTION LLENAR_ECG(ARCHIVO : IN STRING) RETURN M_ECG IS
    FILE ARCH_DATOS            : TEXT IS IN ARCHIVO;
    VARIABLE LINEA_DATOS   : LINE;
    VARIABLE MEM_ROM       : M_ECG;
    VARIABLE DATO              : INTEGER;
BEGIN
FOR ii IN 0 TO 2499 LOOP
        READLINE(ARCH_DATOS,LINEA_DATOS);
        read (LINEA_DATOS, DATO);
        MEM_ROM(ii) := CONV_STD_LOGIC_VECTOR(DATO, NBITS_DATA_ECG);
    END LOOP;
    RETURN MEM_ROM;
END FUNCTION;

------------ FUNCION PARA LLENAR LA ROM QRS ---------------------------
IMPURE FUNCTION LLENAR_QRS(ARCHIVO : IN STRING) RETURN M_QRS IS
    FILE ARCH_DATOS            : TEXT IS IN ARCHIVO;
    VARIABLE LINEA_DATOS   : LINE;
    VARIABLE MEM_ROM       : M_QRS;
    VARIABLE DATO              : INTEGER;
BEGIN
    FOR jj IN 0 TO 31 LOOP
        READLINE(ARCH_DATOS,LINEA_DATOS);
        read (LINEA_DATOS, DATO);
        MEM_ROM(jj) := CONV_STD_LOGIC_VECTOR(DATO, NBITS_DATA_QRS);
    END LOOP;
    RETURN MEM_ROM;
END FUNCTION;
 end PAQUETE;
```

Principal.vhd

```
library IEEE;
Library WORK;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use WORK.PAQUETE.ALL;

entity PRINCIPAL is
   Port ( CLK :   in  STD_LOGIC;
          CLR :   in  STD_LOGIC;
          INI :   in  STD_LOGIC;
          FIN :   out  STD_LOGIC;
          ADDR_S :  out  STD_LOGIC_VECTOR (11 downto 0);
          ADDR_I  : out  STD_LOGIC_VECTOR (4 downto 0);
          CORR :  out  STD_LOGIC_VECTOR (23 downto 0));
end PRINCIPAL;

architecture DESIGN of PRINCIPAL is
SIGNAL i   : STD_LOGIC_VECTOR(4 DOWNTO 0); --CONTADOR PARA LA MEMORIA DEL PATRON QRS
SIGNAL j   : STD_LOGIC_VECTOR(11 DOWNTO 0); --CONTADOR PARA LA MEMORIA DEL ECG
SIGNAL ind : STD_LOGIC_VECTOR(12 DOWNTO 0); --INDICE AUXILIAR (IND = I + (J-16))
SIGNAL E_I   : STD_LOGIC;
SIGNAL E_J   : STD_LOGIC;
SIGNAL EN_QRS  : STD_LOGIC;
SIGNAL EN_ECG  : STD_LOGIC;
SIGNAL RW    : STD_LOGIC;
SIGNAL L_I   : STD_LOGIC;
SIGNAL L_J   : STD_LOGIC;
SIGNAL L_QRS : STD_LOGIC;
SIGNAL L_ECG : STD_LOGIC;
SIGNAL L_X : STD_LOGIC;
```

```
SIGNAL EN_A : STD_LOGIC;
SIGNAL L_A : STD_LOGIC;
SIGNAL SM : STD_LOGIC;
SIGNAL COMP_I : STD_LOGIC;
SIGNAL COMP_J : STD_LOGIC;
SIGNAL COMP_IND : STD_LOGIC;
SIGNAL FIN_I : STD_LOGIC;

SIGNAL DIN_C : STD_LOGIC_VECTOR (23 downto 0);
SIGNAL ECG              : STD_LOGIC_VECTOR (11 downto 0);
SIGNAL QRS              : STD_LOGIC_VECTOR (11 downto 0);
SIGNAL CORR_I           : STD_LOGIC_VECTOR (23 downto 0);

CONSTANT Q_I   : STD_LOGIC_VECTOR(4 DOWNTO 0) := (OTHERS=>'0');--"00000";         --reiniciar a 0 cont_i
CONSTANT Q_J   : STD_LOGIC_VECTOR(11 DOWNTO 0) := (OTHERS=>'0');--"0000000";  --reiniciar a 0 cont_j
CONSTANT N16 : STD_LOGIC_VECTOR(12 DOWNTO 0) := "1111111110000"; -- ( -16 INT )
begin

CONT_I    : CONT_32  PORT MAP(
            CLK         => CLK,
            CLR         => CLR,
            EN          => E_I,
            L           => L_I,
            Q_IN        => Q_I,
            Q           => i
            );

CONT_J    : CONT_128 PORT MAP(
            CLK         => CLK,
            CLR         => CLR,
            EN          => E_J,
            L           => L_J,
            Q_IN        => Q_J,
            Q           => j
            );

CONT_IND  : CONT_AUX PORT MAP(
            A           => i,
            B           => j,
            N16         => N16,
            SA          => ind
            );

ECG_MEM  : MEM_ECG  PORT MAP(
            CLK         => CLK,
            EN          => EN_ECG,
            ADDR        => ind(11 downto 0),
            DATA        => ECG
            );

QRS_MEM  : MEM_QRS  PORT MAP(
            CLK         => CLK,
            EN          => EN_QRS,
            ADDR        => i,
            DATA        => QRS
            );

CORR_MEM : MEM_CORR PORT MAP(
            CLK         => CLK,
            RW          => RW,
            ADDR        => j,
            DIN         => DIN_C,
            DOUT        => CORR_I
            );
```

```
CAL_CORR : MOD_DSP  PORT MAP(
            CLK         => CLK,
            CLR         => CLR,
            E_QRS => QRS,
            E_ECG => ECG,
            L_QRS => L_QRS,
            L_ECG => L_ECG,
            L_X         => L_X,
            EN_A        => EN_A,
            L_A         => L_A,
            SM          => SM,
            RES         => DIN_C
            );

            COMP_I <= '1' WHEN ( i = "11111" )ELSE '0';
            COMP_J <= '1' WHEN ( j = "100111000100" )ELSE '0';
            COMP_IND <= '1' WHEN ( (ind < "0000000000000") OR (ind > "0100110110110") )ELSE '0';

U_CONTROL : CONTROL  PORT MAP(
            CLK                 => CLK,
            CLR                 => CLR,
            INI                 => INI,
            Q_I                 => COMP_I,
            Q_J                 => COMP_J,
            IND                 => COMP_IND,
             EN_I               => E_I,
            L_I                 => L_I,
            EN_J                => E_J,
            L_J                 => L_J,
            EN_QRS              => EN_QRS,
             EN_ECG             => EN_ECG,
            RW                  => RW,
            L_QRS               => L_QRS,
            L_ECG               => L_ECG,
            L_X                 => L_X,
            EN_A                => EN_A,
            L_A                 => L_A,
            SM                  => SM,
            FIN                 => FIN_I
            );

ADDR_S <= j;
ADDR_i <= i;
CORR <= CORR_I;
FIN <= FIN_I;

end DESIGN;
```

Códigos en C del Core del Sistema Embebido

Test.c

```
#include "stdio.h"
#include "xparameters.h"
#include "xuartlite.h"
#include "core.h"

#define UART     XPAR_USB_UART_DEVICE_ID
#define UART_A   XPAR_UARTLITE_1_BASEADDR
#define CORE    XPAR_CORE_0_BASEADDR

XUartLite SERIAL;

int main(){

        int i, j, l, res, res2, muestras[2500], dato;
        u8 fin, ini, *p;
        p=(u8 *)&dato;
        XUartLite_Initialize(&SERIAL, UART);
        CORE_mReset(CORE);

        for(i=0;i<2500;i++){
                muestras[i]=0;
                }

        CORE_mWriteSlaveReg0(CORE, 0, 1);     //se asigna 1 a la variable INI
        fin = CORE_mReadSlaveReg1(CORE, 0); //se lee el valor de la variable FIN
        ini = CORE_mReadSlaveReg0(CORE, 0); //se lee el valor de la variable INI
        fin = CORE_mReadSlaveReg1(CORE, 0); //se lee el valor de la variable FIN

        for(i=0;i<10000;i++){                 //ciclo for para introducir un pequeño retardo
        }
        fin = CORE_mReadSlaveReg1(CORE, 0); //se lee el valor de la variable FIN


        /*Condicionales para verificar el valor máximo del complejo QRS.
         * Si el valor es mayor a 35500 y el n+1 valor es menor a 35500, se ha encontrado
         * el valor máximo.
         * Posteriormente se leen los 16 valores anteriores y los 16 valores siguientes que
         * conforman elcomplejo QRS.
         */
        for(i=0;i<2500;i++){
                CORE_mWriteSlaveReg2(CORE, 0, i);
                res = CORE_mReadSlaveReg3(CORE, 0);
                if(res>355000){
                        CORE_mWriteSlaveReg2(CORE, 0, i+1);
                        res2 = CORE_mReadSlaveReg3(CORE, 0);
                        if(res2<355000){
                                for(j=i;j<(i+32);j++){
                                        CORE_mWriteSlaveReg4(CORE, 0, (j-16));
                                        res = CORE_mReadSlaveReg5(CORE, 0);
                                        muestras[j-16]=res*1000;
                                }
                        }
                                else
                                        continue;
                }
        }

        for(i=0;i<2500;i++){
                dato = muestras[i];
                XUartLite_Send(&SERIAL,p,4);  //se envían los datos a través de la interfaz UART
                for(l=0;l<10000;l++){
                }
        }

        return 0;
}
```

Códigos en Java de la comunicación cliente-servidor y la graficación

Servidor

RdWr.java

```
package Transferir;

public class RdWr {

   int num[] = new int [2500];
   int ind1 = 0, ind2 = 0;

   public int getInt(){
      ind1++;
      return num[ind1];
   }

   public void setInt(int n){
      num[ind2] = n;
      ind2++;
   }
}
```

Comunication.java

```
package Transferir;

import gnu.io.CommPortIdentifier;
import gnu.io.PortInUseException;
import gnu.io.SerialPort;
import gnu.io.SerialPortEvent;
import gnu.io.SerialPortEventListener;
import gnu.io.UnsupportedCommOperationException;
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.*;

public class Comunication implements Runnable, SerialPortEventListener {

   static CommPortIdentifier idPuerto;
   static Enumeration listaPuertos;
   InputStream entrada;
   SerialPort puertoSerie;
   Thread tLectura;
   int valor;
   RdWr leer = new RdWr();

   static final int PUERTO = 5000;
   //Se implementa un thread que es el que se encarga de
   //que la aplicación se quede esperando en el puerto que se haya
   //abierto a que se reciban datos.
   //Primero abre el puerto y luego le fija los parámetros
   public Comunication() {
      // Si el puerto no está en uso, se intenta abrir
      try {
         puertoSerie = (SerialPort) idPuerto.open("AplLectura", 2000);
      } catch (PortInUseException e) {
         System.out.println(e);
      }
```

```
    // Se obtiene un canal de entrada
    try {
       entrada = puertoSerie.getInputStream();
    } catch (IOException e) {
       System.out.println(e);
    }

    // Añadimos un receptor de eventos para estar informados de lo
    // que suceda en el puerto
    try {
       puertoSerie.addEventListener(this);
    } catch (TooManyListenersException e) {
       System.out.println(e);
    }

    // Hacemos que se nos notifique cuando haya datos disponibles
    // para lectura en el buffer de la puerta
    puertoSerie.notifyOnDataAvailable(true);

    // Se fijan los parámetros de comunicación del puerto
    try {
       puertoSerie.setSerialPortParams(9600,
             SerialPort.DATABITS_8,
             SerialPort.STOPBITS_1,
             SerialPort.PARITY_NONE);
    } catch (UnsupportedCommOperationException e) {
       System.out.println(e);
    }

    // Se crea y lanza el thread que se va a encargar de quedarse
    // esperando en la puerta a que haya datos disponibles
    tLectura = new Thread(this);
    tLectura.start();
}

@Override
public void run() {
    try {
       // En los threads, hay que procurar siempre que haya algún
       // método de escape, para que no se queden continuamente
       // bloqueados, en este caso, la comprobación de si hay datos
       // o no disponibles en el buffer de la puerta, se hace
       // intermitentemente
       System.out.println("Entra hilo");
       Thread.sleep(2000);
    } catch (InterruptedException e) {
       System.out.println(e);
    }

}

@Override
public void serialEvent(SerialPortEvent _ev) {
    int dato = 0,temp, i = 3, j = 0;
    switch (_ev.getEventType()) {
       // La mayoría de los eventos no se trata, éstos son los
       // que se producen por cambios en las líneas de control del
       // puerto que se está monitorizando
       case SerialPortEvent.BI:
       case SerialPortEvent.OE:
       case SerialPortEvent.FE:
       case SerialPortEvent.PE:
       case SerialPortEvent.CD:
       case SerialPortEvent.CTS:
       case SerialPortEvent.DSR:
       case SerialPortEvent.RI:
       case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
          break;
       // Cuando haya datos disponibles se leen.
```

```
        // Como sólo se pueden transmitir datos de 8 bits a la vez
        // se usa un ciclo para reconstruir el dato de 24 bits
        // se guarda el dato en el arreglo y se imprime en la consola
        case SerialPortEvent.DATA_AVAILABLE:
          try {
            while (i >= 0) {
              int nByte = entrada.read();
              temp = nByte << (i * 8);
              dato = dato | temp;
              i--;
            }
            i = 3;
            leer.setInt(dato);
            System.out.println(dato);
          } catch (IOException e) {
            System.out.println(e);
          }
          break;
      }
    }
    // Se crea un socket de tipo servidor y se deja a la espera de conexiones
    // Si un cliente se conecta, el servidor comienza a enviar los datos
    // Al terminar, se cierra la conexión con el cliente
    public void initServer() {
      try {
        ServerSocket skServidor = new ServerSocket(PUERTO);
        System.out.println("Escucho en el puerto " + PUERTO);
          Socket skCliente = skServidor.accept(); // Crea objeto
          System.out.println("Sirvo al cliente ");
          OutputStream aux = skCliente.getOutputStream();
          DataOutputStream flujo = new DataOutputStream(aux);
         for(int f=0;f<2500;f++){
          flujo.writeInt(leer.getInt());
         }
         skServidor.close();
      } catch (Exception e) {
        System.out.println(e.getMessage());
      }
    }

    public static void main(String[] args) {
      // Lista de los puertos disponibles en la máquina. Se carga en el
      // mimo momento en que se inicia la JVM de Java
      listaPuertos = CommPortIdentifier.getPortIdentifiers();

      while (listaPuertos.hasMoreElements()) {
        idPuerto = (CommPortIdentifier) listaPuertos.nextElement();
        if (idPuerto.getPortType() == CommPortIdentifier.PORT_SERIAL) {
          if (idPuerto.getName().equals("COM5")) {         // WINDOWS
            // Lector del puerto, se quedará esperando a que llegue algo
            // al puerto
            int i;
            Comunication lector = new Comunication();
            lector.initServer();
          }
        }
      }
    }
}
```

Cliente

Ventana.java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.DataInputStream;
import java.io.File;
import java.io.InputStream;
import java.net.Socket;
import javax.swing.JButton;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.ChartUtilities;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.CategoryPlot;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;

public class Ventana extends ApplicationFrame {
   static final String HOST = "169.254.216.234";
   static final int PUERTO = 5000;

   public Ventana(final String title) {
     super(title);
      final DefaultCategoryDataset dataset = createDataset();
      final JFreeChart chart = createChart(dataset);
      final ChartPanel chartPanel = new ChartPanel(chart);
      chartPanel.setPreferredSize(new java.awt.Dimension(1000, 400));
      chartPanel.setMouseZoomable(true, true);
      this.add(chartPanel, BorderLayout.CENTER);
      JPanel customPanel = new JPanel();
      JButton saveButton = new JButton("Guardar como Imagen JPG");

      // Se agrega un botón para guardar la gráfica como una imagen JPG
      saveButton.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            try {
               ChartUtilities.saveChartAsJPEG(new File("ECG.jpg"),
                     chart, 1800, 750);
               JOptionPane.showMessageDialog(null, "Imagen guardada.",
                     "Confirmación", JOptionPane.INFORMATION_MESSAGE);
            } catch (Exception ex) {
               JOptionPane.showMessageDialog(null, "Error al guardar imagen",
                     "Error", JOptionPane.ERROR_MESSAGE);
            }
         }
      });
      customPanel.add(saveButton);
      this.add(customPanel, BorderLayout.PAGE_END);
   }

   // Se configuran las opciones de la gráfica
   private JFreeChart createChart(
         final DefaultCategoryDataset line_chart_dataset) {

      final JFreeChart chart = ChartFactory.createLineChart("ECG",
            "tiempo", "mV", line_chart_dataset, PlotOrientation.VERTICAL,
            false, false, false);
      chart.setBackgroundPaint(Color.LIGHT_GRAY);//light gray
```

```
        final CategoryPlot plot = chart.getCategoryPlot();
        plot.setBackgroundPaint(Color.BLACK);
        plot.setRangeGridlinePaint(Color.GREEN);
        return chart;
    }

    // Se crea un socket de tipo cliente
    // Se manda una solicitud de conexión al servidor para comenzar a leer
    // los datos y rellenar el objeto con los datos obtenidos
    // Una vez leídos todos los datos se crea y se muestra la gráfica
    private DefaultCategoryDataset createDataset() {
        DefaultCategoryDataset line_chart_dataset = new DefaultCategoryDataset();
        try {
            Socket skCliente = new Socket(HOST, PUERTO);
            InputStream aux = skCliente.getInputStream();
            DataInputStream flujo = new DataInputStream(aux);
            for (int f = 0; f < 2500; f++) {
                line_chart_dataset.addValue(flujo.readInt(), "valor", f);
                System.out.println(flujo.readInt());
            }
            skCliente.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        return line_chart_dataset;
    }

    public static void main(final String[] args) {
        final Ventana demo = new Ventana("Señal ECG Procesada");
        demo.pack();
        RefineryUtilities.centerFrameOnScreen(demo);
        demo.setVisible(true);
    }

}
```