Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Implementación de un algoritmo de reconocimiento de patrones en señales biomédicas para su evaluación en un microprocesador basado en la Arquitectura de Set de Instrucciones RISC V

Informe de Proyecto de Graduación para optar por el título de

Ingeniero en Electrónica con el grado académico de Licenciatura

Gabriel José Madrigal Boza

Versión de 4 de diciembre de 2017

INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERÍA ELECTRÓNICA

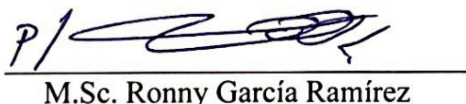
PROYECTO DE GRADUACIÓN

ACTA DE APROBACIÓN

Defensa de Proyecto de Graduación
Requisito para optar por el título de Ingeniero en Electrónica
Grado Académico de Licenciatura
Instituto Tecnológico de Costa Rica

El Tribunal Evaluador aprueba la defensa del proyecto de graduación denominado Implementación de un algoritmo de reconocimiento de patrones en señales biomédicas para su evaluación en un microprocesador basado en la Arquitectura de Set de Instrucciones RISC V, realizado por el señor Gabriel José Madrigal Boza y, hace constar que cumple con las normas establecidas por la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal Evaluador



M.Sc. Ronny García Ramírez

Profesor lector

Ing. Carlos Mauricio Segura Quirós

Profesor lector

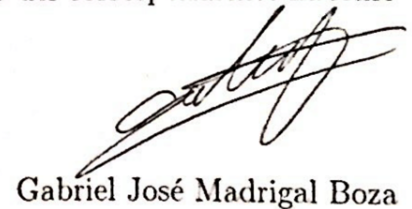
M.Sc. Roberto Molina Robles

Profesor asesor

Cartago, 30 de noviembre, 2017

Declaro que el presente Proyecto de Graduación ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.



Gabriel José Madrigal Boza

Cartago, 28 de noviembre de 2017

Céd: 1-1580-0261

Resumen

En este documento se expone el desarrollo de un sistema de reconocimiento de eventos o patrones en señales de electrocardiogramas (ECG). La detección de eventos se realiza mediante la ejecución de un algoritmo de correlación, utilizando señales biomédicas reales representadas en un formato *bitstream*. El desarrollo del proyecto se divide en dos secciones, una de las cuáles está dedicada al desarrollo del algoritmo y su implementación en C++. Por otro lado, se tiene el desarrollo de la plataforma de hardware en la cual se ejecuta el algoritmo. En el caso del presente proyecto, se utiliza la plataforma *RocketChip*, de uso libre, que se basa en la arquitectura de set de instrucciones RISC V.

Palabras clave: Correlación cruzada, Dispositivos médicos implantables, RISCV, RocketChip

Abstract

In this document, the development of an event/pattern recognition system in ECG signals is presented. The event detection is performed via a cross-correlation algorithm, using real biomedical signals represented in a bitstream format. The development of the project is divided in two main sections, one of which considers the design of the algorithm and its implementation in C++. The other main section discusses the development of a hardware platform in which the algorithm is executed. In the case of the presented project, the hardware platform used is based on RocketChip, an open-source initiative based on the instruction set architecture RISC V.

Keywords: Cross-correlation, Implantable medical devices, RISCV, RocketChip

a mi querida familia

Agradecimientos

A mis padres, Víctor y Virginia, por ser el mejor ejemplo siempre. Han sido el apoyo fundamental durante toda mi vida, gracias al que he logrado cumplir esta meta. A mis hermanas, Aurora y Ana Victoria, por ayudarme en este camino y ser siempre la mejor compañía.

A todos mis amigos y amigas que me han acompañado en este camino. A todos los que conocí durante mi paso en el TEC, los compañeros de Electrónica y del DCILab. A todos los allegados al *Aparta 40*, les debo las mejores experiencias de estos 5 años.

Por último, a los profesores Renato Rímolo Donadio, Alfonso Chacón Rodríguez y Ronny García Ramírez, por darme la oportunidad de trabajar con ustedes y aprender tanto; y al profesor Roberto Molina Robles por su ayuda con este proyecto.

Gabriel José Madrigal Boza

Cartago, 4 de diciembre de 2017

Índice general

Índice de figuras **iii**

Índice de tablas **vi**

Índice de códigos fuente **vii**

1 Introducción **1**
- 1.1 Problema . . . 3
- 1.2 Objetivos . . . 3
  - 1.2.1 Objetivo General . . . 3
  - 1.2.2 Objetivos Específicos . . . 4

2 Marco Teórico **5**
- 2.1 RocketChip SoC . . . 5
  - 2.1.1 Protocolo TileLink . . . 8
  - 2.1.2 Microarquitectura básica del núcleo de procesamiento (Rocket) . . . 9
  - 2.1.3 Memorias Caché de 1er Nivel . . . 10
  - 2.1.4 Soporte de Depuración Externa para RISC-V . . . 11
  - 2.1.5 Diagramas de referencia . . . 12
- 2.2 Correlación Cruzada de Señales . . . 14
- 2.3 Representación de señales en formato *bitstream* . . . 15
- 2.4 Señales de Electrocardiogramas: ECG . . . 16
  - 2.4.1 Mecanismos de conducción eléctrica en el corazón . . . 16
  - 2.4.2 Elementos importantes presentes en un electrocardiograma . . . 18

3 Implementación de un SoC basado en RocketChip **20**
- 3.1 Sistema en Chip Generado . . . 22
- 3.2 Simulaciones y ejecución de programas de prueba . . . 24

4 Correlación cruzada con señales *bitstream* **25**
- 4.1 Obtención de las señales ECG . . . 25
- 4.2 Diseño del modulador ΣΔ . . . 27
- 4.3 Representación *bitstream* de las señales . . . 28
- 4.4 Algoritmo de Correlación Cruzada implementado en C++ . . . 28
- 4.5 Diseño de un filtro paso bajo digital de 3er orden . . . 31

4.6 Filtro de Media Móvil 32
4.7 Comparador y Detector implementados 34

5 Resultados y Análisis **35**
5.1 Sistema en Chip generado 35
5.2 Resultados de la simulación con programas de prueba 36
5.3 Obtención del patrón de comparación para señales ECG 42
5.4 Representación *bitstream* de las señales utilizadas en las pruebas 43
5.5 Respuesta en frecuencia del filtro paso bajo diseñado 45
5.6 Correlación cruzada entre señales de prueba y distintos patrones 46
5.7 Detección de eventos en señales ECG 49

6 Conclusiones y Recomendaciones **52**
6.1 Conclusiones 52
6.2 Recomendaciones 53

Bibliografía **54**

A Resultados Adicionales **57**
A.1 Resultado de la correlación utilizando diferentes señales de prueba 57
A.2 Resultados de la detección de eventos ante señales senoidales de diferentes frecuencias 61

Índice alfabético **63**

Índice de figuras

1.1 Representación de diferentes sistemas médicos implantables. Imagen tomada de [1]. 1

2.1 Ejemplo de un SoC diseñado utilizando RocketChip Generator. Se observan 2 recuadros, cada uno de los cuales contiene una unidad de procesamiento. En un caso es un núcleo Rocket, en el otro una unidad de ejecución fuera de orden (BOOM). Cada uno de los recuadros cuenta con sus memorias caché de 1er nivel, tanto para datos como instrucciones, y ambos núcleos incluyen FPU y coprocesadores. Figura tomada de [5]. 6

2.2 Microarquitectura del núcleo Rocket, con las etapas (simplificadas) del pipeline. En la etapa de decodificación, el bloque denominado RF corresponde al banco de registros (*Register File*). El componente BTB corresponde a una unidad de almacenamiento de destino de saltos (*Branch Target Buffer*), que asiste en la predicción de saltos. Figura tomada de [15]. 9

2.3 Arquitectura del sistema de depuración externa. Se observa como la interfaz DMI sirve de puente entre el módulo de depuración DM y el DTM. Figura tomada de [21]. 11

2.4 Microarquitectura del núcleo de procesamiento Rocket. Se muestran los bloques funcionales de los que está compuesto el mismo. Se pueden observar las señales y colas que comunican el núcleo con la FPU, TileLink, HTIF (Host Target Interface) y PTW. Figura tomada de [15]. 12

2.5 Generador de PC y etapa Fetch. Figura tomada de [3]. 13

2.6 Etapas del pipeline. En la implementación básica no se incluye la unidad de punto flotante. Figura tomada de [3]. 13

2.7 Memoria caché de datos. Figura tomada de [3]. 14

2.8 Ilustración del corazón, en la que se señalan los elementos principales involucrados en los procesos de conducción eléctrica. Las cavidades derechas (aurícula y ventrículo) se ubican a la *izquierda* de la ilustración, según el punto de vista del lector. Figura tomada de [14]. 17

2.9 Ejemplo típico de un electrocardiograma, se observan las principales ondas e intervalos que se presentan en un ECG. Se pueden observar los intervalos RR, PR y QT, además del complejo QRS y las ondas T y P. Figura tomada de [14]. 18

4.1 Ciclo cardíaco completo, con sus diferentes fragmentos informativos, obtenido de la derivación I. Figura tomada de [18]. 26

4.2 Modulación $\Sigma\Delta$ utilizando el paquete de herramientas *python sigma-delta*. Los bloques representan cada una de las funciones utilizadas. Se muestran las señales de entrada y salida, además de los valores usados para cada uno de los parámetros de entrada de las funciones. 27
4.3 Correlación cruzada, ejecutada bit por bit en una de las posiciones del arreglo de tipo **uint32_t**. 29
4.4 Filtro IIR de tercer orden implementado. El filtro implementa la ecuación de diferencias 4.2. 32
4.5 Diagrama conceptual del cálculo de la media móvil. En este ejemplo se utilizan 4 valores de salida del filtro paso bajo, pero este valor se puede parametrizar como N. 33
4.6 Diagrama conceptual del comparador. Las entradas o valores a comparar son la salida del Filtro Paso Bajo y la salida del Filtro de Media Móvil. Esta última está amplificada por una constante K. 34

5.1 Información relevante de la etapa *Write Back* durante la ejecución de un programa de prueba. Se observa la ejecución de una de las sumas del programa, para probar la instrucción *add*. 39
5.2 Comportamiento de algunas señales de la etapa *Write back* del procesador. Se observan el valor del contador de programa, el código hexadecimal de la instrucción ejecutada en ese ciclo y los operandos de la misma. 39
5.3 Comportamiento de algunas señales de la etapa *Exe* del procesador. Se observan el valor del contador de programa, el código hexadecimal de la instrucción ejecutada en ese ciclo y las entradas y salidas de la Unidad Aritmético-Lógica (ALU). 40
5.4 Obtención del promedio de 3 ciclos cardíacos para formar el patrón ECG utilizado en el algoritmo de correlación. Se muestran las 4 señales relevantes, de las cuales 3 corresponden a las señales extraídas de la base de datos. . . . 43
5.5 Representación *bitstream* de la señal ECG utilizada como patrón. Para mejorar la visualización de las señales, se muestra sólo una porción de las mismas. 44
5.6 Espectro de la señal de un electrocardiograma. En (a) se observa el espectro de la señal sin realizarle ningún preprocesamiento. En (b) se muestra el espectro de la señal modulada ($\Sigma\Delta$) en rojo, junto al espectro de la misma señal luego de aplicar el filtro paso bajo diseñado. Además, se puede observar, en rojo, como el efecto de la modulación hace que el ruido de cuantización se mueva a frecuencias altas, y como este ruido se elimina al aplicar el filtro. 44
5.7 Respuesta en frecuencia del filtro paso bajo diseñado. Se observa la frecuencia de corte para la que el filtro fue diseñado, y el valor de la atenuación en dicha frecuencia. 45
5.8 Respuesta en fase del filtro diseñado. Se puede observar la frecuencia de corte para la que se diseñó el filtro. 46
5.9 Patrón y señal de prueba utilizados para la correlación de una señal senoidal. 47
5.10 Correlación de una señal senoidal. 47
5.11 Error en la correlación de una señal senoidal. 48

5.12 Resultados de la correlación de una señal ECG contra su respectivo patrón, caso 1. 48
5.13 Error en la correlación de una señal ECG contra su respectivo patrón, caso 1. 49
5.14 Salida de los filtros paso bajo y de media móvil, junto a la salida del detector, al utilizar señales ECG como entrada al algoritmo. 50
5.15 Salida de los filtros paso bajo y de media móvil, junto a la salida del detector, al utilizar una señal de ruido para probar la eficacia del sistema. 50

A.1 Correlación de una señal cuadrada. 57
A.2 Error en la correlación de una señal cuadrada. 58
A.3 Correlación de una señal diente de sierra. 58
A.4 Error en la correlación de una señal diente de sierra. 58
A.5 Patrón y señal de prueba utilizados para la correlación de una señal cuadrada. 59
A.6 Patrón y señal de prueba utilizados para la correlación de una señal diente de sierra. 59
A.7 Resultados de la correlación de una señal ECG contra su respectivo patrón, caso 2. 60
A.8 Error en la correlación de una señal ECG contra su respectivo patrón, caso 2. 60
A.9 Resultados de la correlación de una señal ECG contra su respectivo patrón, caso 3. 60
A.10 Error en la correlación de una señal ECG contra su respectivo patrón, caso 3. 61
A.11 Salida de los filtros paso bajo y de media móvil, junto a la salida del detector, al utilizar una señal senoidal con frecuencia $f = 15$ Hz. 62
A.12 Salida de los filtros paso bajo y de media móvil, junto a la salida del detector, al utilizar una señal senoidal con frecuencia $f = 200$ Hz. 62

Índice de tablas

2.1 Tabla de verdad de la función lógica XNOR. 15

3.1 Comparación entre dos categorías de arquitecturas de computadores. 21

4.1 Representación binaria de los operandos utilizados en la expresión A del algoritmo *popcount* SWAR, donde j = ((i >> 1) & 0x55555555). 30
4.2 Transformada utilizada en el algoritmo *popcount* SWAR para cada par de bits. 30

5.1 Mapeo de memoria resultante para el Sistema en Chip diseñado. 36
5.2 Resultados de la ejecución de programas de prueba en el sistema diseñado. Los resultados que se muestran son para el conjunto de instrucciones I de RISC V. Todas las pruebas terminan con resultados positivos. 41
5.3 Resultados de la ejecución de programas de prueba en el sistema diseñado. Los resultados que se muestran son para la extensión M de RISC V. Para estas pruebas los resultados obtenidos son positivos. 41
5.4 Resultados de la ejecución de programas de prueba en el sistema diseñado. Los resultados que se muestran son para las extensiones A, C y F de RISC V. Dado que se eliminó el soporte para estas extensiones en el diseño del procesador, para estas pruebas los resultados obtenidos son negativos. 42

Índice de códigos fuente

2.1 Ejemplo de una configuración básica de un microprocesador Rocket. Se observa como se cambian los parámetros del recuadro que contiene el procesador para modificar las memorias caché y la unidad de multiplicación y división del mismo. Con esta clase se pueden instanciar N procesadores con las características especificadas dentro del mismo recuadro. 7
2.2 Configuración para modificar la arquitectura del procesador (para que sea de 32 bits). Al aplicar esta configuración también deben modificarse los parámetros mostrados de las unidades de Punto Flotante y Multiplicación y División. 7
2.3 Configuración utilizada para eliminar la Unidad de Punto Flotante. Al retirar esta unidad se elimina también el soporte para la extensión de instrucciones F de RISC V. 8
2.4 Configuración utilizada para instanciar un módulo de depuración JTAG al sistema. 8
2.5 Configuración para cambiar el archivo de inicialización almacenado en la memoria *boot ROM*. 8
3.1 Configuración del sistema en chip diseñado. Se observan las clases utilizadas para disminuir el conjunto de instrucciones que se implementa. También se observa como se modifican los parámetros de las memorias caché de 1er nivel para disminuir su tamaño. 22
3.2 Configuración base del sistema en chip. Esta configuración es el punto de partida para el diseño del sistema deseado. 23
4.1 Función escrita en C++ para obtener el número de bits en alto de una variable de tipo entero sin signo de 32 bits. Implementa el algoritmo SWAR más conocido como *popcount*. 29
4.2 Función escrita en C++ para obtener la media móvil. El parámetro de entrada es una arreglo de 18 posiciones (para este ejemplo) de tipo flotante. En este se almacenan los últimos 18 valores de salida del Filtro Paso Bajo. 33
4.3 Sección de la rutina principal (escrita en C++) donde se actualiza el arreglo que contiene los N (18) últimos valores de salida del Filtro Paso Bajo. Con estos se calcula el valor actual de la Media Móvil. 33
5.1 Porción del programa de prueba *rv32ui-p-add*. 37
5.2 Definición de los macros utilizados en el programa de prueba *rv32ui-p-add*. . 37
5.3 Información del programa posterior a su compilación. Se muestran tanto las instrucciones resultantes como las direcciones de memoria donde se almacenan. 38

Capítulo 1

Introducción

Desde la aparición del primer marcapasos completamente implantable, los implantes biomédicos han ganado terreno en la industria de la medicina, tanto en lo que respecta a la aceptación del público, como en los avances en miniaturización y diversificación de aplicaciones. Es posible observar como diferentes ramas de la medicina, como la cardiología y la neurología, se han visto beneficiadas por dichos dispositivos. Usualmente, las tareas realizadas por dichos implantes corresponden a la medición de variables físicas y químicas dentro del cuerpo humano, junto, en algunos casos, a la estimulación de órganos con fines terapéuticos. Estas variables medidas se relacionan directamente con los procesos realizados por los diferentes sistemas que componen el organismo, y pueden revelar información importante sobre el mismo. Por ejemplo, las señales medidas del corazón contienen información que puede indicar la presencia de arritmias u otros defectos [9] [10]. El nivel de glucosa en la sangre es un indicador importante para personas que padecen diabetes [12]. Algunas afecciones gastroesofágicas, como el reflujo, pueden ser prevenidas si se conocen los niveles de pH en sitios clave del aparato digestivo [7].

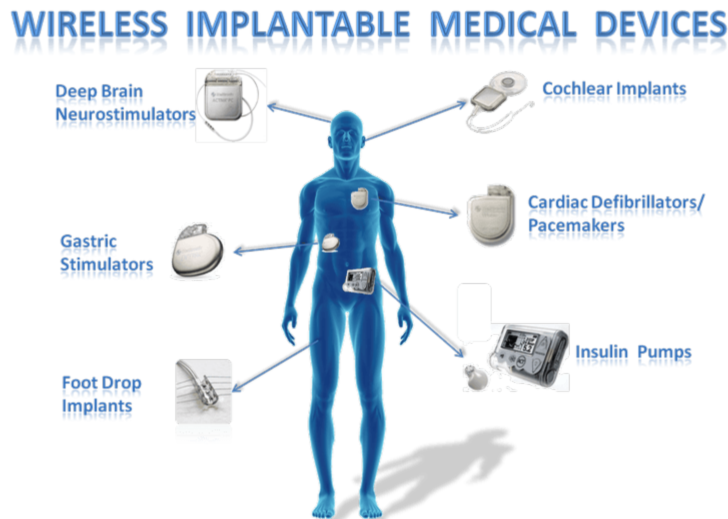


Figura 1.1: Representación de diferentes sistemas médicos implantables. Imagen tomada de [1].

Para ilustrar de una manera más adecuada la versatilidad y diversidad que ofrecen estos sistemas, es útil referirse a la figura 1.1. En esta figura se observan diferentes aplicaciones en las que los sistemas implantables encuentran utilidad.

El impacto de estos sistemas implantables no se reduce a labores terapéuticas o de restablecimiento, como es el caso de los implantes cocleares o los marcapasos. El área de la investigación médica también ha obtenido beneficios de los avances en esta tecnología. En particular, se ha visto un progreso importante en el área de la neurociencia, pues la reducción del tamaño de los implantes ha permitido la implementación de plataformas para el estudio de fenómenos del sistema nervioso. En este ámbito, los implantes son diseñados de manera que puedan obtener información de grandes grupos de neuronas, mientras los sujetos estudiados realizan tareas específicas. Esta información se presenta en la forma de señales eléctricas. De esta manera, es posible estudiar el funcionamiento de las redes de neuronas, su respuesta a estímulos y la relación entre las diferentes partes del sistema nervioso [4] [16].

Uno de los principales motores de los avances en este ámbito (los implantes biomédicos) ha sido la investigación en microelectrónica. El refinamiento y el aumento de la complejidad de los implantes se posibilita gracias a las mejoras en los procesos de fabricación de circuitos integrados. Con cada nueva tecnología, se reduce el tamaño de los transistores, lo que permite integrar circuitos con mayores capacidades en áreas menores. Esta tendencia se encuentra en concordancia con las necesidades de la industria médica, pues para que los implantes sean prácticos, su tamaño debe ser reducido en cuanto sea posible.

El flujo de diseño de los circuitos integrados que componen estos sistemas debe contemplar criterios de diseño particulares. Estos criterios, y restricciones, se derivan de la necesidad de disminuir el consumo de potencia, sin dejar de cumplir las necesidades de la aplicación para la que el sistema se ha planteado. En general, dichos circuitos integrados contienen módulos de procesamiento (en forma de microprocesadores, microcontroladores o máquinas de estados finitos); convertidores analógico/digitales (ADCs) y digitales/analógicos (DACs); módulos de almacenamiento de memoria (memorias RAM, bancos de registros); circuitos analógicos de acondicionamiento de señales (amplificadores, filtros, etc); y sensores para la medición de las variables pertinentes.

Además, dos de los componentes a los que se presta más atención son los concernientes a la alimentación del implante y la comunicación del mismo con sistemas exteriores. En el primer caso, algunas de las técnicas de alimentación se basan en baterías. Entre las variedades de las baterías están las no recargables, diseñadas con miras a funcionar por varios años, y las recargables, que son diseñadas para funcionar por períodos menores de tiempo, pero pueden ser recargadas por medio de enlaces inalámbricos. Otra tendencia observada son los implantes que se alimentan únicamente por medio de enlaces inalámbricos.

En cuanto al aspecto de la comunicación, su importancia radica en que, en general, las mediciones realizadas por el implante deben ser almacenadas en dispositivos externos, para que un especialista pueda analizarlos y valorar el estado del paciente. Además, en los casos en los que se implementa un microprocesador en el implante, es posible hacer modificaciones, ajustes y actualizaciones al programa ejecutado por el mismo. La mayor tendencia en este aspecto es la implementación de la comunicación mediante enlaces inalámbricos.

El proyecto descrito en el presente informe forma parte de una iniciativa de investigación en desarrollo, liderada por profesores e investigadores de la Escuela de Ingeniería Electrónica.

1.1 Problema

Aún cuando la investigación en implantes biomédicos es extensa, los esfuerzos y avances observados se realizan de manera aislada. La necesidad de incorporar módulos de procesamiento a los mismos presenta un reto de diseño a los investigadores, el cual puede ser resuelto por medio de diferentes acercamientos. Por un lado, es posible recurrir a alternativas comerciales, como los procesadores ARM. Esto posibilita la reducción de los tiempos de diseño e implementación. Sin embargo, no es posible optimizar dichos módulos, por lo que no se puede reducir su área dentro del circuito integrado, ni su consumo de potencia. Otra alternativa es el diseño integral del procesador o controlador. Esto permite una alta optimización de los recursos, pero se aumenta el tiempo dedicado al diseño del implante.

Por otro lado, otra característica importante a tomar en cuenta de dichos dispositivos es la necesidad de que sean seguros y tolerantes a fallas. La implementación de estas características implicaría una mayor complejidad a nivel de hardware, lo que conlleva un aumento en el consumo de potencia. Sin embargo, es necesario tomar en cuenta estos aspectos, si los implantes diseñados quieren llevarse efectivamente de los laboratorios de investigación a aplicaciones prácticas en hospitales y clínicas.

1.2 Objetivos

1.2.1 Objetivo General

Desarrollar una unidad de procesamiento y control escalable, descrita a nivel RTL para aplicaciones de dispositivos médicos implantables, que pueda ser adaptada en términos de funcionalidad y recursos, y que posea la capacidad de procesamiento necesaria para aplicaciones médicas del sistema cardiovascular.

1.2.2 Objetivos Específicos

- Utilizar la herramienta RocketChip Generator para generar un microprocesador que cumpla con las especificaciones del Set de Instrucciones RISCV en su versión básica (RISC V-IM de 32 bits).
- Implementar un algoritmo de correlación cruzada como método de reconocimiento de patrones en señales en formato *bitstream*.
- Evaluar el rendimiento y la funcionalidad del algoritmo implementado, utilizando el microprocesador generado.

El resto del informe se presenta como sigue: en el Capítulo 2 se presentan los fundamentos teóricos y conceptos utilizados a lo largo del proyecto, prestando especial atención a los detalles que involucran la microarquitectura implementada en RocketChip y los desarrollos matemáticos en los que se basan las técnicas de representación de datos en formato *bitstream* y la correlación cruzada. Los Capítulos 3 y 4 presentan el desarrollo conceptual y el diseño de la solución propuesta, mientras que el Capítulo 5 contiene los principales resultados obtenidos. Finalmente, en el capítulo 6 se extraen las conclusiones pertinentes en base a los resultados obtenidos, y se presentan algunas recomendaciones.

Capítulo 2

Marco Teórico

2.1 RocketChip SoC

RocketChip es el producto de una iniciativa que busca el diseño e implementación de un Sistema en Chip (SoC, *System-on-Chip*) de uso libre. Dicha iniciativa es liderada por investigadores de la Universidad de California-Berkeley. RocketChip cuenta con todos los elementos necesarios de un SoC funcional: procesador(es), memoria(s), buses de datos y control, previstas para periféricos en un modelo de mapeo de memoria, protocolos de coherencia de memorias caché, etc. El código fuente del sistema está escrito en Chisel, un lenguaje de construcción de hardware embebido en Scala. Gracias a la naturaleza de este lenguaje, el código es altamente parametrizable, lo que permite lograr diseños variados con la modificación de pocas líneas de código, dejando el resto intacto. Para generar los diferentes diseños de Rocket, se utiliza una herramienta denominada *RocketChip Generator* [5].

RocketChip está basado en la Arquitectura de Set de Instrucciones RISC V, e incluye tanto un procesador de ejecución de instrucciones en orden (Rocket), como una máquina de ejecución fuera de orden (BOOM) [5]. El núcleo de procesamiento Rocket es parametrizable. Esto quiere decir que los diferentes dispositivos que lo componen pueden ser añadidos o eliminados deliberadamente, para atender las necesidades de una aplicación específica. Como se mencionó, RocketChip cuenta con los componentes necesarios para la comunicación con dispositivos periféricos. Esta comunicación se lleva a cabo por medio de módulos que cumplan con el protocolo AXI. Además, cuenta con los componentes necesarios para realizar tareas de depuración. Dentro de la arquitectura del SoC, se encuentra la memoria de arranque (*BootROM*), que contiene el código necesario para la configuración inicial de los registros de Control y Estado (*CSR*) del procesador. En cualquier implementación de RocketChip, es necesario incluir las memorias caché de Datos e Instrucciones (*D$* e *I$*, respectivamente) de primer nivel, que cuentan con comunicación directa con el procesador. Además de estas, opcionalmente pueden incluirse memorias de segundo nivel (*L2*).

Dentro de RocketChip los diferentes componentes se organizan en distintos bloques según su funcionalidad. Cada núcleo, junto a las memorias caché de datos e instrucciones, la unidad de punto flotante y el coprocesador (si se incluyen) se agrupan dentro de un recuadro (*Tile*). Cada recuadro se conecta con el resto del sistema utilizando memoria compartida y mapeo de memoria. Para garantizar la coherencia de las memorias caché, se utilizan unidades que cumplan con el protocolo *TileLink*, el cual se describe más adelante. Con este es posible diseñar una interfaz con buses de sistema que cumplan con el protocolo AXI, por ejemplo. En la figura 2.1 se observa un ejemplo de un sistema diseñado con esta herramienta.

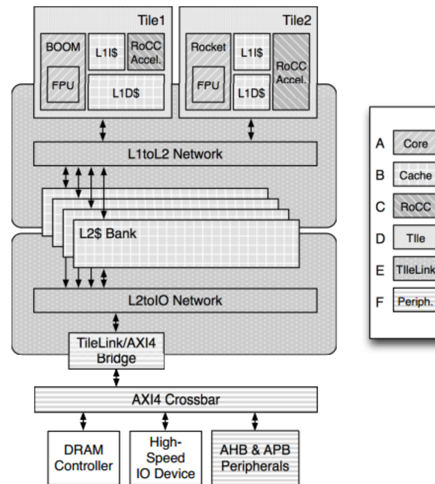


Figura 2.1: Ejemplo de un SoC diseñado utilizando RocketChip Generator. Se observan 2 recuadros, cada uno de los cuales contiene una unidad de procesamiento. En un caso es un núcleo Rocket, en el otro una unidad de ejecución fuera de orden (BOOM). Cada uno de los recuadros cuenta con sus memorias caché de 1er nivel, tanto para datos como instrucciones, y ambos núcleos incluyen FPU y coprocesadores. Figura tomada de [5].

Como se señaló anteriormente, el diseño del sistema puede modificarse con tan sólo alterar el valor de una serie de parámetros. Dentro de las posibilidades disponibles, se puede modificar el tamaño del bus de datos,para que sea 32-bit, 64-bit o 128-bit. También es posible añadir, o eliminar, el soporte para distintos conjuntos de instrucciones de RISC-V, manteniendo siempre el soporte mínimo para el conjunto I (operaciones con enteros) [27]. Ciertos componentes pueden agregarse o eliminarse, como las Unidades de Punto Flotante, las memorias caché de segundo nivel, y los coprocesadores. A continuación, en el código fuente 2.1, se muestra un ejemplo de una configuración básica.

```
class WithNBigCores(n: Int) extends Config((site, here, up) => {
  case RocketTilesKey => {
    val big = RocketTileParams(
      core   = RocketCoreParams(mulDiv = Some(MulDivParams(
        mulUnroll = 8,
        mulEarlyOut = true,
        divEarlyOut = true))),
      dcache = Some(DCacheParams(
        rowBits = site(SystemBusParams).beatBits,
        nMSHRs = 0,
        blockBytes = site(CacheBlockBytes))),
      icache = Some(ICacheParams(
        rowBits = site(SystemBusParams).beatBits,
        blockBytes = site(CacheBlockBytes))))
    List.fill(n)(big) ++ up(RocketTilesKey, site)
  }
})
```

Código fuente 2.1: Ejemplo de una configuración básica de un microprocesador Rocket. Se observa como se cambian los parámetros del recuadro que contiene el procesador para modificar las memorias caché y la unidad de multiplicación y división del mismo. Con esta clase se pueden instanciar N procesadores con las características especificadas dentro del mismo recuadro.

Esta configuración puede utilizarse para instanciar n núcleos Rocket. Implícitamente se trata de una arquitectura de 64 bits, dado que el parámetro *XLen* por defecto tiene un valor de 64. Debido a que la constante *mulDiv* no se encuentra vacía, se implementa la unidad de multiplicación y división por hardware. Con esto se habilita el soporte para instrucciones del conjunto M de RISC-V. Por otro lado, esta configuración incluye una Unidad de Punto Flotante, debido a que por defecto el valor de la variable *fpu* es igual a *true*.

Para cambiar la arquitectura a una de 32 bits, es posible concatenar a dicha configuración una nueva clase, donde explícitamente se declare que XLen debe tener un valor igual a 32. Esta clase se presenta en el código fuente 2.2.

```
class WithRV32 extends Config((site, here, up) => {
  case XLen => 32
  case RocketTilesKey => up(RocketTilesKey, site) map { r =>
    r.copy(core = r.core.copy(
      mulDiv = Some(MulDivParams(mulUnroll = 8)),
      fpu = r.core.fpu.map(_.copy(divSqrt = false))))
  }
})
```

Código fuente 2.2: Configuración para modificar la arquitectura del procesador (para que sea de 32 bits). Al aplicar esta configuración también deben modificarse los parámetros mostrados de las unidades de Punto Flotante y Multiplicación y División.

Como se observa, para que esta configuración funcione correctamente, también es necesario modificar algunos parámetros para las unidades de multiplicación y división, y de punto flotante. Si además se desea eliminar la FPU, incluir un módulo de depuración JTAG (usando el *Debug Transport Module, DTM*), o cambiar el archivo de inicialización almacenado en la memoria *BootROM*, por ejemplo, se pueden utilizar las clases mostradas en los códigos fuente 2.3, 2.4 y 2.5, respectivamente.

```
class WithoutFPU extends Config((site, here, up) => {
  case RocketTilesKey => up(RocketTilesKey, site) map { r =>
    r.copy(core = r.core.copy(fpu = None))
  }
})
```

Código fuente 2.3: Configuración utilizada para eliminar la Unidad de Punto Flotante. Al retirar esta unidad se elimina también el soporte para la extensión de instrucciones F de RISC V.

```
class WithJtagDTM extends Config ((site, here, up) => {
  case IncludeJtagDTM => true
})
```

Código fuente 2.4: Configuración utilizada para instanciar un módulo de depuración JTAG al sistema.

```
class WithBootROMFile(bootROMFile: String) extends Config((site, here, up)
=> {
  case BootROMParams => up(BootROMParams, site).copy(contentFileName =
bootROMFile)
})
```

Código fuente 2.5: Configuración para cambiar el archivo de inicialización almacenado en la memoria *boot ROM*.

2.1.1 Protocolo TileLink

TileLink es un marco de referencia para protocolos que describe la comunicación entre distintos bloques que componen un sistema a escala de circuito integrado. Está diseñado de tal manera que sea un modelo común para las transacciones de coherencia caché dentro de una jerarquía de memoria definida [11]. De esta manera, es posible implementar un protocolo de coherencia y los diferentes agentes necesarios con base en esta especificación.

Con el ánimo de esclarecer el rol de TileLink dentro del sistema en chip, se debe diferenciar entre un protocolo de coherencia y la *política* o algoritmo de coherencia. El algoritmo, entre otras cosas, define como se representan los permisos y transferencias de diferentes bloques de memoria, así como sus reemplazos, de manera que se pueda sostener el modelo *Single Writer-Multiple Readers* [11]. Por otro lado, el protocolo define cuáles mensajes se deben intercambiar entre los agentes involucrados en la transacción para implementar dicho algoritmo. La ventaja que ofrece TileLink para este efecto es que implementa un protocolo sobre el cuál pueden implementarse uno o más algoritmos de coherencia deseados, sin que estos afecten las comunicaciones entre los agentes.

De manera similar a distintos protocolos de comunicación, sigue un esquema de Maestro-Esclavo. Dentro de TileLink, los agentes se denominan *Administrador* y *Cliente*. Utilizando TileLink dentro del diseño del sistema en chip, es posible lograr la comunicación entre los diversos componentes del mismo, ya sean núcleos de procesamiento, coprocesadores, aceleradores, controladores DMA, y demás, manteniendo la coherencia en la jerarquía de memoria.

2.1.2 Microarquitectura básica del núcleo de procesamiento (Rocket)

La microarquitectura básica del procesador Rocket está compuesta por un pipeline de 5 etapas (6 si se toma en cuenta el generador de contador de programa, **genpc**). La ejecución de las instrucciones es en orden. En la figura 2.2 se observa el diagrama simplificado del procesador. El núcleo Rocket es, en general, capaz de ejecutar las instrucciones del conjunto G de RISC V, que incluye las instrucciones básicas para números enteros (conjunto I), una extensión para Multiplicaciones y Divisiones (M), las extensiones para operaciones en punto flotante de precisiones sencilla y doble (F y D, respectivamente), y por último las extensiones que incluyen las instrucciones atómicas (A) y comprimidas (C). Sin embargo, para la implementación más sencilla del procesador es posible eliminar la Unidad de Punto Flotante, además de deshabilitar la ejecución de instrucciones atómicas y comprimidas. Con esto el conjunto de instrucciones que ejecuta se reduce únicamente a IM.

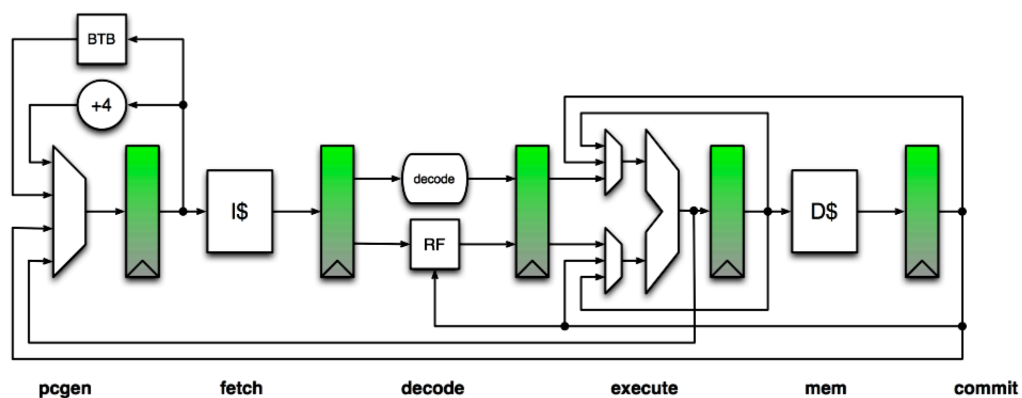


Figura 2.2: Microarquitectura del núcleo Rocket, con las etapas (simplificadas) del pipeline. En la etapa de decodificación, el bloque denominado RF corresponde al banco de registros (*Register File*). El componente BTB corresponde a una unidad de almacenamiento de destino de saltos (*Branch Target Buffer*), que asiste en la predicción de saltos. Figura tomada de [15].

Con el objetivo de reducir el tamaño del procesador, su complejidad, y con esto, su consumo de potencia, se debe implementar una arquitectura de 32 bits, con soporte para la ejecución de un subconjunto de instrucciones menor a G. Al eliminar la unidad de punto flotante (FPU), se deshabilita el roporte para los conjuntos de instrucciones F y D. Con esto el conjunto de instrucciones que ejecuta se reduce a IMAC.

2.1.3 Memorias Caché de 1er Nivel

En el diseño básico se utilizan dos memorias caché dentro del recuadro que contiene el núcleo. Estas memorias caché utilizan un *mapeo asociativo de conjuntos* (*Set-Associative Mapping*). De esta manera, utilizando ciertos parámetros que se encuentran expuestos en el *RocketChip Generator*, es posible modificar su tamaño y asociatividad.

Los parámetros son:

- nWays.
- nSets.

El parámetro *nSets* determina el número de conjuntos utilizados. Estos conjuntos, a su vez, están compuestos por una cantidad de líneas determinada por el parámetro *nWays*. Estas líneas son las unidades fundamentales a las que cada bloque de la memoria principal se asocia utilizando el mapeo. Estos bloques, y consecuentemente, las líneas de las memorias caché, pueden almacenar una cantidad de palabras o bytes según el diseño de la memoria. Para RocketChip, cada línea puede almacenar 64 Bytes.

Según [24] el mapeo asociativo por conjuntos está planteado de manera que tenga las ventajas de los mapeos directo y asociativo, minimizando sus desventajas. Cada dirección de la memoria principal se interpreta dividiéndola en tres partes: la *etiqueta* (*tag*), el *conjunto* (*set*), y la *palabra* (*word*). Los d bits que especifican el conjunto pueden apuntar a cualquiera de los 2^d conjuntos definidos en los parámetros expuestos en RocketChip. Además, para representar el bloque de memoria, se utilizan s bits, que contienen tanto los bits de la etiqueta, como los bits del conjunto. Con estos se puede especificar cualquiera de los 2^s bloques de la memoria principal. Al aumentar la cantidad de conjuntos, el tamaño de la etiqueta disminuye considerablemente con respecto al tamaño de la misma utilizando un mapeo de memoria puramente asociativo. Conforme disminuye la cantidad de líneas por conjunto, el mapeo se acerca más al comportamiento de un mapeo directo. Por otro lado, si la cantidad de conjuntos se acerca a 1, y el número de líneas por conjunto es igual al número de líneas en la caché, el mapeo se comporta como un mapeo puramente asociativo [24].

2.1.4 Soporte de Depuración Externa para RISC-V

Actualmente, RocketChip implementa un sistema de depuración externa que cumple con la norma *RISC V External Debug Support 0.13*, para la verificación del funcionamiento del SoC [21]. Esta depuración se hace por medio de un dispositivo externo, como una laptop, que se conecta con la plataforma RISC-V utilizando hardware de depuración (e.g. un adaptador USB/JTAG). Dentro de la plataforma RISC-V, en este caso RocketChip, se encuentran el módulo de transporte de depuración (*Debug Transport Module*, DTM), una interfaz DMI (*Debug Module Interface*) y un módulo de depuración (*Debug Module*, DM).

Por medio del hardware mencionado, un usuario que utilice el dispositivo externo puede emitir comandos y enviar datos a la plataforma, para así modificar el comportamiento de la plataforma con el fin de verificar su funcionamiento. Utilizando esta herramienta se tiene la capacidad de detener o reanudar la ejecución de un *hart* (*Hardware Thread*). En la figura 2.3 se observa la arquitectura de este sistema de depuración.

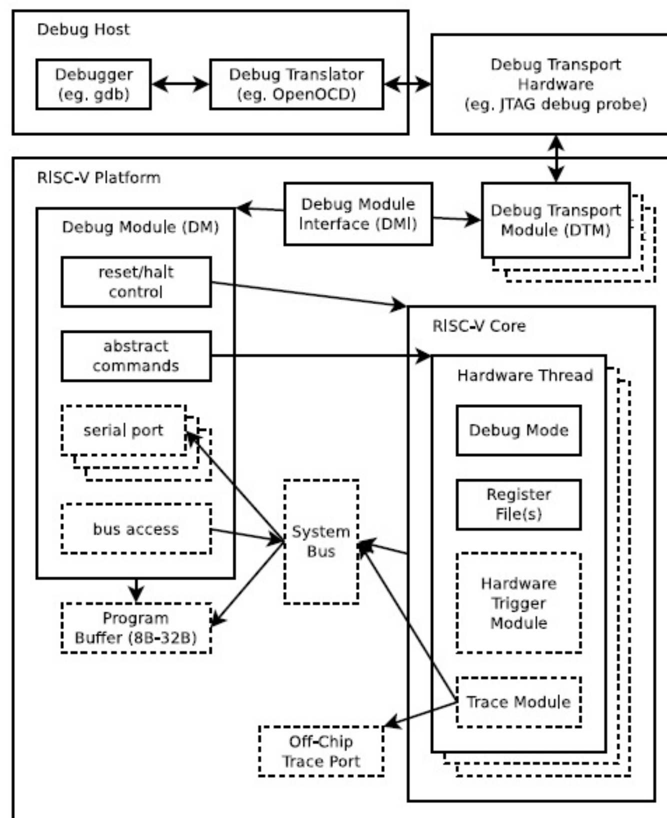


Figura 2.3: Arquitectura del sistema de depuración externa. Se observa como la interfaz DMI sirve de puente entre el módulo de depuración DM y el DTM. Figura tomada de [21].

2.1.5 Diagramas de referencia

En la figura 2.4, se observan las etapas del procesador de una manera detallada. La memoria cache de instrucciones se encuentra en la etapa Fetch. Por otra parte, la memoria cache de datos se observa en la etapa Memory. En la etapa de Ejecución se tienen las unidades aritmético-lógica, IMUL e IDIV. En la implementación básica del núcleo, dado que se elimina el soporte para el conjunto M de instrucciones, la unidad IMUL no aparece.

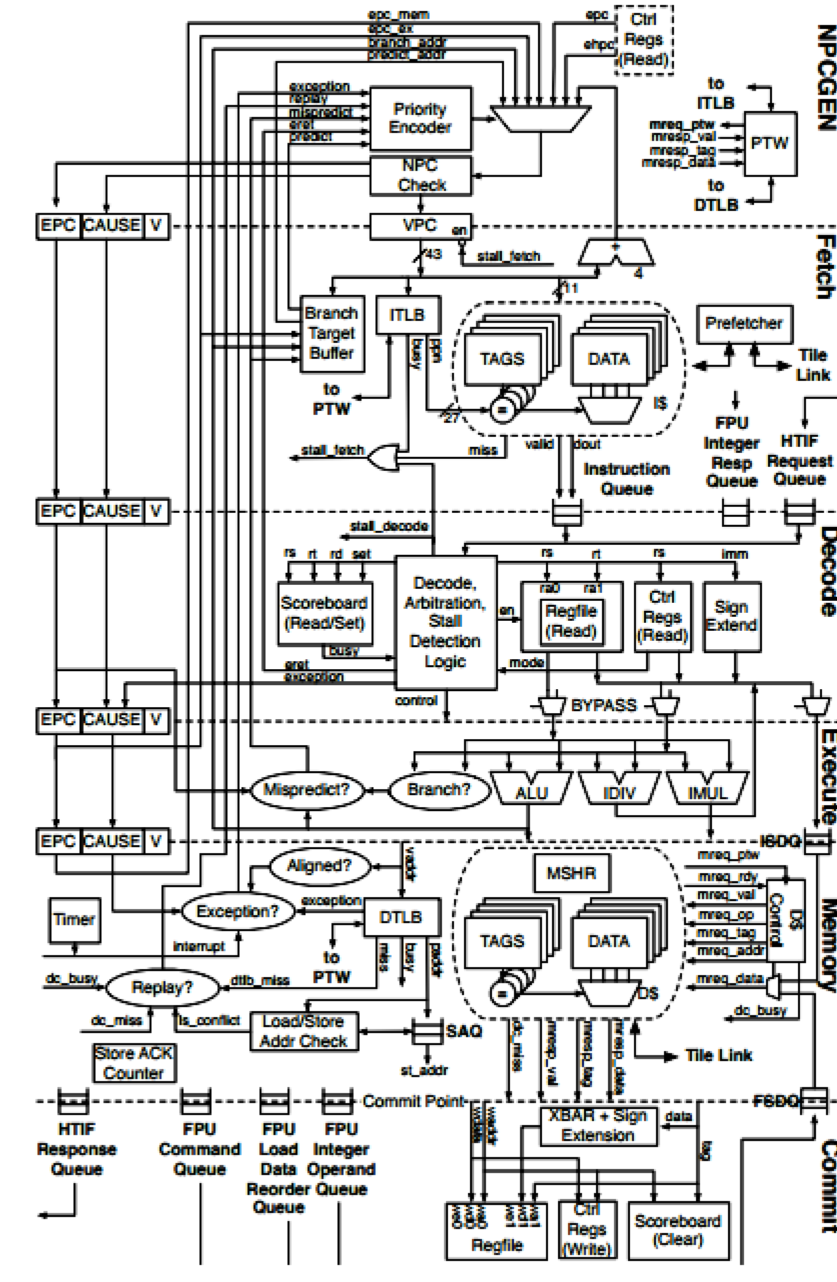


Figura 2.4: Microarquitectura del núcleo de procesamiento Rocket. Se muestran los bloques funcionales de los que está compuesto el mismo. Se pueden observar las señales y colas que comunican el núcleo con la FPU, TileLink, HTIF (Host Target Interface) y PTW. Figura tomada de [15].

En las figuras 2.5, 2.6 y 2.7, se muestran diagramas de bloques para el generador del contador de programa, además de las diferentes etapas del pipeline del procesador. También se tiene un diagrama de la memoria caché de datos. Los nombres de las señales en estos diagramas corresponden a los de la descripción RTL.

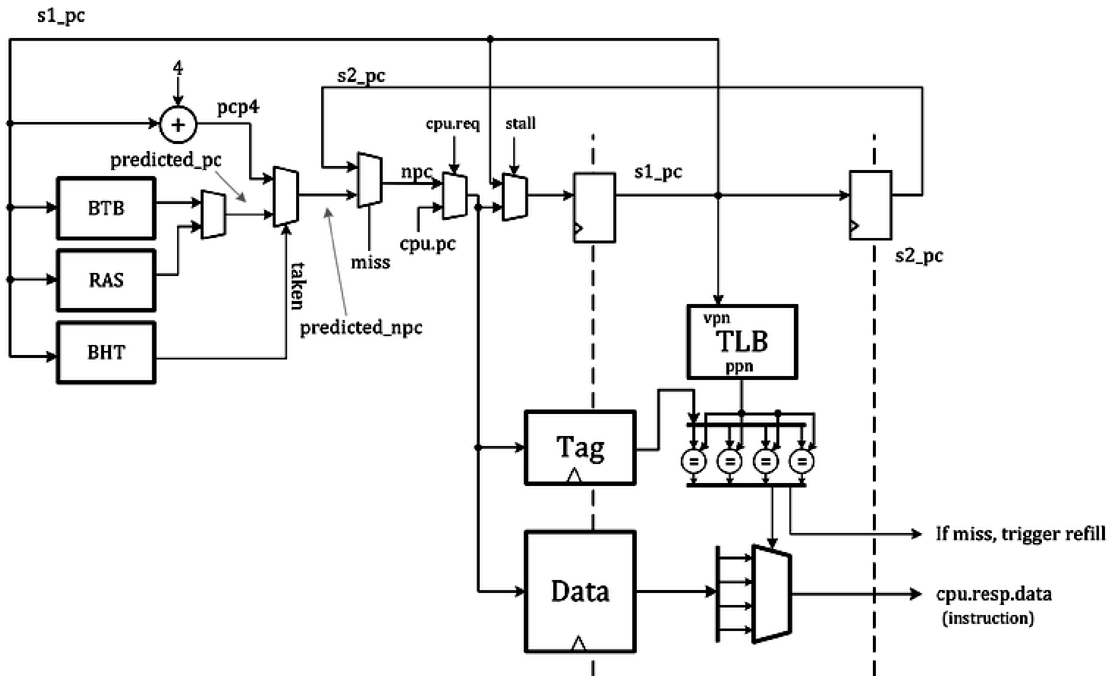


Figura 2.5: Generador de PC y etapa Fetch. Figura tomada de [3].

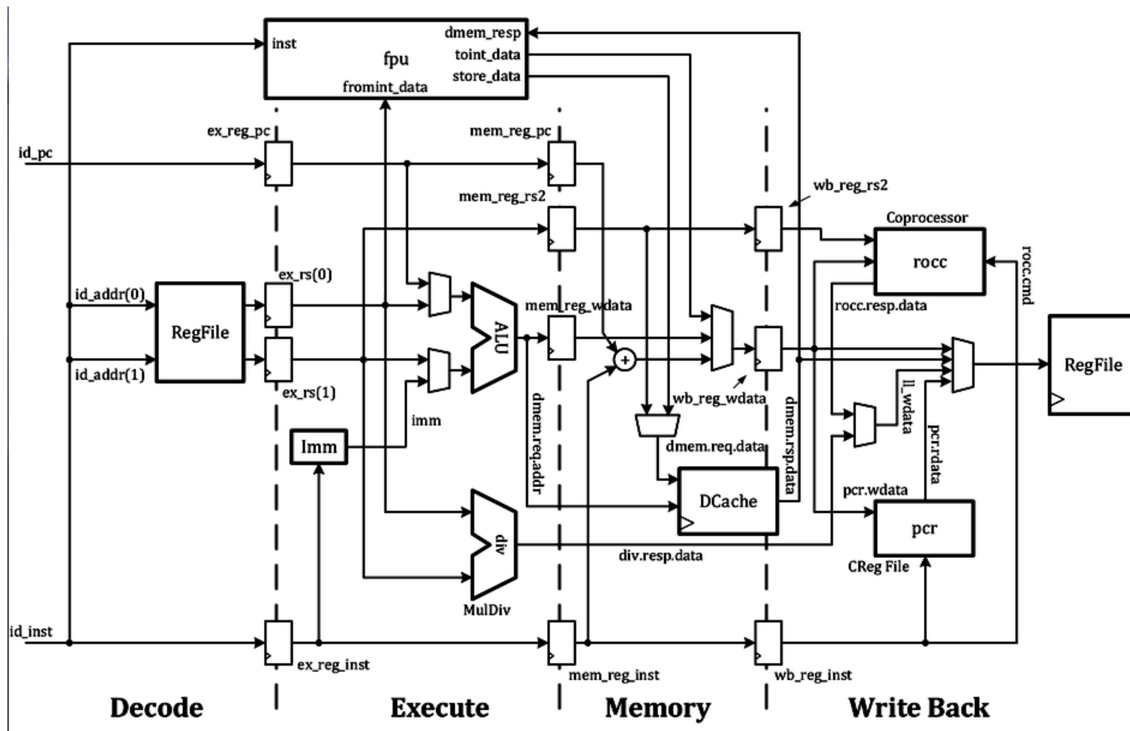


Figura 2.6: Etapas del pipeline. En la implementación básica no se incluye la unidad de punto flotante. Figura tomada de [3].

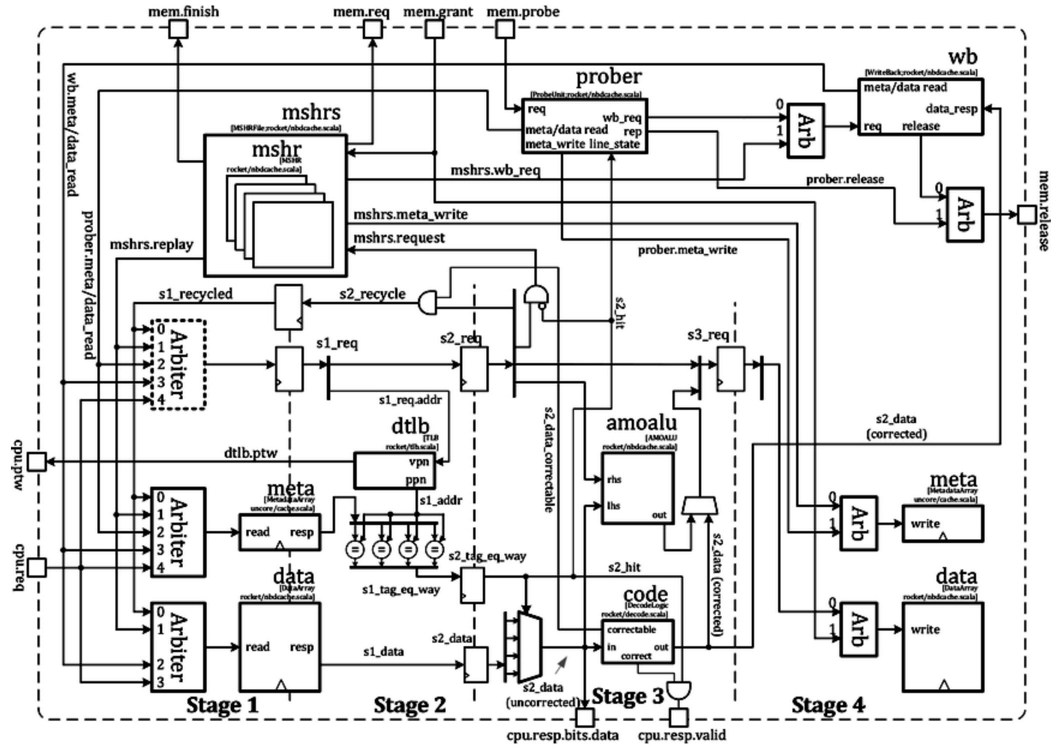


Figura 2.7: Memoria caché de datos. Figura tomada de [3].

Cabe resaltar que estos diagramas son una guía, más que una descripción exacta de la microarquitectura básica generada. Esto porque en estos aún se muestran unidades que se eliminaron para disminuir el consumo de potencia y el área utilizada, como la unidad de punto flotante. Además la ubicación en la que e muestran algunos módulos no corresponde exactamente a su posición en la descripción RTL. Un ejemplo de esto son las memorias caché D e I, que se muestran dentro de las etapas del pipeline, pero en realidad se implementan en módulos separados del núcleo, siempre dentro del mismo recuadro.

2.2 Correlación Cruzada de Señales

Para validar el sistema desarrollado en este proyecto, se propone la implementación de un algoritmo de correlación cruzada para la detección de eventos (latidos) en electrocardiogramas. En esta sección se presentan los fundamentos teóricos que respaldan este método.

La correlación entre dos señales es un método matemático que permite obtener el grado de similitud entre dos señales, o el desfase entre las mismas [22]. Resulta una herramienta útil cuando se desea extraer características de señales desconocidas, valiéndose de la comparación con una señal previamente estudiada.

Matemáticamente se puede definir como

$$(f \star g)(\tau) = \int_{-\infty}^{\infty} f^*(t)g(t+\tau)dt \tag{2.1}$$

para señales continuas. Si se trabaja con señales discretas, la definición cambia a

$$(f \star g)[t] = \sum_{k=0}^{n-1} f^*[k]g[t+k] \tag{2.2}$$

donde f* es el conjugado complejo de f [17].

Se puede observar que, dada la naturaleza de la correlación, es conveniente realizar el cálculo utilizando las señales representadas en un formato *bitstream*. Con esto es posible reducir la carga computacional [25]. La operación adecuada para obtener el resultado de la correlación es una XNOR, ejecutada bit por bit, como se puede observar en la tabla 2.1.

Tabla 2.1: Tabla de verdad de la función lógica XNOR.

A	B	XNOR $\odot$
0	0	1
0	1	0
1	0	0
1	1	1

Se observa como esta función lógica es adecuada para calcular la correlación, pues siempre y cuando el valor de cada bit coincida con entre ambas señales, el valor de salida será un 1 lógico. Con esto, entre mayor sea la similitud de las señales comparadas, mayor será la cantidad de 1's dentro del resultado, como es de esperarse.

2.3 Representación de señales en formato *bitstream*

Normalmente, la representación *bitstream* de señales se usa en las etapas intermedias de convertidores Analógico/Digitales (ADC), o Digital/Analógicos (DAC) de tipo Sigma Delta ($\Sigma\Delta$). Este tipo de dispositivos se denominan convertidores de sobremuestreo, pues trabajan a una frecuencia mayor a la frecuencia de Nyquist de las señales de interés. La tasa de muestreo se representa como

$$F_s = OSR * f_s \tag{2.3}$$

donde F_s es la tasa de muestreo del convertidor, OSR es la razón de sobremuestreo (*Over-Sampling Ratio*), y f_s es la frecuencia de Nyquist. El objetivo de utilizar esta técnica es lograr señales de alta fidelidad, disminuyendo el error de codificación.

En principio, estos dispositivos utilizan un modulador $\Sigma\Delta$ para obtener la representación bitstream de la señal. Posteriormente, se realizan dos procesos adicionales: un filtrado de la señal, para pasar de una modulación por densidad de pulsos a una modulación por pulsos codificados; y por último un diezmado, para bajar el muestreo de la señal resultante a la frecuencia de muestreo de Nyquist. Con esto se obtiene una representación fiel de la señal de entrada, sin acumular datos innecesarios a la salida, pues la tasa de muestreo se disminuye hasta alcanzar el límite donde se puede asegurar que no existe aliasing.

2.4 Señales de Electrocardiogramas: ECG

Un electrocardiograma se define como la grabación y registro del cambio de potenciales eléctricos con respecto al tiempo. Estos potenciales se refieren particularmente a los que se producen debido a la contracción de los músculos cardíacos (miocardio) [14]. Los electrocardiogramas representan una de las herramientas de diagnóstico más utilizadas en cardiología, por la gran cantidad de información que pueden contener.

El uso de los electrocardiogramas se ha extendido debido a las ventajas que posee. En primer lugar, las señales son fáciles de obtener, debido a que se requiere un procedimiento poco invasivo. Los electrodos utilizados para la adquisición de las señales se colocan en la superficie de la piel, sin necesidad de intervenciones quirúrgicas. Otra ventaja que poseen es su bajo costo, lo que convierte a esta técnica en un método posible de implementar en diferentes escenarios.

2.4.1 Mecanismos de conducción eléctrica en el corazón

Para comprender el origen de las señales que se ven representadas en la grabación de un electrocardiograma, es necesario introducir el sistema de conducción eléctrica del corazón. En primer lugar, se debe resaltar que la actividad eléctrica en este órgano es producto de reacciones químicas, las cuáles toman lugar en ciertas secciones particulares del mismo.

Los músculos cardíacos están compuestos, principalmente, por dos tipos de células. Se tienen los cardiomiocitos, que al contraerse generan potenciales eléctricos. A su vez, se tienen células especializadas en la generación y conducción de los potenciales de acción [14]. Los potenciales de acción se refieren a descargas eléctricas dentro del tejido, que cambian la distribución de las cargas eléctricas en el mismo.

En general, en estado de reposo, los cardiomiocitos se encuentran polarizados con un potencial de membrana de alrededor de -90 mV [14]. Es importante aclarar, en este punto, que al mencionar el potencial de membrana se refiere a la diferencia de potencial eléctrico a ambos lados de una membrana celular. Esta diferencia de potencial se ocasiona por la presencia de dos sustancias de diferente concentración de iones a ambos lados de la

membrana. Ahora, un estímulo externo puede iniciar un proceso de despolarización, en el cual se permite la entrada de iones positivos de sodio (y en algunos casos, de calcio) dentro de la membrana. En este proceso se revierte el potencial eléctrico en las células del miocardio. Luego de la despolarización, el músculo cardíaco vuelve a su estado eléctrico inicial.

La fase de repolarización se refiere al cambio hacia abajo del potencial de acción, ocasionado principalmente por la salida de iones de potasio fuera de la célula. Una característica importante de este proceso de repolarización es que durante este el miocardio no puede ser estimulado, lo que protege el músculo contra estímulos prematuros.

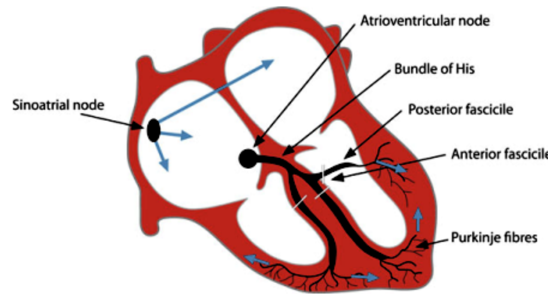


Figura 2.8: Ilustración del corazón, en la que se señalan los elementos principales involucrados en los procesos de conducción eléctrica. Las cavidades derechas (aurícula y ventrículo) se ubican a la *izquierda* de la ilustración, según el punto de vista del lector. Figura tomada de [14].

En la figura 2.8 se muestra una ilustración básica de la estructura del corazón, prestando especial atención a los elementos involucrados en la conducción eléctrica dentro del miocardio. Se observan 4 cavidades principales [28]:

- Aurículas: cavidades superiores, divididas a su vez en aurícula derecha y aurícula izquierda. Estas cavidades se encargan de recolectar la sangre que ya ha circulado por todo el organismo.
- Ventrículos: cavidades inferiores, que se dividen en ventrículo izquierdo y ventrículo derecho. Estas cavidades toman la sangre de su respectiva aurícula, y se contraen para bombear la sangre de vuelta al resto del cuerpo.

Se observa que existe una división en la organización de las cavidades, de manera que los lados izquierdo y derecho se encuentran separados. Esta división tiene un carácter funcional, pues responde a la necesidad de cada uno de los lados de proveer la sangre para una sección del organismo específica. El lado derecho se encarga de la circulación de la sangre en el sistema pulmonar, mientras que la región izquierda hace su parte para el resto de los sistemas del organismo [28].

En un estado normal, las contracciones que permiten el movimiento de la sangre a través de dichas cavidades están reguladas por un marcapasos natural. Esta función se lleva a cabo en el *nodo sinoauricular* (*sinoatrial node*). Este nodo se denota con la abreviatura **SA**. La función principal del nodo SA es la de generar impulsos eléctricos que inician la con-

tracción de la aurícula derecha, mediante procesos de despolarización espontánea. Además, el sistema de conducción incluye el *nodo atrio ventricular* (**AV**), ubicado en la cercanía de la división entre las cavidades superior e inferior. Las células de este nodo también presentan un proceso de despolarización espontáneo, pero a una frecuencia menor que la del nodo SA. Esta actividad funciona (en parte) como un respaldo para la despolarización del nodo SA, y en condiciones normales el efecto de estos procesos es suprimido por la acción de este mismo nodo. Por último, algunos otros elementos importantes de este sistema son el *haz de His*, y las *fibras de Purkinje*. El haz de His es una membrana celular que permite la conducción de pulsos eléctricos desde el nodo AV hacia otras regiones del corazón. Las fibras de Purkinje también presentan *automaticidad*, generando pulsos a una tasa de aproximadamente 30 bpm [14].

2.4.2 Elementos importantes presentes en un electrocardiograma

Los electrocardiogramas almacenan información importante sobre el funcionamiento y estado del sistema cardiovascular del paciente estudiado. Para facilitar su interpretación, en un electrocardiograma se identifican diversas secciones y elementos que evidencian el funcionamiento de los diferentes componentes del sistema de conducción eléctrico del corazón.

En la figura 2.9 se presentan algunas de las secciones, intervalos y ondas más importantes que se observan en un electrocardiograma.

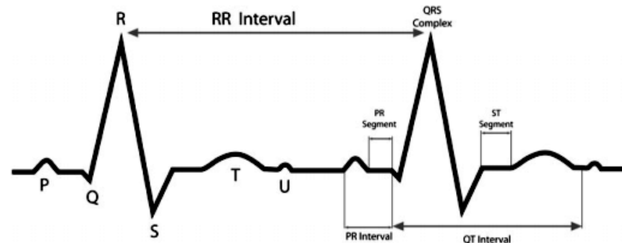


Figura 2.9: Ejemplo típico de un electrocardiograma, se observan las principales ondas e intervalos que se presentan en un ECG. Se pueden observar los intervalos RR, PR y QT, además del complejo QRS y las ondas T y P. Figura tomada de [14].

El primer elemento que se presenta en un ECG es la onda P, característica de la despolarización de las aurículas. El proceso de repolarización de estas cavidades no se puede observar en un electrocardiograma, pues ocurre durante la despolarización de los ventrículos y es opacado por esta actividad. La onda P tiene una duración normal de 0,12 segundos, y su amplitud es cercana al rango de 0,25 - 0,15 mV, al utilizar las derivaciones de las extremidades superiores. Si esta amplitud supera los valores mencionados, y además la onda presenta muescas o deformaciones, se considera que existe un comportamiento anormal, lo que puede indicar deformaciones en las aurículas.

Otro elemento importante que puede observarse es el complejo QRS, que es la manifestación de la despolarización de las cavidades ventriculares. Como se puede intuir de su nombre, este complejo se compone por las ondas Q, R y S. Típicamente, la duración de este complejo no es mayor a los 0,12 segundos, similar a la onda P. Una duración mayor a este valor puede resultar de asincronías en la despolarización de ambos ventrículos. Esto ocurre en casos en los que se presentan síndromes de pre-exitación o contracciones ventriculares prematuras.

El *intervalo RR* representa la duración de un *ciclo cardíaco*, y con este se puede determinar ritmo cardíaco del sujeto de la prueba. En este informe, el ciclo cardíaco utilizado para la obtención del patrón y la detección de eventos corresponde al segmento del electrocardiograma desde la onda P hasta las vecindades de la onda U.

Capítulo 3

Implementación de un SoC basado en RocketChip

Como un primer acercamiento a la resolución del problema descrito, se plantea utilizar un microprocesador de uso libre para evaluar su rendimiento ante tareas y algoritmos típicos de dispositivos médicos implantables. Para este efecto se deben considerar diferentes candidatos para realizar la evaluación. El primer aspecto a tomar en cuenta es la arquitectura de set de instrucciones a utilizar. Para esta variable, la mayor parte del desarrollo tecnológico cae dentro de dos categorías: **RISC** (Reduced Instruction Set Computer) y **CISC** (Complex Instruction Set Computer). Algunas de las ventajas y desventajas de ambas opciones se presentan en la tabla 3.1.

Se observa que, por la naturaleza de los implantes médicos, los computadores RISC presentan características ventajosas. Por ejemplo, al necesitar componentes de hardware de menor complejidad, el área del microprocesador, y su consumo de potencia, se ven disminuidos. Además, a pesar de que cada programa, al expresarse en lenguaje ensamblador, resulta más complejo y extenso por la capacidad reducida de las instrucciones RISC, esta complejidad extra se enmascara dentro de los compiladores, que tienden a ser muy eficientes para estas arquitecturas.

El siguiente paso es escoger, dentro de los Sets de Instrucciones RISC, el más adecuado para el desarrollo del proyecto. Algunas opciones son ARM, OpenRISC, MIPS y RISC V. De estas, ARM y MIPS son propietarias, por lo que su uso está limitado por el pago de licencias. Por otro lado, OpenRISC es una arquitectura que carece de un ambiente activo de desarrollo y mantenimiento, por lo que también se limita su factibilidad. Por otro lado, RISC V cuenta con un conjunto de organizaciones activas en el área de la investigación y el desarrollo comercial. Por ser un estándar vigente, con amplio respaldo, documentación y mantenimiento, presenta las mayores ventajas como plataforma base para el desarrollo de proyectos de hardware libre.

Tabla 3.1: Comparación entre dos categorías de arquitecturas de computadores.

Arquitectura de Computador	Ventajas	Desventajas
CISC	Permite diseñar software con una menor cantidad de instrucciones. El procesador debe ejecutar menos instrucciones.	Aumento de la complejidad del hardware, e. g. el decodificador es más lento y requiere un área mayor [6]. El tiempo de ejecución por instrucción es mayor.
RISC	Hardware más sencillo. Menor tiempo de ejecución por instrucción. Modos de direccionamiento, y formato de instrucciones sencillos [24].	El software se vuelve más complejo y elaborado. Se necesitan más líneas de código para realizar las mismas tareas, i.e. menor densidad de código [6].

Parte del auge experimentado por el set de instrucciones RISC V se debe a su adopción, para la implementación de microprocesadores y sistemas en chip, por parte de grupos de investigación de universidades como ETH Zurich, la Universidad de Bolonia, y la Universidad Industrial de Santander, por nombrar algunos [2]. En su mayoría, estos operan bajo licencias abiertas (como las licencias Apache, BSD y MIT), lo que los transforma en candidatos adecuados para el desarrollo de plataformas estándar de implantes biomédicos. Como ejemplos de estas plataformas se pueden nombrar el SoC RocketChip (junto al procesador Rocket), y los procesadores ORCA, PULPino y OPenV/mriscv [2]. De estas plataformas, se elige trabajar con RocketChip debido a su descripción en alto nivel, su alto grado de parametrización, y la flexibilidad que ofrece para modificar la microarquitectura para adaptarla a la aplicación planteada.

Una vez definida la plataforma de hardware a utilizar, se procede a realizar el diseño del Sistema en Chip deseado, con el que se pueden realizar pruebas de funcionamiento y evaluaciones. RocketChip, además de proveer el SoC, cuenta con un conjunto de *benchmarks* y *asm-tests* (*assembly tests*), para la verificación y estudio de la ejecución de instrucciones RISC V. Con estos programas de prueba es posible conocer el rendimiento de la arquitectura generada, además de verificar la ejecución de las instrucciones relevantes.

3.1 Sistema en Chip Generado

El código en Chisel necesario para describir el SoC deseado se presenta en el código fuente 3.1. Cabe resaltar que, dentro de la clase *WithNTinyCores*, se instancian dos memorias caché de primer nivel, para instrucciones y datos. Sin embargo, la caché de datos (en la declaración original) cuenta con 256 *Sets*. Por esta razón, para disminuir su tamaño (y con esto el tamaño del sistema), se limita este número a 64. Con esta configuración, se genera un *SoC* que cuenta con un solo núcleo de procesamiento de 32 bits, el cuál está desprovisto de FPU, y cuenta con 2 memorias caché de 4096 bytes.

```
class RocketRV32i extends Config(
  // Estas dos primeras configuraciones deben estar presentes
  new WithNMemoryChannels(0) ++
  new WithStatelessBridge ++
  // TinyCores es de 32bits, sin FPU, con MulDiv
  new WithNTinyCores(1) ++
  // Para limitar el Set de Instrucciones a IM
  new WithoutAtomics ++
  new WithoutCompressed ++
  // Las memorias caché están separadas en D e I
  // Se usan solo de primer nivel
  // Para cambiar el tamaño de las memorias caché
  new WithL1DCacheSets(64) ++
  new WithL1ICacheSets(64) ++
  new WithL1ICacheWays(1) ++
  new WithL1DCacheWays(1) ++
  // Para eliminar los TLMonitor
  new WithoutTLMonitors ++
  // Con WithCacheBlockBytes(linesize) se puede cambiar el tamaño de la linea
  new BaseConfig)
```

Código fuente 3.1: Configuración del sistema en chip diseñado. Se observan las clases utilizadas para disminuir el conjunto de instrucciones que se implementa. También se observa como se modifican los parámetros de las memorias caché de 1er nivel para disminuir su tamaño.

En esta definición se optó por eliminar ciertos componentes que proveen funcionalidades del protocolo TileLink. Específicamente, se eliminan los componentes denominados *TLMonitor*, que se encargan de vigilar el desarrollo de las transacciones entre los diferentes agentes [23]. Sin embargo, estos se vuelven innecesarios debido a que en el SoC diseñado existe un único agente capaz de emitir operaciones de escritura y lectura en la memoria, i.e.

el procesador Rocket. Con esto se puede reducir el tamaño y la complejidad del sistema generado, teniendo en cuenta que no resulta necesario tener tantas funcionalidades complejas para la coherencia de las memorias en un sistema con un solo procesador y un solo nivel de caché. Como se puede observar, la clase implementada, **RocketRV32i**, se basa en la configuración **BaseConfig**. Esta clase provee las descripciones para los componentes básicos del sistema, y se muestra a continuación, en el código fuente 3.2.

```
class BaseConfig extends Config(new BaseCoreplexConfig().alter((site,here,up)
   => {
  // DTS descriptive parameters
  case DTSModel => "freechips,rocketchip-unknown"
  case DTSCompat => Nil
  // External port parameters
  case IncludeJtagDTM => false
  case JtagDTMKey => new JtagDTMKeyDefault()
  case NExtTopInterrupts => 2
  case ExtMem => MasterPortParams(
                      base = 0x80000000L,
                      size = 0x10000000L,
                      beatBytes = site(MemoryBusParams).beatBytes,
                      idBits = 4)
  case ExtBus => MasterPortParams(
                      base = 0x60000000L,
                      size = 0x20000000L,
                      beatBytes = site(MemoryBusParams).beatBytes,
                      idBits = 4)
  case ExtIn  => SlavePortParams(beatBytes = 8, idBits = 8, sourceBits = 4)
  // Additional device Parameters
  case ErrorParams => ErrorParams(Seq(AddressSet(0x3000, 0xfff)))
  case BootROMParams => BootROMParams(contentFileName="./bootrom/bootrom.img")
}))
```

Código fuente 3.2: Configuración base del sistema en chip. Esta configuración es el punto de partida para el diseño del sistema deseado.

En esta configuración base, entre otras cosas, se definen los espacios de memoria para la memoria externa y el bus de comunicaciones. Estos se incorporan al Device Tree, que contiene la descripción del mapeo de memoria de la totalidad del sistema. También se instancian en esta clase los módulos de depuración (JTAG Debug Transport Module) y la BootROM. Se puede observar como se carga la imagen a la BootROM.

3.2 Simulaciones y ejecución de programas de prueba

El generador viene acompañado de un conjunto de pruebas, llamadas *asm-tests (assembly tests)*. Estos son programas sencillos que buscan explorar y probar la ejecución de una instrucción en específico. Por ejemplo, se tienen programas de prueba para las instrucciones add, addi, and, andi, beq, etc. Estos programas no son almacenados dentro de la memoria del procesador directamente. En su lugar, son escritos en la memoria durante el *runtime* de la simulación. Para este efecto se utilizan 2 herramientas:

- *Front-end server* de RISC-V (riscv-fesvr): Este se encarga de indicar cuál programa se debe ejecutar, y de emitir las solicitudes para escribir dicho programa en la memoria del sistema. Toma el papel del "maestro" durante el proceso.
- *Debug Transport Module*: denotado como *SimDTM* en el nivel más alto de la jerarquía del sistema. Este se encarga de la comunicación con el módulo de depuración, que se encarga de escribir el programa en la memoria. En relación con el *front-end server*, asume el papel de esclavo.

El *Debug Transport Module* está compuesto por dos archivos fuente separados. Uno especifica una función en C++, mientras que el otro describe el hardware utilizando Verilog. La función en C++, denominada *debug_tick*, se enlaza con el módulo utilizando la interfaz *SystemVerilog DPI (Direct Programming Interface)*.

Durante el *runtime* de la simulación, el núcleo Rocket se inicializa ejecutando código almacenado dentro de la memoria *bootrom*. Seguidamente, el DTM interrumpe este proceso "inyectando" porciones del código de prueba que se desea ejecutar (señalado por el *front-end server*), hasta que el programa de prueba se almacena en su totalidad en la memoria del sistema. Para esto, el núcleo trabaja en el modo de depuración, el cual es diferente al modo máquina debido a que posee mayores privilegios y permisos [26]. Al finalizar el proceso de carga, el DTM hace que el núcleo retome la ejecución desde la primer línea del programa de prueba. Para el modo de depuración se reservan algunos de los registros de Control y Estado (*CSRs*), además de una porción del espacio de memoria. El *Debug Transport Module* se describe en la especificación del Soporte de Depuración Externa de RISC-V.

Capítulo 4

Correlación cruzada con señales *bitstream*

Para demostrar la viabilidad del uso de un microprocesador como Rocket en aplicaciones biomédicas, se implementa un algoritmo de correlación cruzada, para su respectiva evaluación. El algoritmo está basado en una representación en formato bitstream de dos señales, de las cuales una es un patrón conocido. La segunda de las señales resulta de la medición en tiempo real de la variable física de interés. Una vez que se obtiene esta representación, se procede a calcular la correlación cruzada entre ambas señales, para obtener el grado de similitud entre las mismas. La principal ventaja del uso de este tipo de representación es que las operaciones aritméticas se pueden realizar bit por bit (*bitwise*), con lo que se aprovechan las capacidades inherentes de la unidad aritmético-lógica del microprocesador implementado. De esta manera se disminuye la carga computacional requerida por parte del microprocesador.

El algoritmo de detección de eventos con el que se trabaja, ha sido implementado a nivel RTL anteriormente, tanto para su uso con señales acústicas de dispares y motosierras [19] [8] [20], como con electrocardiogramas [25] [17].

El diseño del algoritmo de correlación debe considerar el tipo de señal que se utilizará, tanto para el vector patrón, como para los vectores de prueba. Debido a que el SoC utilizado no cuenta con los periféricos necesarios para el procesamiento de señal mixta (i.e. ADCs ni DACs), dichos vectores deben ser modulados previamente para obtener su representación *bitstream*.

4.1 Obtención de las señales ECG

Para obtener las señales de electrocardiogramas se utilizó la base de datos *ECG-ID Database* [18], disponible en el repositorio en línea PhysioNet [13]. Esta base de datos es producto de la investigación realizada en [18], que tenía como objetivo la demostración de la viabilidad de utilizar los electrocardiogramas, y la información contenida en ellos, como identificadores

biométricos. Esto se basa en la hipótesis de que el ritmo cardíaco y el comportamiento del sistema circulatorio en general es particular y característico de cada persona. Esta base de datos contiene 310 grabaciones de electrocardiogramas de 90 voluntarios, de 20 segundos cada una, obtenidas de la derivación I, la cual corresponde a la medición de la diferencia de potencial entre las manos derecha e izquierda [18]. Las señales se encuentran digitalizadas con una resolución de 12 bits, con un rango de ± 10 mV, y una tasa de muestreo $F_s = 500$ Hz.

Cada una de las grabaciones de esta base de datos presenta tanto la señal pura (i.e. sin pre-procesamiento) como la señal filtrada. La grabación sin filtrar presenta mucho ruido, con componentes en altas y bajas frecuencias que no corresponden a datos relevantes para el estudio del electrocardiograma. Debido a que, como se mencionó anteriormente, las grabaciones contienen características particulares de cada individuo, el patrón obtenido a partir de los datos de cada paciente es adecuado únicamente para hacer pruebas con otras grabaciones de ese mismo paciente.

Para la obtención del patrón, se tomaron 3 porciones, de igual longitud, de la primera grabación del paciente 1. Cada una de las porciones es de 352 muestras, lo que corresponde a 0,7 segundos de grabación. Estas porciones contienen un ciclo cardíaco completo donde se pueden observar el complejo QRS, la onda T y la onda P. Un ejemplo de este ciclo se puede observar en la figura 4.1.

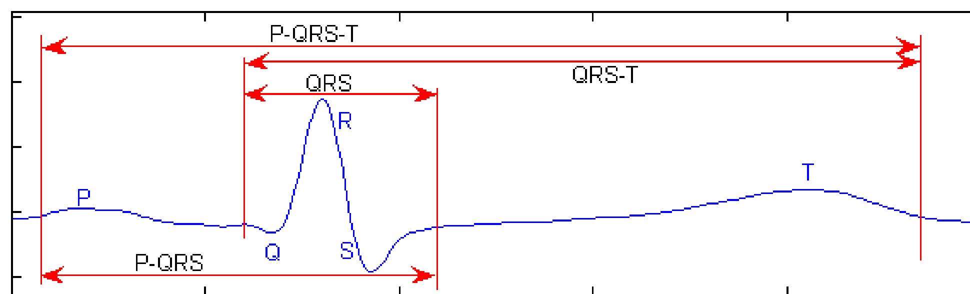


Figura 4.1: Ciclo cardíaco completo, con sus diferentes fragmentos informativos, obtenido de la derivación I. Figura tomada de [18].

Después de separar estas 3 porciones, se calcula un promedio de las tres señales, con lo que se obtiene una señal adecuada para actuar como patrón de comparación. El resto de eventos de la grabación utilizada se utilizan como pruebas para el resto del algoritmo.

4.2 Diseño del modulador $\Sigma\Delta$

Para obtener la representación bitstream de las señales se utiliza un modulador Sigma-Delta ($\Sigma\Delta$). Como se explicó anteriormente, en esta representación, el valor instantáneo de la señal viene dado por la densidad de 1's dentro de una porción de la señal. El modulador $\Sigma\Delta$ se implementa con un script de Python, utilizando el paquete *python-sigmadelta*. Este es una adaptación, de uso libre, del *Delta Sigma Toolbox* para MATLAB. De este paquete se utilizan principalmente dos funciones:

- **synthesizeNTF**: se utiliza para generar una función de transeferencia de ruido (*Noise Transfer Function*). Este estructura es uno de los parámetros de entrada del modulador $\Sigma\Delta$. En esta función se especifican tanto la razón de sobremuestreo (OSR), como el orden del modulador.

- **simulateDSM**: con esta función se toma una señal de entrada y se obtiene a la salida la representación bitstream de la misma.

Para obtener un modulador adecuado para la aplicación, se utiliza una razón de sobremuestreo de 8. El mismo es de orden 2, con dos niveles de cuantización (esto porque la herramienta, al ser implementada en software, permite realizar la modulación con una mayor cantidad de niveles de cuantización). Al parámetro **opt** de **synthesizeNTF** se le asigna un valor de 1, para obtener una función de transferencia optimizada. En la figura 4.2 se muestra con detalle el diseño del modulador.

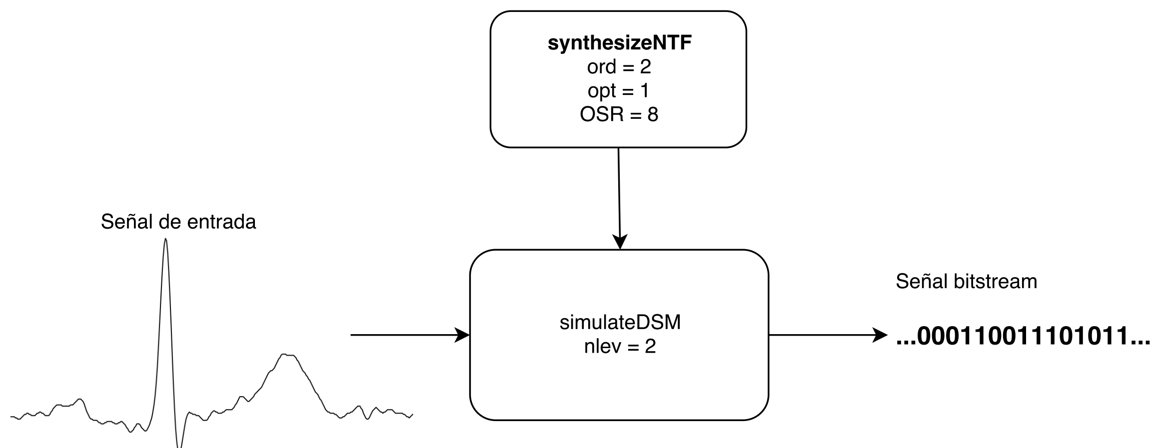


Figura 4.2: Modulación $\Sigma\Delta$ utilizando el paquete de herramientas *python sigma-delta*. Los bloques representan cada una de las funciones utilizadas. Se muestran las señales de entrada y salida, además de los valores usados para cada uno de los parámetros de entrada de las funciones.

La señal obtenida a la salida de esta etapa consiste en un arreglo de números, cuyos valores son 1 o 0. Esto indica que se debe realizar un paso extra para convertir este arreglo a una cadena que sea una representación binaria sobre la que se pueden aplicar operaciones bit por bit.

4.3 Representación *bitstream* de las señales

Como se menciona en la sección 4.2, la salida del modulador se debe convertir a una cadena hexadecimal cuyos valores correspondan bit por bit con los valores del arreglo obtenido. Para esto se utiliza otra herramienta de Python, llamada **bitstring**. Este módulo permite la creación y manipulación de datos en representación binaria.

La representación real de las señales se almacena en una estructura de datos denominada *BitArray*, propia del módulo bitstring. El procedimiento para almacenar los 1's y 0's en esta estructura es leer cada uno de los espacios del arreglo resultante de la modulación $\Sigma\Delta$, y dependiendo del valor (1 o 0), concatenar el valor lógico correspondiente al BitArray. Con esto se logra una cadena que puede ser almacenada en un archivo de texto para posteriormente ser leída por el programa en C++ que se ejecuta en el microprocesador Rocket. Otra alternativa es almacenar esta cadena estáticamente dentro de dicho programa, en caso de que no se cuenta con un sistema operativo al realizar las pruebas.

4.4 Algoritmo de Correlación Cruzada implementado en C++

El algoritmo de correlación implementado se implementa en una función en C++. Los parámetros de entrada para dicha función son dos arreglos de enteros sin signo, de 32 bits (**uint32_t**). Uno de estos arreglos almacena el patrón con el que se desea comparar la señal de prueba. Esta última se almacena en el otro arreglo de entrada. Previo a la llamada a esta función, la rutina principal almacena en el arreglo de la señal de prueba la porción de la señal bitstream correspondiente al paso actual n de ejecución.

En la figura 4.3 se observa una descripción gráfica del cálculo de la correlación. En dicha figura se muestra la correlación para una de las posiciones del arreglo de tipo **uint32_t**. Además se detalla como se ejecuta la función lógica XNOR $\odot$ para cada uno de los bits en la posición respectiva del arreglo.

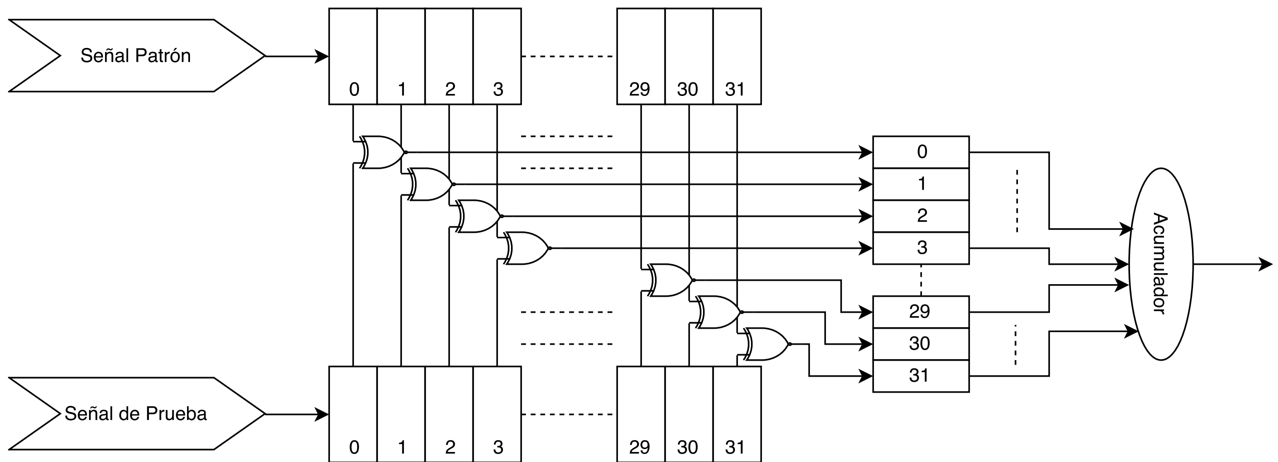


Figura 4.3: Correlación cruzada, ejecutada bit por bit en una de las posiciones del arreglo de tipo **uint32_t**.

La función de correlación utiliza dos variables auxiliares, que sirven para almacenar el resultado de la XNOR aplicada a cada valor, y para almacenar el número de bits en alto de dicho resultado (acumulador). Dado que las variables de entrada son arreglos de múltiples posiciones, se debe utilizar un ciclo para recorrerlas todas y aplicar la XNOR entre los valores respectivos de ambas señales.

Una vez que se aplica la XNOR, se utiliza otra función para contar el número de 1's resultante. Este se calcula para cada paso del ciclo, y se acumula hasta recorrer por completo los arreglos. Finalmente, el valor final del acumulador se devuelve al ámbito desde el que se llama la función. Esta salida se utiliza como entrada para el filtro paso bajo implementado.

La función que calcula el número de 1's dentro del resultado se encarga de calcular el peso de Hamming del mismo. Para este cálculo, se utiliza un algoritmo SWAR, que aplica operaciones "paralelas" sobre cada uno de los bits del registro utilizado. Esta función se muestra en el código fuente 4.1.

```
uint32_t numberOfSetBits(uint32_t i){
    i = i - ((i >> 1) & 0x55555555); // A
    i = (i & 0x33333333) + ((i >> 2) & 0x33333333); // B
    return (((i + (i >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24; // C
}
```

Código fuente 4.1: Función escrita en C++ para obtener el número de bits en alto de una variable de tipo entero sin signo de 32 bits. Implementa el algoritmo SWAR más conocido como *popcount*.

Para entender el funcionamiento de este algoritmo, se ha separado en 3 diferentes expresiones. En A se observa como primero se toma el número original y se le aplica un corrimiento bit por bit hacia la derecha. Al aplicar a este resultado una multiplicación bit por bit con el número hexadecimal 0x55555555 se obtienen los operandos mostrados en la tabla 4.1.

Tabla 4.1: Representación binaria de los operandos utilizados en la expresión A del algoritmo *popcount* SWAR, donde j = ((i >> 1) & 0x55555555).

i	i_0	i_1	i_2	i_3	...
j	0	i_0	0	i_2	...

Se puede afirmar que la relación

$$\langle i_0, i_1 \rangle \geq \langle 0, i_0 \rangle$$

es siempre cierta para cualquier valor de entrada i. Esta relación se cumple para todas las posiciones

$$\langle 2k, 2k+1 \rangle$$

de los registros j e i. Con esto se puede definir una transformada **T**, que describe el resultado de (i - j), como se observa en la tabla 4.2.

Tabla 4.2: Transformada utilizada en el algoritmo *popcount* SWAR para cada par de bits.

2k	2k+1	T(2k, 2k+1)
0	0	00
0	1	01
1	0	01
1	1	11

Se puede observar como el resultado de la transformada indica el número de bits en alto para cada una de las tuplas. Este patrón se puede expandir, como se hace en las expresiones B y C de la función mostrada en el código fuente 4.1, para obtener el peso de Hamming de cada secuencia de 4 y 8 bits, respectivamente.

4.5 Diseño de un filtro paso bajo digital de 3er orden

Como se explicó anteriormente, el modulador ΣΔ añade un ruido de cuantización a la señal muestreada. Convenientemente, este ruido se traslada a bandas de frecuencia altas. Sin embargo, para mantener la integridad de los resultados, después de calcular la correlación entre las señales es necesario realizar un filtrado para eliminar este ruido. El filtro debe ser un paso bajo, y se aplica a la suma de 1's producto de la correlación.

Para el diseño del filtro, i.e. el cálculo de los coeficientes, se utilizó el paquete de herramientas de procesamientos de señales de SciPy (**scipy.signal**). Este provee herramientas para el cálculo de filtros de respuesta infinita al impulso (IIR), utilizando un estilo similar al de MATLAB. En este caso se diseña un filtro Bessel. La elección se hace debido a que este tipo de filtros presenta la respuesta en fase más adecuada, pues posee un retraso de grupo y fase prácticamente planos. Esto es ventajoso para señales como las obtenidas en electrocardiogramas, que presenten diferentes componentes en diversas bandas de frecuencia. Con este tipo de filtro se preserva con mayor fidelidad la forma de la onda a la salida del mismo.

La función **scipy.signal.bessel** se utiliza para obtener los coeficientes en un formato *numerador/denominador*, con el objetivo de facilitar el diseño del filtro en C++. El filtro se calcula para un orden de 3, de tipo pasa bajo digital. La función mencionada también permite optimizar el cálculo de los coeficientes de manera que la respuesta en frecuencia se normalice siguiendo uno de 3 criterios disponibles. En este caso se utilizó la normalización por defecto.

Para la implementación del filtro, se utilizó la siguiente expresión en el domino de Z:

$$Y(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}} X(z) \tag{4.1}$$

Con esta se obtiene la siguiente ecuación de diferencias:

$$y[n] = \frac{1}{a_0}(b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] + b_3 x[n-3] - a_1 y[n-1] - a_2 y[n-2] - a_3 y[n-3]) \tag{4.2}$$

Esta expresión puede ser implementada de manera directa y simple en un programa escrito en C++. El diagrama de bloques del filtro implementado se presenta en la figura 4.4. Se observa que se implementa de la forma directa transpuesta II.

Como se indicó, la entrada del filtro es el acumulador, resultado de la función de correlación. A su vez, la salida del filtro sirve como entrada para dos funciones más: el filtro de media móvil y el comparador y detector.

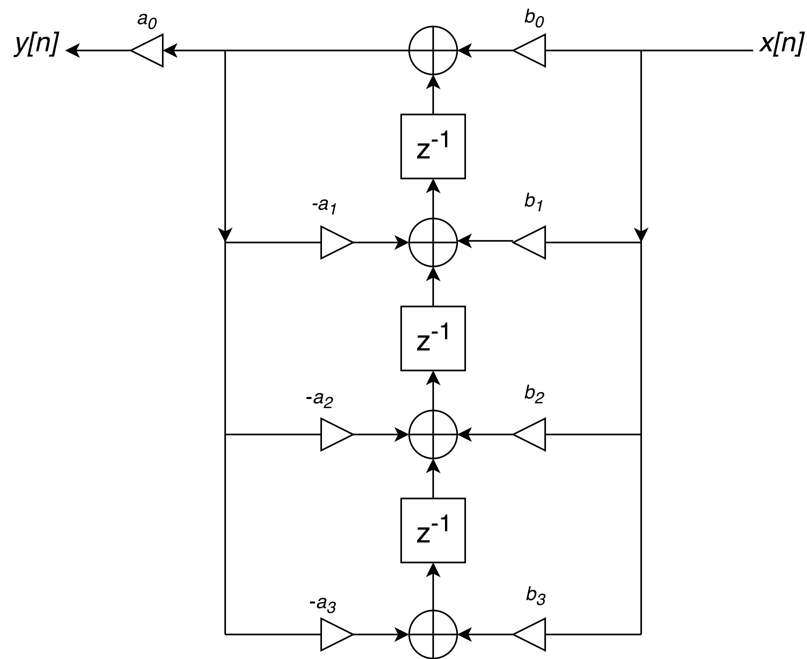


Figura 4.4: Filtro IIR de tercer orden implementado. El filtro implementa la ecuación de diferencias 4.2.

4.6 Filtro de Media Móvil

La justificación para el diseño de un filtro de media móvil se deriva de la necesidad de contar con un umbral adaptativo, con el cual comparar el nivel de la señal a la salida del filtro en el paso actual de la ejecución del programa.

El filtro de media móvil se implementa en una función de C++, llamada desde la rutina principal. Cuenta con un solo argumento de entrada, el cual es un arreglo de valores en coma flotante. En este arreglo se deben almacenar los N valores anteriores de la salida del filtro paso bajo. La función se encarga de sumar todos los valores almacenados en dicho arreglo, para luego dividir el resultado entre N, y con esto obtener el valor esperado. Además, para mejorar el rendimiento del comparador, se amplifica el resultado de la media multiplicándolo por una constante K. El valor de esta constante puede ser ajustado según los resultados experimentales.

En la figura 4.5 se puede observar un diagrama conceptual del algoritmo con el que se calcula la media móvil.

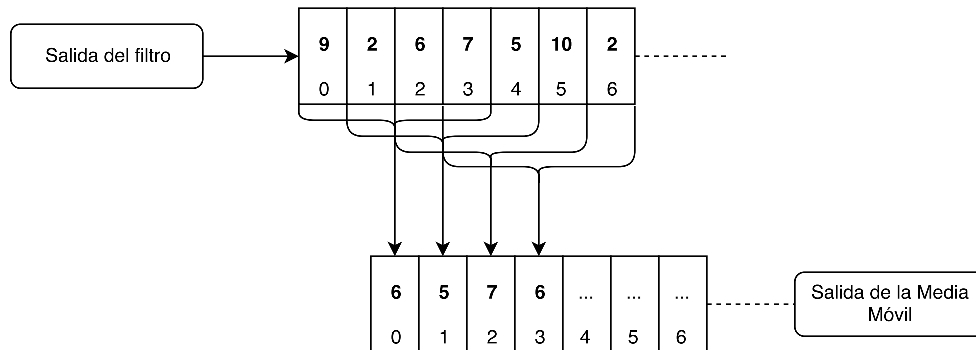


Figura 4.5: Diagrama conceptual del cálculo de la media móvil. En este ejemplo se utilizan 4 valores de salida del filtro paso bajo, pero este valor se puede parametrizar como N.

En el ejemplo mostrado se toman 4 valores anteriores de la salida del filtro paso bajo para hacer el cálculo del valor actual de la media móvil. Sin embargo este es un parámetro que se puede variar, para lograr mejores resultados sin afectar excesivamente el uso de memoria ni la velocidad de ejecución de esta función. La implementación de este algoritmo en C++ se muestra en los códigos fuente 4.2 y 4.3.

```
float movingAverage(float meanHist[18]){
  float result = 0;
  float K = 1.003;
  for(int j=0;j<18;j++){
    result += meanHist[j];
  }
  result = (K*result)/18.0;
  return result;
}
```

Código fuente 4.2: Función escrita en C++ para obtener la media móvil. El parámetro de entrada es una arreglo de 18 posiciones (para este ejemplo) de tipo flotante. En este se almacenan los últimos 18 valores de salida del Filtro Paso Bajo.

```
...
// Compute the movingAverage (amplified by K)
    for(int j=0; j<17;j++){
        filtHst[j] = filtHst[j+1];
    }
    filtHst[17] = filterOut;
    avrgOut = movingAverage(filtHst);
...
```

Código fuente 4.3: Sección de la rutina principal (escrita en C++) donde se actualiza el arreglo que contiene los N (18) últimos valores de salida del Filtro Paso Bajo. Con estos se calcula el valor actual de la Media Móvil.

Al calcular el valor correspondiente de la salida del filtro paso bajo, este debe añadirse al arreglo que almacena los valores de salida anteriores de dicha función. Para actualizar este historial, en la rutina principal se hace un corrimiento de los valores en el arreglo, de manera que se desecha el valor más antiguo, y se agrega el último valor calculado. Para las implementaciones mostradas se utilizan un valor de K y un número de salidas anteriores del Filtro Paso Bajo que pueden modificarse según sea necesario.

4.7 Comparador y Detector implementados

La última sección del algoritmo de correlación se encarga de detectar cuando existe un alto grado de similitud entre la señal de entrada y el patrón utilizado. Con esto se identifican los instantes en los que se da un evento en particular. El diseño conceptual se muestra en la figura 4.6.

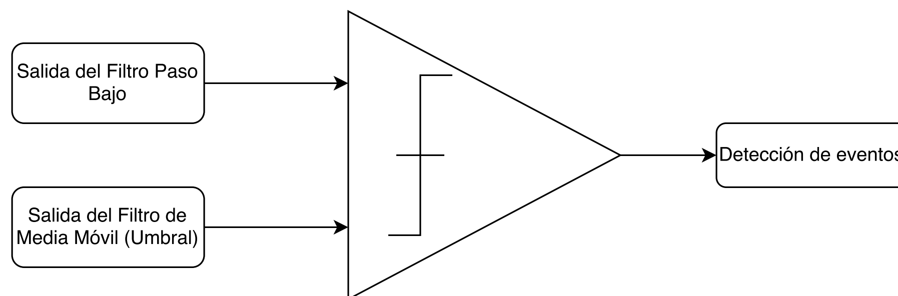


Figura 4.6: Diagrama conceptual del comparador. Las entradas o valores a comparar son la salida del Filtro Paso Bajo y la salida del Filtro de Media Móvil. Esta última está amplificada por una constante K.

Esta sección se implementa directamente en la rutina principal del programa. Siempre que el valor de salida del filtro paso bajo sea mayor que el umbral adaptativo, se puede emitir alguna señal que permite identificar cuando se da dicho evento. Por ejemplo, de ser necesario puede activarse una alarma, o simplemente se puede almacenar el instante en el que se da el evento para su análisis posterior.

Capítulo 5

Resultados y Análisis

En este capítulo se presentan los resultados más relevantes del desarrollo del proyecto. En primer lugar se presentan los relacionados con la implementación del sistema en chip. Se presta atención a las pruebas de simulación realizadas. Luego se presentan los resultados de la implementación del algoritmo de correlación bitstream, con sus diferentes secciones y componentes relevantes. Por último se muestran los resultados obtenidos al ejecutar este algoritmo en una tarjeta de desarrollo ZedBoard, utilizando una FPGA con una instancia del SoC RocketChip.

5.1 Sistema en Chip generado

Al generar los archivos *.v* con la descripción del sistema en chip, se pueden obtener las características del mismo inspeccionando el *device-tree* generado. De este se sabe que la frecuencia base del procesador es de 1 MHz, además de que el set de instrucciones implementado corresponde a **rv32im**.

Por otro lado, las memorias caché de primer nivel poseen 64 *sets*, cada uno con un tamaño de 64 Bytes. De esta manera, como se definió anteriormente, el tamaño de ambas memorias caché es de 4096 Bytes. Adicionalmente se tiene que el sistema en chip cuenta con 1 *hart* (*hardware thread*) Esto se debe a que dentro del sistema se añadió únicamente un *tile*, con un solo procesador.

El mapeo de memoria del sistema se muestra en la tabla 5.1.

Tabla 5.1: Mapeo de memoria resultante para el Sistema en Chip diseñado.

Nombre	Atributos	Dirección de inicio	Dirección final
debug-controller	RWX	0	1000
error-device	RW C	3000	4000
ROM	R X	10000	20000
CLINT	RW	2000000	2010000
interrupt-controller	RW	c000000	10000000
MMIO	RWX	60000000	80000000
DTIM	RWX	80000000	80004000

Los módulos **ROM**, **CLINT**, **MMIO** y **DTIM** se definen como:

- **ROM**: *Read-only memory*, memoria de arranque para el sistema.
- **CLINT**: *Core Local Interruptor*, maneja las interrupciones locales para el procesador. Estas interrupciones se emiten específicamente para un *hart*, lo que las diferencia de las interrupciones globales del sistema, manejadas por el *interrupt-controller*.
- **MMIO**: *Memory-Mapped Input Output*, sección de la memoria que se reserva para dispositivos de entrada y salida.
- **DTIM**: *Data Tightly-Integrated Memory*, memoria de datos.

5.2 Resultados de la simulación con programas de prueba

Una vez que se obtiene el SoC deseado, se deben aplicar pruebas para determinar si el procesador (y el sistema en general) funcionan de manera correcta. Para este fin, Rocket-Chip cuenta con un conjunto de programas de prueba que contienen rutinas basadas en la ejecución de una instrucción RISC V. Dichos programas tienen como fin determinar si el procesador generado puede ejecutar una instrucción particular. Con esto es posible determinar si efectivamente, el procesador generado soporta la ejecución de las instrucciones del set para el que fue diseñado, en este caso RISC V-IM de 32 bits.

A modo de ejemplo, se estudia la ejecución del programa *rv32ui-p-add*. Este contiene varias rutinas para ejecutar sumas simples, cuyos operandos son dos registros de propósito general. En el código fuente 5.1 se observa una porción del programa.

```
...
  #-------------------------------------------------------------
  # Arithmetic tests
  #-------------------------------------------------------------

  TEST_RR_OP( 2,  add, 0x00000000, 0x00000000, 0x00000000 );
  TEST_RR_OP( 3,  add, 0x00000002, 0x00000001, 0x00000001 );
  TEST_RR_OP( 4,  add, 0x0000000a, 0x00000003, 0x00000007 );
...
```

Código fuente 5.1: Porción del programa de prueba *rv32ui-p-add*.

El macro *TEST_RR_OP* se define como se muestra en el código fuente 5.2.

```
...
#define MASK_XLEN(x) ((x) & ((1 << (__riscv_xlen - 1) << 1) - 1))

#define TEST_CASE( testnum, testreg, correctval, code... ) \
test_ ## testnum: \
    code; \
    li  x29, MASK_XLEN(correctval); \
    li  TESTNUM, testnum; \
    bne testreg, x29, fail;

...
...

#define TEST_RR_OP( testnum, inst, result, val1, val2 ) \
    TEST_CASE( testnum, x30, result, \
      li  x1, MASK_XLEN(val1); \
      li  x2, MASK_XLEN(val2); \
      inst x30, x1, x2; \
    )
```

Código fuente 5.2: Definición de los macros utilizados en el programa de prueba *rv32ui-p-add*.

Como se observa, si el resultados es incorrecto, el programa hará un salto hacia la dirección indicada por la etiqueta *fail*. En caso contrario, seguirá con la ejecución de la siguiente prueba.

Para ejecutar este, o cualquier otro programa de prueba, se pueden utilizar tanto las herramientas de Synopsys (**VCS**) como el emulador incluido en las herramientas de RocketChip. Al realizar la simulación se obtienen dos archivos con los resultados: uno que incluye una descripción de las instrucciones y los valores de los registros en la etapa de *Write back*, cuya extensión es .out, y otro que registra el cambio en el valor de las señales del procesador (.vcd) denominado *Value Change Dump*.

Utilizando tanto el .dump, producto de la compilación del programa de prueba, como los archivos mencionados anteriormente, se puede inspeccionar la ejecución del programa, y determinar si el procesador realmente cumple con las especificaciones para las que fue diseñado. En el código fuente 5.3 se muestran las instrucciones en ensamblador que ejecuta el procesador para la prueba número 4, en la que se suman los valores 3 y 7, junto con las direcciones de memoria en las que se almacena dichas instrucciones.

```
...
8000012c <test_4>:
8000012c:  00300093          li   ra,3
80000130:  00700113          li   sp,7
80000134:  00208f33          add  t5,ra,sp
80000138:  00a00e93          li   t4,10
8000013c:  00400193          li   gp,4
80000140:  49df1e63          bne  t5,t4,800005dc <fail>
...
```

Código fuente 5.3: Información del programa posterior a su compilación. Se muestran tanto las instrucciones resultantes como las direcciones de memoria donde se almacenan.

Por otro lado, de la figura 5.1, que muestra un fragmento de la información contenida en un archivo .out (correspondiente a la prueba textitrv32ui-p-add), se puede extraer la siguiente información:

- Núcleo que ejecuta la instrucción (Cn). En este caso se tiene un único núcleo, el C0.
- El ciclo de reloj en el que la instrucción ocupa la etapa *Write back*.
- El primer número entre paréntesis cuadrados indica si hay o no una instrucción válida en la etapa *Write back*.
- El registro en el que se escribe, y el valor que se escribe en el mismo, además de los operandos de la instrucción.
- El código hexadecimal de la instrucción actual, acompañada de su expresión en ensamblador.

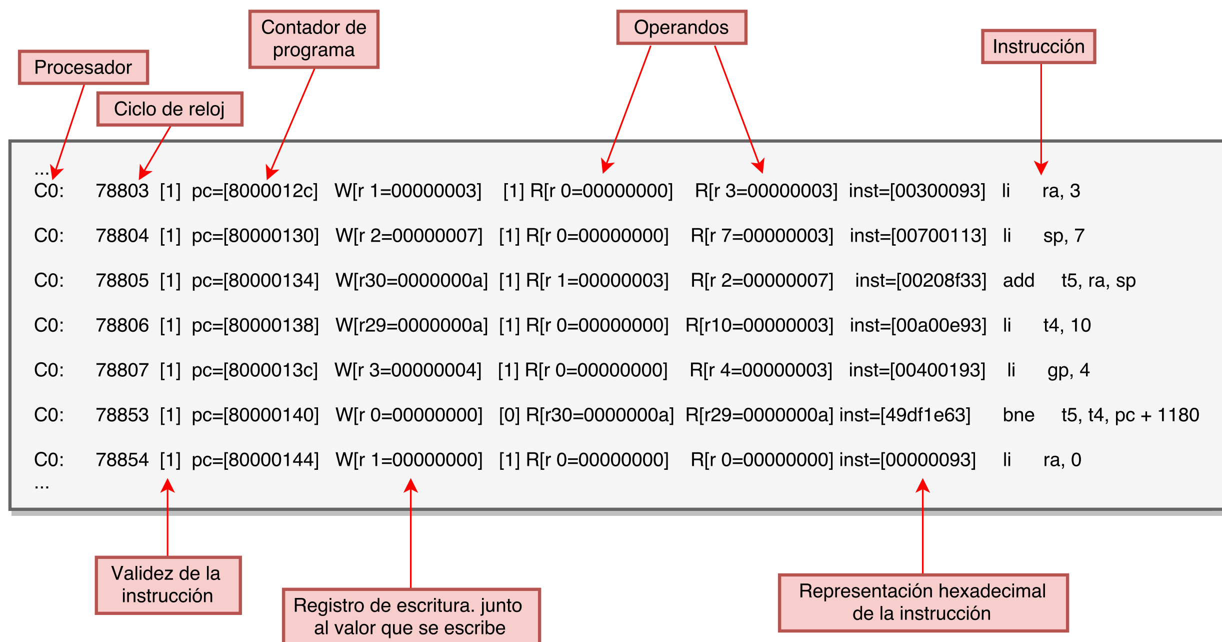


Figura 5.1: Información relevante de la etapa *Write Back* durante la ejecución de un programa de prueba. Se observa la ejecución de una de las sumas del programa, para probar la instrucción *add*.

De la última línea que se muestra en la figura 5.1 se puede inferir que el resultado obtenido en la suma es correcto. Esto porque, al realizar la comparación entre el resultado obtenido y el esperado (ciclo de reloj 78853) se sigue con la ejecución de la siguiente prueba (pc = [80000144]) y no se salta a la etiqueta señalada para las pruebas que fallan (en este caso pc + 1180).

Utilizando los archivos *.vcd* es posible estudiar el comportamiento de las señales de las diferentes etapas del *pipeline*, y con esto corroborar los resultados presentados anteriormente. En la figura 5.2 se muestran las señales de la etapa *Write back* del procesador.

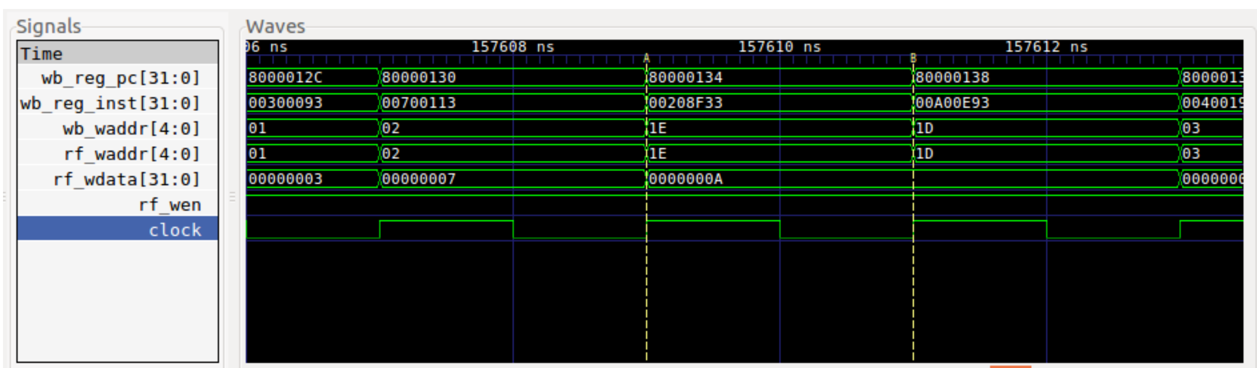


Figura 5.2: Comportamiento de algunas señales de la etapa *Write back* del procesador. Se observan el valor del contador de programa, el código hexadecimal de la instrucción ejecutada en ese ciclo y los operandos de la misma.

En la figura 5.2 se observan las señales del contador de programa y la instrucción presente en el respectivo ciclo de reloj. Entre los marcadores **A** y **B** se tiene la instrucción de suma mostrada en el código fuente 5.3 y la figura 5.1. Se observa que los valores del contador de programa y el código hexadecimal de la instrucción coinciden con los resultados mostrados anteriormente. Además, se observa que la dirección de escritura de esta etapa, coincide con la dirección de escritura del banco de registros (*Register File*, abreviado como *rf*), y tiene un valor de 0x1E. Este valor en decimal es igual a 30, lo que indica que el resultado de la suma se almacena en el registro x30 (también conocido por los alias r30 y t5). Este comportamiento es acorde a lo esperado según los códigos 5.2 y 5.3.

Seguidamente, en la figura 5.3 se muestran señales de la etapa de ejecución del procesador. Igual que en la figura 5.2, se muestran los valores del contador de programa y el código de la instrucción ejecutada. También se pueden observar las entradas y salidas de la ALU.

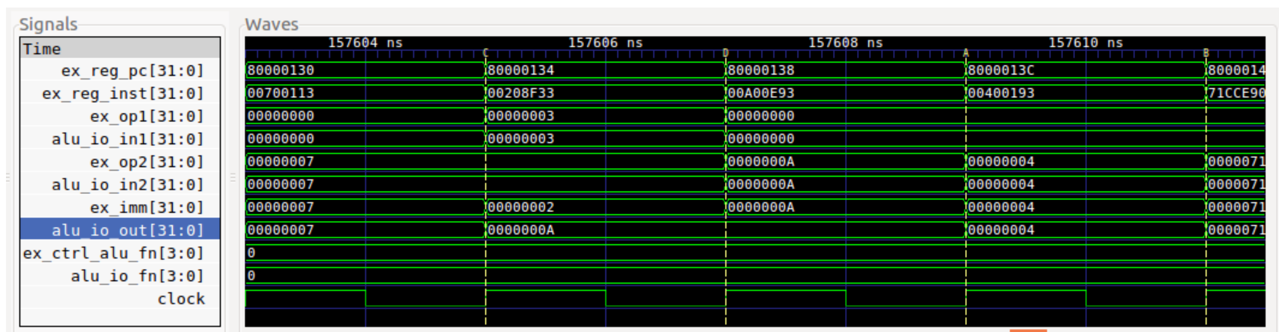


Figura 5.3: Comportamiento de algunas señales de la etapa *Exe* del procesador. Se observan el valor del contador de programa, el código hexadecimal de la instrucción ejecutada en ese ciclo y las entradas y salidas de la Unidad Aritmético-Lógica (ALU).

Entre los marcadores **C** y **D** de la figura 5.3 se observa la misma instrucción de suma estudiada anteriormente. Resulta importante resaltar que, como se observa, dos ciclos de reloj después, esta instrucción llega a la etapa de *Write back*. Esto se evidencia en la presencia de los mismos marcadores utilizados en la figura 5.2. De esto se puede deducir que las instrucciones aritméticas duran un ciclo de reloj en ejecutarse (por etapa). Esto también puede observarse en los archivos *.out*.

Utilizando este mismo procedimiento con resultados de diferentes programas de prueba puede determinarse que instrucciones pueden ejecutarse y cuáles no. Con esto es posible determinar si se cumplió el objetivo de generar un sistema con soporte para el conjunto de instrucciones planteado. En la tabla 5.2 se resume el resultado de la ejecución de los programas de prueba para el núcleo de instrucciones básico de RISC V (I). Además, en la tabla 5.3 se observan los resultados para la totalidad de las instrucciones de la extensión M de RISC V. Se observa que para este conjunto de instrucciones la ejecución de las pruebas también es exitosa.

Tabla 5.2: Resultados de la ejecución de programas de prueba en el sistema diseñado. Los resultados que se muestran son para el conjunto de instrucciones I de RISC V. Todas las pruebas terminan con resultados positivos.

Extensión RISC V	Categoría	Nombre	Resultado
RV32 I	Aritméticas	ADD	√
		ADDI	√
		SUB	√
	Lógicas	AND	√
		OR	√
		ORI	√
		XOR	√
	Loads	LB	√
		LW	√
	Stores	SB	√
	Shifts	SLL	√
	Comparación	SLT	√

Tabla 5.3: Resultados de la ejecución de programas de prueba en el sistema diseñado. Los resultados que se muestran son para la extensión M de RISC V. Para estas pruebas los resultados obtenidos son positivos.

Extensión RISC V	Categoría	Nombre	Resultado
RV32 M	Multiplicación	MUL	√
		MULH	√
		MULHU	√
		MULHSU	√
	División	DIV	√
		DIVU	√
	Residuo	REM	√
		REMU	√

Por último, en la tabla 5.4 se muestran los resultados de la ejecución de algunas instrucciones de las extensiones A, C y F. Se observa que en estos casos ninguna de las simulaciones tiene un resultado positivo. Esto concuerda con lo esperado, ya que al definir el sistema se eliminó la capacidad de ejecutar instrucciones de estos subconjuntos.

Tabla 5.4: Resultados de la ejecución de programas de prueba en el sistema diseñado. Los resultados que se muestran son para las extensiones A, C y F de RISC V. Dado que se eliminó el soporte para estas extensiones en el diseño del procesador, para estas pruebas los resultados obtenidos son negativos.

Extensión RISC V	Categoría	Nombre	Resultado
RV32 A	Suma	AMOADD_W	×
	Lógicas	AMOAND_W	×
	Min/Max	AMOMAX_W	×
		AMOMIN_W	×
RV32 F	Aritméticas	FDIV	×
	Multiplicación - Adición	FMADD	×
	Min/Max	FMAX	×
	Comparación	FCMP	×
RV32 C	General	-	×

En el caso de la extensión C, el programa de prueba aplicado difiere de los demás. En este caso en un mismo programa se ejecutan diversas instrucciones en ensamblador de dicha extensión, a diferencia de los otros casos, donde cada programa de prueba está diseñado para una sola instrucción. Por esto, en la tabla 5.4 sólo se muestra una línea general para esta extensión.

5.3 Obtención del patrón de comparación para señales ECG

El patrón utilizado en la correlación se muestra en la figura 5.4, en rojo. Esta señal corresponde a el promedio de tres ciclos diferentes obtenidos de las grabaciones etraídas de la base de datos utilizadas. Cada uno de los ciclos originales se representan en diferentes colores en la misma figura.

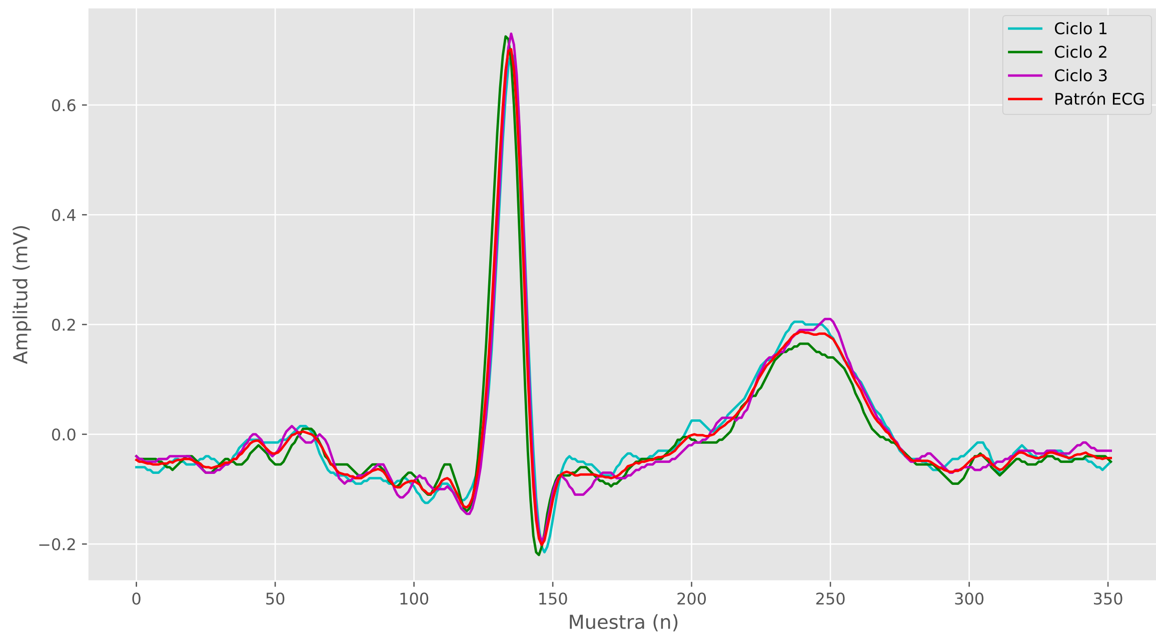


Figura 5.4: Obtención del promedio de 3 ciclos cardíacos para formar el patrón ECG utilizado en el algoritmo de correlación. Se muestran las 4 señales relevantes, de las cuales 3 corresponden a las señales extraídas de la base de datos.

Se puede observar como el patrón obtenido conserva las mismas características generales de las señales originales. Por ejemplo, tanto las ondas P y T, como el complejo QRS, se ven representadas en el patrón.

5.4 Representación *bitstream* de las señales utilizadas en las pruebas

Tanto los patrones utilizados, como las señales de prueba, deben representarse en formato bitstream. Con estas representaciones, como se ha venido mencionando, se alimenta el algoritmo de correlación. A modo de ejemplo se muestra en la figura 5.5 el patrón de las señales del electrocardiograma, junto a su prepresentación en bitstream.

En esta figura puede observarse como la densidad de 1's varía dependiendo del valor instantáneo de la señal modulado. Esta densidad de 1's es mayor cuando la señal alcanza un máximo local, por ejemplo, en el complejo QRS (entre el rango de las muestras 75 a 100).

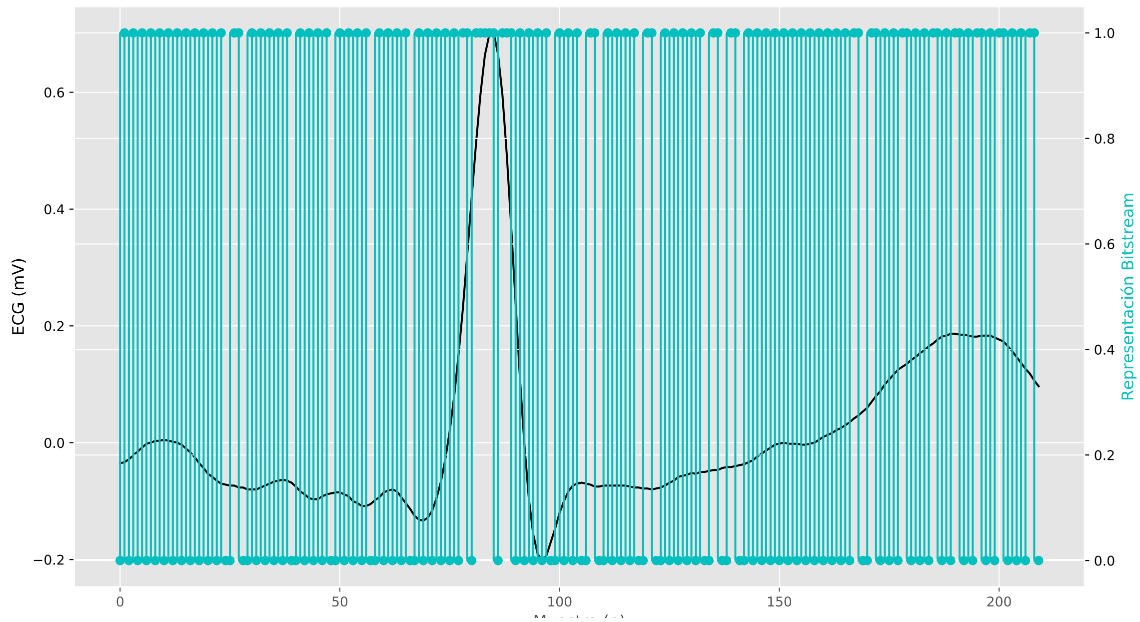


Figura 5.5: Representación *bitstream* de la señal ECG utilizada como patrón. Para mejorar la visualización de las señales, se muestra sólo una porción de las mismas.

El espectro de frecuencia de la señal ECG modulada se muestra en la figura 5.6. En dicha figura se puede apreciar como el ruido de cuantización se mueve a frecuencias altas, y como éste no aparece en el espectro de frecuencia de la señal previa a la etapa de modulación.

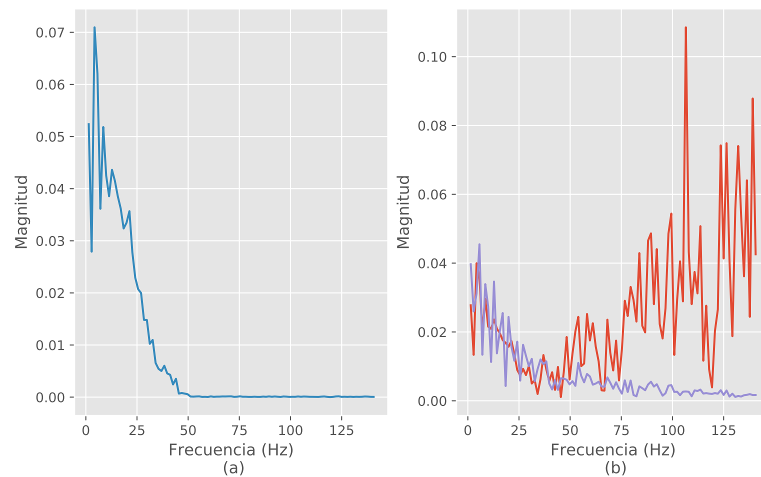


Figura 5.6: Espectro de la señal de un electrocardiograma. En (a) se observa el espectro de la señal sin realizarle ningún preprocesamiento. En (b) se muestra el espectro de la señal modulada ($\Sigma\Delta$) en rojo, junto al espectro de la misma señal luego de aplicar el filtro paso bajo diseñado. Además, se puede observar, en rojo, como el efecto de la modulación hace que el ruido de cuantización se mueva a frecuencias altas, y como este ruido se elimina al aplicar el filtro.

En la figura 5.6 (b) se puede observar como el filtro diseñado elimina efectivamente el ruido de cuantización en frecuencias altas, producto de la modulación ΣΔ. Sin embargo, aún en frecuencias superiores a 50 Hz se observan ciertas componentes con amplitud mayor a 0, lo que no se observa en la señal previa a la modulación (a). También se observa que la amplitud de la señal posterior al filtrado tiene una amplitud ligeramente menor, y cuenta con una distorsión mayor a la de la señal original. Pese a estos defectos, el filtro cumple con el objetivo de eliminar el ruido en altas frecuencias. Se debe hacer la aclaración de que el filtro paso bajo no se aplica directamente a la señal modulada, sino *a la correlación* entre la modulación ΣΔ de las señales patrón y de prueba. Sin embargo, en esta sección se aplicó a la señal modulada para demostrar su funcionamiento.

5.5 Respuesta en frecuencia del filtro paso bajo diseñado

En las figuras 5.7 y 5.8 se muestran la respuesta de amplitud y fase, respectivamente, del filtro paso bajo diseñado. Específicamente, en 5.7 se observa como la respuesta del filtro es plana para la banda de paso, y presenta una atenuación cada vez más pronunciada conforme la frecuencia crece, a partir de los 100 Hz. Además, para la frecuencia de corte (40 Hz) la atenuación es de -6.25 dB.

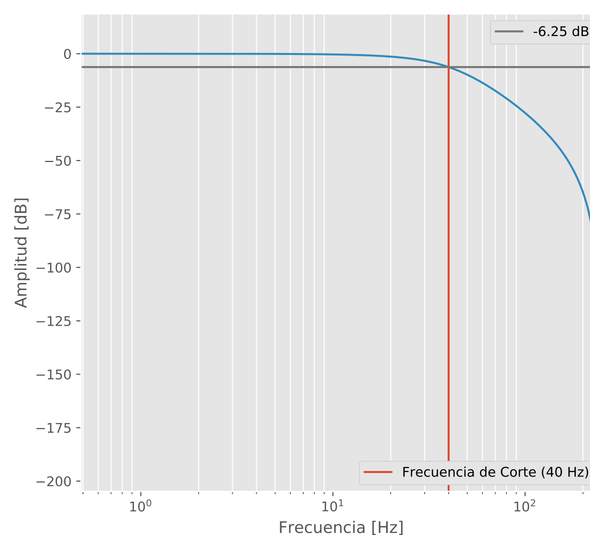


Figura 5.7: Respuesta en frecuencia del filtro paso bajo diseñado. Se observa la frecuencia de corte para la que el filtro fue diseñado, y el valor de la atenuación en dicha frecuencia.

De la respuesta en fase, en la figura 5.8, se observa como para frecuencias dentro de la banda de paso del filtro, el desfase es menor a 150°. Esto es conveniente pues en esta banda se encuentran los componentes de mayor peso en la señal ECG (como se puede observar en la figura 5.6). Sin embargo el desfase es más pronunciado para frecuencias altas.

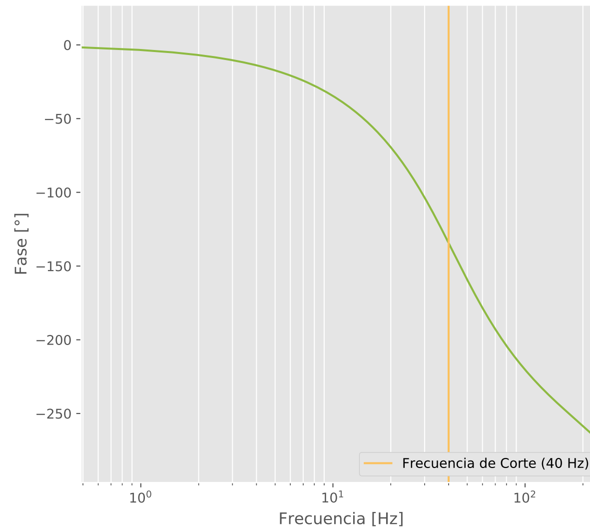


Figura 5.8: Respuesta en fase del filtro diseñado. Se puede observar la frecuencia de corte para la que se diseñó el filtro.

5.6 Correlación cruzada entre señales de prueba y distintos patrones

En esta sección se presentan algunos resultados de las pruebas ejecutadas en una FPGA Xilinx Zynq-7000, en una tarjeta de desarrollo Zedboard. En esta se programó una instancia del SoC RocketChip, y mediante comunicación **ssh** se extrajeron los resultados de la ejecución del algoritmo de correlación en dicho sistema. Estos resultados se almacenan en un archivo **csv** y se grafican utilizando un script de Python.

El primer resultado que se muestra corresponde a la correlación entre señales senoidales. En la figura 5.9 se muestran tanto el patrón utilizado para esta prueba, como la señal de entrada. Se observa que el patrón corresponde a un solo "evento", en otras palabras, está compuesto por un solo ciclo de la señal senoidal. La señal de entrada corresponde a una señal senoidal continua, cuyas características son las mismas de la señal de prueba. Estas características son:

- Frecuencia de la señal: $\mathbf{F} = 15Hz$.
- Tasa de muestreo: $\mathbf{F}_s$ = 500 sps (misma tasa de muestreo de las señales ECG de la base de datos).

- Amplitud = 0.5

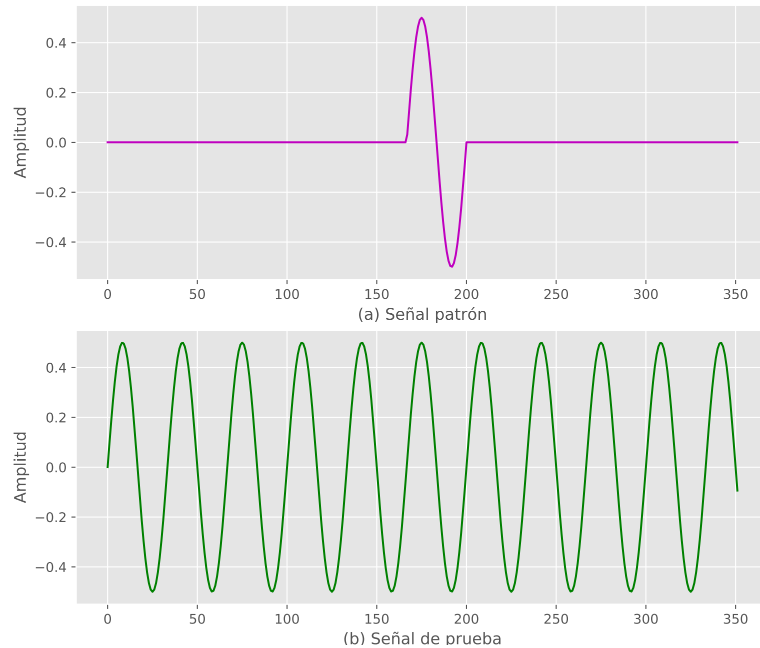


Figura 5.9: Patrón y señal de prueba utilizados para la correlación de una señal senoidal.

Seguidamente, en la figura 5.10 se muestra el resultado de la correlación. En este punto es importante indicar que la correlación teórica para todas las pruebas se obtuvo a partir de un script en Python, utilizando la función **numpy.correlate**. Se observa que la forma de las ondas, tanto para la correlación teórica como para la experimental, son congruentes. Sin embargo, sus valores difieren en gran medida. Esto se debe a que la correlación teórica muestra se determina utilizando valores representados en un formato decimal para las dos señales, mientras que los valores de la correlación experimental corresponden a la cantidad de 1's obtenida en cada ciclo de la ejecución del algoritmo.

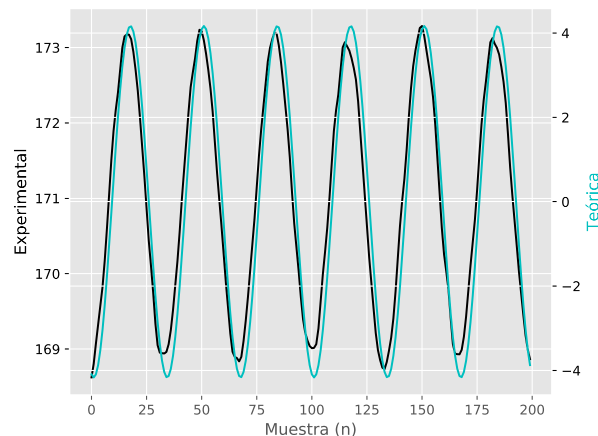


Figura 5.10: Correlación de una señal senoidal.

En la figura 5.11 se muestran el error absoluto para cada muestra de la correlación junto a el error absoluto promedio. Estos valores se calculan según las ecuaciones 5.1 y 5.2.

$$\text{Error absoluto} = |\text{Valor teórico} - \text{Valor experimental}| \tag{5.1}$$

$$\text{Error absoluto promedio} = \frac{\text{Error absoluto}}{\#\text{ de muestras}} \tag{5.2}$$

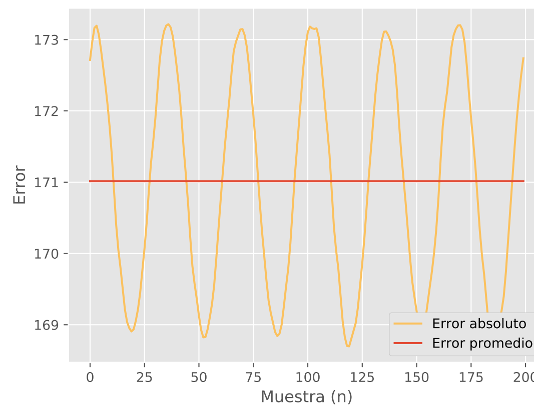


Figura 5.11: Error en la correlación de una señal senoidal.

Se puede observar como el error absoluto oscila alrededor del valor promedio, dependiendo de la muestra para la que se calcula. Además de las pruebas realizadas con señales conocidas, también se muestran los resultados para las señales tomadas de la base de datos de los electrocardiogramas. En la figura 5.12 se muestran las correlaciones teórica y experimental para uno de los casos estudiados. En este se realiza la correlación entre el patrón obtenido y el primer ciclo cardíaco presente en las grabaciones de la base de datos.

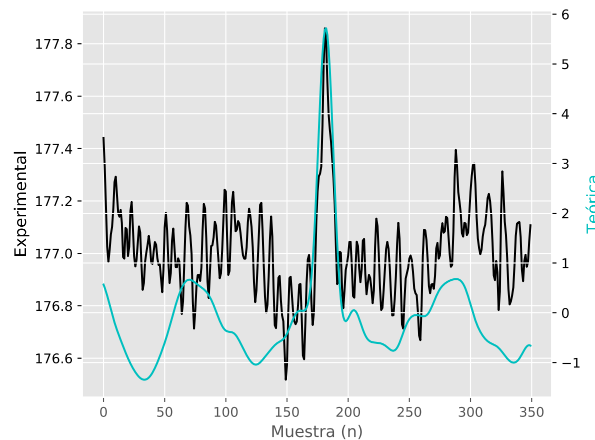


Figura 5.12: Resultados de la correlación de una señal ECG contra su respectivo patrón, caso 1.

Se observa como la correlación alcanza un máximo característico del momento donde las dos señales tienen la mayor similitud. Este máximo representa el instante en el que se da un evento, en este caso, cuando la señal medida y almacenada en el registro de prueba corresponde a un ciclo cardíaco.

Figura 5.13: Error en la correlación de una señal ECG contra su respectivo patrón, caso 1.

Por otro lado, en la figura 5.13 se muestra el error entre los valores teóricos y experimentales, utilizando las mismas definiciones mostradas anteriormente. Se observa como el error es mínimo para el momento en que las señales patrón y de prueba tienen mayor similitud entre sí. En la sección de apéndices se pueden encontrar gráficas con los resultados de otras pruebas aplicadas, utilizando diferentes señales de entrada.

5.7 Detección de eventos en señales ECG

En esta sección se muestran los resultados para la detección de eventos en las señales de electrocardiogramas. Los "eventos" que se buscan detectar corresponden a los ciclos cardíacos. Las pruebas buscan probar la sensibilidad y especificidad del sistema implementado, mediante la aplicación de distintas señales distintas del ECG.

La primer prueba realizada corresponde a la aplicación de las señales de una grabación que contiene 19 ciclos cardíacos, tomadas de la base de datos y que corresponden a grabaciones del mismo paciente del que se tomó el patrón mostrado anteriormente. Los resultados parciales se muestran en la figura 5.14. En esta se observa como se detectan efectivamente los 6 eventos presentes en esta porción de la ejecución de la prueba. Se observa también como el nivel de la salida del filtro de media móvil es mayor al de la salida del filtro paso bajo, como efecto de la amplificación dad por la constante K. Variando el valor de esta constante se puede modificar la sensibilidad del sistema.

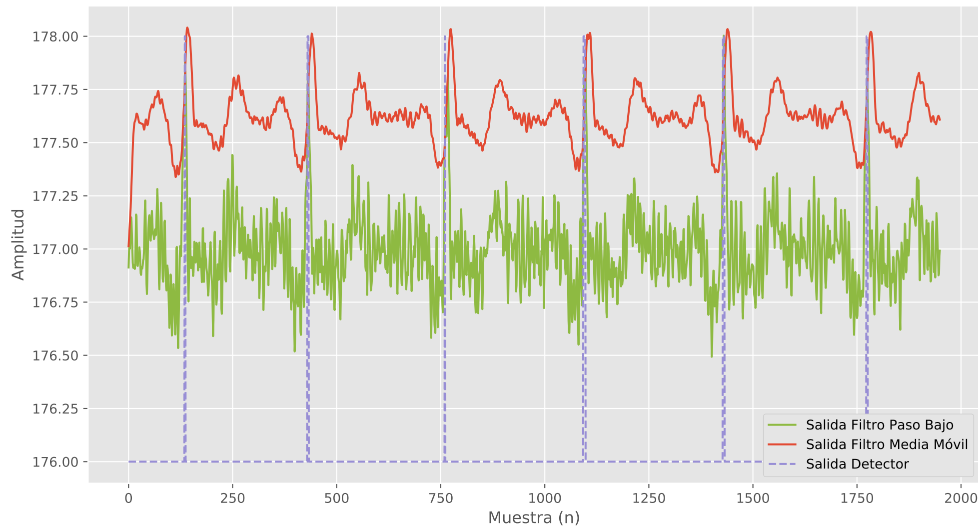


Figura 5.14: Salida de los filtros paso bajo y de media móvil, junto a la salida del detector, al utilizar señales ECG como entrada al algoritmo.

Seguidamente, se hizo una prueba utilizando como entrada una señal de ruido. Esta señal de ruido se obtuvo tomando muestras aleatorias de una distribución Gaussiana normal, con media de 0,5 y derivación estándar de 0,1. Los resultados para esta prueba se muestran en la figura 5.15.

Figura 5.15: Salida de los filtros paso bajo y de media móvil, junto a la salida del detector, al utilizar una señal de ruido para probar la eficacia del sistema.

Se observa como en este caso no se detecta ningún evento, lo que sugiere que ante señales ruidosas, con una distribución similar a la utilizada, el detector se comporta de la manera esperada.

Además se obtuvieron resultados para dos pruebas adicionales, utilizando como entrada señales senoidales de diferentes frecuencias. Las frecuencias utilizadas fueron 15 Hz y 200 Hz, ambas señales con una amplitud igual a 0,5. Los resultados para estas pruebas se encuentran en la sección de apéndices. Los resultados obtenidos indican que, cuando se aplican señales de baja frecuencia, i.e. señales cuya frecuencia está en la banda de paso del filtro paso bajo, la cantidad de falsos positivos asciende y el detector no funciona de manera correcta. Sin embargo, si se aplican señales con frecuencias mayores (e.g. 200 Hz), la cantidad de falsos positivos se reduce (en este caso, disminuye hasta 1). En este comportamiento se evidencia la actuación del filtro paso bajo, que disminuye el efecto de señales indeseadas en frecuencias mayores a la banda de interés. Estos resultados sugieren que es deseable contar con una etapa de discriminación posterior a la salida de los filtros, con la que se pueda discernir entre los eventos de interés y las señales que puedan causar falsos positivos.

Capítulo 6

Conclusiones y Recomendaciones

6.1 Conclusiones

Después de presentar los resultados, y realizar el análisis, es posible concluir que, en primer lugar, el Sistema en Chip generado cuenta con un procesador que tiene soporte para el conjunto de instrucciones RISC V que se planteó al inicio del proyecto. Específicamente, el procesador puede ejecutar instrucciones del conjunto central I y de la extensión M.

En segundo lugar, se logra reducir la complejidad del sistema, eliminando bloques funcionales como la unidad de punto flotante (**FPU**); utilizando una sola unidad de procesamiento dentro de un único *tile*; y reduciendo el tamaño de las memorias caché manteniendo el mismo en un valor factible.

Con respecto a la herramienta *RocketChip Generator*, utilizada para implementar la plataforma de procesamiento, se concluye que, a pesar de permitir un diseño parametrizable de un sistema en chip, la descripción RTL que genera es ilegible y no puede modificarse más allá de lo permitido por los parámetros expuestos en la herramienta.

Por otro lado, el algoritmo implementado es útil para realizar la detección de los eventos deseados, pero necesita una etapa adicional de clasificación y discriminación de las señales medidas para alcanzar un estado que resulte práctico en escenarios reales. La implementación de dicho algoritmo en un lenguaje como C++ permite utilizar los recursos disponibles de manera eficiente, manteniendo la funcionalidad esperada.

Al eliminar la unidad de punto flotante se agrega la carga computacional de la que ésta se encarga al software de la aplicación. Así, existe un balance entre la complejidad del hardware diseñado y la complejidad del software que éste debe ejecutar.

Las diferentes etapas del algoritmo de reconocimiento de patrones presentan los resultados que se plantearon en la fase de diseño. El filtro paso bajo tiene una respuesta adecuada, tanto en frecuencia como en amplitud. La modulación $\Sigma\Delta$ utilizada provee una representación *bitstream* adecuada de las señales utilizadas. Las señales de los electrocardiogramas utilizadas presentan la ventaja de no tener costo asociado a su uso, además de que cuentan con la documentación y calidad necesarias para realizar pruebas realistas.

6.2 Recomendaciones

Teniendo en cuenta las conclusiones presentadas, y los resultados obtenidos de las pruebas experimentales, se recomienda diseñar una etapa adicional de clasificación de los eventos detectados, para disminuir la cantidad de falsos positivos. Además, con una etapa adicional es posible extraer información importante sobre las señales estudiadas.

El diseño del filtro paso bajo puede ser objeto de varias modificaciones, con el fin de mejorar su rendimiento y características. Por un lado, es recomendable aumentar su orden, para cortar el ruido en bandas de frecuencia indeseadas con mayor efectividad. Además, se puede mejorar la respuesta en fase para que la integridad del resultado de la correlación se mantenga al pasar por el filtro.

Además, es posible adaptar el algoritmo para realizar la detección de eventos en señales de otros tipos. Una alternativa puede ser la detección de ataques epilépticos.

Por último, se propone realizar un estudio de la ejecución del algoritmo variando los parámetros de los diferentes componentes del sistema, y tomando otras métricas relevantes como el tiempo de ejecución del programa. Por ejemplo, pueden alterarse los valores de los parámetros principales de las memorias caché, para determinar la configuración que ofrece los mejores resultados.

Bibliografía

[1] Multi-functional Chip Development, Make the Innovation of the Next Generation of Implants. http://www.medicinetechnews.com/multi-functional-chip-development-make-the-innovation-of-the-next-generation-of-implants/. Fecha de acceso: 1/4/2017.

[2] RISC V Cores. https://riscv.org/risc-v-cores/. Fecha de acceso: 10/8/2017.

[3] Rocket Core Overview. http://www.lowrisc.org/docs/tagged-memory-v0.1/rocket-core/. Fecha de acceso: 18/8/2017.

[4] H. Ando, K. Takizawa, T. Yoshida, K. Matsushita, M. Hirata, and T. Suzuki. Wireless multichannel neural recording with a 128-mbps uwb transmitter for an implantable brain-machine interfaces. *IEEE Transactions on Biomedical Circuits and Systems*, 10(6):1068–1078, Dec 2016.

[5] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The *Rocket Chip* Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[6] E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb 2013.

[7] H. Cao, V. Landge, U. Tata, Y. S. Seo, S. Rao, S. J. Tang, H. F. Tibbals, S. Spechler, and J. C. Chiao. An implantable, batteryless, and wireless capsule with integrated impedance and ph sensors for gastroesophageal reflux monitoring. *IEEE Transactions on Biomedical Engineering*, 59(11):3131–3139, Nov 2012.

[8] Josué Andrés Mora Castro. Implementación de algoritmo para calcular la correlación bitstream entre dos señales para detectar sonidos de disparos y motosierras en el bosque. Master's thesis, Escuela de Ingeniería Electrónica, ITCR, 2016.

[9] Y. P. Chen, D. Jeon, Y. Lee, Y. Kim, Z. Foo, I. Lee, N. B. Langhals, G. Kruger, H. Oral, O. Berenfeld, Z. Zhang, D. Blaauw, and D. Sylvester. An injectable 64 nw ecg mixed-signal soc in 65 nm for arrhythmia monitoring. *IEEE Journal of Solid-State Circuits*, 50(1):375–390, Jan 2015.

[10] E. Y. Chow, A. L. Chlebowski, S. Chakraborty, W. J. Chappell, and P. P. Irazoqui. Fully wireless implantable cardiovascular pressure monitor integrated with a medical stent. *IEEE Transactions on Biomedical Engineering*, 57(6):1487–1496, June 2010.

[11] Henry Cook. *Productive Design of Extensible On-Chip Memory Hierarchies*. PhD thesis, EECS Department, University of California, Berkeley, May 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-89.html.

[12] A. DeHennis, S. Getzlaff, D. Grice, and M. Mailand. An nfc-enabled cmos ic for a wireless fully implantable glucose sensor. *IEEE Journal of Biomedical and Health Informatics*, 20(1):18–28, Jan 2016.

[13] Ary L. Goldberger, Luis A. N. Amaral, Leon Glass, Jeffrey M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. Physiobank, physiotoolkit, and physionet. *Circulation*, 101(23):e215–e220, 2000. URL http://circ.ahajournals.org/content/101/23/e215.

[14] L. Polonsky J., Wasilewski. An introduction to ecg interpretation. In Witold Pedrycz Adam Gacek, editor, *ECG Signal Processing, Classification and Interpretation. A Comprehensive Framework of Computational Intelligence*, volume 1 of *1*, chapter 1, pages 1–20. Springer, 1 edition, 2012.

[15] Ben Keller. *CS 250 Laboratory 2 (Version 091713)*. University of California-Berkeley, EECS Department, 2013.

[16] S. B. Lee, B. Lee, M. Kiani, B. Mahmoudi, R. Gross, and M. Ghovanloo. An inductively-powered wireless neural recording system with a charge sampling analog front-end. *IEEE Sensors Journal*, 16(2):475–484, Jan 2016.

[17] O. E. Liseth, D. Mo, H. A. Hjortland, T. S. “. Lande, and D. T. Wisland. Power-efficient cross-correlation beat detection in electrocardiogram analysis using bitstreams. *IEEE Transactions on Biomedical Circuits and Systems*, 4(6):419–425, Dec 2010.

[18] Tatiana Lugovaya. Biometric human identification based on electrocardiogram. Master's thesis, Faculty of Computing Technologies and Informatics, Electrotechnical University, LETI., June 2005.

[19] Gabriel Loría Marín. Diseño de un circuito integrado digital como detector de disparos de armas de fuego y motosierras, basado en correlación bitstream. Master's thesis, Escuela de Ingeniería Electrónica, ITCR, 2016.

[20] Rolando Adrián Moraga Mora. Sistema de detección de patrones acústicos de disparos de armas de fuego y motosierras en bosques tropicales basado en un algoritmo de correlación de bitstream. Master's thesis, Escuela de Ingeniería Electrónica, ITCR, 2016.

[21] Tim Newsome. RISC V External Debug Support. Version 0.13. Technical report, SiFive, July 2017.

[22] F. N. M. Pirchio, P. Julian, P. S. Mandolesi, and A. Chacon-Rodriguez. An adaptive cross-correlation derivative algorithm for ultra-low power time delay measurement. In *2007 IEEE International Symposium on Circuits and Systems*, pages 4016–4019, May 2007.

[23] SiFive. Sifive tilelink specification, version 1.7. Technical report, SiFive, Inc., 8 2017. Pre-release draft.

[24] William Stallings. *Computer Organization and Architecture. Designing for Performance (Ninth Edition)*. Pearson/Prentice Hall, 9 edition, 2013.

[25] A. Burdett T. S. Lande, T. G. Constandinou and C. Toumazou. Running crosscorrelation using bitstream processing. In *Electronic Letters*, volume 43, October 2007.

[26] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanović. The risc-v instruction set manual volume ii: Privileged architecture version 1.10. Technical report, EECS Department, University of California, Berkeley, May 2017.

[27] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2. Technical report, EECS Department, University of California, Berkeley, May 2017.

[28] Catalina Tobón Zuluaga. *Modelización y evaluación de factores que favorecen las arritmias auriculares y su tratamiento mediante técnicas quirúrgicas. Estudio de simulación.* PhD thesis, Departamento de Ingeniería Electrónica, Universidad Politécnica de Valencia., Valencia, 2010.

Apéndice A

Resultados Adicionales

A.1 Resultado de la correlación utilizando diferentes señales de prueba

Primeramente se muestran,en las figuras A.1 y A.3, los resultados de la correlación al utilizar diferentes señales de prueba y de patrón. En las figuras se muestran tanto los resultados experimentales como los teóricos, obtenidos utilizando Python. Para cada uno de estos estudios se calcularon los errores absoluto y promedio, que se muestran, respectivamente, en las figuras A.2 y A.4.

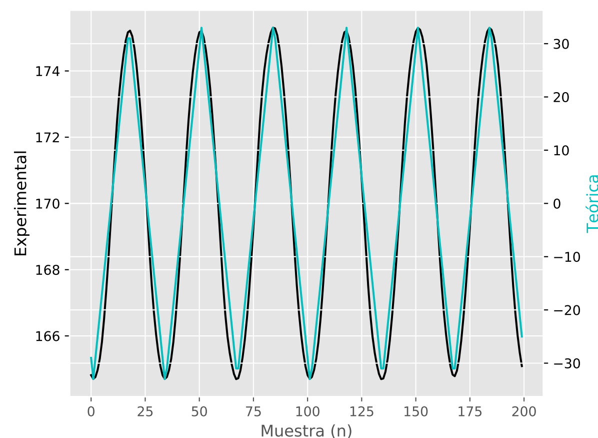


Figura A.1: Correlación de una señal cuadrada.

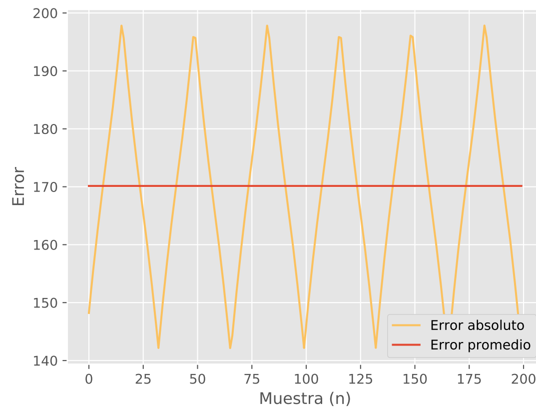


Figura A.2: Error en la correlación de una señal cuadrada.

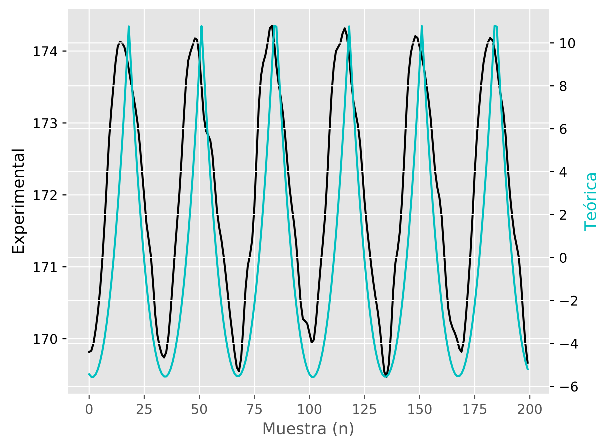


Figura A.3: Correlación de una señal diente de sierra.

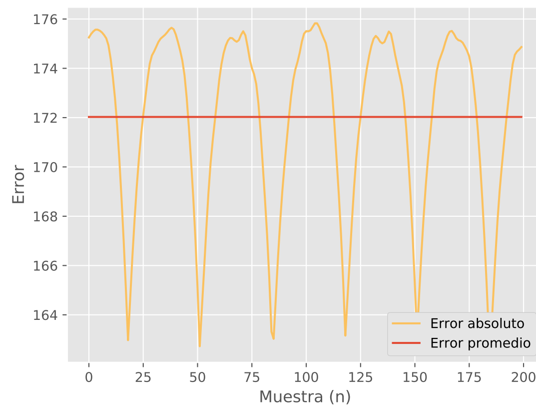


Figura A.4: Error en la correlación de una señal diente de sierra.

Los patrones y señales de entrada de la correlación utilizadas en las pruebas anteriores se muestran en las figuras A.5 y A.6, respectivamente.

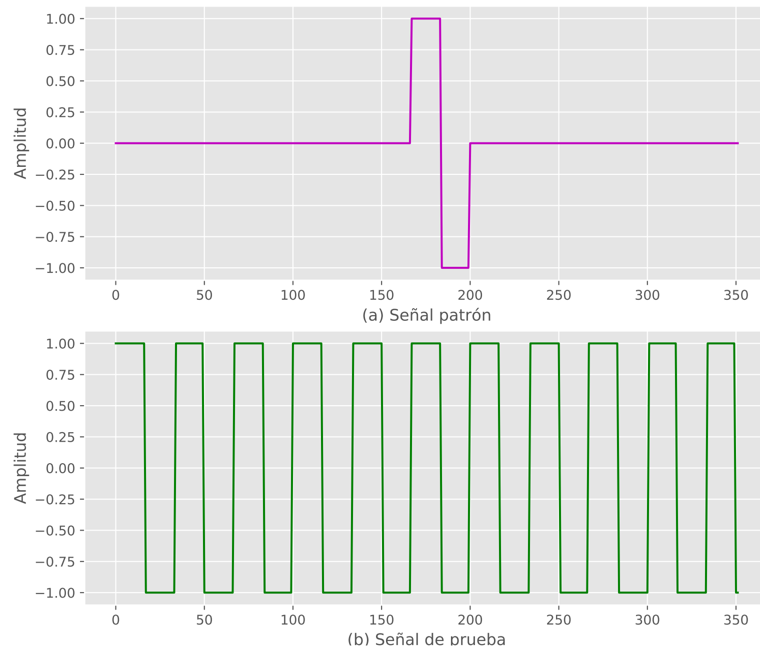


Figura A.5: Patrón y señal de prueba utilizados para la correlación de una señal cuadrada.

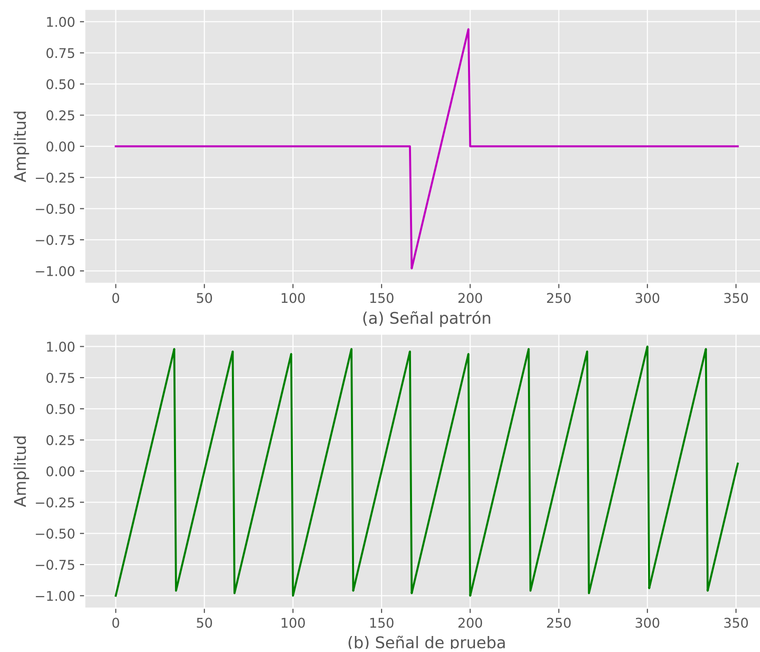


Figura A.6: Patrón y señal de prueba utilizados para la correlación de una señal diente de sierra.

También se realizaron pruebas utilizando señales de entrada del electrocardiograma obtenido, utilizando como patrón la señal mostrada en 5.4. Los resultados experimentales y teóricos se pueden observar en las figuras A.7 y A.9.

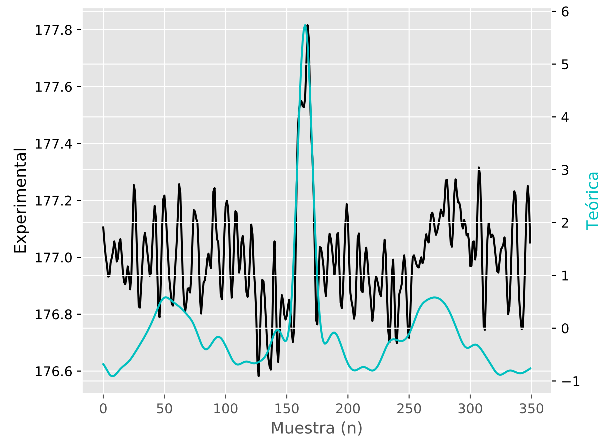


Figura A.7: Resultados de la correlación de una señal ECG contra su respectivo patrón, caso 2.

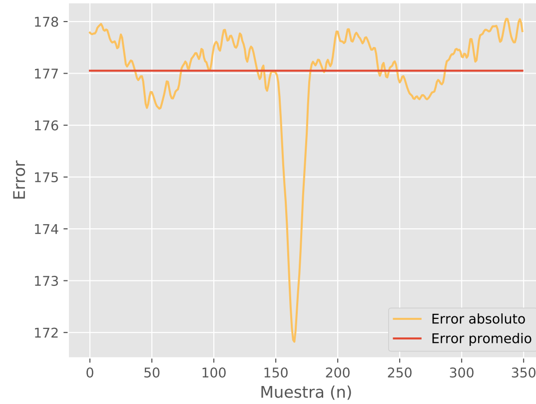


Figura A.8: Error en la correlación de una señal ECG contra su respectivo patrón, caso 2.

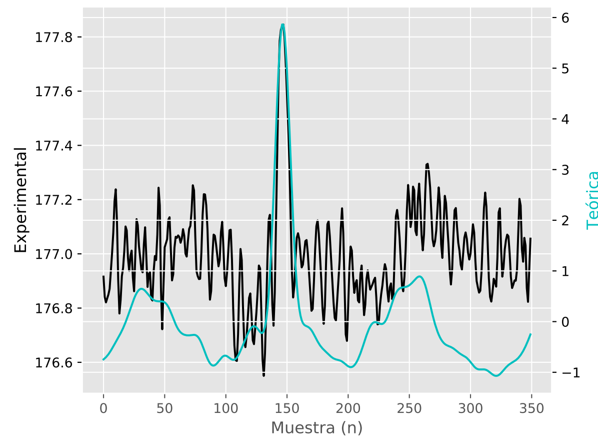


Figura A.9: Resultados de la correlación de una señal ECG contra su respectivo patrón, caso 3.

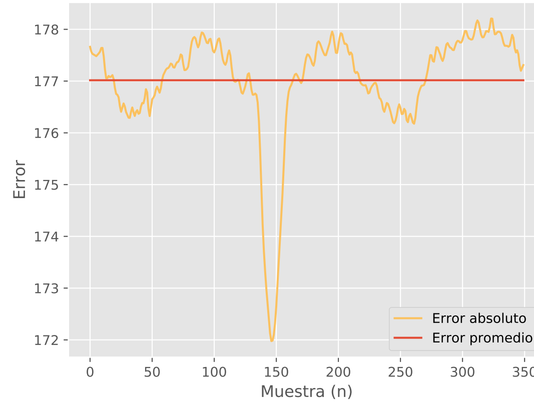


Figura A.10: Error en la correlación de una señal ECG contra su respectivo patrón, caso 3.

Acompañando los resultados de la correlación, el error para cada uno de los casos estudiados se muestra en las figuras A.8 y A.10.

A.2 Resultados de la detección de eventos ante señales senoidales de diferentes frecuencias

En la figura A.11 se observan las salidas del detector y los filtros implementados, al utilizar como señal de entrada una onda senoidal de frecuencia baja. Se observa como hay una cantidad alta de falsos positivos.

Por otra parte, en la figura A.12 se pueden ver las señales de las mismas salidas, al utilizar como señal de prueba una onda senoidal de frecuencia mayor. Sin embargo, para este caso la cantidad de falsos positivos disminuye hasta un valor de 1. Aquí el detector se comporta de la manera esperada y logra discernir entre las señales de interés y señales de prueba distintas.

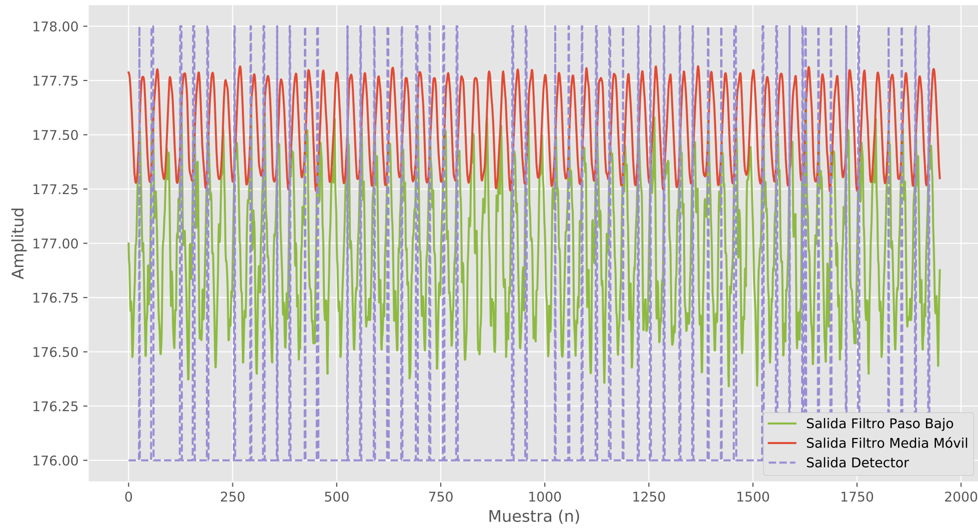


Figura A.11: Salida de los filtros paso bajo y de media móvil, junto a la salida del detector, al utilizar una señal senoidal con frecuencia $f = 15$ Hz.

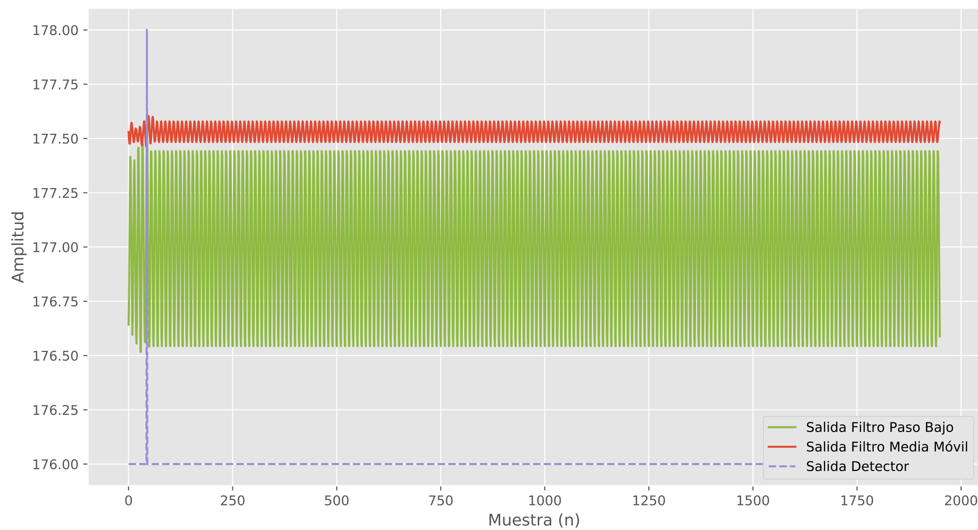


Figura A.12: Salida de los filtros paso bajo y de media móvil, junto a la salida del detector, al utilizar una señal senoidal con frecuencia $f = 200$ Hz.

Índice alfabético

asm-tests, 22
assembly tests, 22
automaticidad, 18

benchmarks, 22
BitArray, 28
bitstream, 4, 15, 25

ciclo cardíaco, 19
conjunto, 10

etiqueta, 10

fibras de Purkinje, 18

hart, 35
haz de His, 18

intervalo RR, 19

mapeo asociativo de conjuntos, 10

nodo atrio ventricular, 18
nodo sinoauricular, 17

palabra, 10