Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Electrónica

Licenciatura en Electrónica

El consumo de recursos y la precisión en dispositivos programables: caso de matrices mal condicionadas.

TESIS

Que para obtener el título de:
Licenciado en Electrónica
PRESENTA:

C. Claudio Guadalupe Cruz Mendoza

Directores de Tesis: **Dra. María Monserrat Morín Castillo (FCE - BUAP)**
Dr. José Rubén Conde Sánchez (FCFM - BUAP)
Dr. José Jacobo Oliveros Oliveros (FCFM - BUAP)

Junio 2022. Puebla, Pue. Méx.

Resumen

En diferentes áreas de las matemáticas y de la instrumentación electrónica existen campos específicos de investigación sobre el planteamiento y uso las matrices denominadas mal condicionadas, las cuales aparecen en sistemas de ecuaciones lineales cuya solución es muy sensible a errores en el lado derecho del sistema, es decir, pequeños errores en el lado derecho del sistema pueden producir grandes cambios en la solución del sistema. Por otro lado, en el desarrollo de aplicaciones en sistemas embebidos es cada vez más común la utilización de hardware especializado para realizar tareas específicas, y una de estas tendencias es el uso de *System on Chip* (SoC) basados en procesadores en conjunto con una sección de lógica programable, lo cual proporciona una solución extensible y flexible. En esta tesis se presenta el desarrollo, análisis y evaluación de una arquitectura de alto desempeño basado en un SoC que plantea y muestra el desempeño ante matrices consideradas mal condicionadas de tamaño 3×3 , para esto se emplea la estrategia del cálculo de la matriz inversa utilizando el método de la adjunta y la multiplicación matricial. Para la creación de la arquitectura se utilizó la Síntesis de Alto Nivel, la cual permite obtener componentes implementados en lógica programable a partir de una función en Lenguaje C. Como parte del sistema, la evaluación es realizada con el *framework* PYNQ empleando el lenguaje Python. El análisis y resultados de la arquitectura son mostrados a través de ejemplos que muestran la operatividad de las matrices mal condicionadas, también se muestran resultados de la arquitectura en términos de los recursos lógicos utilizados en la implementación.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En diversas áreas, tales como la física, ciencias de la salud, control, automatización y optimización por nombrar algunas, requieren el uso de matrices como herramienta principal en la solución de sistemas de ecuaciones. Sin embargo, es importante mencionar la existencia de las matrices mal condicionadas, lo cual deriva en problemas de inestabilidad numérica respecto a los errores que puedan producir al buscarse la solución del sistema. Una característica de las matrices mal condicionadas es que pequeños variaciones numéricas en el lado derecho del sistema pueden producir grandes cambios en la solución del sistema.

Para ilustrar los efectos generados por una matriz mal condicionada, se propone realizar el cálculo de la inversa de una matriz utilizando el método de la adjunta, en donde se busca mostrar los errores de cálculo causados por el aumento o reducción de la precisión del número de dígitos.

Referente a los costos en hardware, se implementarán las operaciones de la inversa de una matriz usando el método de la adjunta y la multiplicación de dos matrices, ambas operaciones tomarán en cuenta matrices de 3×3 como entrada. Se harán dos versiones de las operaciones; empleando datos tipo *float* y *double*.

En la construcción del sistema se eligió una metodología que contempla tanto el diseño de hardware como el de software, la cual consta de seis etapas: 1.- requerimientos funcionales, 2.- especificación detallada, 3.- división de hardware/software, 4.- desarrollo de hardware y pruebas, 5.- desarrollo de software y pruebas, 6.- integración y resultados.

Para realizar estos módulos se usará la herramienta Síntesis de Alto Nivel (HLS), la cual se ofrece como alternativa en el diseño, derivado al aumento de la complejidad de los circuitos. Dicha síntesis permite generar un módulo en Lenguaje de Descripción de Hardware, ya sea en VHDL o Verilog a partir de una funcionalidad especificada en lenguaje C, C++ o SystemC.

La plataforma que se usará es el SoC[1] Zynq del fabricante Xilinx. Contiene un procesador ARM A9 correspondiente a la sección del Sistema de Procesamiento (PS) y

[1]System on Chip: Se define como el bloque funcional que tiene la mayor parte de la funcionalidad del sistema, a excepción de algunos bloques de interfaz[4].

una sección de Lógica Programable (PL), la cual tiene una estructura equivalente a un FPGA[2]. Se propone implementar los módulos en la sección de PL y utilizar el PS con la finalidad de permitir el uso del *framework* Productividad de Python para Zynq, este *framework* incorpora la infraestructura de cuadernos Jupyter de código abierto para ejecutar un núcleo de Python interactivo (IPython) y un servidor web directamente en el procesador ARM del dispositivo Zynq. El servidor web facilita el acceso al núcleo a través de un conjunto de herramientas basadas en el navegador que proporcionan un panel de control, un terminal bash, editores de código y cuadernos Jupyter[27]. Una de las principales características del *framework* es representar los circuitos de la lógica programable como librerías de hardware llamadas *Overlays*, esto esta ideado para que se pueda seleccionar el *Overlay* que mejor se adapte a la aplicación, en el contexto del trabajo, permitirá cambiar entre ambos diseños (*float* y *double*) de una manera rápida y sencilla. También se usarán librerías específicas como MMIO, el cual permite acceder a direcciones en el mapa de memoria del sistema, en particular se pueden acceder a los registros y al espacio de direcciones de los módulos implementados en la sección de PL[21], esto permitirá crear los *drivers* para controlar a los módulos inversa y multiplicación.

1.1. Justificación

Las matrices mal condicionadas son un tema de interés para diferentes áreas de estudio, como puede ser control, automatización, etc., ya que provoca variaciones importantes en la solución del sistema cuando el lado derecho del sistema de ecuaciones lineales tiene pequeñas variaciones. En particular, tomando en cuenta que los sistemas computacionales hacen una representación de números reales como números de precisión finita (discretización y truncamiento), es relevante mostrar como afecta este procedimiento cuando se tiene una matriz mal condicionada.

Este trabajo busca mostrar los efectos de estos errores en los cálculos computacionales y sus costos en hardware. Por lo cual se plantea diseñar un sistema implementado en lógica programable que realice las operaciones de inversa (método de la adjunta) y multiplicación de matrices de $3x3$ utilizando datos tipo *float* y *double*.

1.2. Objetivos

1.2.1. Objetivo General

Determinar la relación entre el consumo de recursos y la precisión en dispositivos programables para sistemas de ecuaciones lineales algebraicas con matrices mal condicionadas de 3×3.

[2]Field Programmable Gate Array: es una matriz de puertas reconfigurables. Se construyen como una matriz de elementos lógicos configurables (LEs), también denominados bloques lógicos configurables (CLB)[3].

1.2.2. Objetivos Particulares

1. Crear módulos implementados en lógica programable para calcular la inversa de una matriz de 3×3.

2. Crear módulos implementados en lógica programable para calcular la multiplicación de matrices de 3×3.

3. Verificar el funcionamiento de los módulos creando una aplicación en Python utilizando el *framework Python productivity for Zynq* (PYNQ).

4. Realizar las operaciones matriciales con truncamiento para analizar el error generado.

1.3. Metodología

La metodología propuesta, para el logro de los objetivos e implementación, es desglosada en seis fases principales, ver Figura 1.1; está organizada tanto para hardware como el software, se muestra los niveles de abstracción del sistema, desde el sistema completo hasta los componentes individuales, su integración y pruebas del sistema [18], por lo que, proporciona una idea general de las tareas a realizar en el desarrollo de la presente Tesis.

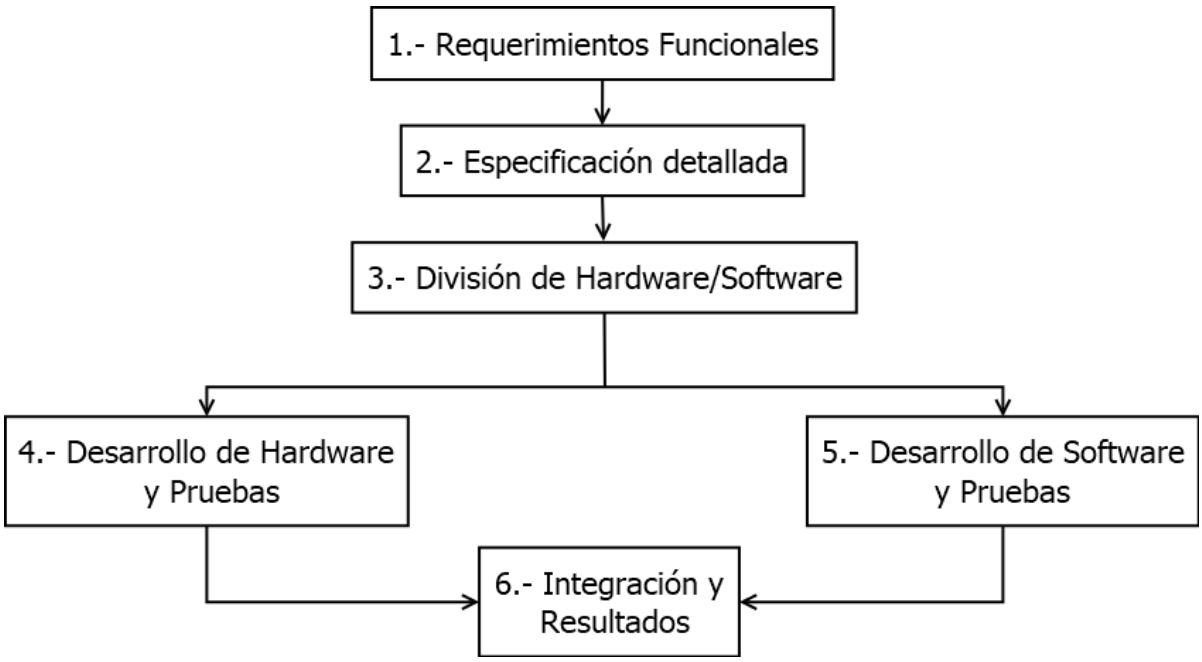


Figura 1.1: Flujo de diseño típico en SoC [10].

1. Requerimientos Funcionales: En esta fase inicial del proceso se captura la información requerida en la creación de la arquitectura y sus componentes. Los requerimientos pueden ser funcionales o no funcionales, es importante capturar las funciones básicas del sistema.

2. Especificación detallada: La especificación es a nivel más claro y preciso con la finalidad de ser verificable con los requerimientos generados.

3. División de Hardware/Software: El propósito es describir como el sistema implementa sus funcionalidades separándose, por un lado, en Hardware; y por otro, en Software. Muchos sistemas complejos se construyen a su vez a partir de diseños base. El sistema completo requiere de secciones de software y hardware, éstos, a su vez, son construidos a partir de componentes más pequeños que necesitan ser diseñados. El flujo de diseño muestra los niveles de abstracción del sistema, iniciando por el más abstracto hasta el diseño para componentes individuales manteniendo la metodología Top-Down(ver Figura 1.2).

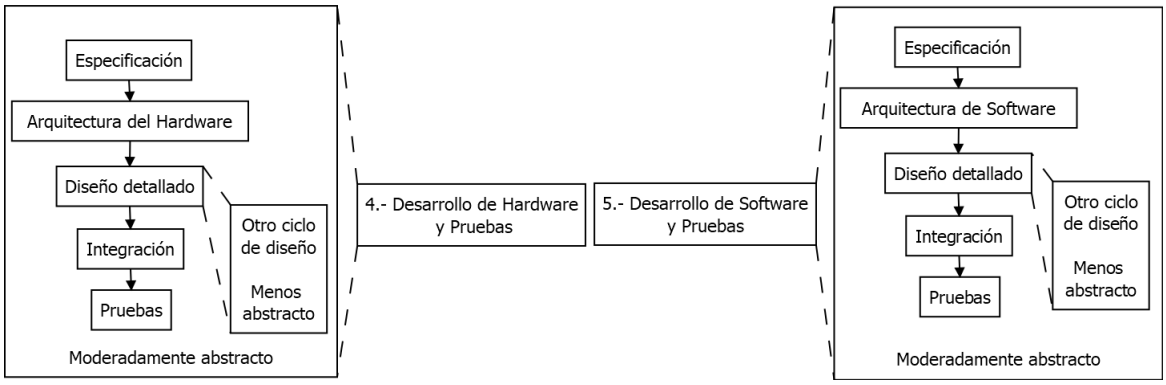


Figura 1.2: Metodología para un diseño Jerárquico[10].

4. Desarrollo de hardware y pruebas: desarrolla la parte correspondiente a los módulos de hardware en la parte lógica del SoC.

5. Desarrollo de software y pruebas: desarrolla la parte correspondiente a los programas por software en la parte del procesamiento SoC, en este caso se ejecutan en el procesador ARM.

6. Integración y resultados: Una vez teniendo los componentes perfectamente conectados y sincronizados se esta en la fase de producción de resultados.

1.4. Organización de la Tesis

La presente tesis se encuentra organizada en cinco Capítulos, cuyo contenido se describe a continuación:

- El Capítulo 2 está subdivido en dos partes: en la primera parte: se plantea la base que resalta la importancia del uso de las matrices mal condicionadas, además, da el contexto en donde se pueden encontrar, y plantea la situación en la que: si no se considera esta condición en la solución de sistemas lineales genera errores considerables. Para esto, se comenzará a describir lo que es una matriz mal condicionada, para continuar con exponer lo que es un problema mal planteado. Se considera el concepto de matrices invertibles, y continua en detallar el cálculo de la solución de la matriz inversa empleando el método de la matriz adjunta. En la segunda parte: se describe brevemente la arquitectura del SoC (*System on Chip*), recurso de importancia en la implementación de la tesis, muestra sus principales características y lo referente a la Síntesis de Alto Nivel (HLS). Por último, se explica que es el *Framework* PYNQ, sus principales componentes y las ventajas de utilizarlo.

- En el Capítulo 3 se enfoca en presentar las particularidades del sistema diseñado para darle solución a "El consumo de recursos y la precisión en dispositivos programables: caso de matrices mal condicionadas"; una vez que se han revisado diferentes metodologías en la literatura, se ha adaptado la mostrada en la Figura 1.1. Así, este Capítulo resalta la parte de diseño de la arquitectura, y desglosa la metodología en sus diferentes fases.

- En el Capítulo 4 se presentan los principales resultados logrados a partir de la operatividad de la implementación realizada, considerando que uno de sus parámetros de entrada es una matriz mal condicionada, y considera el caso del tipo de número para los tipos *float* y *double*. Además, muestra los recursos utilizados para la implementación haciendo la diferencia entre datos tipo *float* y *double*.

- En el Capítulo 5 se presentan las conclusiones obtenidas con base a los resultados logrados del Sistema que utiliza datos *float* y *double*; también, expone los recursos requeridos para la implementación.

Capítulo 2

Elementos Básicos

2.1. Antecedentes

En [2] se realiza la implementación de la inversa de matriz de $2x2$ por el método de la adjunta, esto para destacar como el mal condicionamiento y la precisión puede afectar el diseño del sistema, en términos de recursos, costos, etc. Para ilustrar los costos en hardware, se realiza la implementación en diferentes FPGAs, en especifico de la familia de Xilinx (Spartan-7, Artix-7, Kintex-7 y Virtex-7). La arquitectura es útil para analizar la precisión (Half Precission - 16 bits, Single Precission - 32 bits, Double Precission - 64 bits o Extendend Precision - 79 bits), utilizando la herramienta Simulink de Matlab, en donde se determina la relación de precisión y el mal condicionamiento, concluyendo que la necesidad de recursos parece crecer de forma no lineal cuando la precisión se incrementa.

En [16] se muestra una solución alternativa a las basadas en FPGAs, ya que se realiza mediante un esquema acelerado por la GPU (Implementación de Software) para Gradientes Conjugados (CG), centrado en el SpMV (Multiplicación Matriz-Vector Dispersa, en español).

En [28] se centra en la Terapia Fotodinámica (TFD), la cual tiene importantes aplicaciones médicas como lo es el tratamiento del cáncer, lo relevante es que para optimizar la TFD, se requiere resolver un problema inverso, es decir, desde una distribución de luz deseada se obtienen los parámetros que la han provocado. Los autores utilizan el software FulMonteSW, el cual modela geometrías 3D complejas con mallas tetraédricas y utiliza técnicas de Monte Carlo para modelar las interacciones de los fotones con los tejidos. También realiza la implementación del FullMonteFPGACL: una versión acelerada por FPGA de FullMonteSW que utiliza una FPGA Intel Stratix 10 y el SDK de Intel FPGA para OpenCL.

De acuerdo a las características de los trabajos mencionados anteriormente, se muestra una utilización de herramientas externas como lo es Simulink, u OpenCL. donde la interacción con la arquitectura implementada en el FPGA es hasta cierto punto rígida, no es así utilizando la Síntesis de Alto Nivel, que si bien esta restringido a una sola

plataforma, ya sea Xilinx(HLS), Intel (HLS Compiler) o Microchip (Smart High Level Synthesis) está permite mayor libertad al realizar módulos RTL y poder obtener un módulo descrito en VHDL o Verilog. Otro punto importante del trabajo es la utilización del framework PYNQ, el cual permite agilizar el diseño al permitir la programación en lenguaje Python y además poder tratar los diferentes diseños como librerías de hardware.

2.2. Matrices

2.2.1. Matrices Invertibles

La inversa A^{-1} de una matriz $A \in C^{nxn}$, está definida tal que:

$$AA^{-1} = A^{-1}A = I, \tag{2.1}$$

donde I es la matriz identidad de dimensión nxn. Si A^{-1} existe, entonces A es una matriz no singular, en caso contrario la matriz es singular.

2.2.2. Cofactores y Adjunta

La submatriz de la matriz A, denotada por $[A]_{ij}$ es una matriz de dimensión $(n-1)\,x\,(n-1)$, la cual se obtiene eliminando la hilera i-ésima y la columna j-ésima de la matriz A. El cofactor (i,j) de la matriz se define como :

$$cof\,(A,i,j) = (-1)^{i+j}\,det\left([A]_{ij}\right). \tag{2.2}$$

La matriz de cofactores puede ser creada usando los cofactores:

$$cof(A) = \begin{pmatrix} cof(A,1,1) & \cdots & cof(A,1,n) \\ \vdots & cof(A,i,j) & \vdots \\ cof(A,n,1) & \cdots & cof(A,n,n) \end{pmatrix}. \tag{2.3}$$

La matriz adjunta es la traspuesta de la matriz de cofactores:

$$adj(A) = (cof\,(A))^T. \tag{2.4}$$

2.2.3. Determinante

El determinante de la una matriz $A \in C^{nxn}$ está definida como:

$$det(A) = \sum_{j=1}^{n} (-1)^{j+1}\,A_{1j} det\left([A]_{1j}\right) \tag{2.5}$$

$$= \sum_{j=1}^{n} A_{1j} cof(A,1,j). \tag{2.6}$$

La inversa de una matriz se puede construir usando la matriz adjunta:[12]

$$A^{-1} = \frac{1}{det(A)} adj(A). \tag{2.7}$$

2.3. Número de Condición

En el campo del análisis numérico, el número de condición de una función respecto de su argumento mide cuánto se modifica el valor de salida si se realiza un pequeño cambio en el valor de entrada. Es decir, cuánto cambia $y = f(x)$ si se modifica x, es utilizado para medir cuán sensible resulta una función a variaciones o errores en el valor de entrada, y medir el error de salida dada la entrada.

Si al poner en práctica un sistema, la aproximación de x^δ esta dada por $\epsilon = Ax^\delta - b$, se espera que si $\|\epsilon\|$ es pequeño, entonces $\|x - x^\delta\|$ también lo sea, no obstante, puede suceder que si el sistema es muy sensible, $\|\epsilon\|$ no dependa continuamente del error en el sistema.

Definición 2.3.1. *Dada una norma $\|\cdot\|$ definida sobre R^n, una norma supeditada $|||\cdot|||$ para una matriz A se define como:[11]*

$$|||A||| = sup_{x \in R^n} \frac{\|Ax\|}{\|x\|} \tag{2.8}$$

Definición 2.3.2. *Sea $|||\cdot|||$ una norma matricial subordinada y $A \in M_{m*n}$ una matriz invertible, el número*

$$k = cond(A) = |||A|||||A^{-1}|||. \tag{2.9}$$

se denomina número de condición de la matriz A relativo a la norma $\|A\|$. Cuando se considera sobre R^n la norma euclidiana, se define a la norma supeditada $\|A\|_2$, de la manera siguiente

$$|||A||_2 = \sqrt{\rho\,(A^t A)}. \tag{2.10}$$

donde $\rho\,(A) := max\{|\lambda| : \lambda$ es un a valor propio de A$\}$, denota el radio espectral de A.

Un buen ejemplo del número de condición es el número de condición de una matriz cuadrada no singular, la cual llamaremos A, por lo tanto, se define como:

$$k(A) = \|A\|\|A^{-1}\|. \tag{2.11}$$

Para ser más preciso, la ecuación 2.11 hace referencia al número de condición con respecto a la inversión. Con relación al valor numérico del número de condición, se tienen las siguientes connotaciones:

- Cuanto más grande sea el número de condición, más cercano está la matriz de ser singular.
- El número de condición de la matriz identidad es 1.

- Una matriz singular tiene un número de condición infinito.

A su vez cuando una matriz tiene un número de condición demasiado grande (y singular si es infinito), se puede afirmar que la matriz está Mal Condicionada, en caso contrario la matriz es bien condicionada y no supondrá un problema el realizar operaciones con ellas.

2.4. Ejemplos de Casos de Matriz Mal Condicionada

Un problema típico surge en ingeniería eléctrica cuando se realiza la deconvolución. Si $e(t)$ representa una forma de onda de excitación limitada en el tiempo conocida, $h(t)$ la respuesta de impulso de un sistema lineal y $r(t)$ la respuesta medida del sistema a la excitación, entonces el proceso de recuperar $h(t)$ de $e(t)$ y $r(t)$ se conoce como deconvolución. Por lo tanto, $h(t)$ es la solución de una ecuación integral de Fredholm del primer tipo, cuya versión discreta implica la ecuación matricial no condicionada $Eh = r$. Los errores en la respuesta medida, $r(t)$, conducen a un error amplificado en la respuesta de impulso calculada, $h(t)$, lo que a menudo da como resultado una solución calculada altamente oscilatoria incluso cuando la respuesta de impulso real es bastante suave[15].

Otro problema de matriz mal condicionada se encuentra en el área de transferencia de calor, cuando se trata de la ecuación de calor hacia atrás en el tiempo. Considere la ecuación de calor $u_t = u_{xx}$, $0 \leqslant x \leqslant \pi$, que describe la temperatura $u(x,t)$ de una barra de longitud π en el tiempo t y en la posición x. Suponga que la condición inicial que describe u es $u(x,0) = u_0(x)$, mientras que las condiciones de frontera son $u(0,t) = u(\pi,t) = 0$. El problema retrospectivo en el tiempo consiste en medir la temperatura en el tiempo $t = 1$, $u_1(x)$, por ejemplo, y calcule la temperatura $u_0(x)$. Como ecuación integral el problema es resolver:

$$u_1(x) = \int_{\pi}^{0} k(x,s)u_0(s)\,ds. \tag{2.12}$$

donde:

$$k(x,s) = \frac{2}{\pi}\sum_{n=1}^{\infty} e^{n^2}\text{sen}(nx)\text{sen}(ns). \tag{2.13}$$

Este problema es mal planteado y conduce a problemas con matrices mal condicionadas. Errores en las mediciones de la temperatura $u_1(x)$ usualmente conducen a errores inaceptables en la temperatura calculada $u_0(x)$.

Un tercer ejemplo, que surge en el estudio de las oscilaciones, se refiere a la extracción de frecuencias resonantes naturales de estructuras a partir de mediciones de su comportamiento transitorio. Suponga que $q(t)$ representa una cantidad física medida, como el desplazamiento, que se sabe que es una suma de modos de oscilación natural, y se van a determinar las frecuencias naturales de oscilación. El uso del método de Prony[1] conduce a una ecuación matricial que está mal condicionada. Por lo tanto, pequeños errores en

[1]El método Prony extrae funciones exponenciales complejas amortiguadas (o sinusoides) de una señal

la cantidad medida $q(f)$ conducen a un error amplificado en las frecuencias naturales extraídas.

2.5. Problemas Mal Planteados

De acuerdo con Jaques Hadamard, un problema es bien planteado si se ha demostrado que:

- Existe una solución.
- La solución es única.
- La solución depende continuamente de sus datos.

Por el contrario, una amplia clase de los llamados problemas inversos que surgen en física, tecnología y otras ramas de la ciencia, en particular, problemas de procesamiento de datos de experimentos físicos, pertenece a la clase de problemas mal planteados. Sea z una cantidad característica del fenómeno (u objeto) a estudiar. En un experimento físico, la cantidad z es frecuentemente inaccesible a la medición directa, pero lo que se mide es una determinada transformada $Az = u$ (también llamada resultado). Para la interpretación de los resultados es necesario determinar z a partir de u, es decir, resolver la ecuación:

$$Az = u. \tag{2.14}$$

Los problemas de resolución de una Ecuación 2.14 a menudo se denominan problemas de reconocimiento de patrones. Los problemas que conducen a la minimización de funcionales (diseño de antenas y otros sistemas o construcciones, problemas de control óptimo y muchos otros) también se denominan problemas de síntesis.

Supongamos que en un modelo matemático para algunos experimentos físicos el objeto a estudiar (el fenómeno) se caracteriza por un elemento z (una función, un vector) perteneciente a un conjunto Z de posibles soluciones en un espacio métrico $\hat{Z}$. Supongamos que z_T es inaccesible a la medición directa y que lo que se mide es una transformada, $Az_T = uT$, $uT \in AZ$, donde AZ es la imagen de Z bajo el operador A. Evidentemente, $zT = A^1uT$, donde A^1 es el operador inverso a A. Dado que u_T se obtiene por medición, sólo se conoce aproximadamente. Sea $\bar{u}$ este valor aproximado. En estas condiciones la cuestión sólo puede ser la de encontrar una "solución"de la ecuación:

$$Az = \bar{u} \tag{2.15}$$

Aproximando z_T. En muchos casos el operador A es tal que su inverso A^{-1} no es continuo, por ejemplo, cuando A es un operador completamente continuo en un espacio de Hilbert,

dada resolviendo un conjunto de ecuaciones lineales. Permite la estimación de frecuencias, amplitudes y fases de una señal[9].

en particular un operador integral de la forma:

$$\int_a^b K(x,s)z(s)\,ds \tag{2.16}$$

En estas condiciones no se puede tomar, siguiendo las ideas clásicas, una solución exacta de 2.15, es decir, el elemento $z = A^{-1}\bar{u}$, como una *solución aproximada* de z_T. De hecho: a) tal solución no necesita existir en Z, ya que $\bar{u}$ no necesita pertenecer a AZ; y b) tal solución, si existe, no necesita ser estable bajo pequeños cambios de $\bar{u}$ (debido al hecho de que A^{-1} no es continuo) y, en consecuencia, no necesita tener una interpretación física. El problema 2.15 entonces está mal planteado [17].

2.6. El *System on Chip*

2.6.1. Arquitectura del SoC Zynq-7000

La tarjeta utilizada para la implementación de los diferentes algoritmos requeridos en el desarrollo de la matriz mal condicionada es la tarjeta de desarrollo PYNQ-Z2, la cual, es basada en un SoC Zynq-7000; el diseño de la arquitectura está constituido de dos partes:

- El Sistema de Procesamiento (PS - *Processing System*): Está constituido por un procesador de doble núcleo ARM Cortex-A9, este es alojado físicamente en el chip, por lo que es considerado un *hard processor*.

- Lógica Programable (PL - *Programing Logic*): Esta sección es la correspondiente a un FPGA y para utilizarse se requiere de las herramientas de diseño como VHDL, HLS, u otros.

La Figura 2.1 muestra la estructura completa de un SoC, y como se comentó anteriormente, este contiene un *hard processor* en la sección de PS, sin embargo, tiene la ventaja de poder implementar un *soft processor* como el MicroBlaze u otro procesador de código abierto en la sección PL, lo que representa una enorme ventaja en aplicaciones, ya que incrementa el espectro de posibilidades de uso.

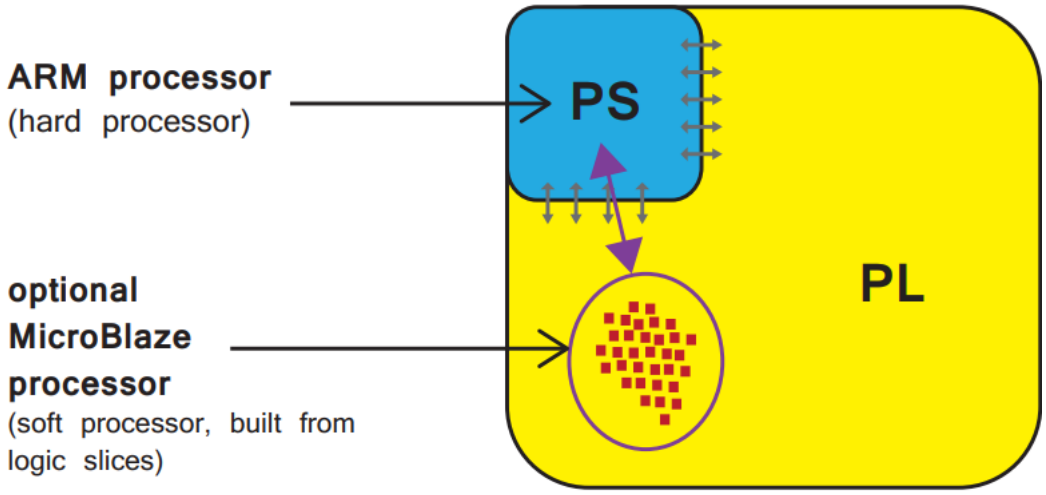


Figura 2.1: Componentes del SoC Zynq [5].

2.6.2. Especificaciones

Algunas de la características de la tarjeta PYNQ-Z2 es que está diseñada para utilizarse con un **PYNQ**, es decir, con **PY**thon, y con un SoC ZY**NQ** (XC7Z020-1CLG400C). La Tabla 2.1 muestra algunas características de la tarjeta.

Tabla 2.1: Características de la tarjeta PYNQ-Z2.

Característica	Descripción
ZYNQ	XC7Z020-1CLG400C
Procesador	650MHz Arm Cortex-A9 Dual-core
Lógica Programable	▪ 13,300 logic slices. ▪ 630KB block RAM ▪ 220 DSP slices ▪ On-chip Xilinx analog-to-digital converter (XADC)
Dimensiones	87mm x 140mm (3.43”x 5.51”)

2.7. Síntesis de Alto Nivel

Esta herramienta es utilizada para el desarrollo de los módulos RTL[2] a partir de la funciones en lenguaje C, C++ o SystemC, el proceso general se muestra en la Figura 2.2, en este trabajo se utilizó el lenguaje C para la creación de los módulos RTL.

[2]RTL: El Nivel de Transferencia de Registro es un nivel de descripción de un diseño digital en el que el comportamiento cronometrado del diseño se describe expresamente en términos de transferencias de datos entre elementos de almacenamiento en lógica secuencial, que puede ser implícita, y lógica combinatoria, que puede representar cualquier computación o aritmética-lógica-unidad-lógica[14].

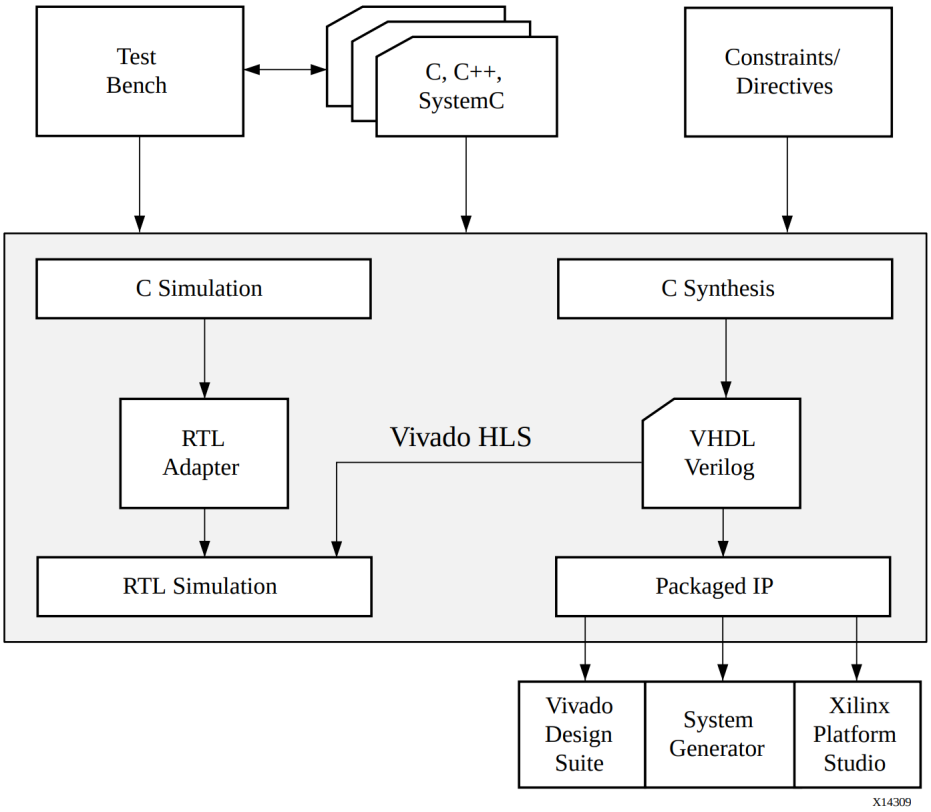


Figura 2.2: Flujo de Diseño usando Síntesis de Alto Nivel (de Xilinx) [26].

De la Síntesis se obtienen las siguientes salidas:

- Implementación RTL en Lenguajes de Descripción de Hardware (HDL): Puede sintetizar el RTL en una implementación a nivel de compuertas y un archivo de flujo de bits FPGA. El RTL está disponible en los siguientes formatos estándar de la industria:
  - VHDL (IEEE 1076-2000)
  - Verilog (IEEE 1364-2001)
- Archivos de Reporte: Son resultados de la síntesis, simulación C/RTL y el empaque a formato IP.

2.7.1. Verificación

Para verificar que el diseño sea funcional es posible utilizar un *Test Bench*. Esta simulación se separa en dos procesos distintos:

- Validación previa a la síntesis que valida que el programa C implemente correctamente la funcionalidad requerida.

- Verificación posterior a la síntesis que verifica que el RTL es correcto.

Para la compilación y simulación Vivado HLS tiene soporte para los siguientes estándares [26]:

- *ANSI-C (GCC 4.6)*
- *C++ (G++ 4.6)*
- *SystemC (IEEE 1666-2006, version 2.2)*

2.7.2. Exportar RTL

El bloque RTL generado se puede exportar y empaquetar los archivos de salida finales como una IP, se pueden elegir los siguientes formatos [26]:

- Vivado IP: Importa al catálogo de Vivado IP para usar en Vivado Design Suite.
- System Generator para DSP: Importa el diseño HLS al System Generator.
- Synthesized Checkpoint (.dcp): Importa directamente a Vivado Design Suite.

2.7.3. Pragma

Un *Pragma* indica al compilador que realice una acción en particular al momento de la compilación. Estos *Pragmas* varían de un compilador a otro[13]. En particular Vivado HLS proporciona *Pragmas* que pueden ser utilizados para optimizar el diseño: reducir latencia, mejorar el rendimiento y reducir el área y recursos utilizados por el dispositivo.

En este trabajo se hizo uso de *Pragmas* del tipo de Síntesis de Interfaz, el cual permite especificar como son creados los puertos RTL a partir de la definición de la función durante la síntesis de la interfaz. A continuación, se describen los modos de protocolos de interfaz utilizados en el presente trabajo:

- bram : Implementa argumentos de matriz como una interfaz RAM estándar.
- s_axilite: Implementa todos los puertos como una interfaz AXI4-Lite. Vivado HLS produce un conjunto asociado de archivos de controlador en lenguaje C durante el proceso Exportar RTL.

2.8. Productividad de Python para Zynq

Para la programación del Sistema se utilizó el Framework PYNQ (Python Productivity for Zynq), cuyo principal objetivo es facilitar y agilizar el diseño. Para esto, combina los siguientes elementos:

- Lenguaje de Alto nivel (Python).
- Overlays de FPGA.
- Una Arquitectura basada en web proporcionada desde los procesadores embebidos.
- El uso de Jupyter en un contexto embebido [27].

En la Figura 2.3 se muestra la interacción usual, en una sesión de Jupyter, el servidor reproduce el archivo del Notebook en un navegador web convencional utilizando HTTP y una combinación de HTTP y protocolos Websockets para el contenido estático y dinámico del documento reproducido. En el back end, el servidor se comunica con un núcleo de ejecución de código utilizando un protocolo de mensajería ZeroMQ de código abierto [6].

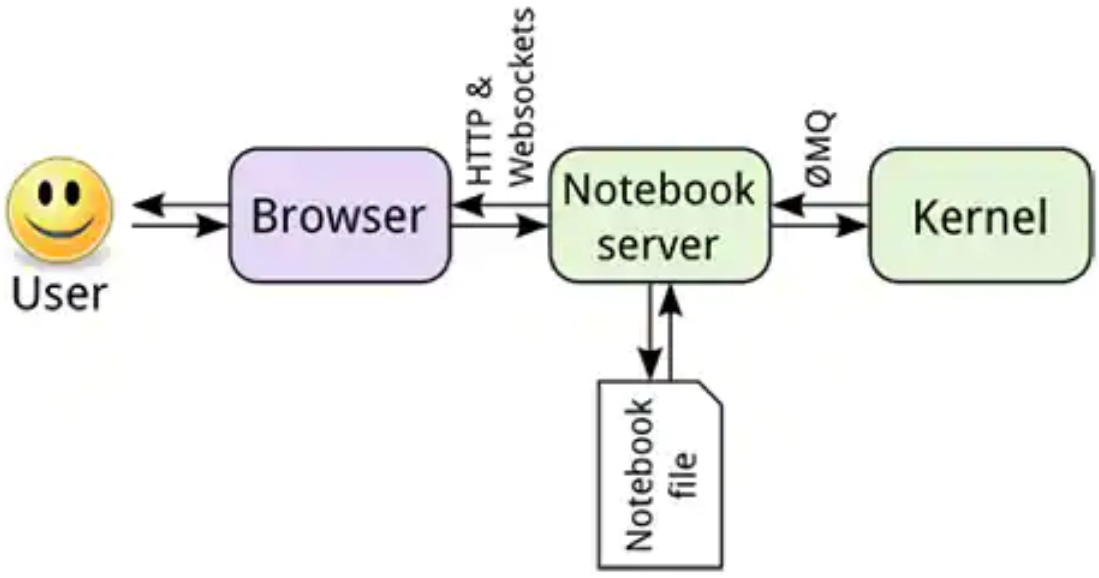


Figura 2.3: Estructura del Framework [6].

La versión de las herramientas de Xilinx para cada versión de PYNQ se muestran en la Tabla 2.2.

Para este trabajo, se utilizó la versión 2019.1 de las herramientas de Xilinx y la versión v2.5 de PYNQ.

2.8.1. Jupyter Notebook

Es un entorno informático interactivo, que permite al usuario crear documentos de diferentes tipos. Jupyter Notebook tiene 3 componentes principales:

- Aplicación web: Permite escribir y ejecutar código de una manera interactiva.

Tabla 2.2: Versiones recomendadas de PYNQ de acuerdo a la versión de Xilinx [27].

Versión de PYNQ	Versión de las herramientas de Xilinx
v1.4	2015.4
v2.0	2016.1
v2.1	2017.4
v2.2	2017.4
v2.3	2018.2
v2.4	2018.3
v2.5	2019.1
v2.6	2020.1

- Kernel: Separa los procesos iniciados por la aplicación web, ejecuta el código y devuelve la salida a la aplicación web. En el caso particular de PYNQ el Kernel por defecto es Python. El Kernel se ejecuta en el PS Zynq mientras que la aplicación web funciona como interfaz para el Kernel.

- Documentos: En lo referente a documentos, se incluyen el contenido de la aplicación web (entradas y salidas), multimedia, etc.

2.8.2. Overlays

Los Overlays, o bibliotecas de hardware, son diseños de FPGA programables/configurables que amplían la aplicación del usuario desde la zona del Processing System del Zynq hasta la zona Processing Logic. Los Overlays se pueden usar para acelerar una aplicación de software o para personalizar la plataforma de hardware para una aplicación en particular.

En el Framework PYNQ un *overlay* hace referencia al hardware implementado en la capa de Lógica Programable del SoC. Esta capa o librería de hardware extiende la funcionalidad del PS. El *Framework* permite controlar este hardware utilizando Python el cual se ejecuta en el PS. Para poder utilizar esta característica se requieren los siguientes elementos del diseño:

- Bitstream: Es el archivo que contiene la información de programación para el FPGA [25].

- Archivo *Tcl*: Es un script el cual contiene la información referente al diseño.

- Archivo *hwh*: Tiene la información del diseño y se utiliza por el software para abstraer toda la información necesaria para construir una aplicación dirigida al dispositivo diseñado [1].

2.8.3. Librerías

MMIO

Esta clase (*class*) de Python permite que el objeto acceda al banco de direcciones asignada en memoria del sistema. En específico a los registros y direcciones de los periféricos que se encuentren en el PL[21], en la Figura 2.4 se muestra la interacción entre el PS que actúa como maestro y una IP implementada en la sección PL del SoC y actúa como esclavo.

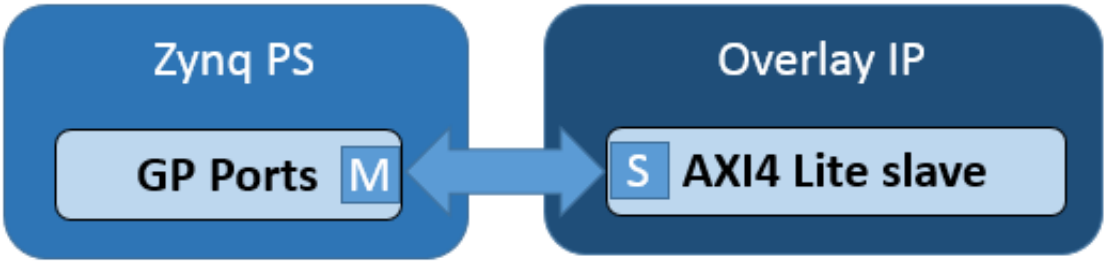


Figura 2.4: Diagrama a bloques de la interacción utilizando la clase MMIO [21].

MMIO proporciona una forma simple pero poderosa de acceder y controlar periféricos. Para periféricos simples con una pequeña cantidad de accesos a la memoria, o donde el rendimiento no es crítico, MMIO suele ser suficiente [21].

Ctypes

Es una biblioteca de funciones externas para Python. Proporciona tipos de datos compatibles con C [7], en la Tabla 2.3 se muestran los tipos de datos utilizados en el presente trabajo.

Tabla 2.3: Ctypes define una serie de tipos de datos primitivos compatibles con C [7].

Ctypes	C	Python
c_int	int	int/long
c_longlong	int64 long long	int/long
c_float	float	float
c_double	double	float

Struct

Este módulo realiza conversiones entre valores de Python y estructuras C representadas como cadenas de Python [8].

Capítulo 3

Implementación del Sistema

3.1. Requerimientos Funcionales

Está sección parte del flujo de diseño planteado en la Figura 1.1, en el punto de Requerimientos Funcionales se inicia a partir de un ejercicio de abstracción; el funcionamiento es descrito así: el sistema requiere de una matriz A de dimensión de $3x3$ de entrada, con la característica de que ésta matriz sea mal condicionada; posteriormente, el sistema determina la matriz inversa (A^{-1}) de A mediante el método de la adjunta, de este procedimiento se obtiene el determinante de la matriz A, también se calcula la matriz C, la cual es el resultado de la siguiente expresión:

$$C = AA^{-1}. \tag{3.1}$$

Del desempeño del sistema se obtienen los tiempos de ejecución de las operaciones (t_e). En la Figura 3.1 se muestran las entradas y salidas del sistema.

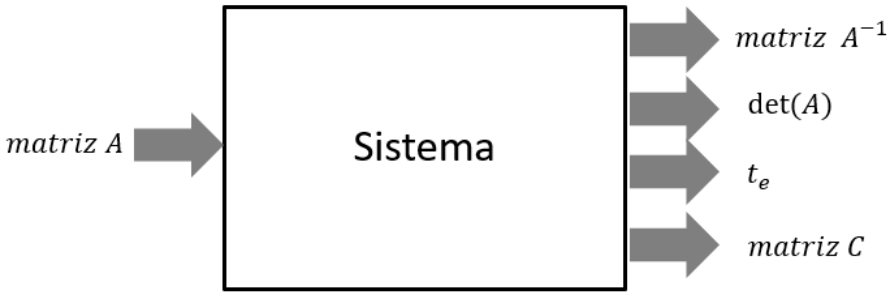


Figura 3.1: Entradas y salidas del sistema.

Adicionalmente a los requerimientos funcionales, el desarrollo del sistema se utilizó el SoC Zynq que se encuentra en la tarjeta PYNQ-Z2, esto porque después de conocer, usar y disponer de diversas tarjetas de desarrollo, por las prestaciones ajustadas a este proyecto de tesis se optó por utilizarse esta tarjeta de desarrollo.

Respecto al hardware del sistema, los recursos utilizados (bloques o IP's [1]) se pueden agrupar en dos categorías: disponibles (Librería de Xilinx) y creadas.

Para la creación de los módulos utilizados para el cálculo de la inversa y multiplicación se utilizó la Síntesis de Alto Nivel (HLS). Una vez que se tienen los módulos RTL de la multiplicación y la inversa se realiza la integración del sistema, la arquitectura y la totalidad de los módulos utilizados se describen en la Sección 3.2.

Para la verificación de la arquitectura desarrollada se utiliza el *Framework* PYNQ, este permite encapsular el diseño a implementarse en la sección de lógica programable del SoC[2] en librerías llamadas *Overlays*. La programación es a través del lenguaje Python, para probar el sistema se tomaron en cuenta matrices mal condicionada, esto se describe en el Capítulo 2.

3.2. División Hardware - Software

De acuerdo con los requerimientos descritos en la sección 1.3, el sistema propuesto utiliza la parte Lógica Programable (PL) y del Sistema de Procesamiento (PS), bajo este esquema, la arquitectura del SoC se muestra en la Figura 3.2.

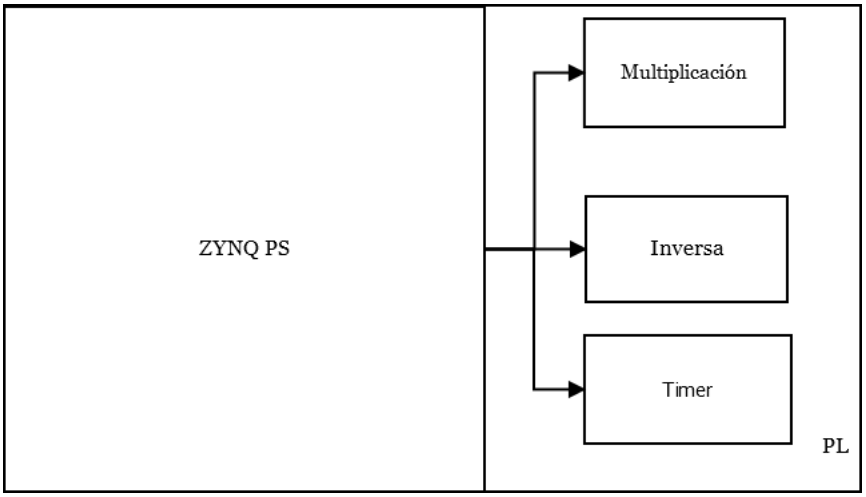
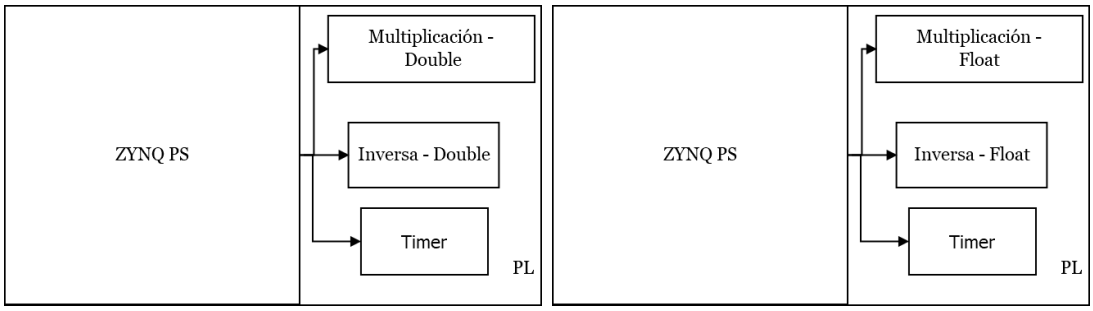


Figura 3.2: Abstracción del sistema considerando la arquitectura del SoC.

El diseño presentado en la Figura 3.2 es la estructura general, para realizar las comparaciones entre los tipos de datos, los únicos bloques que serán modificados de manera sustancial son los encargados de realizar las operaciones: Multiplicación e Inversa. En la Figura 4.4, se muestra la diferencia, tomando en cuenta esto, para referirse al diseño que tiene los bloques (Multiplicación e Inversa) utilizando datos *Float*, se le llamará Sistema

[1]IP: Un núcleo IP (propiedad intelectual) es un bloque de lógica o datos.
[2]SoC: Son las siglas en inglés para Sistema en Chip.

Float, y de igual forma el diseño con los módulos que utilizan datos Double, se le llamará Sistema Double, esta etiqueta servirá sobre todo en el Capítulo 4.



(a) Sistema Double. (b) Sistema Float.

Figura 3.3: Diseños realizados.

3.3. Hardware

Para el diseño del hardware se utilizaron diferentes softwares de la *Design Suite* de Vivado, los cuales se describen a continuación:

- Vivado HLS: Esta herramienta transforma una especificación de C en una implementación RTL (Register Transfer Level) que se puede sintetizar utilizando un FPGA[26].

- Vivado: En esta herramienta se realizó el desarrollo, análisis, síntesis e implementación del código RTL. Se utilizaron plantillas de la librería de Xilinx, así como también código RTL generado en Vivado HLS.

El sistema por lo tanto cuenta con 4 componentes: Zynq 7000, Inversa, Multiplicación y Timer. El diagrama del sistema a bloques realizado en Vivado se muestra en la Figura 3.4.

Es necesario precisar que los bloques Inversa y Multiplicación se encuentran agrupados en módulos jerárquicos dentro de estos bloques se encuentran sub-bloques. En la Tabla 3.1, se encuentran las direcciones asociadas a los elementos esclavos tanto para el Sistema *Float* como para el Sistema *Double*, esto para facilitar la programación.

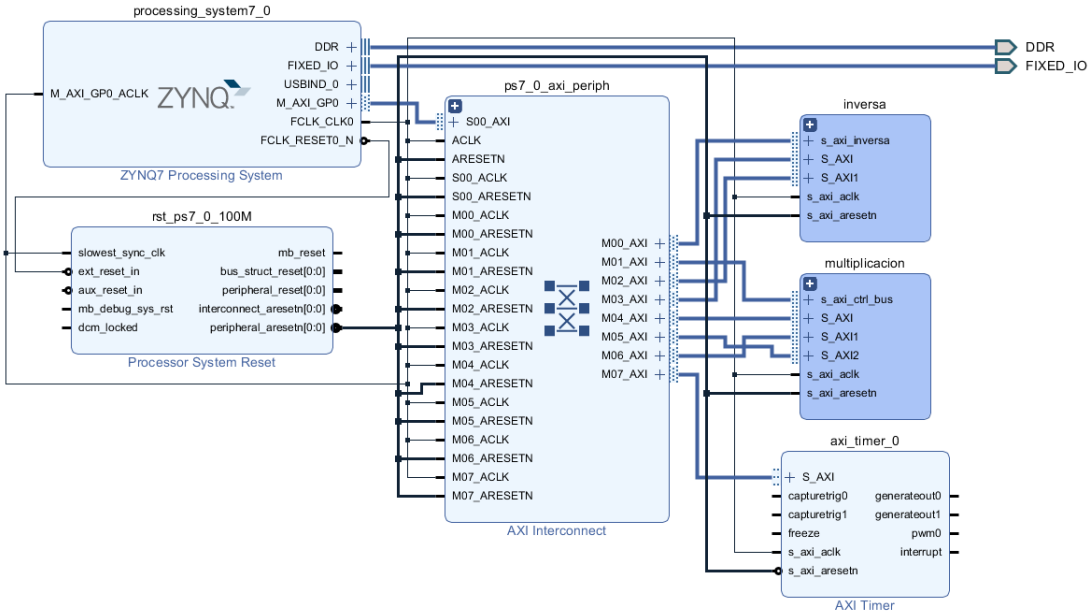


Figura 3.4: Diagrama a bloques del Sistema.

Tabla 3.1: Direcciones de los componentes esclavos.

Bloque	Interfaz esclavo	Offset Address	Rango	High Address
inversa/Inversa_0	s_axi_inversa	0x43C0_0000	64K	0x43C0_FFFF
control_MI/axi_bram_ctrl_0	s_axi	0x4000_0000	8K	0x4000_1FFF
control_MI/axi_bram_ctrl_1	s_axi	0x4200_0000	8K	0x4200_1FFF
control_MM/axi_bram_ctrl_2	s_axi	0x4400_0000	8K	0x4400_1FFF
control_MM/axi_bram_ctrl_3	s_axi	0x4600_0000	8K	0x4600_1FFF
control_MM/axi_bram_ctrl_4	s_axi	0x4800_0000	8K	0x4800_1FFF
axi_timer_0	s_axi	0x4280_0000	64K	0x4280_FFFF
multiplicacion/matrizMult_0	s_axi_ctrl_bus	0x43C1_0000	64K	0x43C1_FFFF

3.3.1. Zynq 7000

El sistema utiliza el PS del SoC, el cual se encarga de las siguientes tareas: Interacción con los aceleradores de Hardware (inversa y multiplicación), *timer* y despliegue de los resultados.

A continuación, se listan la totalidad de los módulos requeridos para utilizar el Zynq 7000 y su utilidad:

- Zynq7 Processing System: Para poder utilizar el PS, se requiere utilizar una IP de Xilinx, el cual es Processing System 7.

Figura 3.5: Entradas y salidas generales de la función Inversa.

- Processor System Reset: Proporciona resets personalizados para la totalidad del procesador del sistema, esto incluye al procesador y periféricos interconectados.
- AXI Interconnect: Es el componente que permite conectar uno o más dispositivos maestros asignados en memoria AXI a uno o más dispositivos esclavos asignados en memoria.

Los tres módulos anteriores se encuentran disponibles en las librerías de IP's de Vivado.

3.3.2. Inversa

Algoritmo para la Inversa (Método de la adjunta)

Como se mencionó en el Capítulo 1, en la síntesis de alto nivel se requiere un función en lenguaje C, en la Figura 3.5 aparecen la entrada y el retorno de la función mientras que en la Figura 3.6 se encuentra el diagrama de flujo de la función Inversa.

Referente al diagrama de flujo (ver Figura 3.6), los tres procesos se realizan en ciclos *for* independientes, esto se aprecia de mejor manera en el código en C (ver Listing 3.1).

Listing 3.1: Función de la inversa de una matriz (Método de la adjunta).

```
DataType Inversa(DataType Orig[DIM][DIM], DataType
inver[DIM][DIM]){
// variables involucradas en calcular el determinante
        DataType d_t1=0;
        DataType d_t2=0;
        DataType d_t3=0;
        DataType d=0;
// variables involucradas en calcular la matriz inversa
        DataType invt1=0;
        DataType invt2=0;
```

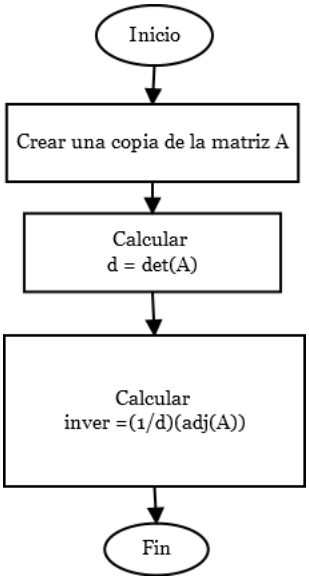


Figura 3.6: Diagrama de flujo de la función Inversa.

```
        DataType invt3=0;
// variable temporal
        DataType m[DIM][DIM];
// ciclo para copiar la matriz Orig
        for (int col = 0; col < DIM; col++){
                for(int fil = 0; fil < DIM; fil++){
                        m[col][fil] = Orig[col][fil];
                }
        }
// ciclo que calcula el determinante
        loop_determinante:for(int i = 0; i < DIM; i++){
          d_t1 = m[1][(i+1)%3] * m[2][(i+2)%3];
          d_t2 = m[1][(i+2)%3] * m[2][(i+1)%3];
          d_t3 = m[0][i] *(d_t1 - d_t2);
          d    = d + d_t3;
  }
// ciclo que calcula la inversa de la matriz
      loop_inversa1:for(int i = 0; i < DIM; i++){
        loop_inversa0:for(int j = 0; j < DIM; j++){
                invt1 = m[(j+1)%3][(i+1)%3] * m[(j+2)%3][(i+2)
                    %3];
                invt2 = m[(j+1)%3][(i+2)%3] * m[(j+2)%3][(i+1)
                    %3];
                invt3 = invt1 - invt2;
```

```
                    inver[i][j] = invt3/ d;
            }
        }
//regresa el valor del determinante
  return d;
}
```

El código tiene las siguientes características:

- DataType: Corresponde al tipo de dato de variable.
- DIM: Corresponde a la dimensión de la matriz.
- Orig e Iver: Corresponden a la matriz original, sea A y la matriz inversa (A^{-1})
- d: Determinante de la matriz Orig.

Adicionalmente se utilizan dentro del código variables temporales tanto para almacenar valores resultados de los tres procesos, como también para los utilizados en el ciclo *for*.

El bloque Inversa se muestra en la Figura 3.7.

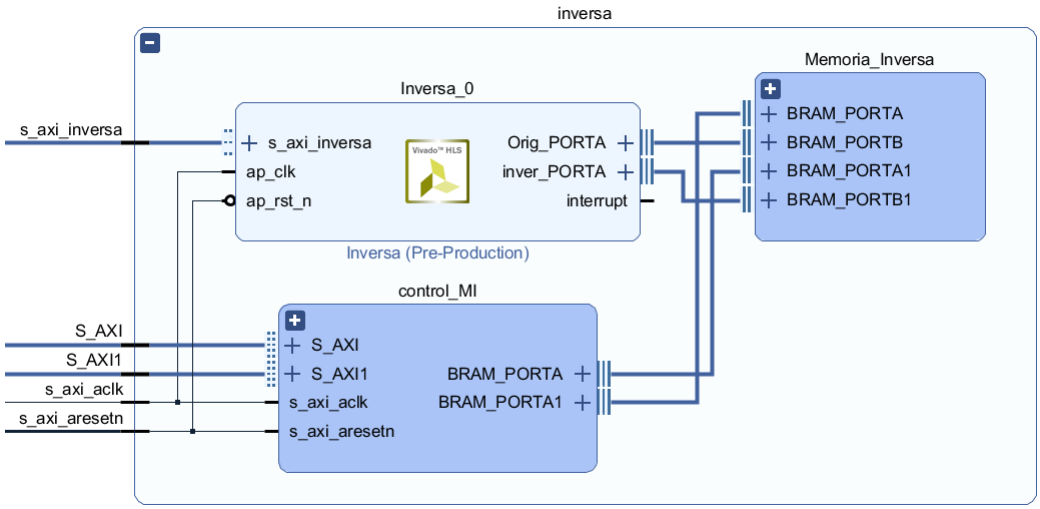


Figura 3.7: Componentes del módulo Inversa.

Su estructura se divide en 3 bloques principales: Inversa_0, Control_MI y Memoria_Inversa.

Inversa_0

Para la implementación RTL fue utilizado el software Vivado HLS. El código utilizado para la síntesis se encuentra la Sección 3.1 con todas las características descritas anteriormente.

El módulo RTL generado en *Vivado HLS* se muestra en la Figura 3.8. En la Tabla 3.2 se encuentra una descripción de los puertos del módulo RTL.

Figura 3.8: Módulo RTL de la Inversa.

Tabla 3.2: Descripción puertos - Inversa.

Puerto	Interfaz	Descripción
s_axi_inversa	s_axi_lite	Comunicación con el PS y retorno del valor del determinante.
Orig_PORTA	bram	Comunicación con el bloque de memoria para leer los datos de la matriz A.
inver_PORTA	bram	Comunicación con el bloque de memoria para almacenar los datos de la matriz A^{-1}.

El código en C de la función sintetizada y las pruebas realizadas para verificar el funcionamiento del módulo se muestran en el Anexo A.

Control_MI

Para que el PS tenga acceso a los bloques de memoria, se utilizó la IP AXI BRAM Controller. Por cada bloque de memoria se debe utilizar la IP ya mencionada.

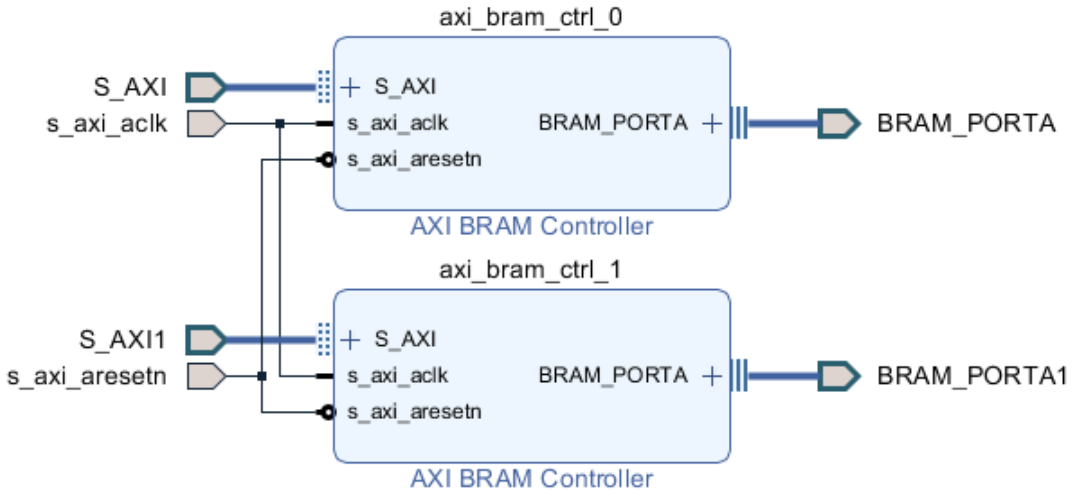


Figura 3.9: Bloques internos de Control_MI.

Memoria_Inversa

Este bloque jerárquico contiene los bloques de memorias en los cuales son almacenadas la matriz Orig e inver. Esto se puede ver en la Figura 3.10. En la Tabla 3.3 se encuentran las características de los bloques de memoria utilizados.

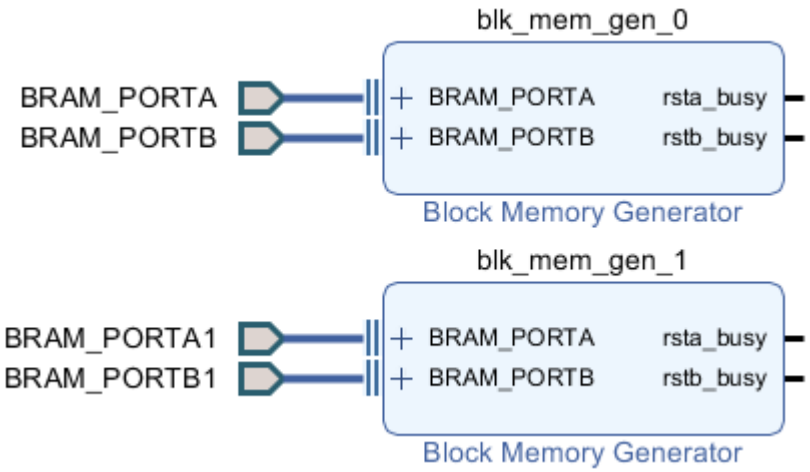


Figura 3.10: Contenido del bloque Memoria_Inversa.

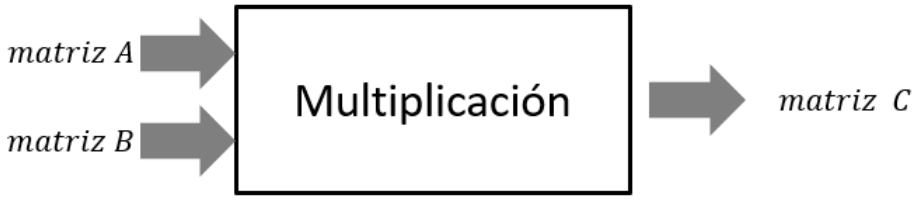


Figura 3.11: Entradas y salidas de la función Multiplicación.

Tabla 3.3: Características de los bloques de memoria.

Característica \ Diseño	Sistema Float	Sistema Double
Modo	BRAM Controller	
Tipo de Memoria	True Dual Port RAM	
Tamaño de la dirección	32 bits	
Tamaño del dato	32 bits	64 bits

3.3.3. Multiplicación

Como se expone en la sección 1.3, se realizó un módulo de multiplicación de matrices, en la Figura 3.11 se encuentra el diagrama a bloques de la función.

Listing 3.2: La función matrizMult realiza la multiplicación matricial.

```
void matrizMult(const DataType A[DIM][DIM],
const DataType B[DIM][DIM],DataType C[DIM][DIM]){
//La variable sum es temporal
DataType sum=0;
L1:for (int ia = 0; ia < DIM; ++ia){
  L2:for (int ib = 0; ib < DIM; ++ib){
    sum = 0;
    L3:for (int id = 0; id < DIM; ++id){
      sum += A[ia][id] * B[id][ib];
      }
      C[ia][ib] = sum;
    }
   }
}
```

La función Multiplicación, tiene las siguientes características:

- DataType: Corresponde al tipo de dato de variable.
- DIM: Corresponde a la dimensión de la matriz.

- A, B y C: Las dos primeras corresponde a matrices de entrada y C al resultado de la operación.

El bloque Multiplicación se encuentra en la Figura 3.12. La estructura se compone de los siguientes tres bloques: matrizMult_0, control_MM y memoria_multiplicacion. El módulo de la multiplicación realiza la siguiente operación:

$$C = AB \tag{3.2}$$

Donde A, B y C son matrices de 3×3.

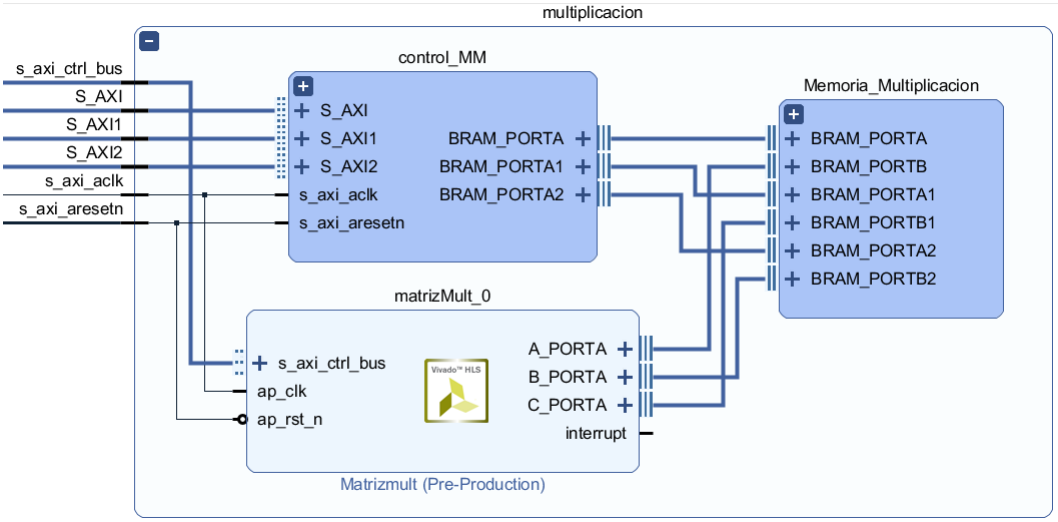


Figura 3.12: Componentes del módulo Multiplicación.

Con base a las características anteriores se elaboró la función en C (ver Listing 3.1) y el módulo RTL obtenido de la síntesis se encuentra en la Figura 3.13.

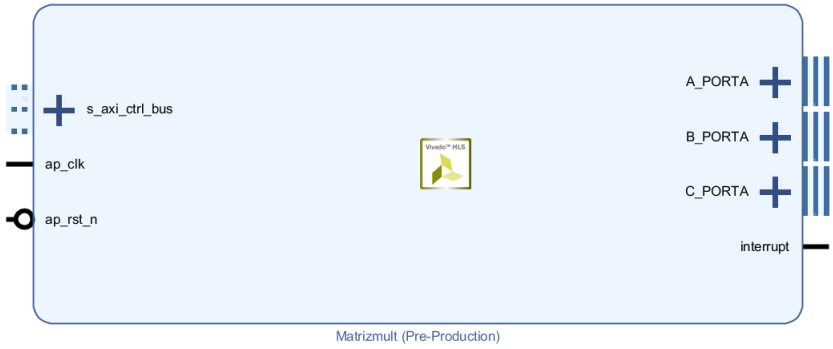


Figura 3.13: Módulo RTL de la multiplicación.

En la Tabla 3.4, se describen los puertos del módulo RTL obtenido.

Tabla 3.4: Descripción puertos - Multiplicación.

Puerto	Interfaz	Descripción
s_axi_ctrl_bus	s_axi_lite	Comunicación con el PS.
A_PORTA	bram	Comunicación con el bloque de memoria para leer los datos de la matriz A.
B_PORTA	bram	Comunicación con el bloque de memoria para almacenar los datos de la matriz B.
C_PORTA	bram	Comunicación con el bloque de memoria para almacenar los datos de la matriz C.

El código en C de la función sintetizada y las pruebas realizadas para verificar el funcionamiento del módulo se muestran en el Anexo A.

Tipos de datos

Los anteriores bloques se realizaron utilizando los siguientes Formatos Estándar para la aritmética de punto flotante (IEEE 754):

- binary32 - (*Single Precision*).
- binary64 - (*Double Precision*).

El número positivo más pequeño y más grande que se puede representar con cada formato es mostrado en la Tabla 3.5.

Tabla 3.5: Características del formato Single y Double.

Nombre	Base	Signo	Exponente	Mantisa	Número Positivo más pequeño	Número Positivo más grande
Single	2	1	8	23	$2^{-126} \approx 1.2x10^{-38}$	$(2-2^{-23})\,2^{127} \approx 3.4x10^{38}$
Double	2	1	11	52	$2^{-1022} \approx 2.2x10^{-308}$	$(2-2^{-52})\,2^{1023} \approx 1.8x10^{308}$

Para facilitar la síntesis, se utiliza un archivo *header* que contiene la definición de la clase (*class*) para cada módulo, ver Anexo A y Anexo B.

3.3.4. Temporizador

En el temporizador se utilizó un IP disponible en la librería de Vivado, en la Figura 3.14 se encuentra el diagrama a bloques del AXI Timer.

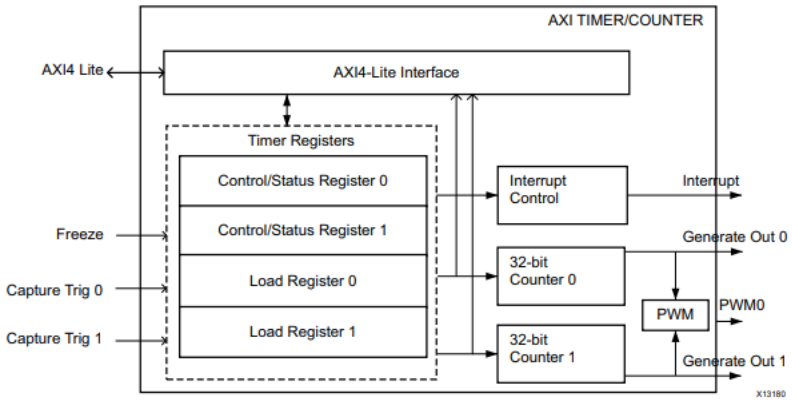


Figura 3.14: Diagrama a bloques del AXI Timer [20].

De manera particular se describe a continuación las características utilizadas del AXI Timer:

- AXI4-Lite Interface: Permite acceder a los registros del temporizador mapeados en memoria.
- Registros del Temporizador: Utiliza un conjunto de registros de 32 bits para cada temporizador/contador, este conjunto de registros contiene el registro de carga, registro de temporizador/contador y registro de control/status.
- Contador de 32 bits: El módulo AXI Timer contiene dos contadores de 32 bits, cada uno de los cuales se pueden configurar para conteos ascendente/descendente y además permite precargar con un valor desde el registro de carga.

En el diseño se utilizó el contador en modo Cascada, los dos contadores/temporizadores operan en cascada como un contador/temporizador de 64 bits.

El intervalo de tiempo asociado al contador ascendente se obtiene con la siguiente expresión:

$$TIMING\_INTERVAL = (MAX\_COUNT - TLRx + 2) * AXI\_CLK\_PERIOD \tag{3.3}$$

Donde:

- TIMING_INTERVAL: Es el intervalo de tiempo en segundos.
- MAX_COUNT: Es el valor máximo que permite el contador, $0xFFFFFFFF$ para un contador de 32 bits.
- AXI_CLK_PERIOD : el periodo el reloj de la interfaz AXI, ($1x10^{-8}$ s = 10 ns).
- TLRx : Valor del conteo.

3.4. Software

El software diseñado para el dispositivo se divide en dos grupos: funciones básicas y pruebas. Los archivos utilizados se encuentran en la Figura 3.15.

```
root@crzc:/# ls /home/xilinx/jupyter_notebooks/TesisResultados/
DoubleDesign.bit  DoubleDesign.hwh  DoubleDesign.tcl  FloatDesign.bit  FloatDesign.hwh  FloatDesign.tcl  Untitled.ipynb
root@crzc:/#
```

Figura 3.15: Archivos utilizados.

Los archivos con nombre *DoubleDesign* y *FloatDesign* son archivos que contienen información referente al diseño (archivos con extensión *.hwh* y *.tcl*) y su respectivo archivo de programación (extensión *.bit*) y finalmente el archivo correspondiente al Jupyter Notebook.

3.4.1. Funciones Básicas

El software tiene tres funciones básicas: *Selección de Overlay*, *Realizar inversa* y *Realizar multiplicación*. A continuación se explican estas funcionalidades.

Selección de *Overlay*

Esta función descarga el diseño a la sección PL del SoC. La función espera como entrada de tipo *string* que determina la elección del diseño de tipo *Float* o de tipo *Double*. Además, la función contiene información referente a las direcciones de los componentes esclavos (ver Tabla 3.1) accediendo a través de la clase *MMIO*, ambos Sistemas *Float* y *Double* comparten la dirección de memoria.

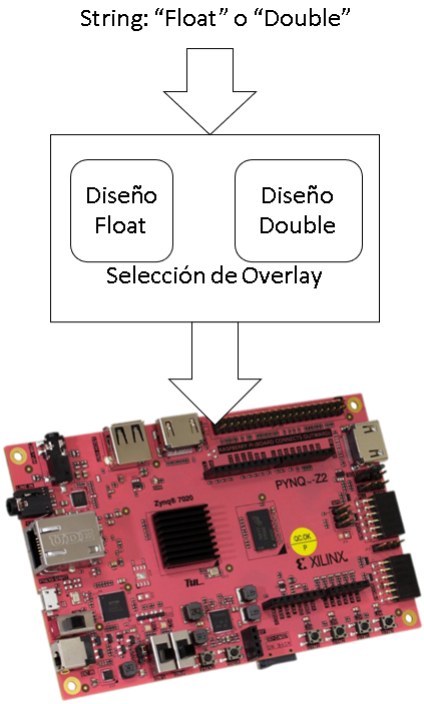


Figura 3.16: Diagrama a bloques de la función Selección Overlay.

Realizar Inversa

Esta función permite la interacción con los componentes del módulo *Inversa* (ver Figura 3.5). En la Figura 3.17 se muestra el diagrama a bloques de la función.

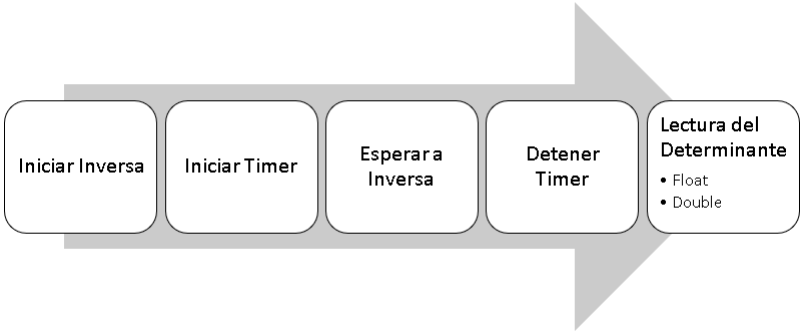


Figura 3.17: Diagrama a bloques para Realizar Inversa.

Para *Iniciar Inversa*, se interactúa con el módulo inversa, se toma a consideración la información que se muestra en la Tabla 3.6 y 3.7, el cual indica las direcciones asociadas al módulo Inversa en ambos Sistemas (*Float* y *Double*), esto, para indicar al módulo cuando iniciar y saber cuando finalizó la operación, también se especifica la dirección para la lectura del determinante.

Tabla 3.6: Información relevante Inversa (Sistema Float).

Dirección	Descripción
0x00	Control signals bit 0 - ap_start (Read/Write/COH) bit 1 - ap_done (Read/COR) bit 2 - ap_idle (Read) bit 3 - ap_ready (Read) bit 7 - auto_restart (Read/Write) others - reserved
0x10	Data signal of ap_return bit 31~0 - ap_return[31:0] (Read)

Tabla 3.7: Información relevante Inversa (Sistema Double).

Dirección	Descripción
0x00	Control signals bit 0 - ap_start (Read/Write/COH) bit 1 - ap_done (Read/COR) bit 2 - ap_idle (Read) bit 3 - ap_ready (Read) bit 7 - auto_restart (Read/Write) others - reserved
0x10	Data signal of ap_return bit 31~0 - ap_return[31:0] (Read)
0x14	Data signal of ap_return // bit 31~0 - ap_return[63:32] (Read)

Un punto importante a remarcar es referente a la lectura del Determinante en el Sistema *Double*, esto porque el valor es de 64 bits y la librería *MMIO* únicamente permite la lectura de datos de 32 bits, por lo tanto el procedimiento es leer los 32 bits menos significativos y posteriormente los 32 bits más significativos para finalmente concatenar ambos valores.

Realizar multiplicación

De manera similar con la función *Realizar Inversa*, la función *Realizar multiplicación* interactúa con los componentes del módulo *Multiplicación* (ver Figura 3.18).

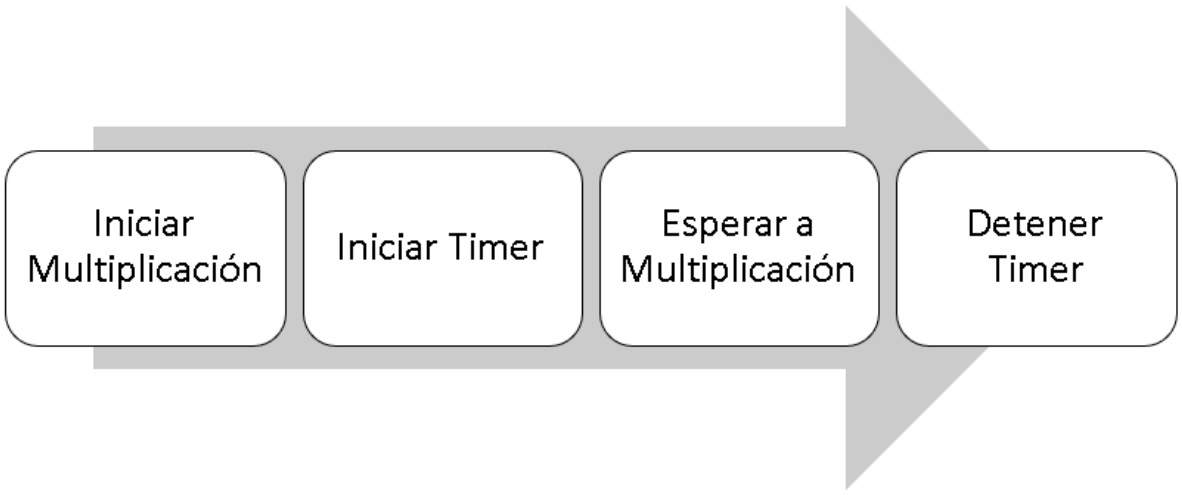


Figura 3.18: Diagrama a bloques para Realizar Multiplicación.

Referente a la interacción con el módulo Multiplicación, únicamente se requieren las direcciones para indicar el inicio de la operación y conocer cuando finaliza la operación, las direcciones se encuentran en la Tabla 3.8.

Tabla 3.8: Información relevante Multiplicación (Sistema Double y Sistema Float)).

Dirección	Descripción
0x00	Control signals bit 0 - ap_start (Read/Write/COH) bit 1 - ap_done (Read/COR) bit 2 - ap_idle (Read) bit 3 - ap_ready (Read) bit 7 - auto_restart (Read/Write) others - reserved

Timer

Para la interacción del Timer se utilizan dos funciones, *Iniciar Timer* y *Detener Timer*.

- Iniciar Timer: Realiza el proceso de cargar el valor al Timer e iniciar, los pasos son los siguientes[3]:

  1. Cargar el valor del contador en el registro *TLR0*, se carga con el valor máximo ($0xFFFFFFFF$).
  2. Poner en alto y después borrar el bit *LOAD0*, para activar la carga.

[3]De acuerdo a la Documentación del AXI TIMER.

3. Poner en alto *MDT0* para colocar el Timer en modo Generar (es decir, a partir del valor cargado el conteo puede ser descendente o ascendente).

4. Poner en alto *UDT0* para seleccionar el conteo descendente.

5. Poner en alto *ENT0* para iniciar el Timer.

- Detener Timer: Una vez que el módulo (*Multiplicación o Inversa* termine la operación, se tiene que detener el Timer y obtener el tiempo de ejecución relacionado con el valor del contador, los pasos son los siguientes:

1. Para detener el contador, se coloca el registro *ENT0* en bajo.

2. *S*e lee el valor del contador que se encuentra en la dirección $0x08$.

3. Se calcula de acuerdo con la ecuación 3.3.

Llenar Matriz

Está función llena los bloques de memoria asociados a la matriz, se debe especificar el bloque de memoria y los valores (en este caso, los 9 elementos de la matriz). Es importante mencionar la diferencia cuando se mandan datos tipo *Float* (32 bits) y tipo *Double* (64 bits), esto porque la clase *MMIO* únicamente puede mandar datos de 32 bits, para los datos tipo *Double* se escriben primero los 32 bits menos significativos, se incrementa la dirección y se concatenan los 32 bits más significativos formando así la palabra de 64 bits.

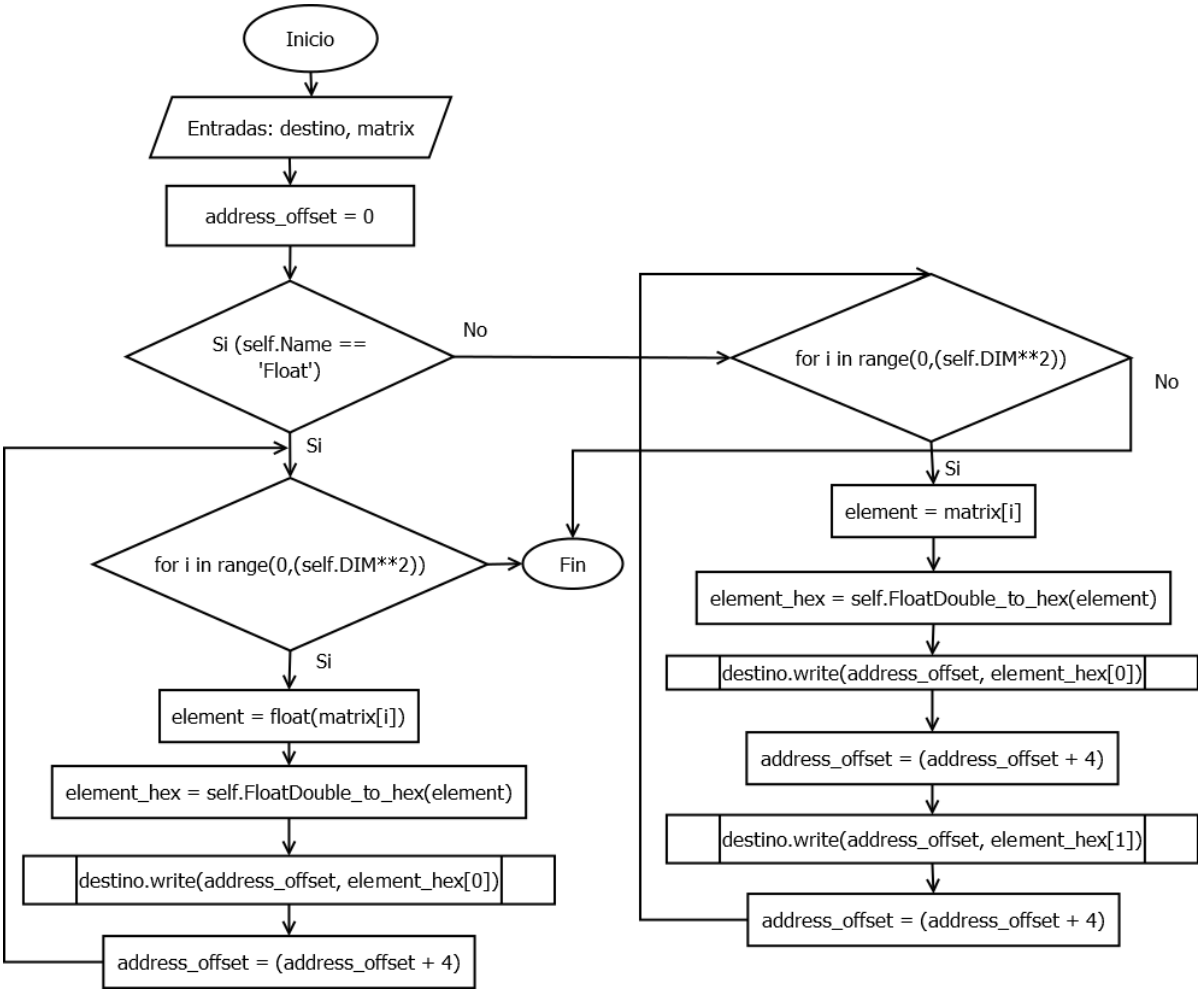


Figura 3.19: Diagrama de flujo de FillMatrix.

Leer Matriz

Está función se encarga de leer los valores almacenados en los bloques de memoria, de igual forma que la función anterior, para los datos de 64 bits se tiene que realizar dos lecturas (para los 32 bits menos significativos y los 32 bits más significativos), en la Figura 3.20 se muestra el diagrama a bloques de la función de manera simplificada.

Figura 3.20: Diagrama a bloques de ReadMatrix.

El código completo referente a las funciones anteriores se encuentran en el Anexo C.

3.4.2. Pruebas

Para probar ambos sistemas se realizaron dos tipos de pruebas: la prueba normal, donde se puede determinar los nueve valores de la matriz (ver Figura 3.21); y una segunda: donde los valores de la matriz son ajustados a diferentes números de dígitos, esto para ver los efectos provocados por el redondeo.

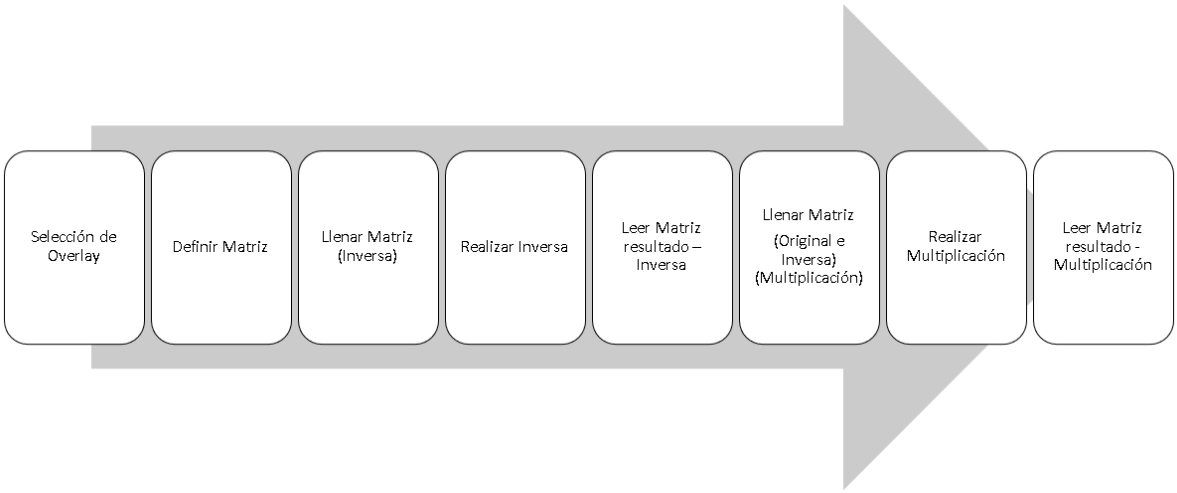


Figura 3.21: Diagrama a bloques de la prueba sin redondeo.

La prueba con redondeo se hace, similarmente, sobre un diseño a la vez (Sistema *Float* o Sistema *Double*), sin embargo, las operaciones se hacen sobre dos matrices en especifico, una truncada $A_T = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & \epsilon_T \end{pmatrix}$ y una matriz no truncada, $A = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & \epsilon \end{pmatrix}$, ambas matrices tienen un elemento ϵ el cual es el valor que será truncado, es importante mencionar que este truncamiento es por software.

La inversa se obtiene de la matriz truncada (A_T) y posteriormente la multiplicación se realiza con la matriz normal (A) y la matriz inversa obtenida. Básicamente se mantiene el mismo procedimiento que se muestra en la Figura 3.21; sin embargo, este proceso está en un ciclo *for*, en donde se redondea a ϵ a los siguientes dígitos: $0, 2, 4, 8, 12, 16, 24$ y 32, este nuevo valor será ϵ_T.

Capítulo 4

Resultados

4.1. Resultados de la Síntesis

La cantidad de recursos utilizados en la implementación se obtuvo a través del *Report Utilization*[24], que se organizaron siguiendo estas métricas:

- LUTs: Componente básico de un FPGA, en este caso del PL, este elemento esencialmente es una tabla de verdad, en donde diferentes combinaciones de entradas devuelven diferentes valores de salida [22].
- Logic LUTs: Se refiere a las LUTs utilizadas para realizar expresiones lógicas.
- LUTRAMs: Los elementos tipo LUT, son implementados como recursos RAM síncrono [19].
- SRLs: Son los elementos tipo LUT utilizados para realizar la lógica de corrimiento (Shift Register Logic) [23].
- FFs: Los Flip - Flops, su estructura básica incluye: Dato de entrada, reloj, reset y dato de salida.
- RAMB: Bloque de construcción básico utilizado para todas las configuraciones de bloques RAM [24].
- DSP: Es una Unidad Lógica Aritmética (ALU) especializada en realizar operaciones orientadas hacia el *Digital Signal Processing*, la unidad está embebida en la sección del PL del SoC. Está compuesta por 3 bloques principales conectados en cadena:
  - Unidad sumadora/restadora.
  - Multiplicador.
  - Sumador/Restador/Acumulador Final.

4.1.1. Recursos Sistema *Float*

En el Tabla 4.1 se encuentran listados los recursos utilizados por cada componente mostrado en la Figura 3.4, para el diseño Sistema Float (ver Figura 3.3b).

Tabla 4.1: Recursos utilizados por el diseño "Sistema Float".

Métrica / Módulo	TOTAL LUTs	LOGIC LUTs	LUTRAMs	SRLs	FFs	RAMB	DSP
TOP	5186	4916	32	238	5935	10	10
AXI TIMER	295	295	0	0	240	0	0
inversa	1759	1671	32	53	1892	4	5
Multiplicación	1049	1019	0	30	1154	6	5
Processing System	0	0	0	0	0	0	0
ps7 axi perip	2070	1916	0	154	2616	0	0
rst ps7	16	15	0	1	33	0	0

También se muestran los recursos por jerarquías: inversa (ver Figura 3.7) en el Tabla 4.2 y multiplicación (ver Figura 3.12) en el Tabla 4.3.

Tabla 4.2: Recursos utilizados por los módulos de la jerarquía inversa.

Métrica / Módulo	TOTAL LUTs	LOGIC LUTs	LUTRAMs	SRLs	FFs	RAMB	DSP
Inversa_0	1287	1222	32	33	1394	0	5
Memoria_Inversa	15	11	0	4	20	4	0
control_MI	454	438	0	16	478	0	0

Tabla 4.3: Recursos utilizados por los módulos de la jerarquía multiplicación.

Métrica / Módulo	TOTAL LUTs	LOGIC LUTs	LUTRAMs	SRLs	FFs	RAMB	DSP
Memoria_Multiplicación	23	17	0	6	30	6	0
control_MM	680	656	0	24	717	0	0
matrizMult_0	346	346	0	0	407	0	5

4.1.2. Recursos Sistema *Double*

De manera similar los recursos utilizados por cada bloque del diseño Sistema *Double* se encuentran en el Tabla 4.4.

Tabla 4.4: Recursos utilizados por el Sistema Double.

Métrica / Módulo	TOTAL LUTs	LOGIC LUTs	LUTRAMs	SRLs	FFs	RAMB	DSP
TOP	8864	8507	64	293	9863	20	28
AXI TIMER	297	297	0	0	240	0	0
inversa	4968	4804	64	100	5246	8	14
Multiplicación	1539	1501	0	38	1726	12	14
Processing System	0	0	0	0	0	0	0
ps7 axi perip	2046	1892	0	154	2618	0	0
rst ps7	16	15	0	1	33	0	0

También, se especifican los recursos utilizados para cada módulo que conforman a la jerarquía inversa (ver Tabla 4.5) y multiplicación (ver Tabla 4.6).

Tabla 4.5: Recursos utilizados en la jerarquía inversa, especificados por submódulo.

Métrica / Módulo	TOTAL LUTs	LOGIC LUTs	LUTRAMs	SRLs	FFs	RAMB	DSP
Inversa_0	4493	4349	64	80	4742	0	14
Memoria_Inversa	15	11	0	4	20	8	0
control_MI	460	444	0	16	484	0	0

Tabla 4.6: Recursos utilizados en la jerarquía Multiplicación, especificados por submódulo.

Métrica / Módulo	TOTAL LUTs	LOGIC LUTs	LUTRAMs	SRLs	FFs	RAMB	DSP
Memoria_Multiplicación	23	17	0	6	30	12	0
control_MM	688	664	0	24	726	0	0
matrizMult_0	828	820	0	8	970	0	14

4.1.3. Comparación de Recursos

En la Tabla 4.7 se muestra la comparación de recursos entre el Sistema *float* y el Sistema *double* en total.

Tabla 4.7: Comparación de recursos utilizados por ambos sistemas.

Diseño / Métrica	Sistema *Float*	Sistema *Double*
TOTAL LUTs	5186	8864
LOGIC LUTs	4916	8507
LUTRAMs	32	64
SRLs	238	293
FFs	5935	9863
RAMB	10	20
DSP	10	28

También, se muestra la comparación de los recursos utilizados por el bloque inversa (ver Figura 3.7) en la Tabla 4.8.

Tabla 4.8: Comparación de recursos utilizados por el bloque jerárquico inversa.

Diseño / Métrica	inversa (Sistema *Float*)	inversa (Sistema *Double*)
TOTAL LUTs	1759	4968
LOGIC LUTs	1671	4804
LUTRAMs	32	64
SRLs	53	100
FFs	1892	5246
RAMB	4	8
DSP	5	14

En la Tabla 4.9 se muestra la comparación de los recursos utilizados por el bloque jerárquico Multiplicación (ver Figura 3.12).

Tabla 4.9: Comparación de recursos utilizados por el bloque jerárquico Multiplicación.

Diseño / Métrica	Multiplicación (Sistema Float)	Multiplicación (Sistema Double)
TOTAL LUTs	1049	1539
LOGIC LUTs	1019	1501
LUTRAMs	0	0
SRLs	30	38
FFs	1154	1726
RAMB	6	12
DSP	5	14

También se muestra la comparación de recursos utilizados por los bloques obtenidos de la Síntesis de Alto Nivel, *Inversa_0* (ver Figura 3.8) y *matrizMult_0* (ver Figura 3.13).

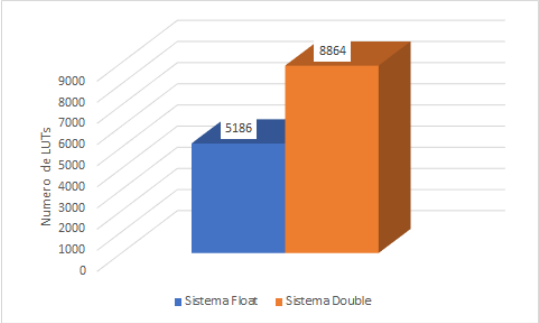
Tabla 4.10: Comparación de recursos utilizados por el bloque Inversa_0.

Diseño / Métrica	Inversa_0 (Sistema Float)	Inversa_0 (Sistema Double)
TOTAL LUTs	1287	4493
LOGIC LUTs	1222	4349
LUTRAMs	32	64
SRLs	33	80
FFs	1394	4742
RAMB	0	0
DSP	5	14

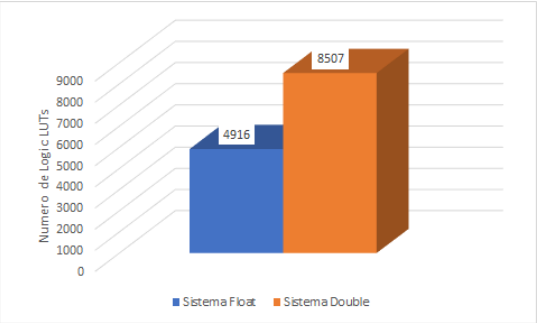
Tabla 4.11: Comparación de recursos utilizados por el bloque matrizMult_0.

Diseño / Métrica	matrizMult_0 (Sistema Float)	matrizMult_0 (Sistema Double)
TOTAL LUTs	346	828
LOGIC LUTs	346	820
LUTRAMs	0	0
SRLs	0	8
FFs	407	970
RAMB	0	0
DSP	5	14

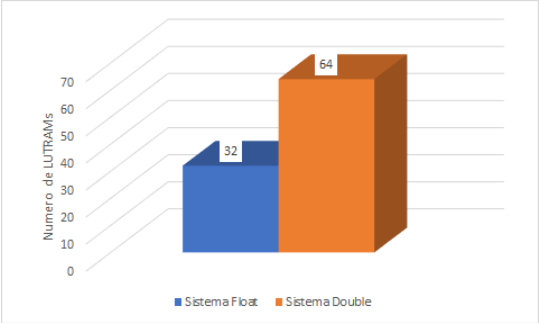
De manera adicional, para que el lector tenga un recurso visual rápido de los resultados se presentan las gráficas que se encuentran las Figuras 4.1, 4.2 y 4.3. Para complementar estos recursos gráficos también se muestran ambos diseños ajustados en el SoC (ver Figura 4.4).



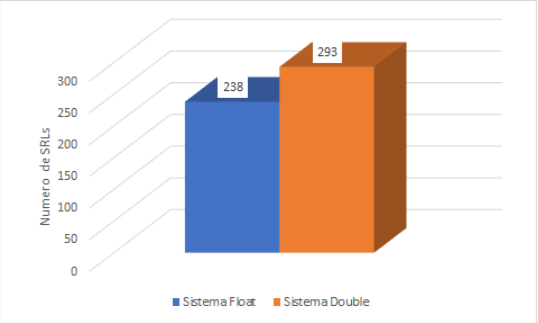
(a) Comparación Total LUTs.



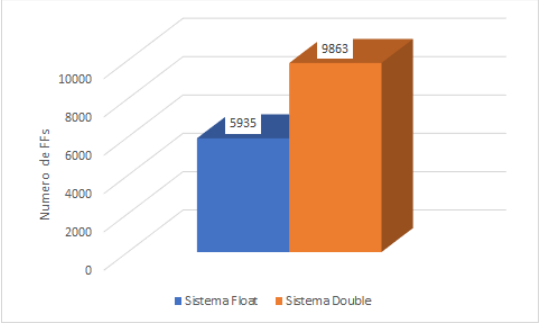
(b) Comparación Logic LUTs.



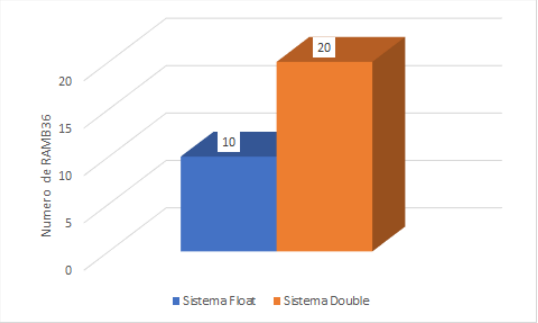
(c) Comparación LUTRAMs.



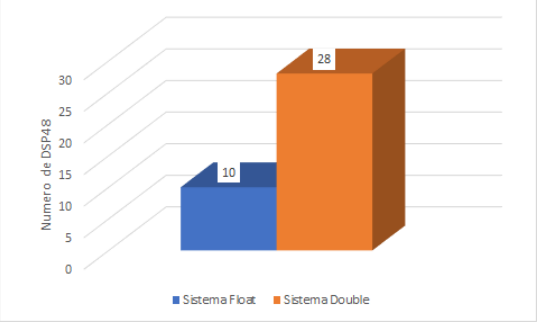
(d) Comparación SRLs.



(e) Comparación FFs.

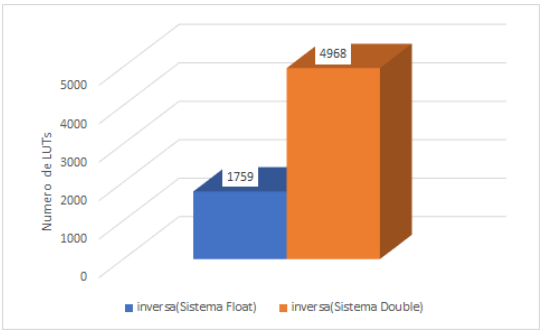


(f) Comparación RAMB.

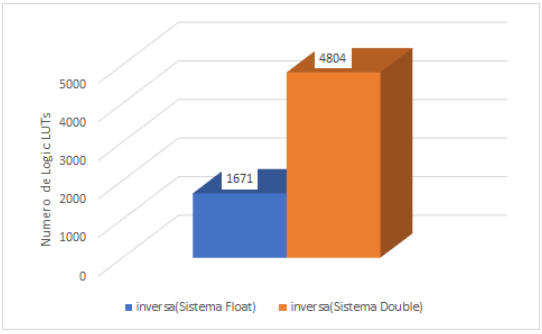


(g) Comparación DSP.

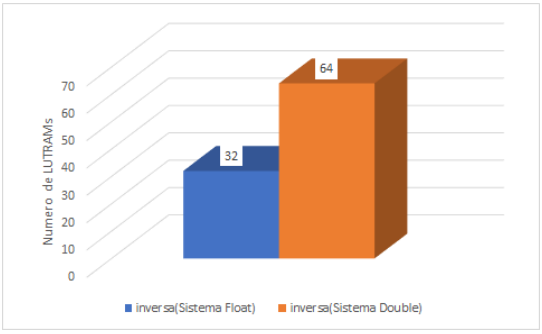
Figura 4.1: Comparación de los recursos utilizados por diseño.



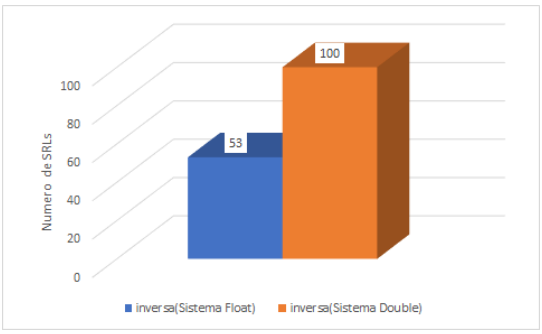
(a) Comparación Total LUTs.



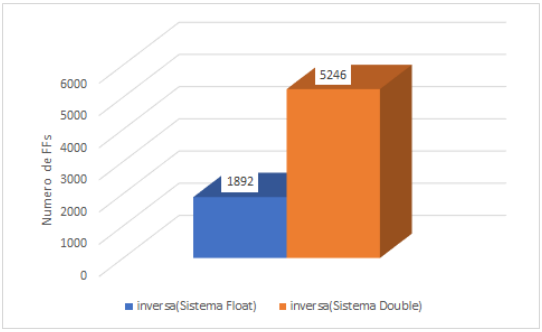
(b) Comparación Logic LUTs.



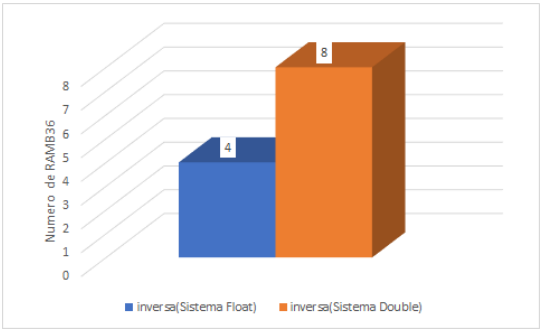
(c) Comparación LUTRAMs.



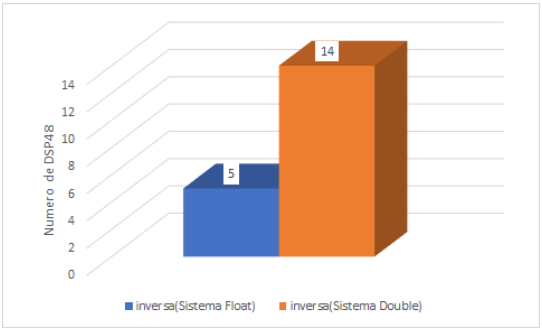
(d) Comparación SRLs.



(e) Comparación FFs.

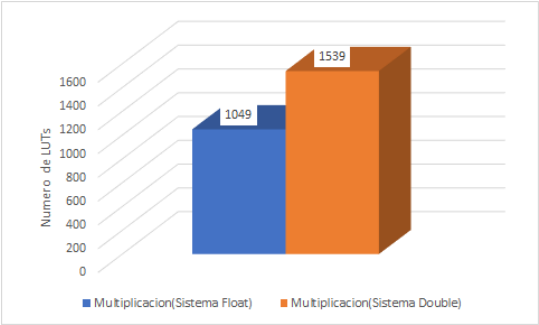


(f) Comparación RAMB.

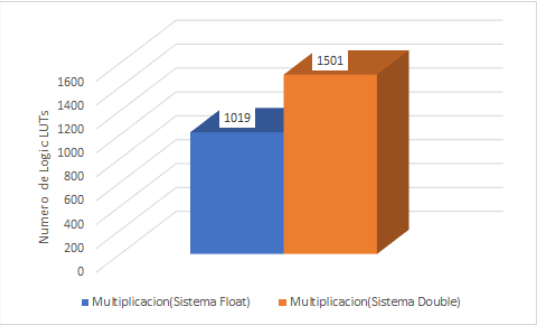


(g) Comparación DSP.

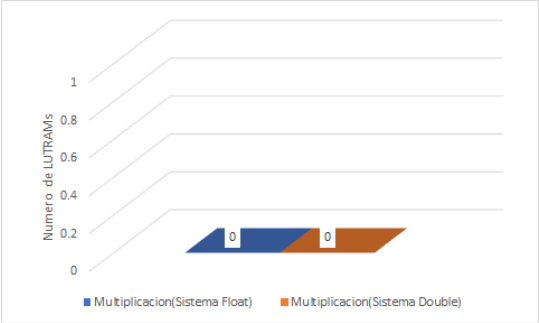
Figura 4.2: Comparación de los recursos utilizados en la jerarquía inversa.



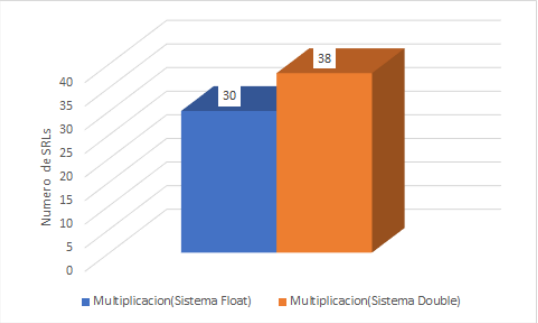
(a) Comparación Total LUTs.



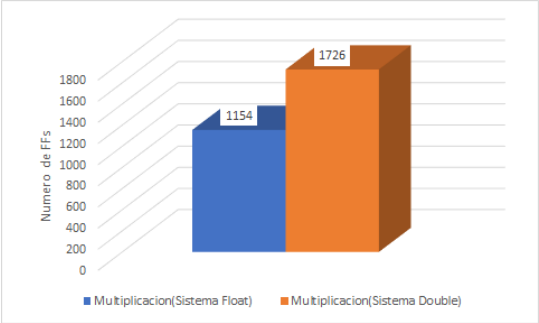
(b) Comparación Logic LUTs.



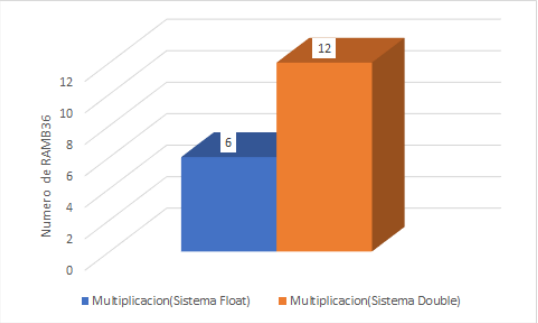
(c) Comparación LUTRAMs.



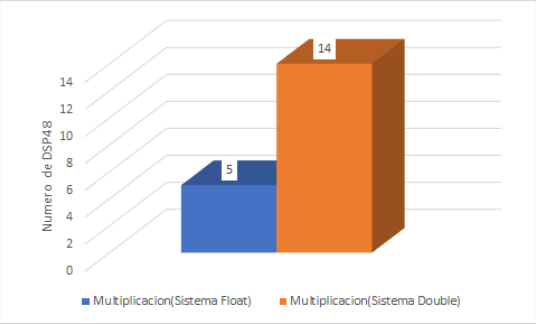
(d) Comparación SRLs.



(e) Comparación FFs.

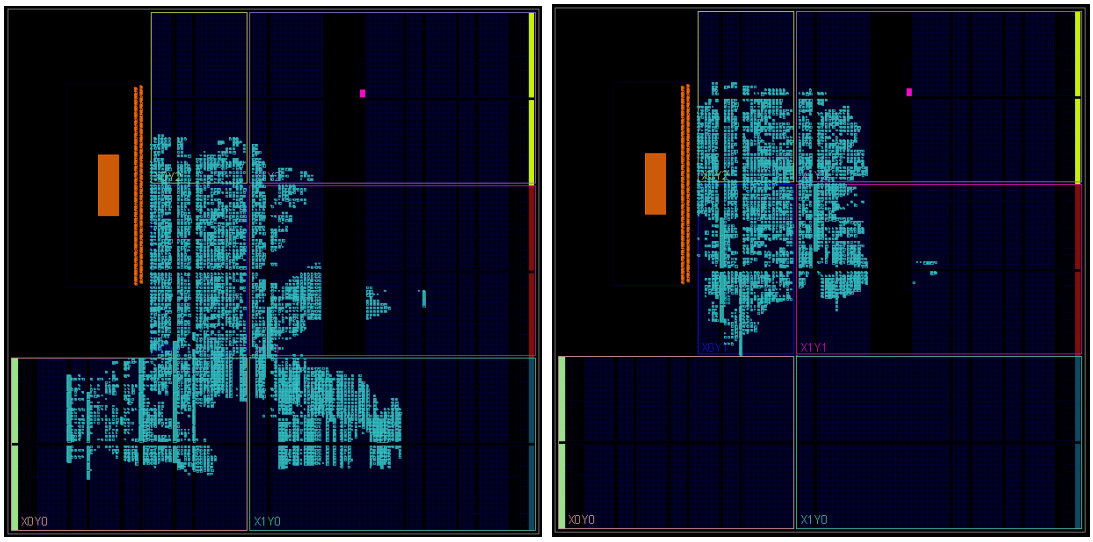


(f) Comparación RAMB.



(g) Comparación DSP.

Figura 4.3: Comparación de los recursos utilizados en la jerarquía multiplicación.



(a) Sistema Double. (b) Sistema Float.

Figura 4.4: Diseños ajustados en el SoC Zynq XC7Z020.

En la Tabla 4.12 se muestra el factor de crecimiento del *Sistema Double* respecto el *Sistema Float*, en donde el recurso que más aumentó fue el *DSP48*, el cual creció un 2.8 veces más, por el contrario el recurso que menos aumentó fue el *SRLs*. Otro recurso que tuvo un aumentó significativo fue el *RAMB36*. El crecimiento de los recursos de *DSP48* y *RAMB36* se debe al aumentó de la palabra, de 32 bits a 64 bits y a la arquitectura utilizada, la cual entre una de sus características es la utilización de bloques de memoria para almacenar las matrices de entrada y salida de los bloques de operación.

Tabla 4.12: Factor de aumentó de recursos del Sistema Double respecto al Sistema Float.

Métrica	Factor de crecimiento
Total LUTs	1.7
LOGIC LUTs	1.7
LUTRAMs	2.0
SRLs	1.2
FFs	1.7
RAMB36	2.0
DSP48	2.8

4.2. Desempeño de Ambos Sistemas.

Para mostrar el desempeño de ambos Sistemas se toman en cuenta las pruebas mencionas en la Sección 3.4.2, en donde particularmente se utilizarán matrices mal condicio-

nadas, para ello se propone una familia de matrices que dependen de un parámetro ε, a saber, $A = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & \varepsilon \end{pmatrix}$.

Es importante resaltar que el número de condición de la matriz A_ε depende precisamente del valor que tome el parámetro ε. Si éste toma valores pequeños, el número de condición será grande.

Una pregunta que surge es la siguiente ¿Cómo son los resultados de la implementación en el FPGA con respecto a su implementación en una computadora personal? Para responder esta pregunta, debemos observar en primera instancia que una computadora personal podría contar con mayores recursos pues realizar más funciones que un dispositivo programable. Éste último puede estar pensado para una tarea específica y/o limitado en recursos. Se realizaron pruebas y se compararon los resultados en los dos sistemas, a saber, en Python y en el FPGA utilizando la arquitectura propuesta. En las Tablas 4.13-4.18 se muestran los resultados de las diferentes pruebas de donde se obtienen los siguientes resultados: 1. La arquitectura diseñada obtiene un resultado mejor en la prueba de redondeo con 8 números decimales. 2. Los resultados entre el diseño Sistema Double y los obtenidos por Python son similares esto debido a que Python utiliza datos de tipo Double por defecto.

Tabla 4.13: Resultados obtenidos con $\epsilon = 0.011$, *cond*$(A) = 335.962$

Característica \ Diseño	Sistema Float	Sistema Double
A^{-1}	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -90.90908813476562 \\ 0.0 & 0.0 & 90.90908813476562 \end{pmatrix}$	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -90.90909090909092 \\ 0.0 & 0.0 & 90.90909090909092 \end{pmatrix}$
Determinante	0.02199999988079071	0.022
$t_{e-inversa}$	0.00030404 segundos	0.00029593 segundos
AA^{-1}	$\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.9999999403953552 \end{pmatrix}$	$\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$
$t_{e-multiplicación}$	0.00028458 segundos	0.00028072 segundos

Tabla 4.15: Resultados obtenidos con Python.

	$\epsilon = 0.011$	$\epsilon = 0.000101$
A^-1	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -90.90909091 \\ 0.0 & 0.0 & 90.90909091 \end{pmatrix}$	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -9900.9901 \\ 0.0 & 0.0 & -9900.9901 \end{pmatrix}$
Determinante	0.02200000000000001	0.00020199999999999998
AA^-1	$\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$	$\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$

Tabla 4.16: Comparación entre diferente cantidad de dígitos de truncamiento utilizando el "Sistema Float".

Dígitos de Truncamiento	8	16	32
ϵ_{aprox}	$5.99999978589949 \times 10^{-8}$	$6.666666507726404 \times 10^{-8}$	$6.666666507726404 \times 10^{-8}$
$cond(A)$	6×10^7	6×10^7	6×10^7
$cond(A_{aprox})$	6.1592×10^7	5.5433×10^7	5.5433×10^7
A^{-1}_{aprox}	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -16666667.0 \\ 0.0 & 0.0 & 16666667.0 \end{pmatrix}$	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -15000000.0 \\ 0.0 & 0.0 & 15000000.0 \end{pmatrix}$	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -15000000.0 \\ 0.0 & 0.0 & 15000000.0 \end{pmatrix}$
$det(A_{aprox})$	1.199999957179898e-07	1.3333333015452808e-07	1.3333333015452808e-07

Tabla 4.14: Resultados obtenidos con $\epsilon = 0.000101$, $cond(A) = 36589.288$

	Sistema Float	Sistema Double
A^-1	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -9900.990234375 \\ 0.0 & 0.0 & 9900.990234375 \end{pmatrix}$	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -9900.990099009901 \\ 0.0 & 0.0 & 9900.990099009901 \end{pmatrix}$
Determinante	0.00020199999562464654	0.000202
$t_e - inversa$	0.00029604 segundos	0.00029286 segundos
AA^-1	$\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$
$t_e - multiplicación$	0.00028108 segundos	0.00028236 segundos

Tabla 4.17: Comparación entre diferente cantidad de dígitos de truncamiento utilizando el "Sistema Double".

Dígitos de Truncamiento	8	16	32
ϵ_{aprox}	6×10^{-8}	$6.66666666 \times 10^{-8}$	$6.666666666666667 \times 10^{-8}$
$cond(A)$	6×10^{7}	6×10^{7}	6×10^{7}
$cond(A_{aprox})$	6.1592×10^{7}	5.5433×10^{7}	5.5433×10^{7}
A^{-1}_{aprox}	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -16666666.666666668 \\ 0.0 & 0.0 & 16666666.666666668 \end{pmatrix}$	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -15000000.015 \\ 0.0 & 0.0 & 15000000.015 \end{pmatrix}$	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -15000000.0 \\ 0.0 & 0.0 & 15000000.0 \end{pmatrix}$
$det(A_{aprox})$	1.2×10^{-7}	$1.333333332 \times 10^{-7}$	$1.3333333333333334 \times 10^{-7}$

Tabla 4.18: Comparación entre diferente cantidad de dígitos de truncamiento utilizando Python.

Dígitos de Truncamiento	8	16	32
$\epsilon_a prox$	$7 \times 10^{-}8$	$6.66666667 \times 10^{-}8$	$6.666666666666667 \times 10^{-}8$
$cond(A)$	6×10^{7}	6×10^{7}	6×10^{7}
$cond(A_a prox)$	52793116.14350214	55432771.922960855	55432771.95067723
$A^{-}1_a prox$	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -1.42857143 \times 10^{7} \\ 0.0 & 0.0 & 1.42857143 \times 10^{7} \end{pmatrix}$	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -15000000.0 \\ 0.0 & 0.0 & 15000000.0 \end{pmatrix}$	$\begin{pmatrix} 0.5 & -0.5 & 0.0 \\ 0.0 & 1.0 & -15000000.0 \\ 0.0 & 0.0 & 15000000.0 \end{pmatrix}$

Capítulo 5

Conclusiones

- A partir de los resultados numéricos, se puede apreciar que al realizar el cálculo de la inversa se aprecia una diferencia sustancial entre ambos tipos de datos, donde el *Sistema Double* arrojó resultados mejores, sin embargo la cantidad de recursos se disparó al doble en muchos elementos.

- Si el valor numérico es redondeado a una cantidad x de dígitos, se puede apreciar que al redondear a 16 números decimales se obtienen resultados aceptables para ambos casos, por lo que el *Sistema Float*, tiene un desempeño mejor a comparación del *Sistema Double*, sobre todo tomando en cuenta los recursos utilizados por ambos Sistemas. Respecto a Python se aprecian unos resultados peores cuando se redondea a 8 números decimales pero para 16 y 32 se obtienen resultados similares a los conseguidos con la arquitectura.

- El utilizar un algoritmo no especializado como la Inversa por el Método de la adjunta, ilustra de buena manera el desempeño que se obtiene al utilizar diferentes tipos de datos frente a matrices mal condicionadas, además de como esta elección en los tipos de datos se ve reflejada en el consumo de recursos.

- Particularmente, se encuentra que la Síntesis de Alto Nivel es una herramienta adecuada en la creación de los módulos, ya que reduce el tiempo invertido en el proyecto comparado con el diseño tradicional utilizando Lenguajes de Descripción de Hardware.

- El uso del *Framework* permite reducir la complejidad de la programación y la interacción con Sistemas complejos, al apoyarse de las diversas librerías de Python.

- Es importante siempre realizar pruebas a tanto a nivel de unidades, integración y sistema, ya que permite corroborar que se cumplan con los requerimientos funcionales, de manera particular se encontraron problemas relacionados a las unidades (creadas con HLS) esto debido a malas prácticas al momento de programar en

lenguaje C, la recomendación es seguir las buenas prácticas de acuerdo a la documentación de la versión del software a usar.

Bibliografía

[1] Confluence Wiki Admin. *HSI debugging and optimization techniques*. 2020. URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841693/HSI+debugging+and+optimization+techniques.

[2] Ignacio Algredo-Badillo y col. "An FPGA-based analysis of trade-offs in the presence of ill-conditioning and different precision levels in computations". En: *PLOS ONE* 15 (jun. de 2020), págs. 1-26. DOI: 10.1371/journal.pone.0234293.

[3] F. Bruno. *FPGA Programming for Beginners: Bring Your Ideas to Life by Creating Hardware Designs and Electronic Circuits with SystemVerilog*. Packt Publishing, 2021.

[4] Veena S. Chakravarthi. *A Practical Approach to VLSI System on Chip (SoC) Design: A Comprehensive Guide*. 1st ed. 2020. Springer International Publishing, 2020.

[5] Louise H. Crockett y col. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. 1st Edition. Strathclyde Academic Media, 2014.

[6] Stephen Evanczuk. *Cree y programe diseños basados en un FPGA de manera rápida con los cuadernos de Python y Jupyter*. 2019. URL: https://www.digikey.com.mx/es/articles/build-and-program-fpga-based-designs-quickly-python-jupyter-notebooks.

[7] Python Software Foundation. *Python 2.7.18 documentation*. 2020. URL: https://docs.python.org/2.7/library/ctypes.html?highlight=ctypes#module-ctypes.

[8] Python Software Foundation. *Python 2.7.18 documentation*. 2020. URL: https://docs.python.org/2.7/library/struct.html?highlight=struct#module-struct.

[9] Yangkang Chen Guoning Wu Sergey Fomel. *Prony Method*. 2020. URL: https://reproducibility.org/RSF/book/tccs/npm/paper_html/node4.html.

[10] Valery Sklyarov Iouliia Skliarova. *FPGA-BASED Hardware Accelerators*. 1st Edition. Lecture Notes in Electrical Engineering 566. Springer International Publishing, 2019.

[11] J. Rey Cabezas J. Infante del Río. *Métodos Numéricos, Teoría, problemas y prácticas con MATLAB*. 4ta Edición. Ciencia y Técnica. Pirámide, 2002.

[12] M. S. Pedersen K. B. Petersen. *The Matrix Cookbook*. 2012. URL: http://www2.compute.dtu.dk/pubdb/pubs/3274-full.html.

[13] Microsoft. *C Pragmas*. 2021. URL: https://docs.microsoft.com/en-us/cpp/c-language/c-pragmas?view=msvc-170.

[14] Benjamin F. Harding RC Cofer. *Rapid System Prototyping with FPGAs: Accelerating the design process*. Embedded Technology. Newnes, 2005.

[15] Edward Rothwell y B. Drachman. "Unified approach to solving ill-conditioned matrix problems". En: *International Journal for Numerical Methods in Engineering* 28 (mar. de 1989), págs. 609-620. DOI: 10.1002/nme.1620280309.

[16] Florian Stock y Andreas Koch. "A Fast GPU Implementation for Solving Sparse Ill-Posed Linear Equation Systems". En: *Parallel Processing and Applied Mathematics*. Ed. por Roman Wyrzykowski y col. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, págs. 457-466.

[17] V.Ya. ArseninA.N. Tikhonov. *Ill-posed problems*. 2012. URL: http://encyclopediaofmath.org/index.php?title=Ill-posed_problems&oldid=25322.

[18] Marylin Wolf. *Computers as Components: Principles of Embedded Computing System Design*. 4th Edition. Engineering professional collection. Elsevier/Morgan Kaufmann, 2017.

[19] Xilinx. *7 Series FPGAs Configurable Logic Block*. English. Ver. v1.8. Xilinx. 27 de oct. de 2016. 74 págs.

[20] Xilinx. *AXI Timer v2.0:LogiCore IP Product Guide*. English. Ver. PG079. Xilinx. 37 págs.

[21] Xilinx. *PYNQ: Python productivity for Xilinx platforms*. 2021. URL: https://pynq.readthedocs.io/en/latest/pynq_libraries/mmio.html?highlight=MMIO.

[22] Xilinx. *SDAccel Development Environment Help*. 2018. URL: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/yeo1504034293627.html.

[23] Xilinx. *UltraScale Architecture Configurable Logic Block*. English. Ver. v1.5. Xilinx. 28 de feb. de 2017. 58 págs.

[24] Xilinx. *UltraScale Architecture Memory Resources*. English. Ver. v1.13. Xilinx. 24 de sep. de 2021. 139 págs.

[25] Xilinx. *Using Xilinx SDK*. 2018. URL: https://www.xilinx.com/html_docs/xilinx2018_1/SDK_Doc/SDK_concepts/concept_fpgabitstream.html.

[26] Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis*. English. Ver. UG902 (v2019.1). Xilinx. 12 de jul. de 2019. 592 págs.

[27] Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis*. English. Ver. 2.5. Xilinx. 27 de oct. de 2020. 345 págs.

[28] Tanner Young-Schultz y col. "Using OpenCL to Enable Software-like Development of an FPGA-Accelerated Biophotonic Cancer Treatment Simulator". En: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. Seaside, CA, USA: Association for Computing Machinery, 2020, págs. 86-96. ISBN: 9781450370998.

Apéndice A

Código en C: Inversa

En Listing A.1 se muestra el código integro en lenguaje C de la función Inversa del cual se realizó la síntesis utilizando Vivado HLS y en Listing A.2 se encuentra el archivo header utilizado, en este caso para el Sistema Float.

Listing A.1: Función en C de la inversa de una matriz(Método de la adjunta) utilizado en Vivado HLS.

```
#include "multiplicacion.h"

float Inversa(Matriz Orig[DIM][DIM], Matriz inver[DIM][DIM]){
#pragma HLS INTERFACE bram port=inver
#pragma HLS INTERFACE bram port=Orig
#pragma HLS INTERFACE s_axilite port=return bundle=inversa
        deter d_t1=0;
        deter d_t2=0;
        deter d_t3=0;
        deter d=0;
        deter invt1=0;
        deter invt2=0;
        deter invt3=0;
        Matriz m[DIM][DIM];

        for (int col=0;col<DIM;col++){
                for(int fil=0;fil<DIM;fil++){
                        m[col][fil]=Orig[col][fil];
                }

        }
```

```
        loop_determinante:for(int i = 0; i < DIM; i++){
        d_t1=m[1][(i+1)%3] * m[2][(i+2)%3];
        d_t2= m[1][(i+2)%3] * m[2][(i+1)%3];
        d_t3=m[0][i] *(d_t1 - d_t2);
        d=d + d_t3;

  }

  {
      loop_inversa1:for(int i = 0; i < DIM; i++){
        loop_inversa0:for(int j = 0; j < DIM; j++){
                invt1 = m[(j+1)%3][(i+1)%3] * m[(j+2)%3][(i+2)
                    %3];
                invt2 = m[(j+1)%3][(i+2)%3] * m[(j+2)%3][(i+1)
                    %3];
                invt3 = invt1 - invt2;
                inver[i][j]=invt3/ d;
        }
     }
  }
  return d;
}
```

Listing A.2: Codigo del archivo Header para la función de la inversa.

```
#ifndef _MatInversa_
#define _MatInversa_



//En typedef se cambian "float" por "double" para
//el Sistema Double
typedef float Matriz;
typedef float deter;

#define  DIM 3

#include <cmath>
Matriz Inversa(Matriz Orig[DIM][DIM], Matriz inver[DIM][DIM]);

#endif
```

Apéndice B

Código en C: Multiplicación

Listing B.1: Función en C de la multiplicación de matrices utilizado en Vivado HLS.

```
#include "multiplicacion.h"

void matrizMult(const dtype2 A[DIM][DIM],const dtype2 B[DIM][
    DIM],dtype2 C[DIM][DIM]){

#pragma HLS INTERFACE bram port = C
#pragma HLS INTERFACE bram port = B
#pragma HLS INTERFACE bram port = A
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl_bus

dtype2 sum=0;
 L1:for (int ia = 0; ia < DIM; ++ia){
         L2:for (int ib = 0; ib < DIM; ++ib){
                 sum = 0;
                 L3:for (int id = 0; id < DIM; ++id) {
                         sum += A[ia][id] * B[id][ib];
         }
                 C[ia][ib] = sum;
     }
  }
}
```

Listing B.2: Código del archivo Header para la función de la multiplicación.

```
#ifndef __Multiplicacion__
#define __Multiplicacion__
```

```
const int DIM = 3;
#include <cmath>

//En typedef se cambian "float" por "double" para
//el Sistema Double
typedef float dtype2;

void matrizMult(const dtype2 A[DIM][DIM],const dtype2 B[DIM][
    DIM],dtype2 C[DIM][DIM]);

#endif
```

Apéndice C

Código en Python

Listing C.1: Código de Python de las funciones basicas.

```
from pynq import MMIO
from ctypes import *
import struct
import time
from pynq import Overlay

class Test(object):

    def __init__(self,):
        self.mem_size = 4096
        self.AXI_CLOCK_PERIOD = 1/100000000
        self.MAX_COUNT = 4294967295
        self.A = None
        self.B = None
        self.C = None
        self.TestMatrix = [2,1,1,0,1,1,0,0,0]
        self.multiplicacion = None
        self.timer = None
        self.inversa = None
        self.DIM = 3
        self.Orig =None
        self.Name = None

    def Overlay(self,Tipo):
        self.Name = Tipo
        if (self.Name =="Float"):
            overlay = Overlay("FloatDesign.bit")
        else:
```

```
            overlay = Overlay("DoubleDesign.bit")
        overlay.download()
        self.multiplicacion = overlay.multiplicacion.
           matrizMult_0
        self.timer = overlay.axi_timer_0
        self.inversa = overlay.inversa.Inversa_0
        Orig_Base_address = 0x40000000
        Inver_Base_address =0x42000000
        A_Base_address = 0x44000000
        B_Base_address = 0x46000000
        C_Base_address = 0x48000000
        self.A = MMIO(A_Base_address, self.mem_size)
        self.B = MMIO(B_Base_address, self.mem_size)
        self.C = MMIO(C_Base_address, self.mem_size)
        self.Orig = MMIO(Orig_Base_address, self.mem_size)
        self.inver = MMIO(Inver_Base_address, self.mem_size)



def PrintMatrix (self,matrix):
    print(f"{matrix[0]} {matrix[1]} {matrix[2]}")
    print(f"{matrix[3]} {matrix[4]} {matrix[5]}")
    print(f"{matrix[6]} {matrix[7]} {matrix[8]}")



def ReadMatrix (self,matrix,CONDITION):
    Data = [0] * (self.DIM**2)
    j=0
    u=0
    cadena = " "
    if (self.Name == "Float") :
        for i in range(0,self.DIM**2):
            received_data = matrix.read(i*4)
            Data[i] =  self.hex_to_FloatDouble(hex(
               received_data))
    else:
        for i in range(0,17,2):
            received_data_low = matrix.read(i*4)
            received_data_upper = matrix.read((i+1)*4)
            new_low = hex(received_data_low)
```

```
                new_upper = hex(received_data_upper)
                if (len(new_low) < 10):
                    size = len(new_low)
                    new_low = new_low[0:2]+"0"*(10-size)+
                        new_low[2:]
                if (len(new_upper) < 10):
                    size = len(new_upper)
                    new_upper = new_upper[0:2]+"0"*(10-size)+
                        new_upper[2:]
                numero = new_upper + new_low[2:]
                Data[u] =  self.hex_to_FloatDouble(numero)
                u = u+1
        for l in Data:
            print(l,end='    ')
            if j==2 or j==5 or j==8 :
                print("\n")
            j=j+1
        if(CONDITION==1):
            return Data

    def FillMatrix (self,destino,matrix):
        address_offset = 0
        if (self.Name == "Float") :
            for i in range(0,self.DIM**2):
                    element = float(matrix[i])
                    element_hex = self.FloatDouble_to_hex(
                        element)
                    destino.write(address_offset,element_hex
                        [0])
                    address_offset = address_offset + 4
        else:
            for i in range(0,self.DIM**2):
                element = matrix[i]
                element_hex = self.FloatDouble_to_hex(element)
                destino.write(address_offset,element_hex[0])
                address_offset = address_offset + 4
                destino.write(address_offset,element_hex[1])
                address_offset = address_offset + 4

    def BeginTimer (self):
        self.timer.register_map.TLR0 = self.MAX_COUNT
        self.timer.register_map.TCSR0.LOAD0 = 1
```

```
        self.timer.register_map.TCSR0.LOAD0 = 0
        self.timer.register_map.TCSR0.T0INT = 0
        self.timer.register_map.TCSR0.MDT0 = 0
        self.timer.register_map.TCSR0.UDT0 = 1
        self.timer.register_map.TCSR0.ENT0 = 1

    def StopTimer (self):
        self.timer.register_map.TCSR0.ENT0 = 0
        self.timer.register_map.TCSR1.ENT1 = 0
        TLRx = self.timer.read(0x08)
        TIMING_INTERVAL = (self.MAX_COUNT - TLRx + 2) * self.
            AXI_CLOCK_PERIOD;
        print(" Realizado en " ,TIMING_INTERVAL," segundos \n")

    def FloatDouble_to_hex(self,f):
        valor = [0]*2
        if (self.Name == "Float"):
            #valor = struct.pack('f', value)
            valort = hex(struct.unpack('<I', struct.pack('<f',
                f))[0])
            valor[0] = int(valort,16)
        else:
            valort = hex(struct.unpack('<Q', struct.pack('<d',
                f))[0])
            if(len(valort)<=10):
                valor[0] = int(valort[2:],16)##lower
                valor[1] = 0
            else:
                valor[0] = int(valort[10:len(valort)],16)
                valor[1] = int(valort[2:10],16) ##upper
        return valor


    def hex_to_FloatDouble(self,s):
        i = int(s, 16)
        if (self.Name == "Float"):
            cp = pointer(c_int(i))
            fp = cast(cp, POINTER(c_float))
        else :
            cp = pointer(c_longlong(i))
            fp = cast(cp, POINTER(c_double))
        return fp.contents.value
```

```
def DoInversa(self):
    print ("Inversa \n")
    self.inversa.write(0x00, 4)
    self.BeginTimer()
    self.inversa.write(0x00, 1)
    while True:
        estado = self.inversa.read(0x00)
        if (estado == 2 or estado == 4 or estado == 6):
            break
    self.StopTimer()
    if (self.Name == "Float"):
        determinante = self.inversa.read(0x10)  #31 - 0
        det = self.hex_to_FloatDouble(hex(determinante))
    else:
        lowValue = self.inversa.read(0x10)  #31 - 0
        uperValue = self.inversa.read(0x14)  #63:32
        new_low = hex(lowValue)
        new_upper = hex(uperValue)
        if (len(new_low) < 10):
            size = len(new_low)
            new_low = new_low[0:2]+"0"*(10-size)+new_low
                [2:]
        if (len(new_upper) < 10):
            size = len(new_upper)
            new_upper = new_upper[0:2]+"0"*(10-size)+
                new_upper[2:]
        numero = new_upper + new_low[2:]
        det = self.hex_to_FloatDouble(numero)
    print ("el determinante es",det )

def DoMultiplicacion(self):
    print ("Multiplicaci n \n")
    self.multiplicacion.write(0x00, 4)
    self.BeginTimer()
    self.multiplicacion.write(0x00, 1)
    while True:
        estado = self.multiplicacion.read(0x00)
        if (estado == 2 or estado == 4 or estado == 6):
            break
    self.StopTimer()
```