Instituto Nacional de Astrofísica, Óptica y Electrónica.

Diseño y desarrollo de pruebas sintetizadas para distintos circuitos digitales ejemplo, en un entorno FPGA-PC.

Por

Tapia Morales Eduardo Alberto

Tesis sometida como requisito parcial para obtener el grado de Maestro en Ciencias, en la especialidad de Electrónica en el Instituto Nacional de Astrofísica, Óptica y Electrónica.

Supervisada por:

Dr. Jorge Francisco Martínez Carballido, director

Coordinación de Electrónica del INAOE

Dra. Gloria Castro Muñoz, co-directora

Universidad Hispana de Puebla

Agosto 2018
Tonantzintla, Puebla

© INAOE 2018

Derechos Reservados

El autor otorga al INAOE el permiso de reproducir y distribuir copias de esta tesis en su totalidad o en partes.

RESUMEN

Desde el desarrollo del primer procesador integrado hasta el día de hoy el número de transistores ha aumentado drásticamente, siendo favorable para el desarrollo de circuitos cada vez más complejos y sofisticados. Sin embargo, debido al aumento en la complejidad, procesos en el desarrollo de un proyecto se han visto afectados, un claro ejemplo de esto es la verificación pues en proyectos de gran complejidad puede llegar a tomar hasta el 70 por ciento de los esfuerzos totales.

Dado que la verificación es un proceso clave en el desarrollo de un proyecto, pues permite elevar la calidad del producto, corrobora el cumplimiento de las especificaciones y es un factor clave en tiempos de comercialización. Diversas tecnologías HW/SW (Co-emulación) han sido desarrolladas por empresas con el propósito de mejorar la verificación; sin embargo, estas suelen ser de acceso limitado para pequeñas empresas e instituciones académicas, debido al alto costo que estas implican.

Este trabajo propone un esquema de comunicación que permite el intercambio de información entre un entorno de hardware y un entorno de software (FPGA-PC) con propósitos de verificación, con lo cual se busca mejorar los tiempos que normalmente tomaría si se utilizaran métodos convencionales basados en software. Esto se realiza por medio de herramientas con acceso libre: Xilinx Vivado, SystemVerilog, C, Python y mediante la modificación del framework PYNQ. El trabajo con este entorno (FPGA-PC) hace posible para instituciones educativas y pequeñas empresas trabajar en verificación, con recursos libres y hardware de costo accesible.

Con propósitos demostrativos se realizó el diseño e implementación de cuatro casos de estudio donde se muestra la utilidad del esquema propuesto bajo cuatro posibles escenarios, pero sin estar limitado a estos. En el primero se trata de un elemento combinacional, en el segundo un elemento compuesto por elementos combinacionales y secuenciales, para el tercero se tomó como base un microprocesador “pipelined MIPS” de cuatro etapas y se dividió en dos particiones distribuidas en dos tarjetas de emulación PYNQ Z1, por último, se tomó como base un elemento bajo prueba compuesto por elementos combinacionales y secuenciales, y se generaron múltiples “Testbenches” distribuidos en SystemVerilog, Python y VHDL, en el entorno FPGA-PC.

AGRADECIMIENTOS

Primero que nada, me gustaría agradecer a todas aquellas personas que han influenciado directa o indirectamente en la persona que hoy soy, incluyendo amigos, maestros y familiares. Pero especialmente me gustaría agradecer a mi madre Susana Morales Reyes y mi padre Eduardo Tapia Escobedo, por ser un pilar indispensable en mi vida, pues si ellos no fueran como son, yo no estaría donde estoy. También me gustaría reconocer a Bernardo Monroy Ortega… “Fernando”, pues representa para mí una persona digna de confianza que siempre ha estado para nosotros.

También me gustaría reconocer al Dr. Jorge Francisco Martínez Carballido, por su apoyo a lo largo de mi estancia en el INAOE, y aprovecho para decir que es un profesor capaz y confiable.

Por último, pero sin ser menos importante, me gustaría dar las gracias al Consejo Nacional de Ciencia y Tecnología (CONACYT) por el apoyo económico otorgado para estudios de maestría durante el periodo 2016-2018.

Tabla de contenido

RESUMEN I
AGRADECIMIENTOS III
LISTA DE FIGURAS VII
LISTA DE TABLAS IX
Capítulo 1 Introducción .... - 1 -
1.1 Motivación .... - 1 -
1.2 Descripción del problema .... - 2 -
1.3 Objetivo General .... - 4 -
1.4 Objetivos específicos .... - 4 -
1.5 Justificación .... - 4 -
1.6 Solución propuesta .... - 5 -
Capítulo 2 Revisión de la literatura .... - 9 -
2.1 Introducción .... - 9 -
2.2 Antecedentes .... - 9 -
2.3 Conceptos básicos .... - 11 -
2.3.1 Verificación .... - 11 -
2.3.2 Factor Humano .... - 13 -
2.3.3 Aproximaciones de verificación .... - 15 -
2.3.4 Intellectual Property (IP). .... - 17 -
Capítulo 3 Diseño e implementación del esquema de comunicación .... - 18 -
3.1 Introducción .... - 18 -
3.2 Planteamiento del esquema de comunicación .... - 18 -
3.2.1 Planteamiento: Interacción .... - 18 -
3.2.2 Planteamiento: Estructura .... - 20 -
3.2.3 Consolidación del esquema de comunicación a un entorno FPGA-PC. ... - 21 -
3.3 Implementación del esquema de comunicación .... - 24 -
3.3.1 Preparación DUT .... - 24 -
3.3.2 Planteamiento de la estructura de verificación .... - 26 -
3.3.3 Generación de archivos fuente .... - 29 -
3.3.4 Ubicación de archivos .... - 35 -
3.3.5 Arranque del Servidor .... - 36 -
3.3.6 Inicialización de la Simulación .... - 37 -
3.3.7 Ejecución de la Co-emulación .... - 39 -

3.4 Funciones para el intercambio de información ... - 40 -
Capítulo 4 Diseño e implementación de casos de estudio ... - 43 -
4.1 Introducción ... - 43 -
4.2 Co-emulación de circuitos puramente Combinacionales ... - 43 -
4.2.1 Simulación: Ejemplo1. ... - 44 -
4.2.2 Interfaz: Ejemplo 1. ... - 47 -
4.2.3 Emulación: Ejemplo 1. ... - 48 -
4.3 Co-emulación de circuitos secuenciales ... - 54 -
4.3.1 Simulación: Ejemplo 2 ... - 55 -
4.3.2 Interfaz: Ejemplo 2 ... - 59 -
4.3.3 Emulación: Ejemplo 2 ... - 61 -
Capítulo 5 Resultados ... - 67 -
5.1 Introducción ... - 67 -
5.2 Caso 1: Circuitos puramente combinaciones ... - 68 -
5.3 Caso 2: Circuitos secuenciales ... - 69 -
5.4 Caso 3: Generación de pruebas en diferentes niveles de abstracción ... - 71 -
5.5 Caso 4: Dispositivo bajo prueba particionado ... - 75 -
Capítulo 6 Conclusiones y trabajo a futuro ... - 80 -
6.1 Conclusiones ... - 80 -
6.2 Trabajo Futuro ... - 81 -
Apéndice A. Código Casos de Estudio. ... - 84 -
A.1 Caso de estudio uno ... - 84 -
A.1.1 File.sv ... - 84 -
A.1.2 File.c ... - 86 -
A.1.3 File0.py ... - 87 -
A.1.4 File1.py ... - 87 -
A.1.5 Servidor.py ... - 87 -
A.2 Caso de estudio dos ... - 88 -
A.2.1 File.sv ... - 88 -
A.2.2 File.c ... - 89 -
A.2.3 File0.py ... - 91 -
A.2.4 File1.py ... - 91 -
A.2.5 File2.py ... - 91 -
A.2.6 Servidor.py ... - 91 -

A.3 Caso de estudio tres ... - 94 -
A.3.1 File.sv ... - 94 -
A.3.2 File.c ... - 95 -
A.3.3 File0.py ... - 99 -
A.3.4 File1.py ... - 99 -
A.3.5 File2.py ... - 99 -
A.3.6 File3.py ... - 99 -
A.3.7 Servidor.py ... - 99 -
A.4 Caso de estudio cuatro ... - 103 -
A.4.1 File.sv ... - 103 -
A.4.2 File.c ... - 105 -
A.4.2 File0.py ... - 110 -
A.4.3 File1.py ... - 111 -
A.4.4 File2.py ... - 111 -
A.4.3 File3.py ... - 111 -
A.4.4 File4.py ... - 111 -
A.4.5 Servidor1.py ... - 111 -
A.4.6 Servidor2.py ... - 113 -
Apéndice B. ... - 115 -
B.1 Requerimientos de Software ... - 115 -
B.2 Requerimientos de Hardware ... - 115 -
B.3 Manuales y guías de interés. ... - 115 -
Apéndice C. Otros Resultados ... - 116 -
Referencias ... - 117 -

LISTA DE FIGURAS

Figura 1 Diagrama a bloques del esquema de comunicación propuesto. .... - 6 -

Figura 2 Flujo de diseño para el uso del esquema de comunicación propuesto. .... - 7 -

Figura 3 Modelo de Re-convergencia. .... - 12 -

Figura 4 Estructura de pruebas Testbench. .... - 13 -

Figura 5 Modelo de re-convergencia: Factor humano... - 14 -

Figura 6 Modelo de re-convergencia: redundancia. .... - 15 -

Figura 7 Flujo de información entre Maestro-Esclavo. .... - 19 -

Figura 8 Extrapolación a múltiples bloques de emulación. .... - 19 -

Figura 9 Estructura de comunicación propuesta. .... - 20 -

Figura 10 Diagrama a bloques del esquema de comunicación propuesto. .... - 22 -

Figura 11 Interacción “Testbench-DUT” .... - 27 -

Figura 12 Casos ejemplo distribución estructura de pruebas en el esquema de comunicación propuesto. .... - 28 -

Figura 13 Distribución de archivos fuente. .... - 35 -

Figura 14 Ventana “Jupyter Notebooks”. .... - 36 -

Figura 15 Entorno gráfico Vivado: Consola TCL. .... - 37 -

Figura 16 Flujo de ejecucion del modelo de co-emulación. .... - 39 -

Figura 17 Conexión de puertos entre simulación, interfaz y emulación. .... - 41 -

Figura 18 Funciones básicas para el intercambio de información. .... - 41 -

Figura 19 Distribución de estructura de pruebas y DUT: Ejemplo 1. .... - 44 -

Figura 20 Estructura de simulación: Ejemplo1. .... - 46 -

Figura 21 Diagrama de flujo "Test Controller (TC)": Ejemplo1. .... - 47 -

Figura 22 Funciones en las tres capas de la interfaz: Ejemplo 1. .... - 48 -

Figura 23 Diagrama a bloques: Ejemplo 1. .... - 49 -

Figura 24 Diagrama RTL: Ejemplo 1 .... - 50 -

Figura 25 IP Personalizada: Ejemplo1. .... - 51 -

Figura 26 Mapeo interno de registros de la IP: Ejemplo1. .... - 51 -

Figura 27 Diagrama RTL IP personalizada: Ejemplo 1. .... - 53 -

Figura 28 Diagrama RTL, Bus S00_AXI IP personalizada: Ejemplo1. .... - 53 -

Figura 29 Diagrama RTL Lógica del usuario: Ejemplo 1. .... - 53 -

Figura 30 Distribución de estructura de pruebas y DUT: Ejemplo 2. .... - 54 -

Figura 31 Estructura de simulación: Ejemplo 2. .... - 58 -

Figura 32 Diagrama de flujo del bloque "Test Controller 1 (TC1)": Ejemplo 2. .... - 59 -

Figura 33 Funciones en las capas de la interfaz: Ejemplo 2. .... - 60 -

Figura 34 Diagrama a bloques: Ejemplo2. .... - 61 -

Figura 35 Diagrama RTL: Ejemplo 2. .... - 62 -

Figura 36 IP Personalizada: Ejemplo2. .... - 63 -

Figura 37 Mapeo interno de registros de la IP: Ejemplo2. .... - 63 -

Figura 38 Diagrama RTL IP personalizada: Ejemplo 2. .... - 66 -

Figura 39 Diagrama RTL de la lógica del usuario: Ejemplo 2. .... - 67 -

Figura 40 Resultados de Co-emulación consola TCL: Caso de estudio 1 .... - 69 -

Figura 41 Resultado Co-emulación consola TCL: Caso de estudio 2. .... - 71 -

Figura 42 4-Bits Carry Look Ahead. .... - 72 -

Figura 43 Estructura de pruebas: Caso de estudio 3. .... - 73 -

Figura 44 Test Patter Generator (TPG): Caso de prueba 3. .... - 73 -

Figura 45 Resultados Co-emulación consola TCL: Caso de estudio 3. .... - 75 -

Figura 46 Diagrama a bloques procesador pipeline: Caso de estudio 4. .... - 76 -

Figura 47 Estructura de pruebas: caso de estudio 4.. - 78 -

Figura 48 Partición 1: Caso de estudio 4. .... - 79 -

Figura 49 Partición 2: Caso de estudio 4. .... - 79 -

Figura 50 Resultado Co-emulación consola TCL: Caso de estudio 4.| .... - 80 -

LISTA DE TABLAS

Tabla 1 Lógica DUT ejemplo 1. .... - 52 -

Tabla 2 Patrones de prueba emulación: Caso de prueba 3. .... - 74 -

Tabla 3 Instrucciones Tipo R: caso de estudio 4. .... - 76 -

Tabla 4 Instrucciones tipo I: caso de estudio 4.. - 77 -

Capítulo 1 Introducción

1.1 Motivación

En 1965 Gordon Moore formuló una predicción basada en datos empíricos, donde estableció que el número de transistores por unidad de área se incrementaría al doble cada año, sin embargo, esta predicción fue modificada en 1975 por el mismo Gordon Moore. El estableció que el número de transistores aumentaría al doble cada dos años [1]. De aquí en adelante esto sería conocido como la Ley de Moore.

Con el avance rápido en las tecnologías de semiconductores, el número de transistores por circuito se incrementó rápidamente y las predicciones de Gordon Moore resultaron ser ciertas. El aumento en la densidad de transistores por área ha permitido la realización de diseños cada vez más complejos y sofisticados [2]. Un claro ejemplo de esto se puede ver en celulares, computadoras, automóviles, etc.

Conforme el número de transistores aumentaba, el diseño y creación de prototipos hardware también cambió. Dando lugar a "dispositivos de cómputo específico que pueden incorporar en un solo chip la funcionalidad de múltiples chips de propósito general o propósito específico" [2], nombrados "System on Chip" (SoC). Este nuevo enfoque ofrece múltiples beneficios, así como múltiples retos. Debido al aumento en la complejidad de los diseños, se estima que entre el cuarenta y setenta por ciento de los esfuerzos en el desarrollo de un nuevo proyecto se enfocan en procesos de verificación, incluso en algunos proyectos de gran complejidad el número de ingenieros encargados de verificación puede llegar a sobrepasar al número de ingenieros de diseño en una proporción de tres a uno [2]–[8].

1.2 Descripción del problema

El aumento en la capacidad de integración de los Circuitos Integrados (IC), ha permitido crear diseños cada vez más complejos, revolucionando así la mañera de diseñar y crear nuevos prototipos hardware. Este fenómeno ha sido denominado “System-on-Chip” (SoC). El diseño basado en SoC ha traído consigo muchos desafíos, pero sin duda el más grande se encuentra en la verificación. “La verificación es el proceso utilizado para demostrar que la intención de un diseño se preserva en su implementación” [9]. Hasta el 70 por ciento del esfuerzo requerido en el desarrollo de un nuevo diseño puede ser consumido en los procesos de verificación [2], [6], [8].

Conforme los nuevos diseños se han tornado cada vez más complejos y la demanda de tiempos de comercialización de dispositivos electrónicos se ha hecho cada vez más corta, se ha provocado que el proceso de verificación sea un punto clave en el cumplimiento de aquellas restricciones de tiempo. En un intento por reducir los tiempos de diseño y/o verificación, se ha promovido el uso de “Intellectual Property” (IP), sin embargo el uso de estas IPs ha traido consigo nuevos problemas a solucionar en cuestiones de seguridad [3], [5], [6],[10].

Procesos tradicionales de verificación basados en software (simulación) resultan imprácticos de utilizar, dado el excesivo tiempo que requieren, y esto a su vez los hace obsoletos; pues conforme aumenta la complejidad de los diseños, los tiempos de simulación aumentan, pudiendo pasar de minutos-horas a días-semanas o aún más. El desarrollo de nuevos procesos de verificación basados en entornos hardware/software, han sido propuestos como una alternativa viable para cumplir con las restricciones de tiempos de comercialización requeridos [3], [4], [11], [12].

Entre los procesos de verificación pre-silicio más comúnmente usados por empresas de diseño de circuitos electrónicos encontramos la verificación basada en técnicas de emulación. La emulación, también conocida como prototipado de gran velocidad, es una técnica que crea prototipos tempranos de sistemas complejos y los mapea a componentes de hardware programable (tales como FPGAs) antes de que éstos sean fabricados con la finalidad de verificar funcionalmente el diseño [13].

El mercado de la emulación actualmente está muy bien remunerado y se espera que siga en crecimiento en los próximos años debido a que ofrece ventajas como velocidad, abstracción, verificación software, beneficios de visibilidad y bajo consumo de energía en comparación con la técnica de simulación software [14]–[16].

Sin embargo, a pesar de las numerosas ventajas que ofrece la emulación y de la gran variedad de productos ofertados actualmente, también es cierto que este tipo de herramientas sólo son accesibles a grandes empresas que utilizan verificación debido a que los costos de los equipos son bastante elevados (más de un millón de dólares) y por lo tanto instituciones educativas y pequeñas empresas no tienen la posibilidad de adquirirlos.

Ante esta situación y con la finalidad de brindar la oportunidad a los estudiantes y pequeñas empresas de aprender y aplicar las técnicas de verificación usadas en los diseños comerciales a gran escala, en este trabajo se presenta el diseño e implementación de un esquema de comunicación que permite recrear el proceso de co-emulación con fines de verificación funcional en diseños digitales de mediana complejidad.

1.3 Objetivo General

- El objetivo general de este trabajo es diseñar e implementar un esquema de comunicación que permita el intercambio de información entre simulación y emulación con la finalidad de facilitar el proceso de verificación funcional de diseños hardware.

1.4 Objetivos específicos

- Describir el esquema de comunicación propuesto y las herramientas software utilizadas en el proceso
- Diseñar e implementar dos casos de estudio de circuitos digitales combinacionales y secuenciales de mediana complejidad utilizando el esquema de comunicación propuesto.
- Extender el esquema de comunicación propuesto a la verificación de diseños hardware particionados en dos o más tarjetas FPGA de tal forma que se permita la colaboración entre tarjetas
- Establecer las bases del proceso de verificación de diseños hardware sobre tarjetas accesibles a estudiantes y empresas pequeñas.

1.5 Justificación

El desarrollo del esquema de comunicación propuesto en el presente trabajo de investigación contribuirá a atender la necesidad de cumplir con tiempos de comercialización más cortos debido a que facilitará procesos de verificación. Un aspecto importante a resaltar es la flexibilidad del esquema propuesto, pues basados en un sistema de pruebas, la descripción de este se puede realizar por medio de SystemVerilog, Python, VHDL o Verilog, proporcionando diferentes niveles de abstracción. Además, las herramientas utilizadas para

su implementación como lo son Xsim, SystemVerilog, DPI-C, C, Python Embebido, Python XMLRPC, PYNQ-Z1, son de acceso libre o de bajo costo, por lo que resulta accesible para instituciones académicas o pequeñas empresas interesadas en realizar co-emulación.

Por otro lado, en el campo de la educación, se espera que este esquema de comunicación sirva como una herramienta auxiliar que permita a los alumnos entender conceptos tales como simulación, emulación y co-emulación de manera interactiva, mediante el ejercitamiento de las estructuras. Además, que sirva como auxiliar en la implementación de estructuras de prueba en cursos de verificación, pues como se resalta en [9], la verificación es una parte importante en el desarrollo de nuevos diseños digitales a gran escala, por lo que resulta indispensable que futuros ingenieros en electrónica estén familiarizados con este concepto.

Se espera que el esquema de comunicación propuesto en la Figura 1 sirva como punto de referencia para esquemas de comunicación de acceso libre más sofisticados.

1.6 Solución propuesta

El diagrama a bloques del esquema de comunicación propuesto en este trabajo de investigación se muestra en la Figura 1. El esquema está constituido por cinco bloques de ejecución y cuatro puentes de comunicación distribuidos en un entorno FPGA-PC. La estructura puede ser agrupada en tres bloques de acuerdo con su propósito como: simulación (bloque 1), interfaz (bloque 2 a bloque 4), y emulación (bloque 5).

El intercambio de información comienza mediante el llamado de funciones C bloqueantes (se ejecuta serialmente, instrucción por instrucción) desde SystemVerilog

mediante el uso de “Direct Programming Interface C” (DPI-C). C se utiliza como medio para realizar la captura de señales generadas en SystemVerilog y enviar la información por medio de “Remote Procedure Call” (RPC) a través del llamado de funciones cliente del paquete XMLRPC Python descritas en archivos Python llamados mediante Python embebido. El servidor XMLRPC Python descrito en el bloque 4, recibe las señales generadas y las envía a la emulación mediante registros mapeados en memoria. La emulación recibe las señales generadas de simulación y retorna una respuesta que es transferida a través de los bloques 4, 3, 2 hasta llegar a la simulación y termina el llamado de la función.

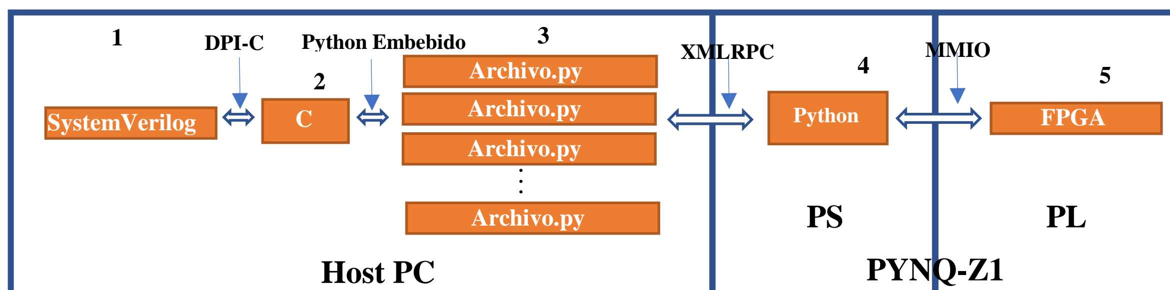


Figura 1 Diagrama a bloques del esquema de comunicación propuesto.

En el diagrama de la Figura 2 se muestra el flujo de diseño para el uso del esquema propuesto en la Figura 1. Para utilizar el esquema de comunicación propuesto, se parte de un diseño que se desea probar en un entorno de hardware, posteriormente se realiza el planteamiento del sistema de pruebas (Testbench) y se distribuye en la estructura del esquema de comunicación.

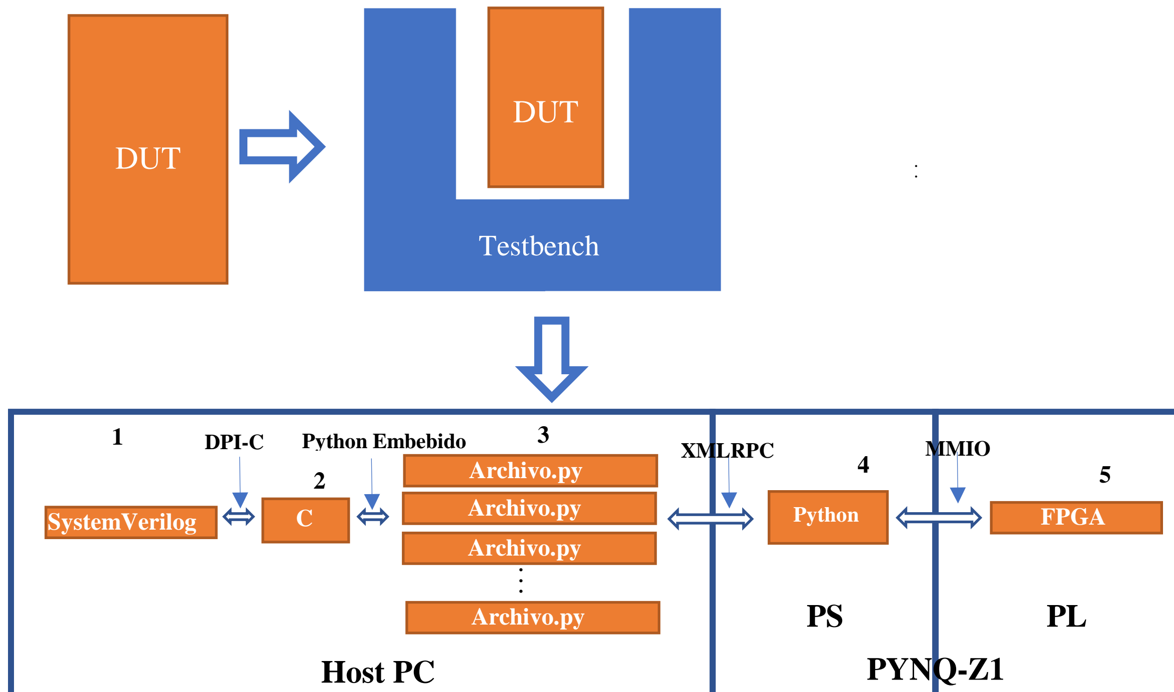


Figura 2 Flujo de diseño para el uso del esquema de comunicación propuesto.

El propósito de este trabajo fue el diseño e implementación de un sistema de comunicación entre simulación y emulación para facilitar procesos de verificación. Así como el desarrollo de distintos circuitos digitales ejemplo, donde se consideran distintos escenarios de verificación. Esto se llevó a cabo en dos etapas.

- Primera etapa: En esta etapa se realizó el planteamiento del esquema de comunicación. Esta se dividió en tres sub-etapas.
    - *Nivel interacción*: En esta sub-etapa se definió como se llevaría a cabo el intercambio de información entre simulación y emulación. Dado que la emulación es superior en cuanto a velocidad con respecto a la simulación, se adoptaron directivas del esquema Maestro-Esclavo, donde la simulación cumple el papel de Maestro y la emulación el papel de Esclavo.
    - *Nivel estructura*: En esta sub-etapa se estableció una estructura que permitiera el intercambio entre simulación y emulación. Tanto simulación

como emulación se encuentran en diferentes entornos, por lo que el uso de una interfaz se vuelve necesario. Se propuso una estructura constituida por tres bloques comunicados entre sí: simulación, interfaz, emulación.

+ *Nivel implementación*: En esta sub-etapa se definieron las herramientas que se utilizaron para la implementación de la simulación, emulación e interfaz.

- Segunda etapa: En esta etapa se realizó el diseño, desarrollo e implementación de diferentes estructuras de prueba para circuitos digitales ejemplo utilizando el esquema de comunicación propuesto. Esta se dividió en tres sub-etapas.

    + *Diseño del* "Device Under Test" (DUT): En esta etapa se realiza el diseño del circuito digital.
    + *Planteamiento del sistema de pruebas*: En esta etapa se define el sistema de pruebas que se aplicará sobre el DUT.
    + *Co-emulación*: Una vez se define el diseño y su sistema de pruebas, se realizó la distribución sobre la estructura de comunicación.

Capítulo 2 Revisión de la literatura

2.1 Introducción

El material presentado en este capítulo corresponde a conceptos básicos necesarios para la correcta interpretación de los capítulos subsecuentes. En la sección 2.2 se presentan un breve resumen de trabajos concernientes al presente trabajo de investigación, donde se citan sus principales características, así como el propósito con el cual fueron creados. En la sección 2.3 se proveen los conceptos básicos relacionados con el tema de investigación de este trabajo, debido a que es un tema bastante extenso y de cierta manera ambiguo se recomienda revisar los libros [6], [8], [17] que proveen un panorama más amplio del proceso de verificación tanto para el desarrollo de un Testbench, aplicación de pruebas a sistemas embebido y System on Chip (SoC).

2.2 Antecedentes

En [12] se resalta el enorme consumo de tiempo que conlleva el proceso de verificación para System On Chip (SoC) de gran complejidad, además se destaca la importancia de metodologías basadas en entornos hardware-software como un método prominente para la reducción de los tiempos de verificación basados en una cadena de escaneo mediante el uso de un Testbench de alto nivel implementado en SystemC aplicado a un circuito de referencia para el control de semáforos, tomado del "International Symposium on Circuits and Systems in 1989 (ISCAS89)" nombrado S400. El intercambio de información es llevado a cabo mediante "Standard Co-Emulation Modeling Interface (SCE-MI)", desarrollado por Accellera para el intercambio de información entre simulación y emulación. La metodología propuesta probó ser 82 por ciento más rápida que simulación RTL basada en software. El trabajo en [18] se enfoca en solucionar los problemas de costos y

comercialización que surgen en metodologías tradicionales donde elementos de software y hardware de un "System On Chip (SoC)" son probados en conjunto una vez que el hardware ha sido manufacturado, a través de un entorno de hardware-software que permite su interacción en etapas tempranas en el desarrollo del proyecto. La descripción de los elementos de software es realizada mediante SystemC, ya que permite la descripción de elementos de hardware también, por su parte para la emulación se utilizó un dispositivo reconfigurable que permite el acceso a través de registros mapeados en memoria. El intercambio de información se realizó por medio de una clase descrita en SystemC que permite el acceso a los registros mapeados en memoria, nombrada "Bus Functional Mode" (BFM). En [19] se desarrolló un "framework" para la verificación de sistemas embebidos, tomando como base para el control, y la descripción de componentes de hardware y software SystemC. Por su parte para la emulación se utilizó la tarjeta "XilinxVirtex II FPGA fabric", donde se implementó un procesador "MicroBlaze softcore", en conjunto con otros módulos ASIC. Para el intercambio de información entre SystemC y el hardware reconfigurable se tomó como base el estándar SCE-MI. En [20] se resalta la importancia de la verificación a nivel funcional pre-fabricación de System on Chip (SoC), así como la importancia de la verificación de tiempos ("timing verification"), además se propone una técnica de verificación temporizada basada en un entorno Hardware-Software, donde se busca reducir las interacciones entre simulación-emulación que afectan la velocidad de la co-emulación mediante la sincronización virtual, donde la sincronización entre simulación y emulación no es llevada a cabo ciclo a ciclo de reloj, en vez de ello se realiza cada vez que ocurre un evento que afecta a los demás componentes.

Después de realizar una revisión de los trabajos mencionados anteriormente, se encontró que la mayoría utiliza SystemC como medio para realizar la descripción de elementos de hardware y software, mediante el uso de funciones, lo cual resulta útil para desarrolladores software, sin embargo en el esquema de comunicación propuesto dicha descripción se hace por medio de SystemVerilog, por lo cual descripciones de hardware en lenguajes HDL pueden ser reutilizadas, además se puede recurrir a procedimientos secuenciales para el control de la ejecución de la co-verificación tales como ciclos for, if, etc. Por otro lado, el uso de Python también provee un medio para la descripción de bloques de software en sistemas embebidos.

Además, algunos trabajos se enfocan en el uso de entornos comerciales para la implementación de la co-emulación desarrollados por empresas, lo que implica un costo extra, por otro lado, al ser un producto comercial, algunos detalles de la implementación permanecen ocultos para el usuario final. En ese sentido el esquema propuesto al hacer uso de herramientas de acceso libre provee un entendimiento transparente de cómo funciona y puede ser modificado según sea requerido.

2.3 Conceptos básicos

En esta sección se proporcionan los elementos básicos utilizados en este trabajo de investigación, y se aborda de manera un tanto general del porque la verificación es un proceso sumamente difícil. La información mostrada en esta sección fue obtenida de los libros [6], [8], [17]

2.3.1 Verificación

Sin duda, sin importar el área de estudio, la verificación es uno de los procesos más importantes en el desarrollo de un proyecto. Especialmente en el desarrollo de System-on-

Chip (SoC) debido a que este llega a consumir entre el cincuenta y setenta por ciento de los esfuerzos totales en el desarrollo de un proyecto, además representa un punto clave para cumplir con tiempos de comercialización más cortos, así como para elevar la calidad del producto. Este es un proceso mediante el cual se busca reconciliar la implementación de un diseño con sus especificaciones, a fin de determinar si cumple con ellas.

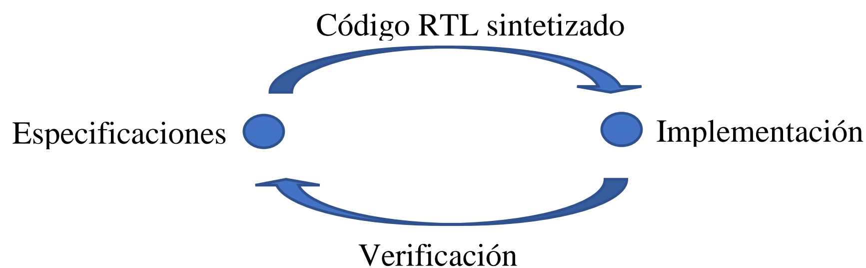


Figura 3 Modelo de Re-convergencia.

Como se muestra en la Figura 3, la verificación puede ser representada mediante el modelo de re-convergencia. Donde la trayectoria uno, representa la transición entre la especificación y la implementación, específicamente en el desarrollo de circuitos digitales se refiere al modelado de circuitos digitales mediante lenguajes HDL sintetizados a nivel RTL a partir de una tabla de verdad, una descripción, etc. Por otro lado, la segunda trayectoria corresponde a la verificación, siendo un proceso mediante el cual se determina que la descripción HDL se realizó de manera precisa acorde a las especificaciones, puntualmente en el desarrollo de sistemas digitales, es un proceso que es realizado por medio de Testbench, una herramienta que provee señales de entradas al dispositivo bajo prueba y adicionalmente observa las salidas (ver Figura 4).

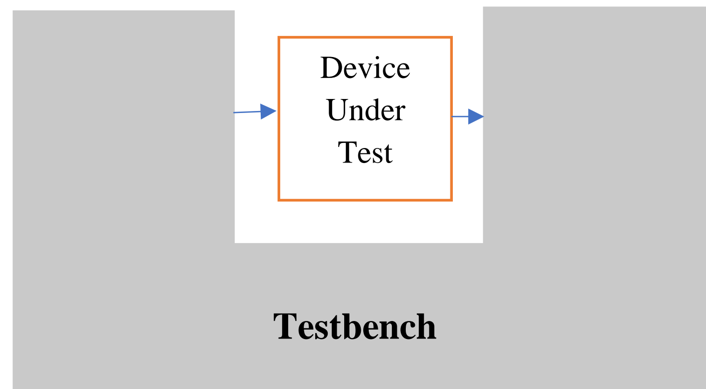


Figura 4 Estructura de pruebas Testbench.

2.3.2 Factor Humano

Un aspecto a considerar en el desarrollo de un proyecto digital es el punto de partida, esto es, la especificación. Las especificaciones como tal se encuentran descritas mediante documentos de texto, por lo que no pueden ser utilizadas directamente como punto de partida en el desarrollo de un proyecto, sino que tienen que pasar por un proceso de interpretación, entrando en juego el "factor humano". Y es aquí donde se encuentra la diferencia entre verificación y validación, mientras que la verificación se asegura de que la interpretación en la que es basado el diseño se mantengan en la implementación, la validación es un proceso más complejo mediante el cual se determina si las interpretaciones de las especificaciones son correctas de acuerdo al propósito del diseño (Ver Figura 5).

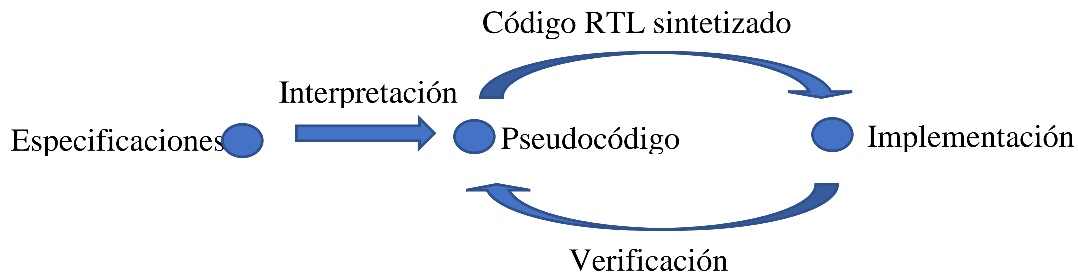


Figura 5 Modelo de re-convergencia: Factor humano.

Debido a que este "factor humano" afecta en el desarrollo de un proyecto, se han adoptado ciertas herramientas para tratar con este problema:

- Automatización: mediante esta técnica se busca quitar por completo la intervención humana del proceso de interpretación de las especificaciones. Sin embargo, no es siempre posible debido a la ambigüedad que presentan ciertos procesos, y habilidades como creatividad e ingenio que no pueden ser remplazadas en el diseño de hardware [17].
- Poka-Yoke: mediante esta técnica se busca reducir el error humano, utilizándolos únicamente cuando es necesario, bajo circunstancias precisas que no presenten ambigüedad. Sin embargo, para ello se requiere que los pasos donde intervenga se encuentren bien definidos [17].
- Redundancia: mediante esta técnica se busca reducir el error humano, generando a partir de una especificación dos interpretaciones independientes entre sí, mediante el uso de dos personas, el punto de partida en el diseño se dará a partir de la interpretación uno, mientras que la re-convergencia se realizará hacia la segunda interpretación, así concluyendo que si ambas son iguales entonces es correcto el diseño (Ver Figura 6) [17].

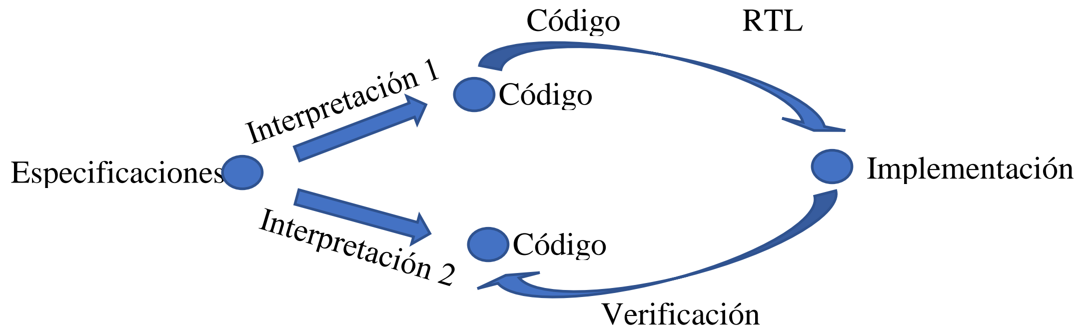


Figura 6 Modelo de re-convergencia: redundancia.

2.3.3 Aproximaciones de verificación

Si bien es cierto que la verificación es un proceso cuyo propósito está bien definido, llevar a cabo la elección de herramientas para cumplirlo depende en gran medida de la experiencia del diseñador, pues no existe como tal una fórmula que matemáticamente determine el correcto funcionamiento del diseño. Por ello existen diferentes aproximaciones para llevar a cabo el proceso de verificación. Dentro de estas tenemos:

- Verificación Formal
  - Comprobación de equivalencia: Este proceso de verificación prueba matemáticamente que el origen y la salida son lógicamente equivalentes, además que la implementación preserva su funcionalidad. Este tipo de aproximación es usualmente utilizada para comparar dos "netlist", y asegurar que su funcionalidad no ha cambiado. También es usado como medio para detectar errores de síntesis, además sirve como medio para comprobar que dos descripciones RTL son lógicamente equivalentes.
  - Comprobación de modelo: Este tipo de aproximación busca comprobar que las aserciones o características de un diseño son siempre alcanzables. Un claro

ejemplo de esto es una máquina de estados, por medio de este tipo de verificación se busca encontrar aquellos estados que no son alcanzables.

- Verificación funcional: El propósito de la verificación funcional es asegurar que la funcionalidad se preserva aun en la implementación. Para lograr esto, se tiene tres tipos de aproximación:
  - Black Box: Cuando se realiza verificación funcional por medio de Black Box, no se tiene conocimiento interno sobre la implementación del diseño, la interacción se realiza exclusivamente a través de la interfaz, esto es, por medio de puertos de salida y puertos de entrada. Usualmente bajo este esquema se carece de controlabilidad y observabilidad, resultando difícil colocar el diseño bajo pruebas en un estado especifico. Además, cuando surge un problema, resulta difícil detectar la causa del problema. Este tipo de aproximación es usada para demostrar que un diseño cumple con su intención, sin importar su implementación.
  - White Box: Al contrario de Black Box en este tipo de aproximación se tiene controlabilidad y observabilidad de las estructuras internas e implementación de un diseño, pudiendo así llevar el diseño a estados en específico, y pudiendo detectar fallas en el diseño en el momento exacto en que estas surgen y donde surgen. Sin embargo, esta requiere de un conocimiento detallado de la implementación del diseño, así como de su estructura, además se encuentra fuertemente amarrada a la implementación y no puede ser utilizada en futuros rediseños.
  - Gray Box: En este tipo de aproximación es una unión entre el Black Box y White Box, donde se busca solucionar los problemas de controlabilidad y

observabilidad del Black Box, y los problemas de dependencia de implementación del White Box.

2.3.4 Intellectual Property (IP).

Una estrategia de diseño comúnmente usada como medio para sufragar los efectos en el aumento de la complejidad de los circuitos digitales tales como System-on-Chip, o Sistemas Embebidos, es dividir el diseño dentro de bloques más simples y conectarlos entre sí, esto se realiza por medio de bloques IP. Cabe resaltar que el reuso de estos también ayuda a reducir la brecha de productividad que existe entre el número de transistores por circuito integrado y el número de transistores que un ingeniero puede aprovechar del circuito integrado (número de transistores que utiliza mediante la descripción de hardware a través de un HDL).

Una IP es un bloque de descripción de hardware mediante el cual se busca estandarizar los circuitos comúnmente usados, tales como memorias, sumadores, multiplicadores, registros, etc. Uno de los principales problemas en el uso de estos es su procedencia de terceros, pues al agregar un bloque que no es diseñado por uno mismo siempre trae consigo cierta incertidumbre, por ello, estos bloques deben contar con ciertas características tales como un buen estilo de codificación, comentarios claros a lo largo del código, una buena documentación, así como Testbench para su verificación.

Como se mencionó los bloques IP son descripciones de hardware reutilizables pues cuentan con cierto grado de confiabilidad. Por ello ingenieros de verificación ponen mucho esfuerzo en realizar bloques IP de verificación de propósito general que puedan ser utilizados con un conjunto de circuitos que compartan características, así de esta manera se ahorra el diseño de la prueba y su descripción, favoreciendo tiempos de desarrollo del proyecto.

Capítulo 3 Diseño e implementación del esquema de comunicación

3.1 Introducción

En esta sección se describe la metodología seguida para el diseño e implementación del esquema de comunicación. En la sección 3.2 se describe el planteamiento del esquema de comunicación en dos niveles de abstracción: interacción y estructura, así como la implementación del esquema en un entorno FPGA-PC. En la sección 3.3 se describen las herramientas utilizadas en la implementación del esquema, y finalmente en la sección 3.4 se realiza una descripción a fondo de los elementos en el esquema de comunicación. Información de interés puede ser encontrada en el Apéndice B., donde se listan los requerimientos de hardware, software, así como algunas ligas y manuales de interés.

3.2 Planteamiento del esquema de comunicación

En esta sección se describe el planteamiento utilizado para el diseño del esquema de comunicación. Como punto de partida se consideró que tanto emulación como simulación corren a distintas velocidades, siendo la emulación superior a la simulación en este aspecto. El segundo aspecto a tomar en cuenta es que tanto simulación como emulación se encuentran en diferentes entornos, por lo que la manera en que se maneja la información es de forma distinta, por ello se planteó una estructura que permitiera el intercambio de señales entre estas. Una vez definidos los aspectos teóricos en el esquema de comunicación, se realizó la selección de herramientas para su implementación.

3.2.1 Planteamiento: Interacción

El intercambio de información entre simulación y emulación es llevado a cabo bajo directivas Maestro-Esclavo. Donde la simulación al ser el elemento más lento funciona como Maestro, dejando a la emulación como Esclavo. A continuación, se describen las principales

características del esquema de comunicación a nivel interacción entre simulación y emulación.

- La simulación controla el intercambio de información. Esto es, la simulación inicia la comunicación con la emulación mediante la asignación de una tarea, la emulación realiza la tarea asignada por la simulación y retorna un resultado. Ver Figura 7.

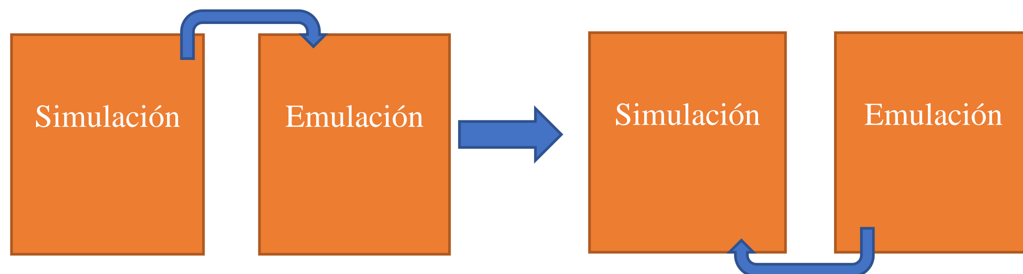


Figura 7 Flujo de información entre Maestro-Esclavo.

- La simulación puede controlar la ejecución de tareas en múltiples bloques de emulación. Ver Figura 8.

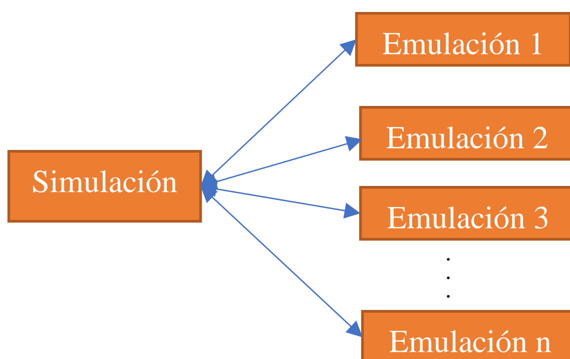


Figura 8 Extrapolación a múltiples bloques de emulación.

- La simulación puede comunicarse con múltiples bloques de emulación.
- La emulación solo se puede comunicar con un bloque de simulación.
- Las peticiones realizadas por simulación son asignadas únicamente a una emulación.
- Cada nodo de la red de emulación es distinguible por su domicilio.

3.2.2 Planteamiento: Estructura

Para realizar la comunicación entre simulación y emulación se debe contar con una estructura que permita el intercambio de información, la estructura originalmente está constituida por un elemento de simulación y un elemento de emulación unidos por un puente de comunicación, sin embargo, se debe tomar en cuenta la diferencia de entornos. Mientras que el elemento de simulación se encuentra en un entorno de software donde la información es almacenada en localidades de memoria, la emulación se encuentra en un entorno de hardware, donde la información se representa por medio de señales eléctricas. Por lo que el intercambio de información no se puede realizar de manera directa, por ello surge la necesidad de agregar un intermediario que permita la traducción de señales generadas en simulación a señales comprensibles para emulación y viceversa.

En el diagrama de la Figura 9 se muestra la estructura planteada para lograr el intercambio de información entre simulación y emulación. Dicha estructura está constituida por cinco elementos: dos puentes de comunicación y tres bloques de ejecución.

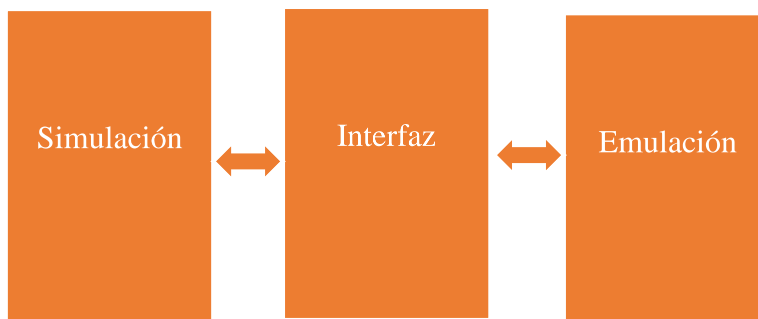


Figura 9 Estructura de comunicación propuesta.

A continuación, se describe el propósito de cada uno de los elementos de la estructura de comunicación propuesta.

- Simulación

  Es un entorno de Software para ejecución de simulación.
- Primer puente de comunicación.

  Permite el intercambio de información entre Simulación e interfaz.
- Interfaz.

  Recibe señales generadas en simulación y las transforma en señales comprensibles para emulación, y viceversa.
- Segundo puente de comunicación.

  Permite el intercambio de información entre interfaz y emulación.
- Emulación.

  Es un entorno hardware para la ejecución de la emulación.

3.2.3 Consolidación del esquema de comunicación a un entorno FPGA-PC.

Para que el esquema de comunicación propuesto sea de utilidad práctica, cada uno de los elementos que componen su estructura debe ser integrado en un único entorno. En esta sección se definen las herramientas utilizadas para la implementación del esquema, así como su propósito.

En el diagrama de la Figura 10, se muestra la distribución de los elementos del esquema de comunicación propuesto. La simulación se encuentra restringida a un entorno de software ejecutándose en un Host PC, mientras que la emulación se ejecuta en la lógica programable, PL, de la tarjeta PYNQ Z1 (entorno de hardware). La interfaz fue divida en tres bloques de ejecución distribuidos entre el Host PC y el sistema de procesamiento, PS, de la tarjeta PYNQ Z1.

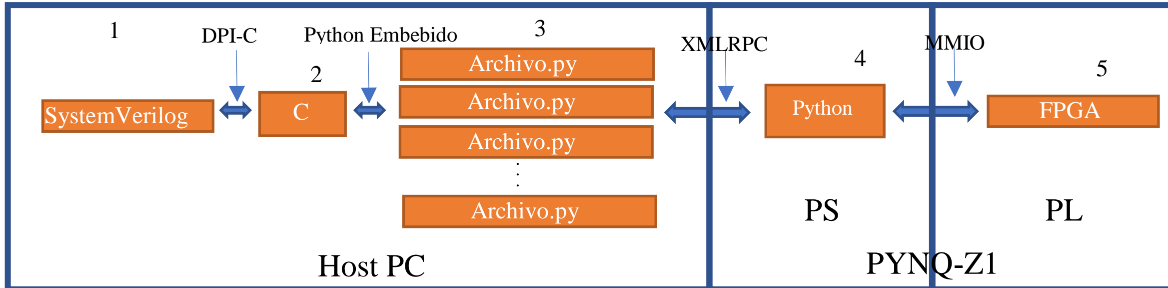


Figura 10 Diagrama a bloques del esquema de comunicación propuesto.

A continuación se describen los bloques que constituyen el esquema mostrado en la Figura 10:

- **Host PC**
  - **SystemVerilog**
    - **Propósito**: Entorno de software encargado de correr archivos de simulación descritos en HDL.
    - **Ejecución**: La ejecución se lleva a cabo a través de Xsim, un simulador provisto por Vivado Design Suite.
    - **Descripción**: La descripción de los elementos de simulación se realiza por medio de SystemVerilog.
  - **DPI-C**
    - **Propósito**: Primer bloque de interfaz, encargado de realizar la captura de datos de la simulación.
    - **Ejecución**: La ejecución se lleva a cabo mediante el llamado de funciones por parte de la simulación y es ejecutado en el Host PC.
    - **Descripción**: La descripción se realiza por medio de código C.
  - **Código de extensión para SystemVerilog**
    - **Propósito**: Segundo bloque de la interfaz que realiza la traducción de señales generadas de simulación a señales entendibles para emulación y viceversa.
    - **Ejecución**: La ejecución se da mediante el llamado de funciones desde C y es ejecutado en el Host PC.
    - **Descripción**: La descripción se realiza mediante código Python, embebido en C.

- **PYNQ-Z1**
  - **PS**
    - **Propósito**: Tercer bloque de simulación encargado de capturar los datos generados en la emulación.
    - **Ejecución**: La ejecución se realiza mediante el llamado de funciones desde Python y es ejecutado en el PS, sistema de procesamiento de la tarjeta PYNQ Z1.
    - **Descripción**: la descripción se realiza por medio de código Python.
  - **PL**
    - **Propósito**: Entorno de hardware encargado de correr archivos de emulación.
    - **Ejecución**: La ejecución se realiza a través de la lógica programable de la tarjeta PYNQ Z1.
    - **Descripción**: La descripción del diseño se realiza mediante archivos de configuración Bitstream y TCL.

Cabe resaltar que el esquema de comunicación mostrado en la Figura 10 cuenta con cinco bloques de ejecución y cuatro puentes de comunicación. Sin embargo, dichos puentes son representativos, pues queda implícito que su implementación se encuentra distribuida en los bloques de ejecución aledaños al puente. A continuación, se describen los puentes de comunicación.

- Puente 1
    - **Propósito**: Permitir el intercambio de información entre el SystemVerilog y Código de extensión.
    - **Descripción**: La comunicación entre los bloques se lleva a cabo mediante "Direct Programming Interface -C" (DPI-C).
- Puente 2
    - **Propósito**: Permitir la ejecución de código Python en DPI.
    - **Descripción**: la comunicación se lleva a cabo mediante el uso de Python Embebido.
- Puente 3

- **Propósito**: Permitir el intercambio de información entre el DPI y PS.
- **Descripción**: La comunicación se lleva a cabo mediante RPC implementado en Python.

- Puente 4
  - **Propósito**: Permitir el intercambio de información entre el PS y PL.
  - **Descripción**: La comunicación se lleva a cabo mediante el uso de registros mapeados en memoria.

3.3 Implementación del esquema de comunicación

En esta sección se describen de forma general, los pasos principales en el uso del esquema de comunicación propuesto en este trabajo de investigación, información más completa puede ser encontrada en los apéndices. A continuación, se listan los pasos para la utilización del esquema de comunicación propuesto.

1. Preparación DUT.
2. Planteamiento de la estructura de verificación.
3. Generación de archivos fuente.
4. Ubicación de archivos
5. Arranque del Servidor.
6. Inicialización de Simulación.
7. Ejecución de la Co-Emulación.

3.3.1 Preparación DUT

Como se mencionó en la sección 3.2.1 el intercambio de información es llevado a cabo bajo directivas Maestro-Esclavo. Lo que significa que el intercambio de información es iniciado por medio de tareas asignadas a la emulación, por parte de la simulación. Para que

este intercambio de información se lleve de manera correcta se deben tener en cuenta las necesidades del circuito al momento de realizar su diseño (pensado para ser probado).

Desde el punto de vista de simulación, los circuitos que corren en la emulación pueden ser divididos en dos grupos:

- Circuitos de respuesta inmediata.
- Circuitos de respuesta no inmediata.

Se entiende por circuito de respuesta inmediata aquel cuyo tiempo de respuesta con respecto a la simulación es superior. Esto es, el tiempo que tarda en cambiar la salida conforme las entradas cambian es casi/o inmediato, esto es, menor a un quinto de ciclo de simulación. Dentro de este tipo de circuitos tenemos los circuitos combinacionales. Sin embargo, también pueden ser considerados los secuenciales, siempre y cuando el tiempo que tarda para procesar las entradas sea menor a los requerimientos de simulación.

Si bien es cierto que la emulación es más rápida que la simulación, cuando se diseña un bloque con el propósito de cumplir una tarea, este puede tardar varios ciclos de reloj en cumplirla (circuitos de respuesta no inmediata), por lo que se debe asegurar que cuando se captura la salida, el bloque haya cumplido su propósito y la salida que se está capturando es la adecuada conforme a las entradas. Dentro de estos circuitos tenemos dos tipos:

- Con propósito no específico.

  En este tipo de circuitos se encuentran aquellos que cumplen tareas rutinarias y una vez configurados la realizan de manera automática. Consideremos un contador de carrera libre ascendente, que cuenta en cada flanco de subida del reloj de la tarjeta, cuya tarea es contar de 0 hasta n indefinidamente, además el tiempo de ejecución de

un ciclo de reloj de la tarjeta es muy superior (menor) a la simulación (mayor) (La simulación no puede seguir los cambios en el contador de la emulación). Para probar este tipo de circuitos es necesario añadir circuitería extra mediante la cual podamos determinar que el circuito ha cumplido un propósito.

- Con propósito específico.

Los circuitos de propósito específico transforman las entradas en salidas, teniendo bien definidos tres pasos importantes: *obtención de entradas, transformación de las entradas, envió de las salidas*.

3.3.2 Planteamiento de la estructura de verificación

"*La verificación es el proceso utilizado para demostrar que la intención de un diseño se preserva en su implementación*" [9]. Una herramienta comúnmente usada para lograr este propósito es el "Testbench", una estructura añadida al dispositivo bajo prueba con la finalidad de generar señales de entrada y comparar las señales de salida generadas por el dispositivo. El reto proviene de determinar qué señales deben ser generadas por parte del "Testbench" para determinar que el dispositivo cumple el propósito con el que fue creado [17], sin embargo no es intención del presente trabajo de investigación definir la generación de las señales de prueba correctas.

Típicamente un "Testbench" consiste en tres bloques: "Test Pattern Generator" (TPG) y "Output Response Analyzer" (ORA) y un controlador para la ejecución de prueba "Test Controller" (TC) que se encarga de que la prueba se lleve de manera correcta. El DUT interactúa con el "Testbench" en la siguiente forma:

1. Una vez iniciada la prueba el controlador se encarga de generar señales a cada uno de los bloques para indicar el inicio de la prueba.
2. TPG se encarga de generar señales de entrada para el DUT.
3. El DUT recibe las señales del TPG y como consecuencia genera señales de salida.
4. El ORA recibe las señales de salida del DUT y determina si son correctas.
5. Este proceso se repite hasta completar la prueba que el controlador tiene definida.

En el diagrama de la Figura 11 se muestra como está constituida una estructura típica para pruebas "Testbench". Cabe aclarar que esta puede ser distribuida en el esquema de comunicación propuesto en diversas formas.

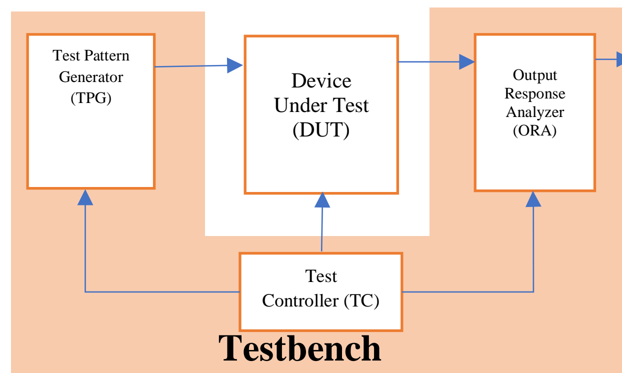


Figura 11 Interacción "Testbench-DUT"

Si consideramos que cada uno de los cuatro elementos en la estructura de pruebas puede ser descrito mediante SystemVerilog, Python o HDL, se tendrían ochenta y un formas posibles para realizar la distribución de los bloques, incluso más si se toma en cuenta particionar cada uno de los elementos y distribuir su descripción en múltiples entornos. A continuación, se muestran los casos más comunes para la distribución de la estructura de pruebas en el esquema de comunicación propuesto.

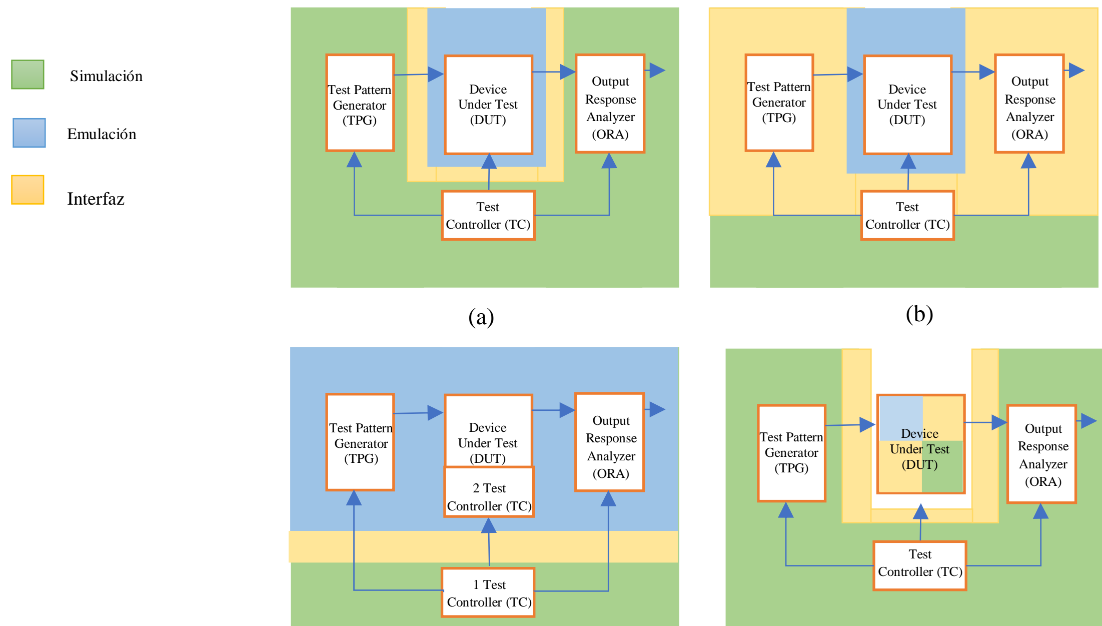


Figura 12 Casos ejemplo distribución estructura de pruebas en el esquema de comunicación propuesto.

En el diagrama de la Figura 12 se observan algunos casos ejemplos para la distribución de la estructura de pruebas en el esquema de comunicación propuesto. Concretamente en la Figura 12 (a) se observa que la parte correspondiente al "Testbench" se ubica en la simulación y el dispositivo bajo prueba se encuentre en la emulación, la interfaz solo sirve como puente de comunicación. En la Figura 12 (b) se observa que la distribución de la estructura de pruebas correspondiente al "Testbench" se encuentra mayoritariamente en la interfaz, mientras que el bloque de control se encuentra distribuido entre interfaz y simulación. Por otro lado en el diagrama de la Figura 12 (c) se observa que la estructura de pruebas "Testbench" se encuentra en su mayoría en la emulación, mientras que el bloque de control se encuentra distribuido entre la emulación y la simulación, la interfaz solo funciona como puente de comunicación, este tipo de ordenamiento sirve para la implementación y ejercitamiento de estructuras "Built-In-Self-Test" (BIST). Posiblemente en el diagrama de la Figura 12 (d) encontremos el caso más interesante, pues se trata del seccionamiento del

dispositivo bajo prueba y distribución de éste en diferentes secciones del esquema de comunicación, cabe resaltar que la distribución sobre diferentes entornos de emulación se refiere a la capacidad del esquema para adaptarse al uso de múltiples tarjetas de emulación. Como podemos observar la manera en que puede ser ordenada la estructura de prueba en el esquema de comunicación propuesto es muy diversa, y en gran medida depende del tipo de prueba que se desee realizar.

3.3.3 Generación de archivos fuente

En esta sección se describe la generación de los archivos fuente para Simulación, Interfaz y Emulación. A continuación, se enuncian los archivos para cada uno de los bloques que componen el esquema de comunicación.

- *Simulación*: La descripción de los elementos de simulación se realiza mediante archivos SystemVerilog.
- *Interfaz*: La interfaz se dividide en tres bloques de ejecución donde el primer bloque utiliza archivos C, el segundo es un conjunto de archivos Python y el tercero utiliza un único archivo Python.
- *Emulación*: Para la emulación se utiliza un archivo de configuración Bitstream, además se realiza la generación de un archivo 'tcl' donde se describen las características generales del diseño.

3.3.3.1 Simulación e Interfaz

Los archivos fuente para Simulación e Interfaz pueden ser generados a partir de un block de notas. A continuación, se describe el procedimiento para la creación de archivos fuente a partir de block de notas.

1. Abrir Editor de Textos (Block de Notas).
2. Editar código fuente.

   En este paso se realiza la escritura de código, la descripción es realizada de acuerdo con un lenguaje de programación, esto es si se utiliza para la simulación el código debe ser descrito en código SystemVerilog, si se realiza para el primer bloque de la interfaz la descripción debe ser en código C, y por último si se utiliza para la descripción de los bloques restantes de la interfaz se debe realizar utilizando código Python.
3. Guardar archivo y asignar extensión.

   En este paso se asigna la extensión del archivo, según el lenguaje en el que haya sido descrito y se guarda el archivo en un directorio. Esto es, si el archivo contiene descripción en código SystemVerilog, se debe asignar la extensión '.sv', si se encuentra descrito en código C, la extensión asignada es '.c', y si se encuentra en código Python, se asigna '.py'.

La edición de código mediante el uso de block de notas es usada típicamente por programadores experimentados que ya tienen experiencia trabajando de esta manera. Es recomendable para nuevos programadores el uso de editores de código fuente, pues poseen características que aceleran y facilitan la edición de códigos fuente tales como resaltado de sintaxis, autocompletar, pareo de llaves, entre otros.

3.3.3.2 Emulación

Los archivos de emulación se generaron mediante Vivado 2016.1, ejecutándose en un entorno basado en Linux. A continuación, se realiza una descripción de los pasos a seguir para la generación de los archivos Bitstream y TCL, desde la generación del proyecto.

i. Creación del proyecto

Ubicados en el entorno gráfico de Vivado, la creación del proyecto se realiza mediante el asistente "Create New Project" ubicado en el menú "Quick start", el cual proporciona una plantilla para la creación y configuración del proyecto, donde se elige, el nombre, directorio, especificaciones de la tarjeta y archivos fuente que contendrá el proyecto (opcional).

ii. Creación diagrama a bloques

La creación del diagrama a bloques se realiza por medio de "Create Block Design" ubicado en la sección "IP Integrator" en el menú "Flow Navigator" ubicado a la izquierda del proyecto. Este proporciona un espacio de trabajo donde se puede diseñar la lógica programable a través de la conexión entre bloques de descripción de hardware, llamados IP. Dentro de los IPs que se pueden añadir se encuentran dos tipos:

- Xilinx's IPs: Este tipo de IPs proporcionados por Xilinx, son bloques de descripción de hardware diseñados para facilitar el diseño y verificación, pues contienen la descripción de módulos comúnmente usados que ya han sido probados tales como Memorias, Sumadores, Multiplicadores, etc.
- Custom IPs: Este tipo de IPs son diseñadas por el usuario cuando desea realizar la descripción de hardware de un módulo que no se encuentra definido en las IPs provistas por Xilinx.

iii. Creación IP personalizada

La creación se realiza por medio del asistente para la creación y empaquetado de IPs, "Create and Package IP…" ubicado en el menú "Tools". Cuando se crea una IP utilizando el asistente para la creación de una IP personalizada, se genera un

proyecto plantilla mediante el cual podemos acceder a la lectura y escritura de registros en el sistema de procesamiento, PS, por parte de la IP, además se puede realizar el control de interrupciones (IP a procesador ZYNQ) a través de registros mapeados en memoria y el empaquetado de la IP. El proyecto está constituido por 3 documentos HDL y un documento XML. Además, se pueden añadir archivos HDL con la lógica del usuario y enlazarlos con los demás archivos.

iv. Diseño del diagrama a bloques

El diseño del diagrama a bloques consiste en la integración de IPs proporcionadas por Xilinx e IPs diseñadas por el usuario, interconectadas entre sí, esto definirá el comportamiento de la lógica programable. Sin embargo, debido a como está diseñado PYNQ para el intercambio de información entre el sistema de procesamiento y la lógica programable, el diseño siempre debe contener dos bloques. Estos son:

- Procesador ZYNQ (Processing_System7).
- Sistema de control de interrupciones (System_Interrups).

La conexión de estos dos bloques con las IPs que componen el funcionamiento de la lógica programable se realiza de manera automática mediante la herramienta "Run connection Automation", que realiza la conexión de los buses para el manejo de interrupciones y el manejo de registros de cada una de las IPs al sistema de procesamiento, además añade el sistema de procesamiento de "reset" y el sistema para el manejo de periféricos.

Una vez diseñado el diagrama a bloques se guardan los cambios, con el comando CTRL+S.

v. Pasos Pre-Bitstream

- Generación de “Output Products”.

  Una vez que el diseño del diagrama a bloques se ha completado, se procede a generar los productos de salida para síntesis, implementación y simulación, a fin de integrar el diseño dentro de un archivo “Top-Level”. Esto es, se describen las conexiones entre los bloques que contiene el diagrama a bloques en un archivo HDL. Este archivo estará descrito en el lenguaje definido al momento de crear el proyecto.

  Para la generación de los “Output Products” nos ubicamos en la carpeta “Design Sources”, en el panel “Sources” ubicado en la ventana “Project Manager”, seleccionamos el diagrama a bloques, damos clic derecho, y elegimos la opción “Generate Output Products”.

- Creación HDL “wrapper”.

  Un diagrama a bloques puede ser integrado dentro de un diseño de nivel superior, o puede ser establecido como un “Top-Level”. Esto se realiza por medio de un archivo HDL “wrapper”, que se encarga de realizar la descripción de las conexiones del bloque en un archivo HDL. Para la generación de un HDL “wrapper” seleccionamos el diagrama a bloques, damos clic derecho y elegimos “Create HDL wrapper”.

vi. Síntesis

La síntesis es el proceso encargado de realizar la interpretación del código HDL a estructuras físicas en el FPGA, además de verificar la correcta descripción de los bloques y las interconexiones. Para ejecutar el proceso de síntesis damos clic en "Run Synthesis" ubicada en el panel "Flow Navigator" en la sección "Synthesis".

vii. Implementación

En la implementación se busca que el diseño no incumpla restricciones físicas del dispositivo. Esto es, que el diseño no sobrepase el número de "slices" disponibles en la tarjeta, o que la asignación de pines para el caso de los puertos no cause conflictos de voltaje. Para ejecutarla nos ubicamos en el panel "Flow Navigator" en el menú "Implementation" y damos clic en "Run Implementation".

viii. Bitstream

El archivo de configuración Bitsream contiene información para la colocación física de CLBs, IOBs, TBUFs (3-state buffers), pines y elementos de ruteo. Para la generación del archivo nos ubicamos en el menú "Flow navigator" en la sección "Program and Debug" y damos clic en "Generate Bitstream". El archivo puede ser encontrado en el directorio del proyecto en la subcarpeta …/Ejemplo.runs/impl1_1.

ix. TCL

El archivo TCL del proyecto contiene información sobre la ubicación en memoria de los buses de cada una de las IPs que contiene el diagrama a bloques, así como del controlador de interrupciones. Para su generación es necesario abrir el diagrama a bloques e ir al menú file/export y dar clic en "export block design".

3.3.4 Ubicación de archivos

Los archivos fuentes generados para el esquema de comunicación son ubicados en directorios en el Host-PC y el sistema de procesamiento, PS, de la tarjeta PYNQ-Z1, ver Figura 13.

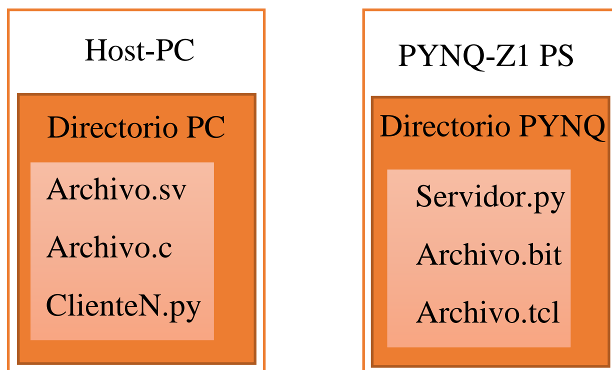


Figura 13 Distribución de archivos fuente.

La ubicación de los archivos correspondientes al Host-PC, se realiza por métodos convencionales. Esto es:

- Crear directorio
- Copiar archivo de directorio fuente.
- Pegar archivo en directorio destino.

Sin embargo, para realizar la ubicación de los archivos en el sistema de procesamiento, PS, de la tarjeta PYNQ–Z1 al no contar con un entorno gráfico, ni los periféricos necesarios para manipulación de archivos remotos, se recurre al uso de "Jupyter Notebooks", una plataforma web que permite el acceso al sistema de procesamiento, PS, para la creación de directorios, así como él envió de archivos desde el Host-PC hacia el sistema de procesamiento, PS.

En la Figura 14 se muestra la ventana de "Jupyter Notebooks". Para realizar la creación de una carpeta damos clic en Folder en el directorio New ubicado a la derecha de la

ventana. Para pasar un archivo del Host-PC hacia el sistema de procesamiento solo basta con dar clic en “Upload” ubicado a la derecha de la ventana.

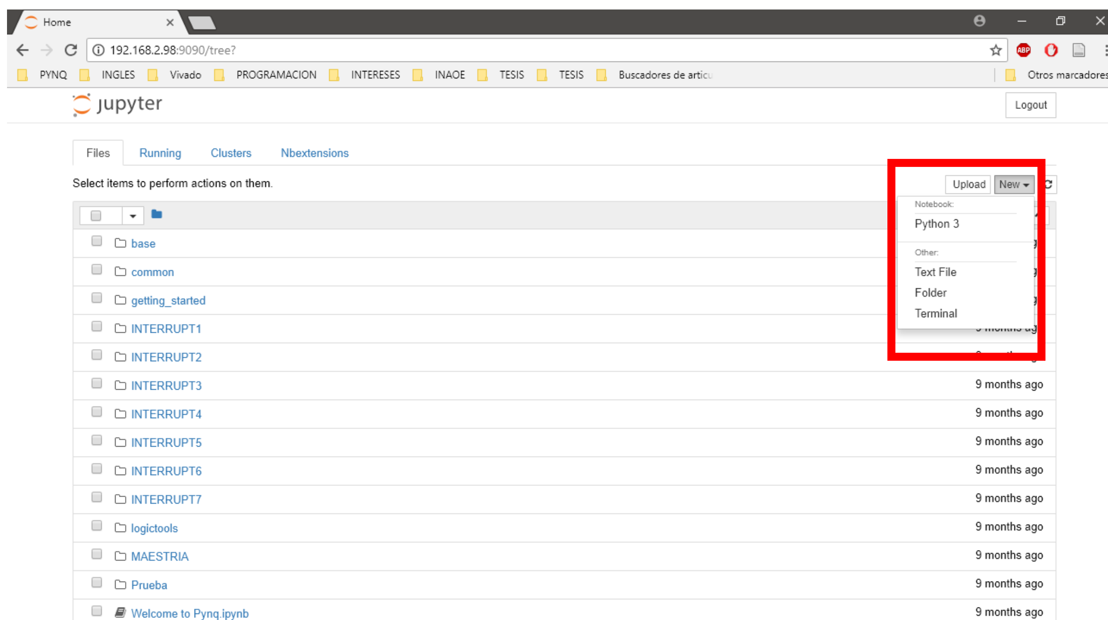


Figura 14 Ventana “Jupyter Notebooks”.

3.3.5 Arranque del Servidor

A través de una terminal Linux en el Host-PC, accedemos al sistema de procesamiento, PS, mediante el siguiente comando:

```
$    ssh   root@192.168.2.99
```

Una vez es ejecutado el comando anterior, la terminal en el Host-PC se puede ver como una terminal propia del sistema de procesamiento, PS, los comandos que sean ingresados de aquí en adelante serán ejecutados en el sistema de procesamiento de la tarjeta y no en el Host-PC.

Mediante el siguiente comando accedemos al directorio donde se encuentran ubicado el archivo fuente del servidor:

```
$    cd    path/Directorio
```

Para arrancar el servidor se utiliza el siguiente comando:

```
$    python3.6  Servidor.py
```

Una vez arrancado el servidor se puede cerrar la terminal del Host-PC

3.3.6 Inicialización de la Simulación

La inicialización de los archivos necesarios para la simulación se realizó a través de la consola TCL. Para acceder a la consola TCL es necesario ejecutar el entorno gráfico de Vivado mediante una terminal Linux, donde se ingresan los siguientes comandos:

```
$    source /opt/Xilinx/Vivado/2016.1/settings64.sh

$    Vivado
```

Una vez ejecutado el entorno gráfico de Vivado centraremos nuestra atención en el bloque correspondiente a la consola TCL (Ver Figura 15).

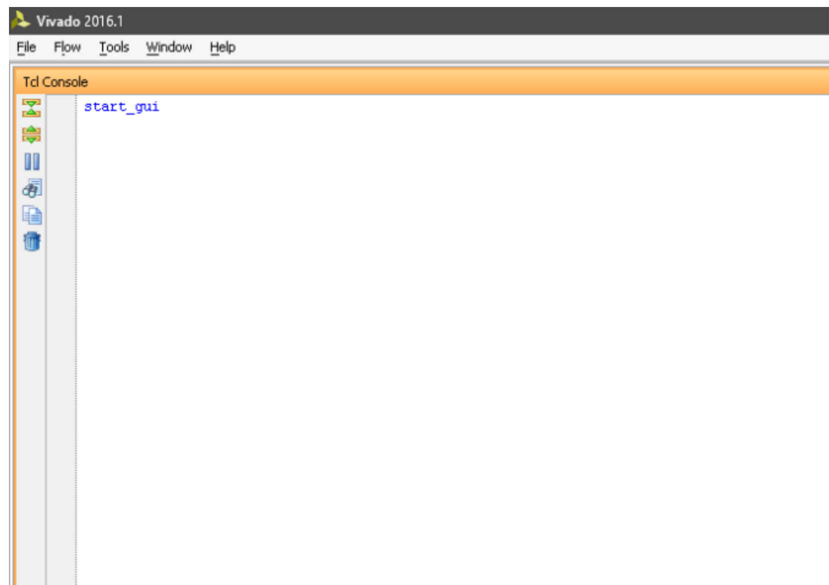


Figura 15 Entorno gráfico Vivado: Consola TCL.

Una vez se realiza al acceso a la consola TCL mediante el entorno gráfico de Vivado, procedemos a ubicarnos en el directorio de trabajo (Lugar donde se encuentran los archivos fuente) ingresando el siguiente comando en la consola TCL.

```
$    cd   path/Directorio
```

Debido a que el intercambio de información entre SystemVerilog con C es llevado a cabo mediante el uso de DPI-C, surge la necesidad de crear bibliotecas compartidas DLL/SO, para permitir el acceso a las funciones descritas en C. Esto se realiza por medio del siguiente comando.

```
$    xsc –v 1 file.c --additional_option –lpython2.7
```

Cabe resaltar que, debido al uso de Python Embebido para realizar el llamado al servidor en el sistema de procesamiento, PS, es necesario indicar al compilador que se usará el "Integrated Development and Learning Environment (IDLE)” de Python, esto es realizado mediante `--additional_option –lpython2.7.`

Una vez se han generado el archivo DLL, se procede a generar el “snapshot”, un archivo que permite la ejecución de la simulación. Esto se realiza mediante el siguiente comando.

```
$    xelab –v 1 file.sv –sv_lib dpi
```

Por último, se procede a iniciar y cargar el archivo de simulación “snapshot” en el simulador, esto se realiza mediante el siguiente comando.

```
$    xsim –g work.snapshot
```

3.3.7 Ejecución de la Co-emulación

Como se mencionó en el planteamiento a nivel interacción en la sección 3.2.1 el intercambio de información es llevado a cabo bajo directivas Maestro-Esclavo donde la simulación cumple el papel de Maestro y la Emulación el papel de esclavo. Por ello para iniciar la ejecución de la co-emulación, se debe iniciar por la ejecución de la simulación. En el siguiente diagrama se muestra el flujo de ejecución del esquema de co-emulación.

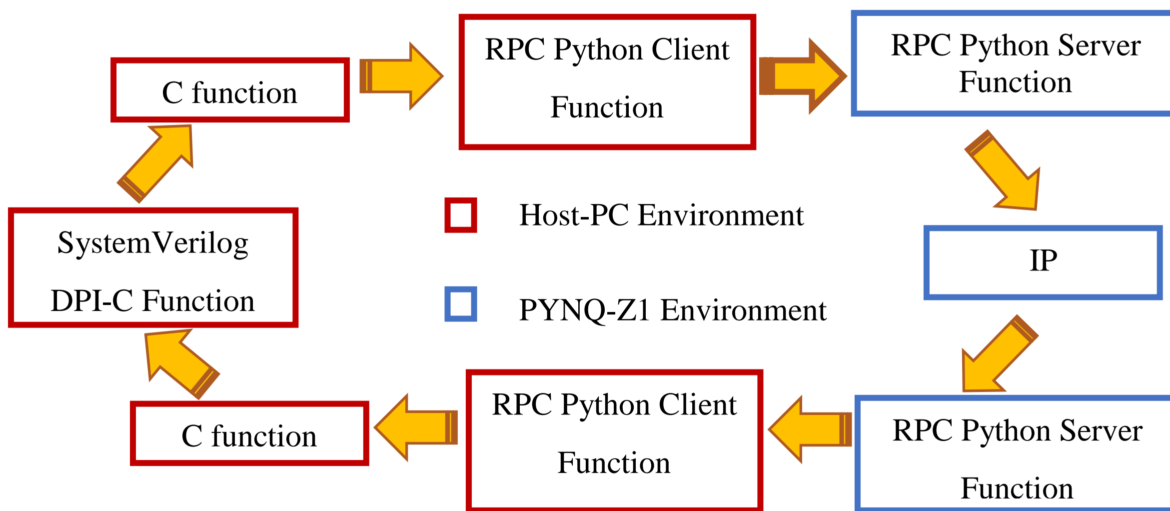


Figura 16 Flujo de ejecucion del modelo de co-emulación.

Como se puede ver en la Figura 16, el modelo de co-emulación parte de la ejecución de una función DPI-C llamada desde SystemVerilog, la cual realiza un llamado a una función en el servidor corriendo en el sistema de procesamiento, PS, de la tarjeta PYNQ-Z1, mediante funciones cliente RPC descritas en archivos Python, llamadas mediante Python Embebido. El servidor corriendo en el sistema de procesamiento, PS, por medio de registros mapeados en memoria se encarga de enviar y recibir información a/de la lógica programable, PL, y regresarla a la simulación terminando así el llamado de la función.

La ejecución de la simulación se lleva a cabo mediante el comando “run”, el cual tiene como parámetros de entrada un número entero y una medida de tiempo que puede ser nanosegundos, picosegundos, etc. A continuación, se muestra un ejemplo para correr 1 nanosegundo de simulación.

```
$    run 1ns
```

3.4 Funciones para el intercambio de información

Como se mencionó en la sección 3.3.2 Planteamiento de la estructura de verificación, el esquema de comunicación propuesto es bastante flexible, pues permite que cada uno de los elementos que componen la estructura de prueba, así como el dispositivo bajo prueba sea implementado mediante SystemVerilog, Python o HDL sintetizado. Es importante resaltar que cada uno de los entornos, ya sea SystemVerilog, C, Python, Vivado, proveen diferentes estructuras para realizar la descripción de los elementos de la estructura de pruebas y en gran medida depende de la experiencia del programador para realizar la descripción; además del propósito del diseño digital. Por ello no se puede dar como tal una estructura general para su implementación, ni realizar una descripción de las diferentes estructuras de cada uno de los lenguajes utilizados, pues no es intención del presente trabajo ser una guía del usuario en SystemVerilog, C, Python o Vivado. Sin embargo, algo que permanece constante es el intercambio de información entre las diferentes capas del esquema de comunicación propuesto. En este sentido, cada uno de los elementos que componen los bloques del esquema de comunicación puede ser agrupado de tres maneras: elemento de la estructura de pruebas o del dispositivo bajo prueba, o elemento de comunicación. En el diagrama de la Figura 17,

se muestra como se encuentran conectadas las interfaces entre simulación, interfaz y emulación.

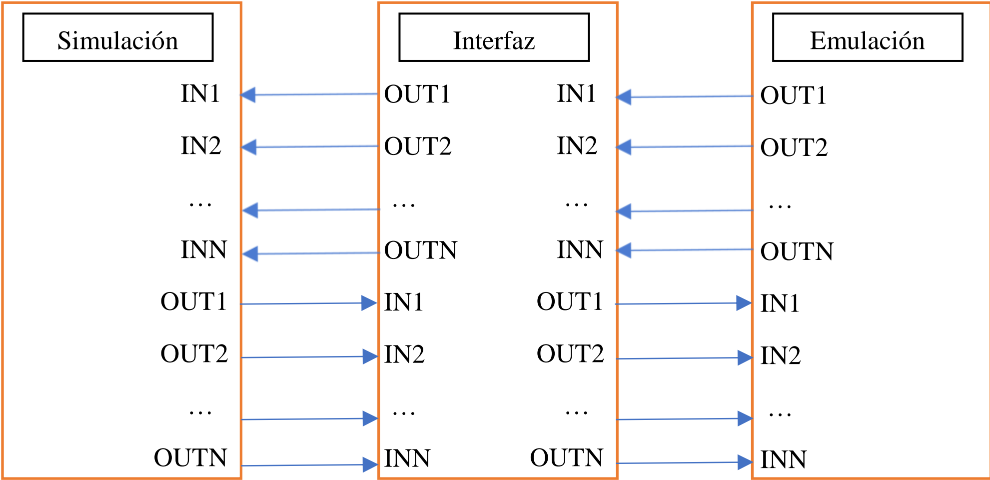


Figura 17 Conexión de puertos entre simulación, interfaz y emulación.

Como tal las interfaces no se encuentran comunicadas y en permanente estado de comunicación, esto es llevado a cabo por medio de funciones llamadas desde simulación, que conectan temporalmente los puertos para el intercambio de información. A continuación, se listan las funciones base para el intercambio de información (ver figura 18)

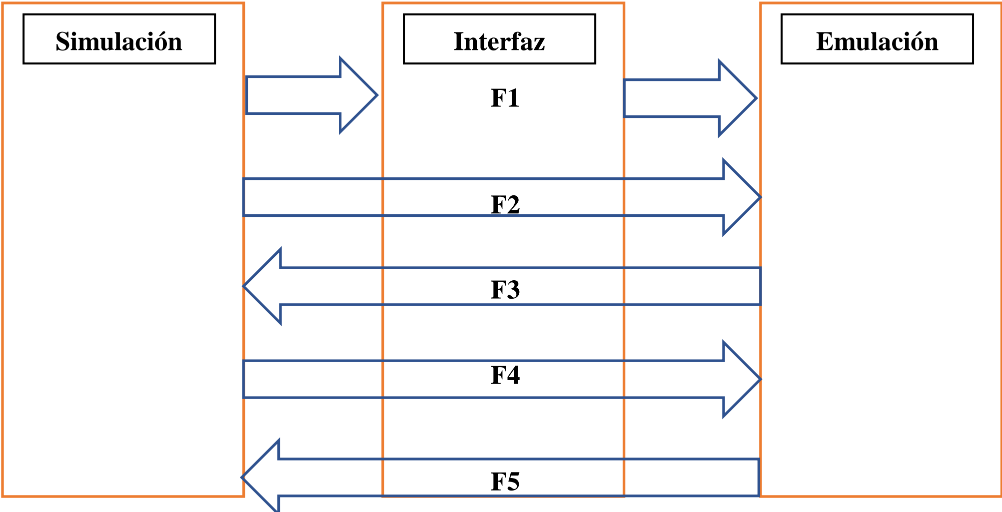


Figura 18 Funciones básicas para el intercambio de información.

Donde:

- F1 representa funciones con cero argumentos de entrada y cero argumentos de retorno del tipo Func();
- F2 representa funciones con un argumento de entrada y cero argumentos de retorno del tipo Func(In);
- F3 representa funciones con cero argumentos de entrada y un argumento de retorno del tipo Func(Out);
- F4 representa funciones con múltiples argumentos de entrada y cero argumentos de retorno del tipo Func(In1, In2, In3, …, Inn);
- F5 representa funciones con cero argumentos de entrada y múltiples argumentos de retorno del tipo Func(Out1, Out2, Out3, …, Out4);

Cabe resaltar que se pueden crear funciones compuestas: una entrada y múltiples salidas, múltiples entradas y una salida, etc. En el Apéndice A. Código Casos de Estudio.se muestran los códigos correspondientes a los cuatro casos de estudio, donse se muestra el uso de funciones compuestas.

Capítulo 4 Diseño e implementación de casos de estudio

4.1 Introducción

Como se describió en la sección 1.3, este trabajo de investigación se enfoca en el diseño e implementación de un esquema de comunicación que permita el intercambio de información entre hardware y software con propósitos de verificación, aprovechando elementos de ambos entornos. Además del diseño e implementación de casos de estudio usando el esquema de comunicación propuesto. De acuerdo con lo planteado en la sección 3.3.1, el esquema es diseñado en base a dos tipos de circuitos, aquellos de respuesta inmediata y aquellos de respuesta no inmediata. En esta sección se presentan dos casos de estudio, el código no es incluido en esta sección, pero éste se incluye en el Apéndice A. Código Casos de Estudio. La sección 4.2 se enfoca en la descripción de un elemento combinacional, considerado de respuesta inmediata. Por otro lado, la sección 4.3 se enfoca en la descripción de un circuito de respuesta no inmediata de propósito no especifico.

4.2 Co-emulación de circuitos puramente Combinacionales

Debido a que los circuitos Combinacionales son considerados un elemento básico en la construcción de circuitos digitales más complejos, en esta sección se describe un DUT puramente combinacional. La distribución de los elementos de prueba (Testbench), así como del DUT se lleva a cabo mediante el diagrama mostrado en la Figura 19, donde todos los elementos correspondientes al "Testbench" son ubicados exclusivamente en la simulación y su descripción es realizada mediante SystemVerilog, mientras que el DUT es ubicado en la emulación, siendo encapsulado en una IP, descrito mediante VHDL y cargado en la lógica programable de la tarjeta PYNQ-Z1, por su parte la interfaz solo actua como puente de

comunicación, recibiendo y pasando señales de simulación a emulación y viceversa. (ver A.1 Caso de estudio uno).

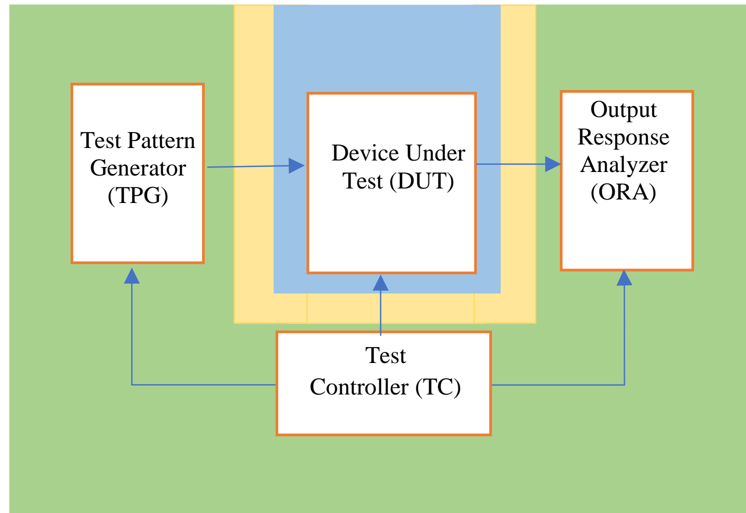


Figura 19 Distribución de estructura de pruebas y DUT: Ejemplo 1.

4.2.1 Simulación: Ejemplo1.

Como se muestra en el diagrama de la Figura 19, los elementos correspondientes a la simulación son: "Test Controller (TC)", "Test Pattern Generator (TPG)" y "Output Response Analyzer (ORA). La descripción de cada uno de los elementos es realizada por medio de estructuras proporcionadas por SystemVerilog. A continuación, se realiza una descripción de dichos elementos.

Test Controller (TC): El control de la prueba fué realizada por medio de un proceso `initial`, donde se aprovecharon estructuras de control no sintetizables para realizar la generación de patrones de prueba, así como para realizar la comparación de las salidas del emulador.

Test Pattern Generator (TPG): El generador de señales se realizó mediante dos ciclos `for` anidados cuya función es proveer el selector SEL y las entradas A, B.

Output Response Analizer (ORA): El analizador de respuesta de salida se realizó por medio de dos elementos; un bloque combinacional descrito en la simulación que se encarga de generar la salida esperada de acuerdo con las entradas enviadas a la emulación, y una sentencia condicional `if` que compara el resultado obtenido de la emulación y la respuesta obtenida del bloque combinacional en la simulación.

El intercambio de información entre simulación y emulación se lleva a cabo por medio de dos funciones. La primera carga un archivo de configuración Bitstream en la lógica programable, PL, de la tarjeta PYNQ-Z1, y la segunda envia y recibe las entradas.

```
Programar ();
```

Cuando esta función es llamada en un proceso `initial`, se accede indirectamente a través de C a una función en el sistema de procesamiento, PS, de la tarjeta y se carga el archivo de configuración Bitstream en la lógica programable, PL, de la tarjeta PYNQ Z1. Específicamente esta función en la capa SystemVerilog realiza el llamado a la función `Programar ();` descrita en la capa C.

```
Opera (A, B, SEL, C);
```

Cuando esta función es llamada en un proceso `initial`, se realiza él envió indirecto de las señales de entrada A, B y el selector SEL hacia la emulación, además se bloquea la simulación hasta recibir una respuesta del valor de C, entonces se finaliza el llamado de la función. Específicamente esta función en la capa SystemVerilog realiza el llamado a la función `Opera (A, B, SEL, C);` descrita en la capa C.

En el diagrama de la Figura 20 se muestran los elementos que constituyen el archivo de simulación.

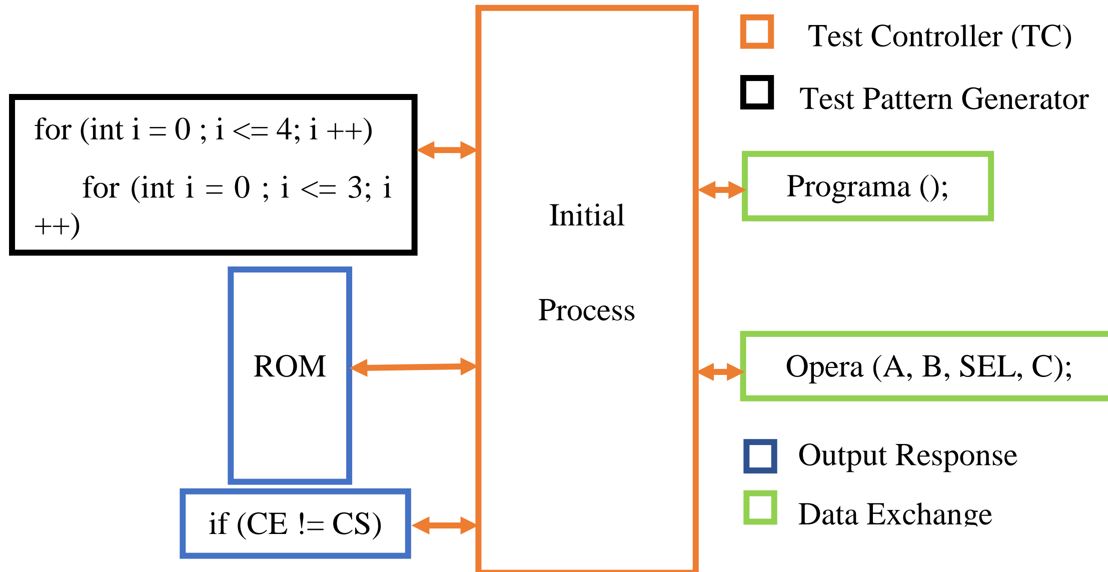


Figura 20 Estructura de simulación: Ejemplo1.

Desde el punto de vista Maestro-esclavo, el intercambio de información se controla desde simulación mediante el bloque `initial`, el cual se encarga de la coordinación de los demás módulos para la correcta ejecución de la prueba. En el diagrama de la Figura 21 se muestra el flujo de ejecución del bloque `initial`.

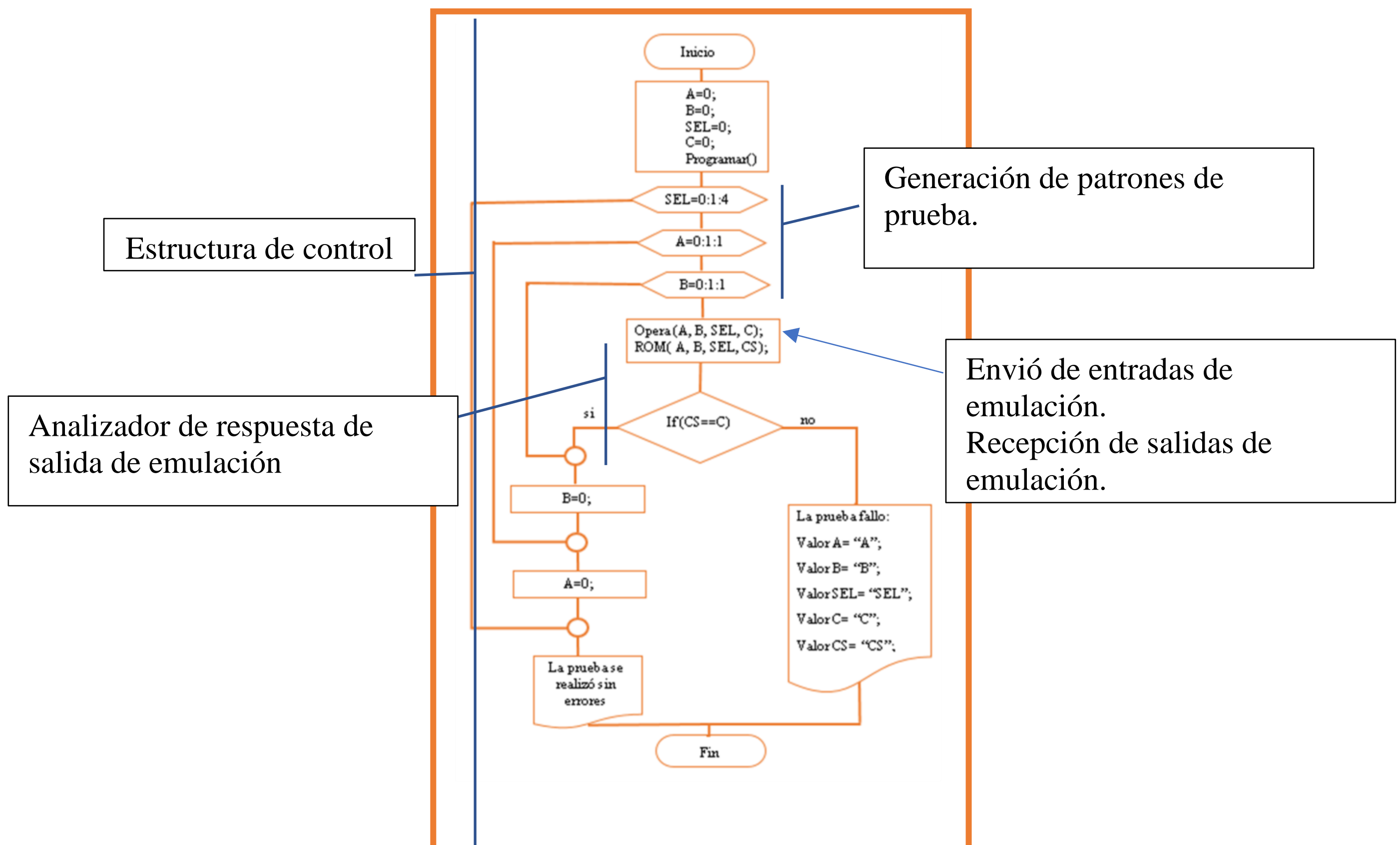


Figura 21 Diagrama de flujo "Test Controller (TC)": Ejemplo1.

4.2.2 Interfaz: Ejemplo 1.

Como se muestra en el diagrama de la Figura 19, la interfaz es utilizada exclusivamente como puente de comunicación. Además, se establecen dos funciones por parte de simulación, para realizar el intercambio de información con la emulación. Por lo que cada capa del esquema de comunicación contará con la descripción de dichas funciones. En el diagrama de la Figura 22 se muestran las tres capas que contiene la interfaz y las funciones correspondientes a cada capa.

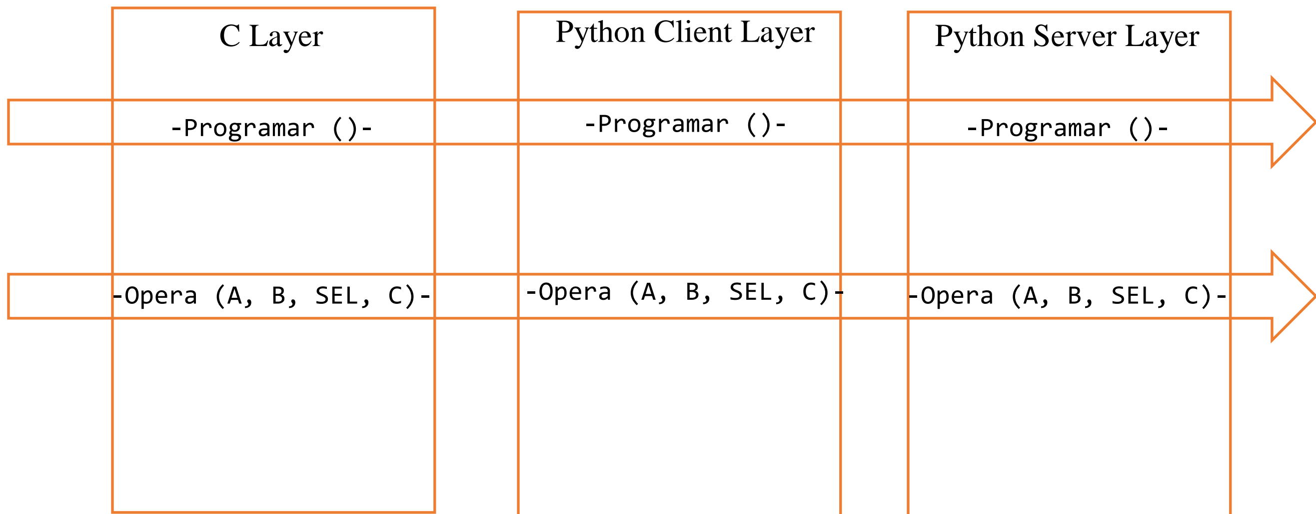


Figura 22 Funciones en las tres capas de la interfaz: Ejemplo 1.

Cabe resaltar que las funciones en la capa C, son llamadas por medio de SystemVerilog a través de DPI y su propósito es acceder a las funciones del servidor Python, mediante el llamado de archivos Python cliente por medio de DPI. Por su parte las funciones en la capa Python Cliente, son llamadas desde C, para acceder a funciones en el servidor Python mediante RPC. La función `Programar ();` en el servidor Python se encarga de cargar el archivo de configuración Bitstream en la lógica programable de la tarjeta PYNQ-Z1 mediante el uso del paquete "Overlay.py". Por otro lado, la función `Opera (A, B, SEL, C);` en el servidor Python se encarga de escribir los valores A, B, SEL en el SLV_REG0, SLV_REG1, SLV_REG2 respectivamente y toma el valor de salida del SLV_REG3 y lo asigna a la variable C, esto se hace por medio de la librería "MMIO.py".

4.2.3 Emulación: Ejemplo 1.

En la Figura 23, se muestra el diagrama a bloques utilizado para realizar la generación del archivo de configuración Bitstream y el archivo TCL. El diagrama está constituido por cinco bloques: 1) ZYNQ7 Processing System, 2) Processor System Reset, 3) AXI

Interconnect, 4) AXI Interrupt Controller y 5) combinacional_V1_0 (Pre_production). Los primeros cuatro bloques son provistos por Xilinx como una plataforma para permitir el intercambio de información entre el sistema de procesamiento, PS, y la lógica programable, PL, de la tarjeta PYNQ Z1, mientras que el último es una IP personalizada que contiene la lógica del usuario.

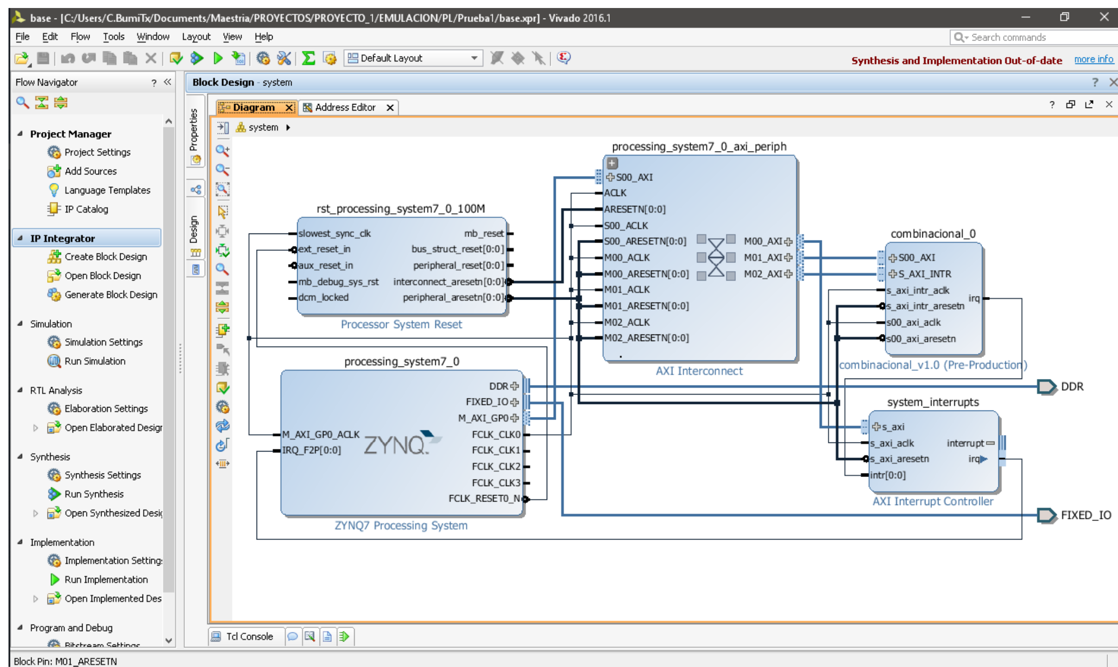


Figura 23 Diagrama a bloques: Ejemplo 1.

En la Figura 24, se muestra el diagrama RTL resultante del diagrama a bloques de la Figura 23, donde el bloque resaltado en azul representa la IP personalizada “combinacional_V1_0 (Pre_Production)”.

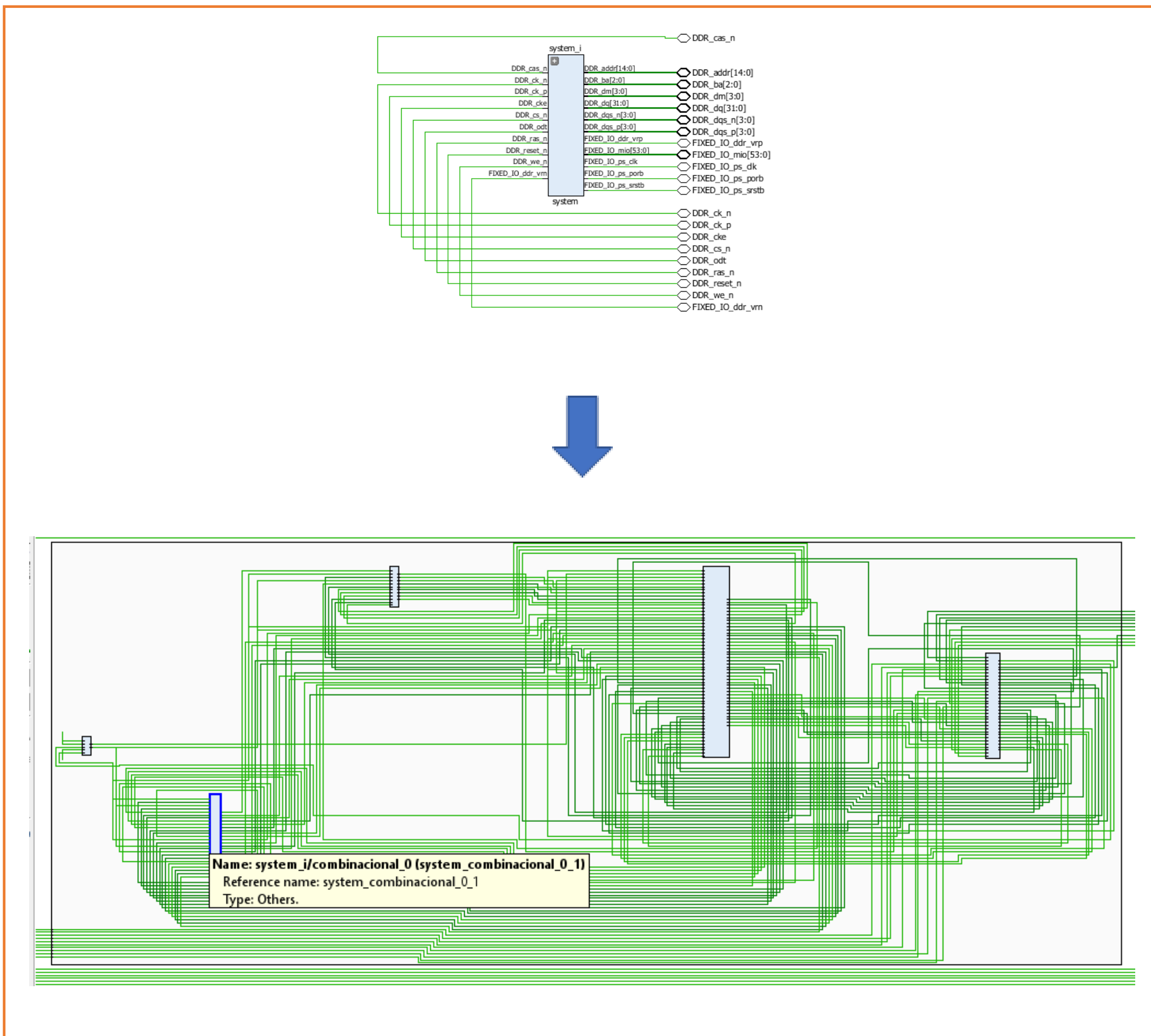


Figura 24 Diagrama RTL: Ejemplo 1

En la Figura 25 se muestra el bloque correspondiente a la IP personalizada resultante al usar el asistente para la creación y empaquetado de IPs “Create and Package New IP…”. Como se observa, la IP cuenta con acceso tanto al Bus S00_AXI, como al Bus S00_AXI_INTR, sin embargo, no cuenta con ningún puerto accesible por medios físicos tales como Switches, Leds, o cualquier otro periférico de la tarjeta.

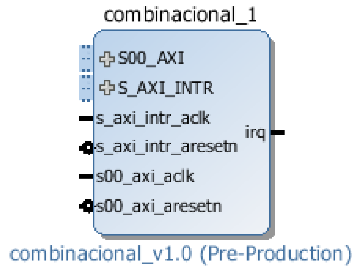


Figura 25 IP Personalizada: Ejemplo1.

Internamente el bus S00_AXI está amarrado a la lógica del usuario mediante el uso de cuatro registros "slv_reg", en esta ocasión no se usó el bus S00_AXI_INTR, pues se considera que este es un circuito de respuesta inmediata y por tanto superior en velocidad a la simulación. En la Figura 26, se muestra cómo se encuentran conectados los registros del Bus S00_AXI a la lógica del usuario, además se indica la dirección de inicio del Bus S00_AXI para la IP, y el OFFSET de cada registro "slv_reg".

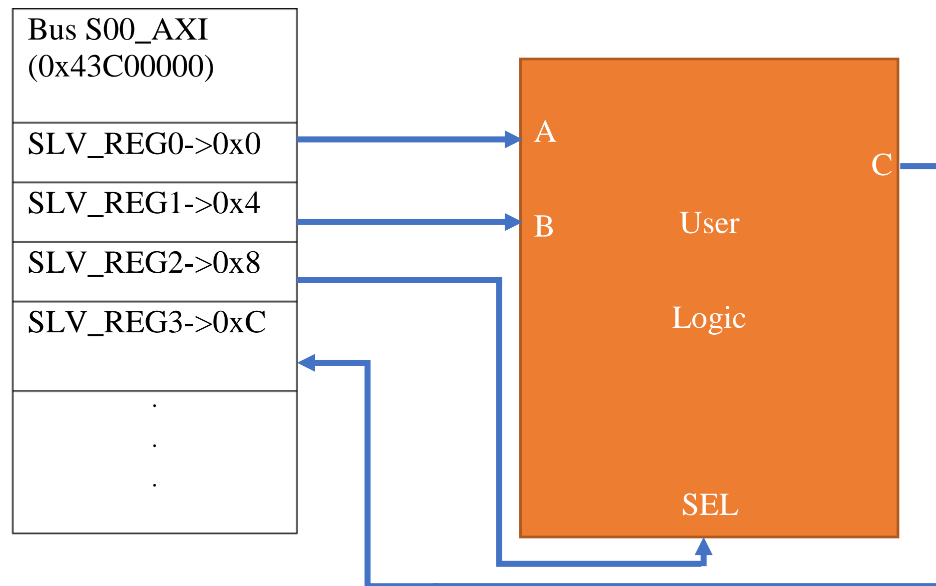


Figura 26 Mapeo interno de registros de la IP: Ejemplo1.

A continuación, se listan las características de la lógica del usuario:

- La longitud de la entrada A y B es de un bit.
- La longitud del selector SEL es de tres bits.
- La salida C es de longitud de un bit.
- La salida es controlada de acuerdo con la siguiente tabla.

Tabla 1 Lógica DUT ejemplo 1.

SEL	C
0x00	A AND B
0x01	A OR B
0x02	A XOR B
0x03	A NAND B
0x04	A NOR B
Others	‘0’

Un aspecto importante para resaltar es la diferencia que existe entre la longitud de los puertos de la lógica del usuario y la longitud de los registros “slv_reg”, esto se logra mediante el uso de una variable auxiliar que sirve como medio para rellenar los bits faltantes con ceros, esto se realiza por medio de una concatenación.

En la Figura 27, se muestra el diagrama RTL de la IP resultante, donde el bloque rojo es el encargado del acceso a los registros para el manejo de interrupciones, mientras que el bloque en rosa se encarga del manejo de los registros para la transferencia de datos entre el sistema de procesamiento, PS, y la IP.

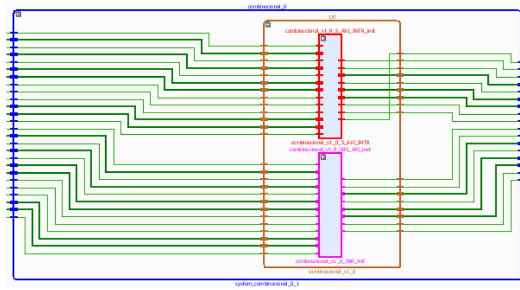


Figura 27 Diagrama RTL IP personalizada: Ejemplo 1.

En la Figura 28, se muestra como está constituido el bloque rosa encargado de la lectura y escritura de registros para el intercambio de información. El bloque resaltado en rojo representa la lógica del usuario.

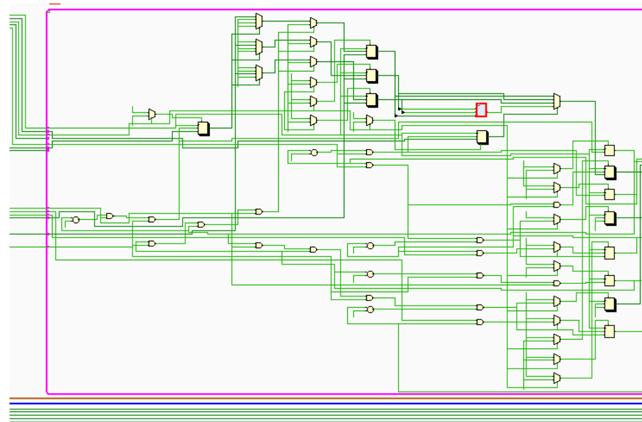


Figura 28 Diagrama RTL, Bus S00_AXI IP personalizada: Ejemplo1.

A continuación, se muestra el diagrama RTL de la lógica del usuario. El cual está constituido a partir de compuertas lógicas y un multiplexor.

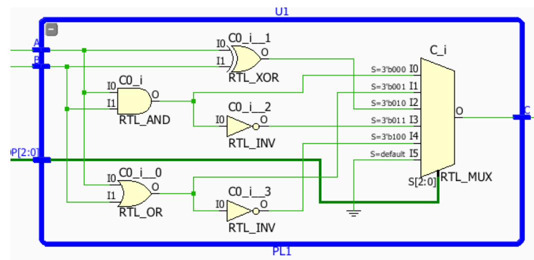


Figura 29 Diagrama RTL Lógica del usuario: Ejemplo 1.

4.3 Co-emulación de circuitos secuenciales

Los circuitos secuenciales constituyen otro elemento básico en la construcción de circuitos digitales más complejos, por ello en esta sección se define como DUT, un elemento mixto compuesto por un bloque combinacional y un bloque secuencial, donde la ejecución llega a tomar varios ciclos de reloj. En el diagrama de la Figura 30 se muestra la distribución de los elementos correspondientes a la estructura de prueba "Testbench", así como del dispositivo bajo prueba. El bloque "Test Controller" (TC) es distribuido en dos partes entre simulación y emulación. Por su parte el dispositivo bajo prueba se encuentra confinado exclusivamente a emulación, la interfaz solo actúa como puente de comunicación. Cabe resaltar que la distribución de la etapa de control se realizó de esa forma, puesto que el circuito seleccionado para la co-emulación puede ser clasificado como de respuesta no inmediata, de propósito no especifico, necesitando así circuitería extra. (ver A.2 Caso de estudio dos)

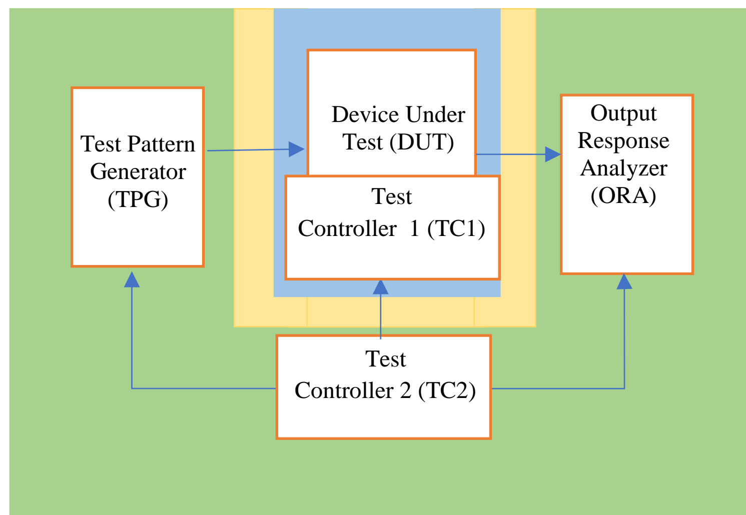


Figura 30 Distribución de estructura de pruebas y DUT: Ejemplo 2.

4.3.1 Simulación: Ejemplo 2

Como se muestra en el diagrama de la Figura 30, los elementos correspondientes a la simulación son: “Test Controller 2 (TC2)”, “Test Pattern Generator (TPG)” y “Output Response Analyzer (ORA). La descripción de cada uno de los elementos es realizada por medio de estructuras proporcionadas por SystemVerilog. A continuación, se realiza una descripción de dichos elementos.

- *Test Controller 2 (TC2)*: El control de la prueba en la simulación se realizó por medio de un proceso `initial`, este cuenta con acceso a dos funciones que le permiten controlar el dispositivo bajo prueba en dos formas distintas, en la primera exclusivamente se realiza el envío de entradas al dispositivo bajo prueba con el propósito de ponerlo en alguno de los cuatro modos de operación, en el segundo este bloque de control se coordina con el bloque de control en la emulación para realizar una prueba coordinada de alguno de los cuatro modos de funcionamiento. Cabe resaltar que se aprovecharon estructuras de control no sintetizables para realizar la comparación entre el resultado de la prueba esperado con el resultado de emulación.
- *Test Pattern Generator (TPG):* La generación de las señales de prueba se realiza por medio de una memoria ROM, donde el recorrido de la tabla es realizado mediante una señal auxiliar SEL controlada por un ciclo `for` en el bloque `initial.` Este bloque se encarga de generar las señales Operador, Carga, Npulsos, además del valor esperado RES. Donde la variable Operador se encarga de la selección de una de las cuatro pruebas disponibles en la emulación, Carga es un valor que dependiendo del tipo de prueba es usado con distintos propósitos, y Npulsos son los pulsos de reloj que debe contar la etapa de control en la emulación dependiendo de la prueba. Por su

parte la variable “ResultadoS” representa el valor esperado de acuerdo con la prueba seleccionada.

- *Output Response Analyzer (ORA):* El analizador de respuesta de salida es realizado por medio de dos elementos; un bloque combinacional descrito en la simulación que se encarga de generar la salida esperada de acuerdo con las entradas enviadas a la emulación, y una sentencia `if`, mediante la cual se compara el resultado obtenido de la emulación y la respuesta obtenida del bloque combinacional en la simulación.

El intercambio de información entre simulación y emulación es llevado a cabo por medio de tres funciones. La primera con el propósito de cargar un archivo de configuración Bitstream en la lógica programable, PL, de la tarjeta PYNQ-Z1; la segunda con el propósito de enviar señales de entrada con la finalidad de poner el dispositivo bajo prueba en un modo de operación, y la tercera con el propósito de mandar a ejecutar una prueba coordinada con el bloque de control descrito en la emulación a uno de los cuatro modos de funcionamiento del dispositivo bajo prueba.

```
Programar ();
```

Cuando esta función es llamada en un proceso `initial`, accede indirectamente a través de C a una función en el sistema de procesamiento, PS, de la tarjeta para cargar el archivo de configuración Bitstream en la lógica programable, PL, de la tarjeta PYNQ Z1. Específicamente esta función en la capa SystemVerilog realiza el llamado a la función `Programar ();` descrita en la capa C.

```
Opera (Operador, Carga);
```

Cuando esta función es llamada en un proceso `initial`, es realizado el envío indirecto de las señales de entrada Operador, Carga hacia la emulación. Esta función, únicamente manda a ejecutar un modo de operación al dispositivo bajo prueba sin esperar ningún resultado de retorno. Específicamente esta función en la capa SystemVerilog realiza el llamado a la función `Opera (Operador, Carga);` descrita en la capa C.

```
Prueba (Operador, Carga, Npulsos, Resultado);
```

Cuando esta función es llamada en un proceso `initial`, se realiza el envío indirecto de las entradas Operador, Carga, Npulsos al bloque "Test Controller 2" (TC2) descrito en la emulación, que se encarga de reconocer el número de prueba, y toma los elementos necesarios para su ejecución. Esta función bloquea la simulación hasta que recibe un resultado de retorno por parte de la emulación en la variable Resultado. Específicamente en la capa de SystemVerilog realiza un llamado a la función `Prueba (Operador, Carga, Npulsos, Resultado);` descrita en la capa C.

En el diagrama de la Figura 31 se muestran los elementos que constituyen el archivo de simulación.

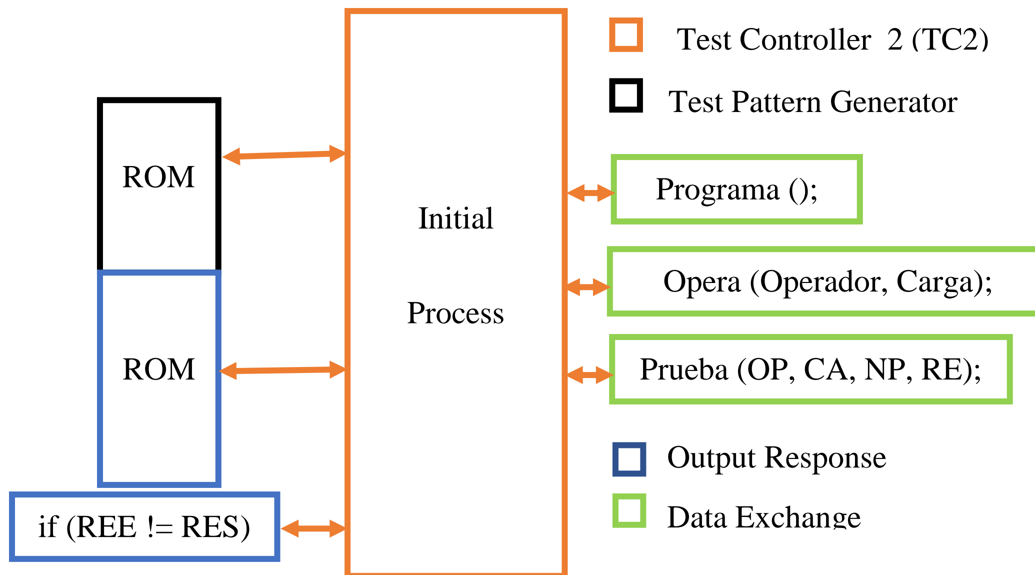


Figura 31 Estructura de simulación: Ejemplo 2.

Desde el punto de vista Maestro-esclavo, la simulación controla el intercambio de información mediante el bloque `initial`, el cual se encarga de la coordinación de los demás módulos para la correcta ejecución de la prueba. Específicamente en este caso, debido a que la función `Opera (Operador, Carga);` tiene el propósito de control del dispositivo bajo prueba, el bloque `initial` es usado también con propósitos de control. En el diagrama de la Figura 32 se muestra el flujo de ejecución del bloque `initial.`

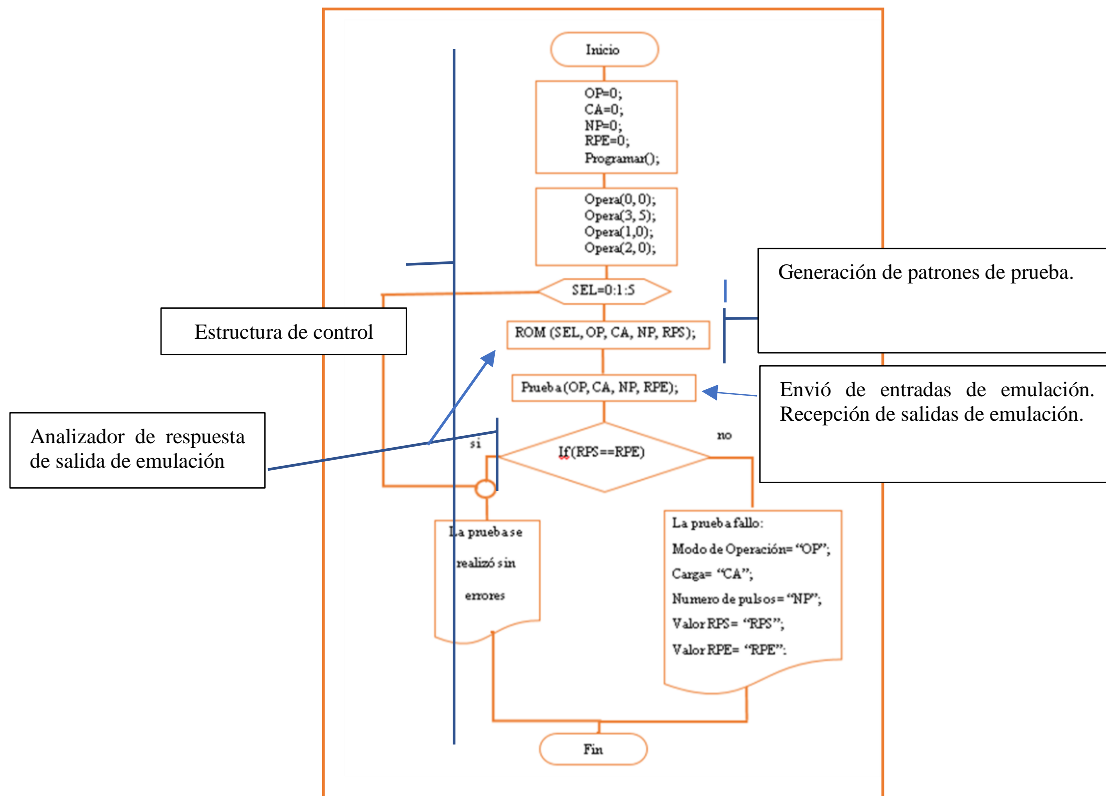


Figura 32 Diagrama de flujo del bloque "Test Controller 1 (TC1)": Ejemplo 2.

4.3.2 Interfaz: Ejemplo 2

Como se muestra en el diagrama de la Figura 30, la interfaz está siendo utilizada exclusivamente como puente de comunicación. Además, se establecieron tres funciones por parte de la simulación, para realizar el intercambio de información con la emulación. Por lo que cada capa de la interfaz en el esquema de comunicación contará con la descripción de dichas funciones. En el diagrama de la Figura 33 se muestran las tres capas que contienen la interfaz y las funciones que corresponden a cada capa.

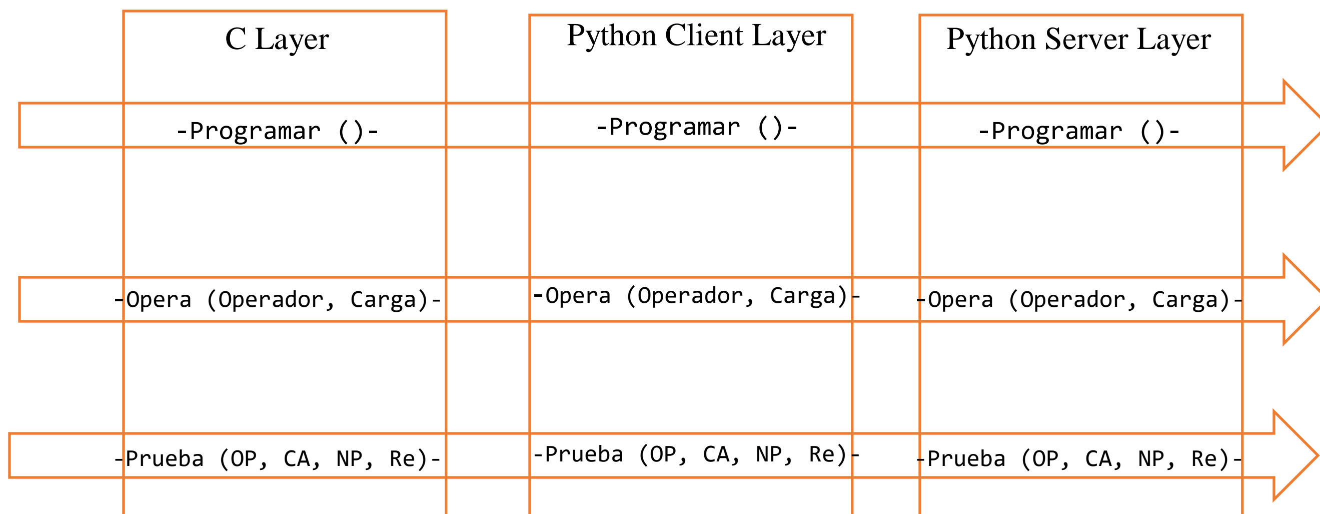


Figura 33 Funciones en las capas de la interfaz: Ejemplo 2.

Como se ha mencionado en el ejemplo uno las funciones descritas en las capas de C y Cliente Python, son utilizadas como medio para pasar argumentos o ejecutar funciones en el servidor Python que se encarga de realizar de la comunicación con la lógica programable de la tarjeta PYNQ-Z1. Por su parte la función Programar (); en el servidor Python se encarga de cargar el archivo de configuración Bitstream en la lógica programable de la tarjeta PYNQ-Z1. Por otro lado, la función `Opera (Operador, Carga);` en el servidor Python se encarga de pasar los argumentos Operador y Carga a los registros SLV_REG2, y SLV_REG3 respectivamente. La función `Prueba (Operador, Carga, Npulsos, Resultado);` se encarga de cargar el valor de 1 en el SLV_REG0, lo que indica al "Test Controller 2 (TC2)" en la lógica programable que se ejecutara una prueba, enseguida asigna los valores OP, CA, NP a los registros SLV_REG2, SLV_REG3, y SLV_REG5, por ultimo crea un flanco en el registro SLV_REG1 pasando su valor de cero a uno, o de uno a cero según sea el caso, lo que da inicio a la prueba. Cabe resaltar que debido a que este tipo de circuito se considera de respuesta no inmediata se utilizaron interrupciones, lo que hace que la función

se quede bloqueada hasta que reciba una señal por parte de la emulación de que se ha concluido la prueba, esto es logrado por medio del Bus S00_AXI_INTR, una vez terminada la prueba toma el valor en el SLV_REG4 y lo asigna a la variable RE, la función termina su ejecución y retorna los resultados hacia la emulación.

4.3.3 Emulación: Ejemplo 2

En la Figura 34, se muestra el diagrama a bloques utilizado para realizar la generación del archivo de configuración Bitstream y el archivo TCL. El diagrama está constituido por cinco bloques: 1) ZYNQ7 Processing System, 2) Processor System Reset, 3) AXI Interconnect, 4) AXI Interrupt Controller y 5) Contador_V1_0 (Pre_production). Donde los primeros cuatro bloques son provistos por Xilinx como una plataforma para permitir el intercambio de información entre el sistema de procesamiento, PS, y la lógica programable, PL, de la tarjeta PYNQ Z1, mientras que el último es una IP personalizada que contiene la lógica del usuario.

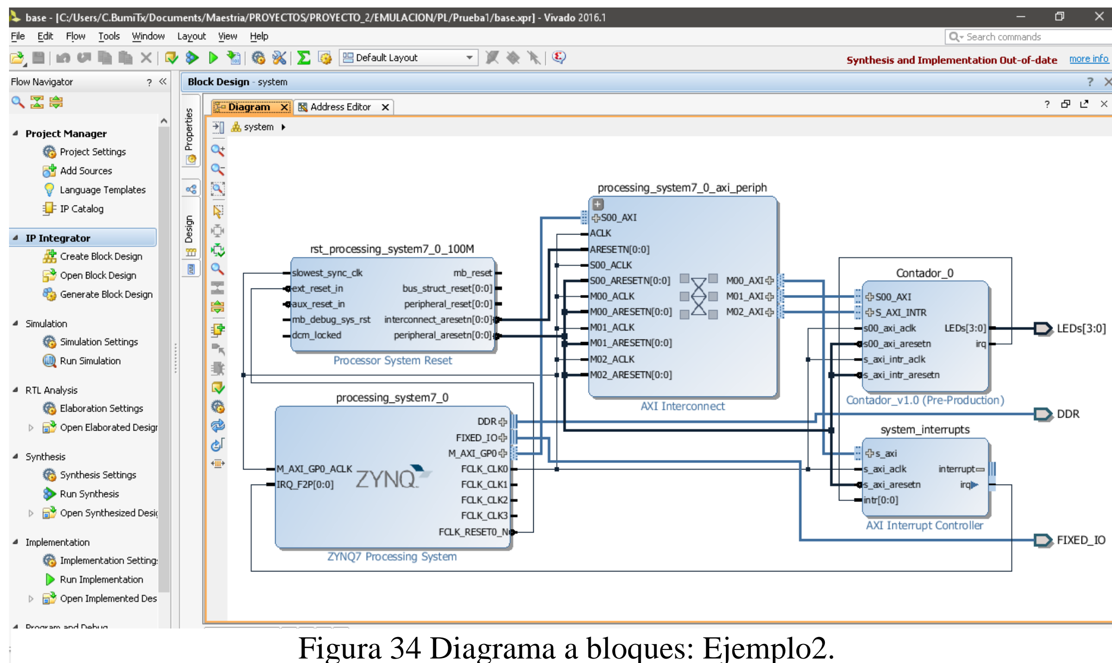


Figura 34 Diagrama a bloques: Ejemplo2.

En la Figura 35, se muestra el diagrama RTL resultante del diagrama a bloques, donde el bloque resaltado en azul representa la IP personalizada “Contador_V1_0 (Pre_Production)”.

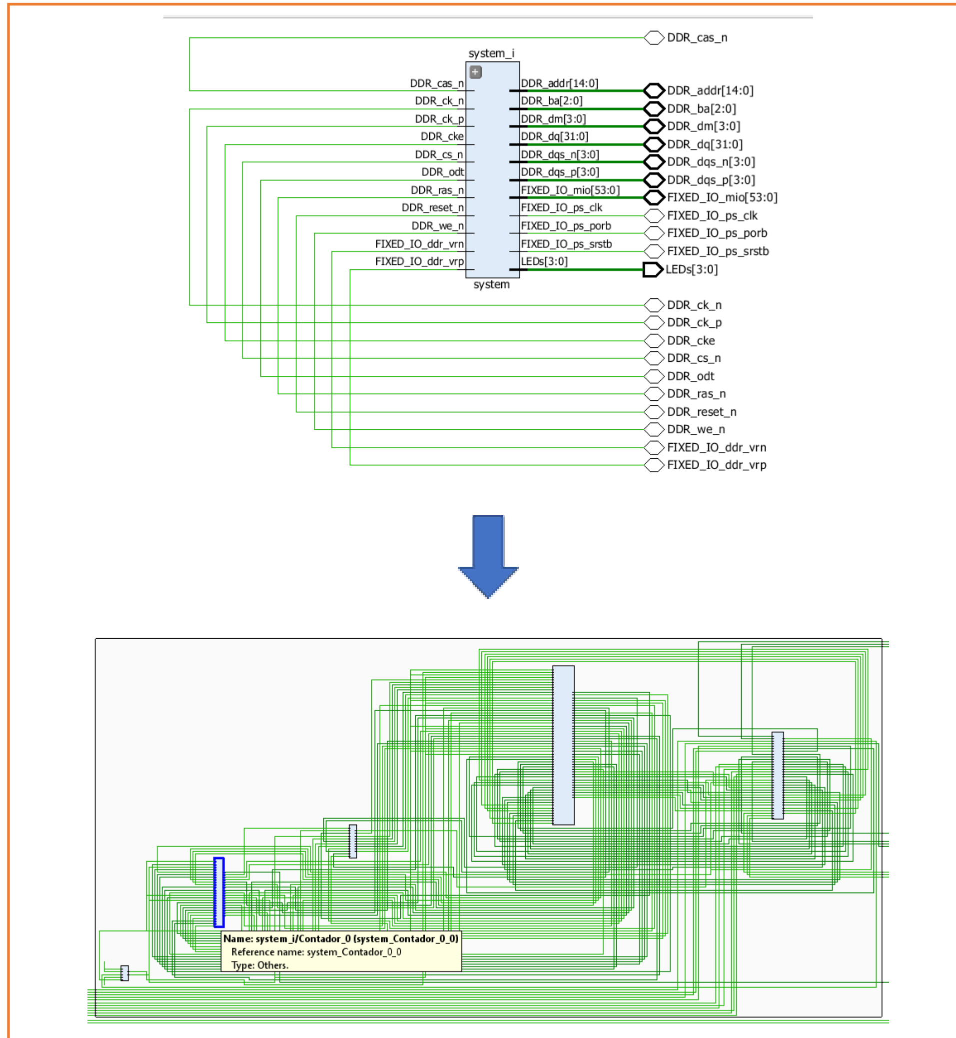


Figura 35 Diagrama RTL: Ejemplo 2.

En la Figura 36, podemos ver el bloque de la IP personalizada resultante al usar el asistente para la creación y empaquetado de IPs “Create and Package New IP….”. Como se observa, la IP cuenta con acceso tanto al Bus S00_AXI, como al Bus S00_AXI_INTR, además cuenta con acceso a cuatro Leds donde se despliega el valor de la cuenta.

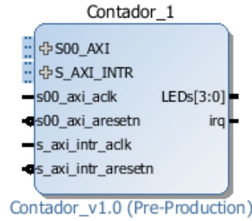


Figura 36 IP Personalizada: Ejemplo2.

Internamente el Bus S00_AXI está amarrado a la lógica del usuario mediante el uso de seis registros "slv_reg"; el Bus S00_AXI_INTR está siendo utilizado para controlar el manejo de interrupciones mediante registros mapeados en memoria, un aspecto importante a resaltar es que el usuario no tiene control directo desde la emulación para el manejo de dichos registros, sin embargo, si se puede controlar la señal que genera la interrupción mediante la lógica del usuario. En el diagrama de la Figura 37 se muestran como están conectados los registros a la lógica del usuario para el Bus S00_AXI, donde se muestra la dirección de inicio del Bus, así como el Offset de cada registro. Por otro lado, se muestran la dirección (Offset) de los registros conectados para el manejo de interrupción, así como la dirección de inicio del Bus S00_AXI_INTR.

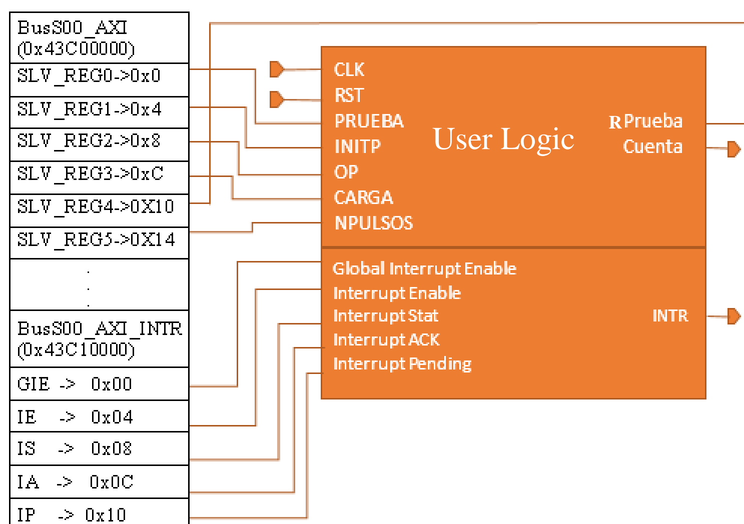


Figura 37 Mapeo interno de registros de la IP: Ejemplo2.

Como se muestra en la Figura 30, la emulación está compuesta por dos estructuras, el "Device Under Test (DUT), y el bloque "Test Controller 1 (TC1)". A continuación, se realiza una descripción de dichas estructuras. Cabe resaltar que estas se encuentran descritas en la lógica del usuario y están amarradas a los buses mediante los archivos plantillas para el manejo de los Buses S00_AXI y S00_AXI_INTR.

- Device Under Test (DUT): El dispositivo bajo prueba consiste en un contador reconfigurable de cuatro bits, con cuatro modos de operación seleccionable mediante la variable OP. Este cuenta con dos puertos de entrada y un puerto de salida
    1. Reinicio (OP=0): En este modo de operación, el dispositivo bajo prueba asigna el valor de nueve a la Cuenta.
    2. Conteo ascendente (OP=1): En este modo de operación el dispositivo bajo prueba toma el valor de la cuenta y lo aumenta por uno en cada flanco de subida del reloj de la tarjeta. Una vez la cuenta llega al valor máximo el siguiente valor que tomará será el valor mínimo.
    3. Conteo descendente (OP=2): En este modo de operación el dispositivo bajo prueba toma el valor de la cuenta y lo disminuye por uno en cada flanco de subida del reloj de la tarjeta. Una vez se alcanza el valor mínimo el siguiente valor que tomará será el valor máximo.
    4. Carga (OP=3): En este modo de operación el dispositivo bajo prueba toma el valor en la entrada Carga y lo asigna al valor de la Cuenta.
- Test Controller 1 (TC1): Cuando la señal Prueba se encuentra en un valor de cero este bloque permanece inactivo, las entradas al dispositivo bajo prueba pueden ser asignadas mediante la función `Opera (Operador, Carga);` por parte de

simulación. Cuando la variable Prueba toma el valor de uno, se activa el bloque y comienza la ejecución de la prueba, esta no puede ser interrumpida hasta que termine, además toma el control de las señales de entrada del dispositivo bajo prueba. Este bloque cuenta con cuatro modos de prueba seleccionables mediante la variable OP. La prueba da inicio cuando se genera un pulso cuadrado de una conmutación de cero a uno, o uno a cero en la variable INITP. El resultado de la prueba es asignado a la variable RPrueba accesible mediante el SLV_REG4. Cuando la prueba ha finalizado se genera una interrupción IP a procesador mediante una variable que genera un pulso de longitud de un periodo de reloj conectada al archivo plantilla para el manejo del Bus S00_AXI_INTR mediante la variable AUXINTR. La variable Prueba e INITP son controladas automáticamente cuando se llama la función `Prueba (Operador, Carga, Npulsos, Resultado);` por parte de simulación.

1. Reinicio (OP=0): Esta prueba toma como entradas la variable OP y la asigna al dispositivo bajo prueba, enseguida lee el valor de la cuenta y lo asigna a la variable RPrueba. El propósito de esta prueba es verificar si el modo de operación OP=0 funciona de manera adecuada, verificando que el valor de la cuenta se reinicia a un valor de nueve.
2. Cuenta ascendente (OP=1): Esta prueba toma el valor de OP, CA y los asigna a las entradas del dispositivo bajo prueba, enseguida cuenta la cantidad de Npulsos, lee el valor de la cuenta y lo asigna a la variable RPrueba. El propósito de esta prueba es verificar si el conteo ascendente se lleva de manera adecuada. Donde CA es el valor de inicio de la cuenta y OP tiene el valor de dos.

3. Cuenta descendente (OP=2): Esta prueba toma el valor de OP, CA y los asigna a las entradas del dispositivo bajo prueba, enseguida cuenta la cantidad de Npulsos, lee el valor de la cuenta y lo asigna a la variable RPrueba. El propósito de esta prueba es verificar si el conteo descendente se lleva de manera adecuada. Donde CA es el valor de inicio de la cuenta y OP tiene el valor de 3.
4. Carga (OP=3): Esta prueba tomo como entradas las variables OP, CA y las asigna al dispositivo bajo prueba, enseguida lee el valor de la cuenta y lo asigna a la variable RPrueba. El propósito de esta prueba es verificar si el modo de operación OP=3 funciona de manera adecuada, verificando que el valor de carga es equivalente al de la cuenta.

En la Figura 38, se muestra el diagrama RTL de la IP personalizada, donde el bloque azul es el encargado del acceso a los registros para el manejo de interrupciones, mientras que el bloque en rosa se encarga del manejo de los registros para la transferencia de datos entre el sistema de procesamiento, PS, y la IP. Mientras que bloque azul representa la lógica del usuario encapsulada mediante un archivo TOP.

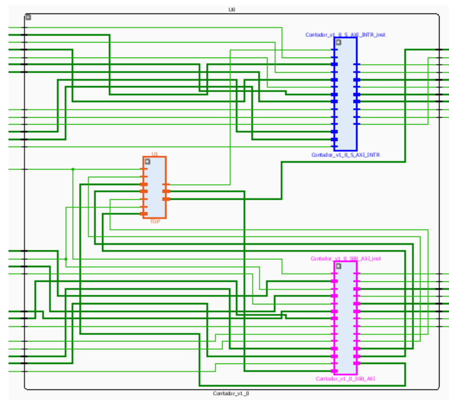


Figura 38 Diagrama RTL IP personalizada: Ejemplo 2.

A continuación, se muestra el diagrama RTL de la lógica del usuario en la IP personalidad, donde el bloque U1 corresponde al DUT, mientras que el bloque U2 corresponde al TC1.

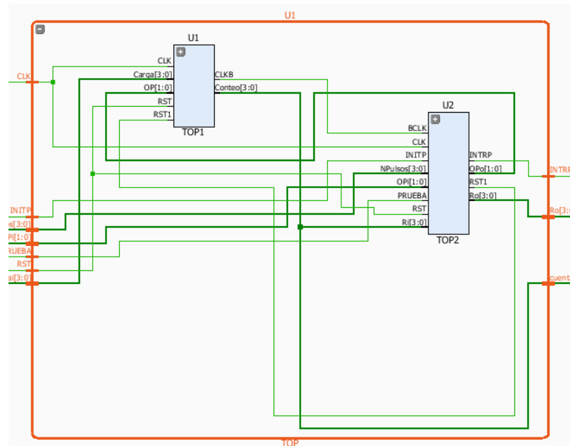


Figura 39 Diagrama RTL de la lógica del usuario: Ejemplo 2.

Capítulo 5 Resultados

5.1 Introducción

El propósito de este trabajo de investigación, fue el desarrollo de un esquema de comunicación que permitiera el intercambio de información entre un entorno de hardware y un entorno de software, para utilizarlo con propósitos de verificación. Con dicho propósito en mente se plantearon cuatro casos de estudio con diferentes características para probar funcionalmente el esquema ante diferentes escenarios, en el primer caso de estudio se probó el esquema de comunicación ante un circuito de respuesta inmediata, el segundo caso de

estudio se probó un circuito de respuesta no inmediata, mientras que en el tercero se utilizó un circuito de respuesta inmediata con tres Testbenches distribuidos en simulación, interfaz y emulación, por último el esquema se probó como un medio para realizar el intercambio de información entre dos particiones de un circuito. En este capítulo se describen las características de los casos de estudio y se exponen las condiciones bajo las cuales el esquema resulta de utilidad, pudiéndolas extrapolar a condiciones similares, o interpolarlas entre sí.

5.2 Caso 1: Circuitos puramente combinaciones

Como primer caso de estudio se tomó como base un circuito puramente combinacional debido a que representa un elemento básico en la construcción de bloques más complejos. Detalles sobre su implementación son tratados en la sección 4.2.

Un aspecto importante a resaltar es que este circuito es de respuesta inmediata, por lo cual, para la comunicación entre simulación y emulación se hizo únicamente a través de registros en el Bus S00_AXI, sin necesitar el Bus S00_AXI_INTR.

La distribución de los elementos de prueba se realizó con base al esquema mostrado en la Figura 19. Donde el dispositivo bajo prueba se encuentra enteramente en la emulación, por su parte los bloques correspondientes al "Testbench" se restringieron exclusivamente a la simulación, y la interfaz actuaba únicamente como puente de comunicación. La prueba consiste de una serie de vectores aplicados a las entradas del circuito, mediante el llamado de funciones desde simulación. Los vectores son generados a partir de una tabla que realiza un recorrido completo de las entradas, empezando por el Selector en 0, y conmutando las entradas A y B de 0 a 1, enseguida se aumenta el selector en 1, y nuevamente se conmuta las entradas A y B de 0 a 1. Esto hasta abarcar todas las posibles combinaciones de entrada. El resultado obtenido por parte de emulación es comparado con un resultado provisto por

una tabla en simulación y se determinaba si es correcto. Un error fue añadido intencionalmente en la tabla de simulación, para ilustrar el caso cuando se produce un error (Ver Figura 40).

```
Se ejecuto programacion
Operacion AND
Vector : 1, Sel : 0, A : 0, B : 0, RS : 0, RE : 0, Tiempo simulacion : @1
Vector : 2, Sel : 0, A : 0, B : 1, RS : 0, RE : 0, Tiempo simulacion : @2
Vector : 3, Sel : 0, A : 1, B : 0, RS : 0, RE : 0, Tiempo simulacion : @3
Vector : 4, Sel : 0, A : 1, B : 1, RS : 1, RE : 1, Tiempo simulacion : @4
Operacion OR
Vector : 5, Sel : 1, A : 0, B : 0, RS : 0, RE : 0, Tiempo simulacion : @5
Vector : 6, Sel : 1, A : 0, B : 1, RS : 1, RE : 1, Tiempo simulacion : @6
Vector : 7, Sel : 1, A : 1, B : 0, RS : 1, RE : 1, Tiempo simulacion : @7
Vector : 8, Sel : 1, A : 1, B : 1, RS : 1, RE : 1, Tiempo simulacion : @8
Operacion XOR
Vector : 9, Sel : 2, A : 0, B : 0, RS : 0, RE : 0, Tiempo simulacion : @9
Vector : 10, Sel : 2, A : 0, B : 1, RS : 1, RE : 1, Tiempo simulacion : @10
Vector : 11, Sel : 2, A : 1, B : 0, RS : 1, RE : 1, Tiempo simulacion : @11
Vector : 12, Sel : 2, A : 1, B : 1, RS : 0, RE : 0, Tiempo simulacion : @12
Operacion NAND
Vector : 13, Sel : 3, A : 0, B : 0, RS : 1, RE : 1, Tiempo simulacion : @13
Vector : 14, Sel : 3, A : 0, B : 1, RS : 1, RE : 1, Tiempo simulacion : @14
Error en Vector: 15, Sel : 3, A : 1, B : 0, Resultado Tablas : 0, Resultado FPGA = 1
$finish called at time : 16 ns : File "/home/fernando/Escritorio/MAESTRIA/PROYECTO1/file.sv" Line 3
```

Figura 40 Resultados de Co-emulación consola TCL: Caso de estudio 1.

5.3 Caso 2: Circuitos secuenciales

Como segundo caso de estudio se tomó como base un circuito mixto compuesto por elementos combinacionales y secuenciales debido a que es otro tipo de circuito básico en la construcción de bloques más complejos. Detalles sobre su implementación son tratados en la sección 4.3.

Como se mencionó anteriormente, este tipo de circuitos son de respuesta no inmediata, pues pueden llegar a tardar varios ciclos de reloj para producir una salida con respecto a las entradas. Por ello para este caso se utilizaron las interrupciones como un medio para regular el intercambio de información por parte de la emulación (la emulación puede avisar a la simulación cuando ha terminado la tarea asignada, y así la simulación puede

acceder a un resultado correcto). Debido a las características del circuito, fue necesario añadir circuitería extra para poder realizar las pruebas al circuito, lo cual constituye una segunda etapa de control, que puede ser gobernada mediante la etapa de control en la simulación. El circuito cuenta con cuatro modos de operación, por ende, la etapa de control descrita en la emulación también cuenta con cuatro tipos de prueba.

La distribución de los elementos de prueba y el dispositivo bajo prueba se realizó conforme al diagrama mostrado en la Figura 30, donde la distribución de los elementos Test Pattern Generator (TPG), Output Response Analyzer (ORA) se realizó mediante estructuras SystemVerilog en el entorno de simulación, por su parte la distribución del "Test Controller" (TC) se dividió entre simulación y emulación, pudiendo así ejecutar pruebas coordinadas entre los dos entornos. Cabe resaltar que la etapa de control ubicada en la emulación toma el control de las señales de entrada en el dispositivo bajo prueba una vez recibida la orden de ejecución por parte de simulación y genera las señales correctas para cada modo de prueba.

El dispositivo bajo prueba se ubicó exclusivamente en la emulación; además es posible ingresar entradas de dos formas, a través de una función en SystemVerilog, o por medio de la etapa de control. Por su parte para la simulación la selección de las entradas es realizada de manera libre, mientras que la etapa de control introduce una secuencia predefinida de acuerdo a la prueba seleccionada.

La prueba consiste de una serie de vectores enviados a la emulación, sin embargo, estos no son aplicados directamente a las entradas del circuito, sino que pasan por una etapa de control implementada en hardware que se encarga de su interpretación y genera los vectores necesarios para probar el circuito. Esto es realizado por medio de una función, que envía tres parámetros a la emulación: OP, Carga, Npulsos. El resultado es regresado a la simulación una vez acaba la ejecución de la función y es comparado con el resultado esperado descrito por medio de una tabla en simulación, donde se determina si el resultado es correcto. Un error fue añadido intencionalmente a la tabla de resultados en simulación, para ilustrar el caso cuando se produce un error (Ver Figura 41).

```
Se ejecuto programacion
Control
Poner circuito en modo de operacion 0 : Reinicio del contador a un valor conocido
Poner circuito en modo de operacion 3 : Cargar un valor de entrada en el contador
Poner circuito en modo de operacion 2 : Cuenta ascendente
Poner circuito en modo de operacion 0 : Cuenta descendente
Inicio de etapa de pruebas
Vector : 1, No.Prueba : 0, Carga : 0, Npulsos : 0, RS : 9, RE : 9, Tiempo simulacion : @7
Vector : 2, No.Prueba : 3, Carga : 7, Npulsos : 0, RS : 7, RE : 7, Tiempo simulacion : @9
Vector : 3, No.Prueba : 1, Carga : 7, Npulsos : 3, RS : 10, RE : 10, Tiempo simulacion : @11
Vector : 4, No.Prueba : 1, Carga : 3, Npulsos : 2, RS : 5, RE : 5, Tiempo simulacion : @13
Error en Vector: 5, No.Prueba : 2, Carga : 3, Npulsos : 2, RS : 2, RE : 1, Tiempo de simulacion : @15
$finish called at time : 15 ns : File "/home/fernando/Escritorio/MAESTRIA/PROYECTO2/file.sv" Line 32
run: Time (s): cpu = 00:00:00.43 ; elapsed = 00:00:07 . Memory (MB): peak = 5860.164 ; gain = 0.000 ; free physical = 232 ; free virtua
```

Figura 41 Resultado Co-emulación consola TCL: Caso de estudio 2.

5.4 Caso 3: Generación de pruebas en diferentes niveles de abstracción

Para el tercer caso de estudio se utilizó como base un sumador de tipo “Carry-Look-Ahead” de 4 bits. Donde la estructura básica del Carry-Look-Ahead como se muestra en la Figura 42 es un sumador de un bit, que recibe como entradas dos parámetros de un bit (Ai, Bi), y un acarreo de entrada (Ci), como resultado genera tres salidas: la suma de los dos bits y el acarreo de entrada (Si), el acarreo propagado (Pi), y el acarreo generado (Gi). Donde Pi

se obtiene a partir de realizar una operación and entre la variable Ai y Bi, donde se busca detectar el acarreo que se propaga cuando tanto Ai como Bi, se encuentran en un estado lógico de 1, en este también se considera indirectamente el caso cuando Ai, Bi, y Ci se encuentran en 1 lógico. Por otro lado, Gi busca encontrar el acarreo generado cuando Ci se encuentra en un estado lógico de 1, y alguna de las dos entradas se encuentra en estado lógico 1, esto se logra realizando una operación xor entre las entradas Ai y Bi, lo cual detecta si alguna se encuentra en estado de 1 lógico, para después compararla con el acarreo de entrada mediante una operación and. Para determinar el acarreo de salida solo basta con realizar una operación or entre Pi y Gi.

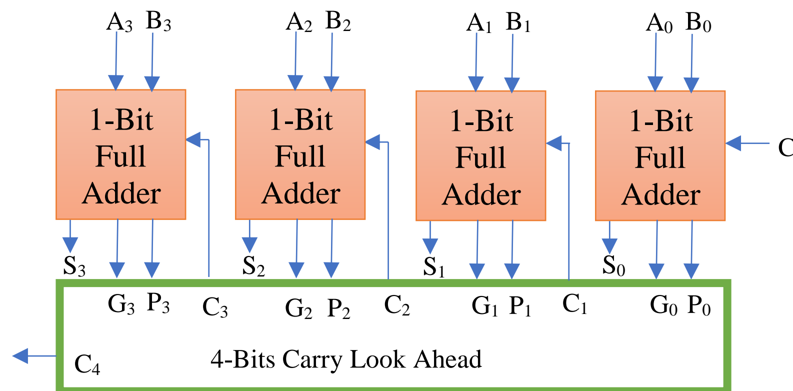


Figura 42 4-Bits Carry Look Ahead.

Para probar el sumador Carry Look Ahead, se crearon tres estructuras de prueba Testbench, localizadas en simulación, Interfaz y Emulación (Ver Figura 43). Cada una de las estructuras cuenta con Test Pattern Generator (TPG), Output Response Analyzer (ORA) y Test Controller (TC).

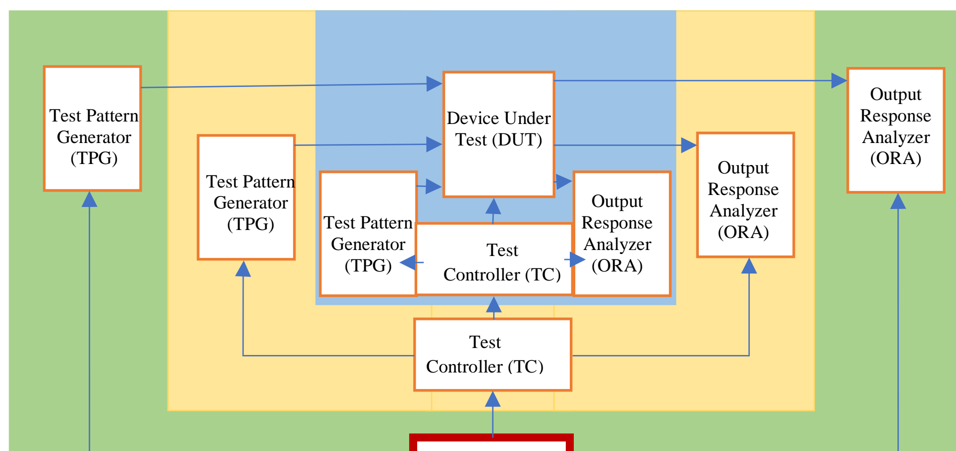


Figura 43 Estructura de pruebas: Caso de estudio 3.

Un aspecto importante a resaltar es que para la generación de los patrones de prueba en la emulación se utilizó el circuito propuesto en [21]. El circuito consiste de un registro de longitud N+2, donde N representa la longitud de las entradas de la suma. La generación de A y B, se realiza a partir de un bloque combinacional que toma como entrada valores en el registro de desplazamiento (ver Figura 44).

Los valores generados a partir del circuito consisten de una serie de doce vectores que provee una alta cobertura para sumadores del tipo Carry-Look-Ahead.

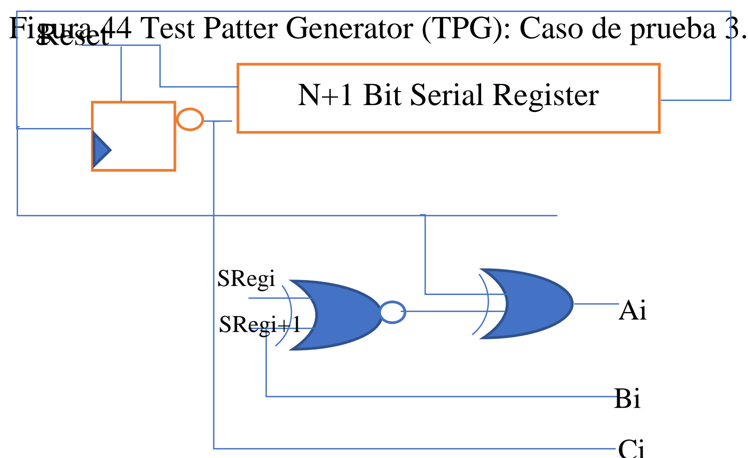


Figura 44 Test Patter Generator (TPG): Caso de prueba 3.

Tabla 2 Patrones de prueba emulación: Caso de prueba 3.

A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0	C_0
1	1	1	1	0	0	0	0	1
1	1	1	0	0	0	0	0	1
1	1	0	1	0	0	0	1	1
1	0	1	1	0	0	1	1	1
0	1	1	1	0	1	1	1	1
0	0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	1	0
0	0	0	1	1	1	1	1	0
0	0	1	0	1	1	1	0	0
0	1	0	0	1	1	0	0	0
1	0	0	0	1	0	0	0	0
1	1	1	1	0	0	0	0	0

Para probar el circuito se crearon tres estructuras de pruebas distribuidas en el esquema de comunicación como se muestra en la Figura 43. La prueba en simulación consiste de una seria de vectores generados a partir de funciones de números aleatorios que son enviados aplicados a las entradas del circuito por medio de una función desde simulación, el resultado obtenido por parte de la emulación es comparado con una suma realizada localmente en simulación. Por su parte, la estructura de prueba en la interfaz es activada por medio del llamado por parte de simulación, y consiste de una serie de vectores generados a partir de funciones de números aleatorios que son enviados mediante la escritura de registros mapeados en memoria, el resultado obtenido por parte de simulación es comparado con una suma realizada localmente en la interfaz, al finalizar la prueba se regresa a la simulación si

el circuito paso la prueba o no. Por último, la estructura de pruebas en la emulación es activada por medio del llamado de una función en simulación, la generación de vectores de prueba es generada a partir del circuito propuesto [21], los resultados de salida del circuito son comparados con valores en una memoria ROM, donde se determina si son correctos o no, al finalizar la prueba se envían los resultados a emulación, en caso de fallo se envía los vectores de prueba, el valor esperado en la memoria ROM, y el valor de salida del DUT (ver Figura 45).

```
Se ejecuto programacion
1) Valores enviados directamente a las entradas del circuito
@3 A = 4, B = 15, Ci = 1, S = 4, Co = 1
2) Prueba realizada por medio de la estructura de pruebas en emulacion
La prueba fallo en ADDRSFALL : 7, SGA : 1, SGB : 15, SGC : 0 SCLAS : 0, SCLAC : 1, SROMS : 1, SROMC : 1, Tiempo de simulacion : 5
3) Prueba realizada mediante la estructura ubicada en la interfaz
La prueba con la estructura de pruebas en interfaz se realizo sin errores
4) Prueba realizada por medio de la estructura ubicada en simulacion
La prueba con la estructura de pruebas en simulacion se realizo sin errores
$finish called at time : 14 ns : File "/home/fernando/Escritorio/MAESTRIA/PROYECTO4/file.sv" Line 76
```

Figura 45 Resultados Co-emulación consola TCL: Caso de estudio 3.

Los códigos correspondientes a este caso de estudio son mostrados en A.3 Caso de estudio tres.

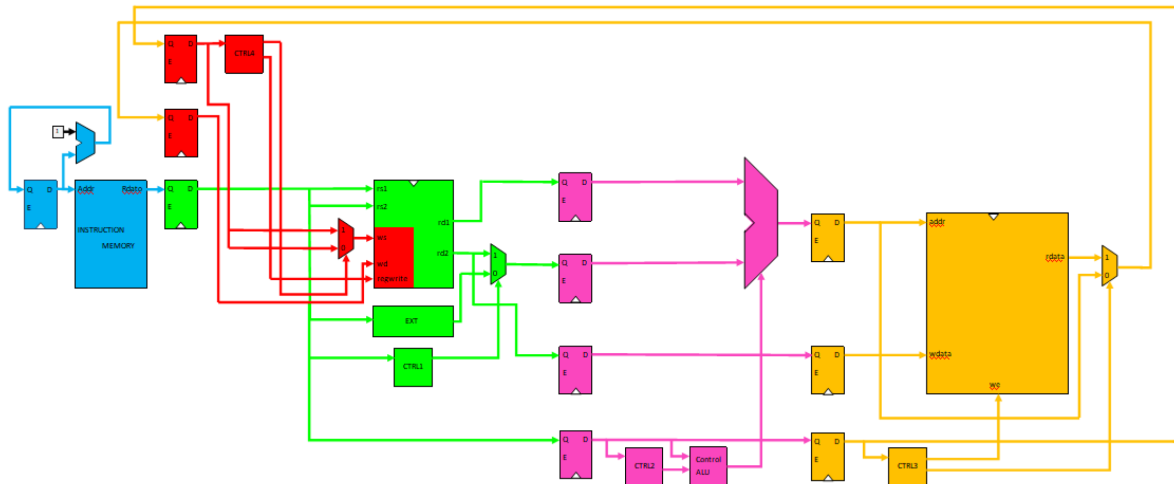
5.5 Caso 4: Dispositivo bajo prueba particionado

Como cuarto caso de estudio se utilizó un procesador RISC de 32 bits con arquitectura tipo Harvard pipelined de cinco etapas: 1) Instruction Fetch, 2) Decode and register fetch, 3) Alu operation, 4) Data fetch (opcional), y 5) Register write back (opcional). En la Figura 46 se muestra el diagrama a bloques utilizado para la implementación, además se distinguen las cinco etapas pipeline a través de colores, donde los bloques azules corresponden a la primera

etapa, los bloques verdes a la segunda, los bloques morados a la tercera, los bloque amarillos a la cuarta, y los bloques rojos a la quinta.

Figura 46 Diagrama a bloques procesador pipeline: Caso de estudio 4.

El procesador cuenta con un set de instrucciones compuesto por cinco instrucciones



tipo R y cuatro tipo I. Las instrucciones tipo R están compuestas por cinco campos, el OP que señala el código de la instrucción, RS la dirección en la memoria de registros del primer operando, RT la dirección en la memoria de registros del segundo operando, RD la dirección de escritura del resultado de la operación, "Shent" un desplazamiento, y F la función a ejecutar. Por su parte la instrucción tipo I está constituida por 4 campos, donde OP señala el código de la instrucción, RS la dirección en memoria de registros del segundo operando, RT una dirección en la memoria de datos donde se almacenará el resultado de la operación, e Immediate un valor entero del segundo operando.

- Tipo R

Tabla 3 Instrucciones Tipo R: caso de estudio 4.

OP (31-26)	RS (25-21)	RT (20-16)	RD (15-11)	Shent (10-6)	F (5-0)
ADD $d=$s+$t;					
000000	SSSSS	TTTTT	DDDDD	00000	100000
SUB $d=$s-$t;					
00000	SSSSS	TTTTT	DDDDD	00000	100010
AND $d=$s and $t;					
00000	SSSSS	TTTTT	DDDDD	00000	100100
OR $d=$s or $t;					
00000	SSSSS	TTTTT	DDDDD	00000	100101
SLT if $d<$t =>$d=1 else $d=0;					
00000	SSSSS	TTTTT	DDDDD	00000	101010

- Tipo I

Tabla 4 Instrucciones tipo I: caso de estudio 4.

OP (31-26)	RS (25-21)	RT (20-16)	Direction Immediate
ADDI $t=$s+Immediate;			
001000	SSSSS	TTTTT	Immediate
LW $t=MEM[$s+Immediate];			
100011	RRRRR	TTTTT	Immediate
SW MEM[$s+Immediate]=$t			
101011	RRRRR	TTTTT	Immediate
SLTI if $s<Immediate => $t=1 else $t=0;			
001010	RRRRR	TTTTT	Immediate

La distribución de la estructura de pruebas, y el dispositivo bajo prueba se realizó conforme al diagrama mostrado en la Figura 47. Como se puede ver el dispositivo bajo prueba fue dividido en dos partes, y fue distribuido en dos tarjetas PYNQ Z1, por su parte la interfaz únicamente actúa como medio de comunicación, la simulación se encarga de controlar el intercambio de información entre las dos mitades del procesador.

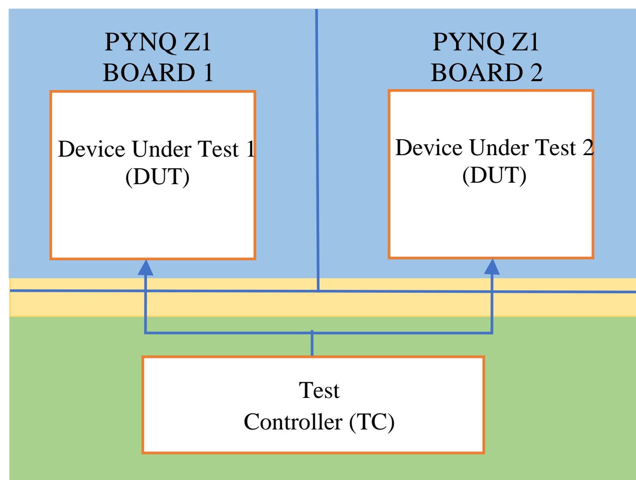


Figura 47 Estructura de pruebas: caso de estudio 4.

La primera tarjeta PYNQ Z1 contine la primera parte del procesador que esta constituida por las etapas pipelined: 1) Instruction Fetch, 2) Decode and Register Fetch y 5) Register Write Back (de acuerdo a la instrucción ejecutada). Como se puede ver en la Figura 48, esta fue conectada al procesador por meido de una interfaz mediante tresregistros de entrada y dos registros de salida.

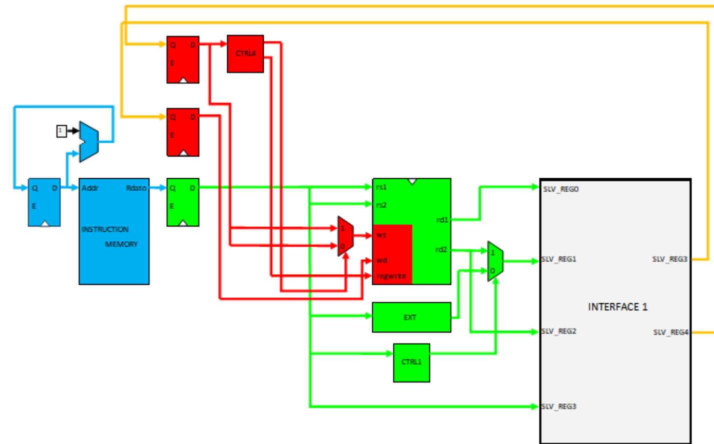


Figura 48 Partición 1: Caso de estudio 4.

Por otro lado, la segunda tarjeta PYNQ Z1 contiene la segunda parte del procesador constituido por las etapas pipelined: 3) Alu operation, 4) Data fetch (de acuerdo a la instrucción en ejecución). Como se muestra en el diagrama de la Figura 49, esta fue conectada al procesador por medio de una interfaz con dos registros de entrada y tres registros de salida.

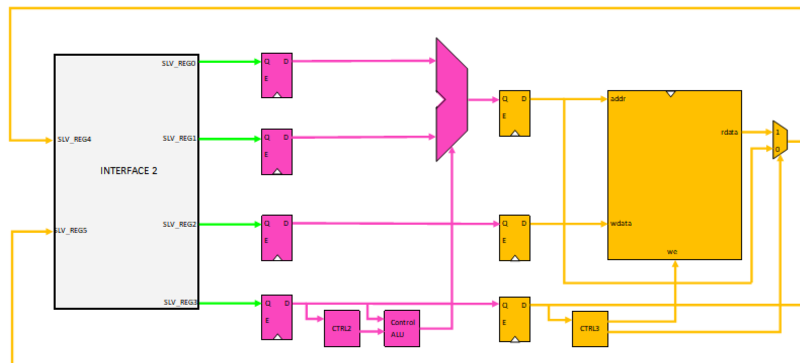


Figura 49 Partición 2: Caso de estudio 4.

Como se muestra en la Figura 47, no existe como tal una estructura de pruebas con generador de vectores de prueba, ni un analizador de respuesta de salida. Por ello, para la ejecución de la prueba se aprovecharon estructuras internas en el procesador. A través de la memoria del programa se realizaron operaciones para modificar el valor en los registros de la memoria de registros a valores controlados, con la finalidad de corroborar la correcta ejecución de las

instrucciones. El acceso al valor de los registros de memoria de registros, así como la ejecución de las instrucciones puede ser controlada desde la simulación a través de funciones. La prueba consiste de los siguientes pasos: a) Lectura e impresión en pantalla del valor inicial de los registros R0 a R31, b) Inicialización de los registros R1 a R31con valores de 1 a 5 en una secuencia repetida hasta alcanzar el registro 31, c) Suma de R1, R2 y almacenar en R2, suma de R3, R4 y almacenar en R4, hasta alcanzar el registro R31 que es sumado consigo mismo, d) Resta de R2 menos R1 y almacenar valor en R2, resta de R4 menos R3 y almacenar en R4, esto hasta llegar al registro R31, el cual es restado consigo mismo. El registro R0 siempre se mantiene en 0 (Ver Figura 50).

```
Se ejecuto programacion
a) Lectura e impresion del valor inicial de los registros
 R0: 0, R1: 0, R2: 0, R3: 0, R4: 0, R5: 0, R6: 0, R7: 0, R8: 0, R9: 0, R10: 0, R11: 0, R12: 0,
b) Inicializacion de los registros con una secuencia predeterminada
 R0: 0, R1: 1, R2: 2, R3: 3, R4: 4, R5: 5, R6: 1, R7: 2, R8: 3, R9: 4, R10: 5, R11: 1, R12: 2,
c) Suma entre registros
 R0: 0, R1: 1, R2: 3, R3: 3, R4: 7, R5: 5, R6: 6, R7: 2, R8: 5, R9: 4, R10: 9, R11: 1, R12: 3,
d) Resta entre registros
 R0: 0, R1: 1, R2: 2, R3: 3, R4: 4, R5: 5, R6: 1, R7: 2, R8: 3, R9: 4, R10: 5, R11: 1, R12: 2,
$finish called at time : 142 ns : File "/home/fernando/Escritorio/MAESTRIA/PROYECTO3/file.sv" L
run: Time (s): cpu = 00:00:00.69 ; elapsed = 00:00:08 . Memory (MB): peak = 5813.664 ; gain = (
```

Figura 50 Resultado Co-emulación consola TCL: Caso de estudio 4.|

Los códigos correspondientes a este caso de estudio son mostrados en A.4 Caso de estudio cuatro.

Capítulo 6 Conclusiones y trabajo a futuro

6.1 Conclusiones

En esta sección se presentan conclusiones obtenidas durante el desarrollo de este trabajo de investigación, así como de los resultados obtenidos:

- Se diseñó e implemento un esquema de comunicación que permite el intercambio de información entre un entorno de FPGA y PC, donde el intercambio de información es llevado bajo directivas Maestro-Esclavo.
- Las herramientas utilizadas tales como Vivado Web Pack, SystemVerilog, C, Python son de acceso libre y la tarjeta PYNQ Z1 es de bajo costo, por lo que, comparado con entornos comerciales desarrollados por empresas con propósitos de verificación emulada y capacidades de mayor escala, el esquema de comunicación propuesto resulta más accesible económicamente hablando.
- El esquema de comunicación permite la descripción de hardware en un entorno de software (simulación) y un entorno de hardware (emulación), a través de SystemVerilog , DPI-C y VHDL/Verilog respectivamente, además permite la descripción de elementos de software mediante Python o C.
- Comparado con otros esquemas donde se utilizan lenguajes de descripción de software, el esquema propuesto utiliza SystemVerilog, un lenguaje de descripción de hardware, por lo cual permite la reutilización de descripciones de hardware HDL, además puede simular procesos secuenciales y concurrentes, por otro lado, al ser también un lenguaje de verificación de hardware cuenta con estructuras para dicho propósito.
- Se comprobó la operacionalidad del esquema diseñado mediante cuatro casos de estudio que presentan diferentes características, entre los que se incluye la emulación particionada de dos tarjetas.

6.2 Trabajo Futuro

Como se ha establecido a lo largo de este trabajo de investigación, este esquema de comunicación entre hardware y software, pretende ser una herramienta utilizable para propósitos de verificación. Además, se proveyeron los elementos necesarios para dicho propósito, sin embargo, existen distintos aspectos del esquema que aún pueden ser mejorados. Con la finalidad de dar seguimiento al esquema de comunicación propuesto, se proponen los siguientes trabajos futuros.

- Actualmente, cada uno de los archivos que componen el esquema de comunicación, son creados manualmente mediante Vivado o Blocs de notas, además la ejecución de la co-emulación es llevada mediante comandos ingresados manualmente en la consola Vivado TCL, así como algunos otros pasos. Por ello se proponen dos líneas de trabajo. En la primera se busca realizar la generación automática de los archivos correspondientes a la interfaz, mediante el uso de un programa Python y un archivo de configuración que los genere de forma automática, dado que las funciones para el intercambio de información básicas presentan estructuras regulares. Como la segunda línea de trabajo se propone la creación de librerías Python mediante las cuales se pueda controlar el envió del archivo servidor, archivo TCL y archivo Bitstream a la tarjeta, la generación de archivos de la interfaz y el control de la ejecución de la co-emulación, entre otros. Un aspecto importante a resaltar es que la automatización completa del proceso nunca será realizable, pues el factor humano juega un papel indispensable en el diseño de hardware, repercutiendo directamente en la estructura de pruebas y el dispositivo bajo prueba.

- Actualmente la lógica programable puede interactuar con C, Python y SystemVerilog, sin embargo, existen diferentes plataformas que ofrecen diversas funciones tales como el modelado de sensores. Por ello se propone extender la comunicación del esquema a otras plataformas con la finalidad de extender sus funciones.

Con los trabajos propuestos se busca que el esquema de comunicación pueda ser controlado de manera semiautomática a través de un lenguaje de alto nivel, así como la integración de múltiples plataformas de acceso libre que añadan funciones al esquema propuesto. Todo esto con la finalidad de crear un esquema multiplataforma, multitarea de acceso libre y sencillo de usar.

Apéndice A. Código Casos de Estudio.

A.1 Caso de estudio uno

A.1.1 File.sv

```
import "DPI-C" function void Programar();
import "DPI-C" function void Opera(input bit [2:0]
Operando, input bit Arg1, input bit arg1, output bit
result);

module always_comb_process();
reg [2:0] sel;
reg [1:0] sel1;
reg resultado;
reg sal;
int prueba;

initial begin
prueba=0;
Programar();

for (int i = 0 ; i <= 4; i ++) begin
      sel=i;
      for (int i = 0 ; i <= 3; i ++) begin
            sel1=i;
            prueba = prueba+1;
            Opera(sel, sel1[1], sel1[0], resultado);

#1 if (prueba == 1) begin
      $display("Operacion AND");
end else if (prueba == 5) begin
      $display("Operacion OR");
end else if (prueba == 9) begin
       $display("Operacion XOR");
end else if (prueba == 13) begin
      $display("Operacion NAND");
end else if (prueba == 17) begin
      $display("Operacion NOR");
end

if (sal != resultado) begin
      $display ("Error en Vector: %g, Sel : %g, A : %g,
B : %g, Resultado Tablas : %g, Resultado FPGA = %g",
prueba, sel, sel1[1], sel1[0], sal, resultado);
#1  $finish;
end

$display("Vector : %g, Sel : %g, A : %g, B : %g, RS :
%g, RE : %g, Tiempo simulacion : @%g", prueba, sel,
sel1[1], sel1[0], sal, resultado, $time);
      end
end

$display("comprobacion hecha sin errores");
#1  $finish;
end

always_comb
begin:Tabla
case(sel)
 3'b000:begin
       case({sel1[1],sel1[0]})
            2'b00:sal=0;
            2'b01:sal=0;
            2'b10:sal=0;
            default:sal=1;
       endcase
      end
 3'b001:begin
       case({sel1[1],sel1[0]})
            2'b00:sal=0;
```

```
            2'b01:sal=1;
            2'b10:sal=1;
            default:sal=1;
       endcase
      end
 3'b010:begin
       case({sel1[1],sel1[0]})
            2'b00:sal=0;
            2'b01:sal=1;
            2'b10:sal=1;
            default:sal=0;
       endcase
      end
 3'b011:begin
       case({sel1[1],sel1[0]})
            2'b00:sal=1;
            2'b01:sal=1;
            2'b10:sal=0;
            default:sal=0;
       endcase
      end
 3'b100:begin
       case({sel1[1],sel1[0]})
            2'b00:sal=1;
            2'b01:sal=0;
            2'b10:sal=0;
            default:sal=0;
       endcase
      end
 default: $display("Valor de sel erróneo");
endcase
end
endmodule
```

A.1.2 File.c

```
#include
"/opt/Xilinx/Vivado/2016.1/data/xsim/include/svdpi.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include </usr/include/python2.7/Python.h>

DPI_DLLESPEC
void Programar() {
	printf("Se ejecuto programacion\n");
	PyObject *pName, *pModule, *pDict, *pFunc,
*pValue;
	Py_Initialize();
	PyRun_SimpleString("import xmlrpclib\n");
	PyRun_SimpleString("s =
xmlrpclib.ServerProxy('URL')\n");
	PySys_SetPath("project_path");
	pName = PyString_FromString("file0");
	pModule = PyImport_Import(pName);
	pDict   = PyModule_GetDict(pModule);
	pFunc = PyObject_GetAttrString(pModule, "PROG");
	  PyObject_CallObject(pFunc, NULL);
	  PyErr_Print();
	Py_DECREF(pModule);
	  Py_DECREF(pFunc);
	Py_Finalize();
}

DPI_DLLESPEC
void Opera(  svBitVecVal *O,
             svBit A1,
             svBit A2,
             svBit *R) {PyObject *pName, *pModule,
*pDict, *pFunc, *pValue, *pArgs;
	Py_Initialize();
	PyRun_SimpleString("import xmlrpclib\n");
	PyRun_SimpleString("s =
xmlrpclib.ServerProxy('URL')\n");
	PySys_SetPath("project_path");
	pName = PyString_FromString("file1");
	pModule = PyImport_Import(pName);
	pDict   = PyModule_GetDict(pModule);
	pFunc = PyObject_GetAttrString(pModule,
"OPERA1");

	int OP;

	unsigned char argu1,argu2, resultado;
	OP=*O;
	argu1=A1;
	argu2=A2;

	pArgs = Py_BuildValue("iii", OP,argu1,argu2);
	  PyObject* item=PyTuple_GetItem(pArgs,0);
	  PyErr_Print();

	PyObject* result = PyObject_CallObject(pFunc,
pArgs);
	resultado= (int)PyInt_AsSsize_t(result);
	  PyErr_Print();
	Py_DECREF(pArgs);
	  Py_DECREF(item);
	  Py_DECREF(result);


	Py_DECREF(pModule);
	  Py_DECREF(pFunc);
	Py_Finalize();

	*R=resultado;
}
```

A.1.3 File0.py

```
import xmlrpclib
def PROG():
    s = xmlrpclib.ServerProxy('URL')
    s.P();
```

A.1.4 File1.py

```
import xmlrpclib
def OPERA1(Operador,arg1,arg2):
    s = xmlrpclib.ServerProxy('URL')
    z=s.O(Operador,arg1,arg2);
    return z;
```

A.1.5 Servidor.py

```
import time
import asyncio
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler
from pynq import MMIO
from pynq import PL
from pynq import Interrupt
from pynq import Overlay

#Programar de inicio
base=Overlay('PL1.bit')
base.download()
#Variable Globales
#Usar bus S00_AXI
Combinacional = MMIO(0x43c00000, 512)
#Interrupciones
#Controlador
C_Interrupt= MMIO(0x41800000, 512)
#Usar bus S_AXI_INTR
Combinacional_I = MMIO(0x43c10000,512)
#Objeto de interrupcion
interrupt = Interrupt('combinacional_0/irq')

# Restrict to a particular path.
class
RequestHandler(SimpleXMLRPCRequestHandler):rpc_paths =
('/RPC2',)

server = SimpleXMLRPCServer(("192.168.2.98",
8000),requestHandler=RequestHandler)
server.register_introspection_functions()

def Programar():
    base=Overlay('PL1.bit')
    base.download()
    return 1

#Register function
server.register_function(Programar, 'P')

#Operacion
def Opera(OP,A,B):
    #Operacion
    global Combinacional
    Combinacional.write(0x0,OP)
    Combinacional.write(0x4,A)
    Combinacional.write(0x8,B)
    SAL=Combinacional.read(0xc)
    return SAL
#Register function
server.register_function(Opera, 'O')

# Run the server's main loop
server.serve_forever()
```

A.2 Caso de estudio dos

A.2.1 File.sv

```
import "DPI-C" function void Programar();
import "DPI-C" function void Opera(input bit [1:0]
Operador, input bit [3:0] Carga);
import "DPI-C" function void Prueba(input bit [1:0]
Operador, input bit [3:0] Carga, input bit [3:0]
Npulsos, output bit [3:0] Resultado);

module always_comb_process();
reg [1:0] Operador;
reg [3:0] Carga;
reg [3:0] Npulsos;
reg [3:0] ResultadoE;
reg [3:0] ResultadoS;
reg [2:0] conta;
int aux;

initial begin
#1 Programar();
$display("Control");
$display("Poner circuito en modo de operacion 0 :
Reinicio del contador a un valor conocido");
#1 Opera(0, 0);
$display("Poner circuito en modo de operacion 3 :
Cargar un valor de entrada en el contador");
#1 Opera(3, 5);
$display("Poner circuito en modo de operacion 2 :
Cuenta ascendente");
#1 Opera(1, 0);
$display("Poner circuito en modo de operacion 0 :
Cuenta descendente");
#1 Opera(2, 0);
$display("Inicio de etapa de pruebas");
 for (int i = 0 ; i <= 5; i ++) begin
      conta=i;
      aux=i+1;
      #1 Prueba(Operador, Carga, Npulsos, ResultadoE);
      #1 if (ResultadoS != ResultadoE) begin
            $display("Error en Vector: %g, No.Prueba :
%g, Carga : %g, Npulsos : %g, RS : %g, RE : %g, Tiempo
de simulacion : @%g", aux, Operador, Carga, Npulsos,
ResultadoS, ResultadoE,$time);
            $finish;
      end
      $display("Vector : %g, No.Prueba : %g, Carga :
%g, Npulsos : %g, RS : %g, RE : %g, Tiempo simulacion :
@%g", aux, Operador, Carga, Npulsos, ResultadoS,
ResultadoE,$time);

 end

$display("comprobacion hecha sin errores");
#1  $finish;
end

always_comb
begin:Tabla
case(conta)
 3'b000:begin
            Operador=0;
            Carga=0;
            Npulsos=0;
            ResultadoS=9;

      end
 3'b001:begin
            Operador=3;
            Carga=7;
            Npulsos=0;
            ResultadoS=7;
```

```
      end
 3'b010:begin
            Operador=1;
            Carga=7;
            Npulsos=3;
            ResultadoS=10;
      end
 3'b011:begin
            Operador=1;
            Carga=3;
            Npulsos=2;
            ResultadoS=5;
      end
 3'b100:begin
            Operador=2;
            Carga=3;
            Npulsos=2;
            ResultadoS=2;
      end
 3'b101:begin
            Operador=2;
            Carga=15;
            Npulsos=8;
            ResultadoS=7;
      end
 default: $display("Valor de sel erróneo");
endcase
end
endmodule
```

A.2.2 File.c

```
#include
"/opt/Xilinx/Vivado/2016.1/data/xsim/include/svdpi.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include </usr/include/python2.7/Python.h>

DPI_DLLESPEC
void Programar() {
      printf("Se ejecuto programacion\n");
      PyObject *pName, *pModule, *pDict, *pFunc,
*pValue;;
      Py_Initialize();
      PyRun_SimpleString("import xmlrpclib\n");
      PyRun_SimpleString("s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')\n");
      PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO2");
      pName = PyString_FromString("file0");
      pModule = PyImport_Import(pName);
      pDict   = PyModule_GetDict(pModule);
      pFunc = PyObject_GetAttrString(pModule, "PROG");
        PyObject_CallObject(pFunc, NULL);
        PyErr_Print();
      Py_DECREF(pModule);
        Py_DECREF(pFunc);
      Py_Finalize();
}

DPI_DLLESPEC
void Opera(  svBitVecVal *Operador,
             svBitVecVal *Carga
) {
      PyObject *pName, *pModule, *pDict, *pFunc,
*pValue, *pArgs;

      Py_Initialize();
      PyRun_SimpleString("import xmlrpclib\n");
      PyRun_SimpleString("s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')\n");
      PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO2");
```

```
	pName = PyString_FromString("file1");
	pModule = PyImport_Import(pName);
	pDict   = PyModule_GetDict(pModule);
	pFunc = PyObject_GetAttrString(pModule,
"OPERA1");

	int OP, Car;
	OP=*Operador;
	Car=*Carga;

	pArgs = Py_BuildValue("ii", OP, Car);
	  PyObject* item=PyTuple_GetItem(pArgs,0);
	  PyErr_Print();

	PyObject* result = PyObject_CallObject(pFunc,
pArgs);
	  PyObject_CallObject(pFunc, pArgs);
	  PyErr_Print();

	Py_DECREF(pArgs);
	  Py_DECREF(item);
	  Py_DECREF(result);


	Py_DECREF(pModule);
	  Py_DECREF(pFunc);
	Py_Finalize();

}



DPI_DLLESPEC
void Prueba( svBitVecVal *Operador,
             svBitVecVal *Carga,
             svBitVecVal *Npulsos,
             svBitVecVal *Resultado
) {
	PyObject *pName, *pModule, *pDict, *pFunc,
*pValue, *pArgs;

	Py_Initialize();
	PyRun_SimpleString("import xmlrpclib\n");
	PyRun_SimpleString("s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')\n");
	PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO2");
	pName = PyString_FromString("file2");
	pModule = PyImport_Import(pName);
	pDict   = PyModule_GetDict(pModule);
	pFunc = PyObject_GetAttrString(pModule,
"MPRUEBA");

	int OP, Car, Pulsos, Resul;
	OP=*Operador;
	Car=*Carga;
	Pulsos=*Npulsos;

	pArgs = Py_BuildValue("iii", OP, Car, Pulsos);
	  PyObject* item=PyTuple_GetItem(pArgs,0);
	  PyErr_Print();

	PyObject* result = PyObject_CallObject(pFunc,
pArgs);
	  PyObject_CallObject(pFunc, pArgs);
	Resul= (int)PyInt_AsSsize_t(result);
	  PyErr_Print();

	Py_DECREF(pArgs);
	  Py_DECREF(item);
	  Py_DECREF(result);


	Py_DECREF(pModule);
```

```
        Py_DECREF(pFunc);
      Py_Finalize();
      *Resultado=Resul;
}
```

A.2.3 File0.py

```
import xmlrpclib
def PROG():
    s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')
    s.P();
```

A.2.4 File1.py

```
import xmlrpclib
def OPERA1(Operador, carga):
    s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')
    z=s.O(Operador, carga);
```

A.2.5 File2.py

```
import xmlrpclib
def MPRUEBA(Operador, carga, Npulsos):
    s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')
    z=s.P1(Operador, carga, Npulsos)
    return z
```

A.2.6 Servidor.py

```
import time
import asyncio
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler
from pynq import MMIO
from pynq import PL
from pynq import Interrupt
from pynq import Overlay

#Programar de inicio
base=Overlay('PL2.bit')
base.download()
#Variable Globales
#Usar bus S00_AXI
Contador     = MMIO(0x43c00000, 512)
#Contador
#Usar bus S00_AXI_INTR
Contador_INTR= MMIO(0x43c10000, 512)
#Objeto de interrupcion
interrupt = Interrupt('Contador_0/irq')

#Controlador de interrupciones
Controlador_INTRP=MMIO(0x41800000, 512)

#Habilitar interrupciones
Contador_INTR.write(0x0, 0x00000001)
Contador_INTR.write(0x4, 0x00000001)

# Restrict to a particular path.
class
RequestHandler(SimpleXMLRPCRequestHandler):rpc_paths =
('/RPC2',)

server = SimpleXMLRPCServer(("192.168.2.98",
8000),requestHandler=RequestHandler)
server.register_introspection_functions()

def Programar():
    global Contador
    global Contador_INTRP
    global Controlador_INTRP
```

```
    global interrupt
    base=Overlay('PL2.bit')
    base.download()
    #Variable Globales
    #Usar bus S00_AXI
    Contador     = MMIO(0x43c00000, 512)
    #Contador
    #Usar bus S00_AXI_INTR
    Contador_INTR= MMIO(0x43c10000, 512)
    #Objeto de interrupcion
    interrupt = Interrupt('Contador_0/irq')

    #Controlador de interrupciones
    Controlador_INTRP=MMIO(0x41800000, 512)

    #Habilitar interrupciones
    Contador_INTR.write(0x0, 0x00000001)
    Contador_INTR.write(0x4, 0x00000001)
    return 1

#Register function
server.register_function(Programar, 'P')

#Operacion
def Opera(OP, Carga):
    global Contador
    Contador.write(0x0, 0x00000000)#Prueba------slv0
    Contador.write(0x4, 0x00000000)#INITP------slv1
    Contador.write(0x8, OP)#OP------------------slv2
    Contador.write(0xC, Carga)#Carga---------------slv3
    Contador.write(0x14, 0x00000000)#Npulsos-----------
----slv5
    return 1
#Register function
server.register_function(Opera, 'O')
```

```
@asyncio.coroutine
def wait_for_value_async(OP, Carga, Npulsos):
        global Contador
        global Contador_INTRP
        global Controlador_INTRP
        global interrupt
        print("E11")
        #Habilitar interrupciones
        Contador_INTR.write(0x0, 0x00000001)
        Contador_INTR.write(0x4, 0x00000001)
        #Entrando a modo prueba
        Contador.write(0x0, 0x00000001)#Prueba------
slv0
        #Configurando prueba
        Contador.write(0x8, OP)#OP------------------
slv2
        Contador.write(0xC, Carga)#Carga---------------
slv3
        Contador.write(0x14, Npulsos)#Npulsos----------
-----slv5
        #Iniciando prueba
        Contador.write(0x4, 0x00000001)#INITP------slv1
        x=1
        while(Controlador_INTRP.read(0x0)==0x01 or
x==1):
            x=2
            print("E2")
            print(Controlador_INTRP.read(0x0))
            yield from interrupt.wait()
            Contador_INTR.write(0xc, 0x00000001)
        print("E3")
        Resultado=Contador.read(0x10);
        #Saliendo del modo prueba y poniendo INITP a 0
        Contador.write(0x0, 0x00000000)#Prueba------
slv0
        Contador.write(0x4, 0x00000000)#INITP------slv1
        return Resultado
```

```
def Prueba(OP, Carga, Npulsos):
    print("entro")
    loop = asyncio.get_event_loop()

Resultado=loop.run_until_complete(asyncio.ensure_future
(wait_for_value_async(OP,Carga,Npulsos)))
    print("salio")
    return Resultado
#Register function
server.register_function(Prueba, 'P1')

# Run the server's main loop
server.serve_forever()
```

A.3 Caso de estudio tres

A.3.1 File.sv

```
import "DPI-C" function void Programar();
import "DPI-C" function void Opera(input bit [3:0] A,
input bit [3:0] B, input bit Ci, output bit [3:0] S,
output bit Co);
import "DPI-C" function void Prueba(output bit RP,
output bit [3:0] ADDRSFALL, output bit [3:0] SGA,
output bit [3:0] SGB, output bit SGC, output bit [3:0]
SCLAS, output bit SCLAC, output bit [3:0] SROMS, output
bit SROMC);
import "DPI-C" function void Prueba2(input bit [3:0] n,
output bit RP, output bit [3:0] A, output bit [3:0] B,
output bit C, output bit [3:0] SS, output bit CS,
output bit [3:0] SE, output bit CE);

module always_comb_process();
reg [3:0] A;
reg [3:0] B;
reg      Ci;
reg [3:0] S;
reg      Co;
//PRUEBA 1
reg      RP;
reg [3:0] ADDRSFALL;
reg [3:0] SGA;
reg [3:0] SGB;
reg       SGC;
reg [3:0] SCLAS;
reg       SCLAC;
reg [3:0] SROMS;
reg       SROMC;
//PRUEBA 2
reg [3:0] n;
reg      RP2;
reg [3:0] A2;
reg [3:0] B2;
reg       C2;
reg [3:0] SS2;
reg       CS2;
reg [3:0] SE2;
reg       CE2;
//PRUEBA3
reg      RP3;
reg [3:0] A3;
reg [3:0] B3;
reg      C3;
reg [3:0] S3;
reg      Co3;
reg [4:0] Suma;


initial begin
//Se inicializan valores para operacion en modo manual
A=4;
B=15;
Ci=1;
//Se programa la tarjeta
#1 Programar();
//Se realiza una suma en modo manual y se imprimen
resultados
#1 Opera(A, B, Ci, S, Co);
#1 $display("1) Valores enviados directamente a las
entradas del circuito");
   $display("@%g A = %g, B = %g, Ci = %g, S = %g, Co =
%g",$time, A, B, Ci, S, Co);

//Se realiza una prueba en el sumador, utilizando la
estructura BIST que trae el proyecto
#1 Prueba(RP, ADDRSFALL, SGA, SGB, SGC, SCLAS, SCLAC,
SROMS, SROMC);
```

```
#1 $display("2) Prueba realizada por medio de la
estructura de pruebas en emulacion");
   if(RP==0) begin
   $display("La prueba fallo en ADDRSFALL : %g, SGA :
%g, SGB : %g, SGC : %g SCLAS : %g, SCLAC : %g, SROMS :
%g, SROMC : %g, Tiempo de simulacion : %g", ADDRSFALL,
SGA, SGB, SGC, SCLAS, SCLAC, SROMS, SROMC,$time);
   end else begin
   $display("La prueba con la estructura de pruebas en
emulacion se realizo sin errores");
   end
//Se manda a realizar una prueba mediante una funcion
en el servidor python
#1 n=2;
#1 Prueba2(n, RP2, A2, B2, C2, SS2, CS2, SE2, CE2);
#1 $display("3) Prueba realizada mediante la estructura
ubicada en la interfaz");
   if(RP2==0) begin
   $display("La prueba fallo en A2 : %g, B2 : %g, C2 :
%g SS2 : %g, CS2 : %g, SE2 : %g, CE2 : %g, Tiempo de
simulacion : %g", A2, B2, C2, SS2, CS2, SE2,
CE2,$time);
   end else begin
   $display("La prueba con la estructura de pruebas en
interfaz se realizo sin errores");
   end

//se manda a realizar una prueba mediante una tarea
descrita en el mismo archivos .SV
   $display("4) Prueba realizada por medio de la
estructura ubicada en simulacion");
#1 Prueba3(2);
//se termina el programa.
#1  $finish;
end
```

```
task Prueba3;
input [7:0] n;
begin
RP3=1;
for (int i = 0 ; i < n; i ++) begin
      A3=$urandom_range(0,15);
      B3=$urandom_range(0,15);
      C3=$urandom_range(0,1);
      Suma=A3+B3+C3;
      #1 Opera(A3, B3, C3, S3, Co3);
      #1 if((Suma[3:0]!=S3)||(Suma[4]!=Co3))
         begin
            RP3=0;
            break;
         end
end

if (RP3==1)
begin
  $display("La prueba con la estructura de pruebas en
simulacion se realizo sin errores");
end
else
begin
      $display("La prueba tuvo fallo en A3 : %0d, B3 :
%0d, C3 : %0d, SS3 : %0d, CS3 : %0d, SE3 : %0d, CE3 :
%d",A3,B3, Co3,Suma[3:0], Suma[4],C3, S3);
end
end
endtask
endmodule
```

A.3.2 File.c

```
#include
"/opt/Xilinx/Vivado/2016.1/data/xsim/include/svdpi.h"
#include <stdio.h>
```

```
#include <string.h>
#include <stdlib.h>
#include </usr/include/python2.7/Python.h>

DPI_DLLESPEC
void Programar() {
	printf("Se ejecuto programacion\n");
	PyObject *pName, *pModule, *pDict, *pFunc,
*pValue;;
	Py_Initialize();
	PyRun_SimpleString("import xmlrpclib\n");
	PyRun_SimpleString("s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')\n");
	PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO4");
	pName = PyString_FromString("file0");
	pModule = PyImport_Import(pName);
	pDict   = PyModule_GetDict(pModule);
	pFunc = PyObject_GetAttrString(pModule, "PROG");
	  PyObject_CallObject(pFunc, NULL);
	  PyErr_Print();
	Py_DECREF(pModule);
	  Py_DECREF(pFunc);
	Py_Finalize();
}

DPI_DLLESPEC
void Opera(  svBitVecVal *A,
             svBitVecVal *B,
             svBit Ci,
             svBitVecVal *S,
             svBit *Co
) {
	PyObject *pName, *pModule, *pDict, *pFunc,
*pValue, *pArgs;

	Py_Initialize();
	PyRun_SimpleString("import xmlrpclib\n");
	PyRun_SimpleString("s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')\n");
	PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO4");
	pName = PyString_FromString("file1");
	pModule = PyImport_Import(pName);
	pDict   = PyModule_GetDict(pModule);

	pFunc = PyObject_GetAttrString(pModule,
"OPERA1");
	int Ap, Bp, Sp;
	unsigned char Cip, Cop;
	Ap=*A;
	Bp=*B;
	Cip=Ci;
	pArgs = Py_BuildValue("iii", Ap, Bp, Cip);
	  PyObject* item=PyTuple_GetItem(pArgs,0);
	PyObject* result = PyObject_CallObject(pFunc,
pArgs);
	  PyObject_CallObject(pFunc, pArgs);
	  PyErr_Print();
	PyObject *SX, *CX;
	SX  =PyTuple_GetItem(result,0);
	CX  =PyTuple_GetItem(result,1);
	Sp =(int)PyInt_AsSsize_t(SX);
	Cop =(int)PyInt_AsSsize_t(CX);

	Py_DECREF(pArgs);
	  Py_DECREF(item);
	  Py_DECREF(result);

	Py_DECREF(pModule);
	  Py_DECREF(pFunc);
	Py_Finalize();

	*S=Sp;
```

```
        *Co=Cop;
}




DPI_DLLESPEC
void Prueba( svBit      *RP,
             svBitVecVal *ADDRSFALL,
             svBitVecVal *SGA,
             svBitVecVal *SGB,
             svBit       *SGC,
             svBitVecVal *SCLAS,
             svBit      *SCLAC,
             svBitVecVal *SROMS,
             svBit       *SROMC
) {
        PyObject *pName, *pModule, *pDict, *pFunc,
*pValue, *pArgs;

        Py_Initialize();
        PyRun_SimpleString("import xmlrpclib\n");
        PyRun_SimpleString("s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')\n");
        PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO4");
        pName = PyString_FromString("file2");
        pModule = PyImport_Import(pName);
        pDict   = PyModule_GetDict(pModule);
        pFunc = PyObject_GetAttrString(pModule,
"MPRUEBA");



        PyObject* result;
        result=PyObject_CallObject(pFunc, NULL);
        PyErr_Print();
```

```
        PyObject *RPx, *ADDRSFALLx, *SGAx, *SGBx, *SGCx,
*SCLASx, *SCLACx, *SROMSx, *SROMCx;

        RPx          =PyTuple_GetItem(result,0);
        ADDRSFALLx   =PyTuple_GetItem(result,1);
        SGAx         =PyTuple_GetItem(result,2);
        SGBx         =PyTuple_GetItem(result,3);
        SGCx         =PyTuple_GetItem(result,4);
        SCLASx       =PyTuple_GetItem(result,5);
        SCLACx       =PyTuple_GetItem(result,6);
        SROMSx       =PyTuple_GetItem(result,7);
        SROMCx       =PyTuple_GetItem(result,8);

        int ADDRSFALLx1, SGAx1, SGBx1, SCLASx1, SROMSx1;
        unsigned char RPx1, SGCx1, SCLACx1, SROMCx1;
        RPx1         =(int)PyInt_AsSsize_t(RPx);
        ADDRSFALLx1  =(int)PyInt_AsSsize_t(ADDRSFALLx);
        SGAx1        =(int)PyInt_AsSsize_t(SGAx);
        SGBx1        =(int)PyInt_AsSsize_t(SGBx);
        SGCx1        =(int)PyInt_AsSsize_t(SGCx);
        SCLASx1             =(int)PyInt_AsSsize_t(SCLASx);
        SCLACx1             =(int)PyInt_AsSsize_t(SCLACx);
        SROMSx1             =(int)PyInt_AsSsize_t(SROMSx);
        SROMCx1             =(int)PyInt_AsSsize_t(SROMCx);

          Py_DECREF(result);


        Py_DECREF(pModule);
          Py_DECREF(pFunc);
        Py_Finalize();
        *RP    = RPx1;
        *ADDRSFALL= ADDRSFALLx1;
        *SGA   = SGAx1;
        *SGB   = SGBx1;
        *SGC   = SGCx1;
```

```
	*SCLAS = SCLASx1;
	*SCLAC = SCLACx1;
	*SROMS = SROMSx1;
	*SROMC = SROMCx1;
}

DPI_DLLESPEC
void Prueba2(	svBitVecVal *n,
		svBit	*RP,
		svBitVecVal *A,
		svBitVecVal *B,
		svBit	  *C,
		svBitVecVal *SS,
		svBit	*CS,
		svBitVecVal *SE,
		svBit	  *CE
) {
	PyObject *pName, *pModule, *pDict, *pFunc,
*pValue, *pArgs;

	Py_Initialize();
	PyRun_SimpleString("import xmlrpclib\n");
	PyRun_SimpleString("s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')\n");
	PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO4");
	pName = PyString_FromString("file3");
	pModule = PyImport_Import(pName);
	pDict   = PyModule_GetDict(pModule);
	pFunc = PyObject_GetAttrString(pModule,
"MPRUEBA2");


	int N1;
	N1=*n;
	pArgs = Py_BuildValue("ii", N1,3);
	PyObject* result;
	result=PyObject_CallObject(pFunc, pArgs);
	PyErr_Print();
	PyObject *RP1, *A1, *B1, *C1, *SS1, *CS1, *SE1,
*CE1;

	RP1		=PyTuple_GetItem(result,7);
	A1		=PyTuple_GetItem(result,0);
	B1		=PyTuple_GetItem(result,1);
	C1		=PyTuple_GetItem(result,2);
	SS1		=PyTuple_GetItem(result,3);
	CS1		=PyTuple_GetItem(result,4);
	SE1		=PyTuple_GetItem(result,5);
	CE1		=PyTuple_GetItem(result,6);

	int Ax, Bx, SSx, SEx;
	unsigned char RPx, Cx, CSx, CEx;
	RPx		=(int)PyInt_AsSsize_t(RP1);
	Ax		=(int)PyInt_AsSsize_t(A1);
	Bx		=(int)PyInt_AsSsize_t(B1);
	Cx		=(int)PyInt_AsSsize_t(C1);
	SSx		=(int)PyInt_AsSsize_t(SS1);
	CSx		=(int)PyInt_AsSsize_t(CS1);
	SEx		=(int)PyInt_AsSsize_t(SE1);
	CEx		=(int)PyInt_AsSsize_t(CE1);


	  Py_DECREF(result);



	Py_DECREF(pModule);
	  Py_DECREF(pFunc);
	Py_Finalize();
	*RP	= RPx;
	*A	= Ax;
	*B	= Bx;
	*C	= Cx;
	*SS	= SSx;
```

```
        *CS    = CSx;
        *SE    = SEx;
        *CE    = CEx;
}
```

A.3.3 File0.py

```
import xmlrpclib
def PROG():
    s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')
    s.P();
```

A.3.4 File1.py

```
import xmlrpclib
def OPERA1(A, B, Ci):
    s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')
    S, Co=s.O(A, B, Ci);
    return S, Co
```

A.3.5 File2.py

```
import xmlrpclib
def MPRUEBA():
    s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')
    RP, ADDRSFALL, SGA, SGB, SGC, SCLAS, SCLAC, SROMS,
SROMC=s.P1()
    return RP, ADDRSFALL, SGA, SGB, SGC, SCLAS, SCLAC,
SROMS, SROMC
```

A.3.6 File3.py

```
import xmlrpclib
def MPRUEBA2(n,d):
    s =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')
    a,b,c,ss,cs,se,ce,rp=s.P2(n)
    return a,b,c,ss,cs,se,ce,rp
```

A.3.7 Servidor.py

```
import time
import asyncio
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler
from pynq import MMIO
from pynq import PL
from pynq import Interrupt
from pynq import Overlay
import random

#Programar de inicio
base=Overlay('PL4.bit')
base.download()
#Variable Globales
#Usar bus S00_AXI
CLA     = MMIO(0x43c00000, 512)
CPIPE   = MMIO(0x43c20000, 512)
#Contador
#Usar bus S00_AXI_INTR
CLA_INTR= MMIO(0x43c10000, 512)
#Objeto de interrupcion
interrupt = Interrupt('CLA_BIST_0/irq')

#Controlador de interrupciones
Controlador_INTRP=MMIO(0x41800000, 512)
```

```
#Habilitar interrupciones
CLA_INTR.write(0x0, 0x00000001)
CLA_INTR.write(0x4, 0x00000001)

P=1;



# Restrict to a particular path.
class
RequestHandler(SimpleXMLRPCRequestHandler):rpc_paths =
('/RPC2',)

server = SimpleXMLRPCServer(("192.168.2.98",
8000),requestHandler=RequestHandler)
server.register_introspection_functions()

def Programar():
    global CLA
    global CPIPE
    global CLA_INTR
    global Controlador_INTRP
    global interrupt
    global P
    #Programar de inicio
    base=Overlay('PL4.bit')
    base.download()
    #Variable Globales
    #Usar bus S00_AXI
    CLA     = MMIO(0x43c00000, 512)
    CPIPE   = MMIO(0x43c20000, 512)
    #Contador
    #Usar bus S00_AXI_INTR
    CLA_INTR= MMIO(0x43c10000, 512)
    #Objeto de interrupcion
    interrupt = Interrupt('CLA_BIST_0/irq')
    #Controlador de interrupciones
    Controlador_INTRP=MMIO(0x41800000, 512)

    #Habilitar interrupciones
    CLA_INTR.write(0x0, 0x00000001)
    CLA_INTR.write(0x4, 0x00000001)
    P=1
    return 1

#Register function
server.register_function(Programar, 'P')

#Operacion
def Opera(A, B, C):
    global CLA
    #Metemos entradas en CLA modo manual
    CLA.write(0x0, A)#A--------------slv0
    CLA.write(0x4, B)#B----------------slv1
    CLA.write(0x8, C)#C------------------slv2
    #Leemos salida del CLA
    S=CLA.read(0x30)#SCLASi.......slv12
    C=CLA.read(0x34)#SCLACi.......slv13
    return S, C
#Register function
server.register_function(Opera, 'O')


@asyncio.coroutine
def wait_for_value_async():
        global CLA
        global CPIPE
        global CLA_INTR
        global Controlador_INTRP
        global interrupt
        global P
```

```
        print("E11")
        #Habilitar interrupciones
        CLA_INTR.write(0x0, 0x00000001)
        CLA_INTR.write(0x4, 0x00000001)

        CPIPE.write(0x0, P)#
        if P==1:
            P=0
        else:
            P=1
        print("valor activador CPIPE",P)

        x=1
        while(Controlador_INTRP.read(0x0)==0x01 or
x==1):
            x=2
            print("E2")
            print(CLA_INTR.read(0x0))
            yield from interrupt.wait()
            CLA_INTR.write(0xc, 0x00000001)
        print("E3")

        #Leer resultados de la prueba
        RP=         CLA.read(0xC)
        ADDRSFALL=  CLA.read(0x10)
        SGA=        CLA.read(0x14)
        SGB=        CLA.read(0x18)
        SGC=        CLA.read(0x1C)
        SCLAS=      CLA.read(0x20)
        SCLAC=      CLA.read(0x24)
        SROMS=      CLA.read(0x28)
        SROMC=      CLA.read(0x2C)
        #Leer resultados de la prueba
        print("RP:          ",RP)
        print("ADDRSFALL:  ",ADDRSFALL)
        print("SGA:         ",SGA)
        print("SGB:         ",SGB)
        print("SGC:         ",SGC)
        print("SCLAS:       ",SCLAS)
        print("SCLAC:       ",SCLAC)
        print("SROMS:       ",SROMS)
        print("SROMC:       ",SROMC)


        return RP, ADDRSFALL, SGA, SGB, SGC, SCLAS,
SCLAC, SROMS, SROMC


def Prueba():
    print("entro")
    loop = asyncio.get_event_loop()
    RP, ADDRSFALL, SGA, SGB, SGC, SCLAS, SCLAC, SROMS,
SROMC
=loop.run_until_complete(asyncio.ensure_future(wait_for
_value_async()))
    print("salio")
    print("RP:          ",RP)
    print("ADDRSFALL:  ",ADDRSFALL)
    print("SGA:         ",SGA)
    print("SGB:         ",SGB)
    print("SGC:         ",SGC)
    print("SCLAS:       ",SCLAS)
    print("SCLAC:       ",SCLAC)
    print("SROMS:       ",SROMS)
    print("SROMC:       ",SROMC)
    return RP, ADDRSFALL, SGA, SGB, SGC, SCLAS, SCLAC,
SROMS, SROMC
#Register function
server.register_function(Prueba, 'P1')


def Prueba2(n):
    global CLA
    rp=1
```

```
    for i in range(0, n):
        a=random.randrange(16)
        b=random.randrange(16)
        c=random.randrange(2)
        s=a+b+c
        cs=s>>4;
        ss=s&15

        CLA.write(0x0, a)#A---------------slv0
        CLA.write(0x4, b)#B-----------------slv1
        CLA.write(0x8, c)#C-------------------slv2
        #Leemos salida del CLA
        se=CLA.read(0x30)#SCLASi.......slv12
        ce=CLA.read(0x34)#SCLACi.......slv13

        if(cs!=ce or ss!=se):
            rp=0
            break

    print("el valor de rp:",rp)
    return a,b,c,ss,cs,se,ce, rp

#Register function
server.register_function(Prueba2, 'P2')



# Run the server's main loop
server.serve_forever()
```

A.4 Caso de estudio cuatro

A.4.1 File.sv

```
import "DPI-C" function void Programar1();

import "DPI-C" function void Programar2();

import "DPI-C" function void Opera1(input bit [31:0]
SMUX3, input bit [31:0] INST4, output bit [31:0] REG1,
output bit [31:0] SMUX2, output bit [31:0] REG2, output
bit [31:0] INST1);

import "DPI-C" function void Opera2(output bit [31:0]
SMUX3, output bit [31:0] INST4, input bit [31:0] REG1,
input bit [31:0] SMUX2, input bit [31:0] REG2, input
bit [31:0] INST1);

import "DPI-C" function void Lreg(output bit [31:0] R0,
output bit [31:0] R1, output bit [31:0] R2, output bit
[31:0] R3, output bit [31:0] R4, output bit [31:0] R5,
output bit [31:0] R6, output bit [31:0] R7, output bit
[31:0] R8, output bit [31:0] R9, output bit [31:0] R10,
output bit [31:0] R11, output bit [31:0] R12, output
bit [31:0] R13, output bit [31:0] R14, output bit
[31:0] R15, output bit [31:0] R16, output bit [31:0]
R17, output bit [31:0] R18, output bit [31:0] R19,
output bit [31:0] R20, output bit [31:0] R21, output
bit [31:0] R22, output bit [31:0] R23, output bit
[31:0] R24, output bit [31:0] R25, output bit [31:0]
R26, output bit [31:0] R27, output bit [31:0] R28,
output bit [31:0] R29, output bit [31:0] R30, output
bit [31:0] R31);
```

```
module always_comb_process();
reg [31:0] SMUX3;
reg [31:0] INST4;
reg [31:0] REG1;
reg [31:0] SMUX2;
reg [31:0] REG2;
reg [31:0] INST1;
reg [31:0] R0;
reg [31:0] R1;
reg [31:0] R2;
reg [31:0] R3;
reg [31:0] R4;
reg [31:0] R5;
reg [31:0] R6;
reg [31:0] R7;
reg [31:0] R8;
reg [31:0] R9;
reg [31:0] R10;
reg [31:0] R11;
reg [31:0] R12;
reg [31:0] R13;
reg [31:0] R14;
reg [31:0] R15;
reg [31:0] R16;
reg [31:0] R17;
reg [31:0] R18;
reg [31:0] R19;
reg [31:0] R20;
reg [31:0] R21;
reg [31:0] R22;
reg [31:0] R23;
reg [31:0] R24;
reg [31:0] R25;
reg [31:0] R26;
reg [31:0] R27;
reg [31:0] R28;
reg [31:0] R29;
```

```
reg [31:0] R30;
reg [31:0] R31;

initial begin


#1 Programar1();
#1 Programar2();
   SMUX3=0;
   INST4=0;
   REG1=0;
   SMUX2=0;
   REG2=0;
   INST1=0;

#1 Lreg(R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10,
R11, R12, R13, R14, R15, R16, R17, R18, R19,R20, R21,
R22, R23, R24, R25, R26, R27, R28, R29, R30, R31);
#1$display(" R0: %g, R1: %g, R2: %g, R3: %g R4: %g R5:
%g R6: %g R7: %g R8: %g R9: %g R10: %g, R11: %g, R12:
%g, R13: %g R14: %g R15: %g R16: %g R17: %g R18: %g
R19: %g R20: %g, R21: %g, R22: %g, R23: %g R24: %g R25:
%g R26: %g R27: %g R28: %g R29: %g R30: %g R31: %g
@%g", R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11,
R12, R13, R14, R15, R16, R17, R18, R19,R20, R21, R22,
R23, R24, R25, R26, R27, R28, R29, R30, R31,$time);

for (int i = 0 ; i <= 33; i ++) begin
#1 Opera1(SMUX3, INST4, REG1, SMUX2, REG2, INST1);
#1 Opera2(SMUX3, INST4, REG1, SMUX2, REG2, INST1);
end

#1 Lreg(R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10,
R11, R12, R13, R14, R15, R16, R17, R18, R19,R20, R21,
R22, R23, R24, R25, R26, R27, R28, R29, R30, R31);
#1$display(" R0: %g, R1: %g, R2: %g, R3: %g R4: %g R5:
%g R6: %g R7: %g R8: %g R9: %g R10: %g, R11: %g, R12:
%g, R13: %g R14: %g R15: %g R16: %g R17: %g R18: %g
R19: %g R20: %g, R21: %g, R22: %g, R23: %g R24: %g R25:
%g R26: %g R27: %g R28: %g R29: %g R30: %g R31: %g
@%g", R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11,
R12, R13, R14, R15, R16, R17, R18, R19,R20, R21, R22,
R23, R24, R25, R26, R27, R28, R29, R30, R31,$time);


for (int i = 0 ; i <= 15; i ++) begin
#1 Opera1(SMUX3, INST4, REG1, SMUX2, REG2, INST1);
#1 Opera2(SMUX3, INST4, REG1, SMUX2, REG2, INST1);
end

#1 Lreg(R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10,
R11, R12, R13, R14, R15, R16, R17, R18, R19,R20, R21,
R22, R23, R24, R25, R26, R27, R28, R29, R30, R31);
#1$display(" R0: %g, R1: %g, R2: %g, R3: %g R4: %g R5:
%g R6: %g R7: %g R8: %g R9: %g R10: %g, R11: %g, R12:
%g, R13: %g R14: %g R15: %g R16: %g R17: %g R18: %g
R19: %g R20: %g, R21: %g, R22: %g, R23: %g R24: %g R25:
%g R26: %g R27: %g R28: %g R29: %g R30: %g R31: %g
@%g", R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11,
R12, R13, R14, R15, R16, R17, R18, R19,R20, R21, R22,
R23, R24, R25, R26, R27, R28, R29, R30, R31,$time);

for (int i = 0 ; i <= 15; i ++) begin
#1 Opera1(SMUX3, INST4, REG1, SMUX2, REG2, INST1);
#1 Opera2(SMUX3, INST4, REG1, SMUX2, REG2, INST1);
end

#1 Lreg(R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10,
R11, R12, R13, R14, R15, R16, R17, R18, R19,R20, R21,
R22, R23, R24, R25, R26, R27, R28, R29, R30, R31);
#1$display(" R0: %g, R1: %g, R2: %g, R3: %g R4: %g R5:
%g R6: %g R7: %g R8: %g R9: %g R10: %g, R11: %g, R12:
%g, R13: %g R14: %g R15: %g R16: %g R17: %g R18: %g
R19: %g R20: %g, R21: %g, R22: %g, R23: %g R24: %g R25:
```

```
%g R26: %g R27: %g R28: %g R29: %g R30: %g R31: %g
@%g", R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11,
R12, R13, R14, R15, R16, R17, R18, R19,R20, R21, R22,
R23, R24, R25, R26, R27, R28, R29, R30, R31,$time);

#1 $finish;
end

endmodule
```

A.4.2 File.c

```
#include
"/opt/Xilinx/Vivado/2016.1/data/xsim/include/svdpi.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include </usr/include/python2.7/Python.h>

DPI_DLLESPEC
void Programar1() {
	printf("Se ejecuto programacion FPGA1\n");
	PyObject *pName, *pModule, *pDict, *pFunc,
*pValue;
	Py_Initialize();
	PyRun_SimpleString("import xmlrpclib\n");
	PyRun_SimpleString("s1 =
xmlrpclib.ServerProxy('http://192.168.2.99:8000')\n");
	PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO3F");
	pName = PyString_FromString("file0");
	pModule = PyImport_Import(pName);
	pDict   = PyModule_GetDict(pModule);
	pFunc = PyObject_GetAttrString(pModule, "PROG");
	  PyObject_CallObject(pFunc, NULL);
	  PyErr_Print();
	Py_DECREF(pModule);
	  Py_DECREF(pFunc);
	Py_Finalize();
}

DPI_DLLESPEC
void Opera1( svBitVecVal *SMUX3,
             svBitVecVal *INST4,
             svBitVecVal *REG1,
             svBitVecVal *SMUX2,
             svBitVecVal *REG2,
             svBitVecVal *INST1
) {
	PyObject *pName, *pModule, *pDict, *pFunc,
*pValue, *pArgs;


	Py_Initialize();
	PyRun_SimpleString("import xmlrpclib\n");
	PyRun_SimpleString("s1 =
xmlrpclib.ServerProxy('http://192.168.2.99:8000')\n");
	PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO3F");
	pName = PyString_FromString("file1");
	pModule = PyImport_Import(pName);
	pDict   = PyModule_GetDict(pModule);
	pFunc = PyObject_GetAttrString(pModule,
"OPERA1");


	int SMUX31, INST41, REG11, SMUX21, REG21, INST11,
Resul;
	SMUX31=*SMUX3;
	INST41=*INST4;
	/*
	printf("SMUX3:%d\n", SMUX31);
	printf("INST4:%d\n", INST41);
	*/
```

```
pArgs = Py_BuildValue("ii", SMUX31, INST41);
  PyObject* item=PyTuple_GetItem(pArgs,0);


PyObject* result;
result=PyObject_CallObject(pFunc, pArgs);
PyErr_Print();


PyObject *REG1X, *SMUX2X, *REG2X, *INST1X;
REG1X  =PyTuple_GetItem(result,0);
SMUX2X =PyTuple_GetItem(result,1);
REG2X  =PyTuple_GetItem(result,2);
INST1X  =PyTuple_GetItem(result,3);
PyErr_Print();

REG11  =(int)PyInt_AsSsize_t(REG1X);
SMUX21 =(int)PyInt_AsSsize_t(SMUX2X);
REG21  =(int)PyInt_AsSsize_t(REG2X);
INST11 =(int)PyInt_AsSsize_t(INST1X);

/*
printf("REG1:%d\n", REG11);
printf("SMUX1:%d\n", SMUX21);
printf("REG2:%d\n", REG21);
printf("INST1:%x\n", INST11);
*/

Py_DECREF(pArgs);
  Py_DECREF(item);
  Py_DECREF(result);


Py_DECREF(pModule);
  Py_DECREF(pFunc);
Py_Finalize();

        *REG1=REG11;
        *SMUX2=SMUX21;
        *REG2=REG21;
        *INST1=INST11;
}

DPI_DLLESPEC
void Lreg(   svBitVecVal *R0,
             svBitVecVal *R1,
             svBitVecVal *R2,
             svBitVecVal *R3,
             svBitVecVal *R4,
             svBitVecVal *R5,
             svBitVecVal *R6,
             svBitVecVal *R7,
             svBitVecVal *R8,
             svBitVecVal *R9,
             svBitVecVal *R10,
             svBitVecVal *R11,
             svBitVecVal *R12,
             svBitVecVal *R13,
             svBitVecVal *R14,
             svBitVecVal *R15,
             svBitVecVal *R16,
             svBitVecVal *R17,
             svBitVecVal *R18,
             svBitVecVal *R19,
             svBitVecVal *R20,
             svBitVecVal *R21,
             svBitVecVal *R22,
             svBitVecVal *R23,
             svBitVecVal *R24,
             svBitVecVal *R25,
```

```
            svBitVecVal *R26,
            svBitVecVal *R27,
            svBitVecVal *R28,
            svBitVecVal *R29,
            svBitVecVal *R30,
            svBitVecVal *R31
) {
      PyObject *pName, *pModule, *pDict, *pFunc,
*pValue;


      Py_Initialize();
      PyRun_SimpleString("import xmlrpclib\n");
      PyRun_SimpleString("s1 =
xmlrpclib.ServerProxy('http://192.168.2.99:8000')\n");
      PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO3F");
      pName = PyString_FromString("file2");
      pModule = PyImport_Import(pName);
      pDict   = PyModule_GetDict(pModule);
      pFunc = PyObject_GetAttrString(pModule, "LREG");


      int RE0, RE1, RE2, RE3, RE4, RE5, RE6, RE7, RE8,
RE9, RE10, RE11, RE12, RE13, RE14, RE15, RE16, RE17,
RE18, RE19, RE20, RE21, RE22, RE23, RE24, RE25, RE26,
RE27, RE28, RE29, RE30, RE31;

      PyObject* result;
      result=PyObject_CallObject(pFunc, NULL);
      PyErr_Print();


      PyObject *RX0, *RX1, *RX2, *RX3, *RX4, *RX5,
*RX6, *RX7, *RX8, *RX9, *RX10, *RX11, *RX12, *RX13,
*RX14, *RX15, *RX16, *RX17, *RX18, *RX19, *RX20, *RX21,
*RX22, *RX23, *RX24, *RX25, *RX26, *RX27, *RX28, *RX29,
*RX30, *RX31 ;

      RX0  =PyTuple_GetItem(result,0);
      RX1  =PyTuple_GetItem(result,1);
      RX2  =PyTuple_GetItem(result,2);
      RX3  =PyTuple_GetItem(result,3);
      RX4  =PyTuple_GetItem(result,4);
      RX5  =PyTuple_GetItem(result,5);
      RX6  =PyTuple_GetItem(result,6);
      RX7  =PyTuple_GetItem(result,7);
      RX8  =PyTuple_GetItem(result,8);
      RX9  =PyTuple_GetItem(result,9);
      RX10  =PyTuple_GetItem(result,10);
      RX11  =PyTuple_GetItem(result,11);
      RX12  =PyTuple_GetItem(result,12);
      RX13  =PyTuple_GetItem(result,13);
      RX14  =PyTuple_GetItem(result,14);
      RX15  =PyTuple_GetItem(result,15);
      RX16  =PyTuple_GetItem(result,16);
      RX17  =PyTuple_GetItem(result,17);
      RX18  =PyTuple_GetItem(result,18);
      RX19  =PyTuple_GetItem(result,19);
      RX20  =PyTuple_GetItem(result,20);
      RX21  =PyTuple_GetItem(result,21);
      RX22  =PyTuple_GetItem(result,22);
      RX23  =PyTuple_GetItem(result,23);
      RX24  =PyTuple_GetItem(result,24);
      RX25  =PyTuple_GetItem(result,25);
      RX26  =PyTuple_GetItem(result,26);
      RX27  =PyTuple_GetItem(result,27);
      RX28  =PyTuple_GetItem(result,28);
      RX29  =PyTuple_GetItem(result,29);
      RX30  =PyTuple_GetItem(result,30);
      RX31  =PyTuple_GetItem(result,31);
      PyErr_Print();
```

```
RE0  =(int)PyInt_AsSsize_t(RX0);
RE1  =(int)PyInt_AsSsize_t(RX1);
RE2  =(int)PyInt_AsSsize_t(RX2);
RE3  =(int)PyInt_AsSsize_t(RX3);
RE4  =(int)PyInt_AsSsize_t(RX4);
RE5  =(int)PyInt_AsSsize_t(RX5);
RE6  =(int)PyInt_AsSsize_t(RX6);
RE7  =(int)PyInt_AsSsize_t(RX7);
RE8  =(int)PyInt_AsSsize_t(RX8);
RE9  =(int)PyInt_AsSsize_t(RX9);
RE10  =(int)PyInt_AsSsize_t(RX10);
RE11  =(int)PyInt_AsSsize_t(RX11);
RE12  =(int)PyInt_AsSsize_t(RX12);
RE13  =(int)PyInt_AsSsize_t(RX13);
RE14  =(int)PyInt_AsSsize_t(RX14);
RE15  =(int)PyInt_AsSsize_t(RX15);
RE16  =(int)PyInt_AsSsize_t(RX16);
RE17  =(int)PyInt_AsSsize_t(RX17);
RE18  =(int)PyInt_AsSsize_t(RX18);
RE19  =(int)PyInt_AsSsize_t(RX19);
RE20  =(int)PyInt_AsSsize_t(RX20);
RE21  =(int)PyInt_AsSsize_t(RX21);
RE22  =(int)PyInt_AsSsize_t(RX22);
RE23  =(int)PyInt_AsSsize_t(RX23);
RE24  =(int)PyInt_AsSsize_t(RX24);
RE25  =(int)PyInt_AsSsize_t(RX25);
RE26  =(int)PyInt_AsSsize_t(RX26);
RE27  =(int)PyInt_AsSsize_t(RX27);
RE28  =(int)PyInt_AsSsize_t(RX28);
RE29  =(int)PyInt_AsSsize_t(RX29);
RE30  =(int)PyInt_AsSsize_t(RX30);
RE31  =(int)PyInt_AsSsize_t(RX31);

/*
printf("Valor actual de los registros\n");
printf("REG0:%d\n", RE0);
printf("REG1:%d\n", RE1);
printf("REG2:%d\n", RE2);
printf("REG3:%d\n", RE3);
printf("REG4:%d\n", RE4);
printf("REG5:%d\n", RE5);
printf("REG6:%d\n", RE6);
printf("REG7:%d\n", RE7);
printf("REG8:%d\n", RE8);
printf("REG9:%d\n", RE9);
printf("REG10:%d\n", RE10);
printf("REG11:%d\n", RE11);
printf("REG12:%d\n", RE12);
printf("REG13:%d\n", RE13);
printf("REG14:%d\n", RE14);
printf("REG15:%d\n", RE15);
printf("REG16:%d\n", RE16);
printf("REG17:%d\n", RE17);
printf("REG18:%d\n", RE18);
printf("REG19:%d\n", RE19);
printf("REG20:%d\n", RE20);
printf("REG21:%d\n", RE21);
printf("REG22:%d\n", RE22);
printf("REG23:%d\n", RE23);
printf("REG24:%d\n", RE24);
printf("REG25:%d\n", RE25);
printf("REG26:%d\n", RE26);
printf("REG27:%d\n", RE27);
printf("REG28:%d\n", RE28);
printf("REG29:%d\n", RE29);
printf("REG30:%d\n", RE30);
printf("REG31:%d\n", RE31);
*/
 Py_DECREF(result);


Py_DECREF(pModule);
```

```
  Py_DECREF(pFunc);
Py_Finalize();


*R0=RE0;
*R1=RE1;
*R2=RE2;
*R3=RE3;
*R4=RE4;
*R5=RE5;
*R6=RE6;
*R7=RE7;
*R8=RE8;
*R9=RE9;
*R10=RE10;
*R11=RE11;
*R12=RE12;
*R13=RE13;
*R14=RE14;
*R15=RE15;
*R16=RE16;
*R17=RE17;
*R18=RE18;
*R19=RE19;
*R20=RE20;
*R21=RE21;
*R22=RE22;
*R23=RE23;
*R24=RE24;
*R25=RE25;
*R26=RE26;
*R27=RE27;
*R28=RE28;
*R29=RE29;
*R30=RE30;
*R31=RE31;
}


DPI_DLLESPEC
void Programar2() {
      printf("Se ejecuto programacion FPGA2\n");
      PyObject *pName, *pModule, *pDict, *pFunc,
*pValue;
      Py_Initialize();
      PyRun_SimpleString("import xmlrpclib\n");
      PyRun_SimpleString("s2 =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')\n");
      PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO3F");
      pName = PyString_FromString("file3");
      pModule = PyImport_Import(pName);
      pDict   = PyModule_GetDict(pModule);
      pFunc = PyObject_GetAttrString(pModule, "PROG");
        PyObject_CallObject(pFunc, NULL);
        PyErr_Print();
      Py_DECREF(pModule);
        Py_DECREF(pFunc);
      Py_Finalize();
}

DPI_DLLESPEC


DPI_DLLESPEC
void Opera2( svBitVecVal *SMUX3,
             svBitVecVal *INST4,
             svBitVecVal *REG1,
             svBitVecVal *SMUX2,
             svBitVecVal *REG2,
             svBitVecVal *INST1
) {
```

```
	PyObject *pName, *pModule, *pDict, *pFunc,
*pValue, *pArgs;


	Py_Initialize();
	PyRun_SimpleString("import xmlrpclib\n");
	PyRun_SimpleString("s2 =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')\n");
	PySys_SetPath("/home/fernando/Escritorio/MAESTRIA
/PROYECTO3F");
	pName = PyString_FromString("file4");
	pModule = PyImport_Import(pName);
	pDict   = PyModule_GetDict(pModule);
	pFunc = PyObject_GetAttrString(pModule,
"OPERA2");


	int SMUX31, INST41, REG11, SMUX21, REG21, INST11,
Resul;
	REG11=*REG1;
	SMUX21=*SMUX2;
	REG21=*REG2;
	INST11=*INST1;

	/*
	printf("\nREG1:%d\n", REG11);
	printf("SMUX2:%d\n", SMUX21);
	printf("REG2:%d\n", REG21);
	printf("INST1:%d\n", INST11);

	*/

	pArgs = Py_BuildValue("iiii", REG11, SMUX21,
REG21, INST11);
	  PyObject* item=PyTuple_GetItem(pArgs,0);
	PyObject* result;
	result=PyObject_CallObject(pFunc, pArgs);
	PyErr_Print();


	PyObject *SMUX3X, *INST4X;
	SMUX3X  =PyTuple_GetItem(result,0);
	INST4X =PyTuple_GetItem(result,1);
	PyErr_Print();

	SMUX31  =(int)PyInt_AsSsize_t(SMUX3X);
	INST41  =(int)PyInt_AsSsize_t(INST4X);

	/*
	printf("SMUX3:%d\n", SMUX31);
	printf("INST4:%x\n", INST41);
	*/
	Py_DECREF(pArgs);
	  Py_DECREF(item);
	  Py_DECREF(result);


	Py_DECREF(pModule);
	  Py_DECREF(pFunc);
	Py_Finalize();




	*SMUX3=SMUX31;
	*INST4=INST41;
}
```

A.4.2 File0.py

```
import xmlrpclib
```

```
def PROG():
      s1 =
xmlrpclib.ServerProxy('http://192.168.2.99:8000')
      s1.P();
```

A.4.3 File1.py

```
import xmlrpclib
def OPERA1(smux3, inst4):
      s1 =
xmlrpclib.ServerProxy('http://192.168.2.99:8000')
      reg1, smux2, reg2, inst1=s1.O1(smux3, inst4);
      return reg1, smux2, reg2, inst1
```

A.4.4 File2.py

```
import xmlrpclib
def LREG():
      s1 =
xmlrpclib.ServerProxy('http://192.168.2.99:8000')
      R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11,
R12, R13, R14, R15, R16, R17, R18, R19,R20, R21, R22,
R23, R24, R25, R26, R27, R28, R29, R30, R31 =s1.REG();
      return R0, R1, R2, R3, R4, R5, R6, R7, R8, R9,
R10, R11, R12, R13, R14, R15, R16, R17, R18, R19,R20,
R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31
```

A.4.3 File3.py

```
import xmlrpclib
def PROG():
      s2 =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')
      s2.P();
```

A.4.4 File4.py

```
import xmlrpclib
def OPERA2(reg1, smux2, reg2, inst1):
      s2 =
xmlrpclib.ServerProxy('http://192.168.2.98:8000')
      smux3, inst4=s2.O2(reg1, smux2, reg2, inst1);
      return smux3, inst4
```

A.4.5 Servidor1.py

```
import time
import asyncio
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler
from pynq import MMIO
from pynq import PL
from pynq import Interrupt
from pynq import Overlay


#Programar de inicio
base=Overlay('PL3_11.bit')
base.download()
#Variable Globales
CPIPES1    = MMIO(0x43c20000, 512)#Controlador de PIPES
INTERFACE1 = MMIO(0x43cF0000, 512)#Primera interface,
entradas y salidas de lo que se encuentra antes del alu
MEM32x32   = MMIO(0x43c60000, 512)#memoria de registros


REG1=0;
SMUX2=0;
```

```
REG2=0;
I1=0;
SMUX3=0;
I4=0;
X=1;


# Restrict to a particular path.
class
RequestHandler(SimpleXMLRPCRequestHandler):rpc_paths =
('/RPC2',)

server = SimpleXMLRPCServer(("192.168.2.99",
8000),requestHandler=RequestHandler)
server.register_introspection_functions()

def Programar():
    global CPIPES1
    global INTERFACE1
    global MEM32x32

    global X
    #Programar de inicio
    base=Overlay('PL3_11.bit')
    base.download()
    #Variable Globales
    CPIPES1    = MMIO(0x43c20000, 512)#Controlador de
PIPES
    INTERFACE1 = MMIO(0x43cF0000, 512)#Primera
interface, entradas y salidas de lo que se encuentra
antes del alu
    MEM32x32   = MMIO(0x43c60000, 512)#memoria de
registros

    X=1;
    return 1
#Register function
server.register_function(Programar, 'P')

#Operacion
def Opera1(SMUX3, I4):
    global INTERFACE1
    global CPIPES1
    global X
    #Operacion
    INTERFACE1.write(0x0, SMUX3)#escribir un valor a WD
o SMUX3
    INTERFACE1.write(0x4, I4)#escribir un valor a INST4

    CPIPES1.write(0x0, X)#
    if X==1:
        X=0
    else:
        X=1
    print(X)

    REG1=INTERFACE1.read(0x08)
    SMUX2=INTERFACE1.read(0x0C)
    REG2=INTERFACE1.read(0x010)
    I1=INTERFACE1.read(0x014)
    print("reg1", INTERFACE1.read(0x08))
    print("SMUX2", INTERFACE1.read(0x0c))
    print("reg2", INTERFACE1.read(0x10))
    print("I1", INTERFACE1.read(0x14))
    return REG1, SMUX2, REG2, I1
#Register function
server.register_function(Opera1, 'O1')


def Registers():
    global MEM32x32
    R0= MEM32x32.read(0x00)
    R1= MEM32x32.read(0x04)
```

```
    R2= MEM32x32.read(0x08)
    R3= MEM32x32.read(0x0c)
    R4= MEM32x32.read(0x010)
    R5= MEM32x32.read(0x014)
    R6= MEM32x32.read(0x018)
    R7= MEM32x32.read(0x01c)
    R8= MEM32x32.read(0x020)
    R9= MEM32x32.read(0x024)
    R10= MEM32x32.read(0x028)
    R11= MEM32x32.read(0x02c)
    R12= MEM32x32.read(0x030)
    R13= MEM32x32.read(0x034)
    R14= MEM32x32.read(0x038)
    R15= MEM32x32.read(0x03c)
    R16= MEM32x32.read(0x040)
    R17= MEM32x32.read(0x044)
    R18= MEM32x32.read(0x048)
    R19= MEM32x32.read(0x04c)
    R20= MEM32x32.read(0x050)
    R21= MEM32x32.read(0x054)
    R22= MEM32x32.read(0x058)
    R23= MEM32x32.read(0x05c)
    R24= MEM32x32.read(0x060)
    R25= MEM32x32.read(0x064)
    R26= MEM32x32.read(0x068)
    R27= MEM32x32.read(0x06c)
    R28= MEM32x32.read(0x070)
    R29= MEM32x32.read(0x074)
    R30= MEM32x32.read(0x078)
    R31= MEM32x32.read(0x07c)
    return R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10,
R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21,
R22, R23, R24, R25, R26, R27, R28, R29, R30, R31

server.register_function(Registers, 'REG')
# Run the server's main loop
server.serve_forever()
```

A.4.6 Servidor2.py

```
import time
import asyncio
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler
from pynq import MMIO
from pynq import PL
from pynq import Interrupt
from pynq import Overlay


#Programar de inicio
base=Overlay('PL3_2.bit')
base.download()
#Variable Globales
CPIPES2    = MMIO(0x43DB0000, 512)#Controlador de PIPES
INTERFACE2 = MMIO(0x43E00000, 512)# Interface 2

REG1=0;
SMUX2=0;
REG2=0;
I1=0;
SMUX3=0;
I4=0;

X1=1;


# Restrict to a particular path.
class
RequestHandler(SimpleXMLRPCRequestHandler):rpc_paths =
('/RPC2',)

server = SimpleXMLRPCServer(("192.168.2.98",
8000),requestHandler=RequestHandler)
server.register_introspection_functions()
```

```
def Programar():

    global CPIPES2
    global INTERFACE2

    global X1
    #Programar de inicio
    base=Overlay('PL3_2.bit')
    base.download()
    #Variable Globales
    CPIPES1    = MMIO(0x43c20000, 512)#Controlador de
PIPES
    INTERFACE1 = MMIO(0x43cF0000, 512)#Primera
interface, entradas y salidas de lo que se encuentra
antes del alu
    MEM32x32   = MMIO(0x43c60000, 512)#memoria de
registros
    CPIPES2    = MMIO(0x43DB0000, 512)#Controlador de
PIPES
    INTERFACE2 = MMIO(0x43E00000, 512)# Interface 2

    X1=1;
    return 1

#Register function
server.register_function(Programar, 'P')


def Opera2(REG1, SMUX2, REG2, I1):
    global INTERFACE2
    global CPIPES2
    global X1
    INTERFACE2.write(0x0, REG1)#REG1
    INTERFACE2.write(0x4, SMUX2)#SMUX1
    INTERFACE2.write(0x8, REG2)#REG2
    INTERFACE2.write(0xC, I1)#instruccion--sumar reg1
con smux1

    CPIPES2.write(0x0,X1)#
    if X1==1:
        X1=0
    else:
        X1=1
    print(X1)

    SMUX3=INTERFACE2.read(0x10)
    I4=INTERFACE2.read(0x14)
    print("SMUX3", INTERFACE2.read(0x10))
    print("instruccion", INTERFACE2.read(0x14))
    return SMUX3, I4

server.register_function(Opera2, 'O2')


# Run the server's main loop
server.serve_forever()
```

Apéndice B.

B.1 Requerimientos de Software

Para usar el esquema de comunicación propuesto se utilizó el siguiente entorno de software:

- Vivado 2016.1
- Ubuntu v.10

B.2 Requerimientos de Hardware

Para usar el esquema de comunicación propuesto se debe tener el siguiente hardware:

- Un kit de desarrollo PYNQ-Z1

B.3 Manuales y guías de interés.

En esta sección se listan información que podría ser de interés para utilizar el esquema de comunicación propuesto, ubicada en manuales provistos por Xilinx, y desarrolladores del framework PYNQ:

- En [22] se encuentra la guía de inicio para el uso de la tarjeta PYNQ-Z1.
- En [23] se encuentra información relacionada con el manejo el diseño de hardware mediante bloques IP en Vivado.
- En [24] se encuentra información relacionada con la creación de IP’s personalizadas.

Apéndice C. Otros Resultados

Como parte de mi trabajo de tesis se realizó una publicación, donde se dan a conocer resultados parciales conforme al caso de estudio 3, correspondiente al Microprocesador pipelined de cuatro etapas. A continuación, se muestra la información correspondiente.

Scheme for Partitioned Co-emulation using Multi PYNQ-Z1 boards, E. Tapia-Morales, J. Martínez-Carballido, and G. Castro-Muñoz, "*ESCS'18 - The 16th Int'l Conf on Embedded Systems, Cyber-physical Systems, and Applications*", pp 17-21, Jul 30-Aug 2, Las Vegas,Nevada, USA.

Referencias

[1] G. Moore, «Excerpts from A Conversation with Gordon Moore: Moore´s Law», 2005.

[2] A. R. Brás, «Systems on Chip: Evolutionary and Revolutionary Trends», presentado en 3rd International Conference on Computer Architecture (ICCA'02), 2002, pp. 121-128.

[3] C.-Y. (Ric) Huang, Y.-F. Yin, C.-J. Hsu, T. B. Huang, y T.-M. Chang, «SoC hw/sw verification and validation», presentado en Proc. of the 16th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 297-300.

[4] R. Wiśniewski, A. Bukowiec, y M. Węgrzyn, «Benefits of hardware accelerated simulation», en *Proceedings of the International Workshop Discrete-Event System Design (DESDes), Poland*, 2001, pp. 229–234.

[5] P. Mishra y R. S. Chakraborty, «Tutorial T2B: Hardware Intellectual Property (IP) Security and Trust: Challenges and Solutions».

[6] P. Rashinkar, P. Paterson, y L. Singh, *System-on-a-chip verification: methodology and techniques*. Springer Science & Business Media, 2007.

[7] N. Surantha, N. Sutisna, Y. Nagao, y H. Ochi, «SoC design with HW/SW co-design methodology for wireless communication system», en *Communications and Information Technologies (ISCIT), 2017 17th International Symposium on*, 2017, pp. 1–6.

[8] P. Mishra y N. D. Dutt, *Functional verification of programmable embedded architectures: a top-down approach*. Springer Science & Business Media, 2005.

[9] M. E. Radu y S. M. Sexton, «Integrating extensive functional verification into digital design education», *IEEE Trans. Educ.*, vol. 51, n.º 3, pp. 385–393, 2008.

[10] A. Basak, S. Mal-Sarkar, y S. Bhunia, «Secure and trusted SoC: Challenges and emerging solutions», en *Microprocessor Test and Verification (MTV), 2013 14th International Workshop on*, 2013, pp. 29–34.

[11] A. W. Ruan, Y. B. Liao, P. Li, W. C. Li, y W. Li, «An improved data communication mechanism for a SOC hardware/software co-emulation environment», en *Communications, Circuits and Systems, 2009. ICCCAS 2009. International Conference on*, 2009, pp. 1029–1032.

[12] B. J. Tomas, Y. Jiang, y M. Yang, «Co-Emulation of Scan-Chain Based Designs Utilizing SCE-MI Infrastructure», *ArXiv Prepr. ArXiv14093276*, 2014.

[13] I. Wagner y V. Bertacco, «Post-Silicon Verification of Multi-Core Processors», en *Post-Silicon and Runtime Verification for Modern Processors*, Springer, 2011, pp. 75–93.

[14] «Emulation market growing. Mentor ups the ante. | EE Times». [En línea]. Disponible en: https://www.eetimes.com/document.asp?doc_id=1317167. [Accedido: 02-jul-2018].

[15] «Emulation - Tech Design Forum Techniques». [En línea]. Disponible en: http://www.techdesignforums.com/practice/guides/guide-to-emulation/. [Accedido: 02-jul-2018].

[16] «Watching the Hardware Emulation Market Take Off | EE Times». [En línea]. Disponible en: https://www.eetimes.com/author.asp?section_id=36&doc_id=1321821. [Accedido: 02-jul-2018].

[17] J. Bergeron, *Writing testbenches: functional verification of HDL models*. Springer Science & Business Media, 2012.

[18] L. Séméria y A. Ghosh, «Methodology for hardware/software co-verification in C/C++ (short paper)», en *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, 2000, pp. 405–408.
[19] D. Carroll y R. Gallery, «Rapid-Prototyping Emulation System for Embedded System Hardware Verification, using a SystemC Control System Environment and Reconfigurable Multimedia Hardware Development Platform», *ITB J.*, vol. 7, n.º 1, p. 5, 2006.
[20] H. Yang, Y. Yi, y S. Ha, «A timed HW/SW coemulation technique for fast yet accurate system verification», en *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS'09. International Symposium on*, 2009, pp. 74–81.
[21] M. D. Pulukuri y C. E. Stroud, «On built-in self-test for adders», *J. Electron. Test.*, vol. 25, n.º 6, p. 343, 2009.
[22] «PYNQ - Python productivity for Zynq», *PYNQ - Python productivity for Zynq*. [En línea]. Disponible en: http://www.pynq.io/. [Accedido: 03-jul-2018].
[23] «ug994-vivado-ip-subsystems». [En línea]. Disponible en: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_3/ug994-vivado-ip-subsystems.pdf. [Accedido: 03-jul-2018].
[24] «ug1119-vivado-creating-packaging-ip-tutorial». [En línea]. Disponible en: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug1119-vivado-creating-packaging-ip-tutorial.pdf. [Accedido: 03-jul-2018].