Universidad Nacional del Sur

Tesis de Magister en Ciencias de la Computación

TLM Para la Verificación de Integración en SoC

Manuel Francisco Soto

Bahía Blanca Argentina

2015

Impreso en Argentina.
ISBN XXX-XXX-XXXX-XX-X
Octubre de 2015

Prefacio

Esta tesis se presenta como parte de los requisitos para optar al grado académico de Magister en Ciencias de la Computación, de la la Universidad Nacional del Sur y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otra. La misma contiene los resultados obtenidos en investigaciones realizadas en el ámbito del Departamento de Ingeniería Eléctrica y de Computadoras y el Departamento de Ciencias e Ingeniería de la Computación en el período comprendido entre Noviembre del 2012 y Noviembre del 2014, bajo la dirección del Dr. Pablo Rubén Fillottrani.

Bahía Blanca, 21 de Octubre de 2015.

Manuel F. Soto
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Universidad Nacional del Sur
Secretaría General de Posgrado y Educación Continua

La presente tesis ha sido aprobada el ..../..../..... , mereciendo la calificación de .....(.....)

Resumen

La verificación de los sistemas digitales se ha vuelto una etapa crucial en el proceso de desarrollar un *System on Chip* (SoC). El esfuerzo que se debe de hacer en esta etapa es sustancial respecto de otras. Debido a esto se ha optado por incrementar los niveles de abstracción al momento de diseñar/verificar un sistema digital. En esta tesis se estudiará uno de estos niveles, TLM o *Transaction Level Modelling*, se presentará su concepción, sus ventajas y desventajas, con el fin de poder diseñar un sistema de mediana complejidad atravesando varios niveles de abstracción. Se utilizarán métodos basados en simulación y métodos formales para verificar algunos de estos niveles.

La tesis se centra como se dijo anteriormente en TLM, dándole un enfoque principal a la transacción como unidad átomica de transferencia de datos en un diseño. En el primer capítulo se hace una introducción a la problemática/motivación, en el segundo capítulo se realiza un revelamiento del estado actual de la problemática, el tercero introduce una breve introducción a TLM y su implementación en *SystemC*, el cuarto presenta la metodología propuesta para afrontar la problemática. En el quinto capítulo se comenta cómo se realizó la implementación de la metodología. En el sexto capítulo se describen los resultados obtenidos. Mientras que el último capítulo se realizará una revisión de los resultados obtenidos, enumerándose los objetivos alcanzados y el trabajo a futuro en el área.

La utilización de una metodología Top-Down facilito la generación de las correspondientes abstracciones del sistema (niveles) a fin de comprender sus problemáticas particulares. Se abordo la verificación tanto de bloques propios como bloques desarrollados por terceros, apreciando las características de los distintos enfoques de verificación según el caso.

La inserción de los métodos formales como una herramienta adicional al flujo propuesto ha demostrado un aporte significativo al momento de realizar la verificación.

La utilización de distintos lenguajes de descripción de hardware evidenció las ventajas y desventajas de cada uno, análogamente se logró apreciar las ventajas y desventajas del entorno de verificación en comparación con entornos de verificación generados por otras metodologías ya establecidas. Por último, se apreció el beneficio de la simulación mixta *SystemC-Verilog* o

SystemC-VHDL, ganando una experiencia en el manejo de distintos lenguajes de HDL con el fin de generar conocimiento respecto de cuando debe de utilizarse o de que manera se debe utilizar los distintos lenguajes.

Agradecimientos

A Dr. Pablo R. Fillottrani, director, el cual me guío durante el desarrollo del trabajo y en la escritura de la Tesis, sin su punto de vista esto no hubiese sido posible. Al Dr. J. Agustín Rodriguez quien me acogió al inicio de la maestría, siendo referente en lo técnico y humano, haciendo varias veces de docente y compañero, explicando la problemática y ayudándome a evacuar cualquier tipo de duda.

A mis compañeros de maestría por su amistad y comprensión en el día a día, Juan Francesconi y Gabriel Pachiana. Al Dr. Martín L. Larrea quien se ha abocado al ayudarme en la redacción en las etapas iniciales de la Tesis, gracias por tu esfuerzo.

A los siguientes centro de investigación/ institutos nacionales:

- IIIE-Conicet
- INTI-CMNB
- DCIC-UNS

A mi familia y amigos, que me han acompañado en este camino, con su apoyo incondicional. Finalmente a mi compañera de vida Luciana, quien ha sabido comprender las inquietudes que este camino me han planteado y me ayudo a superarlas, el nombrarla acá es solo una pequeña forma de agradecimiento.

Índice general

Capítulo 1

Introducción

En este capítulo se presenta una descripción sobre el contexto, metodología, metas del trabajo y la estructura de la tesis.

1.1. Motivación

Con el rápido desarrollo de las herramientas de EDA[1], las tecnologías de fabricación de circuitos integrados (CI) y los nuevos métodos de síntesis automática, los circuitos integrados incorporan cada vez más procesadores, núcleos de propiedad intelectual (IP) y memorias, incrementando notoriamente su complejidad [1–4].

Se ha visto una tendencia hacia la reutilización tanto de diseños como de *testbenchs* o entornos de verificación [2,5–8]. Respecto de los diseños, se ha observado una disminución en el desarrollo de nueva lógica para éstos, en contraparte se ha visto un incremento en la reutilización de diseños *in-house* y en la compra de módulos de propiedad intelectual. Por su parte, se ha observado un gran incremento en la adquisición de *testbenchs* a terceros, con el fin de reestructurarlo y adaptarlo a las necesidades a fin de reducir tiempos en verificación.

La verificación es un área de relevancia en la academia y la industria [9,10]. Estudios recientes estiman que la mitad de todos los chips diseñados requieren fabricarse una o más veces debido a errores, un ejemplo de esto es lo ocurrido con los procesadores de *Intel* [11]. El porcentaje de tiempo de un proyecto invertido en realizar la verificación a alcanzado en el año 2012 al 56 %([3]), incrementándose de ésta manera la cantidad de personal calificado, en los grupos de trabajo, para realizar la verificación.

Más importante aún, estos estudios indican que el 74 % de estas fallas son errores funcionales [4]. En muchos casos la verificación consume más recursos, en términos de tiempo y trabajo, que

[1]Electronic Design Automation.

todas las demás etapas del flujo de diseño digital combinadas.

Por otro lado, se observa que uno de los últimos avances significativos en relación al nivel de abstracción en el que se diseña un sistema ha sido el mapeo a RTL[2] - *Gate Level*. En el año 1999, luego de que un grupo de investigación concretara el desarrollo de la tercera generación del CODEC H263 [12], haciendo una revisión del proceso que se llevo a cabo para la realización del mismo, se observó que era necesario el plantear un nuevo nivel de abstracción.

En respuesta a esta situación se eligió trabajar separando las componentes funcionales del sistema de los recursos de comunicación [13], con el fin de lograr una verificación/simulación más eficiente. El desarrollo de los nuevos testbench es realizado mediante el lenguaje *SystemC* [14], una extensión al lenguaje C++. Posteriormente con el apoyo de empresas como ARM y Cadence Design Systems, se promovió la OSCI [15], bajo el nombre de PV[3] y PVT[4].

Así surgió TLM, siendo ésta una propuesta para minimizar los tiempos de verificación en sistemas de gran escala. La tecnología de TLM ha tenido amplia adopción como se puede observar en [16–22] donde se muestran distintos desarrollos que tienen como propósito la generación de testbench, la integración/conexión de distintos bloques IP (RTL) y la mejora en los tiempos de simulación; todo esto para sistemas modelados a nivel de transacción.

El concepto de transacción permite representar en forma adecuada la interacción entre los componentes de hardware del *SoC*, además provee la interfaz necesaria para un desarrollo temprano del software que va a correr sobre el hardware del *SoC*. El realizar una descripción del sistema a nivel de transacciones provee una descripción ejecutable temprana, por lo tanto se puede realizar un análisis inicial de arquitectura, potencia, tiempos, etc; pudiendo plantear un plan de verificación más extenso y más abarcativo. Debido a que en la transacción se encuentran ocultos detalles tanto de información (datos) como de control (*handshake* de protocolos), se obtiene un beneficio al momento de simular el sistema, reduciendo el tiempo de simulación varios ordenes de magnitud [13].

1.2. Objetivos

El objetivo principal de este trabajo es el estudio, aplicación y desarrollo de tecnologías de verificación de sistemas digitales. Más precisamente, se trabajará con la metodología de verificación/diseño denominada "Transaction Level Modeling" (TLM) o modelado a nivel de transacciones, orientada a la verificación de "*Systems on Chips*" (*SoC*).

Como metas secundarias se pueden enunciar:

[2]Register Transfer Level.
[3]Programmer View
[4]Programmer View Timed

1. Estudiar el comportamiento de sistemas digitales, ya sean estos simples o complejos y comprender su flujo de diseño/verificación.

2. Estudiar el concepto de verificación, más precisamente, verificación funcional y ver su aplicación en distintos niveles de abstracción.

3. Estudiar las bases de TLM, tanto para diseño como para verificación y comprender sus ventajas y desventajas frente a otras opciones.

4. Generar una metodología de verificación válida para el flujo de diseño/verificación de un sistema digital, utilizando distintos niveles de abstracción.

1.3. Marco de Trabajo

Este trabajo fue desarrollado bajo el siguiente proyecto de investigación:

- FSTICS 001 "TEAC", PICT 2010 2657 y PAE 37079.

Mediante la siguiente beca de nivel inicial:

- Proyecto FS TICS No 001/10 - Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT) a través de FONARSEC. Beca de maestría tipo inicial. Periodo: Noviembre 2012 – Agosto 2014. El objetivo de la beca es el estudio y la aplicación de metodologías de verificación funcional de sistemas digitales. Más precisamente, en primer lugar el becario trabajará experimentado con tecnologías de verificación, lenguajes dedicados, simuladores y metodologías estandarizadas, aplicadas a la verificación de bloques funcionales y controladores de protocolos parte de un SOC; en segundo lugar, de acuerdo la experiencia adquirida trabajará en el desarrollo de software para verificación funcional. Se logró incorporar conceptos vinculados a la verificación de integración basada en modelado a nivel de transacciones (TLM). Incorporación de la tecnología de verificación basada en SystemC. Definición de una estructura de verificación basada en SystemC y análisis de un caso de estudio compuesto por bloques de ordenamiento, ethernet, transferencia de datos y memorias.

Utilizándose las siguientes herramientas del programa universitario:

- ModelSim de Mentor Graphics Corp. Simulador de eventos discreto, utilizado para *debug* y análisis de sistemas digitales. Provee soporte nativo para lenguajes de descripción de hardware como: *SystemC*, *VHDL*, *Verilog* y *SystemVerilog*.

- VCS (*Verilog Compiler Simulator*) de Synopsys, herramienta específicamente diseñada para realizar el análisis y debug de diseños. Provee soporte nativo para *SystemVerilog*, Verilog y VHDL.

1.4. Estructura de Tesis

La estructura de la tesis esta organizada de la siguiente manera:

- En el capítulo 2, se aborda la evolución tanto en el diseño como en la verificación del hardware, yendo desde las bases del cómo se hizo el transistor hasta llegar al estado actual del arte. En el capítulo también se destaca que actualmente la tecnología esta llegando a su cota en cuanto a la reducción del tamaño de la unidad básica, debido a limitaciones en el proceso de fabricación. Se enuncian dos posibles soluciones a éste problema.

- En el capítulo 3 se aborda conceptualmente lo que es TLM, sus distintos niveles de abstracción en dos grados de libertad (computacional y temporal), indicando cuál es el flujo sugerido para el desarrollo de sistemas complejos. Se realiza una comparativa entre los distintos niveles de abstracción, sus ventajas y desventajas.

- En el capítulo 4, se propone una nueva metodología de diseño y verificación la cual contempla distintos niveles de abstracción, logrando así una depuración de errores que pueden aparecer a la hora de diseñar un sistema digital complejo. Cada una de las etapas de la metodología propuesta tiene una retroalimentación la cual es descripta en el capítulo, indicando mejoras respecto de otras metodologías conocidas para el flujo de diseño/verificación de un sistema digital.

- En el capítulo 5 se indica cómo fue el desarrollo y la verificación de los componentes del sistema, tanto a nivel de TLM, un nivel intermedio utilizando *SystemC* (a veces para diseño otras para verificar) y finalmente el nivel RTL en el cual cada uno de los componentes es descripto en *VHDL* o en *Verilog*. Se indica análogamente cómo se realizó la verificación en los distintos niveles de abstracción, la utilización de herramientas formales, simulaciones y ejemplos que convalidan estas técnicas.

- En el capítulo 6 se enuncian los resultados obtenidos, demostrándose los resultados obtenidos por el flujo propuesto.

- En el capítulo 7 se realiza un análisis de los resultados obtenidos, se expone cuales fueron los puntos fuertes y cuales los débiles de la metodología propuesta y por último se plantea el trabajo a futuro.

- En los apéndices A, B y C, se encuentra parte del código generado en la Tesis y las publicaciones generadas, con el fin de que sirva como material de consulta.

1.5. Contribución

El resultado obtenido del trabajo se puede apreciar en la aplicación de la metodología propuesta, en donde se evidenció las ventajas y la desventajas de la misma. Se pudo llevar a cabo la integración de componentes generados por terceros y propios, enfocando la verificación de los primeros en el cumplimiento de las especificaciones mientras que, en los segundos se realizó una verificación funcional más exhaustiva. La utilización de métodos formales aportó resultados logrando minimizar los tiempos al momento de realizar la verificación funcional del sistema. Se ganó experiencia en la utilización de distintos lenguajes de descripción de hardware ya que se experimentó con *SystemC*, *VHDL* y *Verilog*. Lo realizado en la Tesis se encuentra plasmado en la publicación *SystemC/TLM flow for SoC Design and Verification*, el mismo se encuentra en etapa de evaluación en la décima edición de la *Escuela Argentina de Micro-Nanoelectrónica tecnología y Aplicaciones* (EAMTA 20015).

Capítulo 2

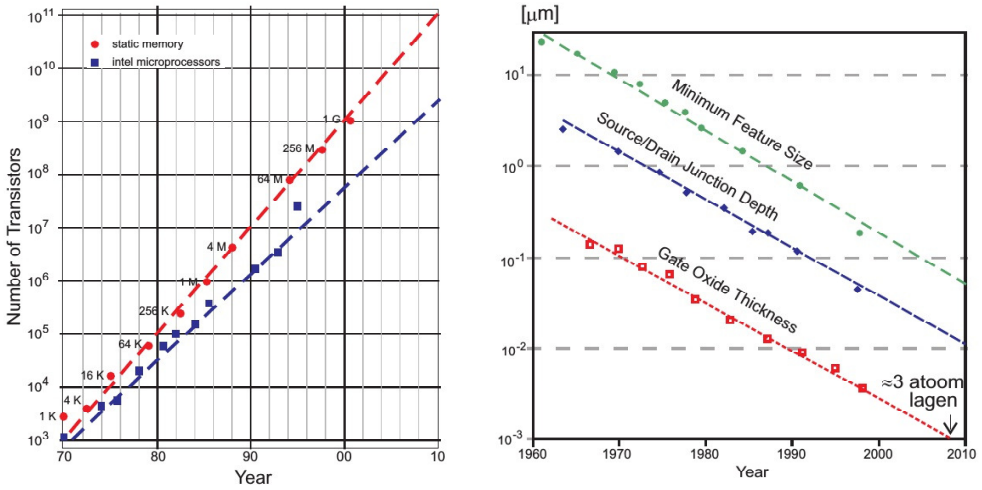
Marco Teórico

En este capítulo se indicará cómo fue evolucionando el desarrollo y la verificación de sistemas digitales, desde sus principios en los años 50 cuando el primer transistor fue concebido, hasta el día de hoy.

2.1. Diseño de hardware

En las décadas recientes, el avance en la tecnología respecto de los materiales de los semiconductores y el proceso de fabricación de éstos ha permitido diseñar circuitos de millones de compuertas o *gates*, en un único chip. El reflejo de este avance se observa en el cambio de la unidad básica de diseño [23], en el año 1947 *William Shockley* y sus compañeros de *Bell Laboratories* concibieron el transistor, logrando así dar inicio a la electrónica de estado sólido. Para la creación del transistor se basaron en el concepto de manejo del flujo de corriente a través de un sólido, como lo es el silicio, agregando de forma adecuada un cierto grado de impurezas [24]. El hecho de que se haya podido generar un elemento que tuviese mejores capacidades para la misma tarea que hasta entonces realizaban las válvulas de vacío, sentó las bases para lo que era el inicio de la industria de semiconductores.

A finales de los años 50, más precisamente en el año 1959, investigadores de *Fairchild Semiconductor* desarrollaron el primer transistor plano y posteriormente el primer circuito integrado plano [25], demostrando los beneficios de la producción en masa de circuitos integrados planos. En abril de 1965, *Gordon Moore* publicó un artículo [26] el cual predecía en forma especulativa cuál iba a ser la evolución de la industria de semiconductores en los próximos diez años, dando a conocer la ley de *Moore* que indica que, la densidad de transistores por unidad de área se multiplicara por un factor de dos cada dos años. El nivel de integración de hardware en estos años era de alrededor de 100 transistores por circuito integrado, esta densidad es conocida por



(a) Números de Transistores. (b) tecnología utilizada.

Figura 2.1: Evolución del hardware [2].

sus siglas en ingles como *SSI* (Small Scale Integration).

Estando ya el transistor instalado como unidad básica de diseño en los años 70, se vio una tendencia hacia las compuertas lógicas en los años 80 [27]; el nivel de integración ascendió a más de 10000 transistores por circuito integrado, este nivel de densidad de transistores se denomina *VLSI* por sus siglas en ingles (*Very Large Scale Integration*). La utilización o descripción de circuitos a nivel RTL logró un cambio radical en la industria en los 90s. Este hecho logró elevar el nivel de abstracción necesario para describir circuitos, siendo la unidad básica el registro.

Finalmente en la ultima década se ha logrado incrementar el nivel de abstracción, de esta forma surgió el *SoC* [28], y su unidad básica de diseño el bloque IP. SoC es el concepto de concebir e integra distintos componentes electrónicos en un único circuito integrado con el fin de obtener un sistema electrónica. Como consecuencia del avance tecnológico, el cual permite aumentar la cantidad de elementos por unidad de área, se han logrado desarrollar sistemas más complejos [1].

Adicionalmente a lo visto en la Figura 3.6a, el informe del grupo *Wilson Research* [3], indica que el 33.3 % de los sistemas de hoy en día contienen más de 20 millones de compuertas lógicas, esto implica un incremento en la complejidad de tareas tales como el conexionado de distintos componentes del sistema, y que la estimación de *power* o *clock gating* sea dificultosa. También es necesario considerar la comunicación entre dispositivos, esto quiere decir la utilización de distintos protocolos, los cuales dependen de las características (velocidad de transferencia por ejemplo) físicas de los dispositivos; otorgándonos de esta forma opciones de protocolo como AHB, APB, AXI, CAN y otros.

El aumento de la complejidad mencionado anteriormente se observó en el diseño de uno de los primeros *SoC* [28], este se encargaba de implementar un decodificador de *Moving Picture Experts Group Phase 2* o*MPEG2*. Se demoró dos años en desarrollarse, su utilizarón ocho meses en el diseño y la verificación inicial del sistema, mientras que otros diez meses fueron necesarios para revisiones. La complejidad de este *SoC* era tal que, a momentos de realizar una simulación, en la cual se viera procesado un único *frame* de vídeo, ésta demandaba 10 horas en finalizar. Considerando además que un *stream* de vídeo contiene 30 frames por segundo, el nivel de tiempo necesario para realizar el *debug* del sistema era alarmante. Una vez finalizada la implementación del sistema, se examinaron los *bugs* encontrados, obteniéndose la siguiente clasificación de bugs [29]:

- *Problemas de especificaciones*: definiciones vagas, lo cual ocasionó un desentendimiento de condiciones necesarias a cumplir y generó mal entendidos entre los grupos de trabajo.
- *Problemas de implementación*: *performance* baja, problemas de interfaz, consumo considerable de potencia, etc.
- *Problemas de verificación*: Lentitud en el software de simulación y problemas de interfaz entre el hardware y el software que éste debía correr.

Análogamente, otra situación que tomo relevancia debido a un error en el diseño no detectado en las primeras instancias de éste, es lo ocurrido con los procesadores *Pentium* en el año 1994 [11], en éstos la unidad de punto flotante tenia un error que no fue detectado por el grupo de verificación, haciendo que *Intel* tuviese que gastar alrededor de $420 millones de dolares para solucionar el error. Gran parte de este dinero fue destinado a la contratación de cientos de personas dedicadas exclusivamente a responder preguntas en los foros con el fin de informar sobre el error.

El hecho de que la complejidad de los sistemas actuales haya incrementado en forma considerable puede interpretarse como lo ocurrido en la historia de la industria del software, en donde fue necesario realizar un cambio de paradigma para dar soporte a las problemática [30]. Análogamente en la industria del hardware, el realizar la migración a través de los niveles de abstracción añade complejidad a la problemática de diseñar. Con el fin de manejarla, nuevas técnicas, conceptos e ideas de distintos campos de la ciencia emergen como solución y se observa una migración hacia un nuevo flujo de diseño [31, 32].

Lenguajes como *SystemC* ([33]) han surgido, siendo ampliamente adoptado por la industria para realizar un primer modelado y diseño del sistema a nivel de arquitectura. *SystemC* es usado para realizar simulaciones a distintos niveles de abstracción, como RTL o señal mixta; y desde el 2007 otorga soporte para transacciones, incorporando una biblioteca de *Transaction Level*

Modelling (TLM) [34]. En los últimos años la gente que trabaja en el consorcio *Open SystemC Initiative*, ha desarrollado la *SystemC Verification Library* o *SCV* por sus siglas en ingles, dando soporte para realizar una verificación inicial del sistema.

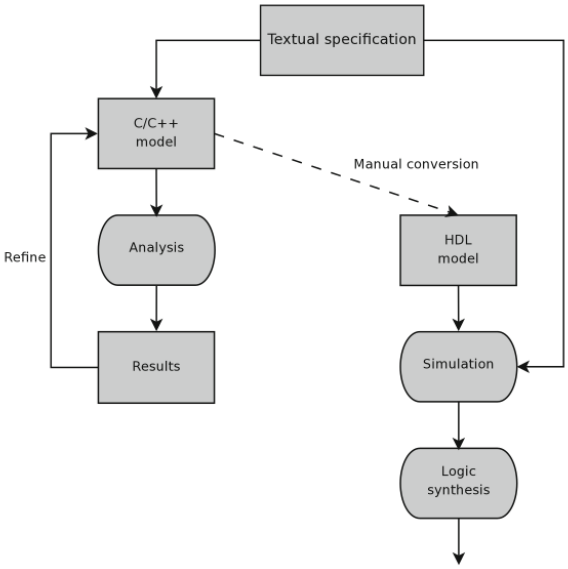


Figura 2.2: Flujo de diseño clasico en *SystemC* [35].

2.1.1. Niveles de Abstracción

Varias razones han llevado a extender el flujo de diseño clásico, observado en la Figura 2.3, hacia otros niveles de abstracción [13]. Estas son:

- *Crecimiento de la complejidad:* El incremento del número de componentes de un sistema y su propia complejidad hacen que características como confiabilidad y *performance* sean difíciles de alcanzar en valores razonables. Por el lado del software, hoy en día los sistemas soportan cada vez más y más aplicaciones.

- *Time to Market*: El tiempo requerido para ir de una idea a su concepción física con el fin de venta debe de minimizarse, de forma tal de que se siga teniendo competitividad en una determinada área del mercado.

- *Incremento en los costos:* Se requiere un grupo humano de trabajo de gran tamaño para desarrollar un sistema que sea competitivo en el mercado hoy en día y adicionalmente a esto, los costos de fabricar se incrementan cada vez que la tecnología se reduce. Por

último, las herramientas para desarrollar el sistema tienen licencias del orden de millones de dólares anuales.

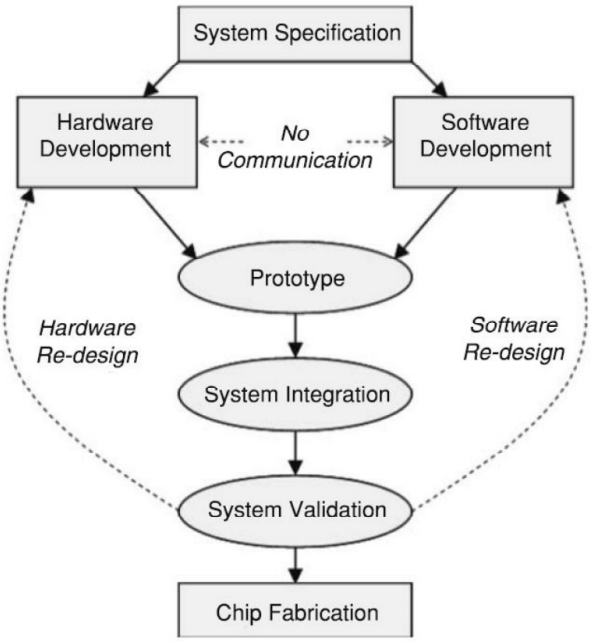


Figura 2.3: Flujo de diseño clásico [13].

Debido a esto, se han propuestos nuevos niveles de abstracción como los que se observan en la Figura 2.4, los cuales permiten desarrollar modelos iniciales mediante software, describiendo algunas características del sistema con el fin de aliviar la carga en el proceso de desarrollo en general.

Es sabido que el diseño a nivel de sistema tiene un gran potencial para realizar de manera correcta un análisis de arquitectura y verificación funcional [36]. Este es un problema crucial en el desarrollo de sistemas complejos hoy en día; el realizar el análisis sobre el comportamiento esperado para un determinado SoC en tiempo real nos da una métrica de cuan importante son los parámetros. El análisis y simulaciones a distintos niveles de abstracción emerge como la correcta aproximación para manejar las dificultades en el desarrollo del sistema, tanto para hardware como para el desarrollo del software.

Finalmente, en los últimos años, el desarrollo de hardware esta llegando a sus límites en cuanto a nivel de compuertas lógicas por unidad de área; esto es debido a su proceso de fabricación, particularmente el proceso de fotolitografía actual se encuentra en la cornisa de sus limitaciones físicas [4]. Por lo tanto, se esta planteando el contexto de dos líneas ideológicas respecto del futuro en la industria de los semiconductores y en especial del diseño de hardwa-

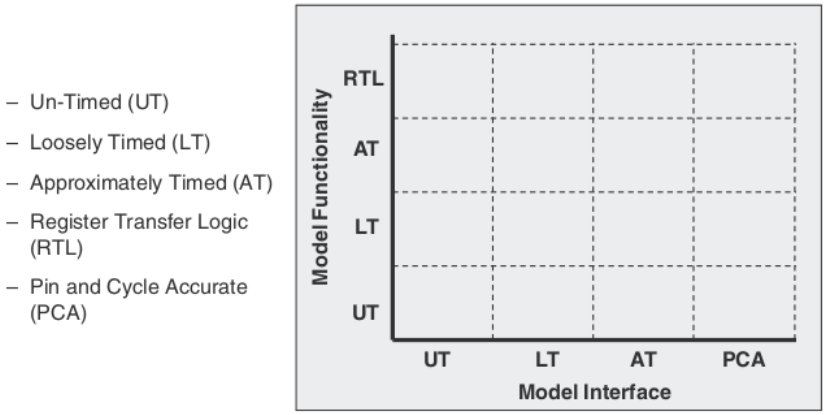


Figura 2.4: Diferentes niveles de abstracción [13].

re. Estas lineas son *More than Moore* y *More Moore* [37, 38]. La primera, *More than Moore* menciona que varias aplicaciones hoy en día como radio frecuencia (RF), *power management*, componentes pasivos, biochips, sensores, *microelectromechanical Systems* (MEMS) tienen igual relevancia en la industria de los semiconductores. La integración de estos componentes a los ya existente dispositivos CMOS, generan un adicional en el valor agregado del producto final. La segunda tendencia, *More Moore* es una continuación en la linea actual de desarrollo tecnológico, intentando aumentar la densidad de transistores por unidad de área.

2.2. Verificación

La verificación es el proceso utilizado para demostrar que una idea o concepción de un diseño se corresponde con su implementación [39]. En el área del *hardware*, la misma se debe realizar en cada uno de los niveles de abstracción que el flujo de diseño de éste contempla; por lo tanto se puede apreciar distintos tipos de verificación según el contexto: verificación formal, verificación funcional, verificación de *timing*, *power*, etc.

La verificación funcional a comparación con las otras, es el cuello de botella del diseño de hardware hoy en día [4, 40], apreciándose el *gap* que existe entre el avance de la tecnología, el diseño y la verificación [41–44] (Figura 2.5).

La verificación funcional, se puede llevar a cabo utilizando métodos estáticos/formales, dinámicos/simulaciones, o métodos híbridos. Los métodos formales proveen una demostración formal (matemática) sobre la correspondencia entre la idea y su implementación, estos son sólo aplicables a implementaciones acotadas o a un conjunto reducido de funcionalidades, aunque actualmente están ganando protagonismo. Siendo utilizada en cerca de una tercera parte de

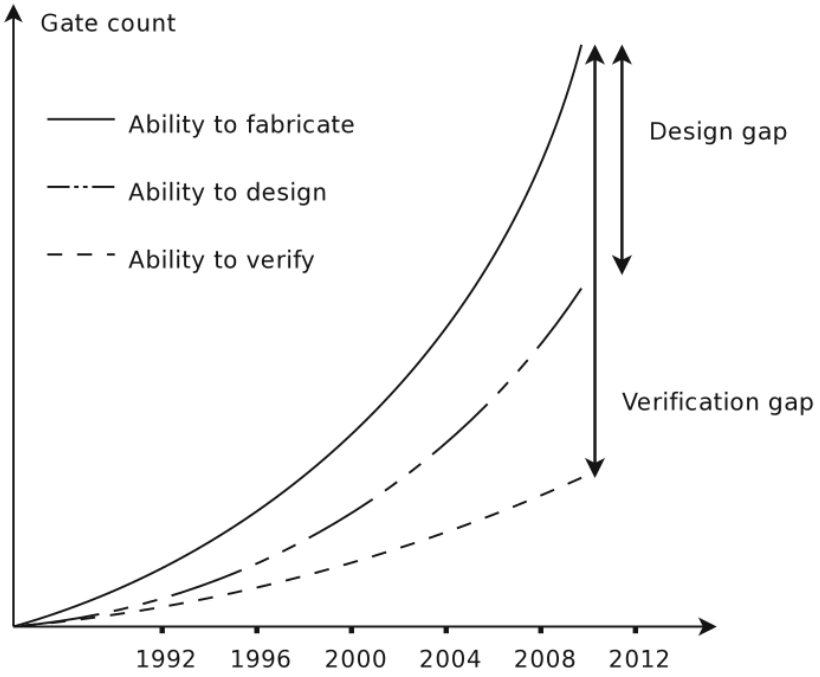


Figura 2.5: Gap de Diseño y Verificación [45].

todos los casos de verificación. Las técnicas más utilizadas las de *Model Checking* [46], esta es utilizada para la verificación de correctitud sobre la descripción de un sistema de estados finitos, busca resolver el problema que: dado un modelo de un sistema y una especificación del mismo se chequea exhaustivamente y de forma automática si ambos se corresponden.

Los métodos basados en simulación o dinámicos, pueden ser aplicados inherentemente de la complejidad del diseño. Aunque estos últimos son efectivos para la detección de errores, el que no sean detectados no asegura la ausencia de los éstos. Para lograr un resultado más refinado en los últimos años (cinco años) se ha visto un incremento del 26 % en la utilización de métricas de cobertura y aserciones [3].

Debido a lo mencionado anteriormente y el aumento en la complejidad de los diseños [4], la verificación también se vio afectada por estos cambios, los efectos que tuvieron los cambios de paradigma en la verificación son similares al problema que se encuentra en un grafo cuando se lo desea recorrer, la explosión de estados, mientras más elementos hay para verificar, el modo tradicional decantaba en una explosión de estados [47–49].

Según estudios, el porcentaje de tiempo necesario para realizar la verificación de un sistema actual insume el 70 % [50] del tiempo requerido en el flujo de diseño de éste, y éste número (70 %) se incrementará en los siguientes años de manera exponencial [51].

Por lo tanto, planteado el contexto actual en el cual se desarrollan los sistemas digitales, viendo que los mismos incrementan su cantidad de componentes ya sean: número de *cores* CPU, sub sistemas multimedia y bloques IP de comunicación; el llevar a cabo interconexión de éstos mediante *crossbars* o *NoCs* se ha vuelto un trabajo relevante en los equipos de verificación.

La verificación de las interconexiones del *SoC* es un desafío que va más allá del correcto comportamiento de las transacciones internas, *power management*, correcto manejo de espacios de direcciones y *bus protocol transaction*.

En la actualidad las herramientas basadas en métodos dinámicos son el principal medio de la industria del desarrollo de hardware para efectuar la verificación [44].

Finalmente los métodos híbridos, combinan nociones de los métodos dinámicos y de los métodos formales a fin de superar los limites impuestos por los métodos formales (*constraints*) y tratando de llegar a los limites de los métodos dinámicos [52].

Dada las condiciones actuales del mercado, nuevos lenguajes de Diseño/Verificación han surgido, estos se basan en conceptos de la Programación Orientada a Objetos o POO; dichos lenguajes se pueden clasificar según [2] en tres grandes bloques:

- *Existentes*: La utilización de los lenguajes clásicos ya sean como VHDL o Verilog, extendiéndolos hacia *SystemVerilog* o utilizando este último en particular.
- *Adaptados*: Adaptar lenguajes ya existentes como C/C++ [53], Java [54], UML, etc.
- *Nuevos Lenguajes*: Se crean nuevos lenguajes a fin de dar soporte a las necesidades actuales, *Rosetta* [55] es un ejemplo de esto.

Capítulo 3

TLM & SystemC

En este capítulo se definirá el concepto de *Transaction Level Modelling* o *TLM*, viendo los distintos niveles de abstracción del mismo y finalmente se comentará el soporte que da *SystemC* a éste. El capítulo se centra especialmente en el concepto de la transacción y la separación de sus partes: comunicacionales y computacionales.

3.1. TLM

3.1.1. TLM breve historia

A finales de los noventas, varias empresas comenzaron a desarrollar sus propios modelos de diseño, mientras que los institutos de investigación y las *start-ups* de EDA proponían una variedad de lenguajes de modelado. Entre estos lenguajes, algunos fueron desarrollados desde cero, otros realizaban la extensión de algún lenguaje ya existente, principalmente esto ocurría con lenguajes de orientados a objetos como C++ o Java. Ejemplo de esto son: SpecC, CowareC, etc. Otros lenguajes propuestos son extensiones de los lenguajes de descripción de hardware como VHDL o Verilog. Un ejemplo de esto es Superlog.

Inicialmente, se desarrollaban varias clases de modelos utilizando distintos lenguajes; estos modelos se desarrollaban a distintos niveles de abstracción dependiendo de las necesidades. Por lo general, inicialmente se requería un modelo en C o C++ cercano al *cycle-accurate*, debido a que se creía que esta era la manera correcta de generar simulaciones que corrieran por lo menos un orden de magnitud más rápido que el RTL. Rápidamente se observo que estos modelos tenían ciertos problemas.

1. Se realizaba un esfuerzo cercano al de codificar el sistema en RTL sintetizable, la única diferencia era que no se consideraban los *constraints* de la síntesis. Esto ocurría ya

que el RTL era todavía el modelo de referencia de todos los sistemas, adicionalmente las herramientas de síntesis todavía no seguían el avance en las tecnologías.

2. La velocidad de las simulaciones que se obtenían de lo codificado en el item 1 no eran las esperadas.

3. Se utilizaban varios lenguajes de especificación u optimizaciones de modelado para mejorar la velocidad de la simulación, igualmente este esfuerzo realizado estaba inherentemente relacionado a simulados y a un grupo de trabajo.

4. Las últimas actualizaciones realizadas en el RTL antes de llevar a cabo el *tape-out*, no se podían ver reflejadas en el modelo de C++ o C. Debido a esto era sumamente difícil reutilizar los modelos de C o C++ como punto inicial de referencia en nuevos proyectos.

Por todas estas razones, se a decidido elevar y homogeneizar el nivel de abstracción, la abstracción es una técnica sumamente poderosa, permitiendo un rápido modelado e implementación del sistema, con el fin de que éste sirviera tanto como modelo de referencia para los desarrolladores de hardware y como plataforma de testo para los desarrolladores de software. Adicionalmente, permitirán el realizar distintos tipos de análisis sobre la arquitectural del sistema [13, 32].

3.1.2. Definición de TLM

Debido a la creciente complejidad en los sistemas digitales, y las presiones inducidas por el *Time to Market*, la industria ha optado por utilizar distintos niveles de abstracción al momento de desarrollar proyectos, con el fin de aumentar la productividad y reducir tiempos.

Uno de los niveles de abstracción utilizados es el de *Transaction Level Modelling* o *TLM* [56, 57]. El actor principal a nivel de *TLM* es la transacción, una transacción se define como una secuencia que describe operaciones de control y transferencia de datos entre dos o más componentes del sistema, ocultando detalles de implementación de los medios (canales) por la que la transacción se lleva a cabo. La idea principal consiste en separar la transacción en una componente comunicacional y una componente computacional se logra así independizarse de situaciones como *handshake* de protocolos, optimizaciones de tarea, etc.

Dado que durante el desarrollo de un sistema o un *SoC*, los grupos de trabajo generan distintos modelos, dependiendo de las necesidades, se ha buscado unificar los aspectos más generales de éstos, para minimizar así las ambigüedades. En la Figura 3.1 se observan los distintos niveles de abstracción de *TLM*, el eje X denota el grado de computación de los modelos, mientras que el eje Y representa la comunicación de éstos. En cada eje se puede observar tres niveles o tres grados de precisión, estos son: *Un-timed*, *Approximated-timed* y *Cycle-timed*. La combinación de

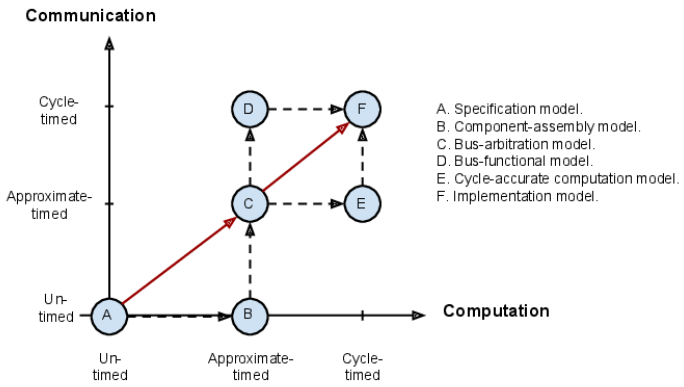


Figura 3.1: Niveles de abstracción [57].

los distintos grados de libertad da como resultado diferentes modelos de un sistema, los cuales son explicados a continuación:

- (*Un-Timed*): Representa un modelo puramente funcional, a este nivel no existen detalles propios de los dispositivos. En la Figura 3.2, se aprecia un ejemplo en donde se observan procesos, variables y la relación de estos.

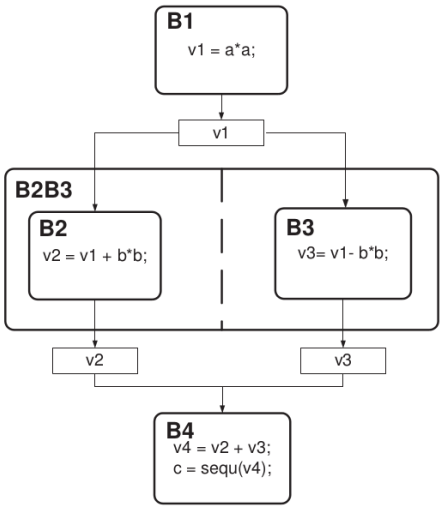


Figura 3.2: Ejemplo, *Un-Timed Model* [57].

- (*Approximate-timed*): A este nivel, se encuentran detalles de implementación a nivel de sistema, éstos pueden ser distintas arquitecturas, mapeo entre las funciones a realizar y los elementos que la realizan a nivel arquitectónico. El tiempo de ejecución aún dista del

tiempo de ejecución de un modelo RTL *cycle-accurate*. En la Figura 3.3 un ejemplo es expuesto, se aprecia como cada componente (PE) tiene su canal de comunicación, estos confluyen a un canal abstracto el cual es controlado por un arbitro.

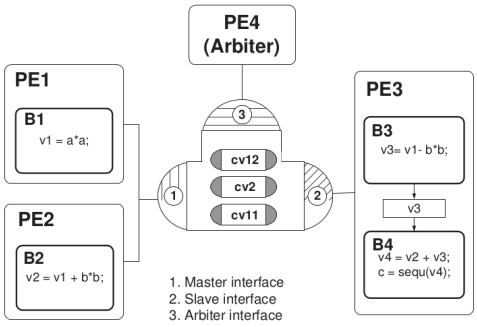


Figura 3.3: Ejemplo, *Approximated-Timed Model* [57].

- (*Cycle-timed*): Contiene detalles tanto del sistema como del *RTL* o *ISS*, éste nivel a diferencia de los anteriores, contempla una estimación temporal de nivel *cycle-accurate*. En la Figura 3.4 se observa un ejemplo, en el cual hay dos procesos que son CPU y los otros dos son hardware dedicado, se puede apreciar porciones de microcódigo y comunicación entre procesos a nivel de señales.

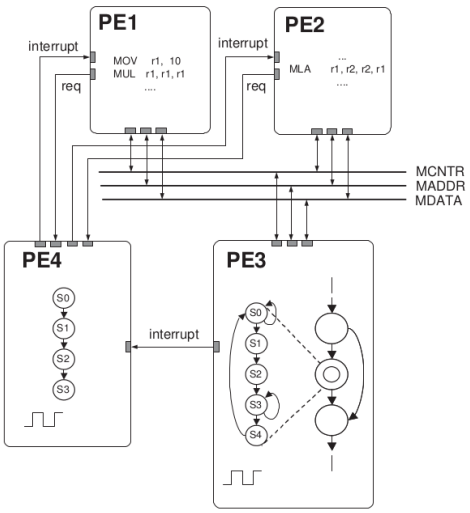


Figura 3.4: Ejemplo, *Cycle-Timed Model* [57].

En base a lo anteriormente nombrado y lo observado en la Figura 3.1, se procederá a describir cada uno de los modelos que se encuentran en la misma:

- *Specification Model*: Este modelo describe las funcionalidades del sistema, sin detalle de implementación alguno. Es similar al *Specification Model* encontrado en [31] y el *Untimed Model* que se puede encontrar en [32]. En este nivel de abstracción, la transferencia de datos es modelada entre procesos a través de variables compartidas sin la utilización de canales, lo cual facilita su futura implementación.

- *Component-assembly Model*: En este nivel de abstracción la entidad principal son los *Processing Elements* (*PE*), los mismos representan la ejecución concurrente. Un *PE* puede ser un elemento *custom* de hardware, un procesador de propósitos generales, un *DSP* o un *IP*. La comunicación entre los *PE* es mediante pasaje de mensaje, de esta forma se da soporte a la transferencia de datos o sincronización sin denotar detalles del protocolo de comunicación o de buses. Este nivel de abstracción se corresponde con el valor *un-timed* del eje de comunicación, mientras que la computación del mismo tiene un nivel de precisión más alto, *Approximate-timed*. El tiempo de computación es calculado a nivel de sistema, utilizando clausulas *waits*. Este nivel es similar al modelo arquitectural definido en [31] y pertenece a los modelos funcionales definidos en [32]. En comparación con el modelo anterior, denota de forma explicita dónde se encuentran los *PEs* en la arquitectura y el mapeo de las funcionalidades a éstos.

- *Bus-Arbitration Model*: Este nivel de abstracción utiliza *buses* para la comunicación entre *PEs*. La transferencia de datos entre elementos computacionales es a través de canales mediante pasaje de mensaje, mientras que el protocolo del bus se puede modelar como bloqueante o no bloqueante. El bus tiene un tiempo aproximado de ejecución, el cual es especificado en el canal mediante sentencias *waits*. Dado que varios canales pueden estar agrupados en un único bus abstracto, dos parámetros se agregan al bus: la prioridad y la dirección del canal. La dirección se encarga de distinguir las diferentes llamadas a distintos *PEs*, mientras que la prioridad denota el orden de acceso al bus. Además, un árbitro del bus es insertado dentro de la arquitectura del sistema, como un nuevo *PE*, para resolver los conflictos. Por lo tanto el entorno en este nivel de abstracción queda compuesta por: *PEs Master*, *PEs Slave*, un *PE árbitro* dentro del canal y un canal abstracto.

- *Bus-Functional Model*: Nivel de abstracción con un grado de comunicación *Cycle-Timed* y de computación *Aproximate-Timed*. A este nivel se conciben dos tipos de bus: uno *Time-Accurate* y otro *Cycle-Accurate*. El primero especifica las restricciones comunicacionales, respecto del protocolo que se desea utilizar. Mientras que el segundo especifica restricciones

temporales en función del *clock* del *Master* del bus. La tarea de refinar el primer modelo hacia el segundo se denomina refinamiento de protocolo, éste es similar a la inserción de protocolos observadas en [31].

El pasaje de mensaje como método de comunicación, es remplazado por el protocolo del canal (similar al protocolo del canal definido en [31]. El protocolo del canal es *Cycle-Accurate* y *Pin-Accurate*, por lo tanto los cables o *signals* del bus son representados por: señales internas o variables. La transferencia de datos sigue siendo *Cycle-Timed*. La interfase del bus provee funciones para soportar las transacciones adecuadas necesarias para respetar el protocolo.

- *Cycle-accurate computation model*: Nivel de abstracción con computación *Cycle-Timed* y comunicación *Approximate-Timed*. Este modelo puede ser generado desde un *Bus-Functional Model*; los *PEs* ya tienen un nivel de implementación de *Pin-Accurate* y son ejecutados a *Cycle-Accurate*. Las componentes custom de hardware son modeladas ya a nivel RTL, mientras que los procesadores de propósito general o DSP son modelados en base a sus set de instrucciones. Para realizar la comunicación entre los *PEs* y capas de abstracción más altas en el modelo, es necesario desarrollar los respectivos wrappers que realicen la transformación de un modelo a otro. Similar al modelo anterior, no es necesario que todos los *PEs* tengas un nivel de refinamiento homogéneo, en el mismo pueden existir distintos *PEs* con distintos niveles de refinamiento siempre que haya presencia de los respectivos wrappers para garantizar la comunicación.

- *Implementation Model*: Dicho modelo tiene la comunicación y la computación al nivel *Cycle-Accurate*. Los componentes son definidos a nivel RTL o de *Intruction Set Simulator* (ISS). El mismo puede ser obtenido de modelos anteriores como: *Bus-Functional Model* o *Cycle-Accurate Computation Model*. La implementación de éste modelo es similar a las descriptas en [31] y [32] bajo los nombres de *Implementation Model* y *Resgister Transfer Model* respectivamente.

En la Figura 3.1 se puede apreciar que una flecha gruesa atraviesa los distintos niveles o modelos de abstracción, dicha flecha describe un flujo natural de diseño/verificación, el cual es aceptable, ya que abarca desde las especificaciones funcionales del sistema hasta un modelo *cycle-accurate* del mismo.

TLM ha demostrado ser la metodología adecuado para lidiar con los problemas clásicos de un *SoC* [13, 58, 59]. A través del diseño de *SoC*, éste (el modelo en *TLM*) sirve como única referencia para los diferentes equipos de trabajo, especialmente sirve como una referencia para las siguientes tres actividades implicadas en el desarrollo de un *SoC* [13], éstas son:

- Desarrollo inicial del Software que va a correr sobre el *SoC*.
- Análisis de Arquitectura.
- Verificación Funcional del Sistema.

3.2. SystemC

En esta sección se dará una breve descripción de *SystemC*, sus principales componentes y su comportamiento en simulación.

3.2.1. SystemC como HDL

SystemC es uno de los lenguajes de especificación y modelado de sistemas mayormente adoptados hoy en día [60]. El mismo ha evolucionado desde sus inicios cuando era un simulador de eventos discreto restrictivo hasta hoy en día que es un lenguaje de modelado de sistemas, en la Tabla 3.1 se aprecia su evolución, volviéndose en el año 2005 *standard*de IEEE [60].

Fecha	Versión	Observaciones
Septiembre 1999	0.9	Primera versión *cycle-based*.
Febrero 2000	0.91	Solución de algunos bugs.
Marzo 2000	1.0	Primera versión estable.
Octubre 2000	1.0.1	Solución de algunos bugs.
Febrero 2001	1.2	Se añadieron característica.
Agosto 2002	2.0	Se añadieron al core canales y eventos.
Abril 2003	2.0.1	Solución de algunos bugs.
Junio 2003	2.0.1	*Language Reference Manual* (LRM) en revisión.
Primavera 2004	2.1	LRM aceptado para IEEE Standard.
Diciembre 2005	2.1v1	IEEE 1666-2005 ratificado.
Julio 2006	2.2	Solución de bugs.
Marzo 2007	2.2	-
Julio 2012	2.3	TLM nativo.
Abril 2014	2.3.1	-

Tabla 3.1: Linea de tiempo de SystemC.

El *core* de *SystemC* se aprecia en la Figura 3.5, viendo como el mismo esta constituido sobre C++ y su estándar, logrando de esta forma una rápida adhesión por parte de desarrolladores.

Así mismo se observa que el *core* provee de un kernel de simulación siendo este un análogo al simulador de eventos que contienen simuladores como *ModelSim* [61] o *VCS* [62]. También se puede observar distintos componentes, tales como: canales, *threads*, diferentes tipos de datos, etc; siendo esto una clara muestra de que *SystemC* ha sido concebido para diseñar y modelar. En la capa de la Figura 3.5 se puede apreciar la biblioteca de verificación, la cual provee de funciones para randomizar, agregar *constraints*, etc; haciendo que la verificación sea más eficaz. También se puede apreciar que el *core* puede interactuar tanto con bibliotecas propias del usuario como con diferentes IPs.

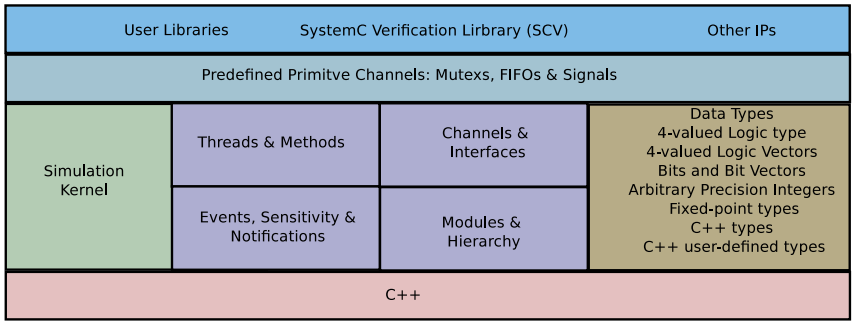
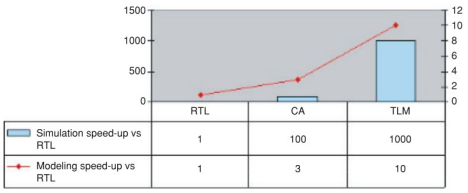


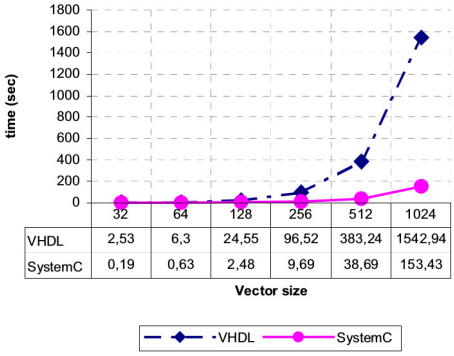
Figura 3.5: SystemC Core.

Dado que ésta tecnología provee a los ingenieros de un entorno más ameno, en comparación con el clásico entorno RTL para el diseño de sistemas, se puede obtener una reducción en los tiempos dedicados a la etapa de diseño, permitiendo una posible exploración del sistema en busca de fallas en un tiempo menor [63].

Debido a que *SystemC* actualmente esta basado en C++, el mismo comparte todas las característica del lenguaje, adicionalmente introduce nuevos tipos de datos, tales como: tipo de dato de *fixed point* y tipo de dato *logic*; siendo éstos orientados al desarrollo de hardware.



(a) Redución de tiempos [13]



(b) Caso de estudio [64].

Las características principales del lenguaje en lo referido al diseño son:

- *Diseño Estructural*: El modulo es el bloque básico de construcción de sistemas en un diseño jerárquico, análogamente los canales son la entidad básica en lo referido a la comunicación entre módulos. En general un diseño tiene un conjunto de módulos comunicándose entre ellos mediante canales proporcionados por el sistema como: *sc_fifo*, *sc_signal*, *sc_mutex*, *etc*; o canales definidos por el usuario. La entrada/salida de recursos a los módulos, se realiza mediante puertos. Una de las principales ventajas provistas por *SystemC*, es que los puertos, no se conectan directo a los canales, sino que éstos hablan con los canales mediante una interfaz. Las interfaz puede variar sin que haya necesidad de variar la estructura de comunicación, canales, otorgando de esta forma la posibilidad de que el diseñador experimente el comportamiento del diseño en distintas situaciones como podrían ser distintas versiones del protocolo, distintos protocolos, etc.

- *Diseño Comportamental*: La funcionalidad de cada bloque se define mediante procesos, éstos pueden implementar la parte computacional de un bloque, análogo a una función en C++, o puede ser un proceso sensible al Kernel de *SystemC* [65]. Para éstos últimos, *SystemC* provee dos macros: SC_METHOD y SC_THREAD. El primero simula el comportamiento de una tarea atómica (similar a los *commit* atómicos de Base de Datos) sin elementos de sincronización; mientras que el segundo provee una característica adicional, éste es sensible a sentencias *wait*.

En la Figura 3.7 se puede apreciar el conjunto de componentes de *SystemC* y sus conexiones, un ejemplo concreto se puede apreciar en el Código 7 del Apéndice.

3.2.2. SystemC HVL

SystemC también puede ser utilizado como un lenguaje de verificación de hardware o *HVL* (*Hardware Verificaction Language*) [32, 66], debido a esto el realizar la verificación funcional de un sistema con *SystemC* reduciría los costos que ésta genera en el flujo de diseño. Esta tarea compete el probar, con cierto grado de eficacia, que dado un diseño este se corresponde con las especificaciones previas. En particular, la verificación funcional busca el analizar ciertas características del diseño con *test* sencillos; esto actualmente involucra comprender el funcionamiento del sistema y el poder definir secuencias de estímulos que estimulen los aspectos que nos parecen relevantes en el DUV (*Design Under Verification*).

A principios del 2014, se realizó el último *release* de la biblioteca de verificación para *SystemC* o como sus siglas en ingles la denominan, *SCV SystemC Verification Library* [67]. Ésta provee mecanismo para realizar la verificación de distintas maneras: basada en transacciones, basada

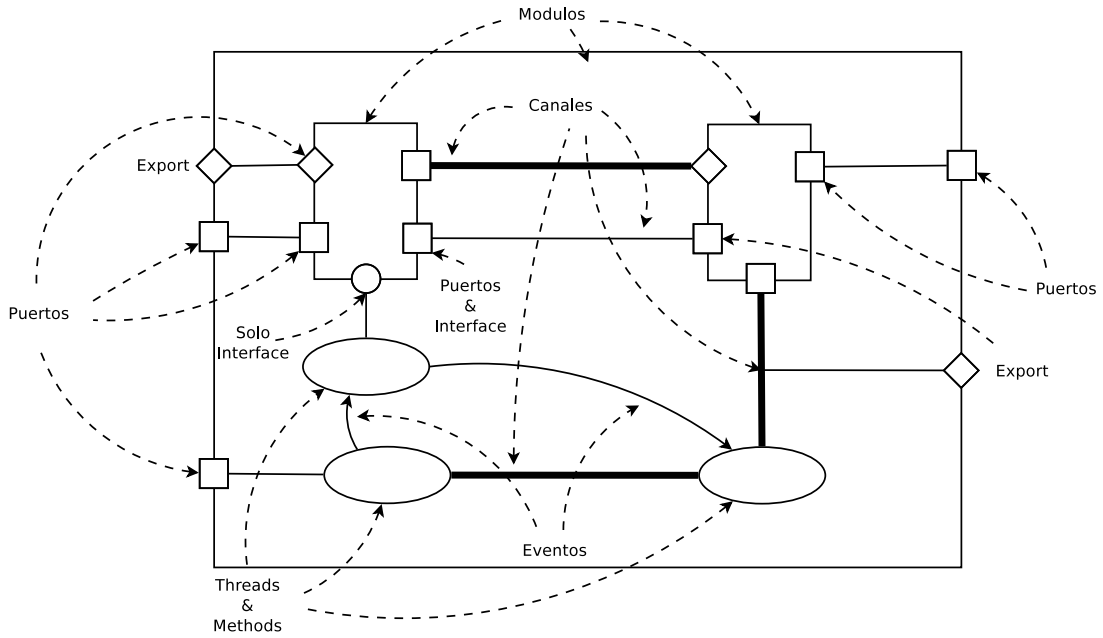


Figura 3.7: Componentes de *SystemC*.

en *constraints*, basada en aleatoriedad de datos o una mezcla de las 3. Usando la introspección de los datos, la biblioteca puede manipular de forma consistente y adecuadamente los datos para lograr un cubrimiento total de las características a verificar. También permite el desarrollo de un entorno de verificación flexible y re-utilizable, de forma tal de disminuir los tiempos en la implementación de distintos *testbenchs*. Las posibilidad de modelar y verificar sistema mediante *SystemC*, hizo que éste sea ampliamente adoptado por la industria [32, 68, 69]. La principal ventaja reside en la posibilidad de describir las características funcionales separadas de las comunicacionales, proveyendo de ésta manera una definición abstracta de la arquitectura del sistema. Análogamente la combinación del modelado a nivel de registros (RTL) y el C++ puro, permite desarrollar inicialmente el software que va a soportar el hardware

3.2.3. Ejecución y Simulación

Una implementación de *SystemC* incluye, el binario ejecutable, el cual consiste de clases, funciones, macros, etc; y un *kernel* propio, el cual implementa la funcionalidades básicas de *SystemC*.

La ejecución de una aplicación de *SystemC* consiste de dos grandes fases o etapas (elaboración y ejecución) y una menor de limpieza o *cleanup* que ocurre al finalizar la ejecución.

La primer fase es la encargada de generar las estructuras de datos e inicializarlas, establecer

la conectividad y preparar el contexto para la segunda fase. La segunda fase o etapa (ejecución) relega el control de la aplicación al kernel de *SystemC*, el cual se encarga de "orquestar" la ejecución de la aplicación creando la ilusión de concurrencia.

La simulación implica la ejecución del simulador de eventos discreto, el *Scheduler*, el cuál decide que procesos se ejecutaran mientras la simulación este corriendo, los pasos que realiza el *Scheduler* son descriptos a continuación:

- Inicialización, durante este paso, cada proceso es ejecutado una única vez (SC_METHOD) o hasta llegar a su punto de sincronización (*wait* en los SC_THREAD).
- Evaluación, se selecciona un proceso, el cual este preparado para resumir su ejecución. Esta situación puede causar notificaciones inmediatas, generando una cadena de ejecución de procesos.
- Si hay algún proceso que pueda correr y todavía no lo hizo, se vuelve al primer paso.
- Actualización, realiza la actualización de todos los procesos.
- Si llega a existir notificaciones pendientes en cuanto a *delta-delay* se selecciona algún proceso que pueda ejecutarse en ese tiempo y se vuelve al segundo paso. No se avanza el tiempo de simulación en el *delta-delay*
- De otra manera, se avance el tiempo de simulación, hasta que algún proceso que este preparado para volver a correr.
- Se determina que proceso se ejecutara y se vuelve al paso 2.

Una vez que el *Scheduler* se encuentra corriendo, este se seguirá ejecutando hasta que no existan más procesos para ejecutarse.

Capítulo 4

Arquitectura

En este capítulo se introduce la metodología propuesta tanto para el diseño como para la verificación de sistemas digitales. Antes de esto se mencionan trabajos similares, centrándose particularmente en los distintos niveles de abstracción. Se utiliza como caso de prueba un SoC el cual sera evaluado para obtener resultados concisos sobre el éxito o no de la metodología. Distintos enfoques de verificación y de diseño se utilizan y describen en este capítulo.

4.1. Antecedentes

Distintos flujos han sido propuestos, un ejemplo de esto, es él planteado por *Rashinkar* [70] (Figura 4.1), ésta propuesta se centra principalmente en el diseño del sistema en cada una de las etapas, inicialmente se realiza una descripción del sistema en C o C++ para posteriormente realizar una implementación en HDL. Se plantea una arquitectura que de soporte a la descripción realizada previamente, esta arquitectura esta compuesta por IP propios o de terceros ya sean estos de hardware o de software. Inicialmente se realiza un testeo de la arquitectura con un *testbench* el cual esta descripto a nivel de abstracción mayor al *pin-accurate*. Para realizar la verificación tanto del hardware como del software se realizan nuevos *testbenchs* acorde a las necesidades de los componentes, el realizar *testbenchs* por separado sin seguir un *golde reference model* o sin un planeamiento puede causar resultados no deseados.

Metodologías que utilizan la POO también han sido estudiadas, *Sinha* [71], propone utilizar lenguajes orientado a objetos para modelar hardware indicando que estos han demostrado ser una buena alternativa ante la creciente complejidad de los sistemas [72, 73] dado que permiten obtener una implementación ejecutable del sistema en forma rápida. *Shina* propone la utilización de *SystemC* en conjunto con estructuras denominadas *Incidence Composition Structure Project* (ICSP) [74] con el fin de dar nociones de proveer información acerca del *layout, floor-planing,*

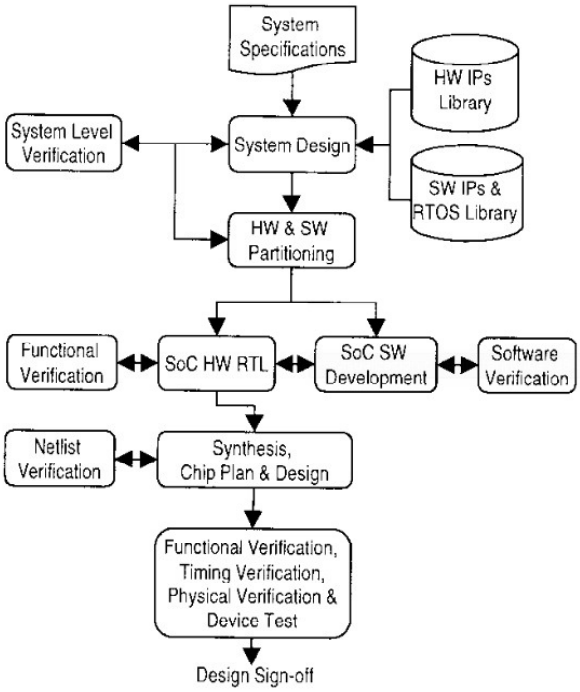


Figura 4.1: Metodología Propuesta por Rashinkar [70].

etc. En el trabajo se menciona el desarrollo de *YAML* el cual es un entorno de desarrollo de sistema, que provee soporte para la utilización de UML y auto-generación de código. Nociones de diseño Top-down son utilizadas, la verificación es mayormente realizada mediante inspección.

En el trabajo [73] se sugiere la utilización de un entorno de desarrollo *OCAPI* que provee soporte para descripción de comportamiento a nivel funcional en *C++* (nivel de abstracción alto), la capacidad de describir el sistema a nivel de registros similar al *RTL*, etc. Se propone realizar un diseño *Top-Down* en donde el primer modelo del sistema se realiza a nivel de C++ de punto flotante (funcional), posteriormente se refina a un nivel *cycle-accurate* y obteniéndose de ésta forma el segundo modelo (Arquitectura) y finalmente se procede a realizar un último refinamiento en donde se especifica por ejemplo el tamaño del *word*, denominado nivel *fixed point*. La verificación se realizo en cada uno de los niveles que se aprecian en la metodología, debido a que hay niveles descriptos en C++ y otros descriptos en HDL es necesario realizar un chequeo de equivalencia entre los resultados a fin de validar los resultados obtenidos.

En el artículo [75] se propone una metodología que consiste de cuatro pasos y la utilización de Objetos para describir el sistema. El primero de los pasos consiste en identificar el modelo computacional que describe el problema, realizando una descripción formal del sistema en MoC [76]. En el segundo paso se determina qué cantidad de *design units* son necesarias para dar

soporte al sistema. El tercer paso es el encargado de realizar el la unión entre los dos anteriores o el *binding*. Por último, se realiza un refinamiento del sistema de forma iterativa hasta llegar a un resultado aceptable.

Siguiendo el flujo natural, los lenguajes de descripción de hardware como *SystemC* también han sido utilizados en metodologías. En [77] se propone separar las características técnicas de la características de la aplicación, este enfoque se denomina *Model Driven Engineering* (MDE). Basándose en UML para obtener una representación inicial del sistema hasta llegar a una representación en RTL del sistema (VHDL o *SystemC*), se sigue el flujo de diseño descripto por *Gajski* en [78] utilizando un *Platform Independent Modele* (PIM) para abstraerse de una determinada tecnología, la verificación de los distintos niveles es realizada ad-hoc.

Mientras que en el trabajo [79] se enuncia un flujo de diseño tanto para el hardware como para el software que va a correr sobre éste. En la especificación del sistema se realiza una descripción inicial de éste en un lenguaje (C++), posteriormente se realiza una descripción en un nivel de abstracción alto, el particionado de las partes dele sistema ya sean estas de hardware o de software y finalmente la simulación del sistema. El flujo completo implica retroalimentación entre la etapa de simulación y la de particionado, logrando de esta forma el realizar una verificación del mismo. Una vez que se logra obtener una representación adecuada del sistema y este esta verificado se procede a la síntesis del mismo. En enfoque abordado por el autor difiere de los existentes en ese momento, en lugar de utilizar la descripción del sistema (C++) como entrada de un sintetizador de alto nivel (HLS, por sus siglas en ingles); se propone seleccionar manualmente las clases que podrían ser potenciales módulos de hardware a fin de realizar la codificación de éstas en VHDL y así llevar a cabo la síntesis del sistema.

En los trabajos [80,81] se plantea una metodología que prioriza un flujo en paralelo, tomando decisiones de diseño y las propiedades que éste debe cumplir, se plantean varios niveles de abstracción con el fin de ir refinando el modelo. Inicialmente plantea la utilización de UML y propiedades PSL para una descripción general del sistema; luego utiliza una maquina de estados abstractas con el fin de verificar que las propiedades se cumplan; y finalmente lleva a cabo la implementación del modelo en *SystemC* con sus propiedades expresadas en *C#*. Por su parte *Schulz-Key* [82] plantea un flujo de diseño en el cual se utiliza *SystemC* como modelo de referencia, posteriormente éste es analizado con el fin de que en futuras iteraciones se mejore el diseño, una vez que se llegó al diseño final, se procede a utilizar un OOAS (*Object Oriented Analysis System*) el cual se encarga de realizar la traducción de *SystemC* a nivel RTL, sus correspondientes simulaciones (con el fin de analizar si el proceso afecto la funcionalidad del sistema) y finalmente su síntesis.

En [69] se propuso un entorno para realizar la validación del sistema el cual contempla distintos niveles de abstracción. Uno de éstos es el nivel de éstos es el de *DEM* o *Design Error*

Modeling, este intenta detectar errores en el diseño basándose en el concepto de *bit coverage* [83], se utiliza la herramienta *AMLETO* [84] para generar un diseño con fallas o mutantes el cual va a ser utilizado por el próximo nivel de abstracción en el flujo. El segundo nivel es el de HLV o *High Level Validation* en el cuál se utiliza un *Fault Simulator* y un generador de patrones automáticos con el fin realizar la comparación entre el diseño generado en el nivel anterior y el diseño con los error corregidos, buscando el lograr una mayor cobertura en cuanto a los errores detectados. Por último, las propuestas como las observados en [85, 86], son similares a la que se enuncia en ésta tesis, discrepan en la técnica utilizada al realizar la verificación en cada una de las etapas de la metodología.

4.2. Metodología

Para el diseño y la verificación de sistemas digitales, en este trabajo se propone seguir una metodología basada en la Figura 4.2. Esta metodología surgió como una conjunción de técnicas de diseño y verificación, del artículo [34] que describe las ventajas y desventajas de TLM, se logro observar cual era el flujo adecuado para diseñar y verificar un sistema. De [13] y [87] se adoptaron distintas decisiones de diseño (diseño de bus, módulos, entidades) y finalmente de [88] y de los artículos previamente revisados, se observó que la separación en niveles de abstracción es esencial al momento de diseñar y verificar hardware.

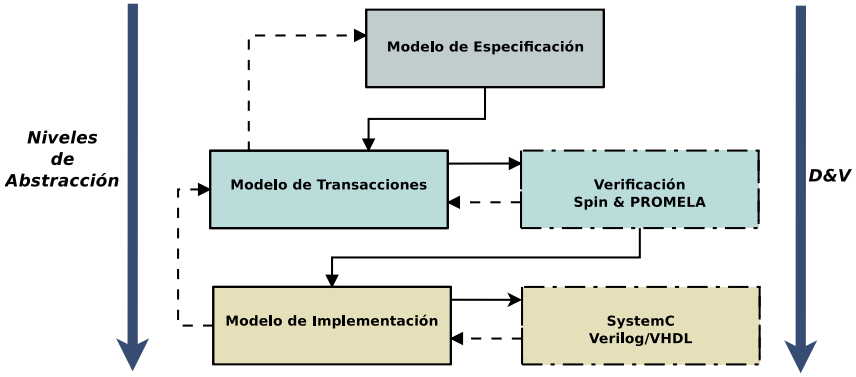


Figura 4.2: Metodología Propuesta.

En nuestra metodología se plantea una verificación particular en cada uno de los niveles de abstracción, con el objetivo reducir en el porcentaje de errores encontrados. Dado que, varios de éstos (un mal diseño de arquitectura, mal utilización de recursos, etc) serían depurados en las primeras etapas del diseño. En la Figura 4.2 se puede apreciar un flujo con distintas etapa, yendo desde un nivel de abstracción alto hasta el nivel más bajo concebido, la implementación; esta

propuesta es similar a la propuesta observada en el artículo [88] en su *Synthesis/Implementation flow*. Adicional a lo observado en [88], aquí se plantea una retroalimentación entre los distintos niveles, con el fin de promover el refinamiento del diseño.

A continuación describiremos brevemente los tres niveles o modelos de abstracción que se conciben en el flujo y sus implicaciones con el diseño.

- *Modelo de Especificación*: Este nivel es el encargado de otorgar tanto a los diseñadores como verificadores una primera aproximación del sistema. En el mismo se describen los actores relevantes del sistema y un conjunto de operaciones básicas de estos, por ejemplo si un actor relevante es una memoria, en este nivel aparecerá ésta y las operaciones para acceder a los elementos como consultas, modificaciones, direccionamiento, etc.

  Para ésta etapa se utilizo un diagrama de bloques, el cual nos provee de una visión general de los actores y las relaciones entre éstos. Adicionalmente al diagrama, se provee las especificaciones funcionales de cada actor con el fin de que éstas sean utilizadas en una próxima etapa.

- *Modelo de Transacciones*: Dicho modelo, es el responsable de las interacciones entre los componentes del sistema, y se basa en transacciones. Una transacción es la secuencia de pasos que debe seguir uno o más componentes para llevar a cabo una operación. Se definen las operaciones tanto de control como de transferencia de datos (en caso de existir) que se deben realizar; en ésta (la transacción) se ocultan detalles de tanto de implementación como de manejo de protocolo de comunicación. El resultado de una transacción puede o no afectar otros componentes.

  Para esta etapa se decidió utilizar los diagramas de secuencias de UML debido a que denotan las interacciones entre los objetos o componentes de un sistema, mostrando como los actores o módulos operan entre ellos y el orden de estas operaciones. Además, se provee un diseño en TLM para tener una primera aproximación de la arquitectura y poder realizar de esta forma una verificación basada tanto en simulaciones, como en métodos formales.

- *Modelo de Implementación*: Finalmente, esta etapa es la de implementación del sistema a nivel de *HDL*. A este nivel lo que se busca hacer es la correcta implementación de las especificaciones, separando las partes correspondientes al *datapath* de las correspondientes al *control* del sistema. Por último en esta etapa es posible utilizar distintos lenguajes de *HDL* para así seguir refinando el diseño, éstos son: *SystemC*, *VHDL*, *Verilog*, etc.

  La verificación del sistema a este nivel se realizará utilizando un entorno desarrollado puramente en *SystemC*, con el fin de aprovechar beneficios como: el uso de la biblioteca

de Verificación de *SystemC*, la interacción entre distintos lenguajes de HDL y el análisis de simulaciones mixtas.

4.3. Arquitectura del Sistema

El sistema que se concibió para analizar la metodología planteada, es el que se observa en la Figura 4.3. Además se puede apreciar distintos módulos, teniendo cada uno de éstos funciones particulares, la función principal del sistema es la del ordenar grandes volúmenes de datos, dando así soporte a las demandas actuales que debe afrontar los grandes sistemas de almacenamiento, como *datacenters*, sistemas de procesamiento digital de señales,análisis de imágenes, etc.

A continuación describiremos brevemente las funcionalidades de cada uno de los bloques que componen el sistema:

- *ETH*: Modulo encargado de interactuar con el mundo exterior, es la interfaz por la cual el sistema recibe datos, los procesa y los entrega de vuelta al mundo exterior. Se concibe para en un futuro, implementar un bloque IP que respete el protocolo Ethernet TCP/IP.
- *DMA*: Modulo encargado de realizar la transferencia de datos entre los dispositivos, el mismos realiza la transferencia de un dispositivo A a un dispositivo B mediante un canal común, en éste caso un *BUS-AHB*.
- *BUS-AHB*: Bus del sistema, el mismo es utilizado debido a su rápida transferencia de datos, permitiendo realizar lecturas/escrituras simultaneas sobre un mismo canal. Adicionalmente este tipo particular de *BUS* es el que mayormente es adoptado en la industria de los semiconductores.
- *SorterX*: Este módulo se encarga de ordenar los datos que el sistema le entrega, devolviéndolos ordenados cuando le son solicitados.
- *Mem*: memoria del sistema, la misma tiene un tamaño proporcional a las capacidades máximas del sistema, logrando así el correcto almacenamiento de los datos.

4.3.1. Interacción entre Componentes

Para llevar a cabo la tarea de ordenar los datos, se concibieron diagramas de secuencias, los cuales nos permiten expresar la interacción entre los distintos componentes del sistema. Además se agregó la idea de un Árbitro o control del sistema, el cual es el encargado de orquestar la

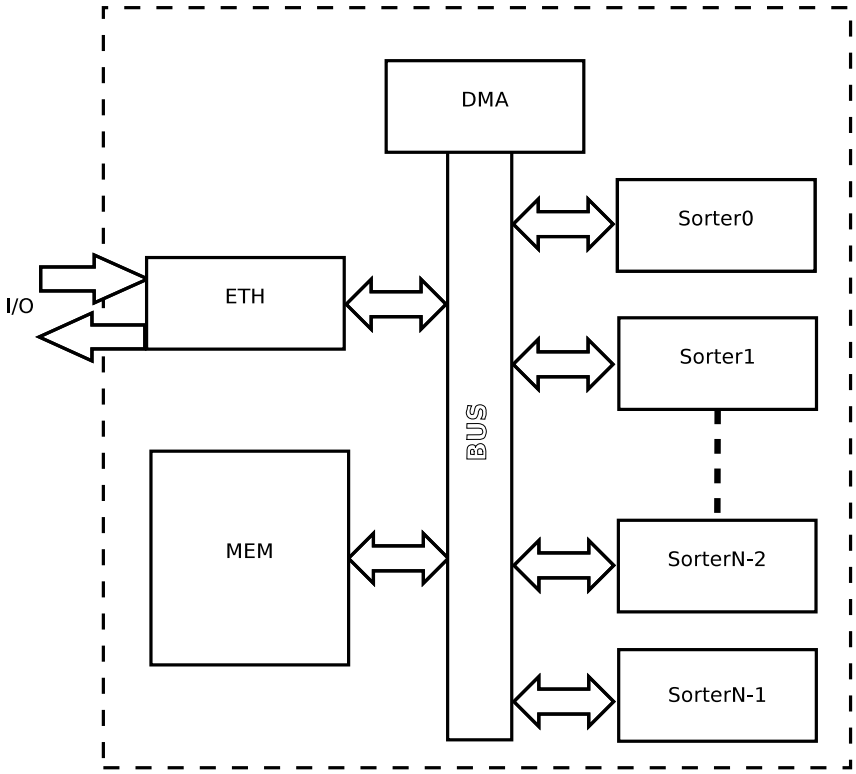


Figura 4.3: Diagrama en bloque del sistema.

interacción de los módulos. Por lo tanto para la concepción de la funcionalidad se previó que el sistema debe dar soporte a las siguientes funcionalidades internas:

1. Recibir información del exterior y almacenarla en memoria.
2. Escribir la información almacenada en memoria en uno de los *Sorters* del sistema, siempre que éste este disponible.
3. Escribir los datos entregados por el *Sorter* en la memoria.
4. Enviar los datos ordenados del sistema hacia el exterior.

Dichas funcionalidades pueden ser pensadas como interacciones entre los distintos módulos y el modulo adicional, previamente nombrado, el Árbitro. Por lo tanto se definieron las siguientes funciones:

- *recv_frame_from_eth_w_mem* : Se encarga de dar soporte a la funcionalidad del item 1 nombrado anteriormente. La interacción de los componentes se puede observar en la Figura 4.4 y su modo de operar es el siguiente:

  1. El árbitro controla si hay datos disponibles haciéndole la correspondiente consulta al modulo ETH.
  2. En caso de haber datos le indica al DMA que deberá realizar una transferencia de datos desde el ETH hacia la memoria (MEM).
  3. El DMA realiza la transferencia como le fue indicada, llevando a cabo lecturas del ETH y las escrituras en la memoria.
  4. Finalizada la operación, se devuelve un mensaje indicando el éxito o el error de la misma.

- *recv_frame_mem_w_sort*: Función encargada de escribir datos desde la memoria hacia el *Sorter* que éste disponible para realizar su tarea (item 2). La interacción de los componentes se puede observar en la Figura 4.5 y su modo de operar es el siguiente:

  1. El árbitro consulta si el *Sorter* esta predispuesto a realizar la tares.
  2. En caso de que sea afirmativa la respuesta, se le indica al DMA que próximamente deberá realizar una transferencia de datos desde la memoria hacia el *Sorter*. En caso de ser negativa la respuesta, se finaliza la función indicando la no concreción de la misma.

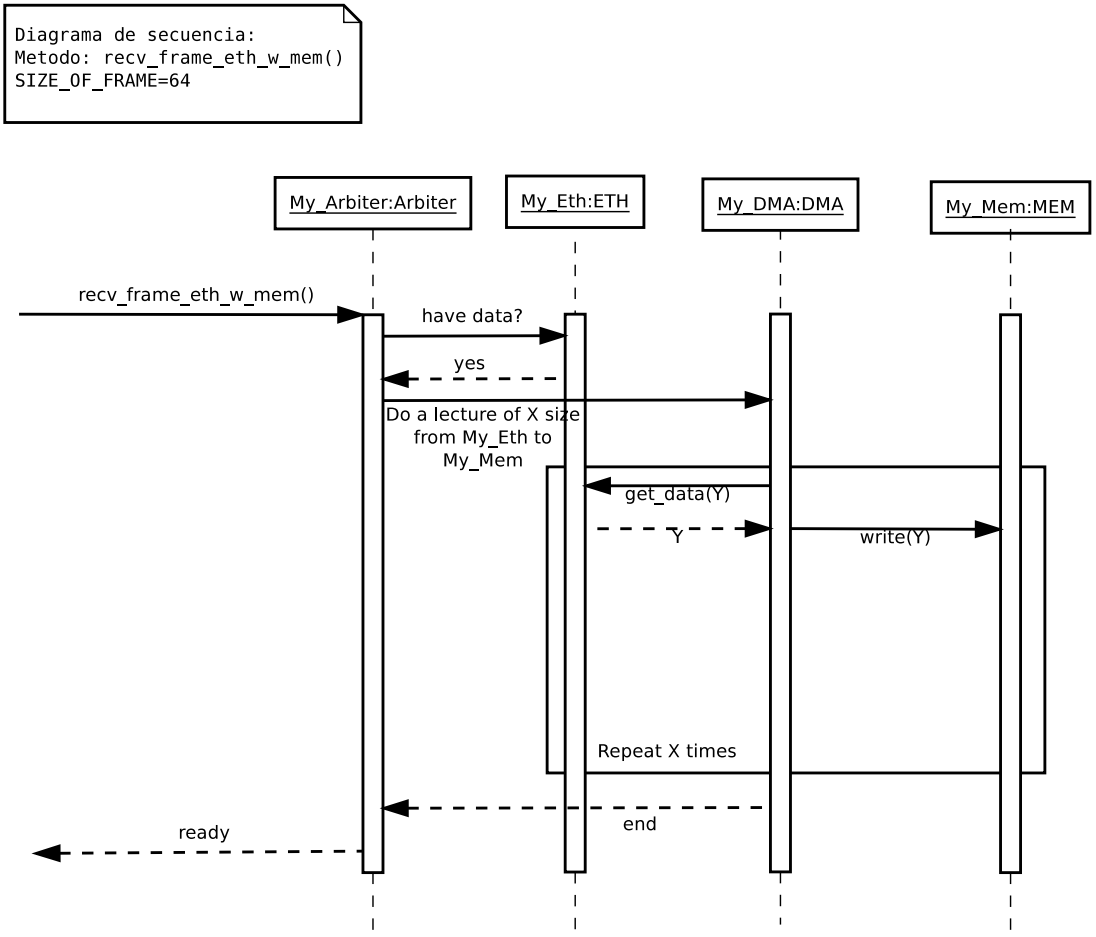


Figura 4.4: Función *recv_frame_eth_w_mem*.

3. El DMA lleva a cabo la transferencia previamente pactada, realizando lecturas desde la memoria para escribir los datos en el *Sorter*. Los datos se transfieren mediante el *BUS* que ambos dispositivos comparten.
4. Finalizada la operación, se devuelve un mensaje indicando el éxito o el error de la misma.

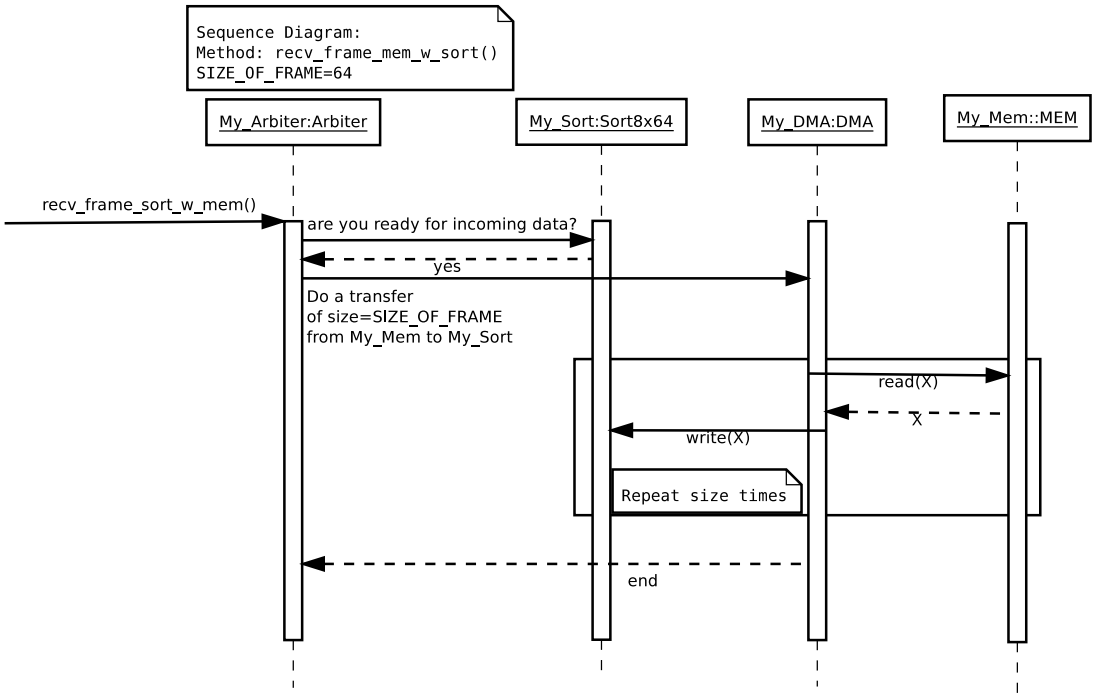


Figura 4.5: Función *recv_frame_mem_w_sort*.

- *recv_frame_sort_w_mem*: Función encargada dar soporte a la escritura de datos desde un *Sorter* a la memoria (item 3). La interacción de los componentes se puede observar en la Figura 4.6 y su modo de operar es el siguiente:

1. El *Sorter* le indica al árbitro que ya finalizó su tarea.
2. El árbitro se encarga de consultar que el DMA este disponible En caso de que sea afirmativa la respuesta, se le indica al DMA que próximamente deberá realizar una transferencia de datos desde la memoria hacia el *Sorter*. En caso de ser negativa la respuesta, se finaliza la función indicando la no concreción de la misma.
3. El DMA lleva a cabo la transferencia previamente pactada, realizando lecturas desde la memoria para escribir los datos en el *Sorter*. Los datos se transfieren mediante el

BUS que ambos dispositivos comparten.

4. Finalizada la operación, se devuelve un mensaje indicando el éxito o el error de la misma.

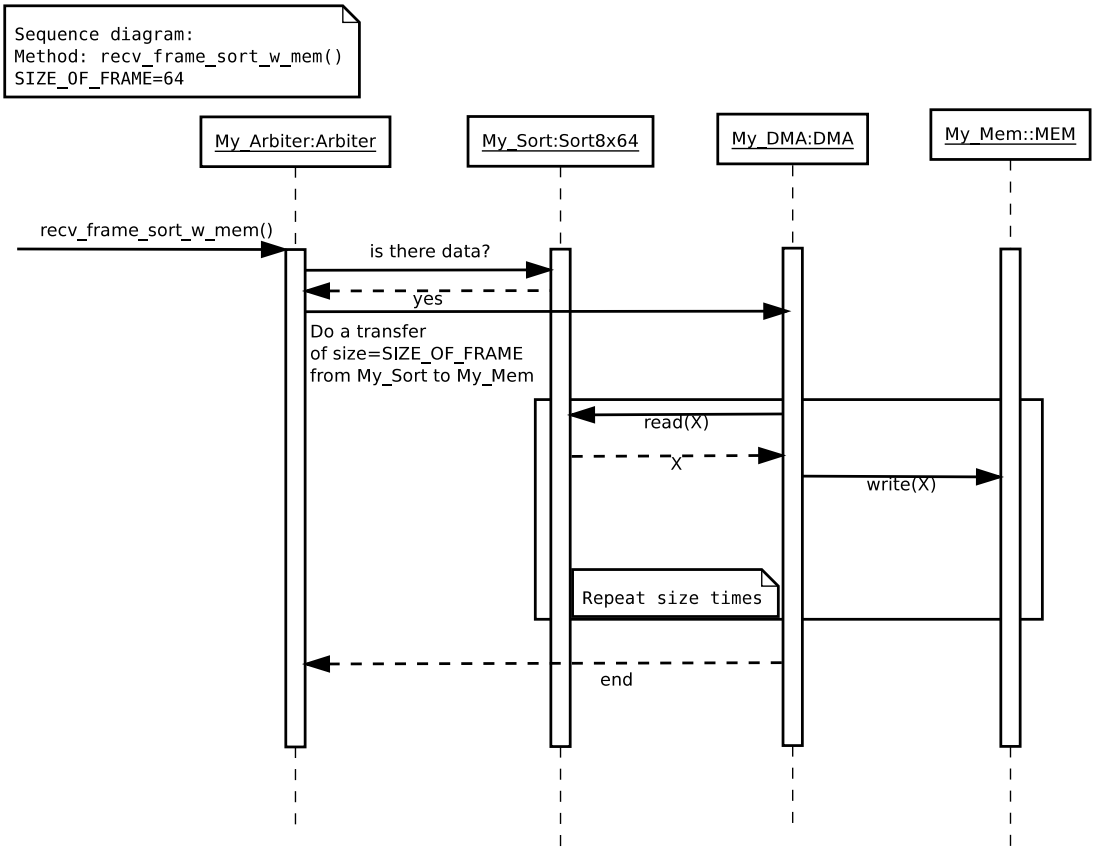


Figura 4.6: Función *recv_frame_sort_w_mem*.

- *senda_frame_to_out*: Esta función es la que se encarga de realizar la transferencia datos desde la memoria hacia el exterior (item 4). La interacción de los componentes se puede observar en la Figura 4.7 y su modo de operar es el siguiente:

1. El árbitro se encarga de consultar que el DMA este disponible. En caso de que sea afirmativa la respuesta, se le indica al DMA que próximamente deberá realizar una transferencia de datos desde la memoria hacia el ETH. En caso de ser negativa la respuesta, se finaliza la función indicando la no concreción de la misma.

2. El DMA lleva a cabo la transferencia previamente pactada, realizando lecturas desde la memoria para escribir los datos en el ETH. Los datos se transfieren mediante el *BUS* que ambos dispositivos comparten.

3. El ETH envía los datos hacia el exterior del sistema.

4. Finaliza la operación, un mensaje es devuelto indicando el éxito o el error de la misma.

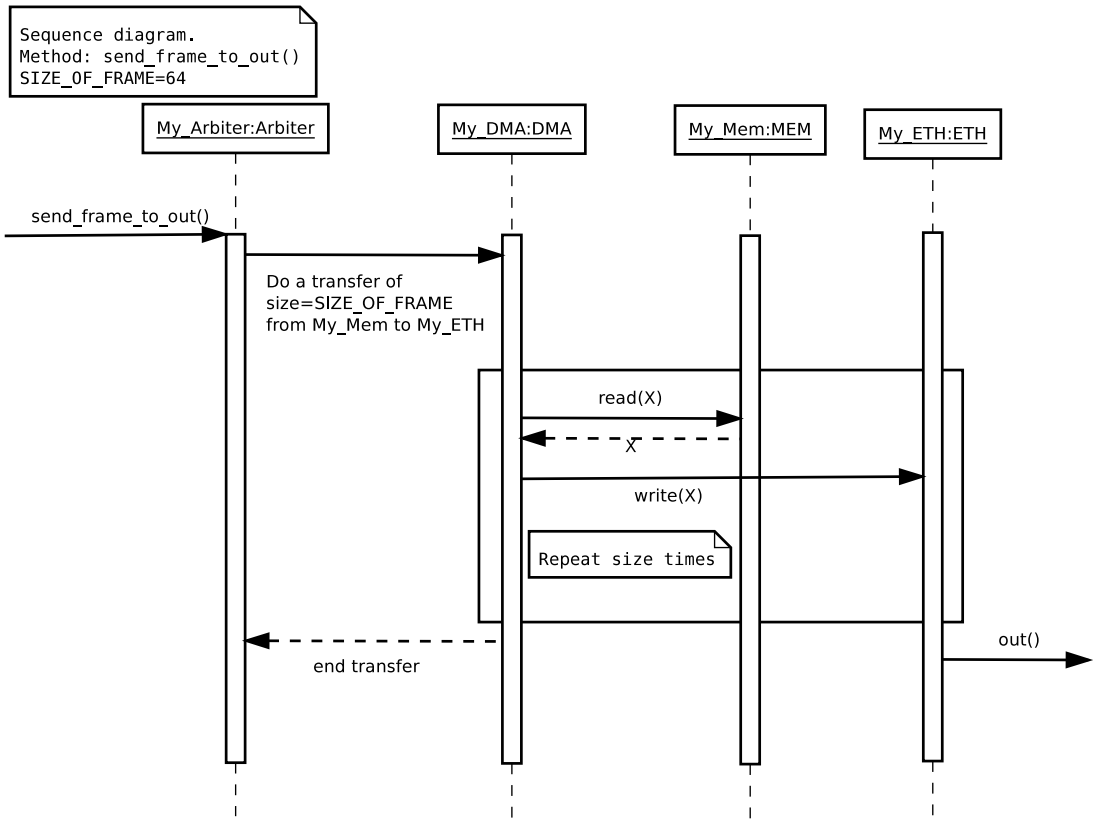


Figura 4.7: Función *send_frame_to_out*.

4.4. Verificación del Sistema

En esta sección se describe cómo se llevó a cabo la verificación de los distintos niveles de abstracción.

4.4.1. Modelo de Especificación

En este nivel de abstracción, el primero en nuestro flujo, la verificación de las especificaciones es realizada mediante observación, prueba y error. Indicando, cuando sea necesario, las características de los componentes del sistema, el soporte que deben de otorgar, etc.

4.4.2. Modelo de Transacciones

La verificación en este nivel de abstracción, el segundo en nuestro flujo, es concebida de dos formas distintas: la primera basada en simulaciones y la segunda utilizando métodos formales.

- *Basada en Simulaciones:* Debido a que la implementación de nuestro modelo en este nivel de abstracción, basado en las descripciones hechas en 3.1.2, es una implementación ejecutable, la verificación funcional sera mediante observación de distintas simulaciones.
- *Métodos Formales:* La verificación del sistema a nivel de transacciones utilizando métodos formales, se realizará mediante un *Model Checker*; es necesario realizar la traducción de nuestro sistema a un lenguaje que el *Model Checker* comprenda. La traducción del sistema se basará en las reglas mencionadas en [89–91], se utilizará *Promela* como lenguaje, mientras que el *Model Checker* será *Spin*.

Spin y PROMELA

En nuestro caso particular vamos a utilizar *Spin*, el cual es un *Model Checker* explícito, que da soporte al diseño y la verificación de sistemas basados en procesos asincrónicos. Tiene sus bases en los sistemas de verificación *on the fly* de protocolos del principio de los 80. *Spin* orienta sus modelos a buscar la correcta *interacción de procesos* abstrayéndose de la computación de éstos. La *interacción de procesos*, en *Spin*, se puede especificar mediante primitivas *rendezvous*, pasaje asincrónicos de mensajes a través de canales, variables compartidas o una combinación de éstas tres.

Spin provee:

1. Notación intuitiva, de forma tal que el diseño del sistema sea lo menos ambiguo posible, obviando detalles de implementación final.
2. Una notación concisa y poderosa la cual nos permite expresar requisitos generales, como correctitud, etc.
3. Una metodología para establecer la consistencia lógica entre el diseño (1) y la correctiva de éste (2).

Por su parte *PROMELA* [92, 93], es una lenguaje de verificación de modelos, creado por *Gerard J. Holzmann*. El mismo es similar a un lenguaje de programación de alto nivel, aunque actualmente se utiliza para la descripción de autómatas finitos. Parte de su semántica se ve influenciada por el lenguaje de guardas de *Dijktras* [94], los procesos secuenciales para comunicación de *Hoare* [95] y C [96]. Este permite la creación dinámica de procesos concurrentes para modelar, por ejemplo, sistemas distribuidos. La comunicación entre los procesos se basa en envió de mensajes a través de canales, los cuales pueden ser sincrónicos *rendezvous* o asincrónicos *buffered*. Los programas en *PROMELA* se pueden verificar a través de *Spin*, con el fin de chequear la correctitud del programa; además éste lenguaje da soporte a *constraints* descriptos en *Linear Temporal Logic* logrando así el ampliar el abanico de propiedades verificables.

Lo beneficioso de *PROMELA* es que su sintaxis es similar a C, por lo tanto el esfuerzo realizado para aprenderlo es mínimo. Un programa en *PROMELA* en general consiste de procesos, canales (para el envió de **mensajes**) y variables. Los procesos son objetos globales los cuales representan entidades concurrentes de un sistema distribuido. Los canales y variables se pueden declarar como globales o como locales a un proceso.

Como se observa en [89–91] el realizar la traducción de *SystemC/TLM* a *PROMELA* ya ha sido estudiada. Otras consideraciones al momento de llevar a cabo la misma son:

- *Scheduler*: El *Scheduler* de *SystemC* se asegura de que un proceso que esta ejecutando no sea interrumpido por otro proceso al menos que el primero explícitamente devuelva en control al *Scheduler* mediante una sentencia *wait* por un tiempo determinado o a la espera de un evento particular. Por lo tanto debemos de asegurarnos de simular el mismo comportamiento. Además el *Scheduler* se asegura de que el tiempo de simulación avance solamente cuando ningún proceso del *pool* de procesos es elegible.

- *Llamadas a funciones*: Estas son usadas tanto para denotar interrupciones como transacciones. Un procesos P realiza una llamada a función dejando su control fuera del componente para que finalmente la función se ejecute en el componente donde ésta definida. La ejecución de P debería continuar solo después de que la llamada a la función finalice, esto significa que si el modulo donde se encuentra definida la función que se llamo realiza un *wait*, P deberá entregar su control al *Scheduler*, dejando así el resto de su ejecución a futuro, cuando el *Scheduler* vuelva a seleccionarlo.

Finalmente se consideraron las siguientes situaciones al momento de modelar el sistema para llevar a cabo su verificación:

1. La abstracción de la arquitectura es esencial en esta etapa. Dado que un sistema en general cuando un proceso P de un módulo I realice una llamada a la función *b_transport()*; I

ejecuta la llamada de *b_transport()* en su *initiator_socket* *i_s*, *i_s.b_transport()*. Donde la llamada sera recibida y procesada por un módulo T, el cual tiene su respectivo *target_socket* *t_s* y una función cualquiera, f, registrada como la función *b_transport* del *target_socket*.

2. Dado que un sistema esta compuesto tanto por módulos como por funciones, se procederá a definir ambos como procesos en *PROMELA*, considerando solamente los procesos de tipo *SC_THREAD* dado que este tipo de procesos es el más común en *SystemC*. Adicionalmente cada función en los módulos debe ser llamada como máximo por un único proceso, ignorando casos distintos y llamadas recursivas.

3. Cada proceso va a esperar únicamente por un evento, en caso de que n procesos P_i esperen por un mismo evento e, dicha situación se puede traducir como n procesos P_i que esperan por n eventos e_i donde $0 \leq i < n$.

4. Debido a que la asociación entre dos módulos para llevar a cabo su comunicación es predeterminada en la etapa de elaboración del sistema, solo consideraremos módulos que se encuentren conectados estáticamente y no cambien respecto del tiempo sus conexiones a fin de reducir la complejidad del diseño del sistema en *PROMELA*.

4.4.3. Modelo de implementación

Para realizar la verificación funcional del sistema a nivel de implementación se propuso generar un entorno de verificación [10] como el observado en la Figura 4.8. Se evidencian distintos módulos, los cuales serán descriptos a continuación:

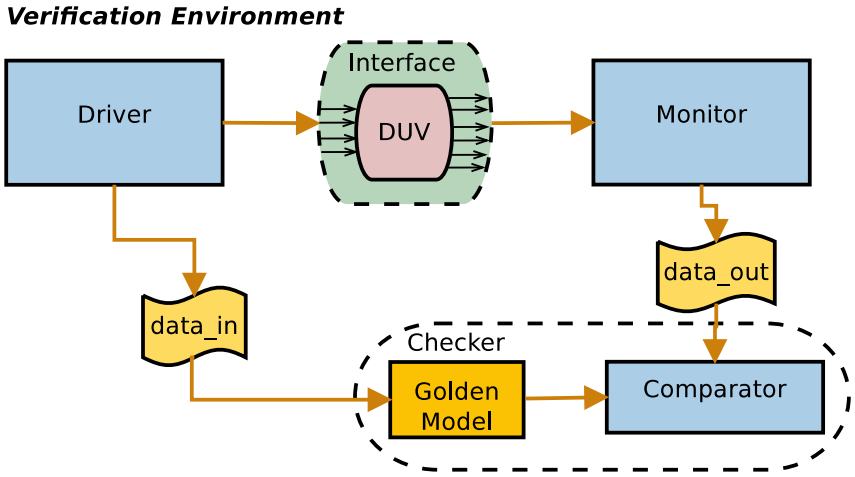
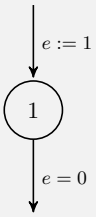
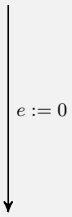
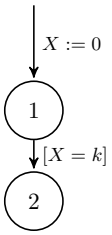
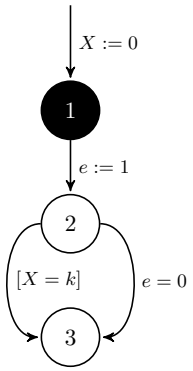
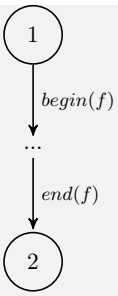
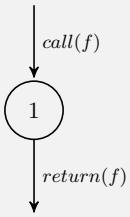


Figura 4.8: Entorno de Verificación.

Tabla 4.1: Principales Elementos de SystemC

Elemento	Representación	Elemento	Representación
wait(e)		e.notify()	
wait(k ns)		wait (e, k ns)	
f()		port.f()	

- *Driver*: Dicho componente es el encargado de generar los estímulos acordes para la correcta estipulación del DUV; el mismo cumple funciones similares al conjunto *Sequencer/Driver* de *UVM*, dado que genera estímulos siguiendo un determinado patrón. El patrón varia respecto el componente a verificar, logrando así con cambios mínimos, el estimular distintos bloques; aumentando su reusabilidad como módulo.

- *Interface*: Módulo encargado de traducir los valores generados por le driver en *SystemC* a valores validos en *VHDL* o *Verilog* según la necesidad; este módulo es necesario para lograr la correcta estimulación del DUV.

- DUV: *Design Under Verification* (DUV), es el bloque o módulo que se desea verificar, éste puede ser desde un multiplexor, una memoria, *Sorter*, modulo eth, DMA, etc.

- *Monitor*: El monitor es el modulo encargado de sensar las señales que salen del DUV, adicionalmente cumple funciones como: llevar a cabo chequeo de protocoles, métricas de cobertura, analizar cuando el sistema ha llegado a un determinado estado, etc.

- *Golden Model*: En todo entorno de verificación es necesario tener un modelo de referencia, en el nuestro, el *Golden Molde* cumple dicha función. Este puede estar escrito en C, C++ o en *Matlab*, debido a la fiabilidad de estos lenguajes.

- *datain&dataout*: Archivos intermedios que serán utilizados por otros módulos, en los mismos se almacenan tanto los estímulos de entrada como de salida del DUV. De esta forma, éstos van a poder ser procesados en un futuro para obtener información relevante al proceso de verificación.

- *Comparator*: Componente que compara las salidas generadas por el *Monitor* y el *Golden Model*; el resultado se lo comunica al *Checker* para que éste genere una respuesta adecuada a la situación.

- *Checker*: Componente ficticio el cual, teniendo una instancia del *Golden Model* y del comparador, se encarga de llevar a cabo el chequeo entre la salida del primero y del segundo y analiza si estas son las esperadas o no; en caso de que ocurra lo primero se da por finalizada la verificación, mientras que si ocurre lo segundo se deberá volver a la etapa de diseño y realizar las modificaciones acordes al DUV para que ésta situación no vuelva a ocurrir.

Como se puede observar el modelo del entorno de verificación no cuenta con bloques que se encarguen de medir la cobertura o de generar secuencias inteligentes para la estimulación del DUV, esto puede ser una desventaja respecto de entornos de verificación planteados siguiente

la metodología de *UVM* [7]. El no contar con una noción de cobertura genera una incerteza al momento de decidir cuando debe de finalizar la verificación de un módulo, IP, etc; esta decisión será tomada analizando los casos de pruebas que se han estimulados y si éstos cubren el total de posibles valores y/o situaciones que pueden existir. En los próximos capítulos de la Tesis se verá cual fue la ventaja o la desventaja del entorno de verificación planteado en este capítulo.

Capítulo 5

Caso de Estudio

En este capítulo se explicará como fue aplicada la metodología propuesta de la Sección 4.2, enumerando las distintas decisiones de diseño que se fueron tomando a través de los distintos niveles de abstracción.

5.1. Diseñon a nivel de TLM

Previo a describir cómo se realizo el desarrollo del modelado del sistema en *TLM* es necesario introducir ciertos conceptos. Los tres principales conceptos que uno debe de contemplar a la hora de desarrollar el diseño de un módulo en TLM son: el modo en que este enviará datos al exterior, el modo en el que él mismo recibirá datos desde el exterior, y la estructura de esos datos. En la Figura 5.1 se pueden observar la estructura que da soporte a lo anteriormente nombrado. A continuación describiremos brevemente la función de cada uno de los componentes.

- *Socket Initiator*: Elemento encargado de realizar el envío de datos hacia el exterior del módulo, el mismo provee al usuario de escrituras bloqueantes y no bloqueantes con el fin de dar soporte a sistemas sincrónicos y asincrónicos.
- *Socket Receiver*: Elemento encargado de recibir datos desde el exterior del módulo, el mismo provee al usuario de lecturas bloqueantes y no bloqueantes, con el fin de dar soporte tanto a sistemas sincrónicos y como asincrónicos.
- *Generic Payload*: Elemento básico para el envío de datos, es utilizado como paquete universal para la transferencia de datos entre modelos *TLM*. El mismo contempla el almacenamiento de distinto tipo de datos, tales como: comando a ejecutar, dirección de memoria, datos crudos, etc. Este puede ser de utilidad para el usuario. Adicionalmente provee de métodos tanto para actualizar y consultar los respectivo atributos.

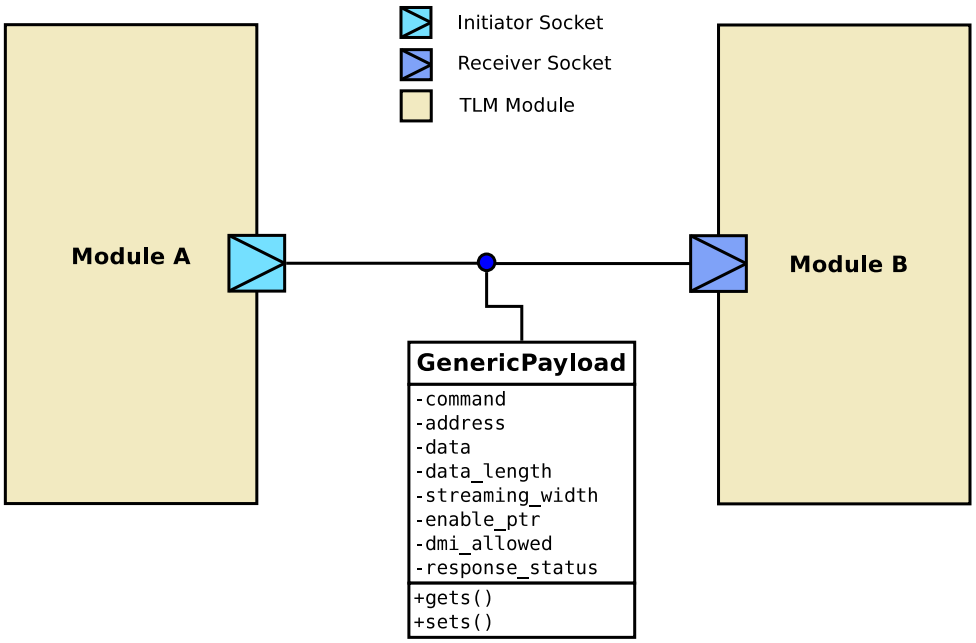


Figura 5.1: Componentes TLM.

Para el desarrollo del sistema a nivel de transacciones, se siguieron los principios de diseño de arquitecturas basadas en bus y las ideas enunciadas en [13] sobre el diseño en TLM, considerando distintas maneras de interconectar componentes. Como es mencionado en el libro anteriormente citado, las capacidades que pueden llegar a tener los canales que interconectan los módulos en *TLM* son las de decodificar la dirección del mensaje y la de dirigir la transacción desde un módulo *Sender* hasta un módulo *Receiver* (Modo *Router*). En nuestro caso particular, se hizo una primera aproximación del sistema basándonos en un entorno regido por módulos *Masters*, *Slaves* y un *Bus* común. Teniendo éste último la capacidad de arbitrar respecto de los paquetes, el estado del sistema, y la secuencia que debe seguir una trama de datos para que el sistema lleve a cabo su cometido. El diseño final en TLM se puede apreciar en la Figura 5.2, en el mismo se pueden observar los siguientes componentes:

- *Módulos Slaves o Pasivos*: Representan componentes pasivos del sistema, éstos solo responden a las consultas tanto de escritura como de lectura.
- *Módulos Master/Slaves o Activos*: Son los encargados de realizar modificaciones sobre los módulos pasivos, éstas pueden ser el de pedir leer información de éstos, querer escribirles información, etc.
- *Bus/Arbitro*: Es el encargado de velar por el correcto funcionamiento del sistema, admi-

nistra los pedidos de los diferentes *Masters* , siendo consciente del estado del sistema, y gerencia según las prioridades de los *Masters* que generan la consulta.

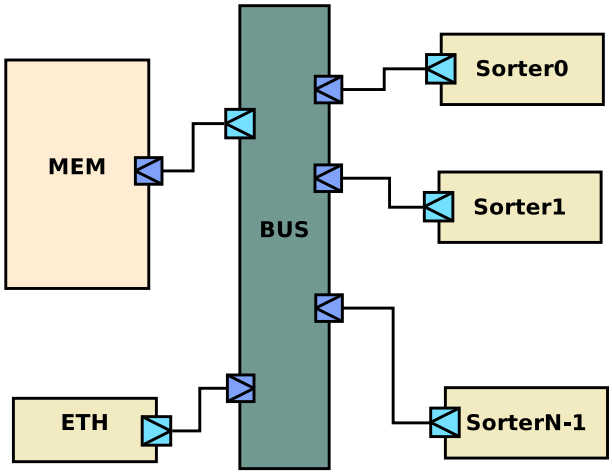


Figura 5.2: Sistema en TLM.

Un ejemplo de código *Master*, que realiza transferencias de datos se observa en el Código 1 del Apéndice. En el código se puede apreciar el componente encargado de iniciar la transacción, un *socket* de tipo *initiator* y el *procedure* encargado de generar el paquete (que será enviado).

Análogamente, los módulos *Slaves* son aquellos que solamente se encargan de dar respuesta a las necesidades generadas por los *Masters*; un ejemplo de esto son las Memorias. En el Código 2 del Apéndice, se puede observar la implementación en *TLM* de un módulo *Slave*, se aprecia el componente *socket* de tipo *target* cuya función es la de recibir paquetes y también se aprecia el *procedure* que procesa al paquete recibido y genera una respuesta.

Finalmente, veremos como es un bus de tipo router [13] implementado, así como lo indica su tipo solo debe direccionar los paquetes recibidos al respectivo receptor (Sorter) que este ocioso. El bus cuenta internamente con componentes tanto para iniciar como para recibir transacciones, y un procedimiento que se encarga de decidir a que *Sorter* enviar el paquete. En el Código 3 del Apéndice, se aprecia su implementación en *TLM*, observándose los componentes *sockets* tanto de tipo *initiator* como de tipo *target*. El hecho de que el bus se haya implementado de esta manera, es para tener una alternativa sencilla para el control, de otra forma se debería de haber creado un módulo responsable de realizar el control del sistema.

5.2. Diseño a nivel RTL

En lo siguiente de la Sección se comentan cuáles fueron las decisiones de diseño elegidas durante el flujo de Diseño/Verificación a nivel RTL.

5.2.1. Sorter

El *Sorter* es el único módulo diseñado *in-house*, implementa una arquitectura especifica para llevar a cabo el ordenamiento de los datos, esto se realiza mediante una variante del algoritmo de la burbuja, debido a su simplicidad al momento de implementar.

El módulo funciona con un cierto grado de paralelismo, el cual es provisto mediante el ingreso concurrente de datos al bloque, los que son encapsulados en paquetes para ser posteriormente ordenados. Cada paquete es inicialmente ordenado, para luego realizar el correspondiente *merge* con los paquetes que ya están almacenados en el sistema, trabajando siempre con un *merge* de dos paquetes, volviendo a almacenar la *parte baja* en la memoria, mientras que la parte alta reingresa al sistema para ser devuelta ordenado.

En la Figura 5.3 se observa el *core* de la micro-arquitectura del sistema, donde los tres bloques principales son:(1) *OESN-X*[1], (2) *OEM-2X*[2] y (3) Memoria interna. Análogamente se pueden observar también registros y multiplexores, los cuales son necesarios para la integración de la unidad de ordenamiento.

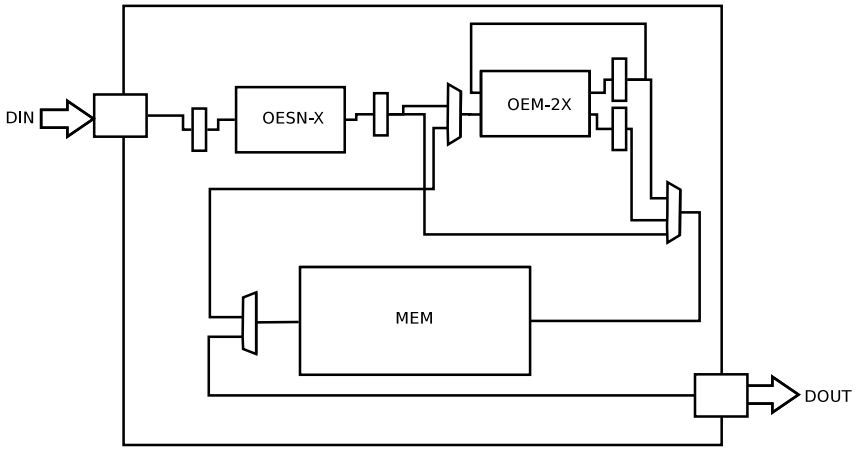


Figura 5.3: IP Micro-Arquitectura.

Una de las consideraciones a la hora del diseño del bloque es tener en cuenta el impacto directo en la mayoría de los recursos que tiene el grado de paralelismo. Tanto los puertos de entrada como

[1]Odd-Even Sorting Network
[2]Odd-Even Merge

de salida necesitan dar soporte a lo anteriormente nombrado, permitiendo transferir paquetes desde y hacia el módulo. En el caso del puerto de salida, el requisito es más fuerte dado que también impacta directamente en el *throughput* del sistema mientras que los puertos de entrada podrían ser serializados sin mayor impacto en el *speed-up*. La memoria interna provee un puerto doble de escritura/lectura con el fin de dar soporte al *pipe* de la arquitectura. Los bloques *OESN-X* y *OEM-2X* se encargan de ordenar los datos a nivel de paquete; éstos fueron concebidos pensando en *Sorting Networks* [97, 98] y consecuentemente también son afectados por el grado de paralelismo del sistema.

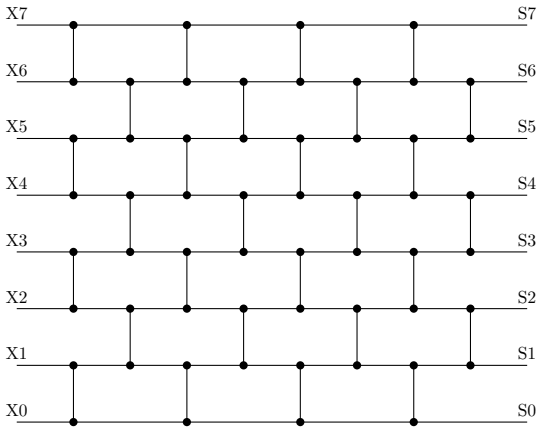


Figura 5.4: Odd-Even transposition Sorting Network.

Las *Sorting Networks* [97] es una estructura ampliamente adoptada, la cual realiza ordenamiento de forma eficaz y con alta *performance*. La implementación de estas, se lleva a cabo mediante redes de bloques de *compare&switch* donde la entrada paralela ingresa a éstos y al final de los mismos se encuentra correctamente ordenada.

En el caso de *OESN-X*, la *network* seleccionada fue una *odd-even sorting network* también conocida como *parsing sorting network*; en la Figura 5.4 se observa la estructura de ésta. Para el bloque *OEM-2X*, dado que se encarga de ordenar dos entradas previamente ordenadas mediante *mergin*, se procedió a la reutilización de las *Sorting Networks* anteriores, y una última etapa de bloques *C&S* los cuáles cumplen una función similar a la ultima etapa de bloques de una *iterative odd-even merge network*. En la Figura 5.5 se observa la estructura de la misma. Ambas estructuras fueron concebidas para dar soporte a una grado de paralelismo de ocho elementos.

El comportamiento dinámico del sistema es controlado mediante dos ciclos iterativos. El ciclo mayor se encarga de obtener cada entrada al sistema y registrarlo en su contados local *SPC*. El ciclo interno realiza el proceso iterativo de unir cada paquete nuevo con los ya existentes que se encontraban guardados en la memoria como se describió previamente. El proceso de

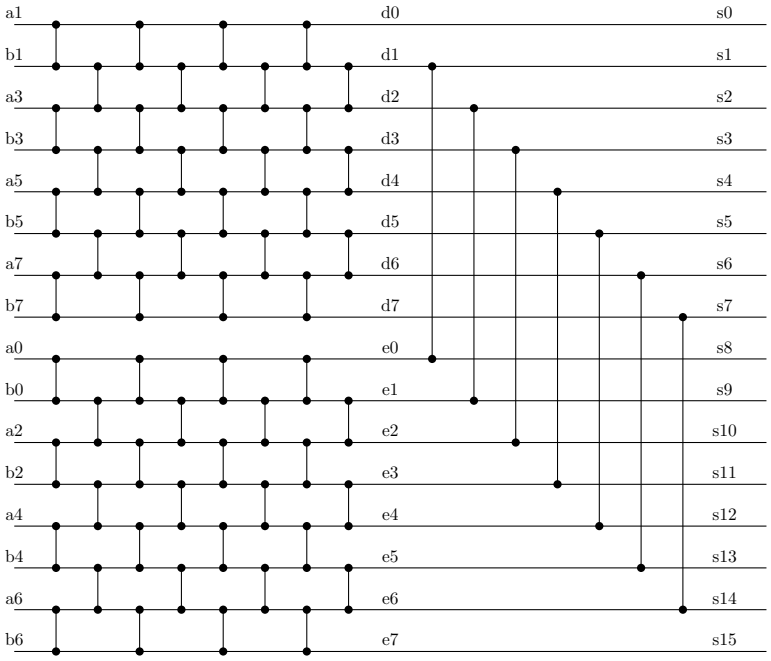


Figura 5.5: OEM-2X.

ordenamiento ejecutado por la arquitectura se describe a continuación:

Dado que el *Sorter* es el único bloque que se realizó a medida, creemos que es una buena practica el realizar un respectivo análisis, en cuanto a las elecciones de diseño que se tomaron.

El análisis de complejidad permite obtener información cuantitativa para determinar si un diseño particular es más eficiente que otros. Relacionado a las estructuras de ordenamiento, *Bitton* en [99], presentan un completo análisis de algoritmos paralelos de ordenamiento en hardware, exponiendo las diferencias de éstos a través del tiempo de ejecución y del área necesaria para implementarlos; esto último se denota con las unidades de procesamiento que necesita cada algoritmo para implementarse. Por otro lado, *Clark* [100], realizaron un análisis cuantioso sobre trece estructuras *VLSI* de ordenamiento y así como *Bitton*, se propusieron métricas que comprenden tanto el área como el tiempo para realizar la comparación entre las distintas estructuras. Por lo tanto, la complejidad de nuestro bloque de ordenamiento también puede ser analizada en cuanto a tiempo y área.

El análisis de tiempo de ejecución de la arquitectura y de área depende del parámetro F_l, el cual define la cantidad de elementos totales a ordenar mientras que el parámetro n define el grado de paralelismo del sistema o la cantidad de elemento a ordenar en cada instancia.

Para el análisis del área de la arquitectura, la unidad básica que se contempló el área del bloque *Compare&Switch* o *C&S*. También por simplicidad el área del bloque de memoria es

Algorithm 1 Algoritmo de Ordenamiento.

$DIN \leftarrow P_i$
$SPC \leftarrow SPC + 1$
$DIN \leftarrow P_i$
if $SPC == 1$ **then**
  $MEM[0] \leftarrow P_S$
else
  if $SPC == 2$ **then**
    $E = MEM[0]$;
    $MEM[0] \leftarrow (P_S, E)_{SL}$;
    $MEM[1] \leftarrow (P_S, E)_{SH}$;
  else
    for $j \leftarrow 1 hasta SPC - 2$ **do**
      $E \leftarrow MEM[j]$;
      $MEM[j] \leftarrow (P_S, E)_{SL}$;
    end for
    $MEM[SPC - 1] \leftarrow (P_S, E)_{SH}$;
  end if
end if

omitido ya que el mismo es proporcional a F_l, dejando *OESN-X* y *OEM-2X* como los bloques o módulos más relevantes en el análisis de área.

El área requerida para el bloque *OESN-X*, que ordena N elementos, depende de lo siguiente: N $C\&S$ bloques para la fase de intercambio, mientras que se requiere $\frac{N}{2}$ $C\&S$ bloques para la fase de ordenamiento par y $\frac{N}{2} - 1$ bloques $C\&S$ para la fase impar. Por lo tanto el área puede ser calculada por la siguiente regla:

$$A_{oesn-x} = \frac{N^2 - N}{2} \tag{5.1}$$

El bloque *OEM-2X* implementa un segundo ordenamiento, organizando de menor a mayor una cantidad 2^k de elementos, el numero requerido de bloques $C\&S$ es igual a:

$$A_{oem-2x} = 2(\frac{N^2 - N}{2}) + (2N - 2) \tag{5.2}$$

El hecho de que en la ecuación 5.2 este presente el termino $2 \times N$ es para dar soporte a la última etapa del Odd-Even Merge Network analizada en [101]. Finalmente el área total del sistema es la suma $A_{oesn-x} + A_{oem-2x}$.

En el análisis de tiempo de ejecución, la unidad básica era el tiempo que requiere un bloque $C\&S$, , el cual esta definido por el*path* combinacional del mismo. La frecuencia de reloj del

bloque por lo tanto queda fijada respecto del bloque que más tiempo demore en realizar su tarea, dado que el sistema cuenta con dos bloques de ordenamiento *OESN-X* y *OEM-2X* donde el primero tarda N fases en ordenar N números, mientras que el segundo tarda $N+1$ fases en llevar a cabo su tarea; se deduce que el bloque mas lento del sistema es *OEM-2X* [97]. Desde que éste bloque requiere $N+1$ fases para ordenar $2*N$ elementos y tiene un orden de tiempo de ejecución en cada una de sus unidades de $O(N^2)$, el modelado del tiempo se puede representar por la ecuación 5.3:

$$T_{OEM-2X} = t_{cs} \times (N+1) \tag{5.3}$$

Entonces el tiempo promedio del sistema queda definido por:

$$\mathbf{T}_{sa} = T_{OEM-2X} \times \sum_{i=0}^{i=F_l/N} i \tag{5.4}$$

Donde dicha sumatoria (Ecuación 5.4) se puede expresar en forma polinomial por la ecuación 5.5

$$\sum_{x=p}^{q} x = \frac{(p+q)(q-p+1)}{2} \tag{5.5}$$

Finalmente el tiempo de ejecución de nuestra arquitectura es representado por la ecuación 5.6

$$\mathbf{T}_{sa}(F_l, N) = T_{OEM-2X} \times \frac{(\frac{F_l}{N})^2 + \frac{F_l}{N}}{2} \tag{5.6}$$

Como ejemplo, si fuera nuestra intención ordenar 512 items, con un grado de paralelismo de 8 y mientras que la ecuación 5.3 sea igual a 1, la ecuación 5.6 debería quedar como sigue:

$$\mathbf{T}_{sa}(512, 8) = T_{OEM-2X} \times \frac{(\frac{512}{8})^2 + \frac{512}{8}}{2} = 2080 \tag{5.7}$$

De lo anterior se deduce que el tiempo para ordenar 512 elementos con un grado de paralelismo de 8 es igual a 2080 unidades de tiempo.

Un ultimó análisis de la arquitectura propuesta para el *Sorter* se puede apreciar en las Figuras 5.7 y 5.6, en donde se observan proyecciones de tamaño ($C\&S$) y tiempo que consumirá el ordenamiento respectivamente comparándose con el valor que tiene nuestro *Sorter* (Sorter) con un teórico óptimo (Th) [102]. Las relación se proyectaron concibiendo que el paralelismo tuviera una relación logarítmica respecto de los datos a ordenar, es decir $n = \log_2(F_l)$.

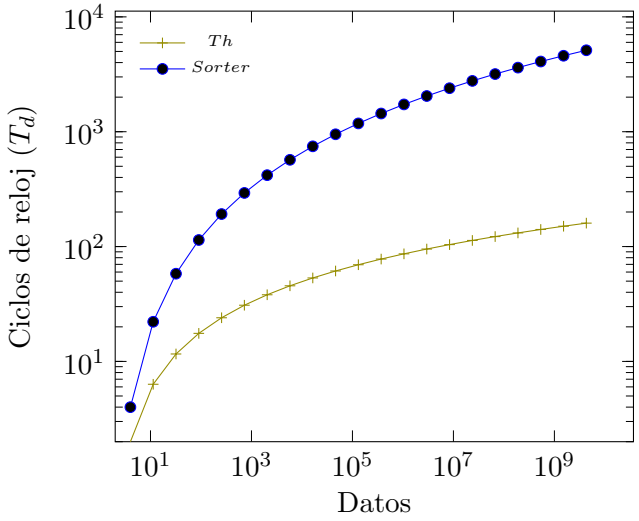


Figura 5.6: Tiempo de ordenamiento.

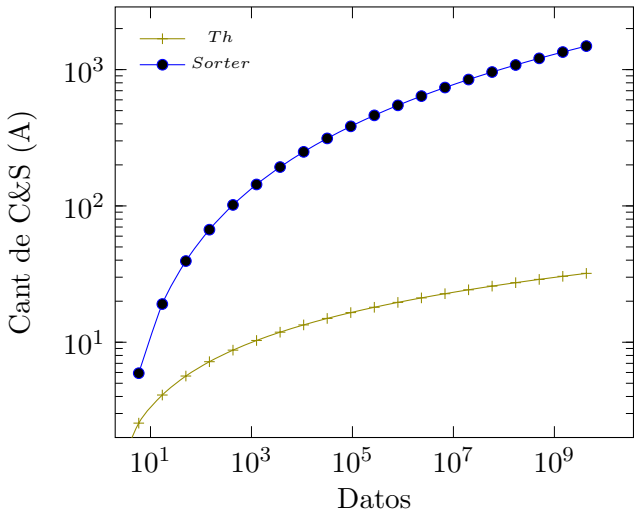


Figura 5.7: Área del *Sorter*.

Sorter en SystemC

En una primera etapa del diseño, la arquitectura del sistema se implemento en *SysemC*. El parámetro que determina el grado de paralelismo afecta directamente el comportamiento de la mayoría de los módulos, por lo tanto se creyó conveniente la implementación de un tipo de dato propio, el cual represente correctamente nuestros deseos.

Para la creación de un tipo datos propio, el mismo debe ser definido como un *struct* o una clase de C++ pero análogamente, se deben definir operaciones tales como: salida de pantalla, asignación, igualdad, como almacenar la información del tipo de dato en formato *wave*, etc.

Posteriormente, cada componente de la arquitectura, *OESN-X*, *OEM-2X* y la memoria interna, fueron definidos como módulos, utilizando las *MACROs* provistas por *SystemC*. Cada módulo debió tener sus respectivos puertos para recibir y enviar datos, según corresponda. Dado que el diseño, del *Sorter* se realizo desde cero, se buscó darle importancia a parámetros tales como el grado de paralelismo, el cual fue representado en un tipo de datos propio de C++, el mismo se puede observar en el Código 6.

Dado que para la representación interna de los datos se utilizó un *struct* (mencionado anteriormente), y ya que el mismo contenía un arreglo de tamaño dinámico se debió realizar la siguiente tarea. Debido a que *SystemC* en su etapa de elaboración (Sección 3.2.3) se encarga de almacenar en memoria, siempre y cuando sean tipos base, el tamaño de las estructuras que el programa/sistema va a utilizar; es necesario indicarle como almacenar un arreglo dinámico. En el capítulo 13 [103] del libro *SystemC for ground up*, se indica como debe realizarse esta tarea. Fue necesario indicar el canal base que se utilizó, la interfaz y los métodos básicos de comunicación para que *SystemC* en su etapa de elaboración pudiera comprender como interpretar nuestro tipo de dato. En el Apéndice en los Códigos 7 y 8 se observa la implementación de nuestro propio canal de comunicación.

Finalizado el diseño interno, se procedió a la elaboración del módulo *top*, el cual contiene puertos de entrada y salida *SC_IN* y *SC_OUT*, siendo éstos puertos especializados de *SystemC*. Éstos transmiten datos de tipo *bool*, para así recibir señales tales como: clock, reset y de sincronización; finalmente se utilizaron otros puertos especializados como lo son: textitSC_FIFO_IN y *SC_FIFO_OUT*, para dar soporte tanto al ingreso como egreso de datos.

El control del sistema se realizo mediante *SC_THREADs* siguiendo los pasos de método de ordenamiento iterativo mencionado en 1, la sincronización del control fue llevada a cabo mediante sentencias *wait* provistas por *SystemC*.

Particularmente se aprovecharon las bondades de C++ para implementar funcionalidades tales como el ordenamiento de los datos, dado que a éste nivel del diseño no necesariamente se deben implementar las *Sorting Networks*, se procedió a implementar esta funcionalidad mediante

la llamada a la función de ordenamiento provista por C++, *sort()*.

Sorter en VHDL

Finalmente, luego de realizar y analizar la problemática planteada de ordenamiento en *SystemC*, se procedió a implementar el bloque en un lenguaje sintetizable: *VHDL*. El desarrollo del mismo se realizo mediante la instanciación de distintos bloques *C&S*, memoria y un control. Se añadieron señales de control con el exterior indicando cuando un dato de entrada es valido, cuando un dato de salida es valida y cuando el bloque tiene necesidades tanto de recibir datos como de entregar datos; el objetivo fue el de facilitar la interacción con otros bloques.

El hecho de utilizar *VHDL* para la descripción del módulo fue beneficioso ya que se pudo generar de forma sencilla el manejo de datos empaquetados; esto no hubiera sido posible si se realizaba en *Verilog* dado que este lenguaje no proporciona la posibilidad de definir tipos de datos propios como arreglos de arreglos.

En la Figura 5.8 se puede apreciar una parte del diseño, en la misma se observan los dos bloques internos que ordenan tanto X datos como 2X datos, en la Figura 5.9 se aprecia el esquemático de la *Odd-Even transposition Sorting Network* descripta anteriormente (Figura 5.4), se pueden apreciar los bloques mínimos (*C&S*) y su conexionado y organización.

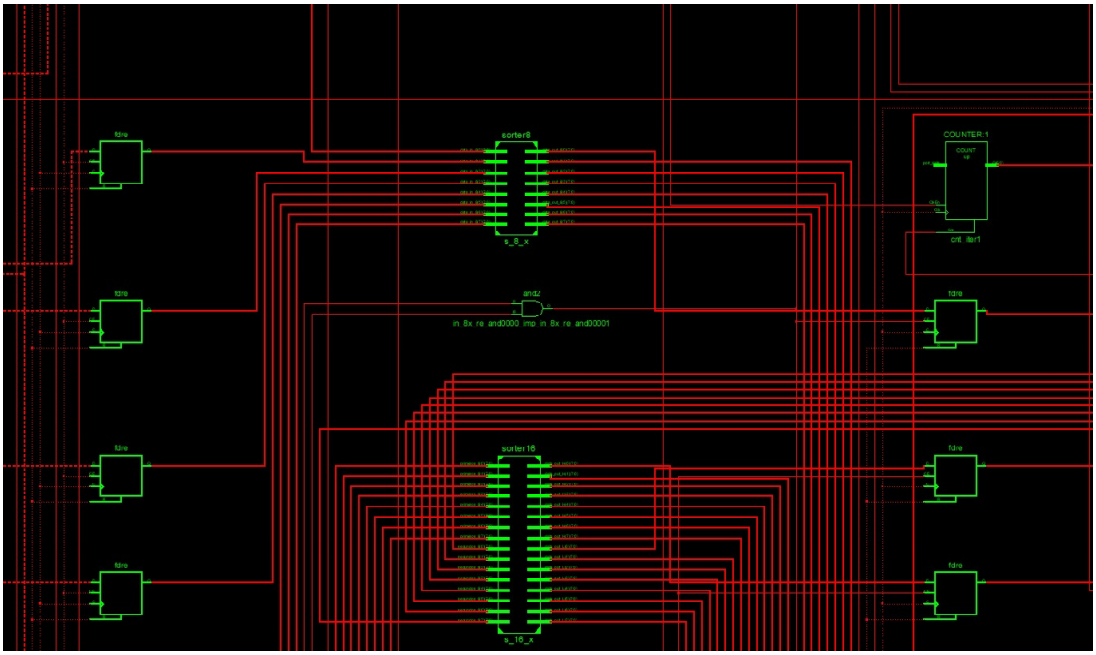


Figura 5.8: Esquemático del módulo Sorter.

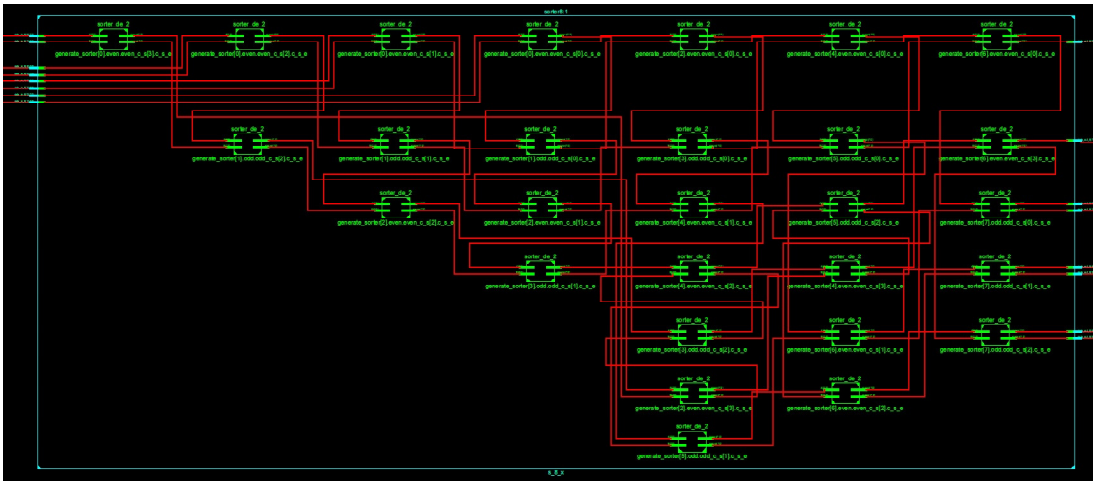


Figura 5.9: Esquemático de Parsing Sorting Network.

5.2.2. ETH

El modulo de *ETH* es el encargado de comunicar al sistema con el exterior, recibir y enviar datos; el mismo fue generado con la herramienta *CoreGen* de *Xilinx* [104], el cual permite crear un bloque IP *ETH* de forma simple y rápida,. En la Figura 5.10 se aprecia el diagrama en bloques del modulo.

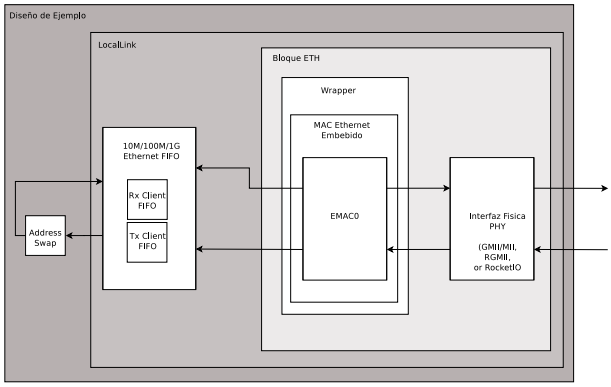


Figura 5.10: ETH Block.

El modelo seleccionado fue *Virtex-5 FPGA embedded Tri-Mode Ethernet MAC Wrapper v1.7*, el cual provee un un bloque con una o dos *Ethernet MAC* [105], que es el responsable del correcto manejo de los *Frames* y la detección de errores sobre éstos. La *MAC* es independiente de si el bloque puede o no conectarse a la interfaz física. En nuestro caso particular hemos utilizado la opción de que el sistema solo tenga una *MAC*, con el fin de simplicar la complejidad del sistema.

El *CoreGen* da la posibilidad de configurar la interfaz física, *PHY*, con soporte para *MII*, *GMII*, *RGMII*, *SGMII* y *1000BASE-X PCS/PMA*. La interfaz física seleccionada fue *GMII/MII*, ésta se encuentra definida en la clausula 22 de [105]. Es una interface paralela que conecta a 10/100 Mbps a las sub capas físicas; dado que dicha velocidad de conexión es más que suficiente para nuestra aplicación, se decidió por utilizar ésta por sobre otras. Finalmente, lo último provisto por la herramienta de generación de bloques de *Xilinx* es el código RTL del bloque, el cual puede ser *Verilog* o *VHDL*, además de un pequeño entorno de *testeo*, para realizar una primera comprobación del funcionamiento del modulo.

El entorno de testeo fue utilizado para analizar el comportamiento del bloque, en primeras instancia; para luego ser reemplazado por un entorno de *SystemC*

5.2.3. DMA

El modulo DMA fue descargado del sitio de *OpenCores* [106], el mismo provee al sistema de la funcionalidad necesaria para realizar la transferencia de datos entre distintos dispositivos; para esto provee de un bus que implementa el protocolo *AMBA AHB-Lite* [107]. Este tiene además las siguientes características: un único canal de comunicación, bit de interrupción, puerto AHB dual *core* con el fin de dar soporte a escrituras y lecturas simultaneas, datos AHB de 32 bits (*word*), Watchdog timer, etc.

Dada las características constructivas del dispositivo, el mismo debe ser configurado para su correcto funcionamiento [108], ésta se realizo mediante el puerto APB [109] que provee el *core*; en la sección siguiente se encontrará detallada la secuencia de configuración.

Dado que el bloque no da soporte a periféricos mediante el *Peripheral Control* como lo indica en la documentación, para llevar a cabo la transferencia de datos entre distintos dispositivos, se procedió a multiplexar las salidas del *DMA* hacia el periférico adecuado.

Otra característica a tener en cuenta, es la descripta en la documentación como *Endiannes byte swapping*, que permite al canal de escritura de datos hacia el exterior soportar *little/big endian*. Por lo tanto las direcciones de lectura y de escritura de los distintos dispositivos tienen que estar alineadas respecto del tamaño del *word* del dispositivo (32 bits en nuestro caso).

Capítulo 6

Análisis y Resultados

En este capítulo se analizan los resultados obtenidos, tanto a nivel de TLM como de HDL, pudiendo ser este último *SystemC* puro, *VHDL* o *Verilog*.

6.1. Verificación a Nivel TLM

La verificación a nivel de TLM o *Modelo de Transacciones* se realizó de dos maneras, la primera basada en simulaciones mientras que la segunda se basa en métodos formales.

Para el primer tipo de verificación se realizo la codificación de los módulos del sistema, siendo estos *Master*, *Slaves* y *Master/Slave*. Se siguieron las reglas de codificación enunciadas en [13,15,87], a fin de obtener una temprana implementación ejecutable del sistema. La evaluación a este nivel consto del análisis del flujo de datos, el ver como los datos se desplazan por el sistema; para ello se analizó tanto la escritura como la lectura de datos que los módulos realizaban. A continuación se detalla el flujo de los datos a través del sistema.

1. Escritura de datos: El módulo ETH lee los datos y los envía al BUS para que sean escritos. El *Sorter* hace un pedido de datos a ordenar, por lo tanto el BUS debe leer datos de la memoria y escribirlos en el *Sorter*. Cuando el *Sorter* finaliza el ordenamiento de los datos, debe de guardarlos en la memoria, por lo tanto se los envía al BUS para que éste finalmente los guarde en la memoria del sistema.

2. Lectura de datos: El módulo ETH recibe un pedido de lectura, lo envía al BUS; éste debe leer los datos internos del sistema (memoria), entregárselos al ETH para que finalmente sean devueltos al exterior como respuesta al pedido de lectura.

En los códigos 1, 2 y 3 del Apéndice, se aprecia la implementación del módulo ETH (*Master*), el módulo BUS (*Árbitro*) y el módulo de Memoria (*Slave*). En el módulo ETH, se aprecia el

initiator_socket, componente por el cual se realiza la comunicación y un procedimiento el cuál es el encargado de generar el *generic_payload*. En el módulo Memoria a diferencia del anterior, se aprecia un *target_socket* dado que la memoria únicamente responde a las necesidades de los paquetes entrantes, ella por su naturaleza nunca va a iniciar una transacción. Finalmente el el módulo del BUS se puede observar la convivencia entre *initiator_sockets* y *target_sockets* y un procedimiento que se encarga de arbitrar. El tipo de datos manejado internamente en el sistema es *int* de $C++$.

Una vez realizada la codificación y su posterior compilación, se obtuvo una implementación ejecutable del sistema. En el código 5 (Apéndice) se aprecia el comportamiento del sistema ante una primera escritura de datos y una posterior lectura de éstos desde el exterior siendo el comportamiento el esperado.

El segundo tipo de verificación realizado a nivel de TLM es mediante métodos formales, se debió considerar lo descripto anteriormente (Sección 4.4.2), lo analizado en [89] dado que es donde se describe explícitamente como debe ser modelado el *Scheduler* y el *Clock* de SystemC en PROMELA y las características de una llamada bloqueando.

Según [15] una llamada bloqueante es integrada al *core* de TLM para dar soporte al estilo de codificación *Lossely-timed*, el uso de llamadas bloqueantes es ideal para cuando se desea que un módulo iniciador de transacciones finalice ésta de una formal sencilla, esto quiere decir una única llamada a función en el módulo receptor de la transacción. El método utilizado es *b_transport* y cuenta de dos argumentos, el primero es la transacción mientras que el segundo es una denotación de tiempo. En el Código 6.1 se observa la definición del método.

```
namespace tlm {
  template <typename TRANS = tlm_generic_payload>
  class tlm_blocking_transport_if : public virtual sc_core::sc_interface {
    public:
      virtual void b_transport(TRANS& trans, sc_core::sc_time& t) = 0;
  };
} // namespace tlm
```

Código 6.1: Definición b_transport.

Adicionalmente, cuando se utiliza una llamada bloqueante en TLM se deben cumplir ciertas reglas, algunas son:

1. El método *b_transport* debe de llamar directa o indirectamente a una sentencia *wait*.
2. El método *b_transport* no debe ser llamado desde un *SC_METHOD*.
3. El módulo iniciador debe reutilizar el objeto transacción a través de las interfaces, DMI e interfaces de debug.

4. La llamada al método *b_transport* indica el tiempo de inicio de una transacción, mientras que su finalización denota el final de la misma.

5. El argumento temporal, permite generar un *offset* del tiempo del método respecto del tiempo de simulación.

6. El módulo receptor de la llamada debe modificar el objeto *TRANS*.

Basándonos principalmente en el análisis realizado en [89] y conociendo como es el comportamiento de una llamada bloqueante en *SystemC* se procedió a codificar dichas llamadas como una conjunción de los elementos observados en la Tabla 4.1. Como es mencionado en [89], al realizar la traducción de la semántica de *SystemC* a PROMELA, se debe considerar lo siguiente:

- El *Scheduler* es *non-preemptive*, se debe asegurar que si un proceso se encuentra corriendo, éste no debe ser interrumpido al menos que él mismo lo solicite de forma explicita mediante una sentencia *wait* o un evento.

- El *Scheduler* de *SystemC* debe asegurar que el tiempo avance solo cuando no se puede elegir ningún proceso de la bolsa de procesos.

- La comunicación entre procesos debe ser mediante llamado de funciones, el llamado de funciones debe ser utilizado tanto para interrupciones como para transacciones.

Análogamente situaciones como: comportamiento del *Scheduler*, comportamiento del *Clock* y el manejo de las llamadas a funciones también fue considerado. Para la primer situación una variable compartida M fue utilizada, si $M > 0$ indica que un proceso se esta ejecutando, cuando $M == 0$ se indica que el *Scheduler* tiene libertad para elegir que proceso se va a ejecutar. El *Clock* del sistema fue considerado con un proceso que se encarga de actualizarlo, el mismo se puede ver con el nombre de *update_clock* en el Código 4 del Apéndice. Por último, para el llamado de las funciones se utilizo una variable compartida entre el proceso que describe el comportamiento del módulo y el proceso que describe el funcionamiento de la función.

Una propiedad interesante de verificar es el asegurar que ningún proceso corra "eternamente", esto se hace con la propiedad $[] <> (M == 0)$ expresada en LTL. Otra propiedad es la de analizar si el *Clock* modelado en PROMELA avance acorde a las necesidades de los módulos, esto se logra con la propiedad $[] <> (enabled(update_clock))$.

El código del sistema en PROMELA del sistema completo se puede observar en el Código 4 del Apéndice. Inicialmente el sistema descubrió un error de codificación, ya que se analizaba una situación que nunca ocurriría. En la Tabla 6.1 se aprecian los resultados otorgados por el *Model Checker*, en cada una de las filas se puede observar la propiedad que se verifico, los estados que se generaron para verificarla, el tiempo insumido y finalmente el resultado de la misma.

#	Propiedad	Estados	Tiempo[s]	Resultado
1	Safety	64	$< 0{,}1$	Éxito
2	Acceptance	64	$< 0{,}1$	Éxito
3	Non-Progress	425	$< 0{,}1$	Éxito
4	$[] <> (M == 0)$	109	$< 0{,}1$	Éxito
5	$[] <> (time_enabled)$	120	$< 0{,}1$	Éxito

Tabla 6.1: Resultados de Spin.

6.2. Verificación a Nivel RTL

Debido a que una gran mayoría de los bloques utilizados en el sistema son de terceros, estos ya se encontraban implementados en un lenguaje de descripción de hardware (Verilog o VHDL), por lo tanto no fue necesario realizar una primera implementación de los módulos en *SystemC cycle-accurate*. Para la verificación de los bloques del sistema que fueron diseñados por terceros se utilizo el entorno de verificación que se describió en la Sección 4.4, dado que es necesario insertar los bloques desarrollados por terceros dentro del entorno, se debe de implementar la interfaz que realice la comunicación *SystemC-Verilog* o *SystemC-VHDL*.

La interfaz que comunica el *DUV* descripto en VHDL o Verilog y el entorno descripto en *SystemC* se implementa mediante una clase privativa propia del entorno de simulación, esto es decir, que la clase varia si se utiliza VCS [62] de *Synopsys* o ModelSim [61] de *Mentor Graphics*. En el primer caso la interfaz entre *SystemC-VHDL* o *SystemC-Verilog* se realiza mediante una opción al momento de compilar el módulo en VHDL o Verilog que se desea estimular, esta opción es llamar al compilador (*vlogan*) con el parámetro *sc_model* [110]; esto genera internamente la interfaz y se vuelve transparente para el usuario. Mientras que en el segundo caso, utilizando la herramienta de *Metor Graphics* es necesario genera la interfaz de forma explicita generando una clase que herede de la clase *sc_foreign_module* y realizar el conexionado (asignación de señales) de forma manual.

Finalmente con la interfaz entre el *DUV* y el entorno de verificación definida e implementada, se procedió a realizar el correcto estimulo del *DUV*. En el caso de los bloques desarrollados por terceros se busco obtener un funcionamiento acorde al de las especificaciones, mientras que en los desarrollos propios de los bloques en primera instancia se realizó un modelo en *SystemC cycle-accurate* para luego realizar la descripción del bloque en VHDL o Verilog.

6.2.1. Sorter

Mientras que el estándar actual de *SystemC* puede ser usado para realizar una verificación básica sobre algún *DUV* la biblioteca SCV[1] provee capacidades de realizar verificación basada en transacciones, *constraints randomization*, manejo de excepciones, etc. Utilizando la introspección de datos, se puede lograr manipular éstos de forma consistente, ya sean datos nativos de C/C++, *SystemC* o los definidos por el usuario. Adicionalmente dicha biblioteca permite realizar *recording* de transacciones, *constraints* variables y *randomization*.

Por los motivos mencionados anteriormente, utilizaremos SCV para realizar la verificación de nuestro diseño; esto será una primera aproximación a la realización de verificación de forma nativa. La biblioteca de verificación (SCV) utiliza la introspección para permitir la manipulación de tipos de datos arbitrarios. Lo que permite a diferentes funciones extraer información de objetos, ya sean estos de propios de C/C++, *SystemC* o especificados por el usuario.

Aprovechando la posibilidad de utilizar *templates* la SCV mapea dichos tipos de datos utilizando una interfaz abstracta, esta es *scv_extensions_if* y a través de ésta se pueden realizar las siguientes operaciones:

- Extracción de información.
- Asignación de valores a los atributos del objeto.
- Randomización.
- *Callback Registration*.

Utilizando la introspección de datos, un objeto puede ser manipulado sin denotar su tipo de dato en tiempo de compilación. Ésta facilidad se logra comprender como una analogía de las PLI[2] de *Verilog*.

Para llevar a cabo la introspección además de la interfaz mencionada anteriormente, SCV provee los siguientes *templates*.

- scv_extensions: Este template los objetos son extendidos para soportar la *scv_exntesions_if* y así poder ser manipulados.
- scv_shared_ptr: Permite compartir información respeto de los objetos entre multiples *threads*.
- scv_smart_ptr: Combina *scv_extensions* y *scv_shared_ptr* para implementar extensiones dinámicas como la randomización, *constraints*, etc.

[1] SystemC Verification Library
[2] Program Language Interface

En nuestro caso particular se utilizaron las scv_extensions para realizar la randomización de los datos, esto se puede apreciar en el Código 12 del Apéndice. En el trabajo de Norris lp [111] se pueden observar ejemplos y modos de utilizar la SCV.

Las aserciones, como previamente se menciono, proveen un mecanismo para controlar si el sistema se comporta de la manera pensada. Este comportamiento se puede extraer de las especificaciones del diseño (caso actual) o se pueden deducir del código (caso actual); además la aserción tiene la posibilidad de indicarle al usuario si el comportamiento no es el adecuado.

SystemC provee manejo de aserciones de manera nativa, que se realiza con la sentencia ***sc_assert*** y que puede indicar al usuario el fallo en la satisfacción del comportamiento, de distintas maneras:

- SC_REPORT_INFO: Indica de que la aserción se disparó y la ejecución continua.
- SC_REPORT_WARNING: Da información respecto del archivo y el proceso en que la aserción se disparó y la ejecución continua.
- SC_REPORT_ERROR: Da información respecto del archivo y el proceso en que la aserción se disparó y la ejecución es finalizada.
- SC_REPORT_FATAL: Da información respecto del archivo y el proceso en que la aserción se disparó y la ejecución es finalizada y se genera un archivo *core*.

Adicionalmente SCV soporta el grabado de transacciones, a partir de este se puede realizar una análisis de la información grabada, este análisis podría otorgar resultados similares a las opciones que *SystemVerilog* (SV) provee para cobertura. Un conjunto de macros son utilizados por SCV para realizar el grabado manual de las transacciones, éstos son:

- scv_tr_db: Crea la base de datos en donde las transacciones se van a grabar.
- scv_tr_stream: *Stream* de transacciones, el mismo consiste en un conjunto de transacciones.
- scv_tr_generator: Generador de un determinado tipo de transacción.

Análogamente a lo visto en las anteriores secciones, SCV provee herramientas para poder generar datos tanto aleatorios como con *constraints*. Esto se logra mediante el uso de la introspección de datos.

SCV provee la clase *scv_random* a modo de reemplazo de las funciones *rand()* y *srand()* de C. Esta clase utiliza el paradigma orientado a objetos para ser utilizada en distintos modelos. Es decir que pueden ser generados a través de un *seed* especifico o utilizar el *seed default*, éste ultimo

genera datos siguiente el algoritmo jrand48(), un ejemplo sería el que se ve en el código 9 (Apéndice), el mismo fue descargado de [112]. También se proveen clases para aplicarle *constraints* a los datos randomizados. Un *constraint* es una clase derivada de *scv_constraint_base*, en donde los atributos que se desean *randomizar* se especifican con el *template scv_smart_ptr*; mediante este *template* se pueden utilizar distintos operadores para generar los *constraints* deseados, éstos operadores son:

- Operadores Aritméticos: +, -, *
- Operadores Relacionales: ==, !=, ¿, ¿=, ¡, ¡=
- Operadores Lógicos: !, &&, ||

Un ejemplo de *constraints* aplicados sería el que se observa en el Código 14 del Apéndice.

Para realizar los test dirigidos se procedió a la modificación de la línea 23 del código, ésta por default comprende valores positivos (código 10 en Apéndice), obteniéndose los resultados observados en la Figura 6.1, en donde en el eje X se aprecia el tiempo de simulación mientras que en el eje Y se aprecia los componentes ordenados en la memoria. Posteriormente se probaron con valores negativos (código 11 en Apéndice), valores ascendentes (código 12 en Apéndice) y finalmente se procedió a estimular el DUV con valores descendentes (código 13 en Apéndice). El comportamiento observado en las figuras se puede apreciar todas las Figuras anteriormente mencionadas se aprecia el comportamiento del Sorter, observándose los accesos a memoria, se observa un regimen de acceso a memoria de orden cuadrático éste es similar al descripto en la Sección 5.2.1.

Una vez realizado los test con distintos tipos de valores, se decidió utilizar randomización en conjunción con *constraints* para ver como respondía el DUV. El archivo de *constraints* es el que se observa en el Código 14 (Apéndice), para la selección del rango de valores a generar se realizo el siguiente análisis.

Dado que el sistema trabaja con variables declaradas como *int* y el rango de valores de un *int* va desde $(-2^{15}+1)$ hasta $(2^{15}-1)$ (Figura 6.2) se puede identificar tres grandes regiones de valores validas: región de valores negativos, región de valores positivos y el cero; por lo tanto un rango que abarque elementos de cada una de las regiones es un caso de prueba interesante de analizar. Del análisis anterior, se selecciono que el rango de prueba sea -999 hasta 999. El DUV fue estimulado con los nuevos valores y se observo que el mismo respondió correctamente.

Una vez realizada las pruebas de los distintos test y viendo que estas eran superadas satisfactoriamente, se procedió a estimular el DUV con una trama de mil*frames* utilizando el test del Código 14 (Apéndice) intentando de esta forma simular un contexto más cercano a la realidad para analizar el desempeño del Sorter.

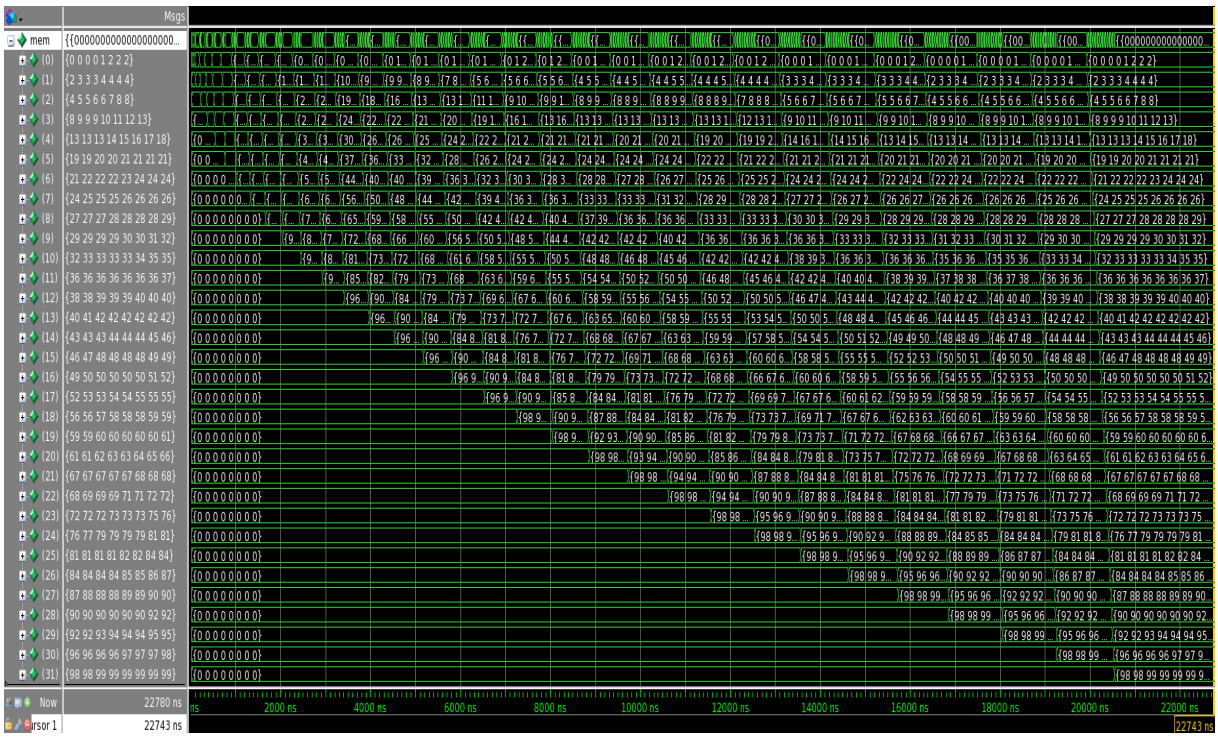


Figura 6.1: Memoria, datos positivos.

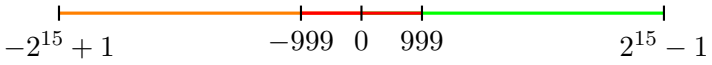


Figura 6.2: Rango de valores.

Posteriormente se realizaron pruebas variando el grado de paralelismo del bloque, haciendo que éste creciera en potencia de 2, por lo tanto en las Figuras 6.1 y 6.3 se puede observar el grado de paralelismo de 2^3 y 2^4, y la correcta respuesta del sistema. En la Tabla 6.2 se observa un resumen de los test realizados.

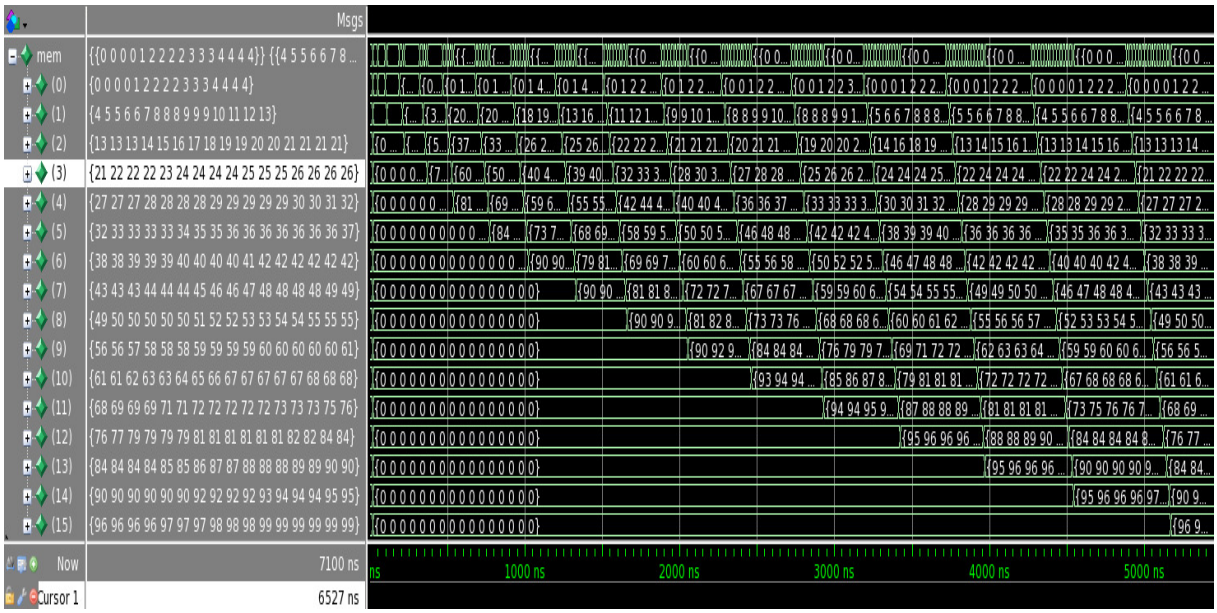


Figura 6.3: Grado de paralelismo 16.

6.2.2. ETH

Para llevar a cabo la verificación del bloque *ETH*. En primera instancia se utilizo un *testbench* provisto por *Xilinx*, observando el correcto funcionamiento del módulo a nivel de simulaciones. Posteriormente, se buscó realizar la síntesis lógica. Una vez lograda ésta, se procedió a generar módulos que permitan la estimulación del bloque en la FPGA. Estos módulos son programas en C utilizan sockets de tipo *RAW*, ya que de esta forma las acciones ocurren en la segunda capa del módulo OSI de redes y se pueden obviar protocolos como TCP o UDP. En la Figura 6.4 se puede apreciar un entorno similar al utilizado para testear el bloque *ETH* en la FPGA, en donde se observan los archivos que contienen lso datos de entrada al bloque y los datos de salidas, ambos

#	Test	Grado de Paralelismo	Frames	Resultado
1	Valores Positivos	2^3	1	Éxito
2	Valores Negativos	2^3	1	Éxito
3	Valores Ascendentes	2^3	1	Éxito
4	Valores descendentes	2^3	1	Éxito
5	Valores Random	2^3	1	Éxito
6	Valores Positivos	2^3	1000	Éxito
7	Valores Negativos	2^3	1000	Éxito
8	Valores Ascendentes	2^3	1000	Éxito
9	Valores descendentes	2^3	1000	Éxito
10	Valores Random	2^3	1000	Éxito
11	Valores Positivos	2^3	1	Éxito
12	Valores Positivos	2^4	1	Éxito
13	Valores Positivos	2^5	1	Éxito

Tabla 6.2: Test realizados en el Sorter.

archivos fueron generados mediante los códigos en C previamente descriptos y éstos se fueron corridos en una PC; el conexionado entre la FPGA y la PC fue mediante un cable de red.

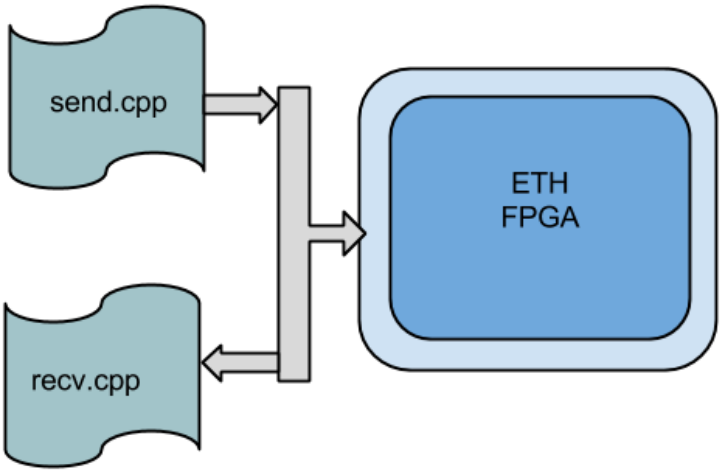


Figura 6.4: Entorno de Test.

6.2.3. DMA

Se procedió a la generación del entorno de verificación reutilizando el que se observa en la Figura 4.8. Adicionalmente se procedió a la implementación de una memoria en *SystemC*, en

primera instancia, a fin de complementar el módulo de DMA con un periférico, observando por lo tanto el comportamiento y la interacción entre estos. El contexto que se planteó era el de analizar el comportamiento del DMA respecto a lecturas y escrituras, desde y hacia un dispositivo, es decir como y en que forma el DMA leía datos de un dispositivo A y los escribía a un dispositivo B, que ocurría si esto fallaba, etc.

Se realizo la configuración del mismo mediante la asignación de valores a registros internos como se observa en la Figura 6.5. A continuación, describiremos paso por paso como se realizo ésta y cuales fueron los valores asignados a los registros *paddr* y *pwdata*.

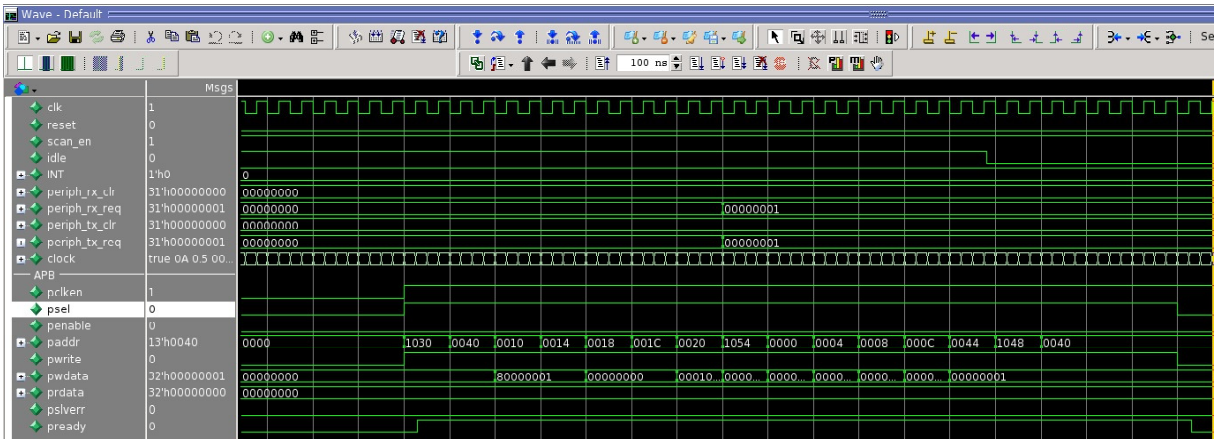


Figura 6.5: Configuración *APB* y registros internos.

Luego de configurado el DMA, se procedió a realizar una transferencia de datos entre dispositivos. En primer caso el dispositivo era el mismo, por lo tanto se leía de una cierta dirección de memoria un dispositivo A para escribir en una segunda dirección de memoria del mismo dispositivo. Una vez realizado el primer test y observándose la respuesta satisfactoria del mismo, se procedió a leer y escribir de la misma dirección de memoria en el mismo dispositivo (A), el resultado fue el esperado ya que se realizaba primero la lectura y posteriormente la escritura obteniéndose finalmente los mismos datos en la memoria. Luego se realizo un test en el cual se procedía a realizar la trasferencia de diez datos con direcciones de lectura y de escritura cercanas (distancia menor a diez) en el mismo dispositivo; se observó un corrimiento de los datos, este comportamiento era esperado. Por último, se planteó la situación en que se realizaba la transferencia en el mismo dispositivo pero este no permitía que se realicen las esctiuras (*Single Port RAM*), este test tubo el objetivo de analizar la profundidad del buffer interno del DMA; el resultado del test fue que los buffers internos tienen una capacidad de almacenamiento de 32 elementos, esto es congruente con lo que se indica en la teoría. Posteriormente se añadió un segundo dispositivo, una memoria en *SystemC*, con el fin de analizar el comportamiento entre

#	Acción	Registro	paddr	pwdata
1	Setear el Core en modo *Independent*.	CORE0_JOINT_MODE[0]	0x1030	0x0
2	Deshabilitar Canal.	CH_ENABLE_REG	0x40	0x0
3.0	Setear datos respecto de la lectura del dispositivo.	STATIC_REG0	0x10	0x80000001
3.1	Setear datos respecto de la escritura del dispositivo.	STATIC_REG1	0x14	0x80000001
3.2	Setear caracteristicas de trabajo en.*BlockMode*	STATIC_REG2	0x18	0x0
3.3	Setear tiempos de espera para lectura y escritura.	STATIC_REG3	0x1C	0x0
4.0	Setear periferico de lectura y de escritura.	STATIC_REG4	0x20	0x00010001
4.1	Setear dirección de lectura.	CMD_REG0	0x0	0x40
4.2	Setear dirección de escritura.	CMD_REG1	0x4	0x20
4.3	Setear tamaño del buffer a transmitir.	CMD_REG2	0x8	0xA
4.4	Setear acción a realizar una vez finalizada la transacción.	CMD_REG3	0xC	0x3
5.0	Iniciar el canal.	CH_START_REG	0x44	0x1
5.1	Iniciar el canal.	CORE0_CHANNEL_START	0x1048	0x1
6	Habilitar Canal	CH_ENABLE_REG	0x40	0x1

Tabla 6.3: Pasos para configurar el DMA.

éstas. Se aplicaron casos similares a los observados en la Tabla 6.4. En la Fig 6.6 se puede apreciar el movimiento de 2 datos entre dos dispositivos (memorias en *Verilog*).

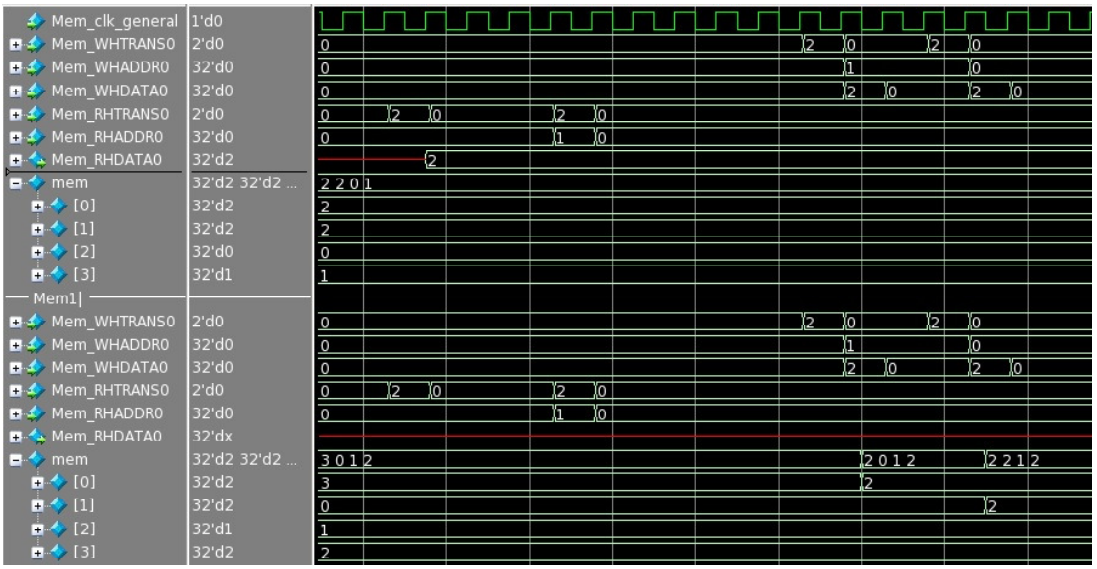


Figura 6.6: Movimiento de datos entre dispositivos.

6.3. Sistema

El análisis del sistema se realizó mediante la reutilización del entorno de verificación, ya que mediante éste situaciones como el control, se resolvieron de forma sencilla.

Dado que el DMA es el bloque principal o la columna vertebral de transmisión del sistema, como se observa en la Figura ; todos los bloques que hacen al sistema debieron de adaptarse para hablar el mismo protocolo que el DMA. Para esto se realizaron adaptaciones a los bloques con el fin de que todos los bloques tengan la misma interface (Figura 6.7), en la Figura previa se puede observar la interfaz que utiliza el DMA, en la misma se aprecian las entradas y las salidas que dan soporte a un subconjunto de funcionalidades del protocolo AHB-AMBA3 [107,108]. Además en la figura se aprecia que hay señales diferentes para escritura (comienzan con w) y para lectura (comienzan con r), logrando de esta forma la escritura y lectura simultanea o paralela, también se aprecian señales identificadoras para indicar cuál es el dispositivo desde donde se realiza la lectura (id_r) y cuál es el dispositivo desde donde se realiza la escritura (id_w).

Un caso particular al momento de adaptar los bloques fue el *Sorter*, dado que el mismo recibe ocho datos en paralelo de 32 bits cada uno. En una primera instancia se había utilizado una de las posibilidades que *VHDL* otorga, el de definir tipos de datos utilizando *arrays*. Debido a

#	Dispositivos	DirR	DirW	BUFS	Observación	Resultado
1	1	0x0	0x0	0xA	-	Éxito
2	1	0x0	0x20	0xA	-	Éxito
3	1	0x0	0x40	0xA	-	Éxito
4	1	0x20	0x0	0xA	-	Éxito
5	1	0x40	0x0	0xA	-	Éxito
6	1	0x0	0x0	0xFF	-	Éxito
7	1	0x0	0x20	0xFF	Debido a que el buffer es mayor a la distancia entre la dir de lectura y la dir de escritura, se observó una repetición en el patrón de datos.	Éxito.
8	1	0x0	0x40	0xFF	Debido a que el buffer es mayor a la distancia entre la dir de lectura y la dir de escritura, se observó una repetición en el patrón de datos.	Éxito
9	1	0x20	0x0	0xFF	-	Éxito
10	1	0x40	0x0	0xFF	-	Éxito
11	2	0x0	0x0	0xA	-	Éxito
12	2	0x0	0x20	0xA	-	Éxito
13	2	0x0	0x40	0xA	-	Éxito
14	2	0x20	0x0	0xA	-	Éxito
15	2	0x40	0x0	0xA	-	Éxito
16	2	0x0	0x0	0xFF	-	Éxito
17	2	0x0	0x20	0xFF	Debido a que el buffer es mayor a la distancia entre la dir de lectura y la dir de escritura, se observó una repetición en el patrón de datos.	Éxito.
18	2	0x0	0x40	0xFF	Debido a que el buffer es mayor a la distancia entre la dir de lectura y la dir de escritura, se observó una repetición en el patrón de datos.	Éxito
19	2	0x20	0x0	0xFF	-	Éxito
20	2	0x40	0x0	0xFF	-	Éxito

Tabla 6.4: Casos de Prueba DMA.

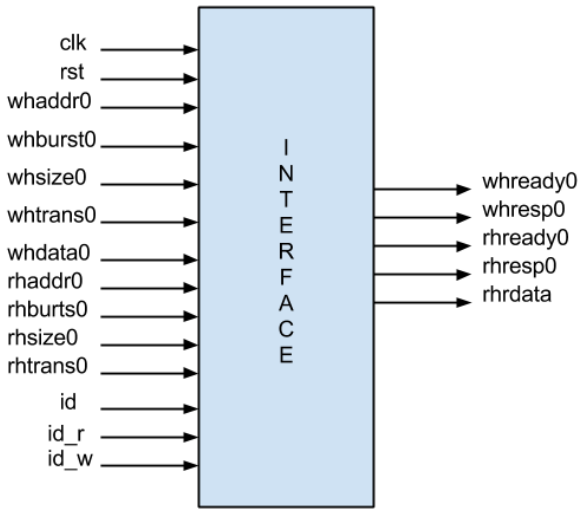


Figura 6.7: Interface DMA.

que el DMA esta descripto en *Verilog*, y *Verilog* no soporta tipo de datos *customizados* se debió realizar una interface que reciba como entrada un *array* de bits con una cardinalidad igual a $word_{size} * parallel_{grade}$ y posteriormente adaptarla al modo en que el *Sorter* recibe los datos. Además por problemas de sincronización, dado que el *Sorter* debe de recibir datos en paralelos, se utilizaron FIFOS, como buffers intermedio para sincronizar. En la Figura 6.8 se observa la situación previamente comentada, apreciándose las fifos que se encargan de guardar los datos que posteriormente entraran al *Sorter* y el wrapper en *Verilog* el cuál permite la conexión entre el *Sorter* y el *DMA*.

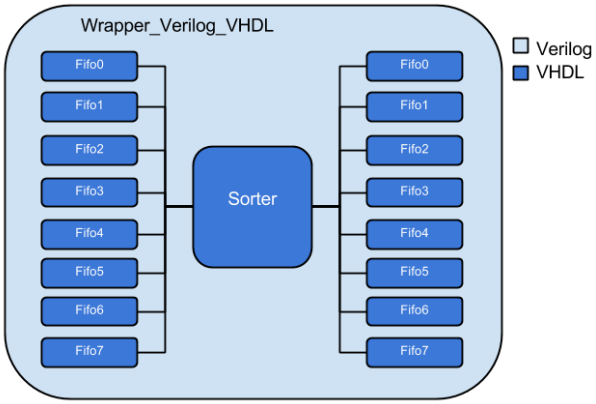


Figura 6.8: Wrapper Sorter.

Finalmente los bloques se conectaron utilizando *SystemC*, debido a que en este contexto es

#	Caso	Dispositivo Lectura	Dispositivo Escritura	Resultado	Obs.
1	`Figura 4.4`	`ETH`	`Mem`	Éxito	-
2	`Figura 4.5`	`Mem`	`Sorter`	Éxito	-
3	`Figura 4.6`	`Sorter`	`Mem`	Éxito	-
4	`Figura 4.7`	`Mem`	`ETH`	Éxito	-

Tabla 6.5: Casos de prueba del sistema.

más sencillo llevar a cabo el control; se procedió a realizar la verificación del sistema llevándose a cabo los siguientes test (Tabla 6.5).

Adicionalmente, viendo que en los diagramas de secuencia descriptos en la sección 4.3.1 solamente se contemplan interacciones entre dos componentes, se decidió añadir una ultima secuencia de prueba que estimule a todo el sistema, ésta se puede observar en la Figura 6.9. En la Figura 6.10 se observa la transmisión de datos entre distintas Memorias en descriptas en Verilog.

6.4. Resultados Obtenidos

El realizar la verificación de bloques de propiedad intelectual tanto de terceros como propios, forzó que la estrategia de verificación varié. Para los bloques desarrollados por terceros la verificación se centro en analizar el comportamiento de éstos ante distintas situaciones y ver si los mismos respondían acorde a las especificaciones, la verificación realizada a éstos bloques fue puramente funcional, no se analizo casos extremos o situaciones anormales. En el caso del *Sorter* la verificación fue mas exhaustiva, se analizo cómo el bloque respondía a distintos estímulos, se realizó un análisis de complejidad y finalmente se siguió el flujo de desarrollar primero un modelo en *SystemC* para luego llegar al la codificación de un modelo en *VHDL*. En la Tabla 6.6 se sintetizan los test y sus respectivos resultados.

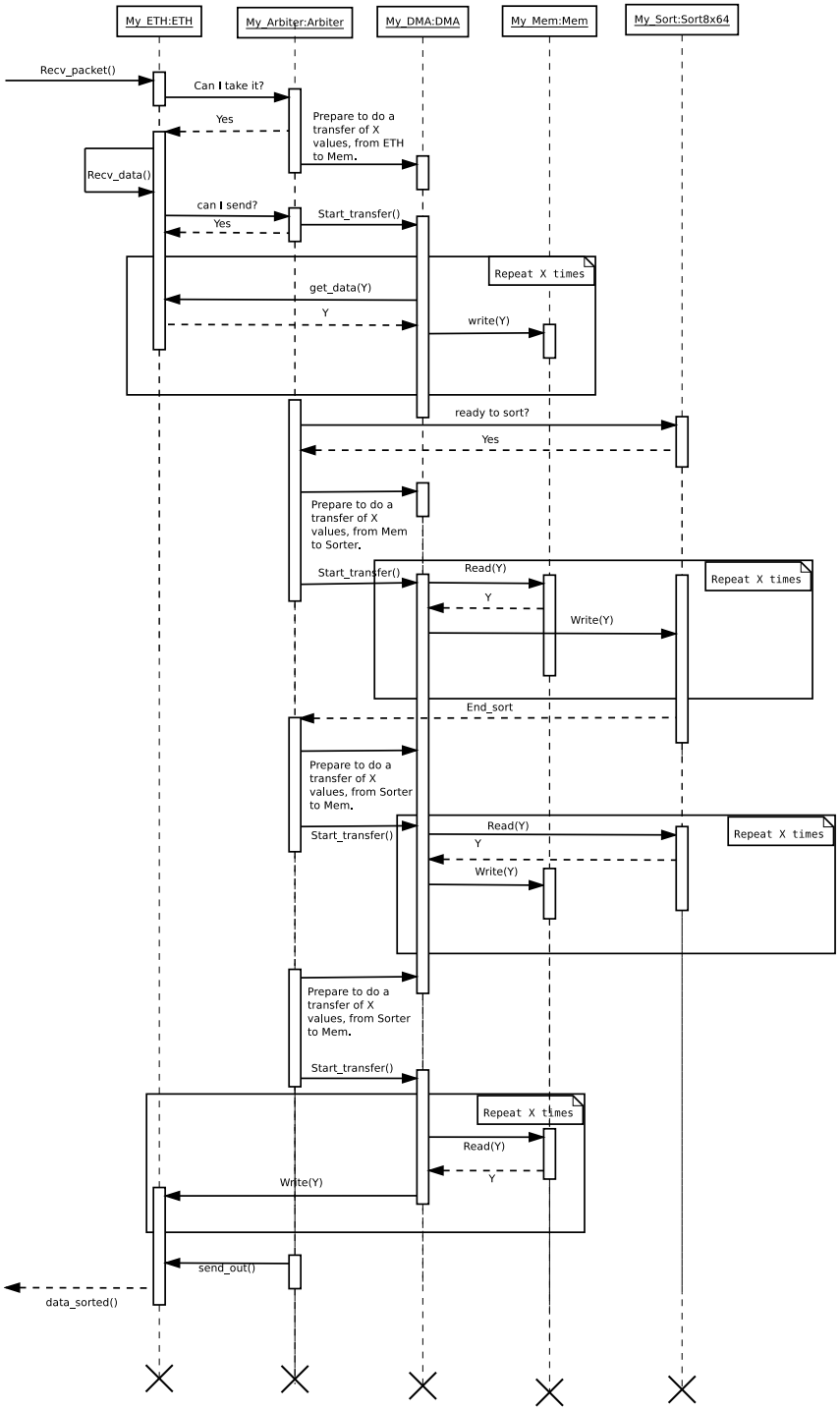


Figura 6.9: Caso adicional.

Nivel	Componente	Test		Resultado
		Sec/Tab	Caso	
TLM	Sistema	Sección 6.1	Item 1	Éxito
			Item 2	Éxito
TLM	Sistema	Tabla 6.1	Caso 1	Éxito
			Caso 2	Éxito
			Caso 3	Éxito
			Caso 4	Éxito
			Caso 5	Éxito
RTL	Sorter	Tabla 6.2	Caso 1	Éxito
			Caso 2	Éxito
			Caso 3	Éxito
			Caso 4	Éxito
			Caso 5	Éxito
			Caso 6	Éxito
			Caso 7	Éxito
			Caso 8	Éxito
			Caso 9	Éxito
			Caso 10	Éxito
			Caso 11	Éxito
			Caso 12	Éxito
			Caso 13	Éxito
RTL	ETH	Sección 6.2.2		Éxito
RTL	DMA	Tabla 6.4	Caso 1	Éxito
			Caso 2	Éxito
			Caso 3	Éxito
			Caso 4	Éxito
			Caso 5	Éxito
			Caso 6	Éxito
			Caso 7	Éxito
			Caso 8	Éxito
			Caso 9	Éxito
			Caso 10	Éxito
			Caso 11	Éxito
			Caso 12	Éxito
			Caso 13	Éxito
			Caso 14	Éxito
			Caso 15	Éxito
			Caso 16	Éxito
			Caso 17	Éxito
			Caso 18	Éxito
			Caso 19	Éxito
			Caso 20	Éxito
RTL	Sistema	Tabla 6.5	Caso 1	Éxito
			Caso 2	Éxito
			Caso 3	Éxito
			Caso 4	Éxito
		Sección 6.3	Figura 6.9	Éxito

Tabla 6.6: Test Realizados.

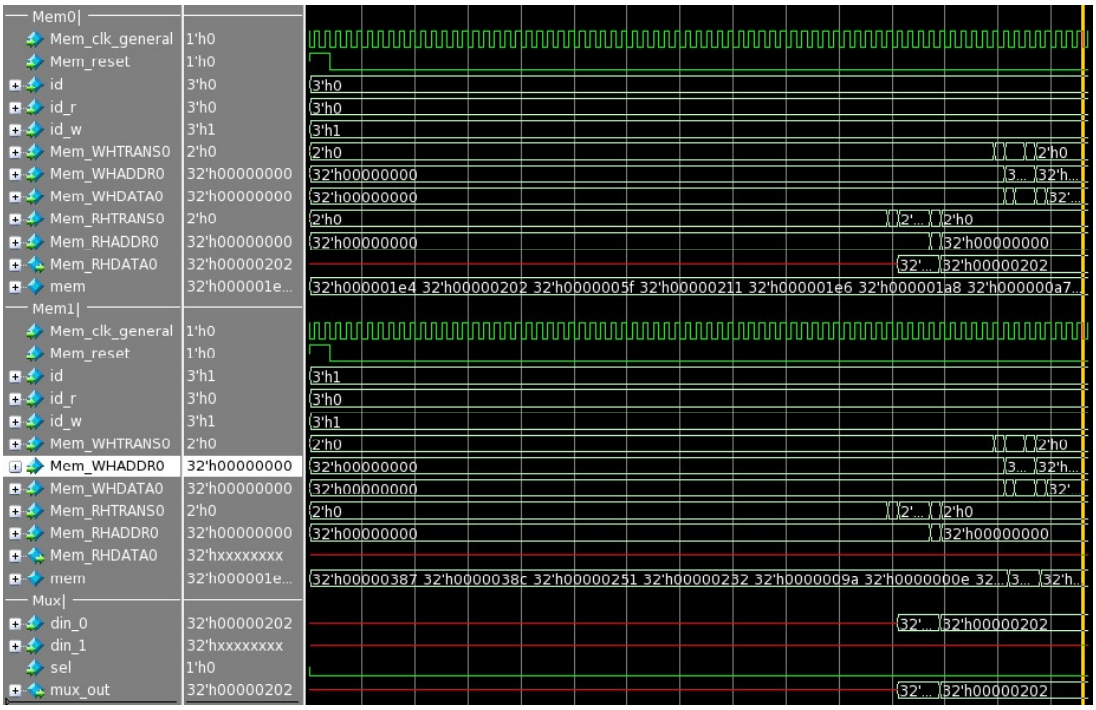


Figura 6.10: Movimiento datos entre memorias.

Capítulo 7

Conclusiones y Trabajo a Futuro

7.1. Resultados Obtenidos

La verificación funcional de sistemas digitales es un tópico de interés hoy en día, tanto para la industria como para la academia. El avance en la tecnología de semiconductores ha proporcionado el escenario necesario para que se alcancen objetivos que hace décadas eran impensados. El hecho de que se estén fabricando semiconductores en 14 nanómetros, es una clara muestra del nivel exponencial de complejidad que los sistemas electrónicos tienen. Adicionalmente, debido a la creciente capacidad que los sistemas tienen para procesamiento, han aparecido distintos protocolos de comunicación que hacen que la coexistencia de los componentes sea dificultosa. En pocas palabras, la complejidad de un sistema digital ha crecido de manera exponencial haciendo que, su tarea de diseño/verificación, incremente su complejidad.

En el ámbito recientemente descripto, se observó que una forma de reducir la complejidad al momento de analizar los sistemas, prototipar, etc; se logra elevando el nivel de abstracción. TLM ha demostrado ser un nivel de abstracción versátil y válido para esta tarea, ya que nos da soporte tanto para situaciones *cycle-accurate* como para situaciones *losely-time*, logrando así explorar en forma completa la arquitectura del sistema digital.

Observando las ventajas que TLM provee ante la problemática actual, se procedió a promover una metodología tanto de diseño como de verificación. En la misma se propuso contemplar la verificación desde un inicio, desde los momentos de especificación del sistema, atravesando distintos niveles de abstracción hasta llegar así al más cercano al hardware, el nivel de HDL. Se utilizaron distintos lenguajes de descripción y de verificación de hardware tales como *Verilog*, *VHDL*, *SystemC*; y distintas bibliotecas que soportan a funcionalidades que los lenguajes tradicionales no contemplaban en una primera instancia, algunos ejemplos son TLM y SCV. Además, en el nivel de transacciones se propuso la utilización de un *Model Checker*, para reali-

zar una verificación con mayor grado de veracidad que la tradicionalmente utilizada basada en simulaciones.

Al aplicarse la metodología a un caso de estudio, un bloque que ordena datos provenientes del exterior (mediante un módulo ETH) a alta velocidad, se observaron diferentes situaciones. La primera y principal, el definir de forma clara, concreta y completa las especificaciones genera una reducción de tiempos, tanto de diseño como de verificación. En el momento de realizar la implementación del sistema a nivel de TLM se abordo distintas concepciones de la arquitectura, las cuales debieron de adecuarse a las necesidades inicialmente planteadas. Se apreciaron situaciones de *deadlock* iniciales las cuales ocurrían por un mal diseño. Además a este nivel de abstracción se procedió a la utilización de un *Model Checker*, más precisamente *Spin* y su versión de java *JSpin*. Se debieron abordar conceptos como es el de describir el sistema mediantes lenguajes de Guardas, dado que se utilizo *PROMELA* y éste utiliza esa concepción. Por ultimo, en la última etapa de la metodología propuesta se genero un entorno de verificación, el cual fue reutilizado para realizar tanto la verificación de bloque como la verificación del sistema en sí. Se observó una reducción de tiempos a la hora de la generación de distintos módulos que estimulen a los bloques/sistema, distintos casos de prueba, etc; dada la reutilización y versatilidad provista por *SystemC* al momento de desarrollar módulos. El poder realizar la interacción entre *SystemC* y *VHDL/Verilog* nos proporciono el ambiente necesario para poder experimentar con diferentes contextos.

A continuación se analizan los resultados obtenidos en comparación con los objetivos inicialmente planteados.

- Objetivo General:
  - El objetivo principal de este trabajo es el estudio, aplicación y desarrollo de tecnologías de verificación de sistemas digitales. Más precisamente, se trabajará con la metodología de verificación/diseño denominada "Transaction Level Modeling" (TLM) o modelado a nivel de transacciones, orientada a la verificación de "*Systems on Chips*" (*SoC*).

    El mismo se considera cumplido por lo expuesto durante el desarrollo de la Tesis, debido a que se ha logrado incorporar de forma adecuada el concepto de transacciones, su abstracción y su implementación en *SystemC* a una metodología de trabajo; esto queda demostrado en los Capítulos 4, 5 y 6.
- Objetivos secundarios:
  - Estudiar el comportamiento de sistemas digitales, ya sean estos simples o complejos y comprender su flujo de diseño/verificación.

Este objetivo se cumplió a lo largo del desarrollo de los primeros capítulos de la tesis, en particular en el Capítulo 4 se aprecia distintas propuestas para el flujo de diseño/verificación y a partir de estas y lo visto en los anteriores capítulos, se logró proponer una metodología que abarca el flujo completo de un sistema digital.

- Estudiar el concepto de verificación, más precisamente, verificación funcional y ver su aplicación en distintos niveles de abstracción.

  Se ha logrado una correcta aproximación a este objetivo, el concepto de verificación fue expuesto en el Capítulo 2 y su aplicación a distintos niveles puede ser observado en los Capítulos 3 y 4.

- Estudiar las bases de TLM, tanto para diseño como para verificación y comprender sus ventajas y desventajas frente a otras opciones.

  Las ventajas y desventajas de TLM han sido expuestas en forma exhaustiva en el Capitulo 3, se ha apreciado las ventajas y desventajas de TLM en el Capítulo 5 en donde se aplican los conceptos a la metodología propuesta.

- Generar una metodología de verificación válida para el flujo de diseño/verificación de un sistema digital, utilizando distintos niveles de abstracción.

  El desarrollo de una metodología propia para llevar a cabo la verificación y diseños de sistemas digitales fue enunciada en el Capítulo 4, aplicada en el Capítulo 5 y sus resultados fueron expuestos en el Capítulo 6; la misma se considera satisfactoria.

Por lo expuesto anteriormente, creemos que la metodología propuesta es sólida y abarca el flujo completo de diseño y verificación de sistemas, y más particularmente, creemos que TLM es el nivel de abstracción adecuado para explorar/analizar/verificar distintas arquitecturas/propuestas de sistemas digitales complejos.

El resultado de esta Tesis se encuentra resumido en el trabajo presentado en la décima edición de la Escuela Argentina de Micro-Nanoelectrónica Tecnología y Aplicaciones (EAMTA) con el nombre de "*SystemC/TLM flow for SoC Design and Verification*".

7.2. Trabajo a Fúturo

Como trabajo a futuro creemos que es necesario introducir en las distintas etapas de la metodología la capacidad de automatización con el fin de minimizar aún más los tiempos tanto de diseño como de verificación de los sistemas. Análogamente, también es importante lograr utilizar la verificación formal para todo el espectro de TLM, dado que en éste trabajo solo se utilizó para las llamadas bloqueantes de sistemas a nivel *Bus-Arbitration Model*. Por último,

un campo que ha ganado relevancia en el último tiempo es la utilización de TLM para para realizar una estimación inicial del *power* del sistema; este enfoque se obtiene utilizando *Unified Power Format [113]* (*UPF*) y TLM, ejemplos de esto es el trabajo realizado por Mbarek [114], el trabajo realizado por Mischkalla [115] y el trabajo realizado por Bouhadiba [116].

Apéndice A: Códigos TLM

Código 1: Código Módulo Master.

```
class Master : public sc_module
{
public:
  // Socket
  simple_initiator_socket<Master> socket;
  // Constructor
  Master(sc_module_name nm);
  // Destructor
  ~Master() {}

private:
  // Data buffer
  int data[SIZE_OF_PACKET*SIZE_OF_FRAME];
  // Thread process
  void thread_process(void);
};

#endif
```

Código 2: Código Módulo de Memoria.

```
class Memory : public sc_module
{
public:
  // Socket
  simple_target_socket<Memory> socket;
  // Constructor
  Memory(sc_module_name nm);
  // Destructor
  ~Memory() {}
private:
```

```
  // Latency for a memory read
  const sc_time LATENCY;
  // Memory array
  int mem[MEM_SIZE];
  // b_transport method
  void b_transport(tlm_generic_payload& trans, sc_time& t);
};
```

Código 3: Código Módulo Bus.

```
class Bus : public sc_module
{
public:
  // Sockets
  simple_target_socket<Bus>*  targ_socket[N_INITIATORS];
  simple_initiator_socket<Bus>* init_socket[N_TARGETS];
  // Constructor
  SC_HAS_PROCESS(Bus);
  Bus(sc_module_name nm) : sc_module(nm)
  {
    // Allocate memory for target sockets
    for (unsigned i=0; i<N_INITIATORS; i++) {
      char txt[20];
      sprintf(txt, "targ_socket_%d", i);
      targ_socket[i] = new simple_target_socket<Bus>(txt);
      targ_socket[i]->register_b_transport(this, &Bus::b_transport);
    }

    for (unsigned i=0; i<N_TARGETS;i++) {
      char txt[20];
      sprintf(txt,"init_socket_%d",i);
      init_socket[i]=new simple_initiator_socket<Bus>(txt);
    }
    for (unsigned i=0;i<NUMBER_OF_SORTERS;i++)
      available_sorter[i]=1;
  }
  // Destructor
  ~Bus()
  {
    for (unsigned int i=0; i<N_INITIATORS; i++)
      delete targ_socket[i];
  }

private:
```

```
35  // Mutex for bus access
    sc_mutex access;
37  int available_sorter[NUMBER_OF_SORTERS];//if there is a 1 in the cell i, the i-
      sorter is available else it is unavailable
    // b_transport
39  void b_transport(tlm_generic_payload& trans, sc_time& t)
    {
41    // Lock bus
      if (trans.get_command()==TLM_READ_COMMAND)
43    {
        access.lock();
45      cout<<sc_time_stamp()<<"Read from memory"<<endl;
        (*init_socket[0])->b_transport(trans,t);
47      access.unlock();
      }
49    else
      {
51      int i=0;
        int flag=0;
53      while (i<NUMBER_OF_SORTERS && flag==0)
        {
55        if (available_sorter[i]==1)
            flag=1;
57        else
            i++;
59      }
        if (flag==1) //there is a available sorter
61      {
          access.lock();
63        available_sorter[i]=0;
          flag=0;
65        cout<<endl<<sc_time_stamp()<<"Send to Sorter["<<i<<"]'s  memory, delay="<<
      t<<endl;
          (*init_socket[i+1])->b_transport(trans,t);
67        wait(t);
          trans.set_command(TLM_READ_COMMAND);
69        cout<<endl<<sc_time_stamp()<<"Read from Sorter["<<i<<"]"<<endl;
          (*init_socket[i+1])->b_transport(trans,t);
71        wait(t);
          trans.set_command(TLM_WRITE_COMMAND);
73        cout<<endl<<sc_time_stamp()<<"Write to memory"<<endl;
          (*init_socket[0])->b_transport(trans,t);
75        wait(t);
          available_sorter[i]=1;
```

```
77          access.unlock();
          }
79        else
          {
81          cout<<sc_time_stamp()<<this->name()<<"There is no available sorter, the
      packet will be stored in memory"<<endl;
            access.lock();
83          trans.set_address(0);
            (*init_socket[0])->b_transport(trans,t);
85          wait(t);
            access.unlock();
87        }
      }
89  }
};
```

Código 4: Código del Sistema en PROMELA.

```
int M=0;
2
#define myturn (M==0 || M==this)
4 #define fail (M<0 || M>8)
#define iteration 1;
6
//loop variable
8 int i=0;

10 //functions
bool t_p=0;
12 byte call_bt=0;
byte b_t=0;
14 byte bus_t
byte call_memory=0;
16 byte call_sorter=0;
byte b_t_sorter=0;
18
#define time_enabled timeout && M==0 && (X_used || BUS_used || Memory_used ||
    Sorter_used)
20
int X=0;
22 bool X_used=0;

24 int BUS=0;
bool BUS_used=0;
```

```
26
int Memory=0;
28 bool Memory_used=0;

30 int Sorter=0;
bool Sorter_used=0;
32


34 active proctype update_clock(){
end_update_clock:
36     do
       :: atomic{time_enabled -> X=X+X_used; BUS=BUS+BUS_used; Memory=Memory+
       Memory_used; Sorter=Sorter+Sorter_used;}
38 progress:
       od
40 }

42 active proctype module1_initiator() {
byte state=1;
44 byte this=1;
end_initiator:
46 do
     :: atomic{myturn && state==1 -> i=0;state=2;M=0;}
48   :: atomic{myturn && state==2 && i<iteration -> t_p=1;state=3; M=2;}
//   :: atomic{myturn && state==3 && t_p==0 && i<iteration -> t_p=1; M=2; }
50   :: atomic{myturn && state==3 && t_p==0 && i>=iteration -> M=this; state=4;}
     :: atomic{myturn && state==4 -> printf("FIN INITIATOR\n"); M=0;}
52 progress_initiator:
od
54 }

56 active proctype thread_process(){
byte state=1;
58 byte this=2;
end_tp:
60 do
     :: atomic{myturn && state==1 && t_p==1 -> printf("Send a request to Memory\n");
        call_bt=1;state=2; M=3;}
62   :: atomic{myturn && state==2 && call_bt==0 ->  state=3; M=this;}
     :: atomic{myturn && state==3 && t_p==1 -> X=0; X_used=1; state=4; M=0;}
64   :: atomic{myturn && state==4 && t_p==1 && X==7 -> X=0; X_used=0; i++; state=1;
        t_p=0;M=1;}
progress_initiator_tp:
66 od
```

```
}

active proctype module2_BUS() {
byte state=1;
byte this=3;
end_bus:
do
  :: atomic{myturn && state==1 && call_bt==1 -> bus_t=1; state=2; M =4; BUS_used
    =1;}
  :: atomic{myturn && state==2 && b_t==0 -> call_bt=0; bus_t=0;state=1; M=2;
    BUS_used=0;}
progress:
od
}

active proctype b_transport_BUS()
{
byte state=1;
byte this=4;
end_bt_bus:
do
  :: atomic{myturn && state==1 && bus_t==1 -> state=2; M=this;}
  :: atomic{myturn && state==2 -> printf("BUS:Send data to write in memory\n");
    state=3; M=this;}
  :: atomic{myturn && state==3 -> call_memory=1; state=4; M=5;}
  :: atomic{myturn && state==4 -> call_memory=1; state=5; M=5;}
  :: atomic{myturn && state==5 -> call_memory=0; call_sorter=1; state=6; M=7;}
  :: atomic{myturn && state==6 -> call_memory=1; state=7; M=5;}
  :: atomic{myturn && state==7 -> state=1;M=3;}
progress:
od
}

active proctype module2_memory() {
byte state=1;
byte this=5;
end_memory:
do
  :: atomic{myturn && state==1 && call_memory==1 -> b_t=1; state=2; M =6;}
  :: atomic{myturn && state==2 && b_t==0 -> call_memory=0; state=1; M=4;}
progress:
od
}
```

```
active proctype b_transport_memory()
{
byte state=1;
byte this=6;
end_bt_memory:
do
  :: atomic{myturn && state==1 && b_t==1 -> state=2; M=this;}
  :: atomic{myturn && state==2 -> printf("Memory Response\n"); state=3; M=this;
    Memory_used=1;}
  :: atomic{myturn && state==3 -> b_t=0; state=1; M=5; Memory_used=0;}
progress:
od
}


active proctype module3_Sorter() {
byte state=1;
byte this=7;
end_sorter:
do
  :: atomic{myturn && state==1 && call_sorter==1 -> b_t_sorter=1; state=2; M =8;}
  :: atomic{myturn && state==2 && b_t_sorter==0 -> call_sorter=0; state=1; M=4;}
progress:
od
}

active proctype b_transport_Sorter()
{
byte state=1;
byte this=8;
end_bt_sorter:
do
  :: atomic{myturn && state==1 && b_t_sorter==1 -> state=2; M=this;}
  :: atomic{myturn && state==2 -> printf("Sorting DATA\n"); state=3; M=this;
    Sorter_used=1}
  :: atomic{myturn && state==3 -> b_t_sorter=0; state=1; M=7; Sorter_used=0;}
progress:
od
}
```

Código 5: Resultado de Simulación del Sistema.

```
0 s@ Initial data
[data(0)=f7] [data(1)=f3] [data(2)=51] [data(3)=dd] [data(4)=73] [data(5)=c1]
[data(6)=12] [data(7)=32] [data(8)=a] [data(9)=61] [data(10)=b8] [data(11)=ab]
[data(12)=78] [data(13)=b3] [data(14)=99] [data(15)=8e] [data(16)=f0] [data(17)=a5]
```

```
[data(18)=50] [data(19)=84] [data(20)=98] [data(21)=67] [data(22)=68] [data(23)=6b]
[data(24)=c8] [data(25)=ca] [data(26)=49] [data(27)=99] [data(28)=81] [data(29)=5b]
[data(30)=88] [data(31)=f9] [data(32)=4f] [data(33)=d9] [data(34)=d7] [data(35)=c2]
[data(36)=9c] [data(37)=6a] [data(38)=75] [data(39)=26] [data(40)=cb] [data(41)=2e]
[data(42)=51] [data(43)=c3] [data(44)=61] [data(45)=ea] [data(46)=53] [data(47)=52]
[data(48)=90] [data(49)=23] [data(50)=d7] [data(51)=2a] [data(52)=8b] [data(53)=40]
[data(54)=15] [data(55)=54] [data(56)=8b] [data(57)=5e] [data(58)=6d] [data(59)=8c]
[data(60)=b9] [data(61)=f6] [data(62)=6] [data(63)=89] [data(64)=d0] [data(65)=de]
[data(66)=cb] [data(67)=ec] [data(68)=49] [data(69)=41] [data(70)=14] [data(71)=94]
[data(72)=ef] [data(73)=65] [data(74)=d7] [data(75)=51] [data(76)=51] [data(77)=aa]
[data(78)=a4] [data(79)=e1] [data(80)=ce] [data(81)=fb] [data(82)=8b] [data(83)=d9]
[data(84)=3c] [data(85)=a0] [data(86)=2e] [data(87)=47] [data(88)=0] [data(89)=9b]
[data(90)=54] [data(91)=39] [data(92)=12] [data(93)=5a] [data(94)=42] [data(95)=e3]
[data(96)=b8] [data(97)=f] [data(98)=50] [data(99)=2] [data(100)=cf] [data(101)=64]
[data(102)=16] [data(103)=3f] [data(104)=4a] [data(105)=6e] [data(106)=11] [data(107)=9b]
[data(108)=19] [data(109)=b5] [data(110)=fc] [data(111)=67] [data(112)=31] [data(113)=9]
[data(114)=41] [data(115)=ec] [data(116)=a9] [data(117)=6f] [data(118)=35] [data(119)=29]
[data(120)=8b] [data(121)=9] [data(122)=e2] [data(123)=9d] [data(124)=63] [data(125)=25]
[data(126)=1] [data(127)=1d] [data(128)=b3] [data(129)=52] [data(130)=9e] [data(131)=4]
[data(132)=36] [data(133)=35] [data(134)=c2] [data(135)=80] [data(136)=a3] [data(137)=d3]
[data(138)=9b] [data(139)=3c] [data(140)=9] [data(141)=19] [data(142)=a4] [data(143)=3a]
[data(144)=22] [data(145)=e5] [data(146)=28] [data(147)=cb] [data(148)=d5] [data(149)=dc]
[data(150)=75] [data(151)=e0] [data(152)=e5] [data(153)=58] [data(154)=fd] [data(155)=49]
[data(156)=fc] [data(157)=0] [data(158)=e5] [data(159)=31] [data(160)=d1] [data(161)=85]
[data(162)=35] [data(163)=8] [data(164)=3a] [data(165)=f7] [data(166)=9] [data(167)=dd]
[data(168)=4c] [data(169)=a4] [data(170)=99] [data(171)=55] [data(172)=bd] [data(173)=3e]
[data(174)=90] [data(175)=df] [data(176)=a4] [data(177)=b8] [data(178)=2c] [data(179)=7a]
[data(180)=95] [data(181)=a1] [data(182)=da] [data(183)=fa] [data(184)=79] [data(185)=d8]
[data(186)=44] [data(187)=76] [data(188)=58] [data(189)=aa] [data(190)=a7] [data(191)=2a]
[data(192)=af] [data(193)=5c] [data(194)=b2] [data(195)=e9] [data(196)=d4] [data(197)=bb]
[data(198)=47] [data(199)=21] [data(200)=df] [data(201)=e0] [data(202)=f5] [data(203)=9e]
[data(204)=20] [data(205)=86] [data(206)=fd] [data(207)=44] [data(208)=be] [data(209)=a9]
[data(210)=3e] [data(211)=d3] [data(212)=ca] [data(213)=19] [data(214)=ce] [data(215)=44]
[data(216)=f1] [data(217)=93] [data(218)=3b] [data(219)=4b] [data(220)=3e] [data(221)=62]
[data(222)=f4] [data(223)=ed] [data(224)=3f] [data(225)=a7] [data(226)=d7] [data(227)=14]
[data(228)=63] [data(229)=1f] [data(230)=b4] [data(231)=c3] [data(232)=7f] [data(233)=aa]
[data(234)=62] [data(235)=1f] [data(236)=b1] [data(237)=df] [data(238)=63] [data(239)=70]
[data(240)=a] [data(241)=a1] [data(242)=45] [data(243)=d4] [data(244)=3a] [data(245)=93]
[data(246)=99] [data(247)=ac] [data(248)=27] [data(249)=54] [data(250)=77] [data(251)=65]
[data(252)=36] [data(253)=6c] [data(254)=d2] [data(255)=75] [data(256)=94] [data(257)=aa]
[data(258)=9] [data(259)=77] [data(260)=49] [data(261)=bd] [data(262)=3b] [data(263)=49]
[data(264)=e8] [data(265)=1d] [data(266)=68] [data(267)=9a] [data(268)=fd] [data(269)=cc]
[data(270)=8a] [data(271)=8] [data(272)=ed] [data(273)=4f] [data(274)=5c] [data(275)=a8]
[data(276)=e3] [data(277)=f5] [data(278)=55] [data(279)=8a] [data(280)=c9] [data(281)=cc]
[data(282)=70] [data(283)=1] [data(284)=b8] [data(285)=43] [data(286)=76] [data(287)=4d]
[data(288)=6e] [data(289)=0] [data(290)=c5] [data(291)=37] [data(292)=3d] [data(293)=1]
[data(294)=80] [data(295)=26] [data(296)=1f] [data(297)=e9] [data(298)=40] [data(299)=9c]
[data(300)=36] [data(301)=cb] [data(302)=24] [data(303)=a3] [data(304)=1b] [data(305)=0]
[data(306)=4c] [data(307)=fe] [data(308)=76] [data(309)=21] [data(310)=8a] [data(311)=40]
[data(312)=6d] [data(313)=7a] [data(314)=41] [data(315)=27] [data(316)=bd] [data(317)=38]
[data(318)=f3] [data(319)=2c] [data(320)=38] [data(321)=39] [data(322)=e3] [data(323)=f4]
[data(324)=3b] [data(325)=64] [data(326)=1c] [data(327)=d9] [data(328)=cd] [data(329)=db]
[data(330)=f5] [data(331)=4] [data(332)=a7] [data(333)=99] [data(334)=28] [data(335)=c3]
[data(336)=99] [data(337)=74] [data(338)=42] [data(339)=10] [data(340)=16] [data(341)=4c]
[data(342)=51] [data(343)=83] [data(344)=c6] [data(345)=12] [data(346)=2a] [data(347)=5]
[data(348)=4a] [data(349)=1f] [data(350)=b0] [data(351)=2] [data(352)=58] [data(353)=94]
[data(354)=77] [data(355)=13] [data(356)=f9] [data(357)=13] [data(358)=6c] [data(359)=47]
[data(360)=ee] [data(361)=62] [data(362)=cb] [data(363)=97] [data(364)=fb] [data(365)=f3]
[data(366)=da] [data(367)=16] [data(368)=e7] [data(369)=9c] [data(370)=26] [data(371)=fd]
[data(372)=e9] [data(373)=f6] [data(374)=82] [data(375)=30] [data(376)=a] [data(377)=ac]
[data(378)=35] [data(379)=d3] [data(380)=4b] [data(381)=e6] [data(382)=56] [data(383)=a4]
[data(384)=7b] [data(385)=cd] [data(386)=37] [data(387)=f4] [data(388)=e0] [data(389)=a4]
[data(390)=bc] [data(391)=cf] [data(392)=7] [data(393)=88] [data(394)=e6] [data(395)=83]
[data(396)=7c] [data(397)=c1] [data(398)=99] [data(399)=64] [data(400)=de] [data(401)=3f]
[data(402)=e2] [data(403)=c8] [data(404)=37] [data(405)=e4] [data(406)=f8] [data(407)=41]
[data(408)=11] [data(409)=ae] [data(410)=15] [data(411)=5d] [data(412)=95] [data(413)=6b]
[data(414)=81] [data(415)=90] [data(416)=b8] [data(417)=b8] [data(418)=6] [data(419)=99]
[data(420)=dc] [data(421)=c2] [data(422)=e9] [data(423)=64] [data(424)=ca] [data(425)=d0]
[data(426)=e7] [data(427)=47] [data(428)=13] [data(429)=1] [data(430)=2b] [data(431)=f1]
```

```
[data(432)=40] [data(433)=8d] [data(434)=3a] [data(435)=77] [data(436)=72] [data(437)=b2]
[data(438)=38] [data(439)=84] [data(440)=e0] [data(441)=4e] [data(442)=e1] [data(443)=76]
[data(444)=39] [data(445)=e2] [data(446)=87] [data(447)=72] [data(448)=1b] [data(449)=d]
[data(450)=c] [data(451)=f8] [data(452)=4f] [data(453)=75] [data(454)=5d] [data(455)=1a]
[data(456)=c6] [data(457)=c4] [data(458)=e0] [data(459)=d9] [data(460)=c5] [data(461)=c]
[data(462)=4b] [data(463)=85] [data(464)=9a] [data(465)=5] [data(466)=fd] [data(467)=8c]
[data(468)=37] [data(469)=36] [data(470)=11] [data(471)=19] [data(472)=4] [data(473)=72]
[data(474)=f] [data(475)=bd] [data(476)=55] [data(477)=96] [data(478)=30] [data(479)=f0]
[data(480)=23] [data(481)=bb] [data(482)=e9] [data(483)=72] [data(484)=32] [data(485)=47]
[data(486)=c] [data(487)=f8] [data(488)=c] [data(489)=ec] [data(490)=52] [data(491)=51]
[data(492)=f9] [data(493)=1d] [data(494)=d6] [data(495)=14] [data(496)=a1] [data(497)=54]
[data(498)=a0] [data(499)=d8] [data(500)=b] [data(501)=32] [data(502)=f1] [data(503)=8e]
[data(504)=24] [data(505)=2] [data(506)=4c] [data(507)=f9] [data(508)=18] [data(509)=fb]
[data(510)=ea] [data(511)=3c]
0 s@ Send data to ETH
0 s@ top.Eth sended

0 sSend to Sorter[0]  memory, delay=0 s

20 nsRead from Sorter[0]

60 nsWrite to memory
180 ns@ Read data from Memory
180 ns@ top.Eth sended
180 nsRead from memory
[data(0)=0] [data(1)=0] [data(2)=0] [data(3)=0] [data(4)=1] [data(5)=1]
[data(6)=1] [data(7)=1] [data(8)=2] [data(9)=2] [data(10)=2] [data(11)=4]
[data(12)=4] [data(13)=4] [data(14)=5] [data(15)=5] [data(16)=6] [data(17)=6]
[data(18)=7] [data(19)=8] [data(20)=8] [data(21)=9] [data(22)=9] [data(23)=9]
[data(24)=9] [data(25)=9] [data(26)=a] [data(27)=a] [data(28)=a] [data(29)=b]
[data(30)=c] [data(31)=c] [data(32)=c] [data(33)=c] [data(34)=d] [data(35)=f]
[data(36)=f] [data(37)=10] [data(38)=11] [data(39)=11] [data(40)=11] [data(41)=12]
[data(42)=12] [data(43)=12] [data(44)=13] [data(45)=13] [data(46)=13] [data(47)=14]
[data(48)=14] [data(49)=14] [data(50)=15] [data(51)=15] [data(52)=16] [data(53)=16]
[data(54)=16] [data(55)=18] [data(56)=19] [data(57)=19] [data(58)=19] [data(59)=19]
[data(60)=1a] [data(61)=1b] [data(62)=1b] [data(63)=1c] [data(64)=1d] [data(65)=1d]
[data(66)=1d] [data(67)=1f] [data(68)=1f] [data(69)=1f] [data(70)=1f] [data(71)=20]
[data(72)=21] [data(73)=21] [data(74)=22] [data(75)=23] [data(76)=23] [data(77)=24]
[data(78)=24] [data(79)=25] [data(80)=26] [data(81)=26] [data(82)=26] [data(83)=27]
[data(84)=27] [data(85)=28] [data(86)=28] [data(87)=29] [data(88)=2a] [data(89)=2a]
[data(90)=2a] [data(91)=2b] [data(92)=2c] [data(93)=2c] [data(94)=2e] [data(95)=2e]
[data(96)=30] [data(97)=30] [data(98)=31] [data(99)=31] [data(100)=32] [data(101)=32]
[data(102)=32] [data(103)=35] [data(104)=35] [data(105)=35] [data(106)=35] [data(107)=36]
[data(108)=36] [data(109)=36] [data(110)=36] [data(111)=37] [data(112)=37] [data(113)=37]
[data(114)=37] [data(115)=38] [data(116)=38] [data(117)=38] [data(118)=39] [data(119)=39]
[data(120)=39] [data(121)=3a] [data(122)=3a] [data(123)=3a] [data(124)=3a] [data(125)=3b]
[data(126)=3b] [data(127)=3b] [data(128)=3c] [data(129)=3c] [data(130)=3c] [data(131)=3d]
[data(132)=3e] [data(133)=3e] [data(134)=3e] [data(135)=3f] [data(136)=3f] [data(137)=3f]
[data(138)=40] [data(139)=40] [data(140)=40] [data(141)=40] [data(142)=41] [data(143)=41]
[data(144)=41] [data(145)=41] [data(146)=42] [data(147)=42] [data(148)=43] [data(149)=44]
[data(150)=44] [data(151)=44] [data(152)=45] [data(153)=47] [data(154)=47] [data(155)=47]
[data(156)=47] [data(157)=47] [data(158)=49] [data(159)=49] [data(160)=49] [data(161)=49]
[data(162)=49] [data(163)=4a] [data(164)=4a] [data(165)=4b] [data(166)=4b] [data(167)=4b]
[data(168)=4c] [data(169)=4c] [data(170)=4c] [data(171)=4c] [data(172)=4d] [data(173)=4e]
[data(174)=4f] [data(175)=4f] [data(176)=4f] [data(177)=50] [data(178)=50] [data(179)=51]
[data(180)=51] [data(181)=51] [data(182)=51] [data(183)=51] [data(184)=51] [data(185)=52]
[data(186)=52] [data(187)=52] [data(188)=53] [data(189)=54] [data(190)=54] [data(191)=54]
[data(192)=54] [data(193)=55] [data(194)=55] [data(195)=55] [data(196)=56] [data(197)=58]
[data(198)=58] [data(199)=58] [data(200)=5a] [data(201)=5b] [data(202)=5c] [data(203)=5c]
[data(204)=5d] [data(205)=5d] [data(206)=5e] [data(207)=61] [data(208)=61] [data(209)=62]
[data(210)=62] [data(211)=62] [data(212)=63] [data(213)=63] [data(214)=63] [data(215)=64]
[data(216)=64] [data(217)=64] [data(218)=64] [data(219)=65] [data(220)=65] [data(221)=67]
[data(222)=67] [data(223)=68] [data(224)=68] [data(225)=6a] [data(226)=6b] [data(227)=6b]
[data(228)=6c] [data(229)=6c] [data(230)=6d] [data(231)=6d] [data(232)=6e] [data(233)=6e]
[data(234)=6f] [data(235)=70] [data(236)=70] [data(237)=72] [data(238)=72] [data(239)=72]
[data(240)=72] [data(241)=73] [data(242)=74] [data(243)=75] [data(244)=75] [data(245)=75]
[data(246)=75] [data(247)=76] [data(248)=76] [data(249)=76] [data(250)=76] [data(251)=77]
[data(252)=77] [data(253)=77] [data(254)=77] [data(255)=78] [data(256)=79] [data(257)=7a]
[data(258)=7a] [data(259)=7b] [data(260)=7c] [data(261)=7f] [data(262)=80] [data(263)=80]
```

```
    [data(264)=81] [data(265)=81] [data(266)=82] [data(267)=83] [data(268)=83] [data(269)=84]
144 [data(270)=84] [data(271)=85] [data(272)=85] [data(273)=86] [data(274)=87] [data(275)=88]
    [data(276)=88] [data(277)=89] [data(278)=8a] [data(279)=8a] [data(280)=8a] [data(281)=8b]
146 [data(282)=8b] [data(283)=8b] [data(284)=8b] [data(285)=8c] [data(286)=8c] [data(287)=8d]
    [data(288)=8e] [data(289)=8e] [data(290)=90] [data(291)=90] [data(292)=90] [data(293)=93]
148 [data(294)=93] [data(295)=94] [data(296)=94] [data(297)=94] [data(298)=95] [data(299)=95]
    [data(300)=96] [data(301)=97] [data(302)=98] [data(303)=99] [data(304)=99] [data(305)=99]
150 [data(306)=99] [data(307)=99] [data(308)=99] [data(309)=99] [data(310)=99] [data(311)=9a]
    [data(312)=9a] [data(313)=9b] [data(314)=9b] [data(315)=9b] [data(316)=9c] [data(317)=9c]
152 [data(318)=9c] [data(319)=9d] [data(320)=9e] [data(321)=9e] [data(322)=a0] [data(323)=a0]
    [data(324)=a1] [data(325)=a1] [data(326)=a1] [data(327)=a3] [data(328)=a3] [data(329)=a4]
154 [data(330)=a4] [data(331)=a4] [data(332)=a4] [data(333)=a4] [data(334)=a4] [data(335)=a5]
    [data(336)=a7] [data(337)=a7] [data(338)=a7] [data(339)=a8] [data(340)=a9] [data(341)=a9]
156 [data(342)=aa] [data(343)=aa] [data(344)=aa] [data(345)=aa] [data(346)=ab] [data(347)=ac]
    [data(348)=ac] [data(349)=ae] [data(350)=af] [data(351)=b0] [data(352)=b1] [data(353)=b2]
158 [data(354)=b2] [data(355)=b3] [data(356)=b3] [data(357)=b4] [data(358)=b5] [data(359)=b8]
    [data(360)=b8] [data(361)=b8] [data(362)=b8] [data(363)=b8] [data(364)=b8] [data(365)=b9]
160 [data(366)=bb] [data(367)=bb] [data(368)=bc] [data(369)=bd] [data(370)=bd] [data(371)=bd]
    [data(372)=bd] [data(373)=be] [data(374)=c1] [data(375)=c1] [data(376)=c2] [data(377)=c2]
162 [data(378)=c2] [data(379)=c3] [data(380)=c3] [data(381)=c3] [data(382)=c4] [data(383)=c5]
    [data(384)=c5] [data(385)=c6] [data(386)=c6] [data(387)=c8] [data(388)=c8] [data(389)=c9]
164 [data(390)=ca] [data(391)=ca] [data(392)=ca] [data(393)=cb] [data(394)=cb] [data(395)=cb]
    [data(396)=cb] [data(397)=cb] [data(398)=cc] [data(399)=cc] [data(400)=cd] [data(401)=cd]
166 [data(402)=ce] [data(403)=ce] [data(404)=cf] [data(405)=cf] [data(406)=d0] [data(407)=d0]
    [data(408)=d1] [data(409)=d2] [data(410)=d3] [data(411)=d3] [data(412)=d3] [data(413)=d4]
168 [data(414)=d4] [data(415)=d5] [data(416)=d6] [data(417)=d7] [data(418)=d7] [data(419)=d7]
    [data(420)=d7] [data(421)=d8] [data(422)=d8] [data(423)=d9] [data(424)=d9] [data(425)=d9]
170 [data(426)=d9] [data(427)=da] [data(428)=da] [data(429)=db] [data(430)=dc] [data(431)=dc]
    [data(432)=dd] [data(433)=dd] [data(434)=de] [data(435)=de] [data(436)=df] [data(437)=df]
172 [data(438)=df] [data(439)=e0] [data(440)=e0] [data(441)=e0] [data(442)=e0] [data(443)=e0]
    [data(444)=e1] [data(445)=e1] [data(446)=e2] [data(447)=e2] [data(448)=e2] [data(449)=e3]
174 [data(450)=e3] [data(451)=e3] [data(452)=e4] [data(453)=e5] [data(454)=e5] [data(455)=e5]
    [data(456)=e6] [data(457)=e6] [data(458)=e7] [data(459)=e7] [data(460)=e8] [data(461)=e9]
176 [data(462)=e9] [data(463)=e9] [data(464)=e9] [data(465)=e9] [data(466)=ea] [data(467)=ea]
    [data(468)=ec] [data(469)=ec] [data(470)=ec] [data(471)=ed] [data(472)=ed] [data(473)=ee]
178 [data(474)=ef] [data(475)=f0] [data(476)=f0] [data(477)=f1] [data(478)=f1] [data(479)=f1]
    [data(480)=f3] [data(481)=f3] [data(482)=f3] [data(483)=f4] [data(484)=f4] [data(485)=f4]
180 [data(486)=f5] [data(487)=f5] [data(488)=f5] [data(489)=f6] [data(490)=f6] [data(491)=f7]
    [data(492)=f7] [data(493)=f8] [data(494)=f8] [data(495)=f8] [data(496)=f9] [data(497)=f9]
182 [data(498)=f9] [data(499)=f9] [data(500)=fa] [data(501)=fb] [data(502)=fb] [data(503)=fb]
    [data(504)=fc] [data(505)=fc] [data(506)=fd] [data(507)=fd] [data(508)=fd] [data(509)=fd]
184 [data(510)=fd] [data(511)=fe]
```

Apéndice B: Códigos SystemC

Código 6: Tipo de datos propio MyType..

```
 int get(int pos)
 {
  return array[pos];
 }

  inline bool operator == (const MyType & rhs) const {
int i,flag;
while (i<SIZE_OF_PACKET && flag)
{
  if (rhs.array[i]==array[0])
    flag=1;
  else
    flag=0;
  i++;
}
return flag;


  }

  inline MyType& operator = (const MyType& rhs) {
memcpy(array,rhs.array,SIZE_OF_PACKET*sizeof(int));
return *this;
  }

  inline friend void sc_trace(sc_trace_file *tf, const MyType & v,
  const std::string& NAME ) {
int i;
for (i=0;i<SIZE_OF_PACKET;i++)
  sc_trace(tf,v.array[i],NAME+"array0");

```

```
    }

    inline friend ostream& operator << ( ostream& os,  MyType const & v ) {
        int i;

  for (i=0;i<SIZE_OF_PACKET;i++)
  {
      os<<"["<<dec<<v.array[i]<<"]";
  }
  os<<endl;
  return os;
  }

};
#endif
```

Código 7: Interface propia: My_IF.

```
#ifndef MY_IF_H
#define MY_IF_H
#include "systemc.h"
#include "mytype.h"

class my_if:public sc_interface
{
public:
  virtual void write(MyType a)=0;
  virtual MyType read()=0;
  virtual void notify()=0;
  virtual void notify(sc_time t)=0;
  virtual const sc_event& default_event() const=0;
};
#endif
```

Código 8: Canal propio: Parallel_CH.

```
#ifndef PARALLEL_CH_H
#define PARALLEL_CH_H

#include "systemc.h"
#include "mytype.h"
#include "my_if.h"
class parallel_ch: public sc_prim_channel, public my_if
{
private:
```

```
11    bool m_req, m_written;
      sc_event m_write_event;
13    MyType m_new_val, m_cur_val;
      parallel_ch(const parallel_ch& rhs){}
15
  public:
17    //constructors

19    explicit parallel_ch():sc_prim_channel(sc_gen_unique_name("p_ch")){}

21    explicit parallel_ch(sc_module_name nm):sc_prim_channel(nm){}

23    void notify(){m_write_event.notify();}

25    void notify(sc_time t){m_write_event.notify(t);}

27    const sc_event& default_event() const {return m_write_event;}

29    void write(MyType val)
      {
31      if (!m_req)
        {
33        m_new_val=val;
          request_update();
35        m_req=true;
        }
37    }


39
      void update()
41    {
        m_cur_val=m_new_val;
43      m_req=false;
        m_written=true;
45      m_write_event.notify(SC_ZERO_TIME);
      }
47
      MyType read()
49    {

51      return m_cur_val;
      }
53
```

```
};
#endif
```

Código 9: Ejemplo de randomización.

```
#include <scv.h>

int sc_main (int argc, char* argv[]) {
  scv_smart_ptr< sc_uint<8> > data;
  cout <<"Value of data pre  randomize : " << endl;
  data->print();
  data->next(); // Randomize object data
  cout <<"Value of data post randomize : " << endl;
  data->print();
  return 0;
}
```

Código 10: Generación de estímulos, valores positivos.

```
        MyType x(rand()%100,rand()%100,rand()%100,rand()%100,rand()%100,rand()
    %100,rand()%100,rand()%100);
```

Código 11: Generación de estímulos, valores negativos.

```
        MyType x(-rand()%100,-rand()%100,-rand()%100,-rand()%100,-rand()%100,-rand
    ()%100,-rand()%100,-rand()%100);
```

Código 12: Generación de estímulos, valores ascendentes.

```
        MyType x(i,i,i,i,i,i,i,i);
```

Código 13: Generación de estímulos, valores descendetes

```
        MyType x(-i,-i,-i,-i,-i,-i,-i,-i);
```

Código 14: Constraints.

```
#include "mytype.h"

template<>
class scv_extensions<MyType> : public scv_extensions_base<MyType> {
public:
    scv_extensions<int> info0;
    scv_extensions<int> info1;
    scv_extensions<int> info2;
```

```
9       scv_extensions<int> info3;
        scv_extensions<int> info4;
11      scv_extensions<int> info5;
        scv_extensions<int> info6;
13      scv_extensions<int> info7;

15    SCV_EXTENSIONS_CTOR(MyType) {
          SCV_FIELD(info0);
17    SCV_FIELD(info1);
      SCV_FIELD(info2);
19    SCV_FIELD(info3);
      SCV_FIELD(info4);
21    SCV_FIELD(info5);
      SCV_FIELD(info6);
23    SCV_FIELD(info7);
      }
25  };

27
    struct packet_base_constraint : public scv_constraint_base {
29    scv_smart_ptr<MyType> packet;
      SCV_CONSTRAINT_CTOR(packet_base_constraint) {}
31  };

33  struct packet_basic_constraint : public packet_base_constraint {
     scv_smart_ptr<int> info0;
35   scv_smart_ptr<int> info1;
     scv_smart_ptr<int> info2;
37   scv_smart_ptr<int> info3;
     scv_smart_ptr<int> info4;
39   scv_smart_ptr<int> info5;
     scv_smart_ptr<int> info6;
41   scv_smart_ptr<int> info7;


43
      SCV_CONSTRAINT_CTOR(packet_basic_constraint) {
45      SCV_BASE_CONSTRAINT(packet_base_constraint);
      SCV_CONSTRAINT (( packet->info0()>-1000) && (packet->info0()<1000));
47    SCV_CONSTRAINT (( packet->info1()>-1000) && (packet->info1()<1000));
      SCV_CONSTRAINT (( packet->info2()>-1000) && (packet->info2()<1000));
49    SCV_CONSTRAINT (( packet->info3()>-1000) && (packet->info3()<1000));
      SCV_CONSTRAINT (( packet->info4()>-1000) && (packet->info4()<1000));
51    SCV_CONSTRAINT (( packet->info5()>-1000) && (packet->info5()<1000));
      SCV_CONSTRAINT (( packet->info6()>-1000) && (packet->info6()<1000));
```

```
53    SCV_CONSTRAINT (( packet->info7()>-1000) && (packet->info7()<1000));
      }
55  };
```

Apéndice C: Publicaciones

Diseño y Verificación de un Filtro de Imágenes

Manuel F. Soto*†, Juan Francesconi*†, Gabriel Pachiana*†, Martín Di Federico*†
*Centro de Micro y Nanoelectrónica del Bicentenerio (CMNB)
Instituto Nacional de Tecnología Industrial (INTI)
†Instituto de Investigaciones en Ing. Eléctrica "Alfredo Desages" (CONICET)
Universidad Nacional del Sur
Email: {msoto, jfrancesconi, gpachiana, martind}@inti.gob.ar

***Resumen*—Las estructuras de filtrado de imágenes se utilizan hoy en día en un amplio rango de aplicaciones, y para muchas de ellas es necesario contar con hardware dedicado debido a restricciones temporales. Este trabajo presenta el desarrollo en RTL de un filtro digital de imágenes basado en el método de convolución. El mismo implementa una arquitectura con procesamiento paralelo, la cual es parametrizable en función del tamaño de la imagen de entrada y del tamaño del kernel, y configurable en función de los valores de este ultimo. Se hace foco en el flujo de diseño estructurado en cascada utilizado para el desarrollo del filtro, incluyendo las etapas de especificación, modelado y diseño, descripción a nivel RTL, verificación y síntesis lógica de la unidad aritmética. Para cada etapa se describen las técnicas, modelos y herramientas utilizados para facilitar el desarrollo.**

I. Introducción

El procesamiento digital de imágenes es utilizado en varios campos de aplicación tales como visión artificial, robótica, diagnostico médico mediante imágenes, etc. Es común que estos sistemas necesiten acondicionar o resaltar características de la imagen de entrada en etapas previas a la aplicación del algoritmo de procesamiento principal. Una de las formas más populares de lograr este objetivo es mediante filtrado por convolución [1], [2].

La convolución discreta en dos dimensiones consiste en la operación entre una imagen X de tamaño $M \times N$ pixels y una matriz W de coeficientes (*kernel*) de tamaño $H \times K$. Esta operación se define de la siguiente manera:

$$y(i,j) = \sum_{h=0}^{H-1} \sum_{k=0}^{K-1} w(h,k) \times x(i-h, j-h), \qquad (1)$$

donde $0 \leq i \leq M-1$, $0 \leq j \leq N-1$, $x(i,j) \epsilon X$ y $w(h,k) \epsilon W$.

Existen diversas formas de implementar esta función. Velocidad, área y consumo son los tres parámetros que, generalmente, se deben equilibrar dependiendo de los requerimientos del sistema. Cuando el tiempo de respuesta es crítico se deben sacrificar los parámetros restantes, como sucede en aplicaciones de tiempo real como las de visión artificial.

La convolución puede ser implementada por software o por hardware. Cuando se requiere cumplir con tiempos de respuesta exigentes se suele recurrir a implementaciones por hardware que permitan un procesamiento con un alto grado de paralelismo como es el caso de el hardware reconfigurable (FPGA, *Field Programmable Gate Array*) o un circuito integrado de aplicación especifica (ASIC, *Application Specific Integrated Circuit*).

Existen diversos trabajos sobre descripción de métodos de convolución mediante RTL (*Register Transfer Level*), como en [3] donde se propone una arquitectura que realiza la convolución de una imagen 2-D utilizando una memoria dedicada. El trabajo realizado por Azizabadi y Behrad [4] se enfoca en la definición de tres arquitecturas de filtrado especificas para filtro gaussiano y filtro de mediana, se lleva a cabo el diseño, implementación y verificación de las propuestas. En [5] se implementa el algoritmo de convolución utilizando una memoria jerárquica la cual permite dar soporte a un procesamiento con un alto grado de paralelismo. En [6] se analizan *kernels* comúnmente utilizados en busca de simetrías y se proponen arquitecturas alternativas para FPGA que aprovechan estas propiedades.

Este trabajo presenta el flujo de desarrollo utilizado para diseñar e implementar en RTL un filtro de imágenes mediante el método de convolución discreto en 2-D. El bloque desarrollado es parametrizable dependiendo del tamaño de la imagen y del *kernel*, y configurable ya que permite la carga serial de distintos *kernels*. El diseño prioriza la velocidad de procesamiento mediante una arquitectura con procesamiento paralelo. Por último se presentan los resultados de las simulaciones del sistema, la síntesis lógica de la unidad aritmética y una breve conclusión.

II. Desarrollo

La metodología utilizada para llevar a cabo el desarrollo del sistema es estructurada y de tipo cascada [7]. Comprende cuatro etapas fundamentales: especificación, modelado y diseño, implementación RTL y verificación mediante simulación.

II-A. Especificación

La especificación es una parte crucial en el flujo de diseño donde se define la funcionalidad, los requerimientos estructurales y otros aspectos prioritarios del sistema.

La funcionalidad del sistema es realizar el filtrado de imágenes. Para esto debe proveer de estructuras que soporten el ingreso tanto de imágenes como de *kernels* parametrizables, y que éstos últimos contemplen valores negativos y fraccionarios. Dado que se debe priorizar la velocidad de procesamiento por sobre área o consumo del sistema, se debe contemplar un cierto grado de procesamiento paralelo en el sistema.

También forman parte de las especificaciones el protocolo de funcionamiento y la interfaz de comunicación del sistema:

1) Interfaz de comunicación: La interfaz planteada debe contar con señales de entrada de: *clk*, *rst*, datos de entrada, poder discriminar si los datos corresponden a un kernel o a una imagen; análogamente debe tener señales de salida que den soporte a la lectura de los resultados.

2) Protocolo de funcionamiento: En una primera instancia el diseño se debe inicializar mediante la señal *rst*, la cual establecerá los valores iniciales de los registros internos. La imagen debe ser cuadrada de tamaño fijo SI en escala de grises y el kernel debe ser cuadrado de tamaño fijo SK donde cada pixel de la imagen será un entero de 0 a 255. Las constantes SI y SK son valores establecidos en la parametrización del sistema. Los datos de la imagen de entrada se suponen cargados en el sistema al momento de realizar el procesamiento. Una vez que, tanto los datos de la imagen como del kernel se encuentran almacenados en el sistema, se procede a realizar la convolución. Para dar inicio a ésta, se estimulan las señales de control. Al finalizar el procesamiento, se da a conocer esta situación mediante una señal de control, luego se podrán obtener los resultados.

II-B. Modelado y Diseño

Una vez que las especificaciones del sistema se encuentran completas, se procede al modelado *top-down* de la arquitectura, al diseño del algoritmo utilizado para implementar la convolución, al modelado de la maquina de estados de la unidad de control, a la definición de la aritmética utilizada en el sistema y al análisis de complejidad del orden de ejecución.

1) Algoritmo de Convolución: La heurística clásica del algoritmo de convolución es ir superponiendo la matriz que representa el kernel sobre la matriz que representa la imagen, comenzando, por ejemplo, en la esquina superior izquierda, e ir desplazando el kernel a lo largo de la fila, bajando a la siguiente fila cuando se llega a la columna final. En cada superposición, se multiplican los valores respectivos de los pixeles de la imagen con los del kernel, luego se suman estos resultados parciales y se almacenan en la misma posición de pixel pero de la matriz resultante.

Una implementación en hardware puede optimizar este algoritmo aplicando ciertos cambios. En lugar de que el kernel recorra la imagen, se puede replicar éste N veces (con N igual a la cantidad de filas de la imagen), logrando una columna de *kernels* que procese la imagen de una solo pasada horizontal (o vertical). Es decir, se va desplazando (operando) la imagen a través de la columna de *kernels*, logrando operar en forma paralela. De esta manera, la convolución completa se lograra en N iteraciones, reduciendo a N el orden de ejecución de la aproximación clásica (N^2), en la sección II-B5 se realiza un análisis más exhaustivo de la complejidad del sistema.

2) Arquitectura: En la arquitectura del sistema (Figura 1) se pueden apreciar dos grandes bloques: el *datapath* y otra denominada *control unit*, o unidad de control.

El *datapath* esta compuesto por tres unidades de memoria (*img_mem*, *kernel_mem* y *result_mem*) y una unidad de procesamiento denominada *processing_unit*. Las memorias se han implementado con el fin de dar soporte a las necesidades de la unidad de procesamiento, las mismas pueden cambiar según las necesidades del contexto del sistema [5].

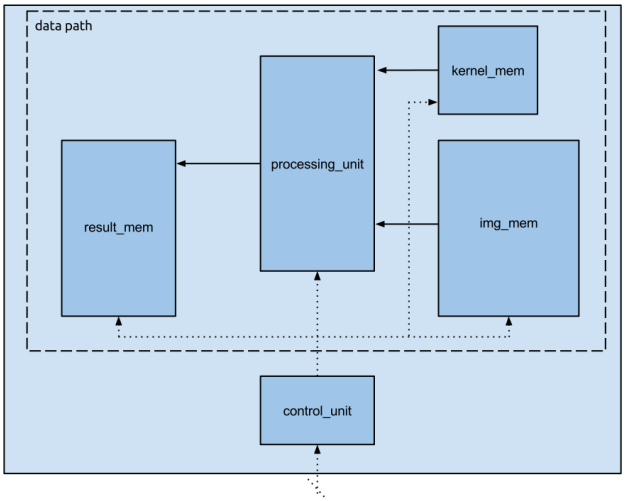


Figura 1: Esquemático general de la arquitectura.

La memoria de entrada (*img_mem*) es la encargada de almacenar la imagen y dar soporte los requerimientos de datos de la unidad de procesamiento. La misma tiene un tamaño de $SI \times SI$ bytes. La memoria de salida (*result_mem*) es la responsable de almacenar los resultados de la imagen procesada y permitir la lectura de ésta desde el exterior, su tamaño debe ser congruente con el de la memoria de entrada. La escritura de la misma se realiza de forma paralela.

La memoria del kernel (*kernel_mem*) es la responsable de almacenar los valores del filtro que se desea aplicar. Su lectura es paralela de todos sus valores, la misma tiene un tamaño de $SK \times SK \times 16$ bits.

La unidad de procesamiento (*processing_unit*), presentada en la Figura 2, es la encargada de realizar la operación de convolución entre el kernel y una porción de la imagen, ésta posee una memoria intermedia *internal_mem* la cuál almacena columnas de la imagen de forma tal que la cantidad de columnas almacenadas coincidan con el ancho del kernel. La convolución se realiza mediante unidades de suma y multiplicación ((*MAC_unit*)) que llevan a cabo la operación Eq. 2.

$$out \leftarrow \sum_{i=0}^{SK-1} \sum_{j=0}^{SK-1} K[i,j] \times M_i[i,j], \tag{2}$$

donde K representa al kernel y la sub-matriz M_i representa la porción de imagen a procesar.

3) Aritmética : La memoria *kernel_mem* almacena valores de 16 bits en punto fijo, de los cuales utiliza los 8 bits menos significativos para la parte fraccionaria y el resto para la parte entera. Estos valores se representan en complemento a la base para poder así dar soporte a números negativos. Este tipo de representación se la denomina de *punto fijo Q7.8* [8]. De esta forma el sistema acepta *kernels* con valores fraccionarios y valores negativos.

El rango de los valores del kernel pueden estar entre $[-128; 127{,}9960938]$ con precisión 2^{-8}.

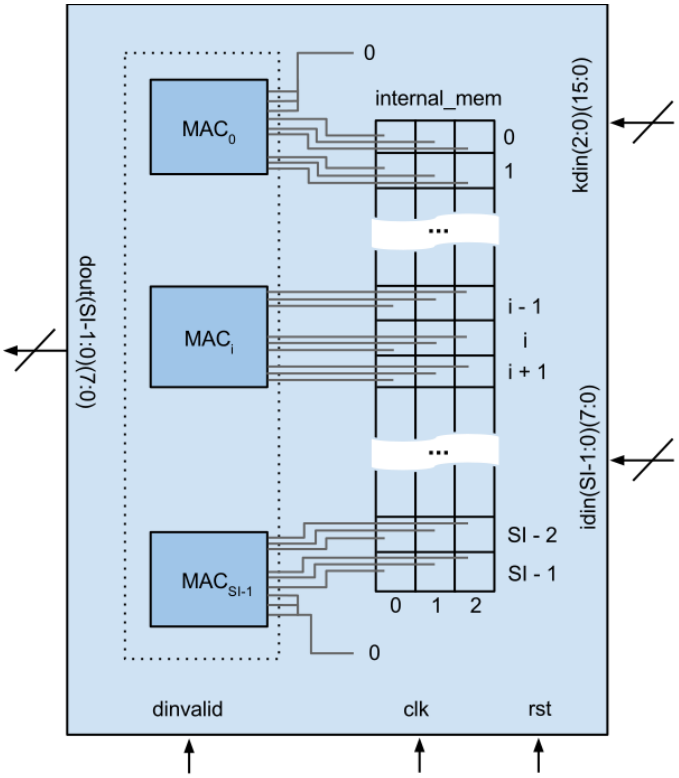


Figura 2: Esquemático de la unidad de procesamiento (*processing_unit*) parametrizada para $K = 3$.

La memoria *img_mem* almacena valores de 8 bits (entero sin signo) para poder representar los valores de las imágenes en escala de gris, es decir, valores de 0 a 255.

La *MAC_unit* realiza la multiplicación de los valores de 16 bits del kernel con los de 8 bits de la imagen, y luego los acumula. Al utilizar aritmética de complemento a la base se logra mayor simplicidad en los cómputos. Las multiplicaciones resultan en números de 24 bits ($16bits \times 8bits$) pero luego deben ser acumulados. El dimensionado de la cantidad de bits del acumulador depende de SK, como decisión de diseño se define un acumulador de ancho 32 bits para dar soporte a $SK \leq 16$. El resultado final esta compuesto por los 8 bits menos significativos de la parte entera del acumulador, si hay *overflow* entonces el valor se establece al máximo representable por 8 bits (255), en caso de resultar en un número negativo entonces se establece a 0.

4) Control: La unidad de control, denominada *control unit*, se encarga de coordinar el funcionamiento de los componentes del *datapath*.

El comportamiento generado es el siguiente: la unidad de memoria *img_mem* ira realizando corrimientos de columnas, permitiendo leer el contenido de la columna (de SI elementos) más significativa de forma paralela. La *processing_unit* recibe de forma paralela la columna de alto SI procedente de la unidad *img_mem* y la ira almacenando en la *internal_mem* de ancho SK. Esta unidad tendrá SI *MAC_unit*s, una por cada elemento de la columna (ver Figura 2). Cada *MAC_unit* toma los respectivos valores de la *internal_mem* y los opera con los valores que la unidad de procesamiento obtiene del *kernel_mem*. También la *control_unit* se encarga de almacenar los resultados de la *processing_unit* en la memoria de salida, de tamaño $SI \times SI$, denominada *result_mem*.

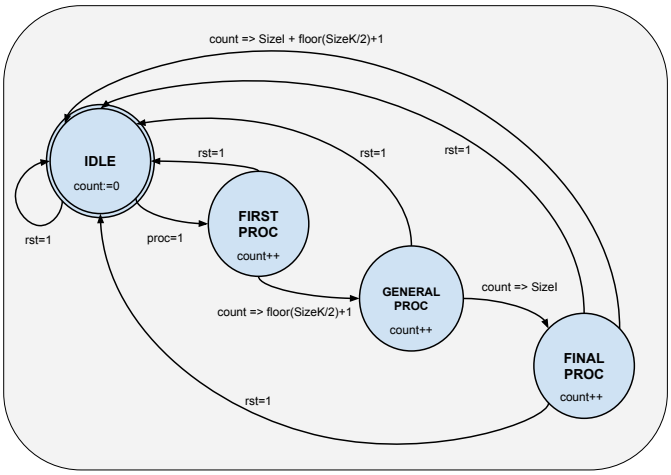


Figura 3: Maquina de estados simplificada implementada por la unidad de control.

Para lograr la dinámica descripta, la unidad de control deberá estimular correctamente los módulos involucrados. La unidad de control se modela mediante una maquina de estados finito, la cual se muestra de forma simplificada en la Figura 3. La lógica de salida de cada estado se explica a continuación:

- **IDLE:** cuando la unidad de procesamiento recibe la señal de *rst*, la maquina de estados pasa a este estado, el cual inicializa un contador *count* en cero. La maquina se queda en este estado hasta que se detecte la señal de entrada *proc* en valor alto.
- **FIRST PROC:** la lógica de este estado inicia la tarea de procesamiento, cargando las primeras columnas de la imagen en la unidad de procesamiento, para luego computar correctamente los primeros valores resultantes de la convolución entre el kernel y la imagen. Una vez cargado el número mínimo de columnas de la imagen necesarias para realizar el filtrado, éste se lleva a cabo para luego indicarle a la memoria de salida que almacene la columna de resultados. La lógica de próximo estado se realiza controlando el valor del contador *count* como se indica en la Figura 3.
- **GENERAL PROC:** la lógica de este estado se encarga de comenzar a pasar las columnas de bytes desde la memoria de entrada hasta la unidad de procesamiento, indicarle a esta ultima que opere, y luego le indica a la memoria de salida que almacene la columna de resultados. Este estado hará que la unidad de procesamiento reciba hasta la ultima columna de la imagen, controlando esto con el valor del contador *count*.
- **FINAL PROC:** la lógica de este estado se encargará de completar el borde final de la imagen con ceros para poder calcular correctamente la convolución en

este borde. Esto lo hará activando la respectiva señal de control para pasar la cantidad de columnas con cero necesarias según el tamaño del kernel utilizado.

5) Análisis de Complejidad : Se lleva a cabo el análisis de orden de ejecución con el fin de poder estimar cual sera el comportamiento final del sistema previo a su implementación.

Como unidad básica de análisis se considera el tiempo que requiere el bloque *MAC* en procesar los datos y entregar un dato valido; dado que estos bloques son puramente combinacional, su orden de ejecución es igual a un valor constante $O(MAC) = O(1)$.

El análisis de complejidad del sistema se realiza sobre el bloque *processing_unit*, dado que el modo de trabajo del sistema es similar al de un gran *shift register*, el mismo debe realizar una cantidad de iteraciones igual al tamaño de la imagen SI. A éste valor se le agrega el retardo correspondiente a operar los bordes de la imagen con el kernel ($\lceil SK/2 \rceil$), siendo el número de iteraciones total $N = (SI + \lceil SK/2 \rceil)$. Esto es una función lineal $f(N)$ con orden de ejecución $O(n)$.

De lo anterior se desprende:

$$O_{total} = O(f(N) * MAC) \quad (3)$$
$$O_{total} = O(f(N)) * O(MAC) \quad (4)$$
$$O_{total} = O(n) * O(1) \quad (5)$$
$$O_{total} = O(n) \quad (6)$$

II-C. Implementación en HDL

La implementación del diseño en HDL se llevó a cabo describiendo los módulos en VHDL. Se propone un diseño *top-down*, el cual consiste en comenzar a trabajar desde la entidad superior (contenedora), denominada *img_filter*, prosiguiendo por el resto de los bloques de menor jerarquía.

Primero se codifican los *dummies* de cada bloque del diseño, donde un *dummie* contiene solo puertos de entrada y salida sin funcionalidad definida (o básica). Estos *dummies* se conectan con el fin de analizar la arquitectura completa, de esta manera se evidenciara en una etapa temprana posibles errores de diseño.

Adicionalmente se define un paquete de VHDL, *img_filter_pkg*, en el cual se especifican distintos tipos de datos y constantes globales utilizadas por la mayoría de los bloques del diseño. De esta manera se controla la parametrización del diseño desde un lugar centralizado. En este paquete se definen las constantes de tamaño del kernel (SK), tamaño de la imagen (SI) y cantidad de bits utilizados para representar un pixel (*tByte*).

Finalmente, se implementa la funcionalidad de cada uno de los *dummies*, que gracias a esta organización se puede realizar en paralelo.

II-D. Verificación Funcional

Una vez codificado cada uno de los bloques, se prosigue con su verificación funcional a nivel de unidad, utilizando un *testbench* desarrollado en el lenguaje de verificación de hardware (HVL) *SystemVerilog*. Por último, finalizada la verificación de unidad de cada uno de los bloques se continua con la verificación de integración del sistema.

Para la verificación general del sistema se implementa un entorno de verificación que permite estimular el diseño con una imagen y un kernel, para luego comparar el resultado con el producido por un modelo de referencia que es estimulado con la misma imagen y mismo kernel.

El entorno de verificación esta conformado por: un *testbench* implementado en *SystemVerilog* y scripts de *Octave* para interactuar con el modelo de referencia. Un esquema de éste se puede observar en la Figura 4.

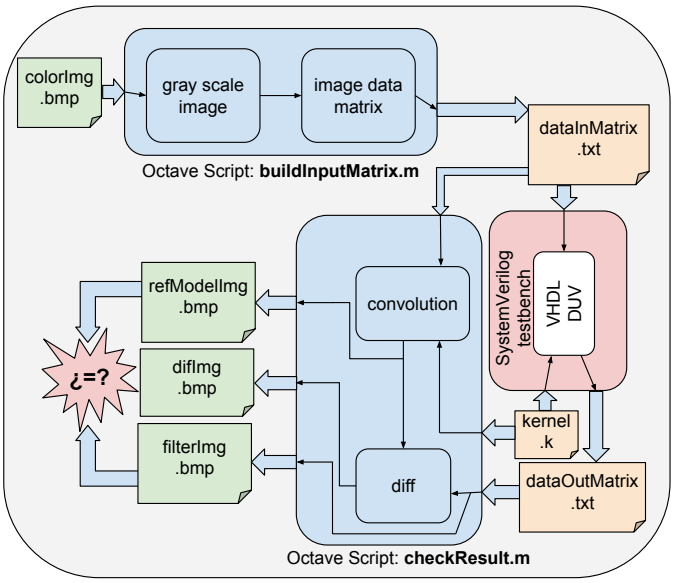


Figura 4: Esquema del entorno de verificación.

El script matemático de entrada, *buildInputMatrix.m*, recibe una imagen color RGB en formato BMP de tamaño $SI \times SI$, *colorImg.bmp*, la convierte a escala de grises y almacena la matriz de valores de los pixels en un archivo de texto, *dataInMatrix.txt*.

Posteriormente el *testbench* (Figura 5) realiza la lectura de los valores del archivo *dataInMatrix.txt*, y junto con los del kernel son cargados a las respectivas memorias internas. Da inicio al procesamiento de la imagen para a continuación leer el resultado desde la memoria de salida, que luego escribe en un archivo de salida, *dataOutMatrix.txt* en formato de matriz.

Para facilitar el trabajo, el *testbench* cuenta con un *Bus Functional Model* (BFM) [9] implementado en la interfaz mediante un conjunto de tareas que se encargan de resolver los detalles de bajo nivel, como temporizado y activación de señales. Por ejemplo, la tarea *write_img(..)* recibe un puntero a la matriz (que representa la imagen de entrada) y coordina las señales para ingresarla al DUV (*Design Under Verification*). Estas tareas aumentan el nivel de abstracción facilitando la implementación de capas superiores como el *Test*.

El archivo de salida producido por el *testbench* es procesado por un segundo script matemático, el cual también recibe

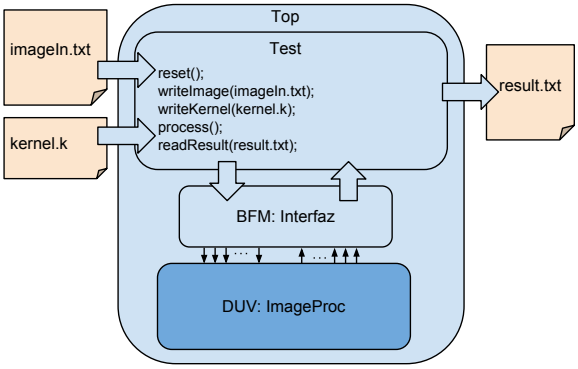


Figura 5: Arquitectura del *testbench* implementado en *SystemVerilog*.

los archivos *dataInMatrix.txt* y *kernel.k* con el fin de calcular la convolución mediante la función *conv2(...)* de *Octave*. La función *conv2(...)* es el modelo de referencia contra el que se compara el resultado del sistema.

Finalmente éste último script genera tres imágenes: una imagen con la convolución del modelo de referencia, una imagen con el resultado del filtro HDL y una imagen con la diferencia entre el modelo de referencia y el resultado del filtro.

III. Resultados

III-A. Simulación

Se simuló el comportamiento del sistema a nivel RTL con diferentes tamaños de imágenes y *kernels* utilizando la herramienta *ModelSim* de *Mentor Graphics* y el tesbench planeteado anteriormente.

Se utilizaron dos imágenes, la primera (figura 6a) la cual tiene un tamaño de 64×64 y la segunda de un tamaño de 128×128 (figura 6b). A ambas figuras se le aplicó un conjunto de *kernels*, uno de estos se puede observar en el Cuadro I. Los resultados de dichas operaciones se pueden apreciar en los Cuadros II y III demostrando el correcto funcionamiento del filtro.

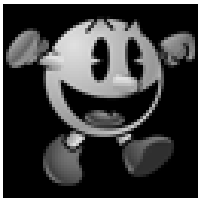
Cuadro I: Kernel Pasa Alto de 3x3.

-1/9	-1/9	-1/9
-1/9	8/9	-1/9
-1/9	-1/9	-1/9

III-B. Análisis de Complejidad

Con el fin de validar lo planteado en la sección II-B5, se realizó el análisis de tiempo de ejecución para una imagen de $SI = 64$, un kernel de $SK = 3$ y un reloj de 100 MHz (frecuencia utilizada en la simulación).

El tiempo que tarda la *MAC_unit* en dar una salida valida es de un ciclo de reloj, $T(MAC) = 10$ns; por lo tanto el tiempo teórico de ejecución es:

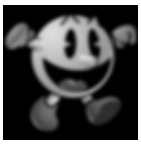


(a) Imagen de entrada 64×64.

(b) Imagen de entrada 128×128.

Figura 6: Imágenes de prueba.

Cuadro II: Resultados de filtrado de imágenes de 64x64 y Kernels de 3x3.

Filtro	Modelo de Referencia	Resultado
Pasa Alto		
Pasa Bajo		
Borde		
Gausiano		

Cuadro III: Resultado de filtrado de imagen de 128x128 y Kernel de 5x5.

Filtro	Modelo de Referencia	Resultado
Gausiano		

$$T_{teo} = T(SI + \lceil SK/2 \rceil) * T(MAC) \quad (7)$$

$$T_{teo} = ((64 + 3/2) * 10) \text{ ns} \quad (8)$$

$$T_{teo} = 650 \text{ ns} \quad (9)$$

El funcionamiento del sistema se observa en la Figura 7 donde se aprecia el inicio del procesamiento (señal *proc* con valor alto durante un ciclo de reloj). Luego se procesa la imagen (*dinvalid* alterna entre valores altos y bajos) y una vez finalizado, (*doutvalid* se encuentra en valor alto) se aprecia un tiempo de simulación de $T_{sim} = 1300\ ns$.

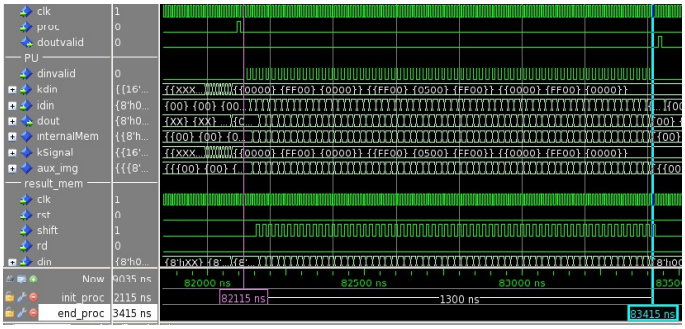


Figura 7: Vista de señales de simulación de *processing_unit*.

Se puede observar que las estimaciones calculadas previamente discrepan en que $T_{sim} = T_{teo} * J$, donde J es una constante de temporizado del control. En este caso $J = 2$ ya que el control tarda dos ciclos en introducir datos validos a la *processing_unit*.

III-C. Síntesis lógica

Con el objetivo de obtener una estimación de área y de velocidad de procesamiento, se realizó la síntesis lógica (tanto para FPGA como para ASIC) de la *MAC_unit* descripta en la sección (II-B3). Se decidió sintetizar ésta unidad debido que su naturaleza permite que el diseño se vuelva parametrizable con el fin de dar soporte a distintos tamaños de imágenes, dando una idea inicial de cual sería el tamaño del sistema en distintos contextos. Otro factor que nos llevo a tomar esta decisión es que es el bloque combinacional de mayor longitud (siendo critico en el *worst time* del *datapath*).

La síntesis para ASIC se realizó con la herramienta *Design Compiler* de *Synopsys* utilizando los scripts de la *Reference Methodology* [10] y la biblioteca de tecnología de 130 nm de TSMC. Se obtuvo como resultado un área total de 3948 *gates* y una frecuencia de operación de 181 MHz.

La síntesis en FPGA se realizo sobre una *Xilinx Virtex5 (xc5vlx110t)* obteniéndose que la unidad ocupa un total de 689 *LUT*s.

IV. Conclusión

En este trabajo se especificó, diseñó, implementó y verificó un filtro por convolución de imágenes en escala de grises. Se diseñó la arquitectura de manera *top down* para lograr una mejor interacción y avance en el flujo de desarrollo. El filtro implementado es parametrizable dando soporte a distintos tamaños de imágenes y distintos tamaños de kernels, aceptando valores negativos y fraccionarios para este último. El diseño fue implementado con la aritmética adecuada en función de los posibles valores del kernel y de las imágenes.

El resultado de la verificación indicó el correcto funcionamiento del sistema. Cada bloque se verificó mediante testbenchs unitarios y posteriormente se verificó la integración del sistema utilizando un entorno basado en scripts aritmeticos y un testbench implementado en *SystemVerilog*.

Los resultados de la síntesis lógica de la *MAC_unit* sugieren un buen desempeño en velocidad de procesamiento, indicando que el mísmo es un bloque adecuado para utilizar en sistemas de tiempo real. En una estimación ideal, donde no se tiene en cuenta el tiempo que insume el *datapath*, el sistema podría procesar una imagen de 1.280 columnas en $7,07 \times 10^{-6}$ segundos, lo que equivale a procesar 141.442 imágenes por segundo o *fps* (*frames per second*).

Como desarrollo futuro se plantea dar soporte al ingreso de datos en modo rafaga e implementar la *MAC_unit* en *pipeline* con el fin de obtener un mayor *throughput*.

Agradecimientos

Los autores desean agradecer al Dr. Claudio Delrieux por introducirlos y capacitarlos en la temática de procesamiento digital de imágenes lo cual dio origen a este trabajo. Adicionalmente los autores desean agradecer al Sr. Scott Hickey, *Aplications Consultant* de *Synopsys*, por su asistencia y recomendaciones para llevar a cabo la síntesis lógica del diseño.

Este trabajo ha sido financiado por el Instituto Nacional de Tecnología Industrial (INTI) y por FSTICS 001 "TEAC", PICT 2010 2657 y PAE 37079.

Referencias

[1] J. Mori, C. Sanchez-Ferreira, D. Muñoz, C. Llanos, and P. Berger, "An unified approach for convolution-based image filtering on reconfigurable systems," in *Programmable Logic (SPL), 2011 VII Southern Conference on*, April 2011, pp. 63–68.

[2] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[3] V. Moshnyaga, K. Suzuki, and K. Tamaru, "A new architecture for in-memory image convolution," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, 1998*, vol. 5, May 1998, pp. 3001–3004 vol.5.

[4] M. Azizabadi and A. Behrad, "Design and VLSI implementation of new hardware architectures for image filtering," in *8th Iranian Conference on Machine Vision and Image Processing (MVIP)*, Sept 2013, pp. 110–115.

[5] H. Jiang and V. Owall, "FPGA implementation of real-time image convolutions with three level of memory hierarchy," in *IEEE International Conference on Field-Programmable Technology (FPT)*, Dec 2003, pp. 424–427.

[6] J. Mori, C. Llanos, and P. Berger, "Kernel analysis for architecture design trade off in convolution-based image filtering," in *Integrated Circuits and Systems Design (SBCCI), 2012 25th Symposium on*, Aug 2012, pp. 1–6.

[7] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs*. Kluwer Academic Publishers, 2002.

[8] W. Padgett and D. Anderson, *Fixed-Point Signal Processing*, ser. Synthesis lectures on signal processing. Morgan & Claypool, 2009.

[9] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[10] Synopsys. (2015) Solvnet - rmgen. [Online]. Available: https://solvnet.synopsys.com/

SystemC/TLM flow for SoC Design and Verification

Manuel F. Soto
CMNB-BHI
INTI
Email: msoto@inti.gob.ar

J. Agustín Rodriguez
IIIE-CONICET
UNS
Email: jrodriguez@uns.edu.ar

Pablo R. Fillottrani
LISSI-DCIC
UNS - CIC
Email: prf@cs.uns.edu.ar

***Abstract*—As systems grow in complexity, their verification becomes a bottleneck on the design flow. In this paper we propose a top-down methodology to perform the complete flow from specifications to Register Transfer Level (RTL). Different abstraction levels such as Transaction Level Modeling (TLM) allows early system verification (with simulation or formal methods), reducing the risk of long redesign cycles. The methodology is validated by showing a case study.**

I. Introduction

During the last years the electronics industry has seen an exponential growth in system's design and verification complexity [1], [2] as predicted by Moore. The possibility to embed processors, different application specific sub-systems and mixed-signal modules, proposes a design space with increased challenges. Both the number and variety of components and the possible interactions between them, require nowadays better techniques to conceive, specify and verify designs.

System level solutions for design need to take special attention to the overall functionality provided by the growing number of Intellectual Property (IP) blocks interacting through bus fabric technologies or networks on chip. Regarding functional verification, it is clear that techniques that proved to be effective at the RTL, are not being enough in this context given that verification is increasingly taking more time in the development flow [3], [4], [5].

In order to handle the challenges that emerge with IP block integration, new concepts and techniques have been proposed, often taking advantage of ideas and resources from other areas of engineering. In this way, industry has been shifting towards newer design paradigms, leaving away the classical design flows to adopt modern approaches that are better suited to manage the complexity of Systems on a Chip (SOC) [6] [7].

This paper introduces a development method for system level design and verification, trying that the flow could be easily adapted in several that require a lot of designs where time-to-market will play a critical role, such as IoT. The flow is applied in the development of an SoC targeting data-center sorting requirements. This case study involved the definition of a system architecture based on third party IPs and an in-house designed sorting core. The system functionality was expressed resorting to the concept of *transaction*. Each functionality was represented by sequences of operations and interactions between components hiding low level implementation details. These transaction level models resulted convenient to define the system level control routines and to verify sub-module integration. The AMBA bus fabric selected for design forced to adapt each IP to implement the communication protocol, and mixed SystemC-RTL simulations proved to be the right approach for unit verification.

The remainder of the paper is organized as follows. The next section presents related work. The third section introduces preliminary concepts. Section 4 presents the proposed methodology describing its different stages for design and verification. Sections 5 and 6 show how it was applied in our case study, from system level specification to RTL, both for design and verification respectively. Finally the last section closes with conclusions and future work.

II. Related Work

Most of the reported work in this context, proposes to rise the level of abstraction by introducing new concepts related to particular characteristics of system integration. Several of the methodologies are particularly based on well known software engineering techniques, like Object Oriented Programming (OOP) and UML.

In [8], Vernalde et al. propose a three stage top-down design and verification flow. The first stage uses C++ to define a functional description of the system, the second one refines this model by introducing timing (cycle-accurate) and structural (architecture) information. Finally an even more accurate model, called fixed point design, is the final version of the system. This design flow is implemented into an environment called *OCAPI* to support the design of complex digital systems.

Sinha et al [9], employ UML diagrams with custom extensions to specify digital system architectures correctly, they propose a top-down design flow as the solution to digital system design complexity.

Doucet et al. [10] suggest a four stage methodology considering the structural information and the functionality of the system; the first and second stage involve the description of intended functionality (computational model) and the hardware needed to support it (design unit). The third stage performs the *binding* between computational model and design units, and finally in the fourth stage a refinement and optimization of the system is done. The language used for each step of the flow is SystemC.

Habbibi et al. [11] propose a parallel flow to do design and verification at the same time. An initial description of the system is implemented in AsmL (Abstract State Machine

Language). This high level model is verified by checking combinational and sequential properties. Later the model is translated into SystemC and is used to generate assertions which are compared with the original design. This is done to check the correctness of different abstractions used in the verification process.

Tomasena et al. [12] propose a framework built in SystemC with support to Assertion Based Verification. They tried to decouple the design flow and the verification flow (this is possible thanks to data introspection capabilities of SystemC Verification Library (SCV)).

Finally Oliveira et al. [13] proposes the use of Universal Verification Methodology (UVM) as a library of SystemC, to make testbench from transaction level to RTL. The proposed System Verification Methodology provides the essential blocks to build a testbench with randomization and coverage in an efficient way, including a macro block to promote the reuse of verification context. This work is based on UVM and Open Verification Methodology.

The design and verification flow presented in this work, proposes to enhance the abstraction level to address the development of complex systems, this is similar to [8], [10]. It based on SystemC and on the well-accepted design flow proposed by Cai et al. [14]. It incorporates the use of formal methods like Habbibi et al. [11] do, but in the second stage of the flow, taking ideas from [15] which are adapted from SystemC to TLM. The verification at the unit level is done with mixed (systemC/RTL) simulations, taking concepts from the work by Oliveria et al.[13] for the testbench design.

III. Background

A. SystemC

SystemC [16] is a system level design language based on C++. It allows to model and execute both hardware and software at several levels of abstraction [17]. The SystemC core basically provides a set of modeling resources (modules, processes, channels and hardware oriented data types) and an event based simulation kernel. A classical design in SystemC may consist of one or more modules defining the structure of the system, a set of process modeling the functionality of each module, channels between them for communication and triggered events for synchronization [18].The simulation is managed by the SystemC scheduler; it controls discrete time advance, the execution of processes and handles channel events.

B. TLM

Transaction Level Modeling [19] is one of the modern approaches for system level design. In [14] Cai et al. present a clear definition of TLM's taxonomy and emphasize the benefits of its adoption. SystemC supports TLM with an additional library. Designing in TLM allows to represent the interaction between modules at a higher level of abstraction than classic RTL design. Communication is represented as function calls without consideration of any timing or bit accurate information. Since simulation times get reduced by hiding implementation details, TLM is often adopted for early system evaluation and performance analysis.

Transaction modeling in SystemC is supported by a set of primitives grouped into what is called an inter-operative layer. The inter-operative layer consists of: sockets, interfaces, payloads and a base protocol. Sockets are the entry point of modules (initiator and target) into the transaction mechanism. The interface provides the blocking and unblocking calls to perform data transportation. The payload is actually the data structure that holds data or commands sent and received between modules. And the base protocol is a set of rules to manage the communication to get high performance in simulations.

C. Spin & PROMELA

Spin is a formal verification tool that supports systems involving asynchronous components [20], [21]. Spin verification models are focused on proving the correctness of process interactions and for this purpose they attempt to avoid as much as possible from internal sequential computations.

Spin supports modules described PROMELA, a verification modeling language. Even its syntax is similar to a programming language like C, it is particularly used to describe finite automatons. Its semantics are influenced by Dijktras's guarded language [22] and Hoare's sequential processes for communication [23]. PROMELA allows dynamic creation of concurrent tasks for modeling, eg. distributed systems. Process interactions can be specified with rendezvous primitives or with asynchronous message passing through buffered channels or shared memory, and with any combination of them.

IV. Design and Verification Methodology

The design and verification flow (Fig. 1), adopts ideas from [14], where a design evolves through three levels. In this case the following levels or stages are the Specification Model, the Transaction Model and finally the Implementation Model.

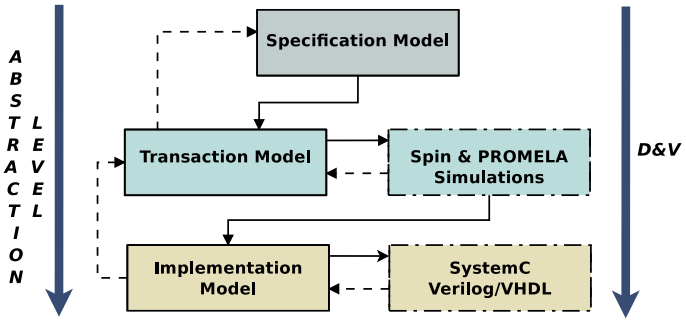


Fig. 1: Design and Verification flow.

The *Specification Model* is intended to provide design and verification workgroups an initial approach to the system. On this level, the main system's entities and their functionalities are expressed. Each relevant type of processing element and memory block is represented with a diagram equivalent to the UML class diagram. Each class of component defines the fundamental data (structures) that are involved in a component entity, and a set of routines that identify the processes which involve changes in the internal state as well as input-output. This description gives a general vision of the system and the

relations between its sub-systems allowing to define the system level architecture.

The *Transaction Model* addresses the interaction between system's components (similar to *Bus Arbitration Level* in [14]). A *transaction* is defined in this work as a sequence of control and data transfer operations that one or more components should do to implement a given functionality, hiding low level details regarding signal timing or bus handshake. At this level, the UML's sequence diagram is used. This kind of diagram is helpful to appreciate the nature of the interactions between blocks. Additionally, a TLM representation results convenient to elaborate on the architecture of the system, explore the design space and analyze performance estimations. Finally, it allows to perform early verification of the system by simulations or based on formal methods; being the latter what was particularly exploited in this work resorting to SPIN&PROMELA.

The *Implementation Model* is similar to the RTL level. In this stage a clear difference between data-path and control is appreciated. The description of each component or entity depends on whether it is an in-house designed block or a third party IP. In case of a designed core, SystemC with accurate timing information is employed; in case of an IP, commonly specified as a VHDL or Verilog module, a SystemC wrapper is implemented to allow integration and verification with the rest of the system. The Implementation Model is verified both at the unit level and integration through simulation.

V. Case Study: an SoC for networking

As a case study for the presented system level design methodology, an SoC targeting data-center sorting requirements was developed. This system integrates several IPs. The fundamental operation of this system it to receive Ethernet frames, sort them and return the sorted data packets into de network. In this section all the design steps involved in this development following the introduced flow are detailed. In the next section verification is addressed.

A. Specification Model

In order to address the design intend, an SoC with sorting resources is designed. The system architecture is shown in Fig. 2 (Only one sorting unit is included, however it is a scalable design provided that enough input-output and internal bus resources are configured for a given number of sorting units). It integrates five main resources: (1) Sorting cores, (2) DMA for data transfer through the internal bus, (3) Shared Memory, (4) Ethernet controller for input-output and (5) an System Controller & Arbiter to manage the overall behavior.

For each class of device or component, *high level operations* are identified as it was introduced in the previous section. For example, a DMA instance would provide functionalities that can be summarized by the following tasks:

- *write_reg_out(X:device)*= Read the data from internal buffers and put them in output registers.
- *read_reg_in(X:device)*= Read data from input register and put them in internal buffers.
- *read_from_x_write_y(X:device, Y:device)*= Read the data from X device and write on Y device.

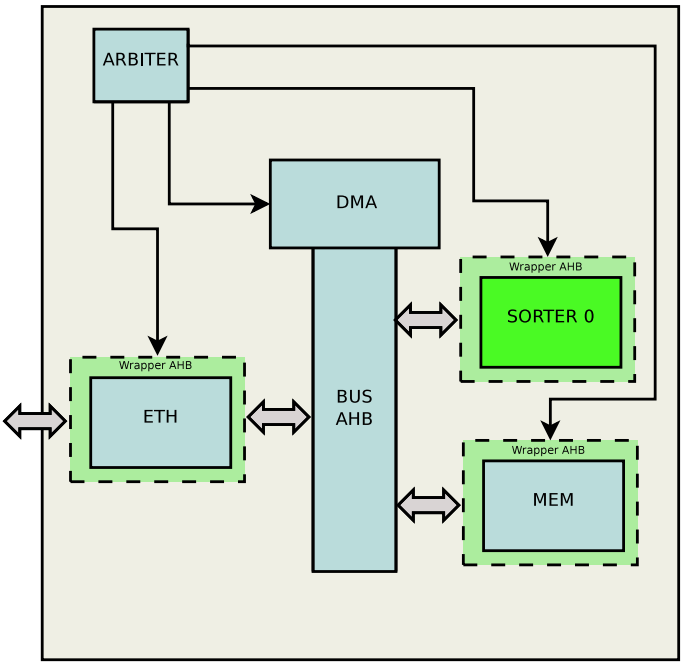


Fig. 2: System Architecture with one Sorter.

B. Transaction Model

Four fundamental transactions were identified that support the required functionality with the given resources:

- (1)**EwM**: moves received data packets and stores them into shared memory;
- (2)**S**$_n$**rM**: gets unsorted data packets from shared memory and sends them to an available sorter core;
- (3)**S**$_n$**wM**: as each sorter core finishes with a data-packet, results are stored back into shared memory;
- (4)**ErM**: sends sorted data-packets from memory to the Ethernet controller transmit buffers.

In transaction level modeling, even though the presented abstractions provide significant information regarding the system functionality, these models need to be implemented too, not with the level of detail that involves an RTL, but with all the functionalities needed to experiment with the system. The framework that allows to test the defined types of transactions is sometimes called virtual prototype.

The SystemC implementation of this kind of framework needs to employ bus architectural design principles [24]. One of the fundamental issues in TLM system specification is the bus design, which can be of hub or router type. For this development a router bus was selected (for the TLM system model), combined with sockets and blocking transfer functions, it helped to minimize the details for communication between components resulting in a loosely timed model.

Each of these transactions can be described with a *Sequence Diagram* similar to *UML's* diagrams which were modified to represent the parallelism of the system in a convenient way. This kind of representation shows how system

components interact to implement a transaction. For example Fig. 3 presents the Sequence Diagram for transaction **EwM**.

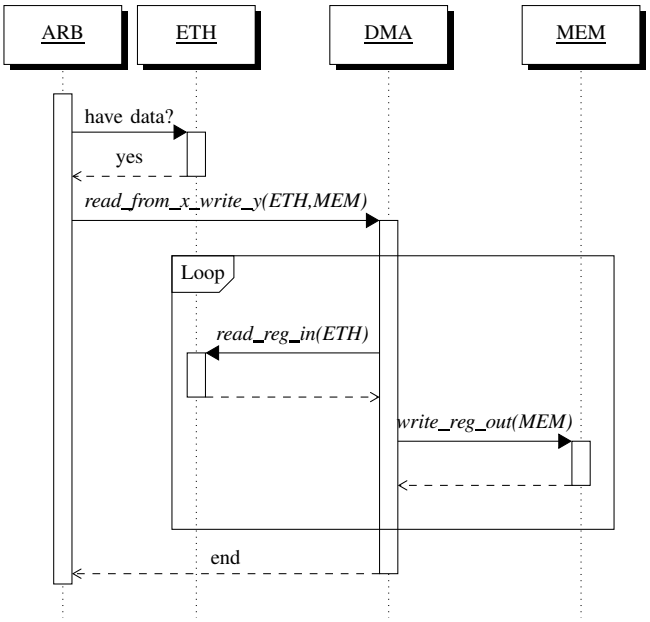


Fig. 3: Transaction **EwM**.

In the diagram it can be appreciated how this type of transaction evolves in time and which task is performed by each of the involved components. The arbiter checks if any data packet is available in the ETH controller receive path. As one is available, it configures the DMA to perform a transfer between the ETH buffers and the system's shared memory. The transaction continues by iteratively reading from ETH buffers and writing into memory until the complete frame is done. It finishes with the synchronization messages from DMA to the arbiter.

Finally, transactions help to represent system functionality. For example, for the most simple version of this design (single *Sorter* core), the sequence S_i defines the overall operations to evaluate for a given frame i.

$$S_i = \{EwM \Rightarrow S_0rM \Rightarrow S_0wM \Rightarrow ErM\}$$

C. Implementation Model

At this stage of the flow, as it was exposed in Section IV the *RTL* description of each IP is needed.

The *ETH Controller* is a third party IP, the *Embedded Tri-Mode Ethernet MAC Wrapper v1.7* provided by the *Cores Generator Tool* from *Xilinx*. This IP was configured with one *EMAC*[1] and a MII[2] physical interface. It was generated in Verilog.

The *DMA* for *AHB Bus* was obtained from *OpenCores*; the chosen version is the one uploaded by *Provartec.Inc*. It was configured with a single channel and one AHB dual port, supporting writes and reads in parallel. The Verilog version was used.

Finally the *Sorter*, the *Memory* and the *System Controller & Arbiter* the are custom designs. They were implemented in SystemC, analyzed, redesigned, verified and finally translated to VHDL.

The Implementation Model also includes the AHB wrappers in order to translate the IP's interface protocols to the system's bus.

[1]Ethernet Media Access Control

[2]Media Independent Interface

VI. Verification & Results

As introduced in Section IV verification is done at transaction level and at implementation level, for the first one, it is performed by simulations and with formal methods. For the last one, the verification environment (Fig. 6) is employed, it supports mixed simulations allowing the refinement of each system's custom IP.

A. Verification at the Transaction Level

The verification at this stage of the methodology is performed in two different ways.

In the first place, verification was performed by simulations. Simulations targeted the following sequences of transactions and results were correct:

- $S_{00} = \{EwM \Rightarrow S_0rM \Rightarrow S_0wM\}$
- $S_{01} = \{ErM\}$
- $S_{02} = \{EwM \Rightarrow S_0rM \Rightarrow S_0wM \Rightarrow ErM\}$

In the second place, Formal Methods were applied. Each of the system's component and their associated methods, the SystemC scheduler and the clock generator were mapped to PROMELA and analyzed using Spin Model Checker.

An example of the automaton obtained for the memory block and its functionality are shown below in Fig. 4 and Fig. 5 respectively.

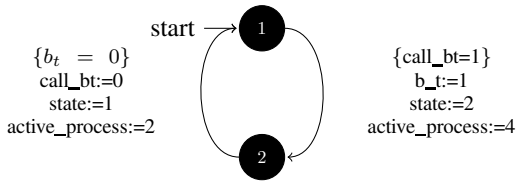


Fig. 4: Memory automaton.

Table I presents the results of the property checking performed by Spin, which allowed to discover a hidden bug not exposed by simulations. Tests 1-2 correspond to the first version and 3-4 to the fixed version.

B. Verification of the Implementation Model

The functional verification of the implementation model was performed with the verification environment presented in Fig. 6. First, the unit verification was done, and then integration verification.

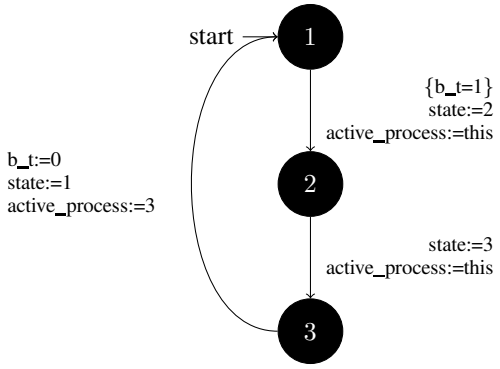


Fig. 5: Blocking transport function automaton.

TABLE I: Spin results.

#	Property	States	Time[s]	Result
1	Safety	62	< 0.1	Fail
2	Acceptance	62	< 0.1	Fail
3	Safety	45	< 0.1	Success
4	Acceptance	46	< 0.1	Success

This environment was designed using the SCV library. It provides data introspection (similar to Verilog PLI [25]) to manipulate data in a consistent way , it also provides (constrained) randomization, exception handle and the feature to perform the recording of variables, and assertions. The assertions (Native support) provide a simple way to observe the system behavior, supporting four different levels for assertion reporting.

This environment provided an early verification resource for the system described in SystemC. Then, it was also used for VHDL/Verilog modules in mixed mode (SystemC-RTL).

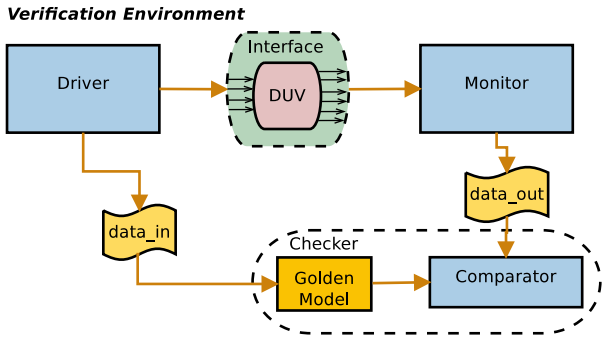


Fig. 6: Verification Environment.

The DMA module and the ETH module are third-party modules, therefore the integration with the others modules were the main goal to verify. First the ETH module was verified with the testbench provided by Xilinx and then with the verification environment in order to check integration aspects. Secondly, the DMA was verified entirely with the verification environment, in Table II test results for the DMA are showed.

TABLE II: DMA tests results.

#	DirR	DirW	Size	Obs	Result
1	0x0	0x0	0xA	-	Success
2	0x0	0x20	0xA	-	Success
3	0x0	0x40	0xA	-	Success
4	0x20	0x0	0xA	-	Success
5	0x40	0x0	0xA	-	Success
6	0x0	0x0	0xFF	-	Success
7	0x0	0x20	0xFF	Due to the size of buffer is bigger than the size between DirR and DirW a repetitive patron is observed.	Success*
8	0x0	0x40	0xFF	Due to the size of buffer is bigger than the size between DirR and DirW a repetitive patron is observed.	Success*
9	0x20	0x0	0xFF	-	Success
10	0x40	0x0	0xFF	-	Success

The set of tests used to verify the Sorter are observed in Table III. These are all values positive, all values negative, ascending and descending values, values with constraint $(-999 <= value <= 999)$ and finally the number of incoming frames into the sorter is variable to simulate a real-work environment.

TABLE III: Sorter tests results.

#	Number of Frames	Test	Result
1	1	all positives	Success
2	1	all negatives	Success
3	1	ascending order	Success
4	1	descending order	Success
5	1	random with constraint	Success
6	variable	all positives	Success
7	variable	all negatives	Success
8	variable	ascending order	Success
9	variable	descending order	Success
10	variable	random with constraint	Success

Finally when the unit verification of all modules was done, the system integration and its verification proceeded. Integration tests are shown in Table IV.

TABLE IV: System tests results.

#	Test	Read Device	Write Device	Result	Obs
1	**EwM**(Sec V-B)	ETH	Mem	Success	-
2	**S$_0$rM**(Sec V-B)	Mem	Sorter	Success	-
3	**S$_0$wM**(Sec V-B)	Sorter	Mem	Success	-
4	**ErM**(Sec V-B)	Mem	ETH	Success	-

VII. Conclusions and Future Work

Nowadays the complexity of systems is a challenge for design and verification teams, the use of a methodology for the entire flow appears as the right solution to minimize the human error. A methodology for system level design and verification was proposed. It was applied to the development of a SoC focused on data-center sorting requirements. The opportunity to rise the level of abstraction, specially withthe transaction level, provided the possibility to perform an early analysis and verification of the proposed architecture.

Transactions proved to simplify and empower the verification task. Formal verification into the transaction level was

incorporated in the flow as a improvement compared with other flows. Finally the use of mixed mode SystemC-RTL simulations promote the re-utilization of testbench, reduce the effort in the environment design and test case development; proving to be the right approach for unit verification.

As future work there are many aspects to be addressed. The first one is to extend the support of formal methods in the *Transaction Model*. Also the use of formal methods on other stages of the flow, specially in the *Specification Model*, which is crucial. The analysis of coverage with SCV statements is also a line of work, as well as SystemVerilog covergropus, in order to improve the tests information at *Implementation Model* level. The application of the proposed methodology to other systems in order to find its application fields and its limitations. The methodology can also be extended with the incorporation of Universal Power Format (UPF) support [26], in order to support an early power statement analysis, since this is an interesting and promising research field. Finally the methodology may be used in different research fields such as Internet of Things (IoT) or Multi-Processors Systems on Chip (MP-SoC), given that the concept of transaction have proved relevant expressive power.

ACKNOWLEDGMENT

This work was partially funded by FSTICS 001 "TEAC", PICT 2010 2657 and PAE 37079. EDA Tools were provided by Synopsys and Mentor through University Program agreements.

REFERENCES

[1] R. H. J. M. Otten and P. Stravers, "Challenges in physical chip design," in *Proc of the 2000 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '00. Piscataway, NJ, USA: IEEE Press, 2000, pp. 84–92.

[2] C. Mead and L. Conway, *Introduction to VLSI Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1979.

[3] H. Foster. (2010, August) Redefining verification performance (part 2). Verification Horizon BLOG. Mentor Graphics. [Online]. Available: http://blogs.mentor.com/verificationhorizons/blog/2010/08/08/redefining-verification-performance-part-2/

[4] H. D. Foster, "Why the Design Productivity Gap Never Happened," in *Proc of the International Conference on Computer-Aided Design*, ser. ICCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, p. 581584. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561943

[5] W. K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[6] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, 1st ed. Springer, Mar. 2000. [Online]. Available: http://www.worldcat.org/isbn/0792378229

[7] T. Grotker, *System Design with SystemC*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[8] S. Vernalde, P. Schaumont, and I. Bolsens, "An object oriented programming approach for hardware design," in *Proc. IEEE Computer Society Workshop On VLSI*, 1999, pp. 68–73.

[9] V. Sinha, F. Doucet, C. Siska, R. Gupta, S. Liao, and A. Ghosh, "Yaml: a tool for hardware design visualization and capture," in *Proc. The 13th International Symposium on System Synthesis*, 2000, pp. 9–14.

[10] F. Doucet, V. Sinha, C. Siska, and R. Gupta, "Microelectronic system-on-chip modeling using objects and their relationships," in *Online Symposium for Electrical Engineers (OSEE)*, 2000.

[11] A. Habibi and S. Tahar, "Design for verification of systemc transaction level models," in *Proc of the Conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 560–565.

[12] K. Tomasena, J. Sevillano, J. Perez, A. Cortes, and I. Velez, "A transaction level assertion verification framework in systemc: An application study," in *Advances in Circuits, Electronics and Micro-electronics, 2009. CENICS '09. Second International Conference on*, Oct 2009, pp. 75–80.

[13] M. F. Oliveira, C. Kuznik, H. M. Le, D. Große, F. Haedicke, W. Mueller, R. Drechsler, W. Ecker, and V. Esen, "The system verification methodology for advanced tlm verification," in *Proc of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '12. New York, NY, USA: ACM, 2012, pp. 313–322.

[14] L. Cai and D. Gajski, "Transaction level modeling: An overview," in *Proc of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '03. New York, NY, USA: ACM, 2003, pp. 19–24.

[15] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, "A systemc/tlm semantics in promela and its possible applications," in *Proc of 14th Workshop on Model Checking Software SPIN (SPIN 2007)*, Berlin, Germany, 2007, pp. 204–222.

[16] Systemc open initiative. OSCI. [Online]. Available: www.accellera.com

[17] J. Stoppe, R. Wille, and R. Drechsler, "Automated feature localization for dynamically generated systemc designs," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, March 2015, pp. 277–280.

[18] H. Xiao, N. Wu, F. Ge, T. Isshiki, H. Kunieda, J. Xu, and Y. Wang, "Efficient synchronization for distributed embedded multiprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2015, cited By 0; Article in Press.

[19] Transaction level modelling reference manual. OSCI. [Online]. Available: www.accellera.com

[20] G. J. Holzmann, *Design and Validation of Computer Protocols*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.

[21] G. J. Holzmann, P. Godefroid, and D. Pirottin, "Coverage preserving reduction strategies for reachability analysis," in *Proc of the IFIP TC6/WG6.1 Twelth International Symposium on Protocol Specification, Testing and Verification XII*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1992, pp. 349–363.

[22] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975. [Online]. Available: http://doi.acm.org/10.1145/360933.360975

[23] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359585

[24] F. Ghenassia, *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[25] Verilog pli tutotial. asic-world. [Online]. Available: http://www.asic-world.com/verilog/pli.html

[26] F. Mischkalla and W. Mueller, "Advanced soc virtual prototyping for system-level power planning and validation," in *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2014 24th International Workshop on*, Sept 2014, pp. 1–8.

Bibliografía

[1] C. Mead and L. Conway, *Introduction to VLSI Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1979.

[2] R. H. J. M. Otten and P. Stravers, "Challenges in physical chip design," in *In Proc of ICCAD*. ACM press, 2000, pp. 84–91.

[3] H. Foster, "Wilson research group, 2012 functional verification study," Mentor Graphics Corp, Tech. Rep., 2013.

[4] A. B. Kahng, "The itrs design technology and system drivers roadmap: Process and status," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 34:1–34:6. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488776

[5] H. Zhaohui, A. Pierres, H. Shiqing, C. Fang, P. Royannez, E. P. See, and Y. L. Hoon, "Practical and efficient soc verification flow by reusing ip testcase and testbench," in *SoC Design Conference (ISOCC), 2012 International*, Nov 2012, pp. 175–178.

[6] R. Srivastava, N. Mudgil, G. Gupta, and H. Mondal, "Soc time to market improvement through device driver reuse: An industrial experience," in *Electronic System Design (ISED), 2012 International Symposium on*, Dec 2012, pp. 56–61.

[7] Accellera. (2009, May) Universal verification methodology. Accellera Organization Inc.

[8] Cadence. Open verification methodology. Cadence Design Inc.

[9] J. Wang, J. Shao, Y. Li, and J. Ding, "Survey on formal verification methods for digital ic," in *Internet Computing for Science and Engineering (ICICSE), 2009 Fourth International Conference on*, 2009, pp. 164–168.

[10] S. Vasudevan, *Effective Functional Verification: Principles and Processes.* Springer London, Limited, 2006. [Online]. Available: http://books.google.com.ar/books?id=wNWib4yqisUC

[11] The pentium chip story: A learning experience. Vince Emery. [Online]. Available: http://www.emery.com/1e/pentium.htm

[12] M. Harrand, J. Sanches, A. Bellon, J. Bulone, A. Tournier, O. Deygas, J.-C. Herluison, D. Doise, and E. Berrebi, "A single-chip cif 30 hz h261, h263, and h263+ video encoder/decoder with embedded display controller," in *Solid-State Circuits Conference, 1999. Digest of Technical Papers. ISSCC. 1999 IEEE International*, 1999, pp. 268–269.

[13] F. Ghenassia, *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[14] *IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual*, 2006.

[15] O. S. Initiative, "Osci standard, osci tlm-2.0 language reference manual," 2009.

[16] S. Abdi, G. Schirner, Y. Hwang, D. Gajski, and L. Yu, "Automatic tlm generation for early validation of multicore systems," *Design Test of Computers, IEEE*, vol. 28, no. 3, pp. 10–19, 2011.

[17] R. Jindal and K. Jain, "Verification of transaction-level systemc models using rtl testbenches," in *Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings. First ACM and IEEE International Conference on*, 2003, pp. 199–203.

[18] M. Sousa and A. Sen, "Generation of tlm testbenches using mutation testing," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '12. New York, NY, USA: ACM, 2012, pp. 323–332. [Online]. Available: http://doi.acm.org/10.1145/2380445.2380498

[19] W. Chen, X. Han, and R. Doemer, "Multicore simulation of transaction-level models using the soc environment," *Design Test of Computers, IEEE*, vol. 28, no. 3, pp. 20–31, 2011.

[20] F. Petrot, M. Gligor, M.-M. Hamayun, H. Shen, N. Fournel, and P. Gerin, "On mpsoc software execution at the transaction level," *Design Test of Computers, IEEE*, vol. 28, no. 3, pp. 32–43, 2011.

[21] N. Bombieri, N. Deganello, and F. Fummi, "Integrating rtl ips into tlm designs through automatic transactor generation," in *Design, Automation and Test in Europe, 2008. DATE '08*, 2008, pp. 15–20.

[22] S. Di Carlo, N. Hatami, P. Prinetto, and A. Savino, "System level testing via tlm 2.0 debug transport interface," in *Defect and Fault Tolerance in VLSI Systems, 2009. DFT '09. 24th IEEE International Symposium on*, 2009, pp. 286–294.

[23] R. Schaller, "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, Jun 1997.

[24] M. Riordan and L. Hoddeson, "The origins of the pn junction," *Spectrum, IEEE*, vol. 34, no. 6, pp. 46–51, Jun 1997.

[25] J. Hoerni, "Planar silicon diodes and transistors," in *Electron Devices Meeting, 1960 Internationa*, vol. 6, 1960, pp. 50–50.

[26] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, Jan 1998.

[27] G. Martin and H. Chang, "System-on-chip design," in *ASIC, 2001. Proceedings. 4th International Conference on*, 2001, pp. 12–17.

[28] M. Ahmad, "Stmicroelectronics and socs," https://www.semiwiki.com/forum/content/4239-stmicroelectronics-socs.html. [Online]. Available: https://www.semiwiki.com/forum/content/4239-stmicroelectronics-socs.html

[29] A. Higashi, K. Tamaki, and T. Sasaki, "Verification methodology for a complex system-on-a-chip," *Fujitsu Scientific and Technical Journal*, vol. 36, no. 1, pp. 24–30, 2000, cited By 0. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-0034206185&partnerID=40&md5=f1c0e76d6dc69a399d7ea4f2351626a0

[30] R. Lafore, *Object Oriented Programming in C++*, 4th ed. Indianapolis, IN, USA: Sams, 2001.

[31] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, 1st ed. Springer, Mar. 2000. [Online]. Available: http://www.worldcat.org/isbn/0792378229

[32] T. Grotker, *System Design with SystemC*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[33] Systemc open initiative. OSCI. [Online]. Available: www.accellera.com

[34] L. Cai and D. Gajski, "Transaction level modeling: An overview," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and*

System Synthesis, ser. CODES+ISSS '03. New York, NY, USA: ACM, 2003, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/944645.944651

[35] *SystemC User's Guide.*

[36] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[37] S. Chen, "More than moore foundry: Challenges for process technology, process characterization and design enablement to address complex cyber physical systems in harsh environments," in *VLSI Technology, Systems and Application (VLSI-TSA), Proceedings of Technical Program - 2014 International Symposium on*, April 2014, pp. 1–1.

[38] A. Heinig, M. Dietrich, A. Herkersdorf, F. Miller, T. Wild, K. Hahn, A. Grunewald, R. Bruck, S. Krohnert, and J. Reisinger, "System integration #x2014; the bridge between more than moore and more moore," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–9.

[39] J. Bergeron, *Writing testbenches : functional verification of HDL models.* Boston: Kluwer Academic, 2000, index. [Online]. Available: http://opac.inria.fr/record=b1099701

[40] S. Palnitkar, *Design Verification with e.* Prentice Hall Professional Technical Reference, 2003.

[41] R. I. M. group, "The soc verification gap," http://www.realintent.com/real-talk/265/the-soc-verification-gap, 2010.

[42] DVCon, "Howt to close the verification gap," http://community.cadence.com/cadence_blogs_8/b/fullerview/archive/2014/03/19/closing-the-electronics-system_2d00_verification-gap, 2014.

[43] H. D. Foster, "Why the design productivity gap never happened," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 581–584. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561943

[44] A. Molina and O. Cadenas, "Functional verification: approaches and challenges," *Latin American applied research*, vol. 37, pp. 65 – 69, 01 2007. [Online]. Available: http://www.scielo.org.ar/scielo.php?script=sci_arttext&pid=S0327-07932007000100013&nrm=iso

[45] D. Große and R. Drechsler, *Quality-Driven SystemC Design.* Springer, 2010. [Online]. Available: http://www.springerlink.com/content/978-90-481-3631-5

[46] S. Merz, "Model checking: A tutorial overview," in *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, ser. MOVEP '00. London, UK, UK: Springer-Verlag, 2001, pp. 3–38. [Online]. Available: http://dl.acm.org/citation.cfm?id=646410.692520

[47] B. Wile, J. Goss, and W. Roesner, *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[48] C. van Eijk and J. A. G. Jess, "Towards the functional verification of large sequential circuits."

[49] M. Salem and H. Foster, "Planning for functional verification closure," in *Design and Test Workshop, 2007. IDT 2007. 2nd International*, Dec 2007, pp. xv–xvi.

[50] H. Foster. (2010, August) Redefining verification performance (part 2). Verification Horizon BLOG. Mentor Graphics. [Online]. Available: http://blogs.mentor.com/verificationhorizons/blog/2010/08/08/redefining-verification-performance-part-2/

[51] W. K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series).* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[52] A. Piziali, *Functional Verification Coverage Measurement and Analysis*, 1st ed. Springer Publishing Company, Incorporated, 2007.

[53] R. Roth and D. Ramanathan, "A high-level hardware design methodology using c++," in *4th High Level Design Validation and Test Workshop*, 1999, pp. 73–80.

[54] P. Bellows and B. Hutchings, "Jhdl-an hdl for reconfigurable systems," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, Apr 1998, pp. 175–184.

[55] P. Alexander, R. Kamath, and D. Barton, "System specification in rosetta," in *Engineering of Computer Based Systems, 2000. (ECBS 2000) Proceedings. Seventh IEEE International Conference and Workshopon the*, 2000, pp. 299–307.

[56] G. Agosta, F. Bruschi, and D. Sciuto, "Static analysis of transaction-level models," in *Proceedings of the 40th Annual Design Automation Conference*, ser. DAC '03. New York, NY, USA: ACM, 2003, pp. 448–453. [Online]. Available: http://doi.acm.org/10.1145/775832.775950

[57] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, 2003, pp. 19–24.

[58] H. M. Le, D. Große, and R. Drechsler, "Scalable fault localization for systemc tlm designs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 35–38. [Online]. Available: http://dl.acm.org/citation.cfm?id=2485288.2485299

[59] B. Bailey, F. Balarin, and M. McNamara, *TLM-driven Design and Verification Methodology*. Cadence Design Systems, 2010. [Online]. Available: http://books.google.com.ar/books?id=zufcbwAACAAJ

[60] "Ieee standard for standard systemc language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan 2012.

[61] "Modelsim, advanced verification and debugging," 11 2004.

[62] "Vcs, functional verificaiton solution," http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx, 2015.

[63] A. Bernstein, M. Burton, and F. Ghenassia, "How to bridge the abstraction gap in system level modeling and design," in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 910–914. [Online]. Available: http://dx.doi.org/10.1109/ICCAD.2004.1382705

[64] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, and E. Carara, "From vhdl register transfer level to systemc transaction level modeling: A comparative case study," in *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design*, ser. SBCCI '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 355–. [Online]. Available: http://dl.acm.org/citation.cfm?id=942808.943929

[65] D. C. Black and J. Donovan, *SystemC: From the Ground Up*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[66] S. Swan and C. D. Systems, "An introduction to system level modeling in systemc 2.0," 2001.

[67] "The systemc verification standard, version 2.0," Accellera. [Online]. Available: http://www.accellera.org/downloads/drafts_review

[68] M. Moy, "Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach," in *The Design, Automation, and Test in Europe (DATE)*, Grenoble, France, Mar 2013. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00761047

[69] W. Müller, W. Rosenstiel, J. Ruf, G. Martin, and T. Grötker, *SystemC : methodologies and applications*. Boston, Dordrecht, London: Kluwer Academic Publishers, 2003. [Online]. Available: http://opac.inria.fr/record=b1105924

[70] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip Verification: Methodology and Techniques*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[71] V. Sinha, F. Doucet, C. Siska, R. Gupta, S. Liao, and A. Ghosh, "Yaml: a tool for hardware design visualization and capture," in *System Synthesis, 2000. Proceedings. The 13th International Symposium on*, 2000, pp. 9–14.

[72] S. Kumar, J. Aylor, B. Johnson, and W. Wulf, "Object-oriented techniques in hardware design," *Computer*, vol. 27, no. 6, pp. 64–70, June 1994.

[73] S. Vernalde, P. Schaumont, and I. Bolsens, "An object oriented programming approach for hardware design," in *VLSI '99. Proceedings. IEEE Computer Society Workshop On*, 1999, pp. 68–73.

[74] Siska, "Icsp technical report," CECS, UC Irvine, Tech. Rep., June 1999.

[75] F. Doucet, V. Sinha, C. Siska, and R. Gupta, "Microelectronic system-on-chip modeling using objects and their relationships," in *in: Online Symposium for Electrical Engineers (OSEE*, 2000.

[76] C. B. Jones, *Systematic Software Development Using VDM (2Nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[77] J. Dekeyser, P. Boulet, P. Marquet, and S. Meftali, "Model driven engineering for soc co-design," in *IEEE-NEWCAS Conference, 2005. The 3rd International*, June 2005, pp. 21–25.

[78] D. Gajski and R. Kuhn, "Guest editors' introduction: New vlsi tools," *Computer*, vol. 16, no. 12, pp. 11–14, Dec 1983.

[79] C. Weiler, U. Kebschull, and W. Rosensteil, "C++ base classes for specification, simulation and partitioning of a hardware/software system," in *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal*, Aug 1995, pp. 777–784.

[80] A. Habibi and S. Tahar, "Design for verification of systemc transaction level models," in *In Proc. Design Automation and Test in Europe*, 2005, pp. 560–565.

[81] ——, "Design and verification of systemc transaction-level models," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 1, pp. 57–68, Jan 2006.

[82] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosentiel, "Object-oriented modeling and synthesis of systemc specifications," in *Design Automation Conference, 2004. Proceedings of the ASP-DAC 2004. Asia and South Pacific*, Jan 2004, pp. 238–243.

[83] F. Ferrandi, F. Fummi, L. Gerli, and D. Sciuto, "Symbolic functional vector generation for vhdl specifications," in *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, March 1999, pp. 442–446.

[84] A. Fin, F. Fummi, and G. Pravadelli, "Amleto: a multi-language environment for functional test generation," in *Test Conference, 2001. Proceedings. International*, 2001, pp. 821–829.

[85] M. Cheema, O. Hammami, L. Lacassagne, and A. Merigot, "Hardware /software codesign of image processing applications using transaction level modeling," in *Computer Architecture for Machine Perception and Sensing, 2006. CAMP 2006. International Workshop on*, Aug 2007, pp. 46–52.

[86] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, and E. Carara, "From vhdl register transfer level to systemc transaction level modeling: a comparative case study," in *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on*, Sept 2003, pp. 355–360.

[87] STACR, *Transaction-Level Modeling (TLM) Guide.* -: STARC, 2008.

[88] P. Panda, "Systemc - a modeling platform supporting multiple design abstractions," in *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, 2001, pp. 75–80.

[89] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, "A systemc/tlm semantics in promela and its possible applications," in *In 14th Workshop on Model Checking Software SPIN*, 2007, pp. 204–222.

[90] K. Marquet and M. Moy, "1 efficient encoding of systemc/tlm in promela."

[91] F. Maraninchi, M. Moy, J. Cornet, C. Helmstetter, and C. Traulsen, "Systemc/tlm semantics for heterogeneous system-on-chip validation," 2008.

[92] G. J. Holzmann, *Design and Validation of Computer Protocols*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.

[93] SpinrootS, "Promela language reference. part of spin's online documentation," http://spinroot.com/spin/Man/promela.html, 2004.

[94] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975. [Online]. Available: http://doi.acm.org/10.1145/360933.360975

[95] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359585

[96] B. W. Kernighan, *The C Programming Language*, 2nd ed., D. M. Ritchie, Ed. Prentice Hall Professional Technical Reference, 1988.

[97] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[98] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[99] D. Bitton, D. J. DeWitt, D. K. Hsaio, and J. Menon, "A taxonomy of parallel sorting," *ACM Comput. Surv.*, vol. 16, no. 3, pp. 287–318, Sep. 1984. [Online]. Available: http://doi.acm.org/10.1145/2514.2516

[100] C. Thompson, "The vlsi complexity of sorting," *Computers, IEEE Transactions on*, vol. C-32, no. 12, pp. 1171–1184, Dec 1983.

[101] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ser. AFIPS '68 (Spring). New York, NY, USA: ACM, 1968, pp. 307–314. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468121

[102] C. D. Thompson, "Vlsi complexity of sorting." *IEEE Transactions on Computers*, vol. C-32, no. 12, pp. 1171–1184, 1983, cited By 49. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-0020886345&partnerID=40&md5=089f0c5d547846ad0d0b149482f32606

[103] D. C. Black and J. Donovan, *SystemC: From the Ground Up.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005, ch. Custom Channels and Data.

[104] Xilinx, "Coregenerator guide," http://www2.informatik.hu-berlin.de/~fwinkler/psvfpga/synthese/ISE-Dokumentaion/docs/cgn/cgn.pdf, 2003.

[105] "IEEE Standard for EthernetAmendment 1: Physical Layer Specifications and Management Parameters for Extended Ethernet Passive Optical Networks," *IEEE Std 802.3bk-2013 (Amendment to IEEE Std 802.3-2012)*, pp. 1–103, Aug 2013.

[106] Opencores. OpenCores. [Online]. Available: www.opencore.org

[107] *AMBA 3 AHB-Lite Protocol.* ARM, June 2006.

[108] Provartec, *PR201 Configurable Dual-Core High Performance AHB DMA Reference Guide*, Provartec.

[109] *AMBA APB Protocol.* ARM, April 2010.

[110] Synopsys, *VCS/VCSi User's Guide*, Synopsys.

[111] C. N. Ip and S. Swan, "A tutorial introductionon the new systemc verification standard," Cadence Design Systems, Inc, Tech. Rep., January 2003. [Online]. Available: http://people.cse.iitd.ac.in/~panda/SYSTEMC/Tutorials/date03a2.pdf

[112] "Verification using systemc part vi." http://www.asic-world.com/systemc/verification6.html, accessed: 2010-09-30.

[113] "Ieee standard for design and verification of low-power integrated circuits," *IEEE Std 1801-2013 (Revision of IEEE Std 1801-2009)*, pp. 1–348, May 2013.

[114] O. Mbarek, A. Pegatoquet, M. Auguin, and H. Fathallah, "Power-aware wrappers for transaction-level virtual prototypes: A black box based approach," in *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on*, Jan 2013, pp. 239–244.

[115] F. Mischkalla and W. Mueller, "Architectural low-power design using transaction-based system modeling and simulation," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, July 2014, pp. 258–265.

[116] T. Bouhadiba, M. Moy, and F. Maraninchi, "System-level modeling of energy in tlm for early validation of power and thermal management," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 1609–1614. [Online]. Available: http://dl.acm.org/citation.cfm?id=2485288.2485671