INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

METODOLOGÍA DE MONITOREO PARA VALIDACION DE CIRCUITOS VLSI

J Ismael Rangel Martínez

México, DF
Noviembre 2009

Tesis entregada al CIC en cumplimiento
parcial de los requerimientos para obtener
el grado de Maestro en Ciencias

DEDICATORIA

Para mi familia, base del empeño y perseverancia en mis estudios.

RECONOCIMIENTOS

Agradezco sinceramente el apoyo brindado por mis directores de Tesis, Marco Ramírez y Luis Villa, de quienes de no haber tenido el impulso, ánimo y supervisión del desarrollo de mi proyecto de Tesis, no hubiera sido posible concretar la meta de ser aceptado para hacer un Internship en Intel. En especial quiero reconocer el papel del Dr. Villa, quien siempre estuvo al tanto de mis avances y de las posibilidades de proyección para mí, ya sea estudiando un Doctorado o yéndome a Intel… muchas gracias Dr. Luis.

Gracias también a los profesores del CIC con quienes cursé las materias del posgrado, sus enseñanzas me han quedado bien grabadas y las estoy probando indiscutiblemente útiles al resolver los problemas que surgen en mis asignaciones ahora que me toca trabajar para una empresa líder en el desarrollo de microprocesadores.

Gracias al Instituto de Ciencia y Tecnología del Distrito Federal (ICyT DF), ya que con su apoyo, a través del proyecto SIP/DF/2007/143, obtuvimos la infraestructura requerida para el desarrollo de esta tesis.

Por último y no menos, quiero agradecer el apoyo de mi madre y hermana, quienes fueron mi principal apoyo moral y fuente de alegrías durante mis estudios. Gracias por confiar en mí y ayudarme a salir adelante.

RESUMEN.

La creciente cantidad de transistores integrables en un Chip ha dado lugar a familias de Circuitos inconcebibles hace tan solo 10 años: **ULSI** (Ultra-Large Scale Integration, con más de 1 millón de transistores), **3D-CI** (Three Dimensional Integrated Circuit, con dos o mas capas de componentes electrónicos activos integrados horizontal y verticalmente), Wafer-Scale Integration (**WSI**) etc., a los cuales se añade la dificultad de diseñar arquitecturas correctas que, a pesar de la progresiva densidad de transistores con la que están hechos, sean completamente funcionales y se desempeñen acorde a lo planeado.

La Validación Funcional o Verificación de los circuitos se está convirtiendo en el mayor cuello de botella en el proceso de producción.

Por otro lado, el desarrollo de nuevas tecnologías y herramientas de síntesis han provocado un aumento desequilibrado entre la habilidad de fabricar y la de diseñar, pero hay un aun mayor y más marcado desajuste con la habilidad actual para verificar.

Mientras los sistemas siguen creciendo en magnitud y complejidad, la demanda de memoria, ciclos de computo y consecuentemente el tiempo requerido para simular y verificar la correcta funcionalidad de dichos sistemas ha superado la capacidad de los equipos y mecanismos usados actualmente.

Los diseñadores de hoy se las están viendo difíciles a la hora de verificar sistemas que ocupan 10 Millones de compuertas cuando el mapa de diseño para los semiconductores muestra que capacidades de un trillón de compuertas serán factibles.

La necesidad de adoptar nuevas estrategias de verificación que aprovechen las ventajas que el computo distribuido brinda es, por tanto, urgente. La inyección de nuevas técnicas de simulación a esta problemática tendrá un impacto especial en el diseño de circuitos digitales, dando una enorme ventaja competitiva a cualquier empresa manufacturera de circuitos integrados.

Una de las estrategias que permite reducir los tiempos de simulación y verificación en la etapa de diseño de los circuitos integrados es justamente la simulación paralela, es decir utilizar múltiples procesadores para ejecutar entidades del sistema en estaciones separadas de trabajo y reducir así los tiempos de simulación.

Como ocurre en cualquier programa distribuido, una simulación fragmentada se compone de diferentes procesos dependientes entre sí e intercomunicados por una capa de transporte de datos y/o eventos. Esta comunicación puede ocurrir por medio de memoria compartida en una estación multiprocesador o mediante paso de mensajes en un ambiente distribuido. Cada uno de estos procesos tiene como objetivo simular una porción del sistema entero siendo modelado.

La simulación paralela de circuitos se basa principalmente en identificar algoritmos para subdividir un circuito en múltiples subcircuitos. Cada subdivisión puede ser comparable a una compuerta lógica, un módulo, un bus de comunicaciones, o cualquier otro objeto que pueda integrar varias de las anteriores. A cada una de estas subdivisiones se le denomina **partición.**

Muchos de los algoritmos de particionamiento encontrados trabajan sobre una equivalencia entre el grupo de subcircuitos y un **DAG asociado** (*Direted Acyclic Graph*)**.** Sin embargo a mayor parte de estos algoritmos se enfocan en realizar particiones a nivel *netlist*, dado que su propósito no es la simulación a nivel transferencia de registros, sino lo que en el proceso de diseño se le denomina *floorplanning* y *placement*.

Estos algoritmos producen subdivisiones de circuitos muy finas las cuales llegan a ser inviables para el objetivo buscado con las estrategias de paralelización, principalmente en lo que respecta al costo de las comunicaciones que se generan entre esta gran cantidad de subdivisiones.

Por otro lado, pocos de estos algoritmos consideran estrategias para el balance de carga en la distribución de procesos.

Diferentes trabajos de investigación exponen ciertos criterios base sobre los cuales determinar la calidad de un algoritmo de particionamiento.

Estos criterios son: minimizar la **comunicación**, maximizar el **paralelismo** y balancear la **carga de procesamiento**. A veces, la naturaleza del grafo compromete el alcance de estos objetivos y se torna difícil encontrar un punto equilibrado de particionamiento sin perder concurrencia o sin incrementar el cutsize entre particiones.

El aporte de este proyecto de Tesis es un indicador de que la simulación paralela y distribuida es ciertamente una estrategia efectiva para obtener tiempos más cortos de verificación funcional de circuitos, eliminar las limitaciones de memoria impuestas por una simulación en un sistema monoprocesador y minimizar los costos inherentes a una ejecución de procesos de forma distribuida.

Se llega a la conclusión de que la aceleración de la simulación es posible si se sabe explotar la información sobre jerarquía, carga de comunicaciones y paralelismo propio del circuito a ser simulado. Parte de esta información fue obtenida mediante pre-simulación, una estrategia que es poco usada para propósitos de particionamiento.

TABLA DE CONTENIDO

INTRODUCCIÓN

La simulación y verificación es un elemento fundamental en el flujo diseño de sistemas orientados a eventos, como los circuitos VLSI, las redes de computadoras, control de tráfico y congestión, y una gran variedad de otros sistemas. El objetivo de la simulación particularmente en los circuitos digitales es detectar errores (debugging) y comprobar (validar) la correcta funcionalidad del diseño antes de dar paso al proceso de fabricación y manufactura.

Mientras estos sistemas siguen creciendo en magnitud y complejidad (cientos de millones de transistores en un chip, procesadores multicore, redes en internet, etc.) la demanda de memoria, ciclos de computo y consecuentemente el tiempo requerido para simular y verificar la correcta funcionalidad de dichos sistemas ha superado la capacidad de los equipos y mecanismos usados actualmente para tal efecto, en otras palabras, se ha convertido en el cuello de botella en virtud de la demanda por nuevos y mejores sistemas. Es por esto que el objetivo de esta investigación en simulación paralela es aprovechar al máximo las plataformas de computo paralelo y distribuido (por ejemplo *Clusters*, *Grids*) con el fin de acelerar las simulaciones de los circuitos VLSI y cubrir las demandas de memoria y cómputo.

El presente proyecto tiene como objetivo desarrollar una metodología para simulación distribuida basada en el particionamiento de un sistema de forma eficiente, delineando el desarrollo de algoritmos para técnicas de sincronización, eventos e intercambio de información, así como para el balanceo de carga entre procesos con el fin de acondicionar modelos de cualquier rango para ser repartidos en un ambiente donde se ejecutaran simultáneamente en un plazo menor al que se tenía con una simulación serializada en un entorno monoprocesador convencional. Dicha metodología está dirigida hacia circuitos descritos con lenguajes de diseño de hardware (HDLs) ampliamente usados como Verilog, VHDL, SystemVerilog y SystemC [1, 2], misma descripción que puede ser a nivel compuerta, RTL o comportamental.

MOTIVACIÓN

La creciente cantidad de transistores integrables en un chip ha dado lugar a la categoría de Circuitos ULSI y VLSI (Ultra / Very Large Scale Integration), en los cuales se añade la dificultad de diseñar arquitecturas correctas que a pesar de la progresiva densidad de transistores con la que están hechos (hablamos de billones de ellos en un solo cm^2), sean completamente funcionales y se desempeñen acorde a lo planeado.

La Ley de Moore, apoyada por las nuevas tecnologías de diseño y los sintetizadores cada vez más confiables, son claros indicadores de que la integración más concentrada de transistores en una oblea de silicio seguirá en ascenso. Dichos indicadores también hacen patente la necesidad de tener un Plan de Verificación que garantice probar - si no es que todos - la mayoría de escenarios posibles en los que el dispositivo bajo prueba (DUT) pueda estar procesando sus datos de entrada, garantizando de este modo la funcionalidad del mismo.

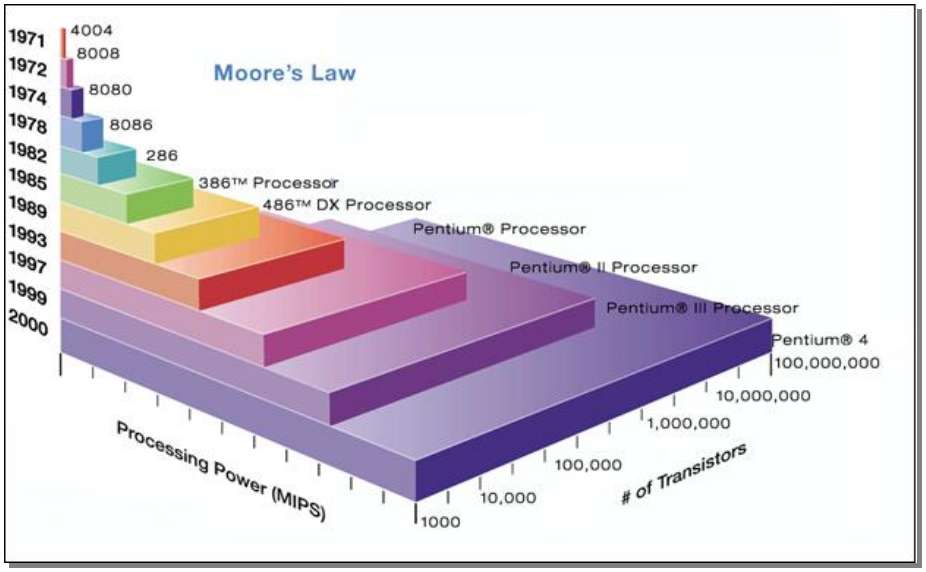


Figura 1. Gordon Moore estimó en 2003 que el numero de transistores integrados en ese año había alcanzado una cifra de 100, 000, 000, 000, 000,000 (10^{17}). Esto equivale al número de hormigas que hay en el mundo [un transistor por hormiga según la estimación del Profesor E. O. Wilson de la U de Harvard].

Los diseñadores de hoy se las están viendo difíciles a la hora de verificar sistemas que ocupan 10 Millones de compuertas cuando el mapa de diseño para los semiconductores muestra que capacidades de un trillón de compuertas serán factibles. De hecho, muchas compañías reportan un 70% de su tiempo y esfuerzo dedicado a la verificación funcional y todo indica que esta cifra continuará ascendiendo a un paso constante. Aunado a esto, la mayoría de los casos en los que se ha tenido que hacer un re-diseño de algún sistema o modificaciones en su estructura, son principalmente debido a fallas funcionales no detectadas a tiempo, y cuando estos rediseños son necesarios, la cifra puede alcanzar hasta un 80%.

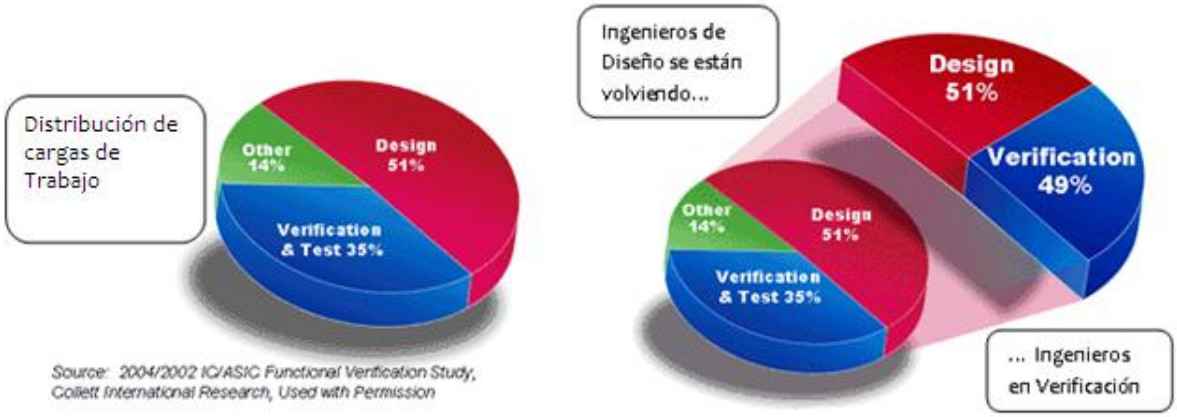


Figura 2. El time-to-market estriba cada vez más en el proceso de verificación del diseño.

Por otro lado, el desarrollo de nuevas tecnologías y herramientas de síntesis han provocado un aumento desequilibrado entre la habilidad de fabricar y la de diseñar, pero hay un aun mayor y más marcado desajuste con la habilidad actual para verificar.

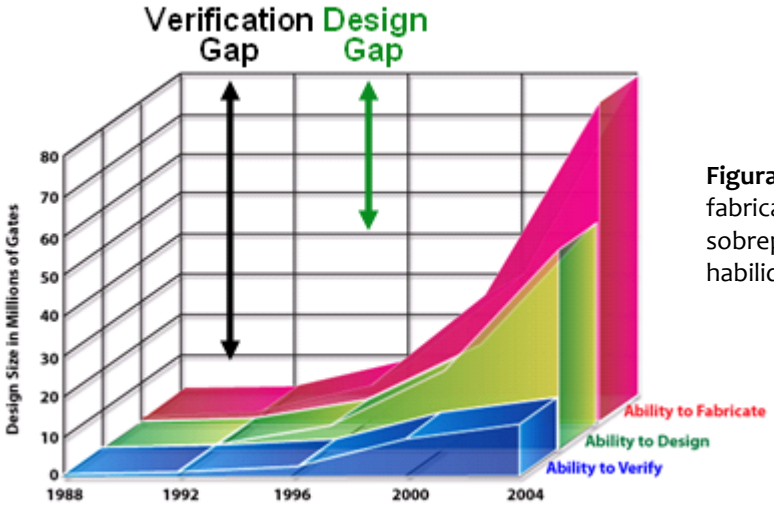


Figura 3. La habilidad actual para fabricar y diseñar circuitos digitales ha sobrepasado considerablemente la habilidad para verificarlos.

La necesidad de adoptar nuevas estrategias de verificación que aprovechen las ventajas que el computo en paralelo brinda es por tanto urgente. La inyección de nuevas técnicas de simulación a esta problemática tendrá un impacto especial en el diseño de circuitos digitales, dando una enorme ventaja competitiva a cualquier empresa manufacturera de circuitos integrados.

CARACTERÍSTICAS DEL PROYECTO

Como ocurre en cualquier programa distribuido, una simulación fragmentada se compone de diferentes procesos dependientes entre sí e intercomunicados por una capa de transporte de datos y/o eventos. Esta comunicación puede ocurrir por medio de memoria compartida en una estación multiprocesador o mediante paso de mensajes en un ambiente distribuido. Cada uno de estos procesos tiene como objetivo simular una porción del sistema entero siendo modelado.

Asimismo, cada proceso genera y recibe nuevos datos o eventos en el transcurso de su ejecución que necesitan ser tratados. Asociadas con cada proceso están colas de entrada y de salida que actúan como buffer de almacenamiento de eventos a ser enviados hacia otro proceso o recibidos desde otro proceso respectivamente. Por último, cada proceso refleja su estatus por medio del uso de variables de estado que nos permitirán saber si ha terminado, hacer retenciones o recuperar el estado de la simulación ante alguna falla.

Los algoritmos para subdividir los diseños de sistemas digitales en subcircuitos satisfacen, por un lado, **el balance en las cargas de trabajo** hacia los procesadores disponibles para realizar la simulación, y por el otro intentan **minimizar la comunicación** entre subcircuitos, asegurando así una **concurrencia máxima** a la hora de correr el testbench. El balance de las cargas de trabajo en un ambiente distribuido se ha estudiado desde diferentes estrategias, la mayoría de éstas se basan en las características propias de los subcircuitos (tamaño, nivel de diseño – compuerta, bloque, módulo-) y la relaciones que existen entre los mismos (número de interconexiones, puertos de entrada/salida, flujo de información) [3, 4, 5].

Una de las estrategias que permite reducir los tiempos de simulación y verificación en la etapa de diseño de los circuitos integrados es precisamente la simulación en paralelo, es decir utilizar múltiples procesadores para ejecutar entidades del sistema en cada estación y reducir así los tiempos de simulación [6, 7, 8]. La simulación paralela de circuitos se basa principalmente en identificar algoritmos para subdividir un circuito en múltiples subcircuitos. Cada subdivisión puede ser comparable a una compuerta lógica, un módulo, un bus de comunicaciones, o cualquier otro objeto que pueda integrar varias de las anteriores [9]. A cada una de estas subdivisiones se le denomina **partición**.

Muchos de los algoritmos encontrados asumen una equivalencia entre la agrupación de subcircuitos dentro del sistema y un **DAG asociado** (*Direted Acyclic Graph*), en donde los subcircuitos de hardware son los nodos y los arcos corresponden a los puertos e interfaces que conectan dichos subcircuitos. Trabajos recientes proponen algoritmos para particionar circuitos en múltiples niveles (*multilevel Partitioning*), los cuales exploran el uso de cómputo en clúster.

La mayor parte de los algoritmos propuestos para simular circuitos VLSI sobre sistemas distribuidos y paralelos se enfocan en realizar particiones a nivel *netlist* [11, 12]. Sin embargo, el foco principal de estas propuestas no es la simulación a nivel transferencia de registros, sino lo que en el proceso de diseño se le denomina *floorplanning* y *placement*. Estos algoritmos producen subdivisiones de circuitos muy finas las cuales llegan a ser inviables para el objetivo buscado con las estrategias de paralelización. Principalmente en lo que respecta al costo de las comunicaciones que se generan entre esta tremenda cantidad de subdivisiones. Por otro lado, pocos de estos algoritmos consideran estrategias para el balance de carga en la distribución de procesos.

Los lenguajes de descripción de hardware (HDL) nos permiten trabajar a nivel más alto que el nivel *netlist*. Estos lenguajes se han utilizado como referencia en el análisis y verificación de circuitos dada su flexibilidad para diseñar. Ofrecen una metodología de diseño jerárquica en las modalidades de *bottom-up* y *top-down.* Por ejemplo, el lenguaje Verilog [2], permite programar

en tres niveles de abstracción: nivel compuerta, nivel trasferencia de registros y nivel comportamental. Debido a que este último nivel de abstracción representa el nivel más alto de programación (permite estructuras *case, if, for*), actualmente el diseño de circuitos se realiza utilizando este modo de programación.

En este proyecto aprovecharemos las ventajas del **diseño jerárquico** de los lenguajes de descripción de hardware para llevar a cabo el particionamiento -en particular del lenguaje SystemC. En primera instancia, planeamos utilizar la granularidad por módulo en el contexto de SystemC - que encaja con el 'module' de Verilog y el 'entity' de VHDL - para generar el particionamiento y la estructura de grafos requerida en la subdivisión de los circuitos. A partir de este punto de referencia, podremos trabajar con granularidades más finas (sub-módulos) o granularidades más gruesas (conjuntos de módulos). Se desarrollará una herramienta de monitoreo y análisis para caracterizar el comportamiento de los circuitos (transiciones internas en los circuitos) y así tener una métrica acerca del uso de los bloques que componen al sistema.

Respecto al **algoritmo de sincronización** a usar para intercomunicar al conjunto de particiones resultante en el ambiente distribuido, se investigaron varios esquemas de comunicación [10]. Nuestro plan preliminar considera aplicar un algoritmo conservativo, el cual se caracteriza por su comportamiento bloqueante. En una simulación distribuida conservativa, si la cola de entrada para algún proceso está vacía, dicho proceso se bloquea y espera cierto tiempo para recibir mensajes de los demás procesos asociados a él. La desventaja de este algoritmo es que dicha suspensión en la ejecución puede incrementar el tiempo de simulación y peor aun caer en un 'abrazo mortal' si se perdió un mensaje de 2 procesos que esperaban entre sí. Por esto, para usar este algoritmo de sincronización debemos encontrar métodos que eviten dichos deadlocks o que los detecten y rompan.

OBJETIVOS

OBJETIVO GENERAL

- Desarrollar una metodología basada en el monitoreo de señales a nivel interfaz entre módulos que permita particionar un circuito digital, de tal modo que la simulación pueda paralelizarse reduciendo el tiempo gastado en el proceso de verificación funcional y debugging.

OBJETIVOS ESPECÍFICOS

- Hacer labor de investigación relacionada con la problemática general implícita en distribuir y paralelizar Simulaciones de circuitos VLSI.
- Aprender, dominar y aprovechar las ventajas de uno de los más recientes estándares en lenguaje de descripción de hardware SystemC [IEEE 1666™-2005] para modelar circuitos integrados.
- Desarrollar una herramienta de monitoreo y generación de estadística para caracterizar el comportamiento de los circuitos descritos en SystemC.
- Diseñar una herramienta de visualización que esboce el grafo que describe al sistema.
- Desarrollar una herramienta para subdividir circuitos digitales, considerando el intercambio de información entre sus módulos, número de puertos de entrada/salida y balance de carga, con tal de que el proceso de simulación pueda ser distribuido en un ambiente distribuido con ejecución concurrente.

RESUMEN

El presente trabajo de tesis detalla el desarrollo e implementación de una metodología que tiene como objetivo distribuir el proceso de simulación aplicado a los modelos de hardware descritos en SystemC particularmente, pero apropiada para extenderse a cualquier otro lenguaje de descripción de hardware como SystemVerilog, Verilog o VHDL, con el propósito de acelerar el tiempo de debugging y agilizar el flujo de verificación usado en la actualidad.

Existen múltiples y bien conocidos lenguajes de descripción de hardware que abrevian el tiempo en que los nuevos sistemas digitales salen al mercado, siempre mejorando las versiones anteriores. Estos lenguajes también son usados por los Ingenieros de Verificación para asegurar la funcionalidad y correcto desempeño del sistema en desarrollo. Sin embargo, el esquema tradicional de verificación no le sigue el paso al continuo incremento en la escala de integración y complejidad de los sistemas digitales, analógicos y mixtos, convirtiéndose en un verdadero cuello de botella dentro del flujo de diseño.

Es por esto que consideramos que nuestra propuesta de un ambiente de simulación paralelo y distribuido en una red de estaciones de trabajo (clúster) tiene el potencial de aliviar el elevado poder de cómputo y la gran demanda de memoria que actualmente requiere la simulación de circuitos de muy alta escala de integración.

Palabras clave: Simulación Paralela, Distribuida. Clúster. Particionamiento. Comunicación, Sincronización de Procesos.

ABSTRACT

The work presented in this thesis describes the design and implementation of a framework intended to distribute the overall simulation process applied to a hardware model description done - particularly - in SystemC but appropriate for whichever hardware description language (HDL) is used, with a view to make less burdensome the traditional verification paradigm and minimize the time spent in debugging.

There are several well known HDLs used to speed up the time-to-market for the designs of today's digital systems, they are also used by Verification Engineers to make sure of their functionality and correctness. However, this traditional flow of verification cannot keep pace with the continuous integration density and complexity of digital, analog and mixed systems, becoming a serious bottleneck inside the design flow.

Therefore, the scheme of a parallelized and distributed simulation environment amongst a cluster built upon a set of workstations that in turn could have multiple processors has the potential to afford the intense demand of memory and computational capability required in current system verification approaches.

Keywords: Distributed/Parallel Simulation. Cluster, Workstation. Circuit Partitioning. Process Synchronization/Communication.

ESTADO DEL ARTE

¿Qué es Verificación funcional?

Este término aplicado al funcionamiento de los circuitos lógicos digitales, analógicos y mixtos, tiene que ver con demostrar que el comportamiento de un sistema coincide con el propósito del diseñador.

El intento es por satisfacer que el funcionamiento del dispositivo bajo prueba (DUT) sea de acorde a lo especificado.

La herramienta típica de evaluación de este argumento consiste en una corrida del modelo del sistema - típicamente descrito usando un lenguaje de descripción de hardware - en un simulador. Para someter al DUT a diferentes escenarios, se usa una cama de pruebas cuyo objetivo es exponer el comportamiento del DUT al mayor número de casos posibles con el fin de revelar fallas potenciales. El término usado para esta técnica es 'estresar' el DUT.

Podemos mencionar varios puntos a favor por los cuales esta metodología encierra una de las actividades clave a la hora de verificar el diseño, pues al emular el comportamiento del mismo lo hace un proceso ideal para evaluarlo. Las ventajas de la simulación son:

Es una forma natural para que el diseñador tenga una retroalimentación acerca de su diseño, y dado que él mismo corre la simulación, interactúa directamente con él usando el mismo lenguaje y abstracciones que usó para estructurarlo. No hay capa alguna de traslación que oscurezca el comportamiento del diseño.

El grado de esfuerzo requerido para depurar y verificar un diseño es directamente proporcional a la madurez del diseño. Esto quiere decir que, mientras más temprano en el desarrollo del diseño, bugs y errores en su comportamiento son encontrados con relativa facilidad. Mientras el diseño va madurando y haciéndose más complejo, toma más tiempo encontrar las incorrectas funcionalidades, haciéndolo más cansado y tedioso después.

La Simulación es aplicable a cualquier diseño, los únicos límites son el tiempo y recursos de cómputo que conlleven realizar simulaciones de comportamientos específicos.

Sin embargo, en el lado negativo de la simulación se presentan algunas desventajas, una de ellas es determinante:

No hay - usualmente, y para los sistemas tan complejos de la actualidad - manera de saber cuando hemos acabado. No hay una forma concluyente de hacer un test completo y satisfactorio, vía simulación, de todos los posibles estados y entradas de un sistema no trivial.

Puede darse el caso en que una simulación tome una cantidad exorbitante de recursos y tiempo de computo para reproducir el comportamiento de miles (quizá millones) de procesos en paralelo simulados tradicionalmente en un ambiente monoprocesador.

Esta segunda desventaja ha motivado la mayoría de la investigación y desarrollo actual de las estrategias alternas en simulación. Esto es porque las simulaciones están empezando a tomar magnitudes de tiempo mucho mayores que los tiempos gastados en la especificación y diseño del sistema.

Para comprender mejor los cambios que necesitamos ocurran en la metodología de verificación, es importante ir a la raíz del problema. Uno de esos problemas, el principal y obvio es la magnitud del diseño.

Como sabemos, la promesa que encierra la ley de Moore está vigente y lo seguirá estando por algunos años más. La relación entre el tamaño del diseño y la complejidad en su verificación es afectada por múltiples factores, por ejemplo:

Por un lado, si ya tenemos un diseño para el cual la complejidad de verificación es ponderada a un valor dado, digamos 5, entonces si la estructura del circuito es duplicada al formar dos entidades del mismo diseño conectadas en paralelo, la complejidad no ha aumentado al mismo grado, ya que el numero de ‘vectores’ es el mismo, lo único que ha cambiado es su longitud. El re-uso de las técnicas de verificación que se emplearon para evaluar el diseño base son viables y, por esto, verificar el nuevo supone una complejidad de verificación de menos del doble, es decir una complejidad aproximada de 7 y máxima - pero no probable - de 9 o 10.

En cambio, si el segundo bloque es conectado en serie tal que las salidas del primero sean las entradas del segundo, la complejidad se ha elevado al cuadrado, pues cada vector del primer circuito tendría un efecto en la entrada del sucesor, y así por extensión también lo tendrá en las salidas finales del sistema. En otras palabras el número de vectores en juego se ha elevado mientras su longitud se mantiene. En este caso podríamos hablar de una complejidad de verificación de 20 a 25 en la escala propuesta. Con este ejemplo observamos que un diseño secuencial conjuntando dos entidades puede resultar hasta en un aumento cuadrático de la complejidad para verificarlo. Estas son solo algunas de las variantes que se presentan al escalar diseños e interconectarlos entre sí. Sin embargo, hay técnicas y principios del cómputo en paralelo que nos ayudaran a hacer menos pesados estos desajustes en el proceso de verificación.

FLUJO ACTUAL DE DISEÑO DE CIRCUITOS VLSI

El siguiente diagrama muestra el flujo actual de diseño de circuitos integrados de aplicación específica (ASICs).

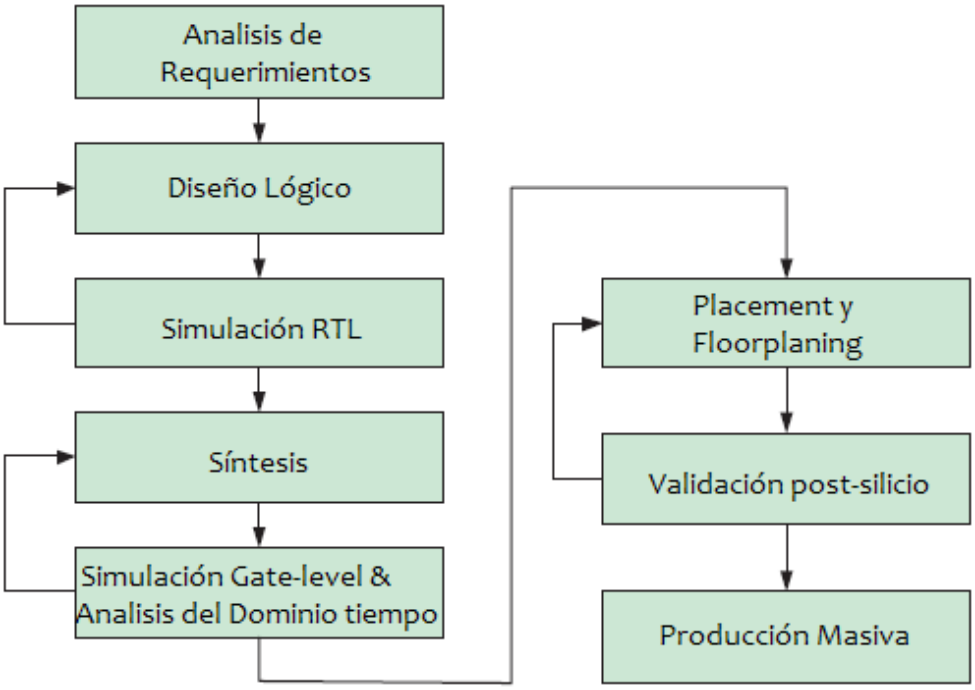


Figura 4. Flujo de diseño actual de los ASICs.

Análisis de Requerimientos.

El diseño del ASIC comienza entendiendo y especificando las funciones requeridas del sistema.

Diseño Lógico.

El grupo de diseño construye un modelo del ASIC usando un lenguaje de descripción de hardware como VHDL o Verilog. Este proceso equivale a escribir un programa en un lenguaje de alto nivel como C/C++. A esta etapa también se le llama diseño a nivel transferencia de registros (RTL). Este nivel de abstracción para modelar el circuito tiene que ver con programar la lógica necesaria para procesar los datos y actualizar el valor de los registros cada nuevo pulso de reloj.

Simulación RTL.

En esta fase entra en juego la verificación funcional para asegurar el correcto comportamiento del DUT. Como ya se mencionó, la forma más común de hacer esto es inyectar un conjunto de vectores a los puertos de entrada del circuito y comparar el resultado con los datos esperados, conocidos como 'golden data'. Si no hubo una coincidencia total con el golden data es porque hay algún error en el código que describe el sistema y se regresa a la etapa de diseño lógico para corregirlo.

Usualmente, el modelo de referencia que arrojará el golden data que a su vez servirá de base comparativa es escrito en otro lenguaje de programación como C o Fortran.

Síntesis.

Por medio de una herramienta que compila el diseño lógico se produce un netlist de bloques estándar, como compuertas And, Nor, Inversores, etc.

Simulación de Tiempos y Gate-level.

Esta etapa es la última fase de verificación pre-silicio que asegura que el netlist sigue comportándose de acorde a lo especificado y las señales son procesadas correctamente cumpliendo con los tiempos mínimos requeridos para que estas sean detectadas.

Placement & Floorplaning.

Una herramienta de placement distribuye estos bloques estándar de los que hablamos en regiones disponibles de la oblea de silicio. Esta etapa coloca los bloques de tal modo que las limitantes de área, espesor de los pozos de la tecnología CMOS utilizada, longitud de conexión entre bloques, etc., se cumplan.

La herramienta de ruteo crea las conexiones eléctricas entre los bloques después de haber determinado su ubicación, resultando en una máscara fotosensible que se imprimirá en la oblea de silicio.

Validación post-silicio y producción.

Una vez obtenido el chip físicamente, se corren pruebas finales para deshacerse de cualquier problema que puede surgir por fallos en la interconexión o tiempos de señales requeridos.

Figura 5. Procesador Intel Quad-Core

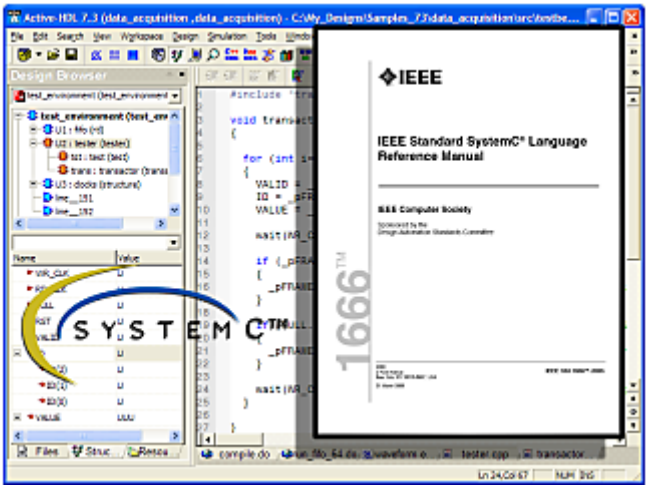
LENGUAJES DE DESCRIPCIÓN DE HARDWARE [HDLS]

En contraste con un lenguaje de programación de software, un HDL incluye elementos sintácticos para expresar tiempo, concurrencia y conectividad como atributos de cualquier clase base - o módulo en el caso de SystemC - . Los HDLs pertenecen a la clase de lenguajes de computadora para descripción formal de circuitos electrónicos. Estos pueden describir operatividad, organización y concurrencia en tiempo de ejecución así como verificar su funcionamiento por medio de la simulación [13].

El simulador puede actuar sobre el modelo de modo discreto usando un motor de ejecución basado en el concepto evaluación-actualización, en donde solo avanza al siguiente ciclo de tiempo cuando todos los procesos se han ejecutado y las señales han sido actualizadas. El otro modo tiene que ver con el modo continuo en donde se simula el comportamiento del sistema desde un punto de vista analógico, tomando en cuenta el tipo de tecnología usada en el circuito, capacitancias características, impedancias, etc. [14]. Los principales y más usados lenguajes de este tipo son Verilog, SystemVerilog, VHDL y SystemC. En este proyecto decidimos utilizar este último por las ventajas que nos da al generar ejecutables del diseño en cuestión fácilmente distribuibles en un ambiente tipo clúster, sin embargo la metodología desarrollada puede aplicarse a cualquiera de los lenguajes disponibles actualmente.

SYSTEMC

En esencia, SystemC es una biblioteca estandarizada por el IEEE [15], portable en C++ y usada para modelar sistemas con comportamiento concurrente. SystemC suministra mecanismos valiosos para modelar hardware mientras se usa un ambiente compatible con el desarrollo de software. Este lenguaje también provee estructuras orientadas a modelar hardware que no están disponibles en los lenguajes de programación orientados a objetos.



El motor de ejecución de SystemC permite la simulación de procesos concurrentes al igual que Verilog, VHDL o cualquier otro lenguaje de descripción de hardware. Esto es posible gracias al uso de un modelo cooperativo multitarea, el cual corre en un kernel que orquesta el intercambio de procesos y mantiene los resultados y evaluaciones de dichos procesos alineados en tiempo. La capacidad para manejar módulos con procesos concurrentes hace de este lenguaje una opción viable para intentar distribuir la simulación.

A su vez, es posible describir el comportamiento de las entidades desde varios niveles de abstracción y puede ser explotado para obtener flujos de diseño y verificación más rápidos.

Por otro lado, ya que C++ implementa la Orientación a objetos -paradigma que de hecho fue creado para técnicas de diseño de Hardware- la abstracción de datos, propiedades, comportamientos y atributos de un modulo (clase base) puede ser el principio para elaborar un diseño jerárquico.

```
#include "systemc.h"

SC_MODULE(dff) {
    sc_in<bool > clk;
    sc_in<bool > din;
    sc_out<bool > dout;

    void p1() {
        dout.write(din.read());
    }



    SC_CTOR(dff) {
        SC_METHOD(p1);
           sensitive << clk.pos();
    }

};
```

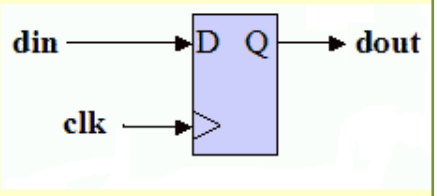


Figura 6. Descripción comportamental de un flip flop en SystemC.

Esta jerarquización de la que hablamos es una característica muy importante de SystemC, ya que esta información en conjunto con la medida de actividad en el sistema, sentará la base para desarrollar los diferentes algoritmos de particionamiento sin tener que recurrir a una granularidad de corte muy fina para alcanzar un balance de carga apropiado.

SIMULACIÓN CONTINUA Y DISCRETA

Simulación es la representación de operaciones y atributos de un sistema por medio un motor de ejecución que sabe interpretar los procedimientos que manejan las señales, datos, puertos, etc. propios del DUT. El conjunto de atributos del modelo simulado en cualquier instante se conoce como el estado de la simulación [16].

Hay dos categorías principales de simulación: discreta y continua. [17,18].

En una *simulación continua*, el estado de la simulación es evaluado como el resultado de un conjunto de ecuaciones diferenciales que describen el comportamiento del diseño en función de varios parámetros. Por ejemplo, para un circuito descrito a nivel transistor, la corriente, la resistencia y capacitancia característica puede ser simulada en tiempo continuo, ya que el comportamiento de estos componentes electrónicos del transistor está gobernado por una fórmula matemática.

$$I_D = \beta\left[(V_{GS} - V_T) - \frac{V_{DS}}{2}\right]V_{DS} \qquad \text{donde } \beta = \frac{\varepsilon_{ox} W \mu_e}{t_{ox} L}$$

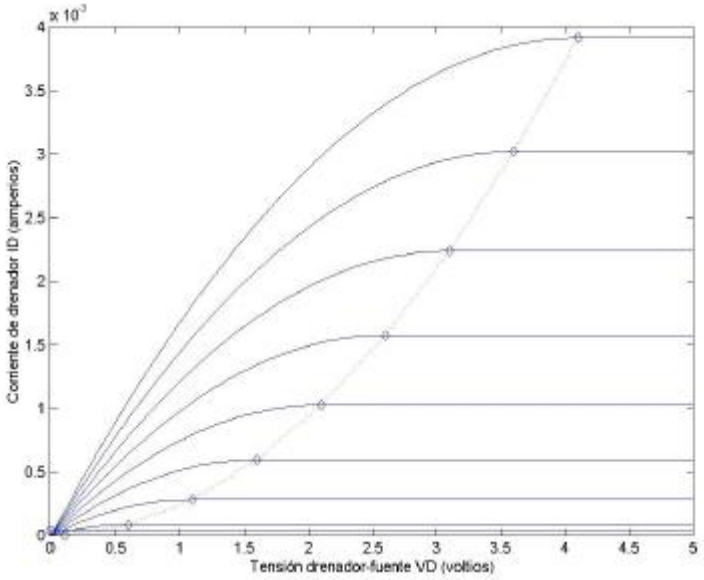


Figura 7. Función que expresa la Corriente en el drenador de un Transistor de tipo CMOS y respuesta ante varios Voltajes de entrada.

Desafortunadamente, las ecuaciones matemáticas empleadas para correr una simulación continua pueden ser exageradamente intensivas computacionalmente hablando, lo que resulta en una simulación demasiado lenta. Este tipo de simulaciones son útiles cuando los circuitos son descritos en un nivel de abstracción muy bajo, como el de nivel transistor o compuerta [19].

En cambio, la *simulación discreta* tiene la ventaja de ser usualmente más rápida aunque no tan exacta del modelo. Aun así, sigue dando una aproximación razonable del comportamiento de este.

Este tipo de simulación se subdivide a su vez en 2 categorías: orientada a eventos discretos y basada en tiempos discretos.

La primera describe un ambiente de simulación en donde solo el acontecimiento de nuevos eventos puede provocar un cambio en el estado de la simulación [20]. En el transcurso de este 'disparo' hacia el siguiente, el estado de la simulación no cambia. Este esquema permite un uso más eficiente de los recursos ya que solo se evalúan nuevos estados como resultado de un nuevo evento.

El kernel de este tipo de simulación usa dos estructuras para actualizar el estado de la simulación. La primera es el estado actual y la segunda es la cola de eventos, ordenada por marcas de tiempo que cada evento posee. El kernel lee eventos de la cola actualizando los procesos asociados a dicho evento y removiéndolos hasta que la cola queda vacía, lo cual implica que la simulación ha terminado.

Por otro lado, la actualización de los procesos que estaban 'escuchando' por un evento dado puede a su vez disparar un nuevo evento, que será insertado en la cola usando la marca de tiempo asociada como índice para ordenarlo. Una vez procesado el evento, dicho timestamp es usado para avanzar en el dominio del tiempo y actualizar el estado de la simulación.

Una descripción a bloques del algoritmo usado por la simulación orientada a eventos se muestra en la siguiente figura:

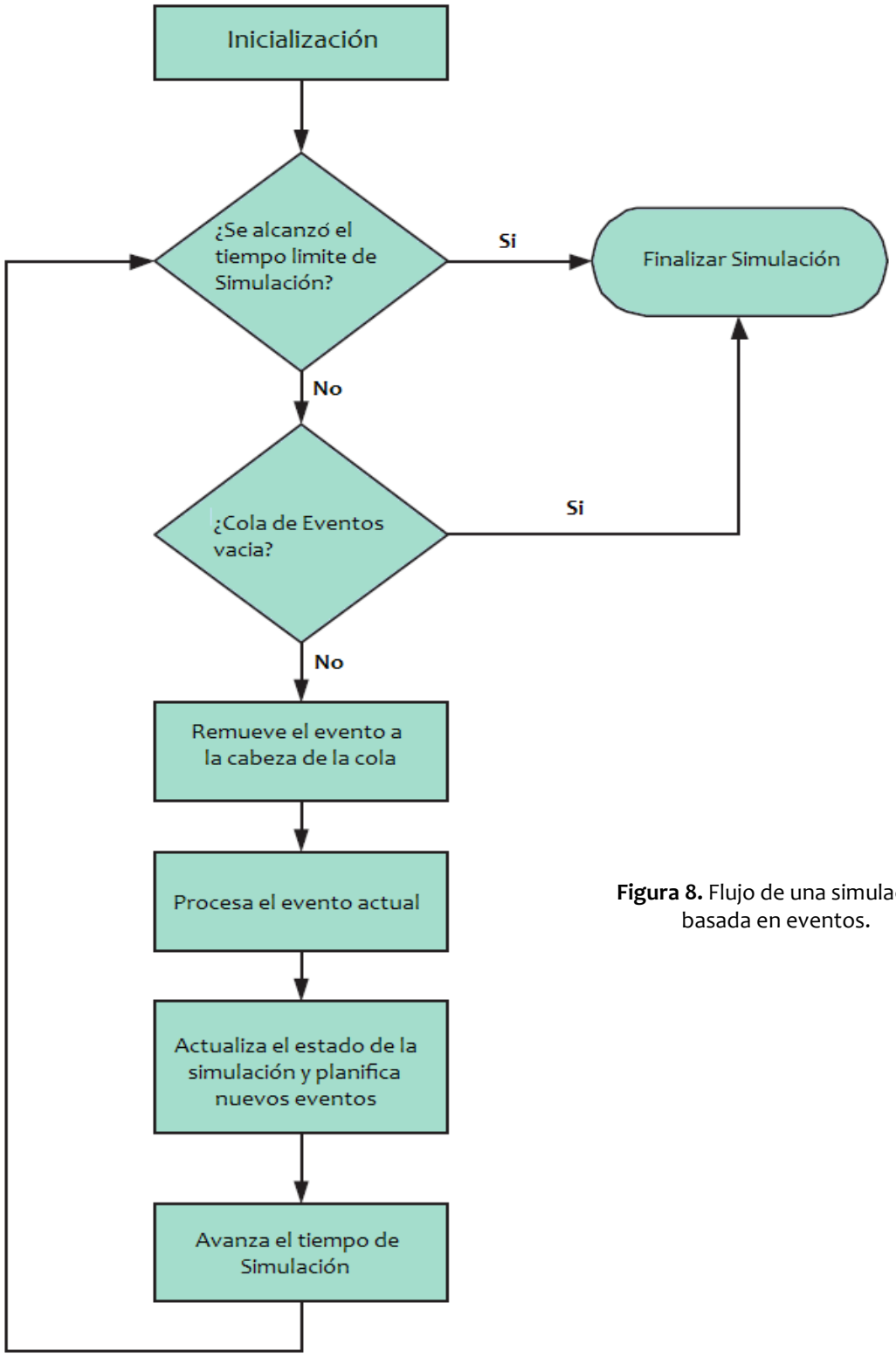


Figura 8. Flujo de una simulación basada en eventos.

Por último, la simulación basada en tiempos discretos usa incrementos de tiempo espaciados de modo uniforme, también llamados 'ticks'. En este entorno, cuantos más pequeños los incrementos de tiempo, la precisión en el estado de la simulación aumenta aunque reduce el tiempo de simulación como resultado de una mayor carga de trabajo, pues en cada periodo especificado, los atributos del modelo necesitan ser evaluados.

Por ejemplo, en la simulación de la trayectoria de un proyectil, la posición respecto a la altura y distancia que este ha alcanzado son calculados cada tick usando las ecuaciones que describen las fuerzas que actúan sobre dicho cuerpo.

En el caso de la simulación de un circuito digital, cada variable puede ser evaluada en función de expresiones lógicas o booleanas.

SIMULACIÓN DE CIRCUITOS DIGITALES

Los simuladores lógicos son una herramienta ampliamente usada para analizar el comportamiento de los circuitos digitales. Estos simuladores son usados en la etapa de verificación de un diseño de hardware para comprobar la correcta funcionalidad del dispositivo y aplicar un análisis de ajuste de tiempos al sistema.

La simulación discreta se aplica en este tipo de modelos y puede ser definida de tal forma que se obtenga un gráfico con trazos de las formas de onda. Los cambios en señales implicadas en el circuito quedan así registrados en los instantes en que ocurrió un evento o en el periodo de tiempo discreto predefinido para tomar las muestras. Este registro de los cambios en el estado de las señales se puede usar para un análisis de fallas, facilitando a los diseñadores de ASICs localizar el error en el código que describe al circuito, sin necesidad de utilizar instrumentos de control electrónicos como osciloscopios o analizadores lógicos, lo que implicaría llegar hasta la etapa de post-silicio o fabricación para poder detectarlos.

SIMULACIÓN DISCRETA BASADA EN EVENTOS DE SYSTEMC

El motor que usa SystemC para simular un diseño usa la misma característica que los simuladores para Verilog o VHDL: el paradigma de evaluación-actualización. Una primera fase de operación en el proceso de simulación es la elaboración. La ejecución de código previo a la sentencia sc_start() es conocida así. Esta fase se caracteriza por la inicialización de estructuras, instancias y conectividad entre módulos. Podemos pensar en esta fase como el linking que hace C antes de ejecutar un programa.

Después de esta etapa, en la línea donde se ejecuta la sentencia sc_start(), el kernel de simulación toma el control de la fase de ejecución, la cual planifica y orquesta la ejecución de procesos dando una noción de concurrencia.

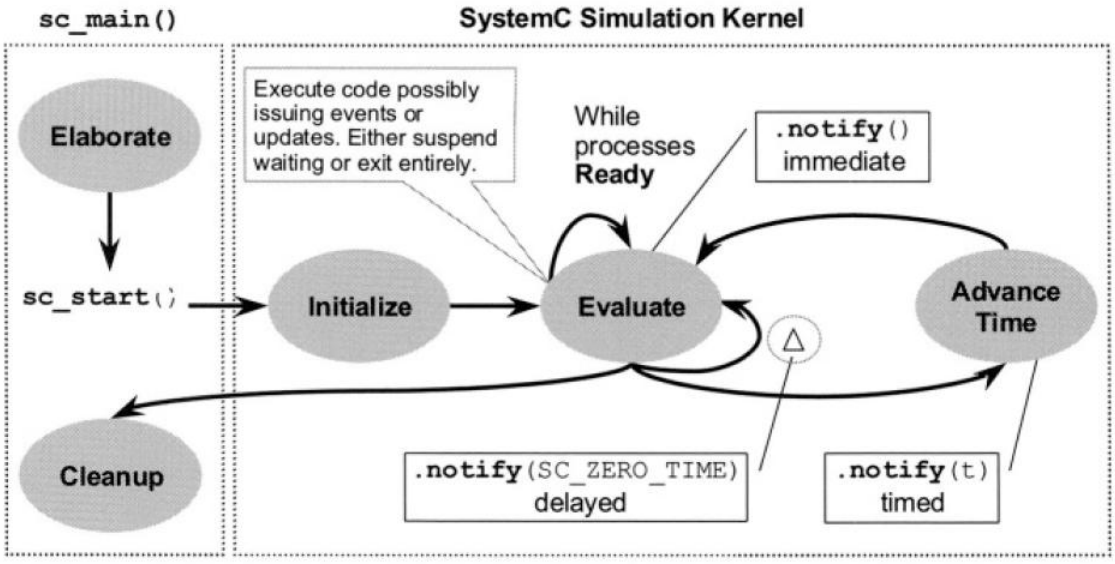


Figura 9. Diagrama de las fases principales del proceso de simulación en SystemC.

En esta etapa de trabajo, un proceso es puesto en marcha cuando algún evento a los que es sensible ocurre. Varios procesos pueden comenzar en el mismo instante de tiempo en la simulación, es por esto que todos los procesos y resultados son evaluados y actualizados antes

de avanzar al siguiente ciclo de simulación. Una evaluación seguida por la actualización de los atributos se le conoce como ciclo-delta. Las estructuras que el motor de simulación usa para planificar, ejecutar y poner en espera a los procesos son un conjunto de pilas, que junto con las pilas que guardan los eventos y las señales por ser actualizadas sientan la base para orquestar el avance de la simulación.

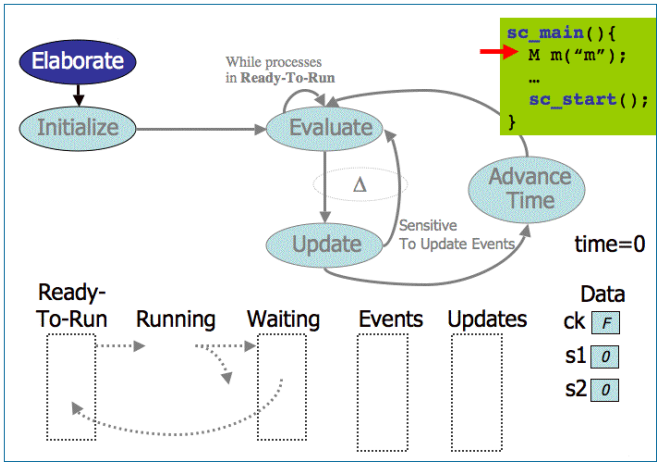


Figura 10. En la etapa de elaboración se hace la instanciación de módulos y port binding. Las Pilas están vacías puesto que ningún proceso se ha planificado.

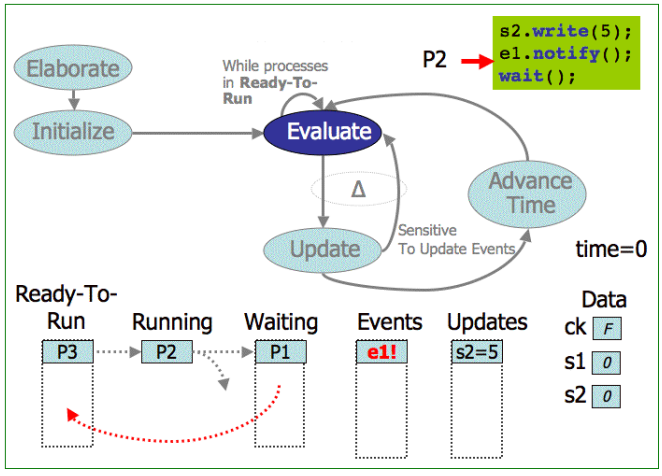


Figura 11. En la etapa de ejecución los procesos se van moviendo entre las pilas según van siendo disparados por los eventos a los cuales son sensibles.

Las pilas de eventos registra eventos que pondrán procesos en espera como listos y la pila de actualizaciones guarda variables que han sido modificadas por los procesos para escribirlas un vez que se vacíe la pila de procesos listos y no haya ninguno en ejecución.

En caso de no tener más procesos para ser evaluados en la pila de procesos listos y ningún evento, la simulación termina y una fase final de post-procesamiento o destructores de objetos hace una limpieza de memoria (cleanup).

MODELO DE SIMULACIÓN DISCRETA

Entre los componentes básicos que podemos listar en un circuito lógico están las compuertas - ANDs, ORs, XORs, INVERSORES, etc. – El circuito puede ser representado por un grafo en el cual los nodos representan esas compuertas y los arcos las conexiones o nets/wires.

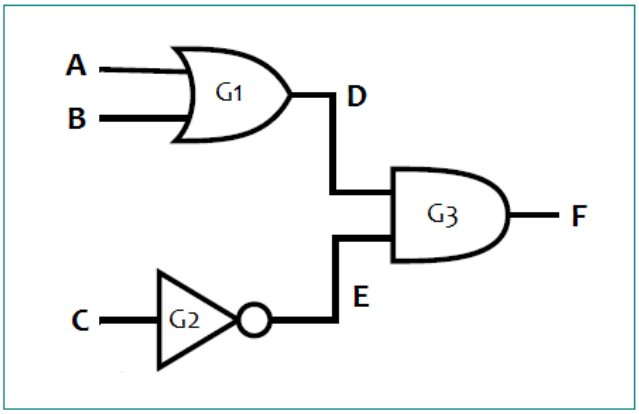
Los nodos estarán asociados a un proceso en software, mismo que en el ámbito de una simulación distribuida se les conoce como Procesos Lógicos (PLs).

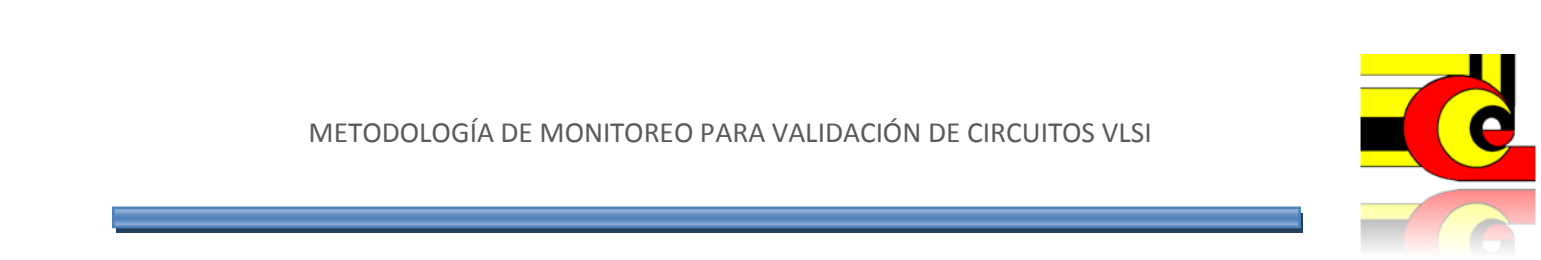
La cola de entrada hacia un proceso lógico corresponde a la lista de puertos conectados a la entrada de la compuerta lógica, mientras que la cola de salida se relaciona con la salida de la misma.

Hay un conjunto finito de tipo de datos que las colas pueden almacenar y este está dado por los tipos que maneja el lenguaje de descripción de hardware usado. Así, por ejemplo, en el caso de SystemC, podemos enviar valores booleanos para el caso de las compuertas lógicas; números enteros, flotantes, cadenas de caracteres, etc. para módulos más complejos. En realidad, todos los tipos soportados por el lenguaje C se pueden manejar a través de estas colas.

Una transición en cualquier señal asociada a algún Proceso Lógico es escrita con el valor nuevo al final de la cola de entrada. Cuando el PL recibe este nuevo dato, evalúa la salida y escribe el resultado en su FIFO (cola) de salida.

Como ejemplo podemos considerar un circuito lógico sencillo compuesto por tres compuertas. El circuito tiene tres señales de entrada: A, B y C, una salida: F, y 2 señales internas: D y E:





Podemos escoger cualquier combinación de valores iniciales para las señales de entrada, digamos A=1, B=0 y C=1. En este caso, la compuerta OR (G1) tiene asociadas a su FIFO de entrada las señales A y B, por lo tanto al escribirlas el proceso lógico que representa a la operación de la OR establecerá un valor de 1 en su cola de salida y limpiará su cola de entrada borrando los valores previamente escritos (1,0).

Del mismo modo, el inversor (G2) tendrá registrado en su cola de entrada el valor de C=1, al procesarlo lo borrará de su cola de entrada y pondrá un 0 en su cola de salida. Hasta este momento, las colas de entrada para G1 y G2 han quedado vacías y las señales intermedias D y E que representan las colas de salida de G1 y G2 tienen los nuevos valores.

Continuando con el flujo de los datos, la compuerta AND (G3) tiene a D y E asociados a su cola de entrada, así que los valores generados se escriben en esta. Como D=1 y E=0, el proceso lógico de esta compuerta escribirá un 0 en su cola de salida, limpiando en la FIFO de entrada los valores recién procesados.

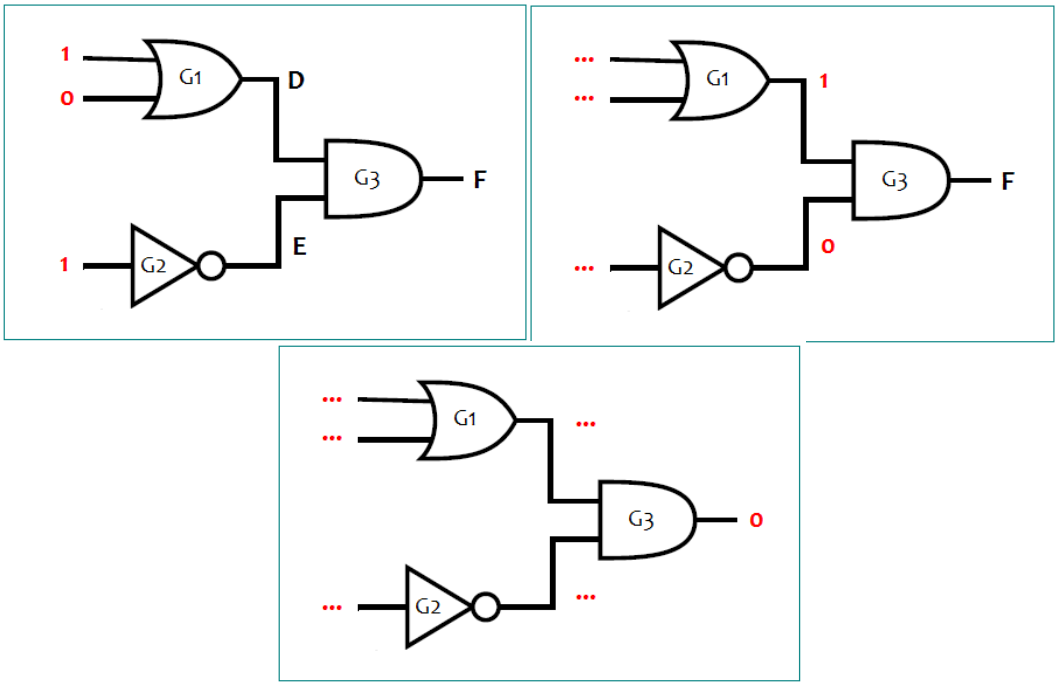


Figura 10. Ejemplo del flujo de datos en el modelo de simulación descrito.

La plataforma de computo para la cual está diseñada esta metodología es una integrada por maquinas multiprocesador conectadas en red. El paralelismo de la simulación resulta de la ejecución de un proceso en cada núcleo de la estación multiprocesador. En una misma estación se podrán mandar a ejecutar procesos que tienen un alto grado de dependencia en el que la comunicación se va a dar constantemente, ya que en esta los accesos son a memoria normalmente compartida por los procesadores reduciendo la latencia de comunicación.

Por otro lado, habrá subconjuntos de otros procesos que no impliquen un alto grado de comunicación, los cuales pueden ser repartidos entre las estaciones de trabajo para que cada una ejecute su conjunto asignado y en caso necesario, hacer uso de la red para mandar o recibir datos de otros procesos. En este caso la sobrecarga de comunicación es un factor clave que debemos minimizar al repartir los procesos ya que la latencia de comunicación en una red es mucho mayor dado que la memoria no es compartida, para una red de cómputo distribuido cada computadora tiene su propio espacio de memoria.

En general, la simulación paralela y distribuida estará formada por un conjunto de procesos lógicos o PLs (Proceso Lógico) en donde cada uno representa una parte de la funcionalidad del sistema modelado. Cada proceso lógico genera eventos que serán enviados a otros procesos y también estará recibiendo eventos de otros. Asimismo, además de manipular los eventos recibidos, un PL puede generar eventos locales que también necesitan ser procesados.

Un Proceso Lógico tiene asociadas dos colas en las que recibe y coloca eventos a ser enviados, la cola de entrada y la cola de salida respectivamente. Los eventos son almacenados en estas colas por orden de tiempo, así que pueden ser consideradas también como FIFOs en donde el primer evento en entrar es el primer evento que va a procesarse o a enviarse según sea el caso.

Un diagrama del sistema distribuido para un diseño con 4 procesos lógicos se muestra en la siguiente figura:

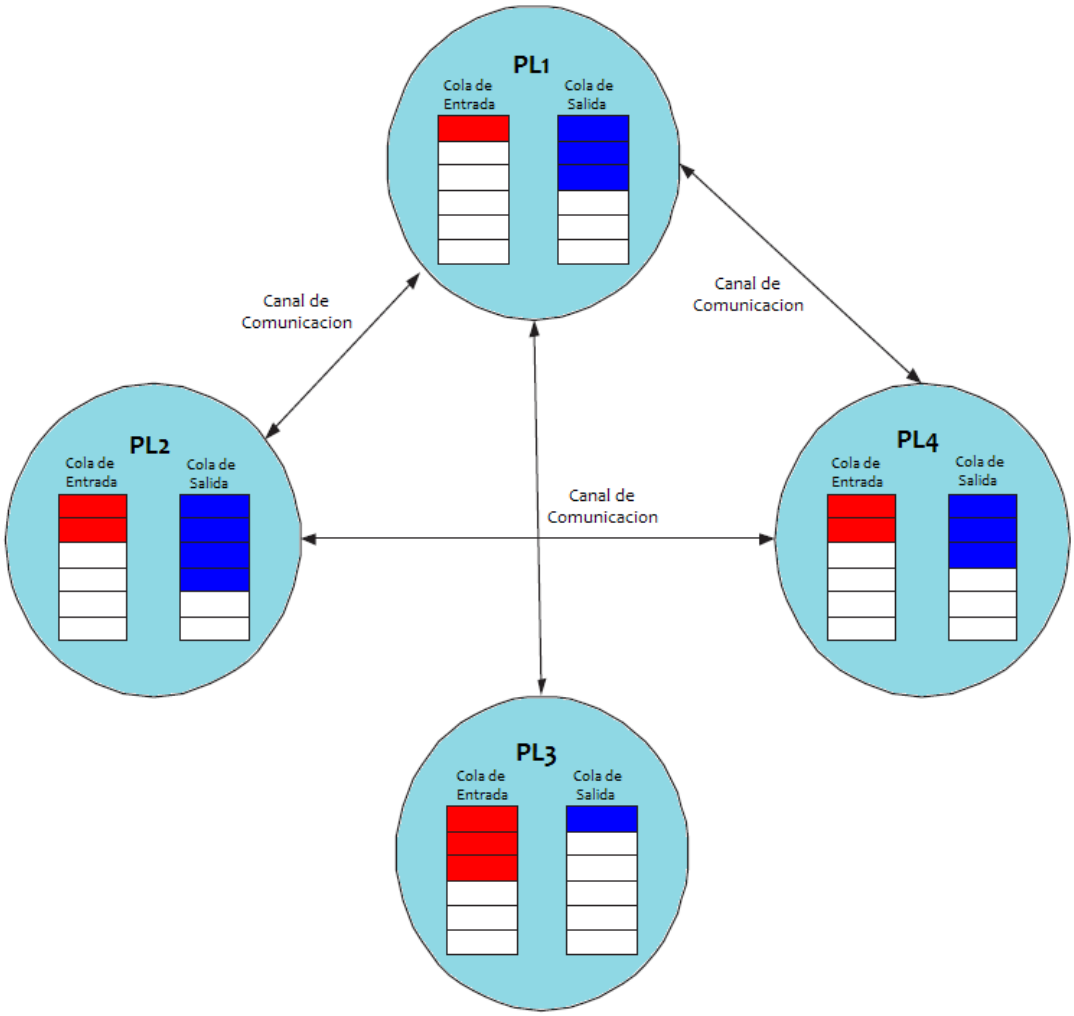


Figura 11. Ejemplo de un modelo con 4 PLs y sus canales de comunicación.

El proceso con mayor carga de comunicación es PL1, que envía y recibe datos a todos los demás, en cambio, PL3 solo se comunica con PL1. Una estructura de datos comúnmente usada para representar las conexiones entre los nodos (procesos) es la matriz de adyacencia, esta es una representación abstracta del grafo donde el elemento A_{ij} representa la ausencia (0) o presencia (1) de un canal de comunicación entre el proceso i y el j.

Para el ejemplo de los 4 procesos, la matriz de adyacencia se construye de la siguiente forma:

	PL1	PL2	PL3	PL4
PL1	0	1	1	1
PL2	1	0	0	1
PL3	1	0	0	0
PL4	1	1	0	0

De la primer fila podemos sacar la misma conclusión: el PL1 es el nodo con mayor número de canales de comunicación (la fila tiene el mayor numero de 1's comparada con las demás).

PL1	0	1	1	1

A su vez, PL3 es el que menos comunicación presenta (solo hay un canal de comunicación indicado en su fila).

PL3	1	0	0	0

Dado que los canales de comunicación tienen sentido, el grafo que describe al circuito debe ser un grafo dirigido y es por ello que necesitamos 2 elementos diferentes para saber si el nodo i envía datos al nodo j, ya que no necesariamente el PL j también envía 'paquetes' al PL i.

Si, por ejemplo, PL3 fuera un proceso de intensivo uso de ciclos de cómputo y sin mucha necesidad de transferencia de datos con PL1, la carga de procesamiento del sistema (overhead) se puede disminuir mandando a PL3 a ejecución en una sola estación de trabajo y los otros 3 que no son tan 'pesados' a una estación multicore distinta, donde se espera que la comunicación sea mayor.

Reducir la latencia de comunicación e incrementar la concurrencia es definitivamente el propósito principal del método usado para subdividir el circuito. Por medio de este pequeño modelo podemos hacer patente que el algoritmo de particionamiento debe tener una métrica sobre el número de transiciones entre procesos y del uso de CPU de cada proceso en la cual basarse para poder elegir la mejor composición de los subconjuntos de módulos a la hora de subdividir el diseño y distribuirlo en la plataforma de simulación.

ANÁLISIS DE LA METODOLOGÍA

Como punto de partida hacia el objetivo de este proyecto, y después de haber explorado el estado del arte del paralelismo en una Simulación Distribuida, comenzamos el diseño de la metodología para simulación distribuida basados en la siguiente Hipótesis:

Un sistema digital está compuesto por un conjunto de particiones intrínsecas, mismas que realizan una tarea o función particular para que, interrelacionados y en su totalidad procesen los datos a la entrada. Sin embargo, en esta jerarquización, los bloques no presentan la misma cantidad de transacciones; en la práctica, hay grupos de bloques que presentan una mayor actividad y comunicación entre ellos comparados con otros.

PRE-SIMULACIÓN

Simular el modelo por un lapso de tiempo breve antes de aplicar el particionamiento dentro de nuestra metodología es una fase muy significativa, ya que nos proveerá la métrica de actividad entre los módulos que componen al sistema, misma que a su vez será usada por el algoritmo de particionamiento para minimizar el costo de comunicaciones y balancear la carga de uso de CPU.

Correr una pre-simulación [29] es un método eficiente para evaluar la carga de transiciones dentro del diseño. Lo que esperamos con esta primera etapa de la metodología es recolectar una medida del flujo general de información dentro del circuito para saber que módulos presentan mayor actividad entre sí. La extracción temprana de esta métrica va a servir como indicativo del nivel de transiciones que se presentaran sobre los canales de comunicación y entre los bloques del diseño a lo largo del proceso general de simulación.

Cualquier algoritmo de particionamiento debería satisfacer apropiadamente el compromiso entre balanceo de carga de trabajo y mínima comunicación entre particiones.

Con una carga de trabajo variable entre cada partición, algunas estaciones de trabajo pudieran completar su procesamiento y permanecer desocupadas (en modo idle, desperdiciando ciclos de cómputo) esperando por nuevos eventos de otros procesos. Por otro lado, la simulación requiere comunicación periódica entre los módulos del sistema. Si dos módulos que se comunican bastante entre si son asignados a la misma partición, no se incurrirá en ningún costo por latencia debido al transporte de datos en la red.

Una de las principales limitaciones de los algoritmos de particionamiento actuales es la falta de conocimiento acerca de la carga real de comunicaciones y de trabajo computacional en el proceso de simulación previo a su ejecución.

La técnica aquí usada - llamada pre-simulación [30] - propone una ejecución de la simulación por un corto periodo de tiempo para medir la cantidad de transiciones en las señales que usa cada submódulo del circuito y así estimar la carga de computacional y de comunicaciones entre ellos.

El estudio que presenta [29] demuestra que la cantidad de transiciones es relativamente constante para cada señal o puerto del circuito a lo largo de la simulación. Las mediciones se hicieron tanto en las señales de entrada/salida del sistema así como en las internas, manteniendo un contador particular que es incrementado cada reevaluación de la señal correspondiente. Los resultados obtenidos por este trabajo demuestran generalmente útil el proceso de pre-simulación para pronosticar la carga de trabajo de los módulos que componen al sistema y la tasa de comunicaciones entre ellos.

Una alternativa que encontramos [37,38] es la estimación de carga de trabajo estática en el diseño, que implica hacer un parsing del código y formar así la estadística basados desde el número de líneas de código por modulo hasta el numero de procesos, numero de estructuras de control (if, else, for) número de compuertas/nets/registros usados, etc. Sin embargo, esta estimación -aunque más rápida- es mucho menos precisa ya que no captura el comportamiento dinámico y efectivo el sistema –si bien puede usarse para complementar la valoración dinámica.

ALGORITMOS DE PARTICIONAMIENTO

Tener una plataforma de simulación paralela y distribuida no garantiza una reducción del tiempo de simulación. Los costos de comunicación entre procesadores en un ambiente distribuido para un netlist que describe a un circuito con millones de compuertas lógicas es simplemente inmenso, y el tiempo de simulación resultante puede incluso superar el tiempo que en una sola estación de trabajo de forma serializada pudo haber gastado.

El problema de particionamiento de grafos entra en la rama NP-completos de la teoría de la complejidad. En otras palabras, el tiempo requerido para resolver problemas de este tipo usando cualquier algoritmo actual incrementa muy rápido conforme el tamaño del problema aumenta. Como resultado, el tiempo requerido para resolver incluso grafos con un moderado número de nodos (unos varios cientos de ellos) puede llegar a ser impráctico.

La mayoría de los algoritmos de particionamiento actuales [21, 22, 23, 24] están orientados al particionamiento de circuitos VLSI a nivel netlist. Estos algoritmos son usados para las etapas de floorplanning y placement de las que hablamos anteriormente en el flujo de diseño de los ASICs. Estos algoritmos producen un cantidad de cortes (cutsize) intolerable para un sistema de simulación distribuida debido a la gran cantidad de subdivisiones. El costo de comunicaciones que resulta de estos esquemas de particionamiento es muy alto y peor aún, los cortes resultantes no tienen estimación alguna del uso de CPU para un balance o equilibrio de cargas entre los procesadores.

En cualquier diseño actual de ASICs se tiene una metodología bien establecida basada en la jerarquización del diseño. Esta jerarquía se ve reflejada en módulos e interfaces que constituyen al circuito y que se mandan crear (instancian) a la hora de simularlo.

En un primer algoritmo de particionamiento, proponemos el uso de la información intrínseca de la jerarquía del diseño para generar las particiones. En este algoritmo el cutsize estará dado por la cantidad de módulos o entidades que componen al diseño, el elemento básico de particionamiento - o más precisamente: la granularidad de particionamiento - es por modulo.

Una forma común de resolver problemas NP-completos está basada en la heurística. Aplicar un algoritmo que trabaje "razonablemente bien" en la mayoría de los casos pero para el cual no hay prueba de que siempre será rápido y que siempre dará el mejor resultado posible.

Algunos trabajos [25,26] exponen ciertos criterios base en los cuales determinar la calidad del algoritmo de particionamiento. Estos criterios son: minimizar la comunicación, maximizar el paralelismo y balancear la carga de procesamiento. A veces, la naturaleza del grafo compromete el alcance de estos objetivos y se torna difícil encontrar un punto equilibrado de particionamiento sin perder concurrencia o sin incrementar el número de conexiones entre particiones.

COMUNICACIÓN

Considerando que la distribución de procesos ocurrirá sobre una red distribuida de estaciones de trabajo, la comunicación juega un papel determinante en el desempeño de la simulación. En casos en donde el diseño esté compuesto de una cantidad considerable de módulos, el costo de comunicación entre PLs debería ser reducida tanto como sea posible.

La correspondencia entre el número de cortes (cutsize) y el costo de comunicación es directa: entre menor número de cortes, menor número de mensajes son transferidos entre una estación y otra.

Esta proporcionalidad, sin embargo, no es una verdad universal. Puede haber canales de comunicación entre procesos con un tráfico denso de comunicación cuando algunos otros canales apenas intercambien información entre procesos.

La gran mayoría de algoritmos actuales de particionamiento son estáticos, lo que significa que los cortes son hechos sin haber pre-simulado el diseño. La propuesta en este trabajo de tesis es –como ya se abordó - correr una *pre-simulación* con el objetivo de recolectar información sobre el dinamismo del sistema.

PARALELISMO

Dentro de un diseño puede haber módulos de ejecución que puedan ser simulados sin la necesidad de sincronización entre ellos o incluso ninguna. Una métrica para evaluar la concurrencia que un algoritmo de particionamiento puede dar es descrita por [27], en donde se proponen dos estrategias para obtener un mayor grado de concurrencia al hacer los cortes:

El algoritmo de particionamiento en cono usa nodos con cero o el mínimo número de conexiones de otros nodos posible a la entrada de algún nodo y el mismo criterio para las conexiones de salida del o los nodos finales de la partición. El primer nodo que formara la partición es un nodo con un 'fan-in' de cero (o aproximado a cero) y se terminara el corte cuando se llegue a uno o varios nodos con un 'fan-out' de cero.

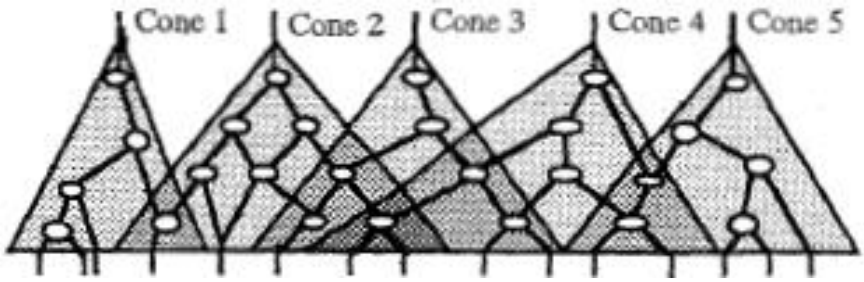


Figura 12. Particionamiento en cono para maximizar concurrencia.

La segunda propuesta consiste en hacer uso de los niveles del grafo para crear las particiones. En este esquema, se define el número de particiones como el ancho máximo del grafo, es decir, el número de nodos en el nivel más 'poblado' del grafo. Después de obtener este parámetro, se asignan todos los nodos del primer nivel a diferentes particiones cada uno, pasando así por todos los niveles donde en el nivel con mayor ancho, cada nodo será asignado a una partición.

BALANCE (EQUILIBRIO) DE CARGAS DE PROCESAMIENTO

En una plataforma no homogénea de estaciones de trabajo conectadas en red, el rendimiento de la simulación está limitado por su estación más lenta de trabajo, por lo que las cargas de cómputo no deberían de ser distribuidas arbitrariamente. Por el contrario, una repartición de procesos proporcional a la capacidad de cómputo de cada estación puede resultar en un avance continuo y fluido del proceso de simulación.

EL criterio para determinar el 'peso' de los módulos que componen a una partición puede estar basado en varias características: el número de eventos que el PL ejecuta, el número de líneas de código en el que está escrito, la cantidad de compuertas/registros que utiliza, etc. Una vez más, la mejor medida que podemos tener es corriendo una pre-simulación para determinar que bloques son los más activos.

TRABAJOS RELACIONADOS

En problema NP-completo de particionamiento de grafos es comúnmente abordado con técnicas heurísticas, [28] presenta panorama masivo del estado del arte de particionamiento de circuitos VLSI y hablaremos de algunos de ellos:

Particionamiento aleatorio es el algoritmo más simple y rápido de implementar. Este asigna compuertas lógicas a procesos aleatoriamente. El balance de carga consiste en que el elemento básico de particionamiento es una compuerta, por lo tanto aunque aleatoriamente, todas las particiones van a consistir del mismo número de compuertas y por ende la carga de CPU estará normalizada, sin embargo el costo en las comunicaciones puede ser muy alto pues no tiene noción alguna del flujo de la comunicación entre compuertas, solo las asigna arbitrariamente a cada partición.

El ***Particionamiento en cadena*** distribuye entradas primarias del circuito entre el número de particiones deseadas, ya que estos nodos son los que generaran nuevos eventos que serán las semillas para poner al sistema en funcionamiento. El algoritmo prosigue con una búsqueda en profundidad o hacia abajo en los niveles del grafo siguiendo una de las múltiples salidas que puede tener el nodo del cual surgió la partición. Hace lo mismo con el siguiente nodo encontrado hasta llegar a un nodo final de salida del sistema o hasta toparse con uno ya asignado. Cuando sobran nodos sin asignar, se escogen aleatoriamente y se van asignando a la partición con menor número de nodos (o compuertas lógicas, que es también la granularidad pensada para este algoritmo).

Particionamiento con replicación deja de lado la estrategia de asignar cada nodo solo a una partición y permite que el cutsize se reduzca asignando un nodo a 2 o más particiones si tal replicación es conveniente:

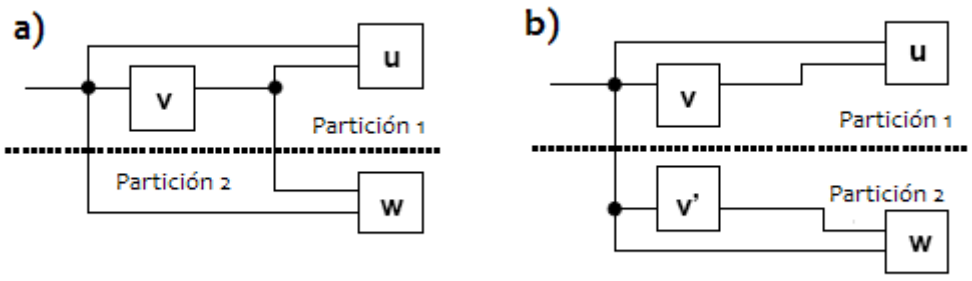
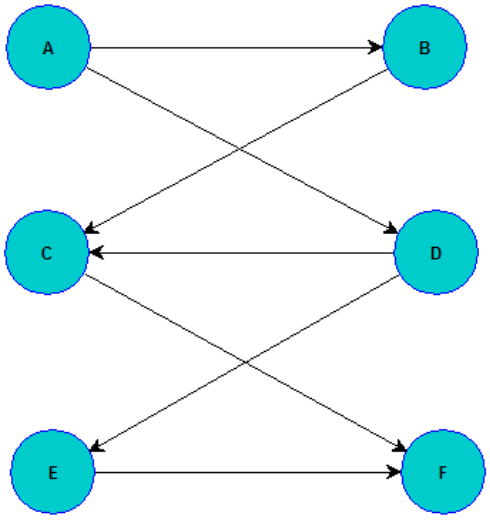


Figura 12. a) Muestra un circuito con un cutsize=2 (dos canales de comunicación fueron cortados) b) Por otro lado, la replicación del nodo v reduce el cutsize a 1.

Particionamiento con mínimo cutsize. En este algoritmo se escoge un nodo de la parte 'alta' del grafo que describe al sistema y se hace una agrupación de nodos de modo que el número de canales de comunicación interrumpidos (cutsize) sea el mínimo. El costo de comunicación para las particiones resultantes es de este modo el menor posible y la latencia (costo en tiempo que le toma a la red transferir información de una partición ejecutándose en una estación de trabajo a otra) de la red no afectara tan drásticamente el flujo de la simulación.

Por ejemplo, consideremos el grafo ilustrado en la figura… ¿cuál sería la distribución de los nodos para obtener un mínimo cutsize si se quieren obtener 2 particiones?



Usando el algoritmo descrito, este comenzaría toman al nodo A como el primer elemento de la Partición 1 y obtendría las siguientes posibilidades:

Partición 1, compuesta de los nodos A, C y E, el número de canales de comunicación que se han interrumpido (marcados con un triangulo) son los de A hacia B y D, el de B hacia C, C hacia F, D hacia C y E y por ultimo E hacia F. La partición 2 está formada por los nodos B, D y F. El cutsize de esta tentativa de particionamiento es de 7.

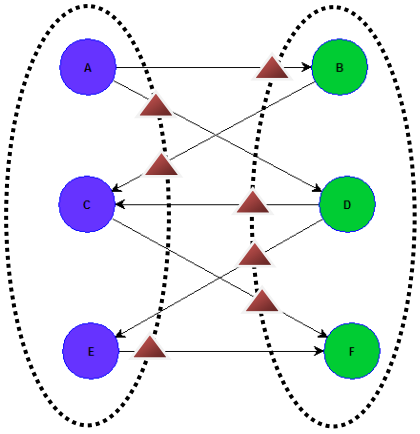


Figura 13. El cutsize resultante al particionar el grafo del ejemplo en esta forma es de 7.

Un segundo intento asocia los nodos A, C, y D a la primera Partición y los nodos B, E y F a la segunda. Al medir el cutsize se obtiene un parámetro de 4 canales interrumpidos.

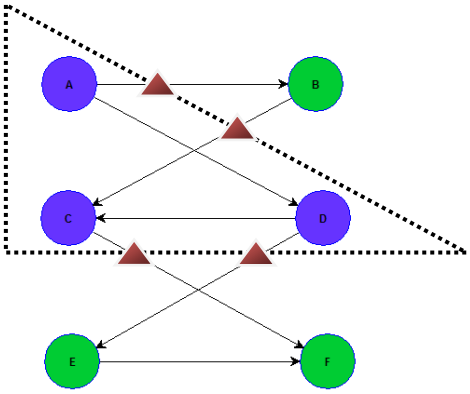


Figura 14. Cutsize = 4 en este caso.

En otro intento, el algoritmo forma la Partición 1 compuesta de los nodos A, B y C y la Partición 2 compuesta por D, E y F. En este caso se obtiene un cutsize de 3 como se muestra en la figura.

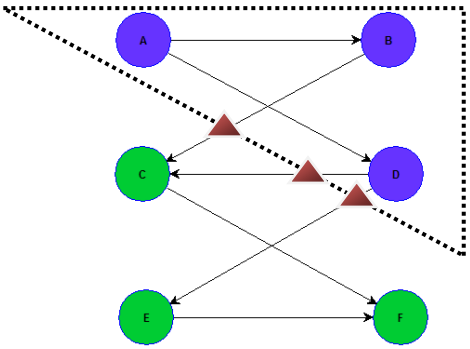
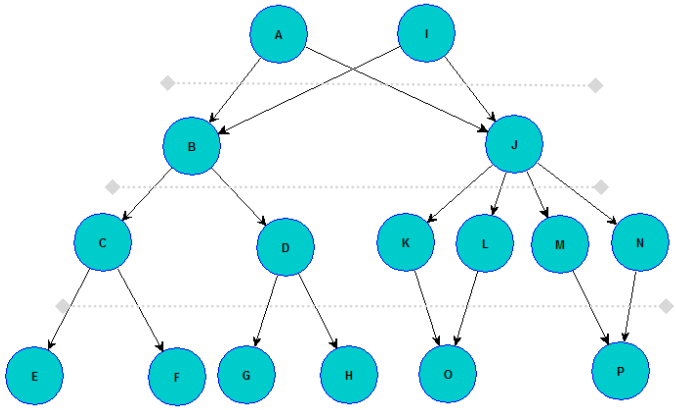


Figura 15. Cutsize = 3. Es el menor cutsize obtenido.

Para este último caso, la medida del cutsize es bastante aceptable, pues de 7 canales de comunicación, se han interrumpido menos de la mitad, lo que repercutirá favorablemente en la carga de comunicación que la red tendrá que transportar entre las estaciones de trabajo.

Particionamiento en recorrido entrada a salida. Este algoritmo se distingue por ser uno de los que mayor concurrencia a la hora de ejecutar las particiones logra por su forma de integrar los subconjuntos. En este caso, el grafo es atravesado de acuerdo a su profundidad, buscando recursivamente en los niveles inmediatamente abajo del nodo actual para seguir formando la partición, logrando una asignación de nodos *fuertemente conectados* en la misma partición. El nodo inicial tiene que ser un nodo en el primer nivel del grafo y lo recorre de la forma descrita hasta llegar a la cantidad dada por el número de nodos entre el número de procesadores disponibles.

Como ejemplo para este algoritmo, supondremos un grafo con 4 niveles de profundidad y 16 nodos repartidos entre esos niveles. El grafo tiene 2 nodos en el nivel inicial, 2 en el segundo nivel, 6 en el tercero y 6 en el nivel final. Asumiendo que tenemos 2 estaciones de trabajo disponibles, esto nos da el numero $^{16}/_{2}$ = 8 nodos asignables por cada partición.



El algoritmo parte haciendo una asignación en profundidad desde el nodo A, incorporando a B por estar conectado al nodo A en el nivel inferior inmediato. Ahora el nodo de referencia es el B, del cual desciende C que también se adjunta a la partición. Finalmente, en un primer recorrido en profundidad llegamos hasta el nodo E asignándolo a la partición 1. Como aun quedan 4 nodos por asignar a la partición, se busca recursivamente las posibles derivaciones de los niveles más altos del grafo. En este caso se buscan mas nodos hijos del último nodo tomado como referencia (nodo C), encontrando al nodo F. Los nodos D, G y H a su vez son asignados a la primera partición siguiendo esta lógica, sumado los 8 nodos.

La segunda partición de arma de forma similar, recorriendo el grafo desde el nodo I, asignando los nodos en el siguiente orden: J, K, O, L, M, P y N al final.

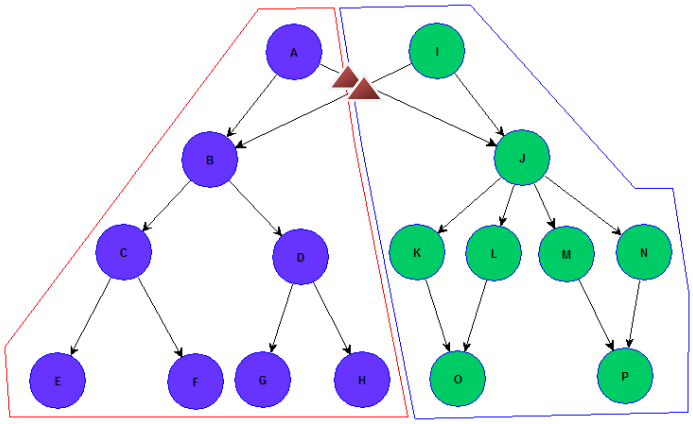


Figura 16. Particionamiento resultante del grafo del ejemplo usando el algoritmo de búsqueda en profundidad para alcanzar concurrencia máxima. En este caso el cutsize resulta óptimo además al ser de 2.

CONCLUSIÓN: PRE-SIMULACIÓN Y ALGORITMO DE PARTICIONAMIENTO:

En resumen, el algoritmo de particionamiento por implementar debe formar subconjuntos con nodos de procesamiento (módulos del circuito) que proporcionen un punto de equilibrio entre máxima concurrencia (paralelismo), mínima latencia de comunicación (mínimo cutsize) y balanceo de carga entre las particiones resultantes. Con este objetivo, va a ser muy conveniente el uso de la información acerca de la carga de transiciones en el circuito recabada durante la pre-simulación para ajustar estos tres compromisos.

SINCRONIZACIÓN.

Existen 3 principales categorías en las que caen los algoritmos de sincronización para una simulación paralela [31]: periodo de avance fijo, conservativo y optimista.

En el primero, el tiempo de simulación de las diferentes particiones es sincronizado según un intervalo fijo preestablecido. Dependiendo en la carga de trabajo de cada estación, algunos procesadores pueden quedar ociosos por esperar a que otros procesos avancen su ejecución.

El segundo y tercer algoritmo avanzan el tiempo de simulación de forma asíncrona con el fin de mantener ocupadas a las estaciones de trabajo.

El algoritmo optimista de ejecución paralela trabaja de forma tal que los procesos mantienen una cola de entrada que almacena los eventos provenientes de otros procesos y ordenados en forma creciente de acuerdo al valor de su marca de tiempo (timestamp), sin embargo si un evento con un timestamp menor al que está en curso de ejecución llegara (conocido como rezagado), se produce una reinstanciación (rollback) y los procesos deben restaurar su estado justo antes de la llegada de dicho evento y meterlo al principio de la cola de eventos. El ejemplo claro es presentado como el algoritmo de Time Warp [32].

Este algoritmo usa colas de salida en las que se almacenan copias de los mensajes mandados (checkpoint) por cada proceso para poder producir los rollbacks necesarios. Sin embargo, la posibilidad de tener rollbacks en cascada es latente y con esto las demandas de memoria serían enormes, volviéndose necesario aplicar otras técnicas de ahorro de memoria como el checkpointing periódico en lugar de hacerlo cada nuevo evento.

Por último, el algoritmo de sincronización conservativo trabaja usando esta misma idea donde cada proceso tiene asociadas colas de eventos de entrada y de salida que actúan como buffer de almacenamiento para enviar hacia otro proceso o para recibir desde otro, pero sin asociar marcas de tiempo para cada evento, ahorrando memoria de este modo.

Respecto a la política de sincronización elegida para el ambiente distribuido, se desarrolló precisamente un algoritmo de tipo *conservativo,* basado en el principio de causalidad (el principio de causa y efecto), lo que nos asegura un mecanismo controlado y caracterizado por su comportamiento bloqueante [33].

En una simulación conservativa, si la cola de entrada para algún proceso queda vacía, dicho proceso se bloquea y espera cierto tiempo para recibir mensajes de los demás procesos asociados a él, con esto logramos una muy decente demanda de memoria. La desventaja de este algoritmo es que dicha suspensión en la ejecución puede incrementar el tiempo de simulación y peor aun caer en un deadlock si se perdió un mensaje de dos procesos que esperaban entre sí. Por esto, para usar este algoritmo de sincronización debemos encontrar métodos que eviten deadlocks e inaniciones o bien que las detecten y rompan [34].

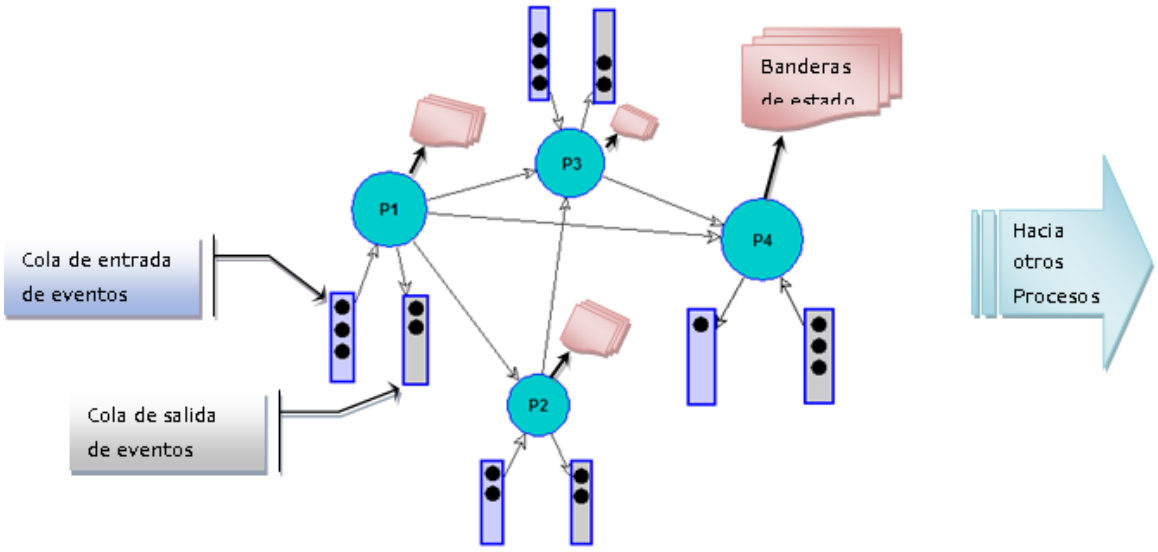


Figura 17. En el algoritmo de sincronización conservativo, los procesos estarán activos mientras haya eventos/datos en su cola de entrada.

Un posible mecanismo que solucione esta problemática consiste en empaquetar una marca de tiempo o timestamp en cada cierto número de mensajes que un proceso envía su vecindad, comprometiendo a los demás procesos para lleguen hasta esa marca y poder avanzar. La obvia desventaja de esta aproximación es que para un proceso que envía un gran número de mensajes a otros, la latencia de comunicación aumenta. A pesar de esto, si se mandaran mensajes solo sobre demanda, el costo de dicha latencia se podría reducir considerablemente [35].

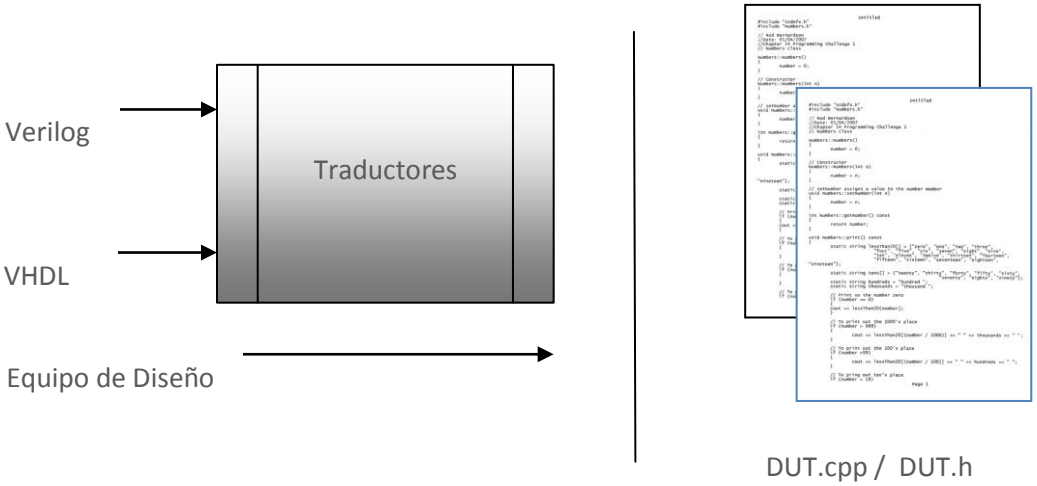
IMPLEMENTACIÓN DE LA METODOLOGÍA

El diseño de un sistema digital de alta escala de integración (VLSI) comienza típicamente escribiendo un modelo de referencia en algún lenguaje de programación de alto nivel (C/C++, Perl, etc.) que será utilizado después para generar el modelo usando un lenguaje de descripción de hardware como Verilog, VHDL, SystemVerilog o SystemC. De hecho, las herramientas EDA de hoy en día permiten tener código en cualquiera de estos lenguajes describiendo módulos, entidades, testbenches, etc. con la finalidad de integrar un solo sistema.

El lenguaje que usaremos en el desarrollo de esta metodología es SystemC. Hay varias rutas por las cuales podemos llegar a tener el código que describe a un sistema en este lenguaje listo para ser simulado: traducido de algún otro lenguaje como VHDL o Verilog, o escrito directamente en SystemC por el equipo de diseño.

Hasta ahora hemos probado varias alternativas de traducción y para Verilog, los traductores V2SC de Mazdak & Alborz Design Automation y Verilator de Veripool han pasado satisfactoriamente las traducciones de Verilog a SystemC.

Para VHDL, el traductor dentro del VSP Compiler resultó ser el mejor, aunque es una herramienta comercial (se corrió un trial). Otras opciones son VH2SC o VHDL-to-SystemC-Converter de www-ti.informatik.uni, sin costo pero menos efectivas.



SYSTEMC COMO LENGUAJE DE DESCRIPCIÓN DE HARDWARE.

SystemC está basado en C++, uno de los lenguajes más ampliamente usados por ingenieros de software. El lenguaje consiste en un conjunto de bibliotecas que contienen clases con las construcciones necesarias para implementar sistemas de hardware desde varios niveles de abstracción en C++ (desde modelos abstractos hasta modelos RTL precisos en tiempo). Los mismos beneficios derivados de la estandarización de Verilog y VHDL a nivel transferencia de registros pueden ser obtenidos con este lenguaje, mientras que es posible escribir código para una especificación ejecutable rápidamente para después irla refinando y bajar de nivel hasta llegar al RTL o incluso Gate-Level.

La versatilidad de este lenguaje es una de las características que nos llevo a utilizarlo en el desarrollo de este proyecto, además de que, por estar comprendido en un ambiente de desarrollo bajo C++, diferentes algoritmos de sincronización para la ejecución distribuida de la simulación pueden ser probados directamente dentro del diseño sin tener que usar interfaces para conectarlo con otra plataforma de desarrollo (como lo requeriría un modelo en Verilog si se le quisiera implementar una comunicación con sockets con otro modulo vía TCP/IP por ejemplo).

Por otro lado, ya que C++ aplica la orientación a objetos -paradigma que de hecho fue creado para técnicas de diseño de hardware- la abstracción de datos, propiedades, comportamientos y atributos de un modulo (clase) puede ser el principio para elaborar un diseño jerárquico. Para el desarrollo de este proyecto, nos valdremos de la plataforma de Microsoft Visual Studio 2008 en conjunto con SystemC para modelar el hardware y hacer las pruebas necesarias que retroalimentaran el análisis previamente hecho. La información para instalación y set up del ambiente puede ser encontrada en [36].

Un buen ejemplo de cómo se construye una jerarquía para cierto diseño usando SystemC lo podemos encontrar en un sumador de 8 bits. Con una perspectiva de desarrollo bottom-up de este diseño, primero programamos las instancias mínimas que procesarán los datos a la entrada: las compuertas lógicas.

En SystemC, todos los módulos se especifican en un par de archivos (Modulo_X.h y Modulo_X.cpp) para después incluir el archivo cabecera en cualquier otro diseño donde se quiera usar (instanciar un objeto de esa clase). Considerando el ejemplo del sumador, primero creamos un modulo que se comportará como una compuerta lógica XOR de dos entradas. Por convención, en el archivo .h escribimos la definición del modulo (nombre, puertos de entrada, salida, etc.), así como el nombre de la(s) función(es) donde se escribirá el código para su comportamiento. En el archivo .cpp se hace uso de esos puertos/atributos declarados para implementar el funcionamiento del modulo:

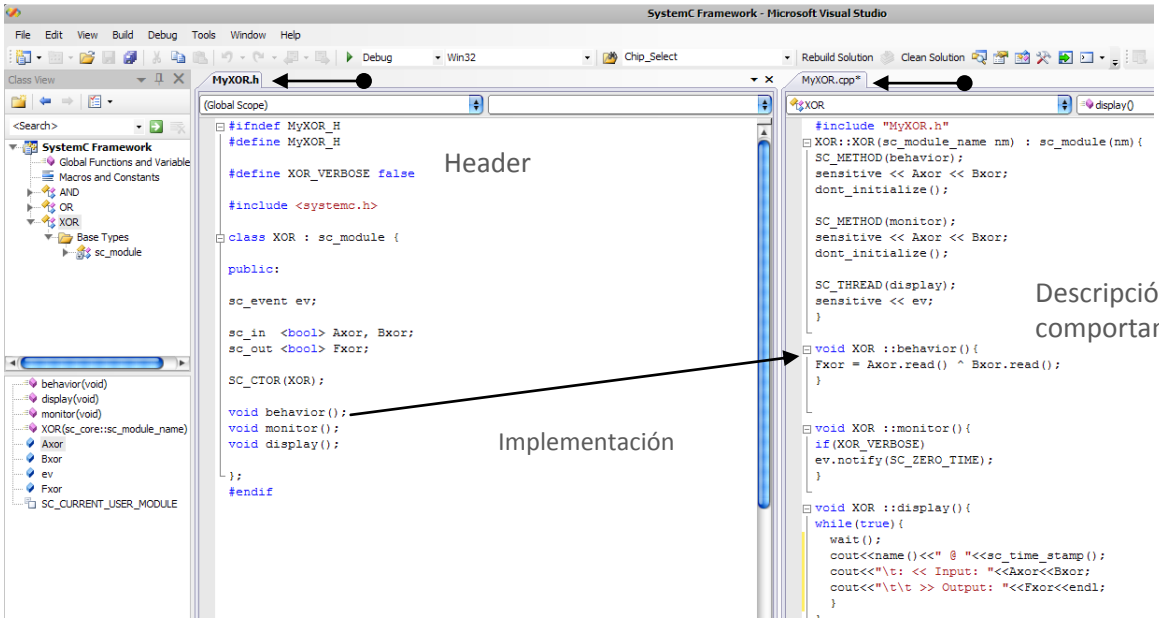


Figura 18. Archivos .h y .cpp de un modulo [compuerta AND] en SystemC

De este modo se construye una clase que define un modulo en SystemC, mismo modulo que podrá ser instanciado en algún otro diseño declarándolo como un objeto (submódulo) dentro del cuerpo del programa (recuerde que C++ es un lenguaje orientado a objetos). Continuando con nuestro ejemplo, para poder construir un sumador de un bit necesitamos codificar el funcionamiento de un par de compuertas más, la AND y la OR.

El procedimiento es el mismo para el caso de estas compuertas, el único cambio se hace en la función que implementa el comportamiento del modulo (en el caso de la XOR, esa función es behavior()). Para el caso de la AND, ahora se usa el operador && para hacer la operación and de sus dos entradas y asignarlo a la salida. En el caso de la OR, el operador de C++ que se usa es ||.

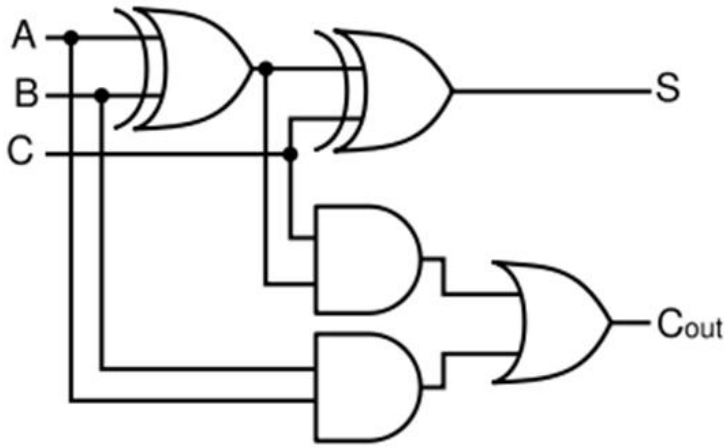


Figura 19. Sumador completo de un bit con entrada de acarreo.

Una vez que las tres clases se han codificado, es posible crear un nuevo modulo que use instancias de ellas y, usando los puertos de entrada y salida de cada uno, se interconecten de tal modo que el comportamiento general de dicho módulo dependa del comportamiento particular de sus submodulos y del flujo de dichas conexiones.

En el caso del sumador completo, tenemos que instanciar 2 objetos del 'tipo' XOR, otros dos del tipo AND y un último objeto del tipo OR para después conectarlos de forma apropiada y obtener el funcionamiento de un sumador de un bit.

En la figura debajo podemos observar cómo se construye el sumador de un bit. Se instancian el número y tipo de compuertas ya descritas y se interconectan usando señales auxiliares. Este proceso se le conoce como 'port binding' en SystemC.

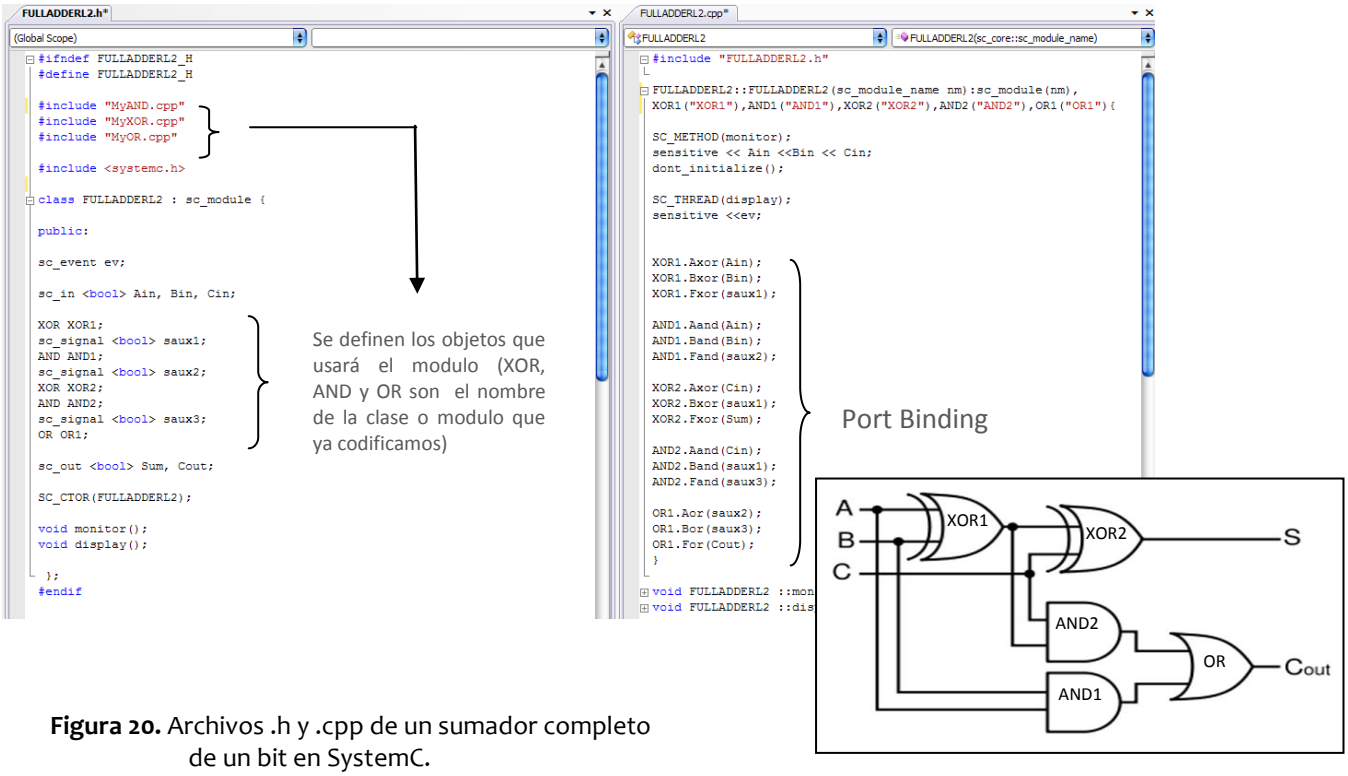


Figura 20. Archivos .h y .cpp de un sumador completo de un bit en SystemC.

En este caso no fue necesario declarar una función de comportamiento ya que el comportamiento del sumador completo esta dado por los submódulos que contiene el diseño. Para el sumador no se necesita más que conectar las compuertas según la figura.

Hasta este punto ya tenemos una jerarquía de dos niveles: los módulos que simulan las compuertas lógicas y el top level que es el sumador completo. Para seguir con el ejemplo, con el fin de diseñar del sumador de 8 bits, podemos replicar este sumador de un bit pasando por otros dos niveles jerárquicos. El siguiente nivel consiste en instanciar cuatro módulos del tipo sumador completo e interconectarlos de tal forma que sumen dos vectores de 4 bits cada uno.

El arreglo en cascada de 4 sumadores de un bit completo es el equivalente al sumador de cuatro bits, conectando el bit de acarreo del primer sumador con el bit de entrada del siguiente, y así sucesivamente como se muestra en la figura:

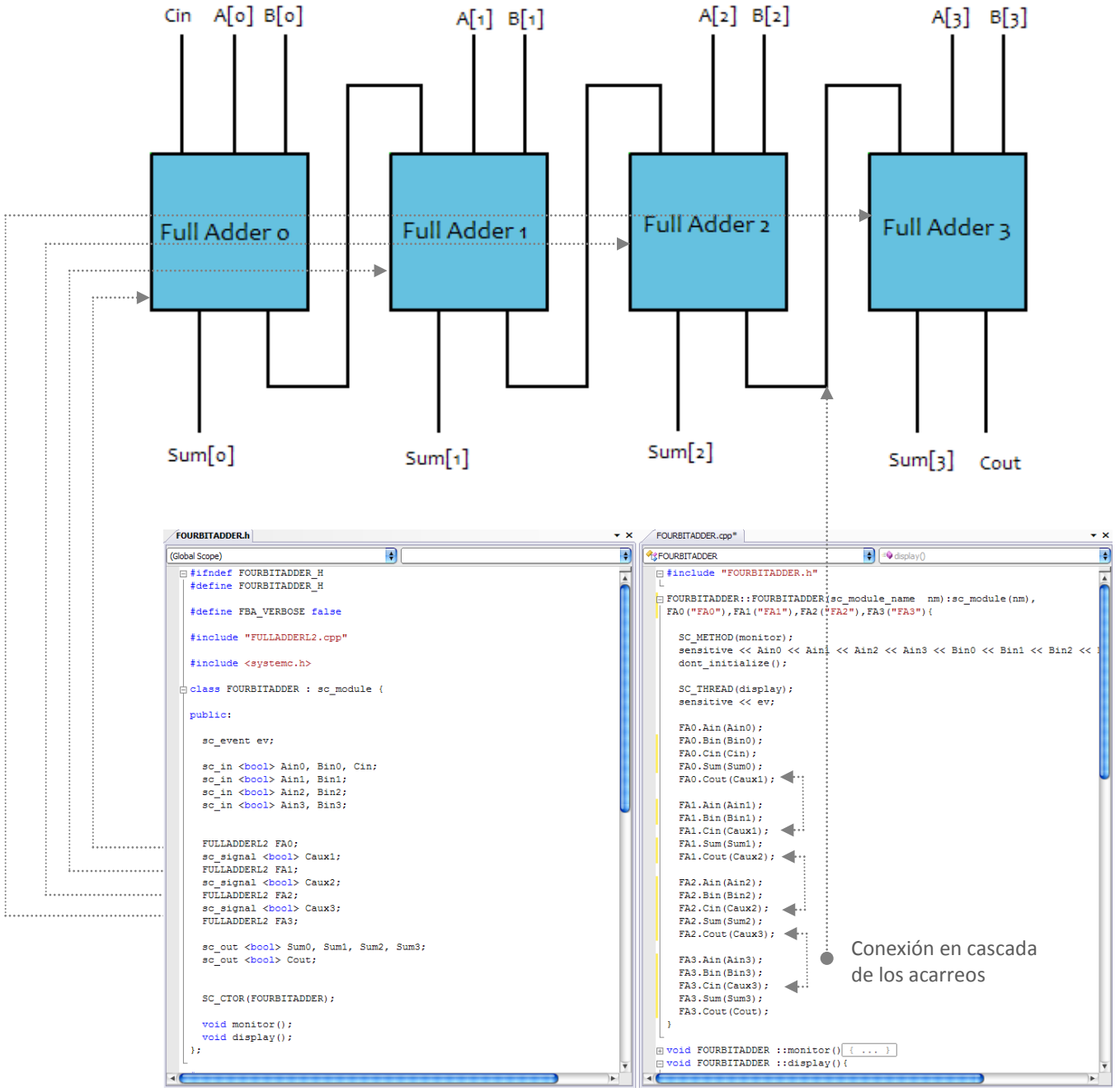


Figura 21. Sumador de 4 bits construido a partir de sumadores completos de un bit conectados en cascada. La implementación en SystemC se muestra abajo.

El diagrama de clases que se puede obtener en Visual Studio muestra que la composición del modulo actual es efectivamente de 4 entidades de tipo 'FULL_ADDER', cada uno de los cuales a su vez contiene submodulos ANDs, XORs y ORs.

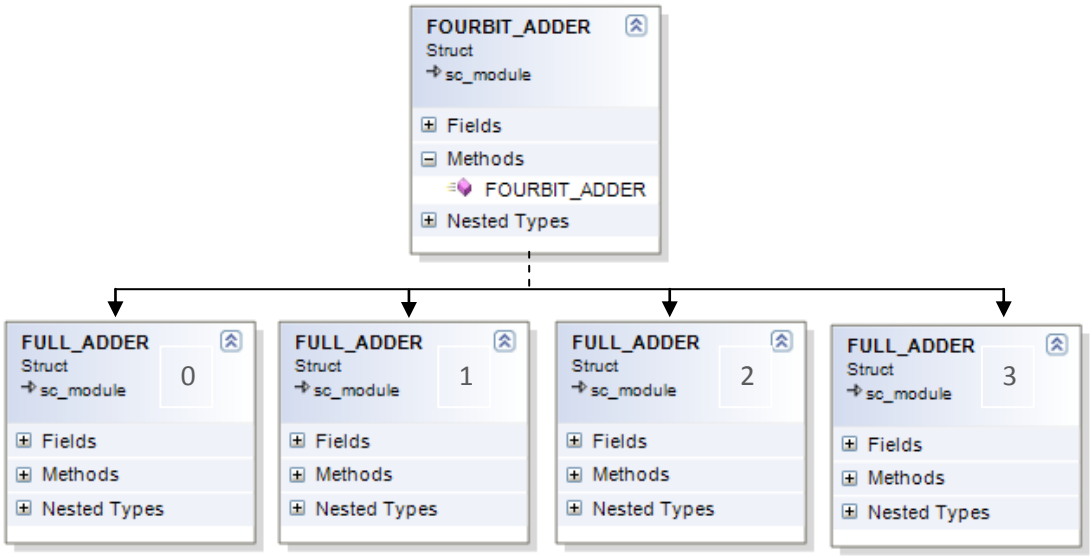
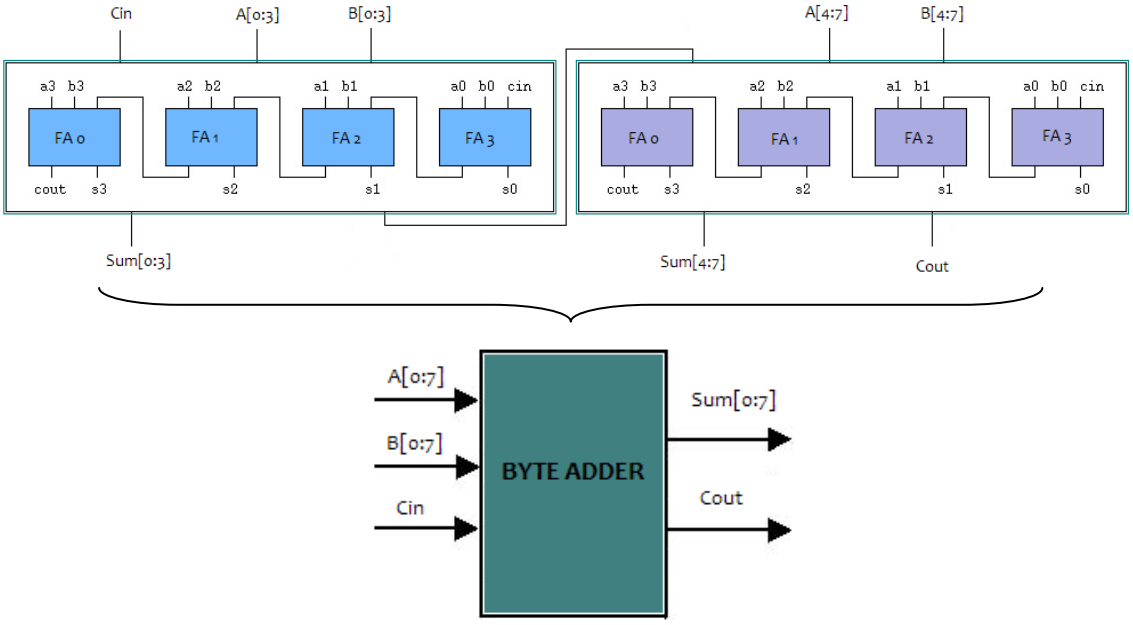


Figura 22. Diagrama de clases del sumador de 4 bits en SystemC usando la plataforma Visual Studio.

Finalmente, en un cuarto nivel de jerarquía, podemos instanciar dos módulos sumadores de cuatro bits y en la misma forma conectarlos para obtener un sumador de 8 bits.



Este ejemplo nos da un buen panorama de cómo el paradigma de la programación orientada a objetos puede ser aplicado también a diseños de hardware en SystemC.

Mediante un esquema de desarrollo bottom-up se fue construyendo la jerarquía final del sistema pasando por 3 niveles de submódulos dentro de ella (BYTEADDER contiene FOUR_BIT_ADDERs que a su vez contienen FULL_ADDERs, los cuales por ultimo están construidos con submodulos ANDs, XORs y ORs).

El crecimiento de tal jerarquía para el ejemplo considerado puede incluso seguir creciendo hasta crear un sumador de 32 o 64 bits con este esquema de diseño.

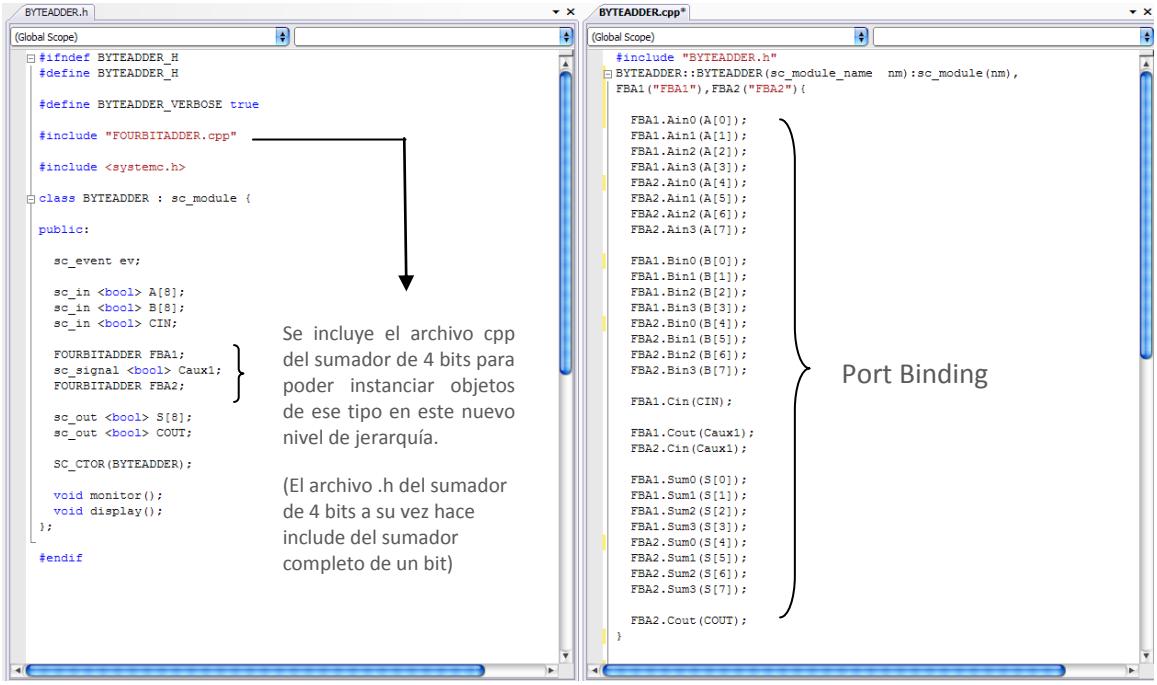


Figura 23. Para concluir el ejemplo, se muestran los archivos cpp y h del sumador de 8 bits con entrada de acarreo.

PRE-SIMULACIÓN Y MONITOREO

Con el fin de revelar aquellos módulos que presentan una mayor cantidad de transiciones durante la simulación, pretendemos generar una estadística basada en el monitoreo de las señales implicadas en el funcionamiento del DUT (diseño bajo prueba) usando un testbench de entrada aleatorio para estimular el circuito.

Con este fin, primero debemos conocer los nombres y tipos de dato que manejan las señales que tendrán asociados dichos monitores - cierto diseño puede tener, por ejemplo, señales que operen sobre números enteros, bits, números flotantes, caracteres, vectores de cualquiera de los anteriores, etc. -

Para lograr la identificación de estos puntos de monitoreo, haremos un parsing del archivo .h del DUT, ya que es en ese archivo donde se definen los nombres y tipos de los puertos de entrada, salida, señales internas, interfaces, etc. del circuito. Una vez que se ha recolectado esta información, se debe crear y vincular un monitor por cada señal encontrada e implementar una función dentro del diseño que se inicialice desde la misma instanciación de modulo y mantenga a los monitores registrando los cambios durante la pre-simulación.

Sin embargo, SystemC no cuenta con monitores de señales predefinidos que puedan ser asociados a cualquier puerto o interfaz (como lo hace SystemVerilog), así que se agregó la tarea de implementar una biblioteca de monitores para las señales con los tipos más usados en SystemC.

Esta biblioteca cuenta hasta ahora con 20 monitores distintos para 4 tipos de puertos. El funcionamiento de cualquiera de estos monitores es idéntico: con el parsing del archivo primero se obtiene el tipo de puerto a monitorear, este 0puede ser de entrada, salida, bidireccional o simplemente una señal interna del circuito. Después se verifica el tipo de dato que se está conduciendo en el puerto en cuestión para crear una variable dentro del monitor del mismo tipo. Esta variable tiene como objetivo ir guardando el estado anterior de la señal para continuar comparando cada vez que la simulación avance e incrementar el contador específico para tal monitor en caso de haber cambiado de valor.

A continuación se enlistan los puertos y tipos de señales soportados en los que se han implementado monitores hasta ahora. Para tipos de puertos bidireccionales que manejan vectores aun no se han programado los monitores.

Puerto del Sistema	Tipos de dato que maneja	Soporta Monitoreo sobre Vector	Ejemplo
sc_in	bool char int float double	✓	`sc_in <bool> A;` `sc_in <int> A[8];`
sc_out	bool char int float double	✓	`sc_out <char> A;` `sc_out <float>A[4];`
sc_signal	bool char int float double	✓	`sc_signal<double>A;` `sc_signal<int>A[16];`
sc_inout	bool char int float double	AUN NO	`sc_inout <bool> A;` `sc_inout <int>A[2];`

Tabla 1. Listado de monitores disponibles en la biblioteca SC_MONITORS para puertos del DUT

Complementarios a estos monitores, quedan por implementar dentro de la biblioteca los tipos de dato declarados por el usuario, ya que en C++ esto es posible. Sin embargo, se ha creado una plantilla para probar cualquier nuevo tipo de dato sobre estos puertos y monitorearlo durante la pre-simulación.

Para implementar un monitor de un tipo de dato nuevo o definido por el usuario, simplemente sobrecargamos la función utilizada en SC_MONITOR.h con los parámetros apropiados.

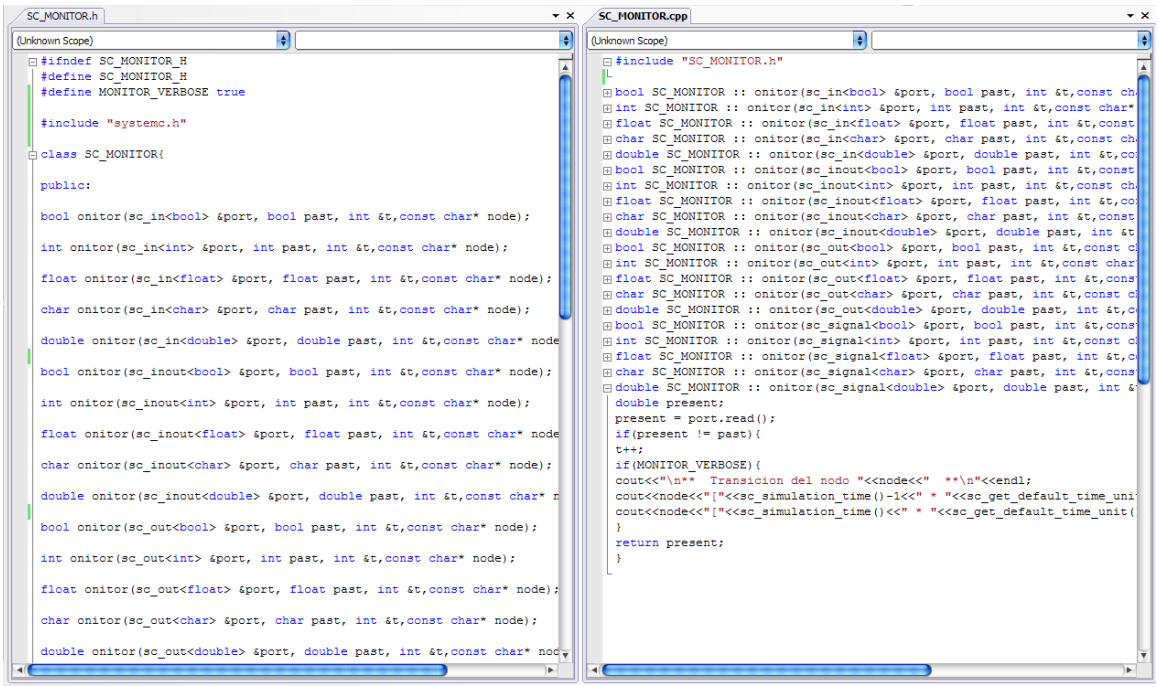


Figura 24. Archivos .h y .cpp de SC_MONITORS, la biblioteca de monitores para señales en SystemC.

En la plantilla TEST_MONITOR se pueden probar nuevos tipos de datos a ser monitoreados, solo hay que especificar una nueva señal que lo use, aplicar el monitor usando la metodología y correr la pre-simulación para ver el reporte.

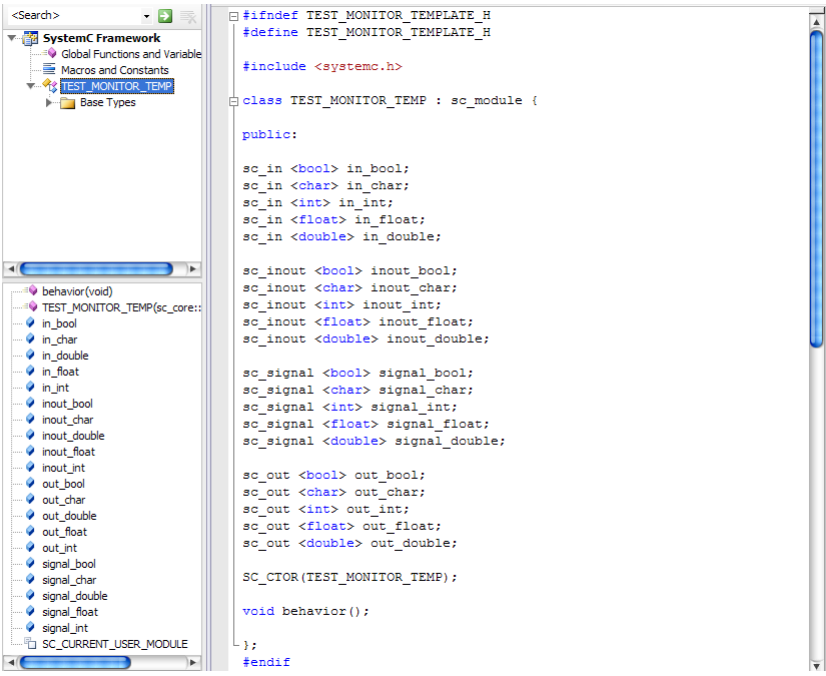


Figura 25. Plantilla para probar nuevas señales con monitores asociados.

CONSTRUCCIÓN DEL GRAFO DEL CIRCUITO

Una vez corrida la pre-simulación y con la estadística de actividad del DUT, podemos usar la métrica obtenida para construir el grafo del diseño en donde el peso de cada arco estará dado por la cantidad de transiciones que se generaron en el nodo de origen y así tener una medida del costo de comunicaciones que podría tener para una partición separar un canal de comunicación dado.

La idea es armar el grafo de tal forma que cada nodo tenga un peso asociado correspondiente a la carga computacional y los arcos que salgan de él tengan también un peso asociado a la cantidad de transacciones que el nodo exhibió durante la pre-simulación.

Con esto en mente, ampliamos el parsing del diseño para encontrar después de los nodos, los posibles arcos que representaran interacción entre señales del DUT.

Por ejemplo, al encontrar una asignación del tipo:

```
Fand = Aand.read() && Band.read();
```

dentro de la descripción del circuito (en este caso la linea se tomó del modulo AND), podemos ver que la señal Fand depende de dos señales, asi que en este caso dos arcos correspondientes se tienen que mapear al grafo saliendo de los nodos Aand y Band, y conectandolos hacia el nodo Fand. Cabe mencionar que este parsing se hace en el archivo cpp del DUT, ya que es en este donde se establece el comportamiento del circuito mediante asignaciones, port binding, etc.

Combinando la informacion de nodos, arcos y pesos asociados construimos el grafo usando una herramienta que nos dara una visualizacion bastante aclarativa del trabajo que el algoritmo de particionamiento tiene que hacer. Esta herramienta es una aplicación en java para representacion de grafos disponible en [39] llamada Vertex-Edge Graph y especializada en este tipo de aplicaciones de matematicas discretas. El formato para describir el grafo sin embargo, no es usado como una matriz de adyacencia, sino mediante un archivo de texto con los parametros que se definen a continuación.

En el archivo del grafo se deben especificar algunas directivas, por ejemplo si el grafo es dirigido, si tendrá pesos, el color de los nodos, etc. Para definir un vértice, el formato es:

X(e:Pos_x,Pos_y, Ancho, Alto, n:Alias, c: RRR,GGG,BBB),

En donde X es el nodo, e es la posición del nodo en pantalla (coordenadas x,y), Ancho y Alto definen el tamaño del nodo, el cual tenemos planeado que este asociado con la carga computacional del nodo. Es decir, mientras más exhaustivo sea el trabajo de ese nodo, mas grande se desplegara en comparación con los demás nodos. Alias es el nombre a mostrar del nodo y c define el color en formato RGB.

En cuanto a los arcos, el programa los lee de esta forma:

{N_Orig, N_Dest, d:1, v:Peso_Arco, [c:RRR,GGG,BBB]}

En donde N_Orig es el nodo de donde sale el arco y N_Dest es el nodo al que se conecta, d:1 significa la dirección ya descrita pues si se especifica d:0, se invierte el arco; v especifica el peso del arco, en este caso el numero de transiciones que se presentaron de N_Orig hacia N_Dest; y por último, el arco también puede tener un color especifico (opcional).

Con la información de todos los nodos y todos los arcos se genera entonces el archivo para que sea interpretado por la aplicación. El applet lee un archivo con un conjunto de vértices seguido por un conjunto de arcos descritos con el formato visto:

directed, weighted,

{[LISTA DE VERTICES],

[LISTA DE ARCOS] }

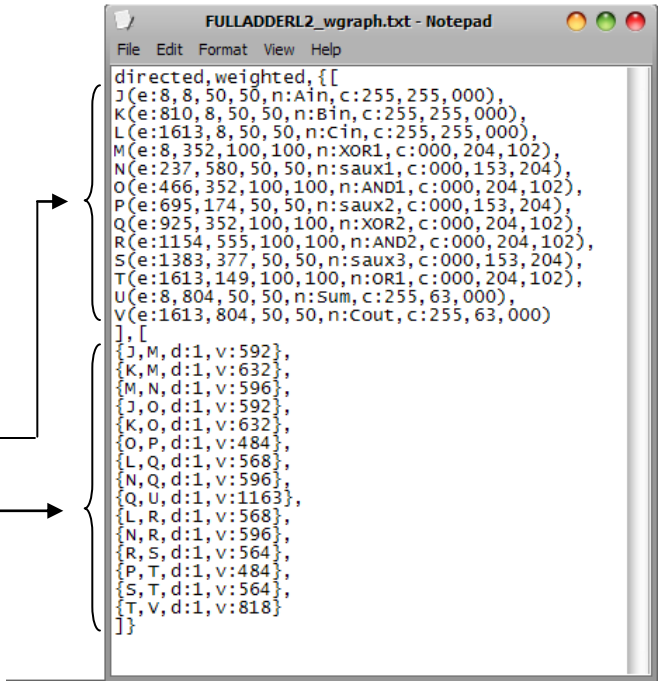


Figura 26. Archivo con el formato para descripción de un grafo usado por la aplicación Vertex-Edge Graph

Un ejemplo de cómo será interpretada la información una vez que se haya reunido la información de la pre-simulación y el parsing de los archivos del DUT es el siguiente:

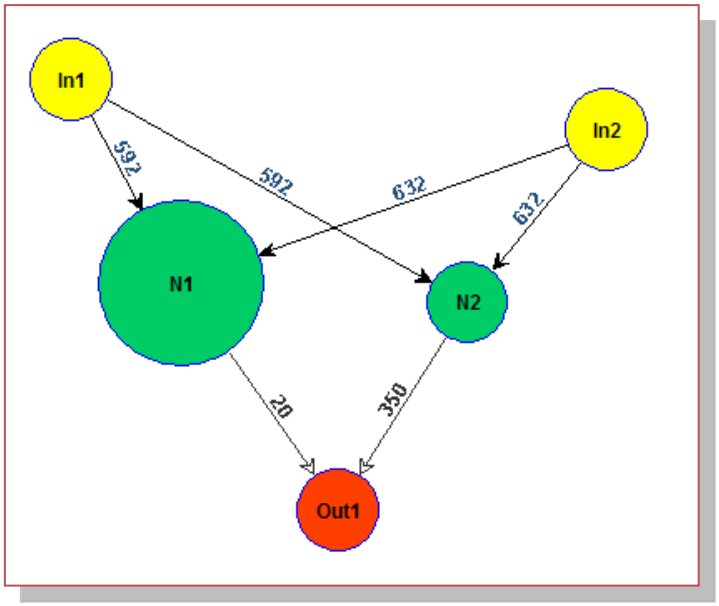


Figura 26. Ejemplo de la representación en grafo de un DUT después de haber corrido la pre-simulación.

Con este ejemplo podemos hacer varias observaciones que serán validas para cualquier grafo que describa un DUT dentro de nuestra metodología:

- Los nodos de entrada serán amarillos.
- Los nodos que representan submodulos o instanciaciones de otros módulos dentro del DUT se representaran con color verde.
- El tamaño del nodo será proporcional a la carga de cómputo requerida por el nodo para procesar sus datos
- La cantidad de transacciones de un nodo hacia otro es el número mostrado sobre el arco que los interconecta.
- Los nodos que desempeñen un papel de señal interconectando uno o varios submodulos dentro del DUT serán pintados de color azul (en el ejemplo no hay este tipo de nodos.

- Los nodos rojos representan nodos de salida del circuito.

Respecto al ejemplo, el DUT está compuesto por 2 nodos de entrada, 1 de salida y dos instanciaciones de otros submodulos (N1 y N2). La información que podemos obtener a primera mano del grafo es que tanto N1 como N2 son muy diferentes. Mientras que N1 es un submódulo con gran carga computacional, su nivel de actividad hacia la salida es poca, sin embargo N2 representa un submódulo con más ligera carga computacional pero con mayor nivel de comunicación hacia la salida.

Esta información relevante del DUT es la que el algoritmo de particionamiento deberá tomar en cuenta al seccionar el circuito y distribuir las cargas computacionales y de comunicaciones apropiadamente sobre la red de estaciones de trabajo para su ejecución en paralelo.

FORMATO DEL ARCHIVO DE PARTICIONAMIENTO

Una vez que contamos con la representación del sistema en forma de un grafo dirigido y con sus pesos asociados en cuanto a demanda de ciclos de cómputo y el número de transiciones en los canales de comunicación de cada nodo, el algoritmo de particionamiento debe leer el archivo que describe al grafo únicamente con la información de los nodos y los pesos, ya que los datos sobre la posición X, Y, y los alias están por demás en el archivo para el programa Vertex-Edge descrito anteriormente. Después de filtrar esta información, se pretende tener un archivo de particionamiento con un formato estándar para aplicar diferentes algoritmos de particionamiento al sistema.

Con respecto al formato del archivo con el que el algoritmo de particionamiento va a trabajar, se decidió escribir un archivo de texto describiendo en cada línea la información por nodo correspondiente, generando un archivo con n líneas donde n es el numero de nodos del grafo.

El formato es como sigue:

- La primer línea contiene 3 parámetros: n, a, w. Donde n es el numero de nodos, a es el total de arcos y w es un par de bits para especificar los pesos que tendrá asociados a los nodos y a los arcos del grafo a particionar.

- Las siguientes n líneas corresponden a la información de los n nodos del grafo.

La tabla debajo describe los posibles valores para w y su significado. Este valor puede ser omitido si no se consideraran pesos asociados a nodos y arcos del grafo (00).

w	Significado
00	Sin pesos asociados ni para nodos, ni para arcos.
01	Se especifica para asociar pesos a los arcos
10	Con pesos asociados a los nodos.
11	Con pesos asociados tanto para nodos como para arcos

Las restantes n líneas del archivo (sin considerar comentarios, de la forma "% comentario … ") describen toda la información pertinente que corresponde al nodo siendo descrito. Esa información consta de la siguiente estructura:

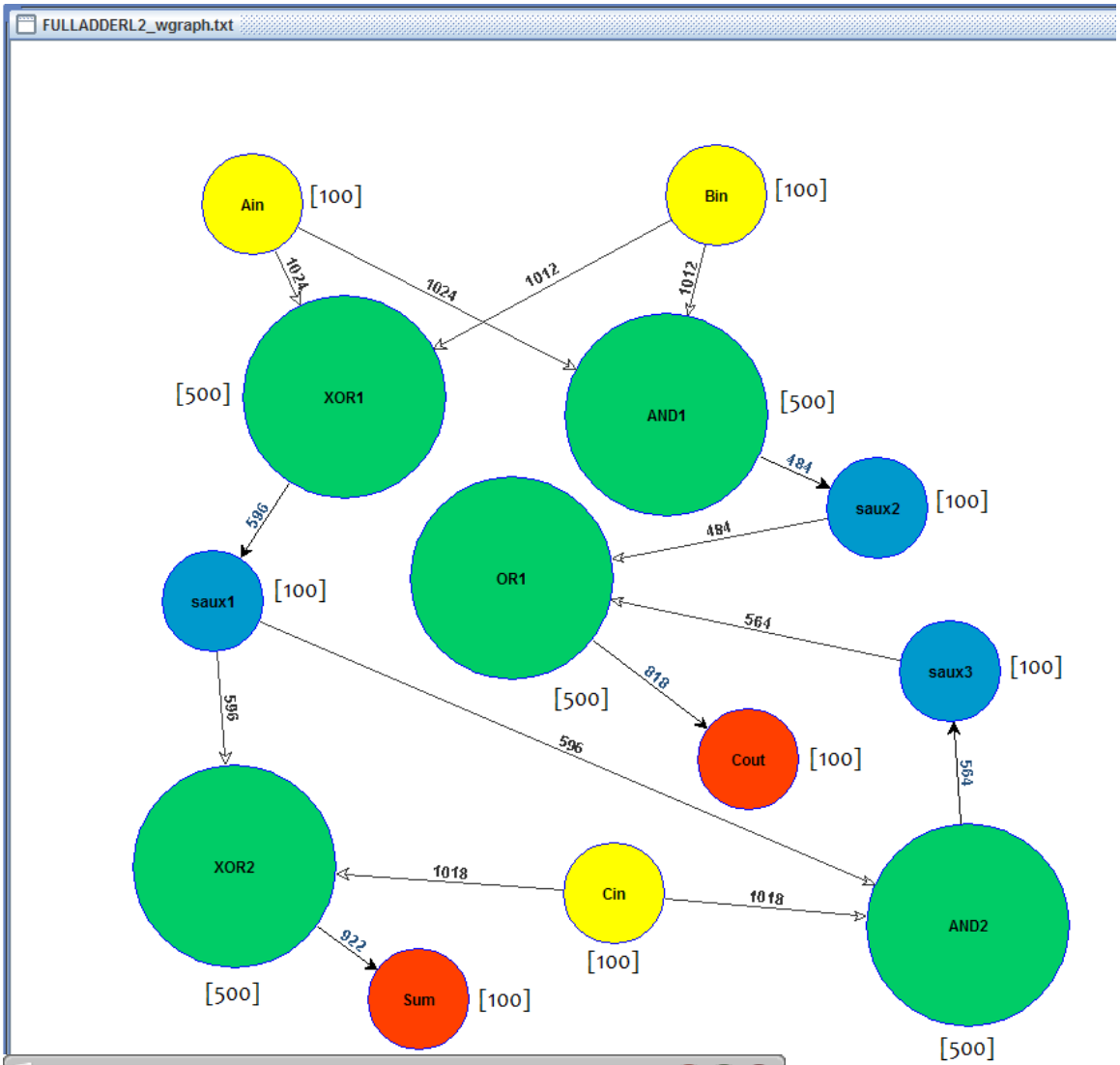
Md/If, In/Pr/Out NODO: {$PESO_{NODO}$,} [N_ADY_1,{$PESO_{ARCO1]}$}], [N_ADY2,{$PESO_{ARCO2}$}], …. , [N_ADY_k,{$PESO_{ARCOk}$}]

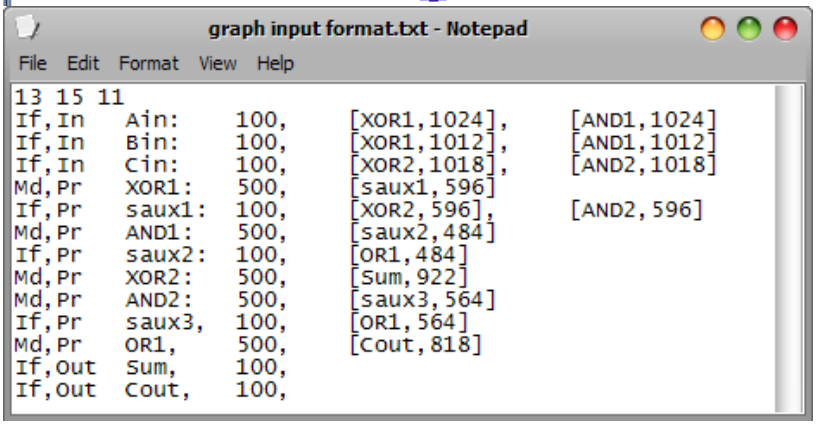
Donde:

- Md indica si el nodo es un submódulo del sistema o If indica que se trata de una interfaz del sistema.
- In/Pr/Out indica si el nodo es de entrada, de procesamiento, o de salida del sistema.
- NODO es el nombre del nodo, $PESO_{NODO}$ es el peso asociado a la carga de trabajo del mismo, y las duplas [N_ADY_k, $PESO_{ARCOk}$] son los posibles nodos conectados o adyacentes al nodo en cuestión y el peso asociado a la conexión entre esos dos nodos.

  *Note que los parámetros entre llaves {} dependen del parámetro w, ya que si se estuviera especificando un grafo sin pesos, esta información no sería tomada en cuenta.

Es así como cada nodo tiene un vector de adyacencia asociado cuya información es sobre pesos y los nodos con los que está conectado.




graph input format.txt - Notepad

File Edit Format View Help

```
13 15 11
If,In   Ain:    100,    [XOR1,1024],    [AND1,1024]
If,In   Bin:    100,    [XOR1,1012],    [AND1,1012]
If,In   Cin:    100,    [XOR2,1018],    [AND2,1018]
Md,Pr   XOR1:   500,    [saux1,596]
If,Pr   saux1:  100,    [XOR2,596],     [AND2,596]
Md,Pr   AND1:   500,    [saux2,484]
If,Pr   saux2:  100,    [OR1,484]
Md,Pr   XOR2:   500,    [Sum,922]
Md,Pr   AND2:   500,    [saux3,564]
If,Pr   saux3,  100,    [OR1,564]
Md,Pr   OR1,    500,    [Cout,818]
If,Out  Sum,    100,
If,Out  Cout,   100,
```

Figura 27. Representación del grafo que describe al sumador de 1 bit con su correspondiente formato para ser leído por el algoritmo de particionamiento.

De este modo, si el nodo v tiene k nodos adyacentes a el, entonces la línea para v en el archivo del grafo tendrá 1+2*k parámetros, el primero para el peso de v y las siguientes k parejas para el nombre del nodo al que está conectado y el peso en el arco que los une (canal de comunicación).

En la figura 44 se muestra el grafo con pesos asociados del sumador de 1 bit con acarreo y el archivo de texto que lo describe, mismo que será usado como entrada para el algoritmo de particionamiento.

ALGORITMO DE PARTICIONAMIENTO BASADO EN MÓDULOS

El esquema de particionamiento que adoptamos tiene como objetivo mantener una granularidad de particionamiento un tanto más gruesa que los algoritmos actuales, los cuales adoptan una granularidad muy fina dividiendo hasta niveles de compuerta provocando un gran costo de comunicaciones entre procesos, degradando considerablemente el desempeño de la simulación paralela.

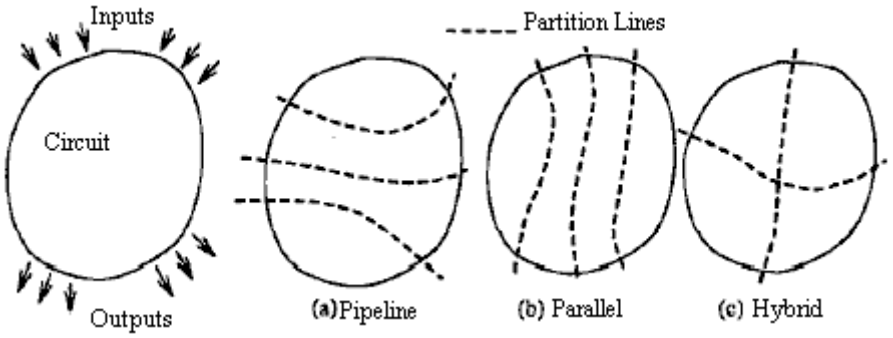


Figura 28. Tres patrones de particionamiento.

En nuestra metodología, la granularidad de partición es por modulo (SC_MODULE en SystemC o lo que equivale a la construcción 'module' de Verilog). Esto reducirá bastante la complejidad del grafo que representa el diseño y asegurará que los componentes que pertenecen a un mismo

módulo sean mapeados siempre al mismo proceso y este, a su vez, ejecutado en el mismo procesador.

[40] demuestra que una forma de particionamiento por modulo en conjunto con un patrón de formación de particiones paralelo da un resultado balanceado entre costo de comunicación y carga de trabajo sobre procesadores al distribuir la simulación.

La implementación de nuestro algoritmo consiste en identificar los submodulos que instancia el DUT en su definición. Para hacer esto, el archivo .h del DUT debe hacer #includes de las clases que contienen la implementación de los submodulos que se quieren usar –como ya lo vimos en el ejemplo del sumador-.

Al hacer un recorrido por cada uno de estos #include, podemos obtener la jerarquía y la información de puertos de entrada/salida de todos aquellos submodulos que están siendo instanciados y que compondrán al DUT. Esta fase donde se localizan submodulos se ha incorporado con la fase de parsing del archivo para encontrar los puertos del DUT –la fase que estudiamos en la construcción del grafo-.

Haciendo primero un parsing de las clases que se están incluyendo en el archivo del DUT, podemos mapear cada uno de esos submodulos con nodos en el grafo del circuito. Estos submodulos se instancian cuando en el DUT se declaran con el nombre del modulo más un identificador, por ejemplo las líneas dentro del sumador:

```
XOR XOR1;
AND AND1;
XOR XOR2;
AND AND2;
OR OR1;
```

estan especificando que el DUT va a constar de 5 submodulos, 2 del tipo XOR, 2 del tipo AND y 1 del tipo OR.

La información obtenida por esta etapa de descomposición nos ayudará a "reconstruir" el diseño mapeando cada submódulo a una entidad de ejecución por separado, con la idea de que se ejecuten concurrentemente.

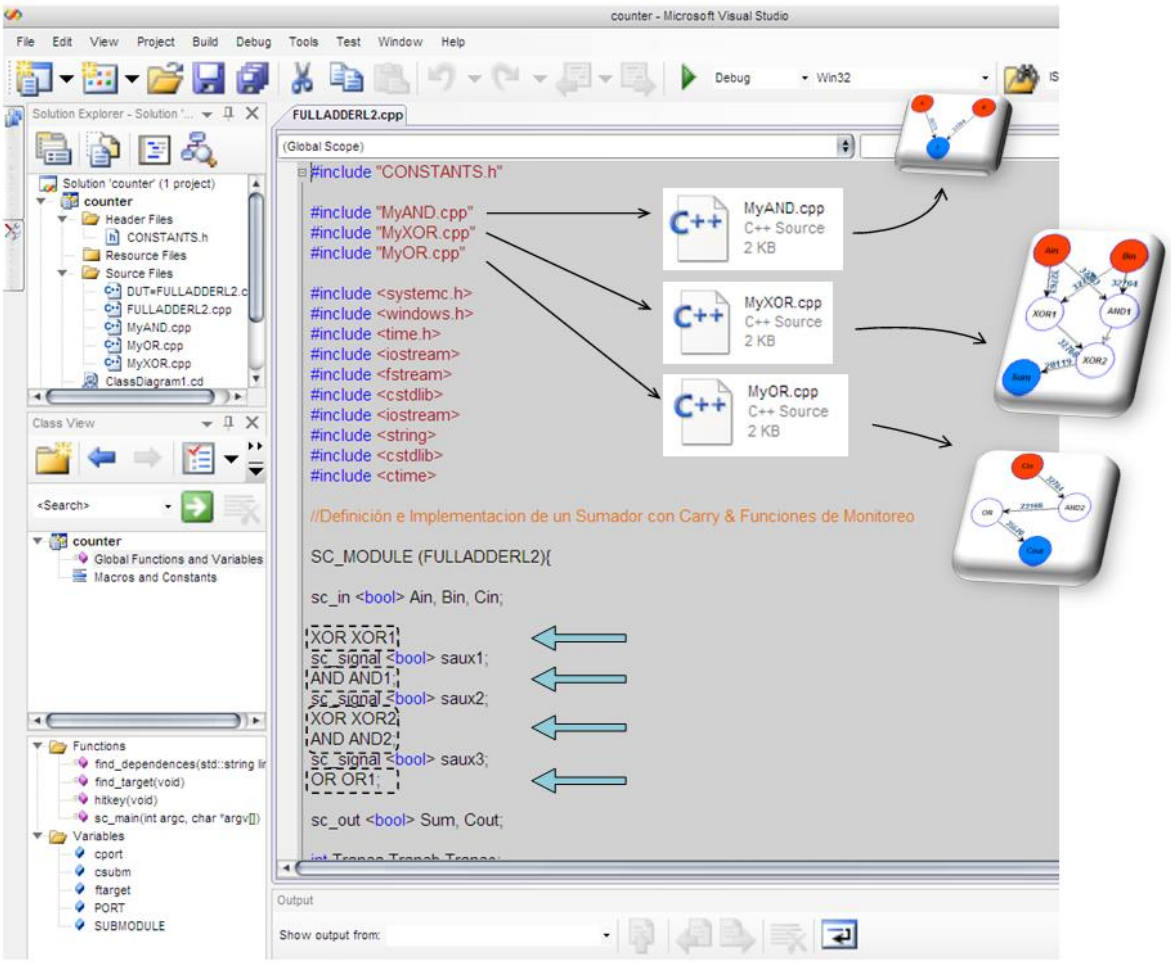


Figura 28. Un DUT que usa submodulos debe primero incluir las clases que los definen. Cada submodulo tiene un comportamiento distinto, con numeros de puertos de entrada/salida/procesamiento diferentes [como se muestra en los grafos].

La idea es crear una instancia ejecutable por cada modulo para su ejecucion en paralelo, de este modo, la etapa de particionamiento debe generar tantos nuevos archivos para ser compilados y ejecutados en paralelo como submodulos tenga el diseño.

La generación de estos archivos es bastante simple. Una vez teniendo la informacion de la clase que define a cada submódulo, se crea un nuevo archivo con el nombre del modulo instanciado en el DUT haciendo include de esa clase. En ese mismo archivo se crean las colas de entrada y de salida del modulo que ahora se ejecutará como un proceso independiente, al mismo tiempo de implementar el mecanismo de lectura y escritura de datos en las colas que usa el algoritmo de sincronizacion.

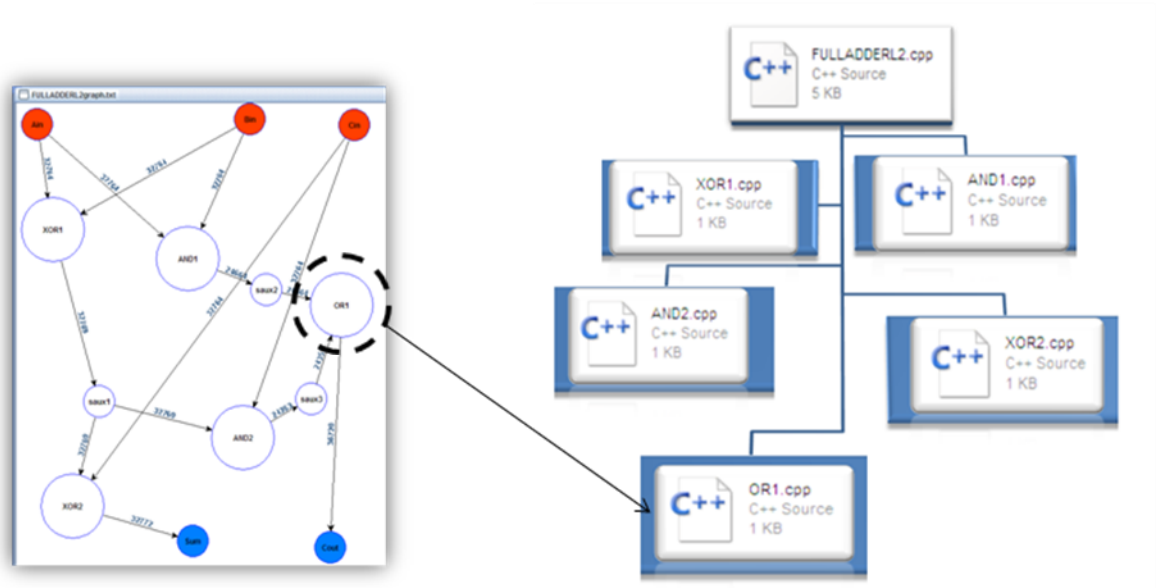


Figura 29. En el ejemplo del sumador, al submodulo OR1 le es creado un archivo OR1.cpp que contendrá la funcionalidad del mismo mas las colas de entrada y salida para comununicarse con otros procesos. Esto pasa para cada submódulo dentro del DUT.

En cuanto a los puertos de entrada del proceso que se decribe, todos seran alimentados por la cola de entrada y los puertos de salida a su vez cargaran la cola de salida con los datos procesados. La funcionalidad del proceso nunca se altera ya que el archivo clase en donde se describe el comportamiento queda encapsulada, solo se usa para instanciar un modulo de ese tipo por separado.

POST-PARTICIONAMIENTO

Después de esta etapa, un Script se encargará de hacer una compilación de los archivos en C++ usando la biblioteca de SystemC para todas y cada una de las particiones. Esto generará varios archivos ejecutables –el mismo número de particiones- teniendo así listos los elementos que serán distribuidos por el motor de ejecución paralela.

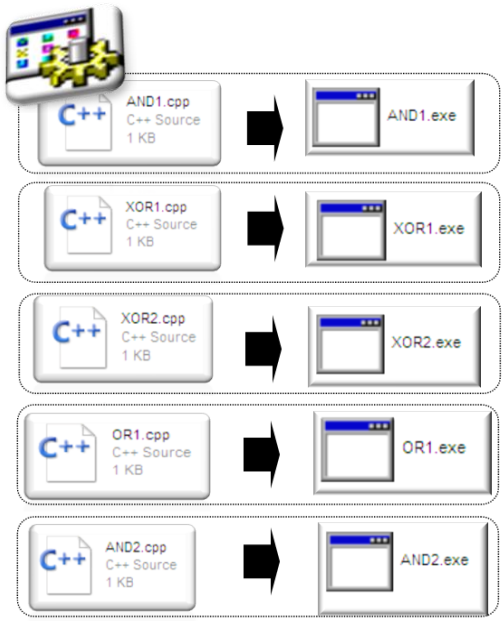


Figura 30. Por medio de un script de compilación, se generaran los ejecutables de los submodulos instanciados dentro del DUT.

En cuanto al tema de sincronizacion entre procesos, algo que se debe resolver antes de incluir las colas en cada proceso es el orden en que se van a recibir y escribir los datos. Respecto a la recepción, un proceso puede recibir datos en la cola de entrada de varios procesos mas, asi que hay que definir bien el orden en el que los puertos de entrada del modulo lean de la cola de entrada.

Por otro lado, en la cola de salida también se debe llevar un registro del orden en el que los puertos de salida escribiendo a la hora en que los datos sean transferidos a otros procesos, de tal forma que se distribuyan de acuerdo a lo que se espera en la cola de entrada de tales procesos.

La información para mantener el orden del que hablamos surge del port binding del DUT, donde según se van conectando los puertos de entrada y salida de los submodulos, es el orden en el que se leerán o escribirán en sus respectivas colas.

En el caso del binding del sumador, al construir el archivo que implementara el modulo XOR1, se debe especificar que en la cola de entrada para ese proceso primero vendrá el dato para que sea leído por el puerto de entrada Axor y el segundo dato para Bxor. En el caso de Fxor, este único puerto de salida en el submódulo será escrito en la primera línea de la cola de salida, misma que debe ser mapeada a la segunda línea en la cola de entrada del submódulo XOR2.

```
XOR1.Axor(Ain);
XOR1.Bxor(Bin);
XOR1.Fxor(saux1);

AND1.Aand(Ain);
AND1.Band(Bin);
AND1.Fand(saux2);

XOR2.Axor(Cin);
XOR2.Bxor(saux1);
XOR2.Fxor(Sum);

AND2.Aand(Cin);
AND2.Band(saux1);
AND2.Fand(saux3);

OR1.Aor(saux2);
OR1.Bor(saux3);
OR1.For(Cout);
```

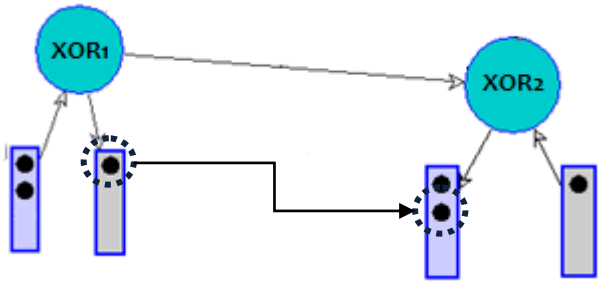


Figura 31. Port Binding de submodulos en el Sumador. El orden de lectura/escritura en las colas de cada proceso está dado por el mismo orden en que se conectan mediante el port binding.

Con el fin de que la estación maestro encargada de realizar la distribucion de procesos resuelva la dependencia de datos entre ellos previo al inicio de la simulación, se va a crear un archivo de configuración con el formato que se describe a continuación.

Con el fin de poder correr varias simulaciones del mismo DUT en la red de cómputo paralelo, se debe asignar un identificador para cada conjunto de procesos que serán lanzados. En el primer renglón del archivo de configuración se escribe este identificador. Dicho PID (Identificador Único de Proceso) va a delimitar el conjunto de procesos que pertenecerán a una corrida de la simulación, evitando así que dos procesos con el mismo nombre lean o escriban datos dentro del flujo de simulación del que no fueron lanzados.

A partir del segundo renglón, se indica información perteneciente a cada uno de los procesos separada por comas. El primer campo pertenece al nombre del proceso, el segundo tiene que ver con la carga computacional estimada para tal proceso. Del tercer elemento en adelante se indican las dependencias con otros procesos.

El listado de estas dependencias comienza con el nombre del proceso que necesitará los resultados que éste produzca, a continuación el número de transacciones hechas hacia este proceso (dato obtenido de la pre-simulación), seguido por la posición de la cola de salida que tendrá asignada en este proceso y por último la posición en la cola de entrada del proceso receptor.

Esta secuencia se repite para todos y cada uno de los procesos que estén leyendo de la salida del proceso en cuestión. Las demás líneas hacen lo propio para los demás procesos y sus dependencias, dando como resultado un archivo como el que se muestra en la figura.

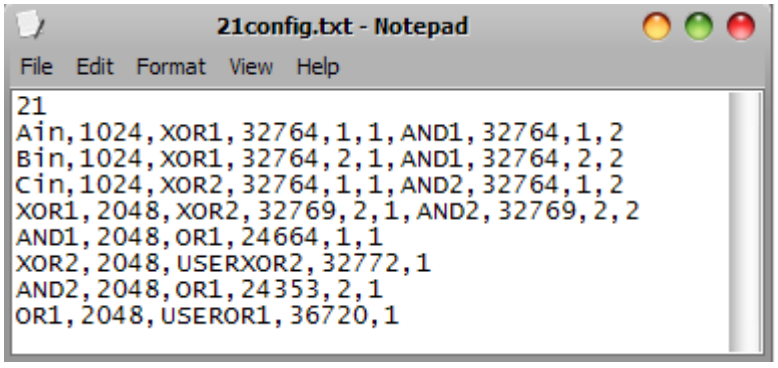


Figura 32. Archivo de Configuración generado al lanzar la simulación del Sumador con un PID de 21.

EXPERIMENTACIÓN

Con el fin de exponer el flujo completo de la metodología, usaremos un circuito decodificador hexadecimal a display de siete segmentos como caso de prueba. Asumiendo que el lector está familiarizado con este tipo de circuito, el DUT propuesto es un buen caso de estudio ya que a cada segmento del display le puede ser asociado un modulo como driver del segmento. Los resultados del tiempo de ejecución al paralelizar la ejecución de cada uno de estos módulos nos darán una medida de la efectividad de este proyecto.

Para diseñar el circuito que convertirá una señal de 4 bits representando un número hexadecimal en un numero/carácter en el display, se usó una perspectiva bottom-up en SystemC. Primero se codificaron los módulos que controlaran cada 'LED' (segmento) del display, cada uno de ellos cuenta con cuatro entradas y una salida de tipo binario. El comportamiento de cada uno de ellos está dado por la expresión lógica resultante del mapa de karnaugh para el segmento en cuestión.

A_SEGMENT.h

```
(Global Scope)
#ifndef A_SEGMENT_H
#define A_SEGMENT_H

#include "systemc.h"

class A_SEGMENT : sc_module {

public:

sc_in <bool> Ab0, Ab1, Ab2, Ab3;
sc_out <bool> Aout;

SC_CTOR(A_SEGMENT);

void a_beh();
};
```

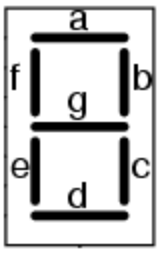
A_SEGMENT.cpp

```
A_SEGMENT
a_beh()
#include "A_SEGMENT.h"

A_SEGMENT::A_SEGMENT(sc_module_name  nm):sc_module(nm){

  SC_METHOD(a_beh);
  sensitive << Ab0 << Ab1<< Ab2<< Ab3;
}

void A_SEGMENT ::a_beh(){
Aout = (!Ab3&&!Ab2&&!Ab1&&Ab0)||(!Ab3&&Ab2&&!Ab1&&!Ab0)||(Ab3&&!Ab2&&Ab1&&Ab0)||(Ab3&&Ab2&&!Ab1&&Ab0);
}
```



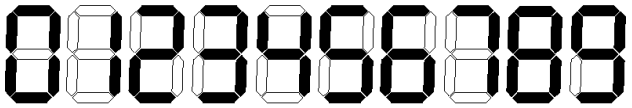


Figura 33. Implementación del primer modulo que será driver del segmento A del display.

En un segundo nivel de jerarquía del diseño, se implementó el modulo 'HtoSSD' que hace una instanciación de cada uno de los módulos que contralaran las salidas. Este modulo cuenta con 4 entradas binarias y siete salidas del mismo tipo. A cada salida se le hace el port binding del correspondiente submódulo y las entradas son comunes para todos:

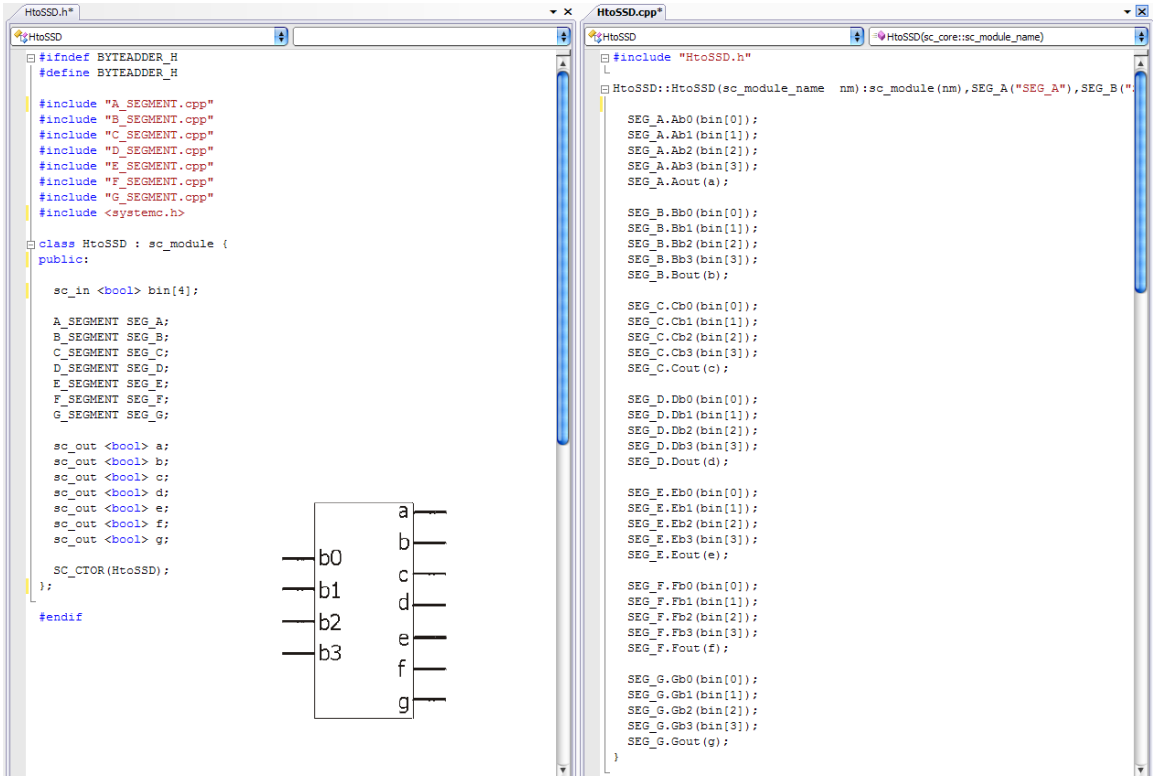


Figura 34. Código del DUT que instancia 7 submodulos para implementar el funcionamiento del display.

Una vez que el código del DUT está listo, la primera etapa de la metodología tiene como objetivo obtener una estadística del dinamismo del sistema por medio de una pre-simulación del mismo. La plataforma de verificación distribuida se desarrolló con el lenguaje java como una aplicación independiente ejecutable fuera del ambiente se desarrollo y simulación del circuito. Este programa implementa las fases de la metodología como las hemos descrito hasta ahora y controla las etapas de esta haciéndole ver al usuario el estatus o la fase de la metodología.

La aplicación puede irse ejecutando etapa por etapa a elección del usuario con algunas diferencias como obtener el grafo pero sin pesos asociados por ejemplo, o ejecutar todas las fases de la metodología sobre un diseño. En este ejemplo solo usaremos esa opción [Aplicar Metodología de Verificación Distribuida] sobre el DUT propuesto para demostrar lo que haría cada etapa por separado.

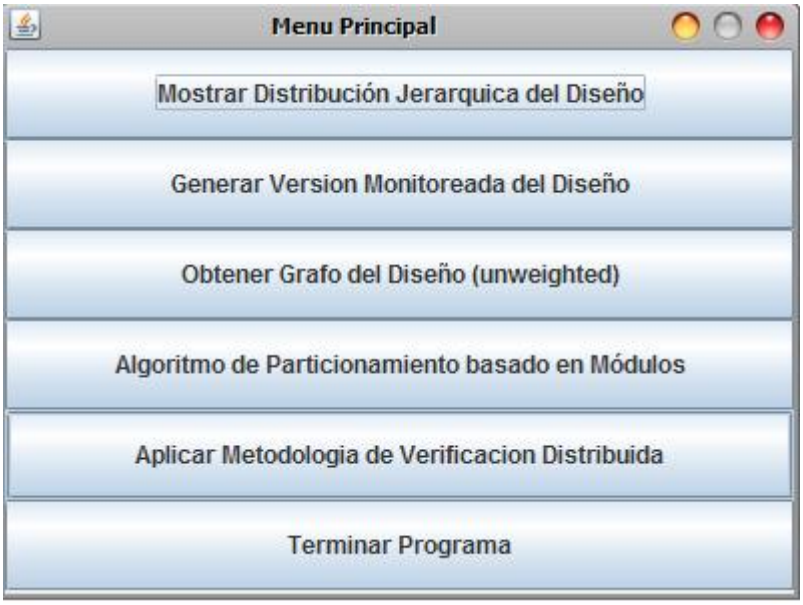


Figura 35. Menú de la aplicación que controla el flujo de la metodología.

Al seleccionar esa opción, el programa pide al usuario seleccionar el diseño y corre el parsing del mismo para armar la jerarquía y encontrar los submodulos que están siendo instanciados en el DUT. Para cada modulo, hace un barrido de puertos de entrada y de salida que son los que recibirán y enviaran datos a las colas una vez que se haya particionado el diseño.

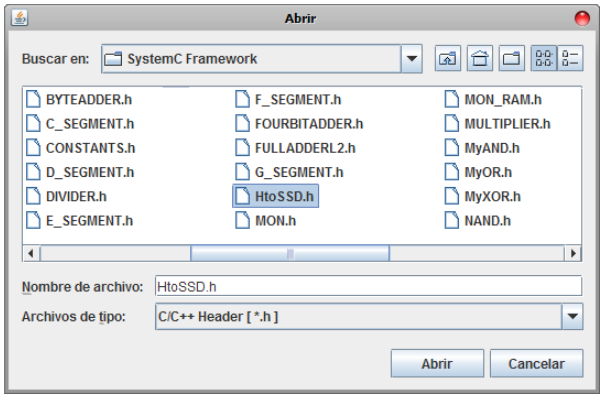


Figura 36. Browser para seleccionar el archivo cabecera [.h] que describe al DUT a ser sometido al flujo de la metodología.

Con la información jerárquica se construye un árbol con las señales del DUT y se permite al usuario seleccionar los puertos en los que se implementaran monitores para la pre-simulación. Por default, la aplicación selecciona todas las señales encontradas en el DUT.

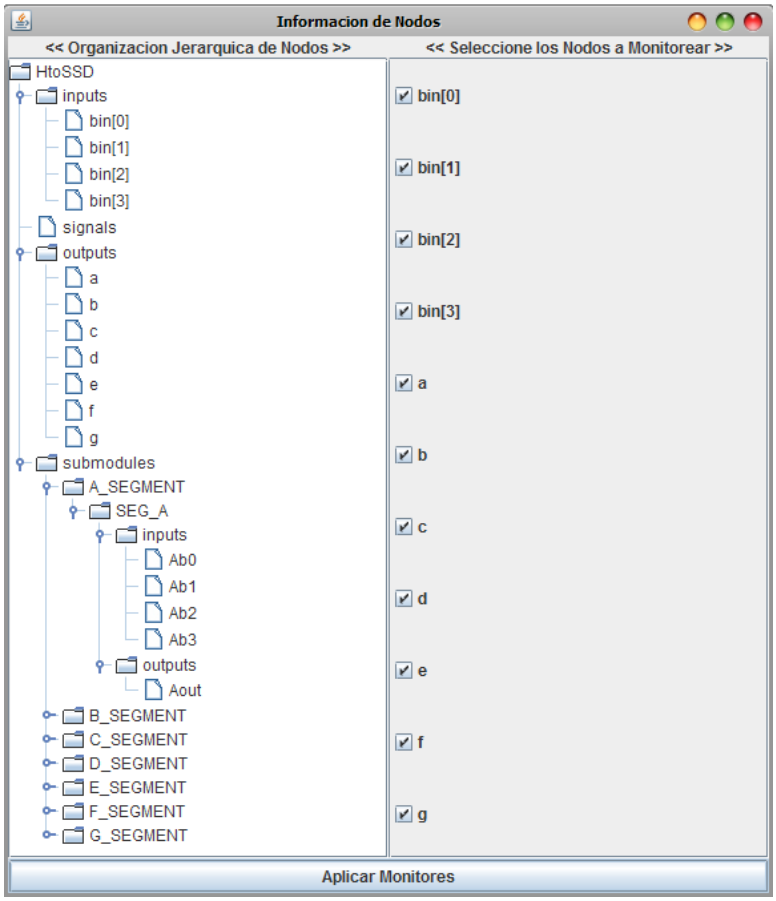


Figura 37. Selección de Monitores para el circuito del ejemplo.

Para el DUT de ejemplo, el árbol enumera un vector de cuatro entradas y siete señales de salida en el circuito. En este árbol también aparecen las instancias de los submódulos usados por el diseño, en total siete submodulos. En la figura solo se expandió el modulo SEG_A para ver que dentro de él también se identificaron cuatro señales de entrada y una de salida; este parsing de señales se hace para cada submódulo presente en el DUT.

Al dar click en aplicar monitores, una nueva versión del DUT se genera con las funciones de monitoreo añadidas. Los nuevos archivos .h y cpp del diseño reemplazan a los originales y estos últimos son respaldados dentro del directorio creado por la aplicación con el mismo nombre del DUT en el path de trabajo. Adicionalmente, es posible activar el switch dentro de la biblioteca de monitores para que se informen los cambios detectados en la pre-simulación.

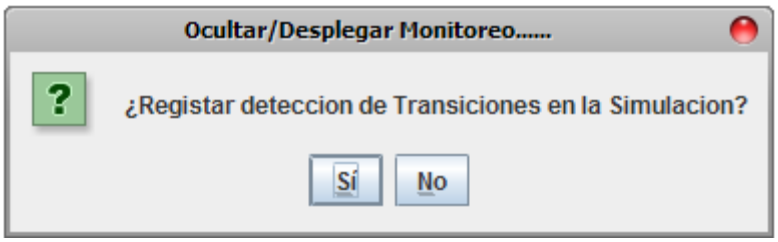


Figura 38. Opción para desplegar información sobre transiciones en las señales monitoreadas.

Una vez que los nuevos archivos se han generado, la aplicación queda en stand by esperando a que el usuario recompile el diseño y corra la pre-simulación.

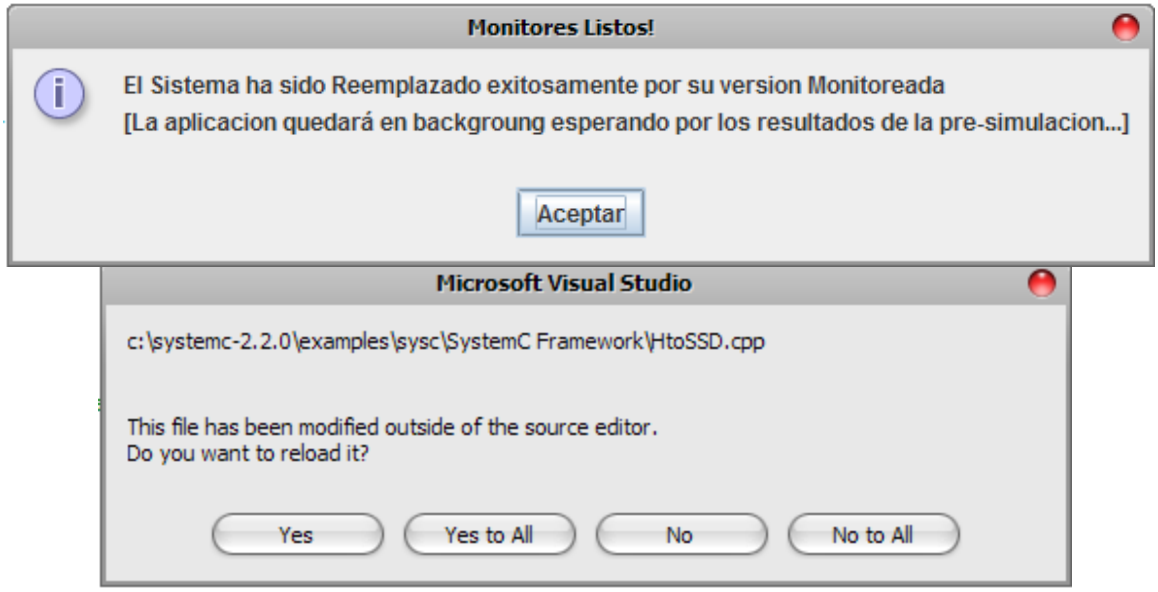


Figura 39. La aplicación queda en espera de la pre-simulación y Visual Studio detecta las nuevas versiones de los archivos [se necesita hacer rebuild del proyecto]

Después de recompilar el diseño y correr la pre-simulación, si activamos el switch de 'verbosidad' en los monitores, para cada cambio en cualquier señal donde se aplicó un monitor, se desplegara información con el tiempo, el valor anterior y el nuevo valor de la señal. Al final de la corrida, se despliegan los valores finales con que se quedó cada contador asignado a la señal y esta información es escrita en un archivo DUT_stat.txt donde DUT es el nombre del diseño. Es justo este archivo el que la aplicación queda esperando para armar el grafo con pesos del diseño.

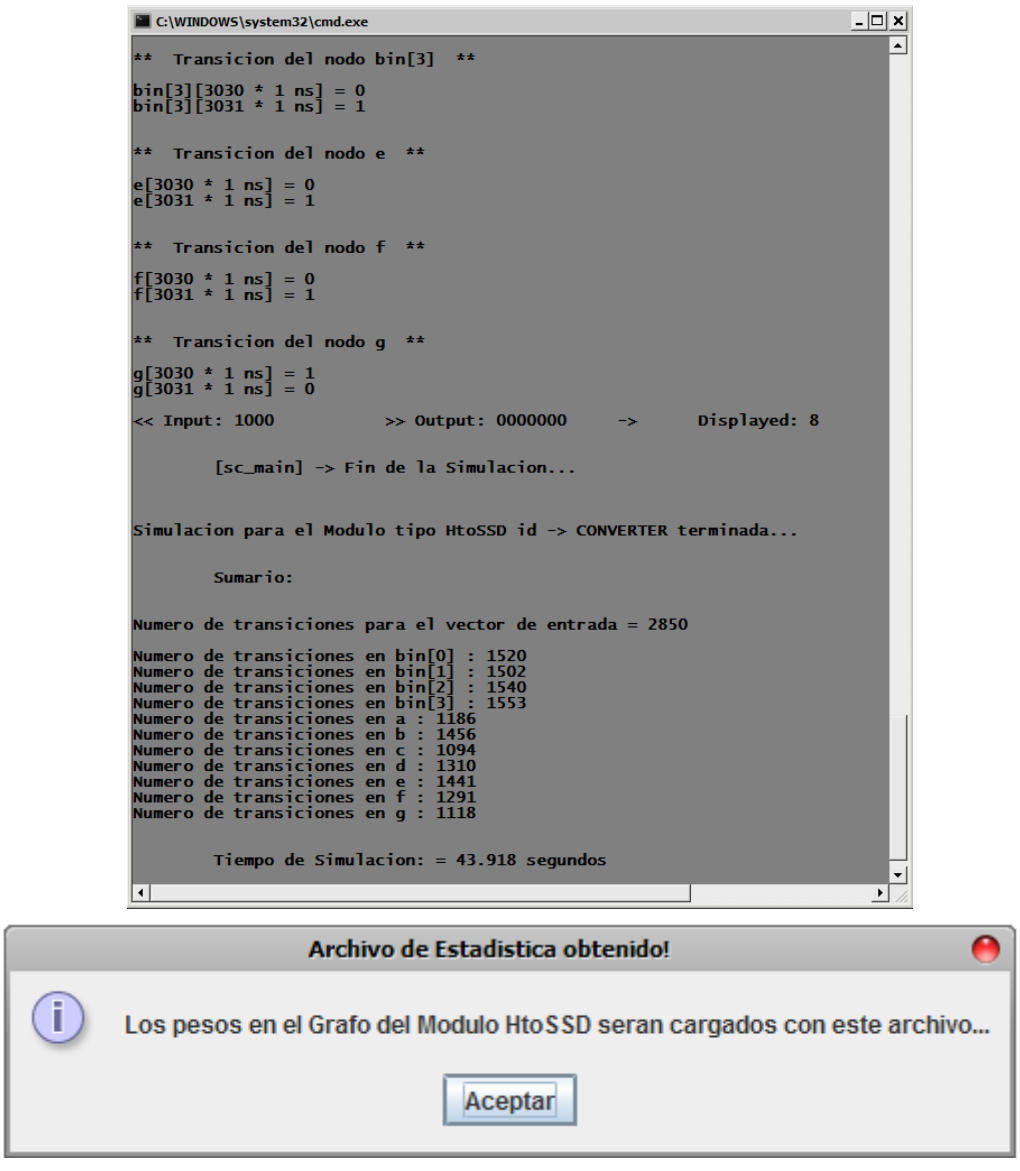


Figura 40. La estadística generada por el monitoreo en la pre-simulación es leída por la aplicación y cargada al grafo del circuito.

En la siguiente etapa de la metodología, se construye el grafo del DUT y se asocian los pesos en los arcos con la estadística obtenida, dando como resultado un 'weighted graph' del diseño, que es un archivo DUT_wgraph.txt.

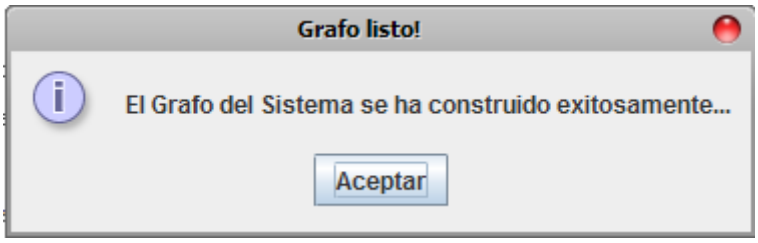


Figura 41. El archivo del grafo del circuito se crea una vez obtenida la métrica de dinamismo en el DUT.

Este archivo contiene la información del gafo con el formato apropiado para ser leído por la aplicación Vertex-Edge de la herramienta CPMP de la que hablamos en la sección de implementación del presente proyecto. La aplicación se encarga de abrir el visor de grafos por nosotros, solo tenemos que seleccionar el grafo a desplegar:

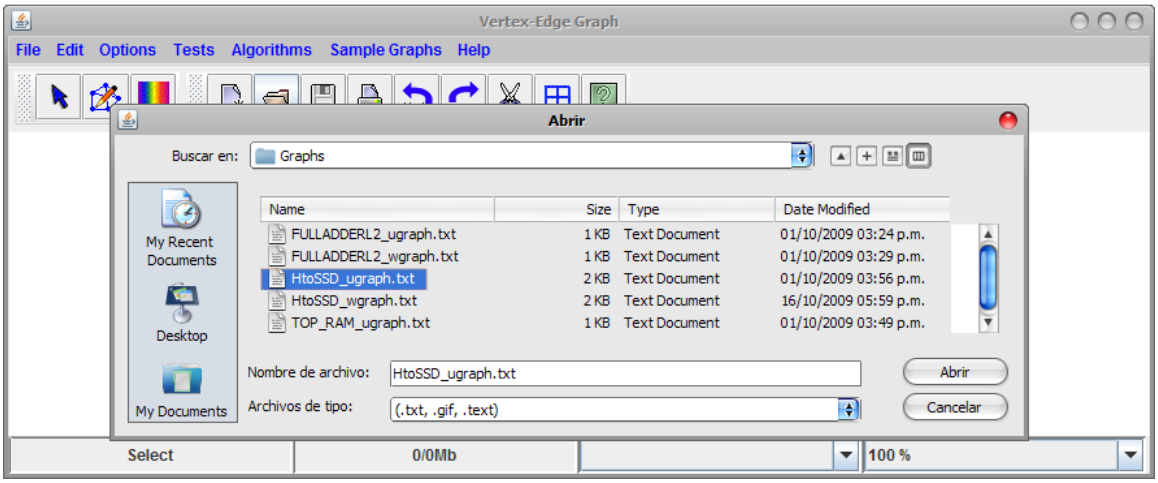


Figura 42. El archivo del grafo se crea dentro de la carpeta "Graphs" creada por la aplicación y se visualiza con la aplicación VERTEX_EDGE.

Este grafo contiene todas las señales y los submodulos previamente mostrados cuando se construyo el árbol de la jerarquía, además cada arco refleja la actividad que hubo en el nodo.

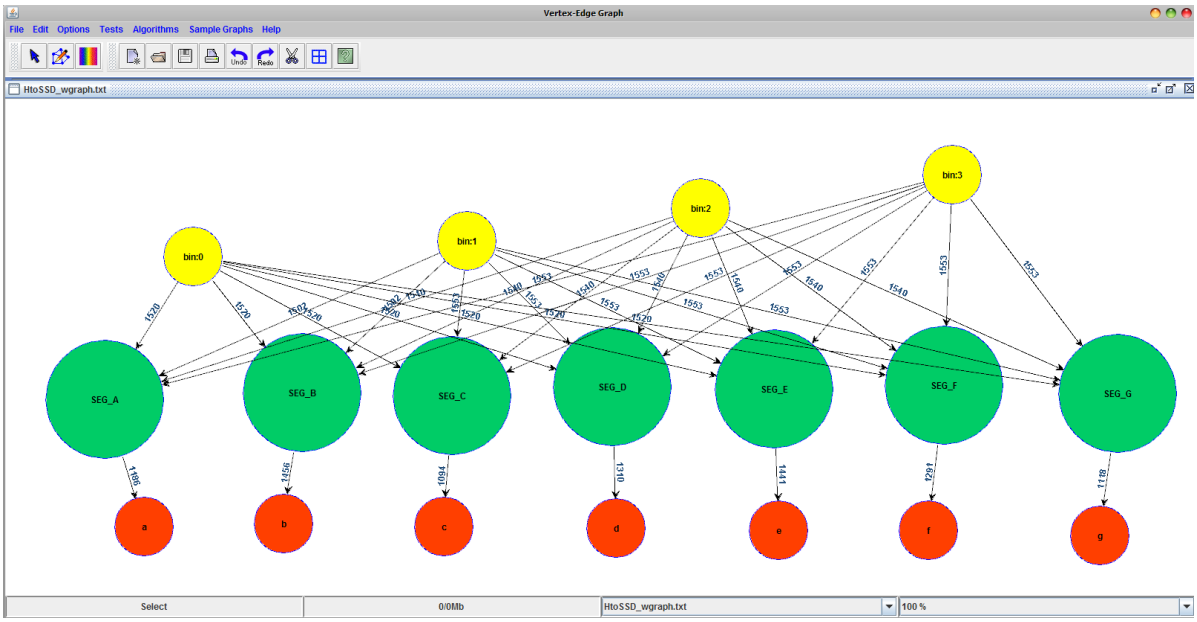


Figura 43. Grafo con pesos asociados (obtenidos mediante pre-simulación) para el DUT.

Cabe mencionar que la información de los pesos asociados a los nodos en el grafo queda implícita en el tamaño del nodo como podemos observar en los nodos grandes (submodulos = 500) en comparación con los pequeños (interfaces = 100). Los colores también tienen su papel en el grafo: amarillo para nodos de entrada, rojo para nodos de salida, verde para nodos de procesamiento que son submodulos y azul para nodos de procesamiento que son interfaces.

El archivo con el formato para particionamiento del DUT que se está estudiando es:

graph input format2.txt - Notepad

File Edit Format View Help

```
18 35 11
If,In   bin[0]: 100,    [SEG_A,1520],   [SEG_B,1520],   [SEG_C,1520],   [SEG_D,1520],   [SEG_E,1520],   [SEG_F,1520],   [SEG_G,1520]
If,In   bin[1]: 100,    [SEG_A,1502],   [SEG_B,1502],   [SEG_C,1502],   [SEG_D,1502],   [SEG_E,1502],   [SEG_F,1502],   [SEG_G,1502]
If,In   bin[2]: 100,    [SEG_A,1540],   [SEG_B,1540],   [SEG_C,1540],   [SEG_D,1540],   [SEG_E,1540],   [SEG_F,1540],   [SEG_G,1540]
If,In   bin[3]: 100,    [SEG_A,1553],   [SEG_B,1553],   [SEG_C,1553],   [SEG_D,1553],   [SEG_E,1553],   [SEG_F,1553],   [SEG_G,1553]
Md,Pr   SEG_A:  800,    [a,1186]
Md,Pr   SEG_B:  800,    [b,1456]
Md,Pr   SEG_C:  800,    [c,1094]
Md,Pr   SEG_D:  800,    [d,1310]
Md,Pr   SEG_E:  800,    [e,1441]
Md,Pr   SEG_F:  800,    [f,1291]
Md,Pr   SEG_G:  800,    [g,1118]
If,Out  a,      100,
If,Out  b,      100,
If,Out  c,      100,
If,Out  d,      100,
If,Out  e,      100,
If,Out  f,      100,
If,Out  g,      100,
```

Figura 44. Archivo para particionamiento del grafo que describe al DUT.

Del archivo mostrado, el algoritmo debe ser capaz de identificar 4 nodos de entrada, 7 de salida y 7 submodulos, además de sus correspondientes cargas de procesamiento y comunicación.

Dado que el primer algoritmo de particionamiento implementado en esta metodología está basado en crear particiones a nivel submódulo, la salida esperada por el algoritmo son 7 particiones, una que implemente el comportamiento de SEG_A, otra para SEG_B y así sucesivamente hasta SEG_F.

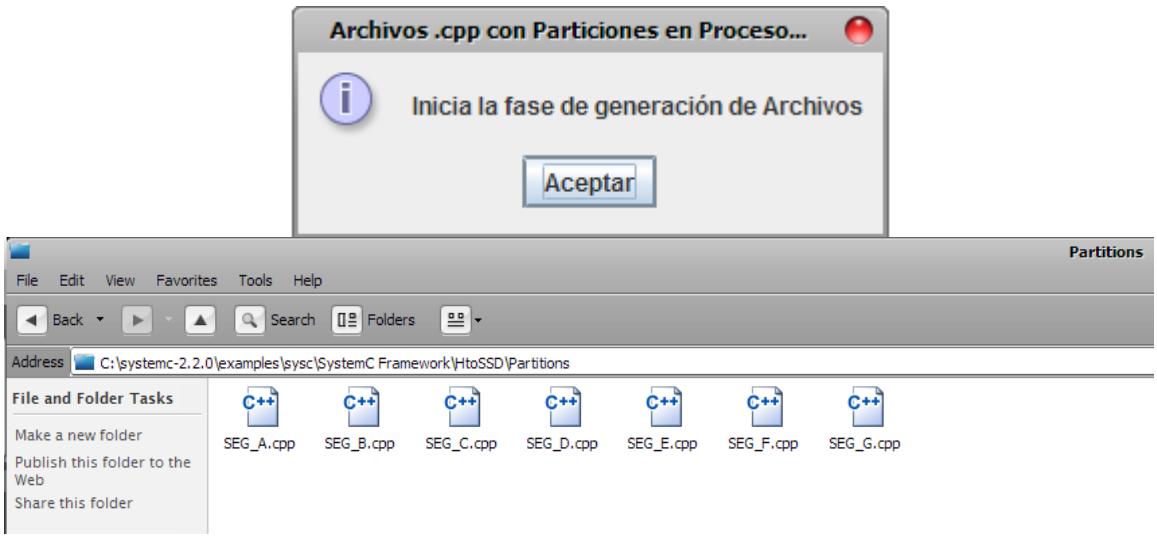


Figura 45. Generación de los archivos que instancian cada partición (en este caso cada modulo).

Para cada archivo generado se agrega el mecanismo de sincronización antes descrito. Mediante el uso de FIFOs (estructuras de datos donde el primer elemento que entra es el primer elemento que sale, también conocidas como 'colas') de entrada y de salida, la partición estará comunicándose con las demás leyendo y/o escribiendo en estas colas.

En un primer esquema de comunicación, estas colas se implementaran en archivos de lectura o escritura para la cola de entrada y cola de salida respectivamente.

Para asociar el nombre de los archivos que mantendrían la información de las FIFOs, se decidió usar el siguiente formato:

SimID + NOMBRE_DE_LA_PARTICIÓN + IN/OUT.txt

Donde:

SimID es un número entero que identifica la simulación, pudiendo así correr varias simulaciones en paralelo de un mismo DUT pero con diferente ID.

NOMBRE_DE_LA_PARTICIÓN en este caso corresponde al nombre de un submódulo instanciado en el DUT, ya que nuestro algoritmo de particionamiento usa esta granularidad. Sin embargo, para posteriores algoritmos en donde varios submodulos pueden ser integrados, el mismo algoritmo puede asignar nombres a las particiones y este parámetro es el que se usara para los archivos que controlan las FIFOs.

Por último, IN indica que el archivo está manejando la FIFO con los datos para entrada de la partición y OUT los datos que genera esa partición.

Así, por ejemplo, para la partición del segmento 'a' generada por ser un submódulo del Decodificador, y para una simulación con un ID = 24, los archivos de las FIFO serian:

24SEG_AIN.txt <- para la FIFO de entrada y

24SEG_AOUT -> para la FIFO de salida.

Información adicional sobre el modo de ejecución de cada partición es añadida en el encabezado de cada archivo, por ejemplo:

```
// CMD LINE $>SEG_A.exe [Simulation ID] [Polling Time in Seconds]
// Ex: $>SEG_A.exe 14 1
```

Figura 46. Los archivos que contienen el código para cada partición se generan usando el algoritmo de particionamiento.

De este modo, cada partición (archivo .cpp) generada contiene código en C++ que crea estos dos archivos, hace un polling cada cierto tiempo de la FIFO de entrada para saber si hay nuevos datos esperando a ser procesados y una vez usados, limpia la FIFO de entrada (restaura el archivo IN.txt) y escribe los datos en la FIFO de salida (concatena en el archivo OUT.txt).

La siguiente y última fase de la metodología está a cargo de un script que hará el trabajo de compilar y hacer el linking de cada uno de los archivos generados para crear el ejecutable correspondiente. Este archivo de procesamiento por lotes lleva el nombre del DUT y la extensión .bat (HtoSSD.bat en este caso).

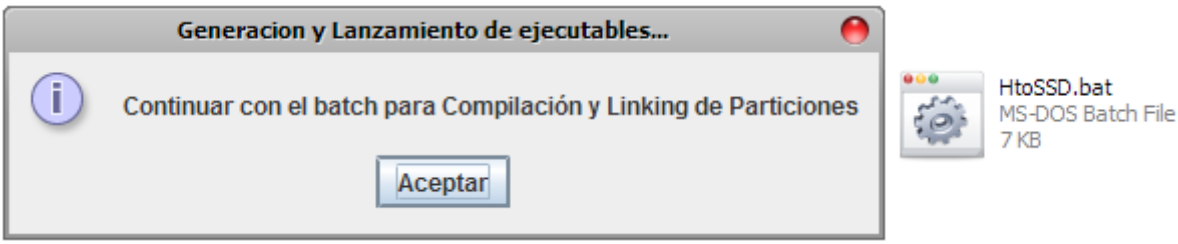


Figura 47. Un script compila y hace linking de todos y cada uno de los archivos correspondientes a las particiones y genera los ejecutables.

Este archivo de procesamiento por lotes da la posibilidad de enviar el pool de procesos a ejecución, aunque el verdadero propósito es que se asignen a estaciones de trabajo distribuidas. Si se elije continuar con la ejecución del script, los archivos .exe de las particiones serán ejecutados pero estarán esperando por datos a la entrada.

El formato para cada entrada que pudiera tener una partición es una línea de texto comenzando con el carácter '$' y finalizando con '#', separando con comas cada dato, así por ejemplo el ejecutable de la partición SEG_A con 4 entradas y 1 salidas crea sus archivos:

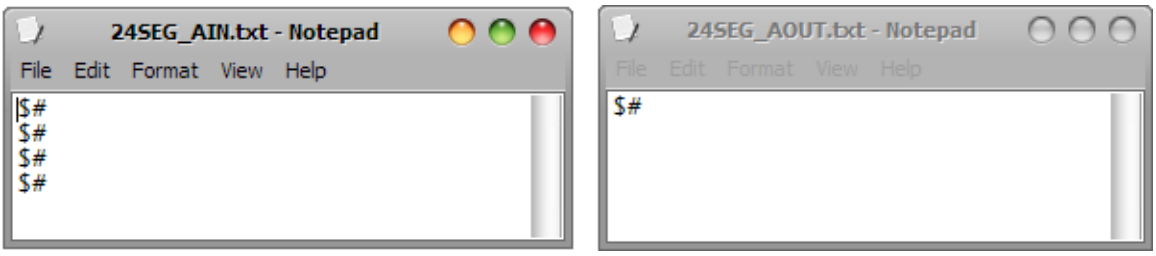


Figura 48. Archivos que controlan las FIFOs de la partición SEG_A

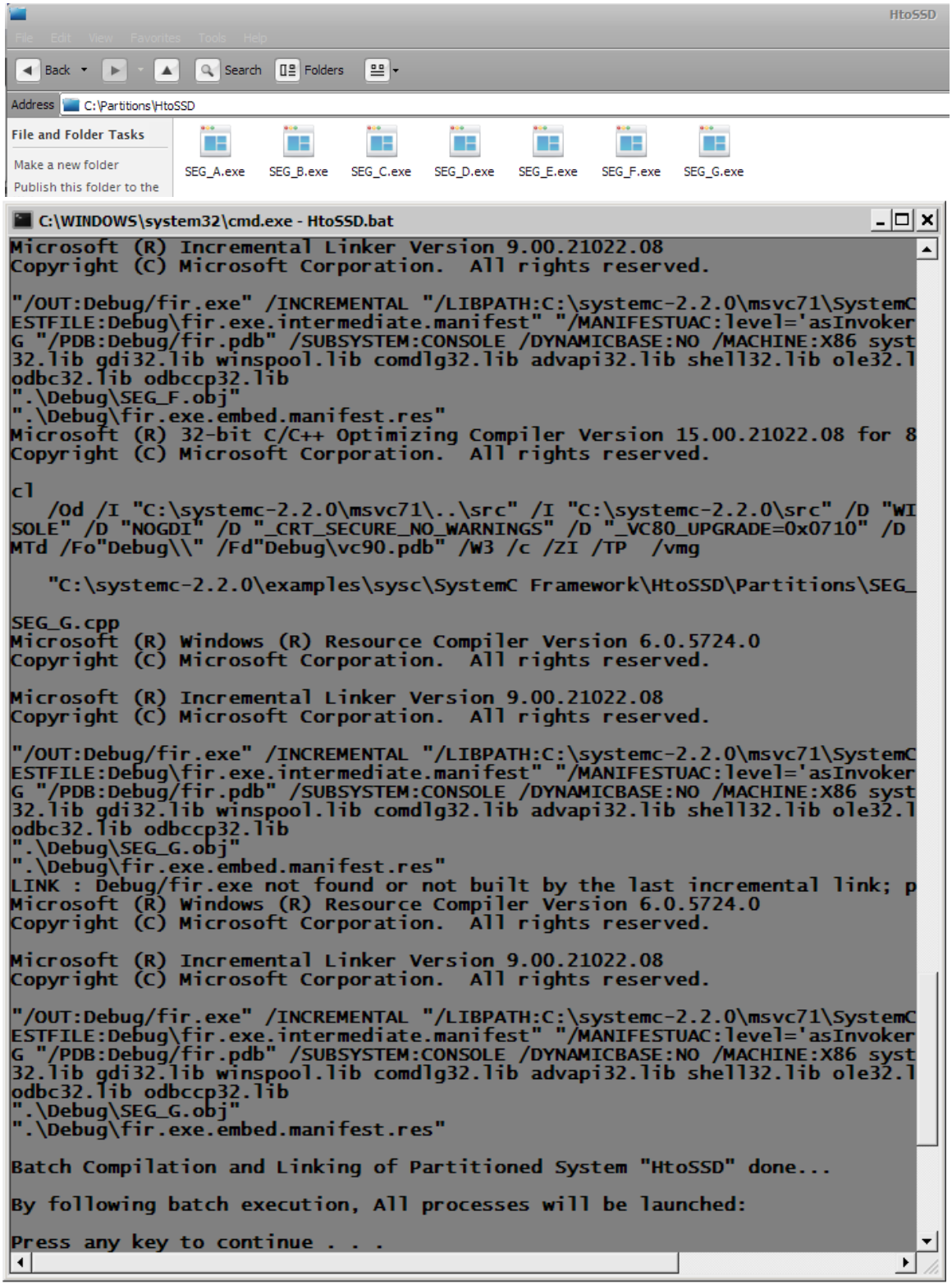
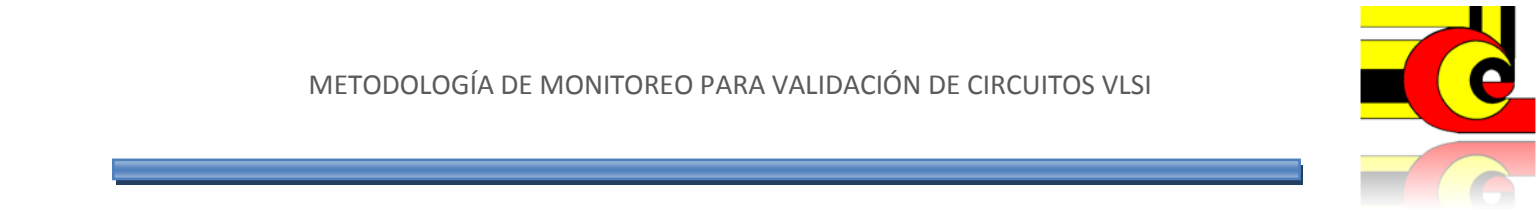


Figura 49. Ejecutables generados por el script, el cual como último paso puede iniciar la ejecución de todas las particiones.



En el DUT que se está usando en este experimento se eligió un ID para la simulación igual a 24 y un tiempo de polling de 2 segundos, mismo que cada partición espera para consultar su FIFO de entrada por nuevos datos y procesarlos en caso de encontrarlos:

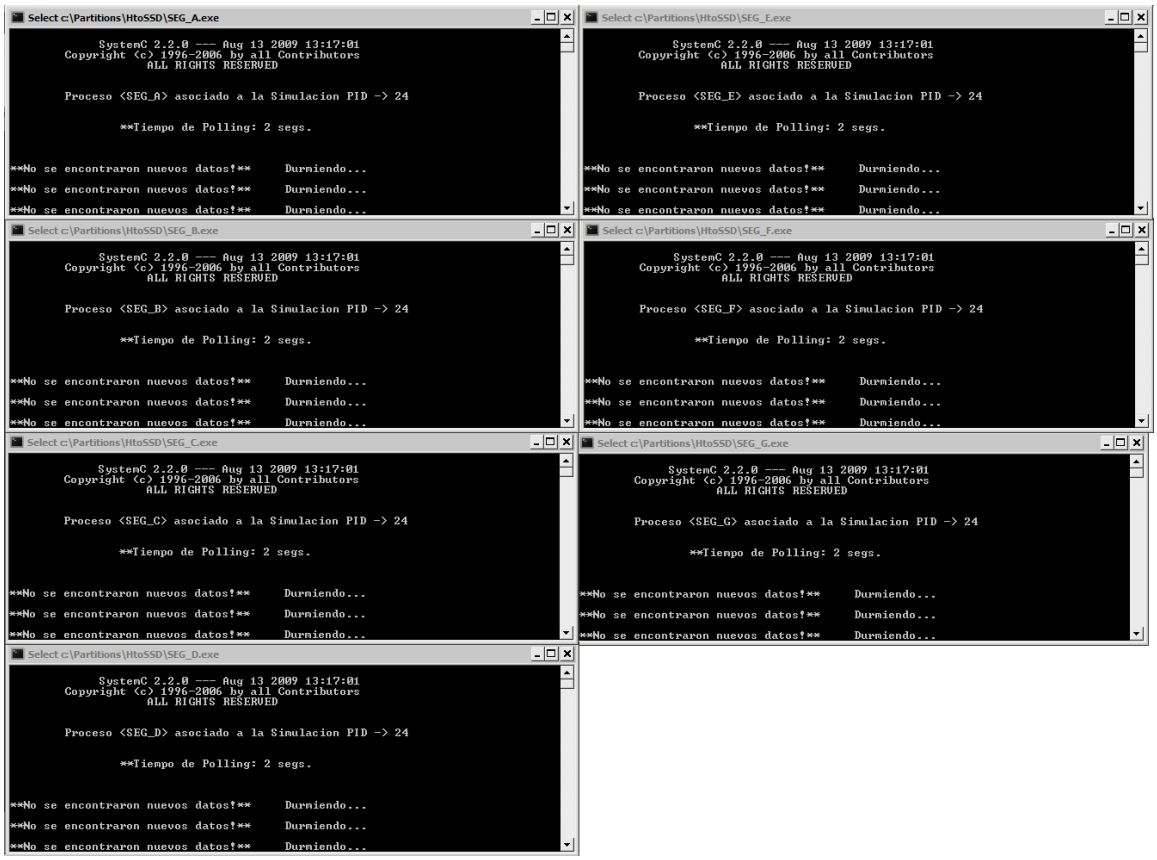


Figura 50. Pool de ejecución de las particiones como procesos independientes.

Una vez lanzada la ejecución de las particiones, la aplicación que controla el flujo de la Metodología termina y se finaliza el programa.

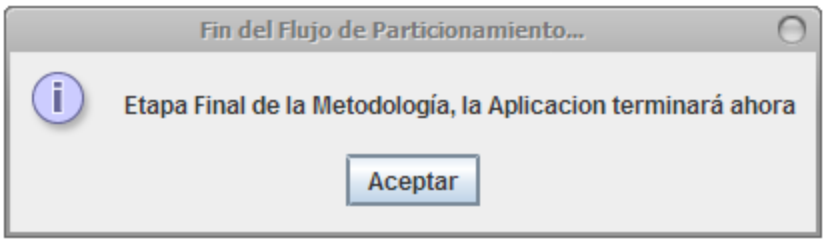


Figura 51. La ejecución de los procesos creados para cada partición es la fase final de la aplicación.

Como comprobación de que las particiones procesan correctamente los datos a la entrada, escribiremos valores en el proceso que corre la partición con el segmento A del DUT. Este dato es un vector de cuatro valores (los valores binarios correspondientes al número hexadecimal).

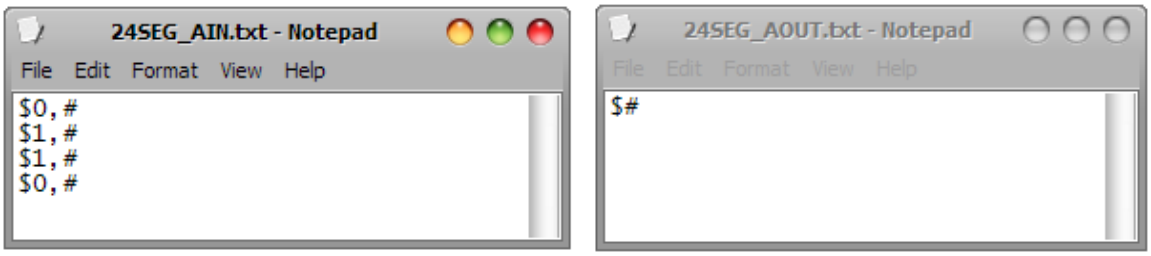


Figura 52. Escritura en la FIFO de entrada para el proceso de la partición 'SEG_A'

Al escribir y guardar esta información en el archivo que controla la cola de entrada del proceso, la siguiente vez que se hace el polling preguntando por nuevos datos, el proceso reconoce estas nuevas entradas y la usa para generar el dato de salida.

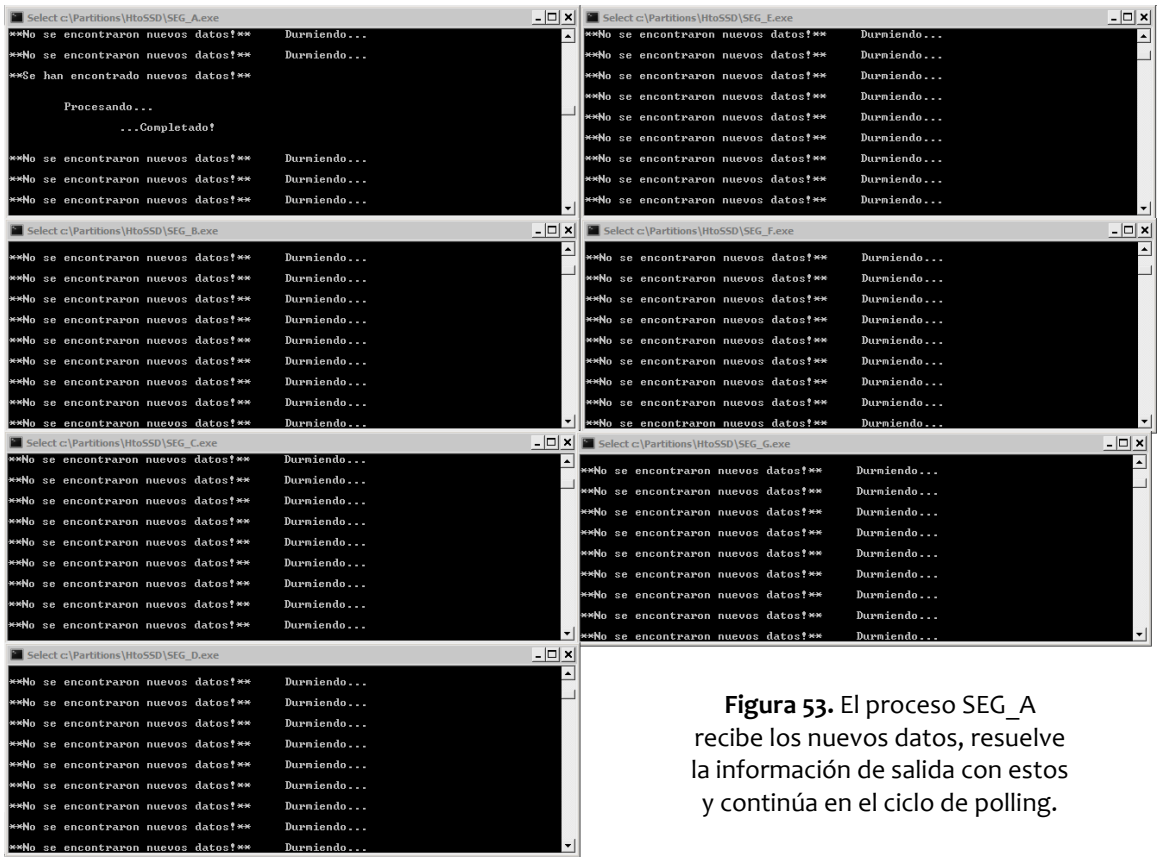


Figura 53. El proceso SEG_A recibe los nuevos datos, resuelve la información de salida con estos y continúa en el ciclo de polling.

Para asegurar el comportamiento bloqueante del algoritmo de sincronización, una vez que los datos a la entrada han sido usados, se limpia la FIFO y se restablece a su estado original. Por otro lado, los datos de salida generados se encolan en la FIFO de salida. De este modo, el proceso ve su FIFO de entrada limpia la siguiente vez que hace polling y así continua hasta que otro proceso le escriba datos de nuevo.

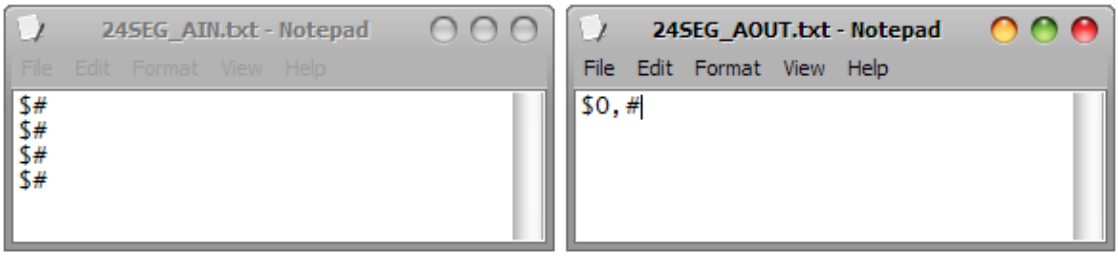


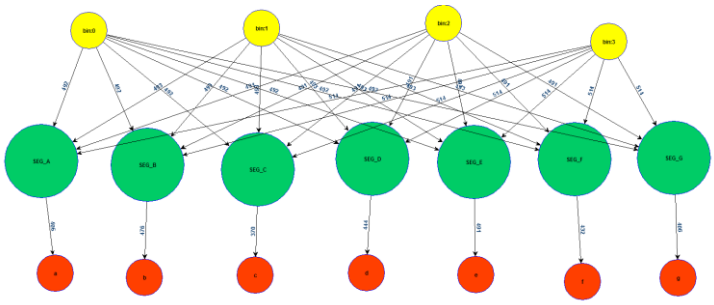
Figura 54. Una vez procesados los datos, la FIFO de entrada se limpia y los datos resultantes se escriben al final de la FIFO de salida.

RESULTADOS

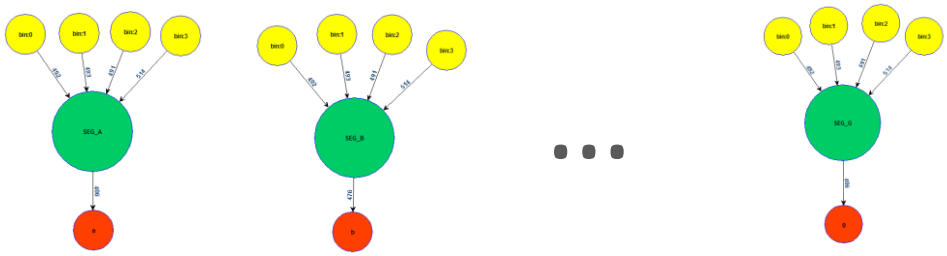
Para analizar la ganancia en el rendimiento total de simulación de un circuito particionado mediante la metodología del presente proyecto (speed up, o aceleración del tiempo de simulación), se hizo una estimación del tiempo de simulación de cada proceso en particular y se comparó contra el tiempo total que en un principio le tomaba al circuito procesar su testbench.

La partición tomada como referencia para hacer la comparativa fue la que mayor carga de trabajo y comunicaciones presentó durante la pre-simulación, en contraste con el tiempo requerido por el circuito sin particionar. *Este esquema de evaluación está asumiendo una mínima latencia de comunicación entre estaciones de trabajo, medida que en buena parte depende de la red implementada en el sistema distribuido.

El caso del DUT siendo el decodificador a siete segmentos es el mejor ejemplo de aceleración en la simulación, ya que las particiones resultantes dieron un cutsize de 0 (considerando solo cortes de canales de comunicación entre particiones, no entre nodos de entrada o salida), pues todos los submodulos son independientes entre sí.



Partición 1 (“SEG_A”) Partición 2 (“SEG_B”) Partición 7 (“SEG_G”)



Para este DUT, la partición con mayor actividad y mayor carga de trabajo (aunque en realidad todos los submodulos en este caso tienen una carga de trabajo equivalente por procesar) corresponde al proceso SEG_E, mismo que se usó para hacer la comparación en el caso de que la red donde se distribuyen los procesos sea homogénea y el equipo al cual se le haya asignado dicha partición tenga las mismas características que las demás estaciones de trabajo.

DUT	Cama de prueba (número de Vectores)	Tiempo de Simulación sin Particionamiento (en segundos)	Tiempo de Simulación usando Particionamiento (en segundos)	Speed Up
HtoSSD (Decodificador Hexadecimal a Display de 7 Segmentos)	1024000	1679.67	926.08	1.81x
HtoSSD	512000	924.69	489.79	1.89x
HtoSSD	64000	118.98	56.93	2.08x

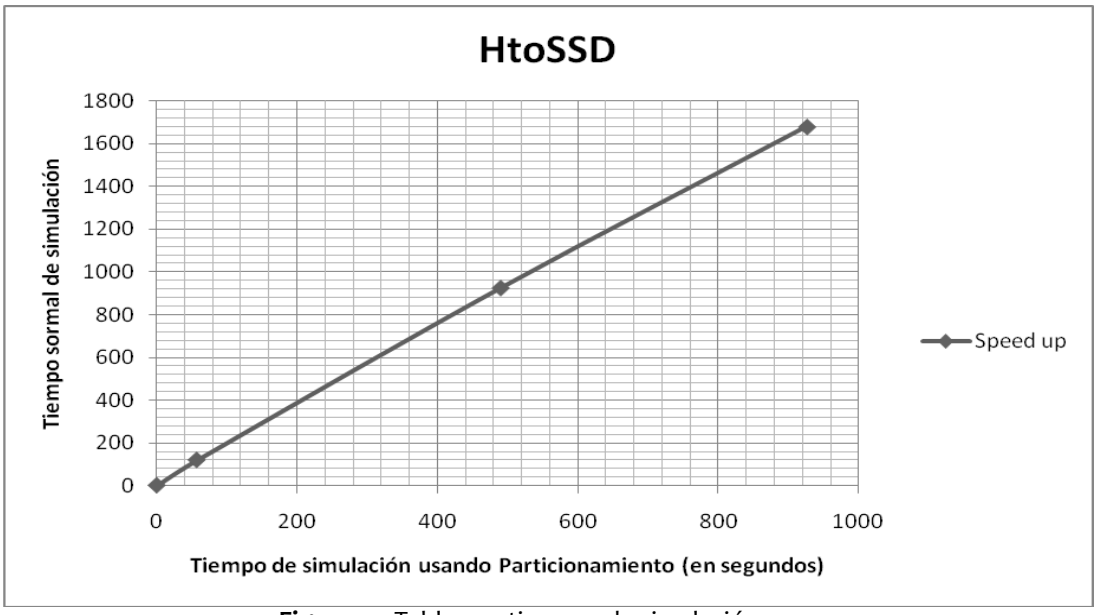


Figura 55. Tabla con tiempos de simulación y grafica que muestra que el speed up se mantiene constante para este caso de prueba.

Información adicional acerca de este test case:

Tiempo gastado en Particionamiento	Cutsize	Tiempo de Presimulación	Algoritmo de Particionamiento usado	Partición con mayor carga de Cómputo	Partición con mayor número de transiciones	Características de la Estación de trabajo
10 segundos	0	24 segundos	Particionamiento basado en Submódulos	SEG_E Carga Homogénea	SEG_E	Intel Core Duo @ 2.4 GHz 3.0 Gb de RAM

Con este resultado se demuestra que invirtiendo unos 35 segundos aproximadamente en pre-simular el circuito para obtener una métrica del comportamiento interno de este y particionarlo usando esta información -es decir un 2% del tiempo de simulación total aproximadamente - se puede simular el mismo diseño con la misma cantidad de vectores de prueba en la mitad de tiempo (speed up de 2x), manteniendo esta medida relativamente constante al aumentar el contenido del vector de pruebas.

El segundo caso de prueba es un sumador de 1 Byte particionado hasta el nivel de jerarquía más bajo, donde los submodulos representan comportamientos a nivel de compuerta lógica.

DUT	**Cama de prueba (número de Vectores)**	**Tiempo de Simulación sin Particionamiento (en segundos)**	**Tiempo de Simulación usando Particionamiento (en segundos)**	**Speed Up**
BYTE_ADDER (Sumador de un Byte con carry)	512000	550.59	393.27	1.40x
BYTE_ADDER	64000	111.29	78.37	1.42x
BYTE_ADDER	32000	66.63	43.54	1.53x
BYTE_ADDER	16000	35.02	20.68	1.69x

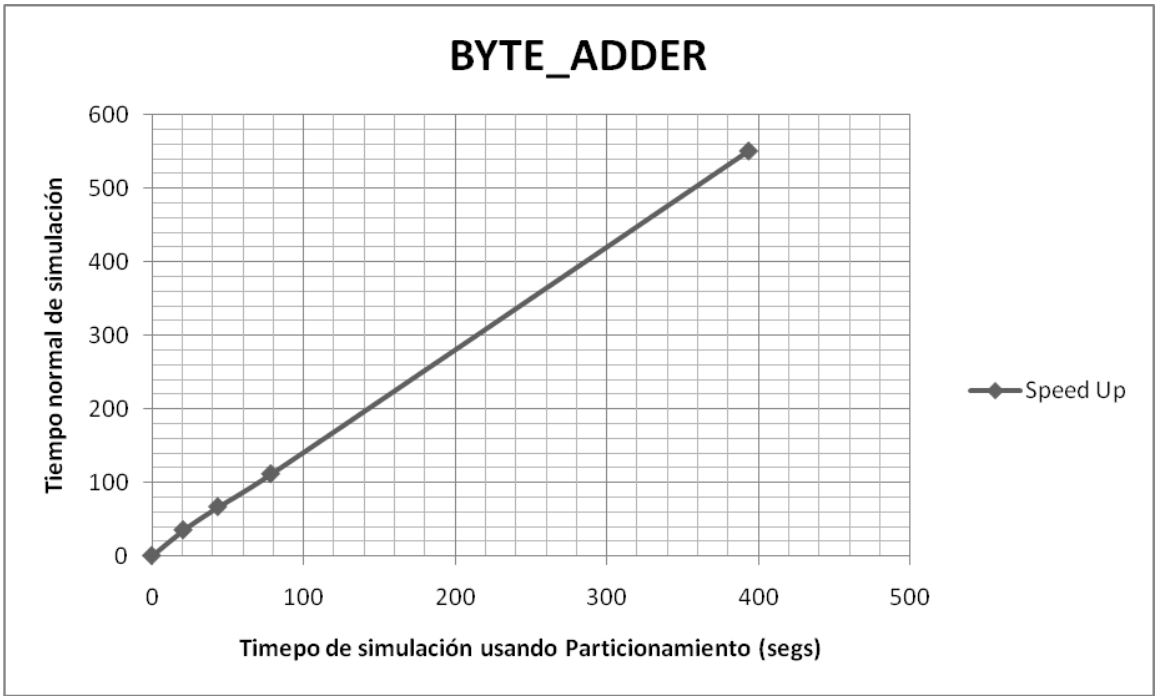


Figura 56. Para el caso del sumador, el tiempo ganado no fue del doble (1.5 aproximadamente) pero el speed up también se mantuvo constante.

Información complementaria:

Tiempo gastado en Particionamiento	Cutsize	Tiempo de Presimulación	Algoritmo de Particionamiento usado	Partición con mayor carga de Cómputo	Partición con mayor número de transiciones	Numero de Particiones
16 segundos	4	31 segundos	Particionamiento basado en Submódulos	XOR2 Carga Homogénea	XOR2	40

Los resultados y las graficas de ambos casos de prueba confirman una disminución del tiempo de simulación al particionar el sistema en procesos que se puedan ejecutar independiente y simultáneamente.

CONCLUSIONES Y TRABAJO FUTURO

El aporte de este proyecto de Tesis es un indicador de que la simulación paralela y distribuida es ciertamente una estrategia efectiva para obtener tiempos más cortos de verificación funcional de circuitos, eliminar las limitaciones de memoria impuestas por una simulación en un sistema monoprocesador y minimizar los costos inherentes a una ejecución de procesos de forma distribuida.

Se llega a la conclusión de que la aceleración de la simulación es posible si se sabe explotar la información sobre jerarquía, carga de comunicaciones y paralelismo propio del circuito a ser simulado. Parte de esta información fue obtenida mediante pre-simulación, una estrategia que es poco usada para propósitos de particionamiento.

Las principales contribuciones de esta Tesis son:

- Comprobar que uno de los más recientes estándares IEEE de lenguajes de definición de hardware, SystemC, puede ser usado para modelar circuitos integrados al igual que Verilog o VHDL, con la particularidad de poder obtener ejecutables del diseño.
- Una biblioteca de monitores para puertos y tipos de datos más usados en el lenguaje, mismo que se usaran para medir la cantidad de transiciones en las señales, wires y submodulos dentro del circuito.
- Pre-simulación con monitores asociados, misma que arrojará la métrica sobre numero de transiciones por cada señal monitoreada y tener así una idea del costo en la carga de comunicaciones ante diferente cutsize.
- Generar el grafo con pesos del circuito, de tal forma que los nodos tengan un paso asociado a la carga computacional que impone el mismo y los arcos con pesos asociados al número de transacciones hechas entre los nodos que conecta.

- Primer Algoritmo de Particionamiento, donde la granularidad es por módulo y cada partición es un wrapper de uno de estos, obteniendo un número total de particiones igual al total de submódulos instanciados en el nivel más alto (top) del circuito.

- Implementación de un mecanismo de sincronización conservativo entre los procesos generados, utilizando FIFOs de Entrada y de Salida para cada partición.

- Una aplicación en java que controla el flujo completo de la metodología para someter un diseño ante las diferentes fases del proceso de paralelización y distribución propuesto por este proyecto. La aplicación fue diseñada con la posibilidad de ejecutar cada una de estas etapas independientemente por el usuario o corriendo en background para controlarlas por si misma.

Los condiciones de aceleración de la simulación estudiadas durante el estudio del estado del arte sobre simulación paralela y distribuida fueron confirmadas por los experimentos realizados, y se concluye que con el fin de minimizar el trafico de mensajes entre las múltiples particiones resultantes del circuito, el cutsize debe ser minimizado, y que con el fin de obtener una ganancia máxima en tiempo de simulación, el algoritmo de particionamiento debe considerar un balance en las cargas de trabajo al generar las particiones.

TRABAJO FUTURO

Para tener una verdadera medida de la aceleración en la simulación, es preciso crear un agente maestro encargado de repartir los procesos en estaciones distribuidas de trabajo conectadas en red y estar administrando las colas de los procesos para enviar y recibir los datos que entre dichos procesos es necesario intercambiar.

Hasta ahora hemos trabajado en un esquema que aprovecha la información intrínseca del diseño para particionarlo en bloques donde cada sección representa un submódulo que es parte de la funcionalidad del circuito y que se puede instanciar automáticamente usando el archivo que define la clase del submódulo.

Sin embargo, en un algoritmo posterior pretendemos explotar la estadística ya obtenida en cuanto al comportamiento del circuito para hacer los cortes.

El objetivo de esta segunda alternativa de particionamiento es formar un conjunto de bloques donde cada uno estará constituido por nodos cuyo rango de actividad estén dentro de un umbral aun a definir.

De este trabajo en proceso se deriva la necesidad por robustecer la biblioteca de monitores complementándola con algunos puertos e interfaces especiales que aun no son monitorizables y poder obtener con esto la métrica de actividad interna para cualquier diseño.

Por otro lado, un mecanismo de sincronización vía TCP/IP usando sockets como colas de entrada y salida de datos en los procesos particionados, impactaría considerablemente al reducir la latencia de comunicación.

GLOSARIO

Corner Case

Escenario de verificación con valores específicos en las entradas y un estado general del diseño que es difícil de alcanzar durante el proceso de simulación.

Cutsize

Número de canales de comunicación entre procesos que el algoritmo de particionamiento interrumpe para formar las subdivisiones del circuito.

DUT (Device Under Test)

Es el diseño o prototipo a ser sometido a las pruebas correspondientes bajo un ambiente de verificación preestablecido.

Floorplaning and Placement

En el diseño de circuitos, este proceso toma en cuenta las limitantes geométricas de la tecnología usada para crear una plantilla que cumpla con dichas especificaciones y esta pueda ser estampada sobre el material de silicio y así crear la electrónica del circuito.

Latencia

Es una medida de tiempo de retraso experimentado en una red al mandar paquetes de una estación de trabajo a otra.

Nivel Top del diseño

Es el nivel de jerarquía más alto del diseño donde se mandan instanciar otras entidades (submódulos) que compondrán al diseño y le proporcionarán su funcionalidad.

Overhead

Este término hace alusión a una sobrecarga de trabajo que puede darse al crear particiones con un desequilibrio en la carga de trabajo en los procesos que las componen.

Pool de procesos

Conjunto de ejecutables que después de la etapa de particionamiento, compilación y linking de los archivos, es creado y está listo para ser distribuido sobre una red de cómputo.

Simulación

Es el medio por el cual se reproduce el comportamiento del sistema para comparar los resultados con la funcionalidad proyectada por el diseñador.

Testbench

Un ambiente de prueba que provee la infraestructura necesaria para meter una serie de estímulos a un diseño y analizar resultados. Es un medio por el cual diferentes pruebas pueden ser escritas para ejercitar y verificar características particulares de un diseño.

Time to Market

Es el periodo de tiempo que le lleva a un producto salir a la venta al mercado, este parámetro es muy importante para la industria manufacturera de circuitos integrados ya que los avances tecnológicos en fabricación y diseño van en aumento continuo creando una mayor competencia.

Verificación Funcional

Es la tarea de verificar que el diseño logico del sistema se conforma a la especificacion de lo que el producto debe cumplir, la pregunta que intenta responder este proceso (llevado a cabo por medio de simulacion) es: ¿El diseño hace lo que tiene que hacer?

Workload

Carga de trabajo computacional que un proceso dado puede requerir al ejecutarse.

Wrapper File

Es un archivo plantilla que permite la instanciación de cualquier modulo con solo incluir la clase base que lo implementa. Este archivo contiene las FIFO de lectura y escritura de cada partición y el algoritmo de sincronización entre procesos.

REFERENCIAS

[1] IEEE standard VHDL language reference manual, IEEE Std. 1076-1993, 1994.

[2] IEEE standard. 1364-2001, Verilog HDL Reference Manual, IEEE press, 2001.

[3] Tun Li, Yang Guo, Sikun Li, FuJiang Ao, GongJie Li: Parallel verilog simulation: architecture and circuit partition. ASP-DAC 2004: 644-646

[4] M. Bailey, J.V.Jr. Briner and R.D. Chamberlain. "Parallel logic simulation of VLSI systems," ACM Computing Surveys, 1994, 26(3): 255-294.

[5] P Banerjee, Parallel algorithms for VLSI computer-aided design, PTR Prentice Hall Press, 1994.

[6] Carl Tropper. Parallel Discrete-Event Simulation Applications. Journal of Parallel and Distributed Computing, 62:327–335, 2002.

[7] Herve Avril and Carl Tropper. Scalable clustered time warp and logic simulation. VLSI design, 00:1–23, 1998.

[8] M. Bailey, J. Briner, and R. Chamberlain. Parallel logic simulation of vlsi systems. ACM Computing Surveys, 26(03):255–295, Sept. 1994.

[9] Y.-S. Kwon, C.-M. Kyung: ATOMi: An Algorithm for Circuit Partitioning Into Multiple FPGAs Using Time-Multiplexed, Off-Chip, Multicasting Interconnection Architecture. IEEE Trans. VLSI Syst. 13(7): 861-864 (2005)

[10] Lijun Li, Hai Huang, and Carl Tropper. Towards distributed verilog simulation. International Journal of Simulation, Systems, Science & Technology.

[11] A. B. Kahng A. E. Caldwell and I. L. Markov. Design and implementation of the duccia-mattheyses heuristic for vlsi netlist partitioning. In Proc. Workshop on Algorithm Engineering and Experimentation (ALENEX), Baltimore, pages 177–193, Jan. 1999.

[12] Vipin Kumar George Karypis, Rajat Aggarwal and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. In ACM/IEEE Design Automation Conference, pages 526–529, 1997.

[13] The free encyclopedia WIKIPEDIA. Hardware description language. < http://en.wikipedia.org/wiki/Hardware description language.>

[14] *Electronic Design Automation For Integrated Circuits Handbook*, by Lavagno, Martin, and Scheffer, ISBN 0-8493-3096-3, a survey of the field of EDA. The above summary was derived, with permission, from Volume I, Chapter 16, *Digital Simulation*, by John Sanguinetti.

[15] IEEE Std 1666™-2005 Open SystemC Language Reference Manual at < http://standards.ieee.org/getieee/1666/download/1666-2005.pdf>

[16] Michael Pidd. An introduction to computer simulation. In *Proceedings of the 26th* conferences on winter simulation, December 1994.

[17] Bernard P. Zeigler, editor. Multifacetted Modelling and Discrete Event Simulation. Academic Press, London, 1984.

[18] Franois E. Cellier, editor. *Continuous System Modeling*. Springer-Verlag, New York, 1991.

[19] A. E. Ruehli, editor. Circuit Analysis, Simulation and Design. Elsevier Science Publishing Company, Inc., Amsterdam, North-Holland, 1986.

[20] Bernard P. Zeigler, editor. Multifacetted Modelling and Discrete Event Simulation. Academic Press, London, 1984.

[21] S. Smith, M. Mercer, and B. Underwood. An analysis of several approachesto circuit partitioning for parallel logic simulation. In Proc. Int. Conference on Computer Design, IEEE, pages 664–667, 1987.

[22] A. B. Kahng A. E. Caldwell and I. L. Markov. Design and implementation of the fiduccia-mattheyses heuristic for vlsi netlist partitioning. In Proc. Workshop on Algorithm Engineering and Experimentation (ALENEX), Baltimore, pages 177–193, Jan. 1999.

[23] R.Chamberlain and C.Henderson. Evaluating the use of pre-simulation in vlsi circuit partitioning. In PADS94, pages 139–146, 1994.

[24] Swaminathan Subramanian, Dhananjai M. Rao, and Philip A. Wilsey. Applying multilevel partitioning to parallel logic simulation. In Parallel and Distributed Computing Practices, volume 4, pages 37–59, March 2001.

[25] H. K. Kim and J. Jean. Concurrency preserving partitioning(cpp) for parallel logic simulation. In 10th Workshop on parallel and distributed simulation(PADS'95), pages 98–105, May 1996.

[26] S. Subramaninian et al. Study of a multilevel approach to partitioning for parallel logic simulation. In IPDS00, May 2000.

[27] G. Saucier, D. Brasen, and J.P. Hiol. Partitioning with cone structures. IEEE/ACM International Conference on CAD, pages 236–239, 1993.

[28] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: A survey. Integr. VLSI Journal, 19(1-2):1–81, Aug 1995.

[29] Chamberlain R. D. and Henderson C. Evaluating the use of presimulation in vlsi circuit partitioning. In Proc. 1994 Workshop on Parallel and Distributed Simulation, pages 139–146, 1994.

[30] N. Manjikian and W. M. Loucks. High Performance Parallel Logic Simulation on a Network of Workstations. In Proc. of the 7th Workshop on Parallel and Distributed Simulation, pages 76-84, 1993.

[31] M. Bailey, J.V.Jr. Briner and R.D. Chamberlain. "Parallel logic simulation of VLSI systems," ACM Computing Surveys, 1994, 26(3): 255-294.

[32] D.A. Jefferson. Virtual Tiem, ACM Transaction on Programming Languages and Systems, Vol 7 No. 3, Julio 1985.

[33] K. Chandy, J. Misra. Distributed Simulation: A study in the design and verification of distributed programs, IEEE Trans, pp 440-452.

[34] B. Groselj and C. Tropper, The Distributed Simulation of Clustered Processes, Distributed Computing, vol. 4, pp. 111-121, 1991.

[35] K. Chandy, J. Misra. Asynchronous Distributed Simulation: A Case Study in the Design and Verification of Distributed Programs, IEEE Trans, pp 156-198

[36] SystemC with Microsoft Visual Studio 2005 and 2008 [Consulta: 4-10-2007] <> http://www.dti.dk/_root/media/27325_SystemC_Getting_Started_artikel.pdf

[37] Kevin L. Kapp, Thomas C. Hartrum, and Tom S. Wailes, An Improved Cost Function for Static Partitioning for Parallel Circuit Simulations Using a Conserative Synchronization Protocol. *Parallel and Distributed Workshop*, 1995

[38] Pavlos Konas and Pen-chung Yew, Partitioning for Synchronous Parallel Simulation, *Parallel and Distributed Simulation Workshop*, 1995

[39] © 2006 S. A. Keller, Michigan State University and Core-Plus Mathematics Project, Western Michigan University <> http://www.wmich.edu/cpmp/CPMP-Tools/

[40] Proceedings of the 17th International Conference on VLSI Design (VLSID'04) 1063-9667/04 © 2004 IEEE