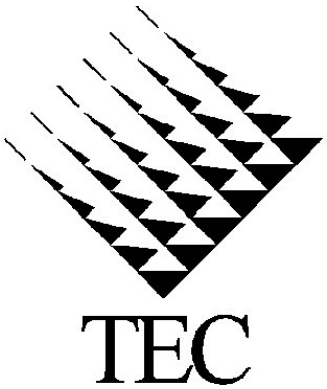
Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica



Verificación lógica de los modelos sintetizados para circuitos integrados

Informe de Proyecto de Graduación para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura

José Daniel González Rojas

Cartago, Noviembre de 2010

INSTITUTO TECNOLOGICO DE COSTA RICA

ESCUELA DE INGENIERIA ELECTRONICA

PROYECTO DE GRADUACIÓN

TRIBUNAL EVALUADOR

Proyecto de Graduación defendido ante el presente Tribunal Evaluador como requisito para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura, del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal

Ing. Roberto Pereira Arroyo

Profesor lector

Ing. Leonardo Rivas Arce

Profesor lector

Ing. Alfonso Chacón Rodríguez

Profesor asesor

Los miembros de este Tribunal dan fe de que el presente trabajo de graduación ha sido aprobado y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica

Cartago, 8 de noviembre de 2010

Declaración de autenticidad

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

En los casos en que he utilizado bibliografía, he procedido a indicar las fuentes mediante las respectivas citas bibliográficas.

En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.

8 de noviembre de 2010

José Daniel González Rojas

Cédula 1-11265-0515

RESUMEN

La clasificación de fallas es el proceso de evaluación o clasificación de una serie de pruebas de acuerdo a su eficacia en la detección de defectos de fabricación. El principal objetivo de la clasificación de fallas es medir y mejorar la calidad de la prueba en la producción.

La verificación lógica es parte del proceso de clasificación de fallas y consiste en ejecutar todas las pruebas de la colección o grupo de pruebas en el modelo lógico y comparar los resultados con otro simulador de resultados, como un simulador de VHDL. Cuando se encuentra una divergencia en la comparación es necesario aislar el error en el modelo. Una vez finalizada la verificación lógica del modelo se inicia la simulación de fallas. Cualquier retraso en la verificación lógica significa un retraso en la escritura de pruebas funcionales.

La verificación lógica es un proceso que requiere de mucho tiempo, desde horas hasta semanas, sobre todo por la complejidad de los circuitos integrados. Por lo tanto, con este proyecto se pretende acelerar el rastreo y la localización de las fallas lógicas encontradas en los modelos sintetizados de circuitos integrados.

En la solución de este proyecto fueron considerados dos algoritmos: el rastreo que realiza el seguimiento de una divergencia compuerta por compuerta hasta el origen y la búsqueda modular que aísla una divergencia en el módulo de menor jerarquía definido en el modelo sintetizado. La búsqueda modular proporciona una buena idea de dónde está el origen de la falla y requiere de menor tiempo de simulación, por lo que se decidió implementar este algoritmo como la solución para el proyecto.

Las pruebas realizadas a la solución implementada y a los algoritmos que la conforman son explicadas y analizadas en este documento. Los resultados obtenidos son satisfactorios tanto en tiempo de ejecución como en la localización de las divergencias en los modelos.

Palabras claves: modelos de fallas, clasificación de fallas, verificación lógica, divergencia, búsqueda modular, rastreo.

ABSTRACT

Fault grading is the process of evaluating or grading a series of tests according to their effectiveness in detecting manufacturing defects. The main purpose of the fault grading is to measure and improve the quality of the test during.

Logic verification is part of what is called Fault grading process and consists in the execution of all the tests from the test suite in the logic model and the comparison with the results of prior simulations, like VHDL simulator. When a mismatch is found during the comparison process, it becomes necessary to isolate the error in the model. Once finished the logic verification of the model, starts the fault simulation, which means that any delay during the logic verification entails a delay during the functional test writing.

The logic verification is a process that takes long time, from hours to weeks, mainly for the complexity of the integrated circuits under test. This project aims at accelerating the tracking and tracing of the logical faults found in synthesized models of integrated circuits.

In the solution of the project two algorithms are considered: the traceback that tracks a mismatch from gate to gate to the source and the modular search that isolates a mismatch in the smaller module defined in the synthesized model. The modular search provides a good idea of where is the origin of the failure and requires less simulation time and thus it became the chosen solution.

The tests performed on the solution implemented and the algorithms that form are explained and discussed in this paper, the results are satisfactory both in runtime as in the location of mismatches in the models.

Keywords: faults models, fault grading, logic verification, mismatch, modular search, traceback.

A mi familia y amigos.

Agradecimiento

Agradezco a Dios por ser mi fortaleza, guiarme y poner en mi camino a todas aquellas personas que han sido soporte y compañía durante los años de estudio.

A mis padres por el apoyo y las enseñanzas, a ellos les dedico este proyecto.

Al Ing. Ronald García Fernandez por compartir su conocimiento conmigo y asesorarme a lo largo de todo el proyecto.

Al Profesor Alfonso Chacón por el apoyo y recomendaciones dadas durante el proyecto.

INDICE GENERAL

INDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1: INTRODUCCIÓN

El rápido crecimiento y complejidad de los circuitos integrados en la actualidad requiere de pruebas funcionales de calidad que permitan detectar chips defectuosos en el proceso de manufactura. *Fault grading* es la metodología usada para evaluar o clasificar una serie de pruebas de acuerdo a su eficacia en la detección de posibles defectos de fabricación.

En la primera etapa de proceso de *fault grading* se sintetiza un modelo lógico a partir de las especificaciones dadas para circuito en el HDL, la segunda consiste en realizar la verificación lógica del modelo sintetizado y en la última se a simulan las pruebas funcionales sobre el modelo para determinar la cobertura de fallas.

1.1 Divergencias entre los modelos sintetizados y el HDL

El modelo sintetizado del circuito integrado es una representación a nivel de compuertas lógicas y registros de lo detallado en el HDL. Las pruebas funcionales diseñadas previas a la etapa de ensamblaje y prueba, son simuladas sobre el modelo sintetizado para observar el comportamiento que tendrá el circuito una vez fabricado.

El proceso de síntesis del modelo al ser realizado por una herramienta de software no se encuentra libre de errores, por lo que suele encontrarse diferencias entre el comportamiento descrito en el HDL y el modelo sintetizado.

La verificación lógica del modelo sintetizado permite identificar y corregir las divergencias encontradas entre el modelo y el HDL. Sin la verificación lógica del modelo, no se puede continuar con el proceso de *Fault Grading* y se generan retrasos en el diseño de las pruebas funcionales.

1.2 Síntesis del problema

Se generan retrasos en el proceso diseño de pruebas funcionales para los circuitos integrados debido a las divergencias presentes entre los modelos sintetizados y el HDL.

1.3 Solución seleccionada para acelerar el proceso de verificación lógica

Como es explicado en detalle en la sección 4.2 de este documento, para acelerar el proceso de verificación lógica de los modelos sintetizados de circuitos integrados fueron consideradas dos alternativas: el *traceback* y la búsqueda modular. El *traceback* consiste en realizar el seguimiento de una divergencia desde el puerto de salida donde es observada hasta su origen y la búsqueda modular se enfoca en aislar la divergencia en el módulo de menor jerarquía definido en el modelo sintetizado.

La búsqueda modular fue seleccionada por las razones describe en la sección 5.1 y 5.2, las cuales incluyen la capacidad del algoritmo para identificar rápidamente el módulo de menor jerarquía donde se origina de la falla y una disminución del tiempo simulación comparado con el *traceback*. Estas características hacen que la búsqueda modular satisfaga las necesidades planteadas por la empresa Intel y acelere el proceso de verificación lógica.

CAPÍTULO 2: META Y OBJETIVOS

2.1 Meta

Optimizar la verificación lógica de modelos sintetizados con el fin de reducir al mínimo la interacción humana en el proceso.

2.2 Objetivo general

Acelerar el rastreo y localización de las fallas lógicas encontradas en modelos sintetizados de circuitos integrados, con el fin de facilitar a los ingenieros la verificación lógica.

INDICADOR: Reducción del tiempo empleado para localizar las fallas de los modelos sintetizados de circuitos integrados, en un caso específico de prueba, en al menos un 50%.

2.3 Objetivos específicos

1. Implementar un algoritmo que permita rastrear y localizar las fallas encontrada en los modelos sintetizados de circuitos integrados.
   INDICADOR: bloque de lógica definido en el modelo sintetizado donde se produce el error.

2. Realizar un análisis comparativo del tiempo empleado en rastrear las fallas de un modelo sintetizado de forma manual y el tiempo empleado utilizando el algoritmo implementado, en un caso específico de prueba.
   INDICADOR: porcentaje de reducción de tiempo.

CAPÍTULO 3: MARCO TEÓRICO

En este capítulo se explicaran los principales conceptos relacionados con el diseño en HDL y el proceso de *fault grading*. Además, se hace referencia a algunos algoritmos utilizados en la solución del problema que motivó este trabajo de graduación, como por ejemplo la Distancia de Levenshtein.

3.1 Lenguajes de descripción de hardware

Los lenguajes de descripción de hardware (HDL) son lenguajes de alto nivel, similares a los de programación, con una sintaxis y semántica definidas para facilitar el modelado y descripción de circuitos electrónicos. Se utilizan para describir desde celdas base de un circuito ASIC hasta sistemas completos. [1]

En los años ochenta, se desarrollan y consolidan los lenguajes: Verilog y VHDL. Estos lenguajes son utilizados especialmente para modelar el comportamiento de un componente, verificar su funcionamiento y sintetizar una descripción correcta por construcción.

3.1.1 Características del HDL

Los lenguajes de descripción de hardware se caracterizan por permitir diferentes niveles de abstracción a la hora de describir un diseño. Los niveles de abstracción hacen referencia al detalle en que se encuentra una descripción HDL respecto a la implementación física de la misma [2]. Desde la perspectiva de simulación y síntesis, los niveles de abstracción son:

- Algorítmico, funcional o comportamental: utilizado para diseños de alto nivel donde se describe únicamente el comportamiento del circuito o sistema sin hacer ninguna relación a la implementación. Indica el comportamiento como una relación funcional entre las entradas y salidas, principalmente utilizando sentencias de alto nivel (for, if, while, etc.) no necesariamente sintetizables.
- Arquitectual o de transferencia de registros (RTL): el diseño se describe mediante las transacciones existentes entre bloques de registros, lógica booleana y sentencias de

alto nivel sintetizables. Se desarrolla una distribución de bloques funcionales y planificación en el tiempo de las funciones a realizar.

- Lógico: utilizado en diseños de bajo nivel. El circuito se describe mediante lógica booleana.

Otra característica de los modelos HDL es el estilo de descripción, también conocido como la forma de diseñar circuitos. De acuerdo a su complejidad se puede distinguir las siguientes categorías [3]:

- Flujo de datos: descripciones basadas en ecuaciones o expresiones que reflejan el flujo de información y las dependencias entre datos y operaciones.
- Comportamental: descripción algorítmica del funcionamiento del circuito, que contienen descripciones secuenciales del algoritmo correspondiente. Hace referencia a descripciones similares a la de los programas de software. No obstante, una descripción comportamental puede ser sintetizable.
- Estructural: se utiliza en circuitos que requieren más de una función, por lo que se segmenta el circuito en sub-circuitos para facilitar el diseño. Cada componente es caracterizado utilizando una descripción del flujo de datos o comportamental. En este estilo se reflejan directamente componentes por referencia y conexiones entre ellos a través de sus puertos de entrada/salida.

3.1.2 Proceso de diseño utilizando VHDL o Verilog

En un procedimiento de diseño (flujo de diseño) basado en VHDL o Verilog hay varios pasos a seguir. El primer paso del diseño consiste en la construcción del diagrama en bloque del sistema a partir de los objetivos y requerimientos planteados, tales como funciones del circuito, máxima frecuencia de operación y puntos críticos del sistema. Seguidamente se programa el código en VHDL o Verilog para cada módulo y sus interfaces. El compilador analiza el código y determina errores de sintaxis.

EL siguiente paso es la simulación del código o simulación funcional que permite verificar que el circuito funciona correctamente. Se puede realizar una verificación del comportamiento funcional, en donde se estudia el comportamiento lógico de las compuertas, y una verificación

de tiempo, en donde se incluyen demoras en las compuertas como los tiempos de establecimiento (set-up time) y mantenimiento (hold time).

La fase final del diseño incluye la síntesis del diseño y simulación del código sintetizado. La síntesis consiste en reducir una descripción realizada en un lenguaje de alto nivel de abstracción a un nivel de compuertas que pueda ser implementado en un circuito. En este proceso la descripción del circuito es convertida en un listado de conexiones (*netlist*) entre las compuertas, registros, multiplexores, etc. del dispositivo lógico. El proceso de síntesis depende de la configuración y especificaciones del diseño. Generalmente una misma función es implementada de diferentes formas, de acuerdo a la síntesis que se utilice, sin comprometer la funcionalidad del diseño pero sí su desempeño. [3]

La simulación del código sintetizado permite evaluar el circuito para determinar realmente quedó sintetizado correctamente. Ya que la sustitución de funciones dependerán de las directivas de síntesis especificadas.

3.2 *Fault grading*

Fault grading es el proceso de evaluar o clasificar una serie de pruebas de acuerdo a su eficacia en la detección de posibles defectos de fabricación. Los propósitos del *Fault grading* son servir como un proceso de desarrollo de pruebas y medir y mejorar la calidad de las pruebas en producción. *Fault grading* es implementado con la ejecución de simulaciones de fallas, mecanismo que consiste en la evaluación y clasificación de pruebas completas. [4]

La tarea de escribir pruebas para grandes circuitos en VLSI es muy difícil ya que los diseños tienen gran complejidad y se requiere comprensión y conocimientos en profundidad de ellos. Además, la inexactitud en la evaluación de la eficiencia y cobertura de las pruebas existentes hace difícil de identificar las zonas que no han sido cubiertas por las pruebas (tests holes).

3.2.1 Etapas del *Fault grading*

Como se muestra en la Figura 3-1, el *Fault grading* es un proceso que conlleva al menos cinco pasos que se describen a continuación:

1. Construcción del modelo: en esta etapa se construye un modelo lógico del circuito. Este modelo es la copia libre de defectos del circuito usado como base de comparación durante el proceso de simulación de fallas, también conocido como modelo en buen estado.
2. Verificación lógica: consiste en ejecutar en el modelo lógico cada una de las pruebas de la colección o grupo de pruebas (‘test suite”) y comparar los resultados contra las simulaciones de otro simulador, por ejemplo un simulador de VHDL. Una vez que en el modelo lógico se simulan correctamente las pruebas se puede comenzar con la simulación de fallas.

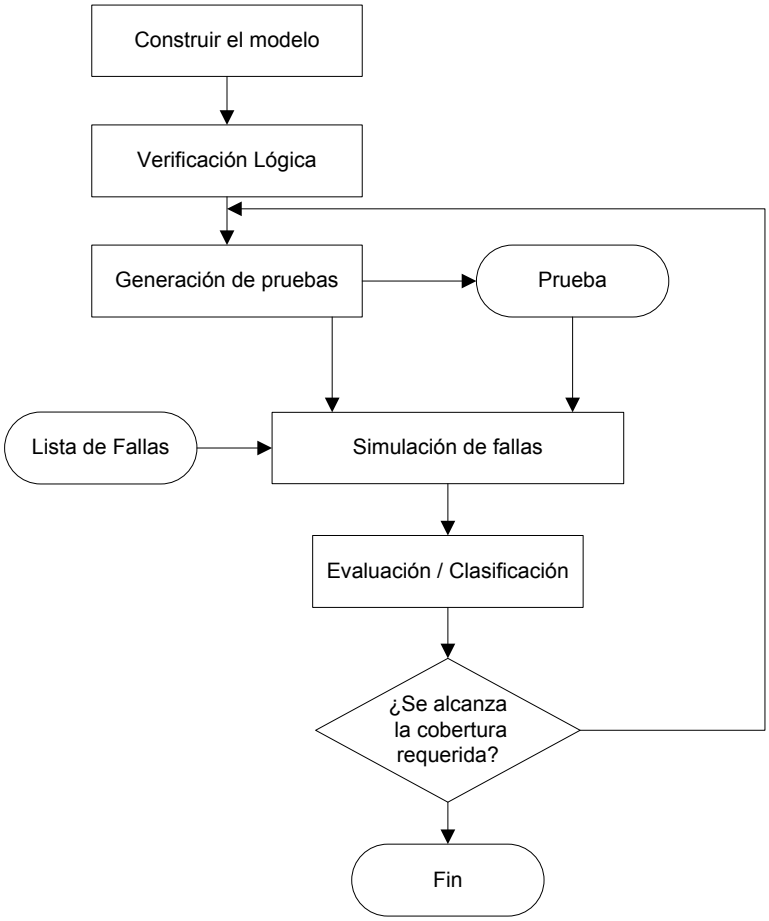


Figura 3-1 Etapas en el proceso de Fault Grading

3. Preparación y mejorar del conjunto de pruebas: preparación de un conjunto de pruebas iniciales, las cuales serán mejoradas mediante la adición de otras pruebas posteriormente.
4. Simulación de fallas: analiza el comportamiento del circuito bajo condiciones especiales definidas para la simulación de fallas.
5. Evaluación – Clasificación: evalúa los resultados de la simulación de fallas y la eficiencia del grupo de pruebas utilizado (puede ser el total del grupo de pruebas o un subgrupo). La cobertura de fallas, también conocido como *Fault Coverage* calculada en este paso se comprueba con la cobertura requerida. Si la cobertura no es adecuada, se regresa al paso 3 del proceso (mejorar el conjunto de pruebas).

3.2.2 Generación de pruebas

Una falla es la representación de un defecto caracterizado por una condición física, la cual provoca que el circuito deje de operar en la forma requerida. Un fallo es una desviación en la ejecución de un circuito o sistema de su comportamiento especificado y representa un estado irreversible en el componente, el cual debe ser reparado para lograr el comportamiento establecido en el diseño. Un error de circuito es una señal de salida errónea producida por un circuito defectuoso. Un defecto del circuito puede provocar una falla, una falla puede causar un error de circuito, y un error de circuito puede resultar en un fallo del sistema [5].

Para probar un circuito con *n* entradas y salidas *m*, un conjunto de patrones de entrada se aplica al Circuito Bajo Prueba (CUT - Circuit Under Test) y sus respuestas son comparadas con las respuestas buenas obtenidas de un circuito libre de defectos. Cada patrón de entrada es conocido como vector de prueba. [6]

Para probar un circuito completo se requieren de muchos patrones de prueba, sin embargo, es difícil saber cuántos vectores de prueba son necesarios para garantizar el correcto funcionamiento de un circuito. Para un CUT de n-entradas de lógica combinacional, se puede aplicar 2^n posibles patrones de entrada para la prueba de las fallas, este enfoque se denomina *prueba exhaustiva*. Desafortunadamente, la prueba exhaustiva no es práctica cuando el n es muy grande. Por otra parte, la aplicación 2^n patrones de entrada a un circuito de lógico secuencial no garantizan que todos los posibles estados se han visitados. En estos casos se analiza cada entrada en la tabla de verdad para el circuito de lógica combinacional para utilizarlo como patrón de entrada. Este tipo de enfoque se conoce como *prueba funcional*. En la

práctica, las pruebas funcionales son consideradas por muchos diseñadores e ingenieros para probar los CUT lo más posible en un modo similar de operación. Ya sea con pruebas exhaustivas o funcionales el mayor problema es la falta de una medida cuantitativa de las fallas que se detecten por un conjunto de vectores de prueba. [5]

Un enfoque más práctico es seleccionar patrones específicos de prueba basados en la información del circuito estructural y un conjunto de modelos de fallas. Este enfoque se denomina *prueba estructural*. Los vectores de prueba se enfocan en fallas que resultan de defectos en el circuito fabricado, disminuyendo de esta forma el número total de patrones de prueba y por consiguiente ahorrando tiempo y mejorando la eficiencia de prueba. Las pruebas estructurales no garantizan la detección de cualquier defecto de fabricación, ya que los vectores de prueba son generados a partir de modelos de fallas específicos; sin embargo, el uso de modelos de fallas proporciona una medida cuantitativa de las capacidades de detección de errores de un conjunto dado de vectores de prueba en un modelo de fallas determinado. Esta medida se llama cobertura de fallas y se define como:

$$\text{Cobertura de fallas} = \frac{\text{Número de fallas detectadas}}{\text{Número total de fallas}} \tag{3-1}$$

Es casi imposibilidad de obtener una cobertura de fallas de 100% debido a la existencia de defectos no detectables. Una falla no detectable significa que no hay prueba para distinguir un circuito sin fallos de un circuito defectuoso que contengan esa falla.

3.2.3 Modelos de fallas

Debido a la diversidad de posibles defectos en los circuitos de integración en escala muy grande, conocidos como VLSI, es difícil generar pruebas para los defectos reales. Los modelos de fallas son necesarios para generar y evaluar un conjunto de vectores de prueba.

Por lo general, un buen modelo de fallas debe cumplir dos requisitos: debe reflejar con exactitud el comportamiento de los defectos y debe ser computacionalmente eficiente en términos de simulación de fallas y la generación de patrón de prueba. Muchos modelos de fallas se han propuesto, pero desafortunadamente no existe un modelo de fallas único que refleje con precisión el comportamiento de todos los posibles defectos que pueden ocurrir.

Como resultado, una combinación de diferentes modelos de fallas es usado frecuentemente en la generación y evaluación de los vectores de prueba [5].

Para un modelo de fallas dado existirán k diferentes tipos de fallas que pueden ocurrir en cada sitio potencial de falla ($k = 2$ para la mayoría de los modelos de fallas). Un circuito dado contiene n sitios con posibles fallos, lo que puede variar dependiendo del modelo fallas. Suponiendo que sólo puede haber una falla en el circuito, entonces el número total de posibles fallas individuales, conocido como *single-fault model*, esta dado por:

$$\text{Número de fallas individuales} = \text{k x n} \tag{3-2}$$

En la realidad, múltiples fallas pueden ocurrir en un circuito. El número total de posibles combinaciones de múltiples fallas, denominado *multiple-faults model*, está dada por:

$$\text{Número de múlitples fallas} = (\text{k} + 1)^{\text{n}} - 1 \tag{3-3}$$

El modelo de *stuck-at fault* es el modelo de fallas más común utilizado actualmente en la industria, ya que modela el comportamiento lógico de las fallas físicas más frecuentes. El modelo *stuck-at fault* fue originalmente desarrollado para representar los diseños a nivel de compuertas lógicas. En este nivel se supone que, independientemente de la tecnología utilizada en la implementación de las compuertas, la mayoría de los defectos internos se manifestarán como un malfuncionamiento lógico de las compuertas [4].

El modelo *stuck-at fault* consiste en transformar el valor correcto de una línea en una señal defectuosa que parece estar fijada a un valor lógico constante, ya sea un valor lógico de 0 ó 1, refiriéndose a *stuck-at-0* (SA0) o *stuck-at-1* (SA1) respectivamente.

Existen otros modelos, como el *connector-switch-attenuator (*CSA), el *stuck-open* and el *stuck-short*, los cuales fueron específicamente desarrollados para el nivel de transistores. Sin embargo, son modelos muy complejos y requieren mucho tiempo de ejecución.

3.2.4 Simulación de fallas

Un simulador de fallas emula un grupo de defectos para un circuito con el fin de determinar el número de fallas que son detectadas por un conjunto de vectores de prueba dado.

El propósito de la simulación de fallas es observar el comportamiento del diseño eléctrico bajo condiciones de fallas simuladas. La simulación de fallas únicamente puede iniciarse después de la verificación lógica, ya que provee de una descripción completad de cómo opera una versión libre de defectos del circuito bajo prueba.

La simulación de fallas es el proceso de tomar la versión libre de defectos del circuito como control y sistemáticamente insertar fallas en el circuito para observar si los vectores de prueba pueden detectar las diferencias entre los dos circuitos. Si las diferencias lógicas son observadas entre la salida del circuito libre de defectos y el circuito con defectos, el defecto es considerado cubierto por el test [4]. La comparación en la simulación de fallas se muestra en la Figura 3-2.

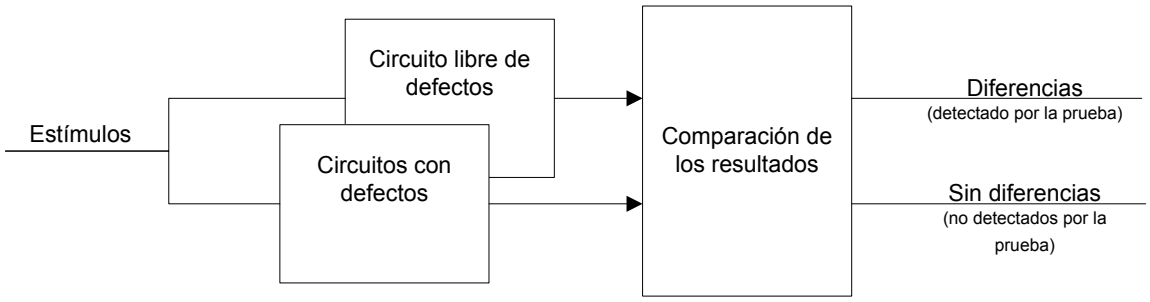


Figura 3-2 Comparación en la simulación de fallas

Debido a que el analizador de fallas detectadas tiene muchas fallas por emular, el tiempo de simulación de fallas es mucho mayor que el requerido para la verificación del diseño [5]. Para acelerar el proceso de simulación de fallas, se han desarrollado mejoras a los enfoques el siguiente orden.

- Simulación de fallas paralelas: hace uso del bit-paralelismo en las operaciones lógicas procesadas por una computadora digital. Así, por ejemplo en un equipo de 32 bits 31 faltas se simulan simultáneamente.

- Simulación deductivo: reduce todos los valores de las señales en cada circuito defectuoso a partir: de los valores de circuito libre de errores y la estructura del circuito.

- Simulación de fallas concurrentes: es esencialmente una simulación por eventos para emular fallos en un circuito de la manera más eficiente. Los aceleradores de simulación

de fallas basados en Hardware se basan en el procesamiento en paralelo y también proporcionan una importante aceleración durante puramente simuladores de fallas basadas en software.

Para circuitos analógicos y de señal mixta, la simulación de fallas tradicionalmente se realiza a nivel de transistor utilizando simuladores de circuitos. Desafortunadamente, la simulación de fallas analógica es una tarea que requiere mucho tiempo y, incluso para circuitos sencillos, una simulación completa de fallas no es factible.

3.3 Formatos de *dumpfile*

Un archivo *dumpfile* ofrece los resultados detallados paso a paso de una simulación para el análisis, re-simulación, o verificación del diseño [7]. Existen varios formatos de *dumpfiles*, entre ellos se encuentra el VCD, FSDB y VPD.

3.3.1 El archivo VPD

El VPD es un tipo de archivo binario, propiedad de la compañía Synopsys, utilizado para trazar las formas de onda de señales. El nombre VPD proviene de la abreviatura de VCD Plus. EL tamaño del archivo generado con este formato es muy pequeño frente a VCD. Por ejemplo, si se utiliza 1012K bytes para un archivo VCD, en un archivo VPD para el mismo circuito se requieren 27K bytes. [8]

3.3.2 El archivo FSDB

El archivo FSDB es un formato de archivo binario que proporciona una representación binaria de los datos de simulación de varios formatos, incluyendo VCD y EVCD Verilog y VHDL [9]. El término FSDB es el acrónimo de *Fast Signal Database*.

Los archivos FSDB tienen ventajas sobre el estándar de formato de archivo VCD. La principal es que un archivo FSDB es más compacto que un archivo estándar de VCD. Normalmente, un archivo FSDB es de 5 a 50 veces más pequeño que un archivo VCD [10].

3.3.3 El archivo VCD

El VCD por sus siglas en inglés (*Value Change Dump*), es un archivo de estímulos que contiene un registro de únicamente las señales que cambian de estado. Este archivo muestra el valor lógico que tiene una señal cada ciclo o vector en el que cambia su estado y utiliza los caracteres del código ASCII para definir los nombres y los valores de las señales. [11]

```
$date
        Wed Aug  4 06:12:52 2010

$end
$version
        VCS MX(MHPI-based)
$end
$timescale
        1ps
$end
$scope module system $end
$scope module smca $end
$scope module dmi $end
$var wire 1 ! ckcore_zcz1n00nfw $end
$var reg 1 " pcierstb $end
$var wire 1 # cck_usync_zcznfwh $end
$var wire 1 $ vccsocvid_1p05 $end
$var wire 1 % dfx_glb_clk_override_zcznfwh $end
$var wire 1 & dfx_glb_scan_rst_zcznfwh $end
$var wire 1 ' dfx_glb_scan_en_zcznfwh $end
$var wire 1 ( dfx_glb_scan_clk_zcznfwh $end
$var wire 1 ) dfx_glb_edt_clk_zcznfwh $end
$var wire 1 * dfx_glb_edt_update_zcznfwh $end
$var wire 1 + dfx_glb_capture_usync_zcznfwh $end
$var wire 128 , dfx_glb_side_chain_zcznfwh [127:0] $end
$var wire 20 - dfx_glb_dft_cntl_zcznfwh [19:0] $end
$var wire 1 . vss $end
$var wire 1 / dfx_xxx_coreclk_freeze_zcznfwh $end
$var wire 1 0 dfx_scan_debug_test_mode_dc_nczfwoh $end
$var wire 1 1 dfx_nctaaryfrz_zcznfwh $end
```

Figura 3-3 Ejemplo del encabezado de un archivo de formato VCD

El formato de archivo VCD, para la herramienta de síntesis y simulación del modelo, puede describirse utilizando los estándares $dumpvars o $dumpports, los cuales son aceptados actualmente por la IEEE.

- Formato $dumpvars

El formato $dumpvars fue el primer formato para un archivo VCD que fue aceptado por la industria, por lo tanto la IEEE lo adoptó y lo estandarizó.

En un archivo VCD los posibles valores lógicos que pueden ser especificados para cada señal o pin son 01XZ. Los pines declarados como entrada y como salida pueden tener cualquiera de los cuatro valores lógicos definidos sin requerir modificación. Sin embargo, los pines que son entrada y salida a la vez son divididos en dos pines independientes, y en el archivo VCD se separan los controladores de las señales.

- Formato $dumpports

El formato $dumpports es otro tipo de especificación para archivos VCD que utiliza 25 caracteres para la representación de los valores lógicos de las señales (LlHhTXx?01AaBbCcFfDdUuNnZ). Este formato llegó después de $dumpvars y ahora está estandarizado por la IEEE. En el mercado existen diversos simuladores que generan archivos en formato VCD $dumpports

El principal beneficio de utilizar $dumpports es que proporciona un valor de carácter único para una pin de entrada y salida, a diferencia de $dumpvars, no está dividido en pines de entrada y salida independientes.

3.4 Búsqueda difusa de cadenas

La búsqueda difusa de cadenas, también conocida como *Fuzzy String Matching*, es una técnica de búsqueda de coincidencias aproximadas de un patrón en una cadena. La coincidencia entre cadenas se mide en términos del número de operaciones primitivas necesarias para convertir la cadena en una coincidencia exacta. Este número se llama la distancia de edición entre la cadena y el patrón. Existen tres operaciones primitivas:

- Inserción: musa -> musas
- Eliminación: musa -> usa
- Sustitución: musa -> masa

Algunas búsquedas difusas también requieren de la transposición, donde las posiciones de dos letras en la cadena se intercambian, como una operación primitiva.

La correspondencia de cadenas se utiliza principalmente para buscar o reemplazar algún subtexto o patrón. Entre más grande sea el texto que se está examinando, mas importante es la eficiencia del algoritmo de búsqueda.

3.4.1 Distancia de Levenshtein

La distancia de Levenshtein es un algoritmo tal que dadas dos cadenas de caracteres devuelve un número entero que da una idea de la distancia o parecido entre ellas. El número entero se calcula contando las transformaciones que son necesarias para obtener a partir de una las cadenas la otra. Las posibles transformaciones incluyen sustituciones, inserciones y eliminaciones de caracteres. [12]

La distancia Levenshtein toma este nombre del científico ruso Vladimir Levenshtein, quien ideó el algoritmo en 1965. En la actualidad, este algoritmo se utiliza en el desarrollo de aplicaciones tales como sistemas para la revisión de faltas ortográficas automatizada en textos, reconocimiento de voz y análisis de ADN. [13]

El algoritmo se basa en creación de una matriz de enteros que tenga tantas filas como la longitud de una de las cadenas más uno, y tantas columnas como la longitud de la otra más uno. La primera fila y la primera columna se rellenan con los valores 0,1,2,3... hasta llenarlas por completo, y a partir de ahí se va rellenando el resto de la matriz los costos de la transformaciones, y quedándonos siempre con el mínimo. Cuando la matriz está completamente llena, la última casilla de la matriz nos dará la distancia de Levenshtein.

Los costos de la transformación son:

- El elemento de la fila superior más uno (eliminar carácter).
- El elemento de la izquierda más uno (insertar carácter).
- El elemento anterior de la diagonal más: 1 si las celdas comparadas son diferentes ó 0 si son iguales (sustituir carácter).

3.4.2 Distancia de Damerau-Levenshtein

Se llama distancia de Damerau-Levenshtein al número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra. Las operaciones permitidas son la inserción, eliminación, sustitución o transposición de dos caracteres. Lo que la distingue de

la distancia de Levenshtein es que cuenta con una segunda operación de edición: la transposición.

3.4.3 Distancia de Hamming entre 2 cadenas

La distancia de Hamming se concibió originalmente para la detección y corrección de errores en comunicaciones digitales. Sin embargo, las ventajas del algoritmo son aplicables a diversos tipos de cadenas y no se limita a cadenas numéricas. [14]

La distancia de Hamming entre 2 cadenas es igual al menor número de sustituciones que se requiere para convertir una cadena en otra. No obstante, las cadenas analizadas deben ser de la misma longitud. Dado que el algoritmo se basa en el costo de la transposición de una cadena en otra, las cadenas de longitud desigual darán lugar a altas penalizaciones para la transposición.

CAPÍTULO 4: PROCEDIMIENTO METODOLÓGICO

En este capítulo se describe el origen y las características del problema, las alternativas consideradas en la solución y el procedimiento seguido durante el diseño y desarrollo de la herramienta software.

4.1 Reconocimiento y definición del problema

En esta sección se explican la importancia de los modelos sintetizados y la metodología utilizada originalmente para encontrar el origen de los errores de comportamiento presentes en los modelos.

4.1.1 Antecedentes

En las primeras etapas del proceso de diseño de circuitos integrados se definen y describen las funciones que se desean implementar en el nuevo hardware. A partir de estas especificaciones de diseño se describe el comportamiento de los distintos bloques del circuito haciendo uso de un Lenguaje de Descripción de Hardware (HDL – Hardware Description Language). El diseño debe ser verificado en varias partes del proceso para asegurar que en la implementación física del circuito integrado no se presenten errores funcionales.

En la etapa de ensamblaje los circuitos integrados pasan por una serie de pruebas que permiten detectar varios tipos de problemas como lo son: cortocircuitos, circuitos abiertos y fallas debido a la velocidad de operación, entre otros. Estas pruebas son realizadas mediante el uso de equipos de prueba especializados, los cuales se comunican con los circuitos y ejecutan patrones de prueba. Los patrones de prueba están diseñados para ejercitar diferentes funciones de la arquitectura y es por medio de puntos de observación ubicados dentro del circuito, tal es el caso de los circuitos de scan-chain, LFSR, ATPG, JTAG y otras arquitecturas conocidas por sus siglas en inglés como DFT (*Design for Test*), que se determina si el comportamiento del circuito es el esperado; en caso de que se registren valores erróneos se procede a descartar el dispositivo.

Las pruebas son diseñadas en forma paralela al proceso de diseño del circuito integrado, mucho antes de la etapa de ensamblaje y prueba, por lo que para observar el comportamiento que tendrá el circuito integrado ante una determinada prueba funcional se deberá realizar una simulación sobre un modelo lógico, sintetizado a partir del HDL. Este modelo sintetizado es una representación a nivel de compuertas lógicas y registros de los bloques descritos en el HDL; que no sólo debe tener el mismo comportamiento que el de las especificaciones del diseño, sino que debe modelar en sus nodos los posibles defectos en el proceso de manufactura.

4.1.2 Divergencias entre los modelos sintetizados y el HDL

Los modelos sintetizados de circuitos integrados se generan a partir del HDL por medio de herramientas de software. Este proceso de síntesis no se encuentra libre de errores, ya que se presentan diferencias entre el comportamiento descrito en el HDL y el modelo sintetizado.

En la Figura 4-1 se muestra un circuito básico de prueba, el cual recibe un vector de entrada igual a “111”. El vector de salida esperado en circuito según las simulaciones realizadas en el HDL es igual “00”, sin embargo, el vector obtenido del modelo sintetizado es “10”. Esta diferencia entre el resultado obtenido en el HDL y en el modelo sintetizado es conocida como una divergencia (*mismatch*), y está caracterizada por una señal a la salida del circuito y un vector de tiempo. Por ejemplo, para el circuito mostrado la divergencia sería S0 en el vector de tiempo 0.

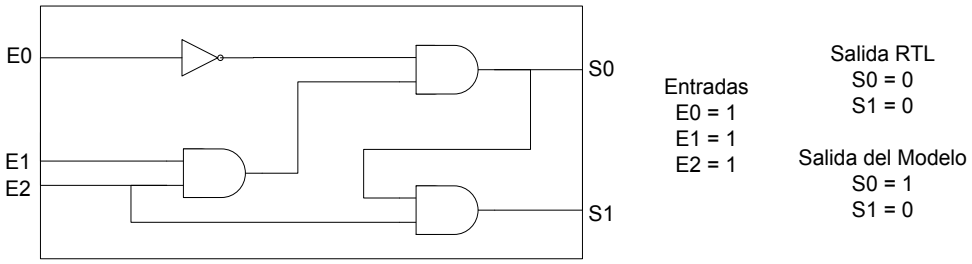


Figura 4-1 Divergencias entre los modelos sintetizados y el HDL

Durante la etapa de diseño el HDL es verificado funcionalmente, por lo tanto en el proceso de verificación lógica se parte del hecho de que el comportamiento que tiene el circuito descrito en el HDL es el esperado. Esto significa que las diferencias encontradas entre los resultados obtenidos en el modelo sintetizado y el HDL son producto de errores en el modelo sintetizado del circuito.

Como se muestra en la Figura 4-2, los modelos sintetizados utilizados suelen ser de gran tamaño y complejidad; en su mayoría están conformados por un módulo principal que está integrado por varios módulos secundarios que a su vez cuentan con sub módulos. Los circuitos o módulos donde se originan las divergencias son desconocidos, la única referencia con que se cuenta son los puertos o pines de salida donde se propagan lo errores.

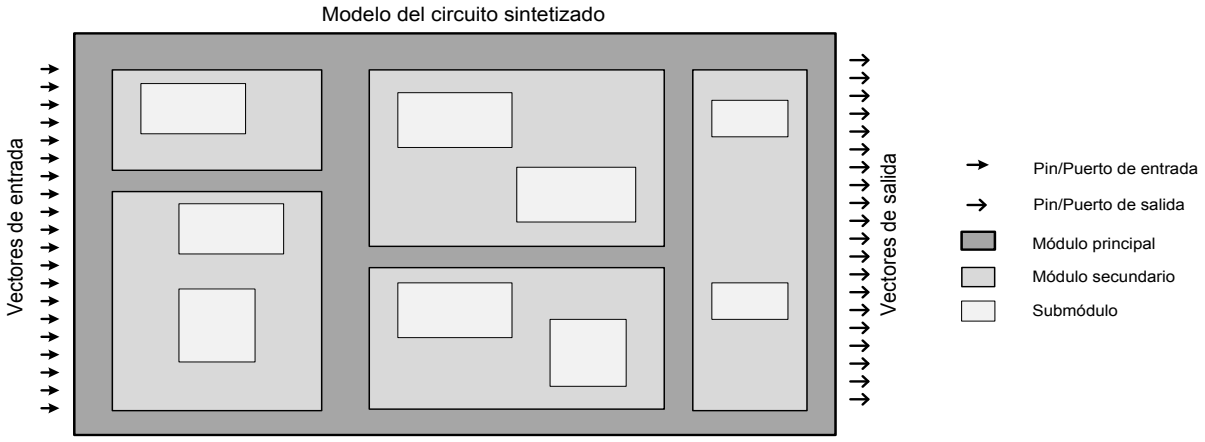


Figura 4-2 Ejemplo de un modelo sintetizado de circuito integrado

En la Figura 4-3 se representa como una falla se propaga a diferentes módulos del diseño y puede reflejarse en varios pines o puertos de salida. Un factor a consideración es el hecho de que el error se puede originar en un vector de tiempo diferente de aquel en que se observa su efecto en los pines de salida. Ello puesto que el diseño esta conformado por circuitos secuenciales, los cuales no siempre están controlados por un mismo reloj y requieren de ciclos de reloj extra para propagar los datos.

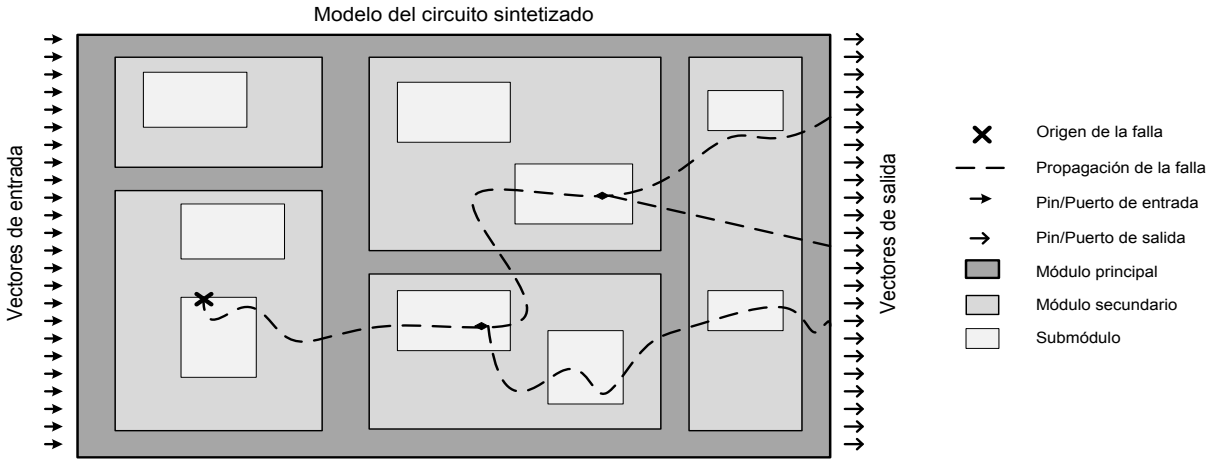


Figura 4-3 Propagación de errores en los modelos sintetizados de circuitos integrados

Determinar el origen de un error que se ha propagado a la salida, conocido como aislamiento de fallas, es un proceso lento que requiere de sucesivas simulaciones y análisis de la lógica del circuito, principalmente porque no solo se debe hacer una búsqueda espacial en el circuito sino también temporal.

4.1.3 Aislamiento de fallas

El aislamiento de fallas consiste en determinar cuál es la señal o bloque de lógica definido en el modelo sintetizado que produce el error. Este proceso requiere de mucho tiempo, desde horas hasta semanas, principalmente por la complejidad de los circuitos integrados y la existencia de lógica con realimentación.

El proceso de asilamiento de fallas puede dividirse en dos etapas: simulación de pruebas funcionales en el modelo sintetizado y análisis de los errores encontrados. Como primero paso se debe simular la prueba funcional hasta el vector de tiempo donde se detecta un error. Como se muestra en la Figura 4-4, a partir de este punto se analiza la lógica predecesora y se determinan las posibles rutas de donde proviene el error. Conforme se va retrocediendo en la lógica del circuito y en el tiempo se tiene que repetir el proceso.

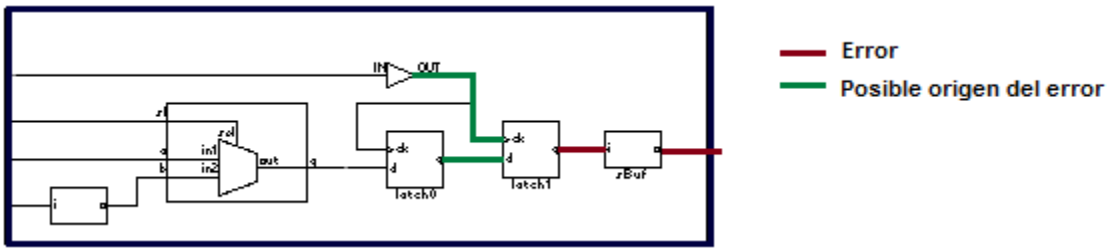


Figura 4-4 Aislamiento de fallas en modelos sintetizados

Por ejemplo, a un registro que presenta un valor incorrecto en su pin de salida se le deben analizar la entrada de datos y el reloj. Con una exploración del circuito se establece que el origen de la falla proviene ya sea de un error en el reloj, un error en la entrada de datos, un error en ambos pines o no existe un error. En el caso de que el error esté presente en alguno de los pines de entrada se debe seguir analizando la lógica predecesora. En el caso de que no se encuentre error es necesario analizar el estado anterior del registro, lo que implica simular nuevamente en el modelo hasta el estado anterior del registro.

Una vez aislado el origen de la falla se debe determinar si el origen es de carácter lógico, si es un problema inherente a la herramienta de síntesis o si se trata de una mala interpretación de la herramienta de simulación. Una vez que todas las fallas son corregidas, la simulación de una prueba funcional debe finalizar sin divergencias entre el comportamiento esperado en el modelo y el HDL.

4.1.4 Importancia de la verificación lógica

En el proceso de diseño de pruebas funcionales es indispensable contar con un modelo sintetizado, que haya sido validado, ya que este permite determinar: el porcentaje del circuito que ha sido ejercitado por las pruebas, los bloques o funciones del diseño que no han sido estimulados y si existen nodos en el circuito que han sido ejercitadas por la prueba pero que no logran propagarse hasta los puntos de observación del sistema. A partir de estos datos obtenidos del modelo sintetizado se diseñan otras pruebas funcionales que ejerciten la lógica faltante o se diseñan nuevos bloques que faciliten la observación de fallas.

Si no se cuenta con un modelo sintetizado verificado se pueden presentar retrasos en el proceso de diseño de pruebas funcionales y por consiguiente comprometer las pruebas que se le realizan a los productos en el proceso de manufactura. Esto se traduce en pérdidas económicas para la compañía ya que en el proceso de manufactura se deberá invertir dinero en más equipo de prueba.

4.2 Evaluación de las alternativas

Las alternativas consideradas para acelerar el aislamiento de fallas son el *traceback* y la búsqueda modular. El *traceback* es utilizado actualmente en la industria, requiere de mucho tiempo de simulación y análisis, sin embargo, es altamente efectivo en la identificación del origen de las fallas en el modelo. La búsqueda modular es una alternativa planteada a partir de un análisis general del problema y de fallas aisladas en procesos de verificación lógica anteriores.

A continuación se explican con mayor detalle cada una de las alternativas analizadas para el planteamiento de una solución al problema. Además, se hace un estudio de algunos algoritmos que permitan realizar una comparación entre cadenas de caracteres.

4.2.1 Traceback

El *traceback* es un nombre dado a cualquier método que permite determinar el origen fiable de un objeto o problema. En el proceso de aislamiento de fallas, el *traceback* es un procedimiento que permite seguir un error encontrado en la salida del modelo sintetizado a través del camino que sigue hasta su origen. Un error puede observarse en más de un pin de salida y tener rutas hacia el origen, como el ejemplo de la Figura 4-3, por lo que el *traceback* puede ser un proceso laborioso y no fácil de resolver.

Durante el proceso de *traceback* se observarán casos donde las divergencias observadas en los pines de salida siguen un camino hacia origen de la falla que incluye pocas compuertas y registros, como es sería el caso del circuito mostrado en la Figura 4-5. Sin embargo, lo más común y que a su vez requiere de más trabajo es encontrarse con caminos que incluyen circuitos secuenciales complejos que pueden contar con retroalimentación en sus entradas. En la Figura 4-5 se muestra un ejemplo de retroalimentación donde el error se origina en un registro (identificado con la X), se realimenta al circuito por medio de la entrada E0 y es observado en el pin de salida S3.

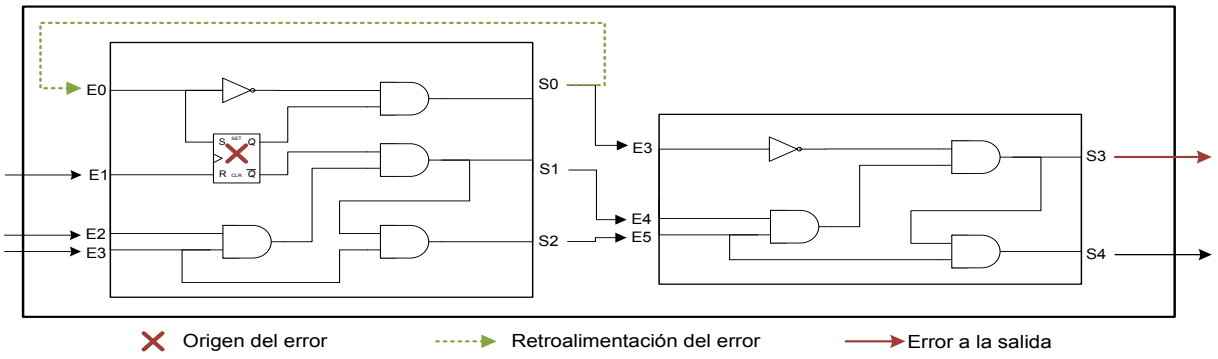


Figura 4-5 Retroalimentación de errores en lógica secuencial

Una falla tiene una propagación espacial en las compuertas lógicas, y temporal en los elementos de memoria del circuito; por esta razón el origen de la falla es diferente al punto donde se observa. En circuitos con retroalimentación el rastreo de una falla puede necesitar

pasar varias veces por un mismo camino hasta encontrar el punto de origen, por lo que son necesarias muchas simulaciones y tiempo de análisis; si a esto le agregamos módulos de mayor tamaño que incluyen mas compuertas lógicas y elementos de memoria, es comprensible la gran inversión de tiempo y recursos que puede llevar un solo rastreo. Por ejemplo, en la verificación funcional del controlador de memoria de un microprocesador se emplearon 3600 horas ingeniero.

4.2.2 Búsqueda modular

Los circuitos integrados diseñados en la actualizada en su mayoría son descritos estructuralmente en HDL. Esto significa que un circuito se encuentra definido como un conjunto de circuitos más pequeños, que a su vez se están conformados por otros circuitos que realizan funciones más simples.

En el proceso de síntesis del circuito, el modelo lógico se crea con base en la estructura definida por los diseñadores en el HDL. Por lo tanto en el modelo sintetizado se encuentran definidos conjuntos de módulos estructurados jerárquicamente, como en la Figura 4-6. Para hacer referencia a un sub módulo se debe especificar el nivel jerárquico en que éste se encuentra; por ejemplo, para citar al sub módulo L1 se usaría una expresión como /CPU/CACHE/C0/L1.

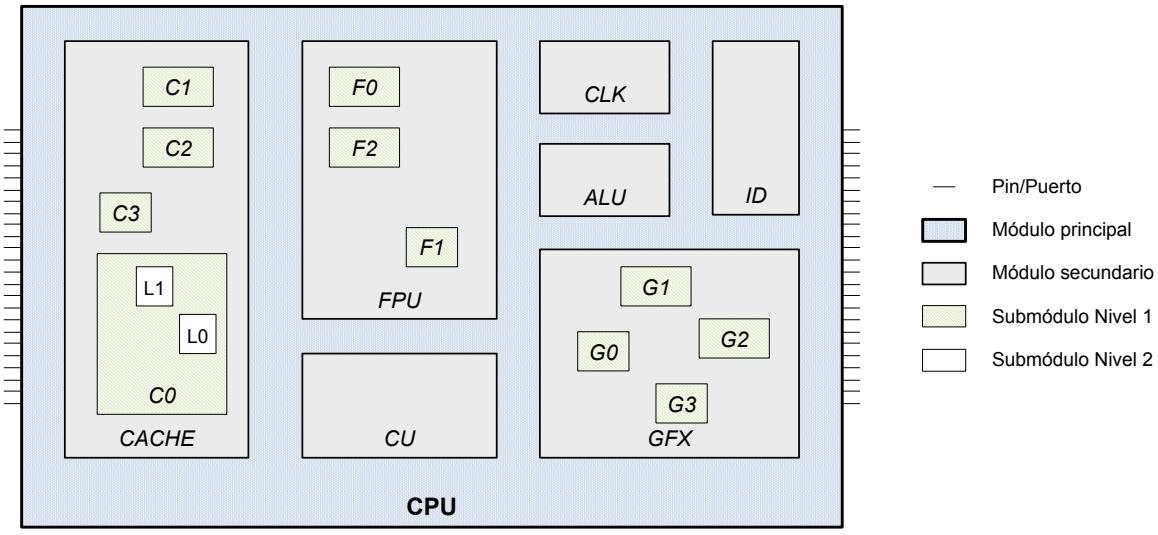


Figura 4-6 Diseño modular en los sistemas sintetizados

El algoritmo planteado para agilizar la verificación lógica del modelo pretende analizar el módulo principal en busca de errores en los pines de salida, y en el caso de que existan, se repetirá la simulación observando las salidas de los módulos secundarios. Si se encuentran errores en alguno módulo secundario se deberá repetir la simulación, pero esta vez se observanrán únicamente las salidas de los sub-módulos de nivel 1 definidos en el módulo secundario donde se halló el error. El proceso se repetirá sucesivamente hasta determinar el módulo de menor jerarquía y tamaño donde se puede aislar una falla.

Basandose en la Figura 4-7 y suponiendo que existe un error en el submódulo de nivel 2 llamado L1, el algoritmo a implementar debe simular la prueba funcional en el modelo sintetizado observando las salidas de los sub-módulos pertenecientes a las siguiente jerarquías: /CPU, /CPU/CACHE, y /CPU/CACHE/C0. En la primera simulación /CPU se observan los pines de salida de los módulos secundarios (CACHE, FPU, CU, ALU, CLK, ID y GFX); en la segunda simulación /CPU/CACHE los pines de los sub-módulos de nivel 1 (C0, C1, C2 y C3); en la simulación final /CPU/CACHE/C0 se observan los sub-módulos de nivel 2 (L0 y L1). Con el resultado de la última simulación se puede determinar que el error proviene del submódulo de nivel 2 L1, ya que únicamente se observan errores en las salidas de este.

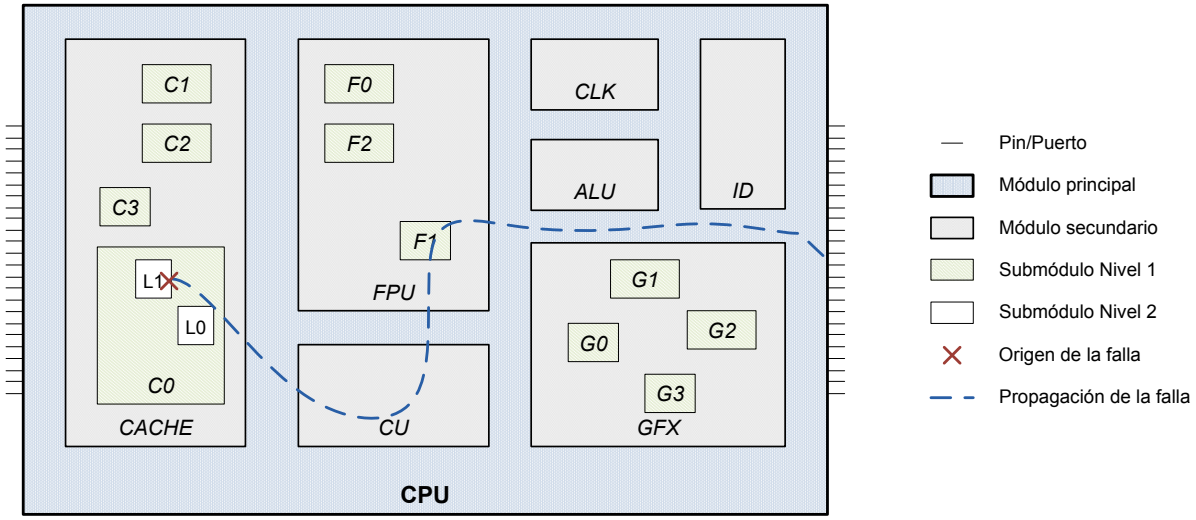


Figura 4-7 Búsqueda modular de errores en modelos sintetizados de circuitos integrados

Las fallas encontradas en una simulación están reflejadas en los pines de salida y cuentan con un vector de tiempo que representa el momento en que es observada en la simulación. Por esta razón, en el ejemplo de la Figura 4-7 los módulos secundarios FPU y CU aunque son

parte del camino que sigue el error, no son analizados, ya que en ellos los errores se observan tiempo después que en el módulo CACHE. El descartar módulos donde se originan las fallas por medio del vector de tiempo en que se observan es una ventaja que presenta la búsqueda modular. Además facilita la búsqueda en circuitos que tiene retroalimentación de la lógica, ya que permite enfocarse en un módulo y momento (vector de tiempo donde se observa por primera vez una falla) en particular.

4.2.3 Comparación de jerarquías

El código en HDL utilizado para generar los modelos empleados en *fault grading* está descrito con módulos, los cuales representan secciones del circuito y están definidos respetando una jerarquización. En el modelo sintetizado la jerarquización no siempre es equivalente a la definida en el HDL ya que la herramienta de síntesis al tener que modelar únicamente un comportamiento realiza optimizaciones, entre las que se encuentran la unificación o partición de los bloques definidos en el HDL. Por ejemplo, el módulo C0 de la Figura 4-6 está definido en el HDL como /CPU/CACHE/C0 y en el modelo sintetizado puede encontrarse como /CPU/CACHE_C0.

Las diferencias en el nombramiento de las jerarquías hace necesario contar con un procedimiento que permita conocer el nombre de una jerarquía definida en el HDL a partir del nombre establecido en el modelo sintetizado, y viceversa. Este proceso se debe realizar independientemente del enfoque que se tome para la solución del problema, ya sea *traceback* o búsqueda modular.

Para realizar una traducción entre las jerarquías, se propuso utilizar algún algoritmos de búsqueda difusa de caracteres utilizado en la industria, tales como: Levenshtein, Damerau-Levenshtein y Hamming entre 2 cadenas. Existen también algoritmos que permiten identificar semejanzas entre cadenas de caracteres tales como: Soundex y Double Metaphone; sin embargo, estos algoritmos son fonéticos y basan su comparación en las semejanzas sonoras que pueden tener algunas letras en el idioma inglés, por lo que no son aplicables a la traducción de jerarquías.

Los algoritmos de búsqueda difusa reciben como parámetro de entrada 2 cadenas de caracteres correspondientes a las jerarquías en análisis. Debido a las variantes en el nombramiento de las jerarquías las cadenas de caracteres pueden tener diferentes longitudes;

es por esto que el algoritmo de Hamming al necesitar como entrada cadenas de igual longitud queda descartado.

Los algoritmos de búsqueda difusa para detectar posibles errores en el proceso de traducción de jerarquías, requieren únicamente de las transformaciones de eliminación, inserción y sustitución de caracteres. Como se muestra en la Tabla 4-1, el algoritmo de Damerau-Levenshtein tiene también la propiedad de analizar posibles transposiciones de caracteres; esta propiedad no es requerida en la traducción de jerarquías por lo que para la implementación de la solución se considerará únicamente el algoritmo de Levenshtein.

Tabla 4-1 Transformaciones realizadas por los algoritmos de búsqueda difusa de caracteres en cadenas de caracteres

Transformaciones permitidas en las cadenas de caracteres / ***Algoritmos***	*Eliminar carácter*	*Insertar carácter*	*Sustituir carácter*	*Transposición de caracteres*
Distancia de Levenshtein	x	x	x	
Distancia de Damerau-Levenshtein	x	x	x	x
Distancia de Hamming entre 2 cadenas			x	

4.3 Síntesis de una solución

Dado que el objetivo de este proyecto es reducir el tiempo empleado para localizar las fallas de los modelos sintetizados y la retroalimentación en la lógica de los circuitos es el principal obstáculo a resolver en la verificación lógica se ha seleccionado el algoritmo de búsqueda modular como la solución para el problema planteado en este proyecto.

Para elaborar una solución robusta al problema es preciso conocer el proceso de verificación lógica e investigar sobre aspectos relacionados a esta permite tener una visión más amplia del problema. Por esta razón en este proyecto se hizo una investigación inicial que contó con las siguientes etapas:

1. Trabajar con las herramientas de diseño utilizadas en la verificación lógica, especialmente con la encargada de realizar la síntesis del modelo y la simulación de

fallas, para determinar funciones que pueden ser utilizadas en el rastreo y localización de los errores lógicos en modelos sintetizados.

2. Investigar sobre los errores lógicos más comunes encontrados en los modelos sintetizados.
3. Aislar fallas en un modelo sintetizado utilizando la metodología Trace Back, para conocer el proceso y observar posibles alternativas para la solución del problema.
4. Investigar sobre los formatos de archivos generados por herramientas de simulación lógica, tales como VCD, FSDB, VPD y TBL, para determinar cuál es el formato que mejor se adapta al proyecto.

Con base en la investigación previa se definió un algoritmo que permitiera acelerar el rastreo y localización de errores lógicos en los modelos sintetizados. A partir del algoritmo se diseñaría y programaría una herramienta de software con las siguientes funciones:

1. Detección de fallas para un módulo específico.
2. Simulación de pruebas funcionales con parámetros definidos por el algoritmo.
3. Análisis de los resultados de la simulación.
4. Generación de reportes que permitan almacenar los resultados.

Finalmente, se debían realizar simulaciones con la herramienta de software desarrollada para determinar el tiempo requerido por el programa para aislar fallas; con base en estos resultados se haría una comparación entre tiempo de ejecución empleado por algoritmo implementado y la verificación lógica de forma manual.

4.4 Implementación de la solución

La primera etapa de este proyecto estuvo enfocada en la investigación. Se analizaron las etapas involucradas en el proceso de verificación lógica para determinar posibles áreas para mejorar. Además, se estudiaron las herramientas de software involucradas en el proceso de síntesis y simulación de los modelo de circuitos integrados, con el fin de conocer las características y limitaciones.

La segunda etapa del proyecto consintió en obtener los módulos y las jerarquías definidas para un circuito. En esta fase se estudiaron el HDL, el netlist, el modelo sintetizado y los archivos con formato FSDB y VCD como las posibles fuentes de información para obtener la jerarquización modular.

Con la obtención del nombre de las jerarquías definidas en el modelo y en el HDL se procedió a implementar un algoritmo que realizara una traducción entre ambas. Para este proceso se hace uso de un algoritmo de búsqueda difusa de caracteres conocido como la distancia Levenshtein, ya definido.

Las jerarquías definidas en el modelo permitieron navegar el circuito para aislar una falla y la traducción entre jerarquías se utilizó para realizar las simulaciones. Con estos datos obtenidos se programó el algoritmo que analiza una jerarquía en particular. Una vez finalizado este análisis se procedió a implementar un algoritmo de interpretación de resultados del análisis modular y otro para controlar la recursivamente para módulos de menor estatus jerárquico.

La validación de la herramienta es la última etapa desarrollada. En esta se utiliza ya la herramienta para realizar la verificación lógica del modelo sintetizado de un circuito DMI. Cada uno de los algoritmos implementados fue evaluado individualmente para comprobar su correcto funcionamiento y como un todo para determinar si con su utilización se acelera la verificación lógica de los modelos.

4.5 Reevaluación y rediseño

Rediseñar algunos de los planteamientos iniciales permitió cumplir los objetivos propuestos y obtener una herramienta eficiente. El primer algoritmo sometido a rediseño fue el algoritmo para obtener los puertos de entrada y salida de cada uno de los módulos definidos en el circuito bajo prueba. Este algoritmo pretendía obtener la información a partir del HLD, no obstante, al realizarse algunas pruebas se encontró que existían:

- Diferencias en el nombre recibido por un mismo módulo en el HDL y en el modelo sintetizado.

- Puertos de entrada o salida definidos en el modelo sintetizado que no son parte del diseño en HDL, ya que pertenecen a lógica de circuitos de pruebas que no son funcionales o visible para los usuarios.

Por lo tanto, se descartó la opción de extraer los módulos y puertos del HDL y se evaluó obtener esta información a partir del *netlist*. El *netlist* al ser base para la síntesis del modelo presenta el mismo número de pines que el modelo sin embargo, al igual que al utilizar el HDL se encuentran diferencias en el nombramiento de los módulos. Con el fin de cerciorarse de que los datos utilizados en le verificación lógica son correctos se procedió a extraer la información requerida directamente del modelo sintetizado. Para obtener la información a partir del modelo se debe hacer uso de la herramienta de síntesis y simulación utilizada por la empresa, la cual presenta limitaciones en la interfaz de adquisición de datos y un el juego de instrucciones.

Otras de los aspectos sometidos al rediseño fue el formato de los archivos *dumpfiles* utilizados para la simulación de las pruebas funcionales. Inicialmente, el formato fsdb fue el seleccionado para ser usado como estímulo en las simulaciones, no obstante, dado que los requerimientos de la empresa varían entre proyectos fue necesario agregar la opción de poder utilizarse archivos con formato vcd. Este cambio requirió modificaciones únicamente en el algoritmo planteado para la búsqueda de errores ya que el pre-proceso ya contemplaba la utilización de los dos formatos de archivos.

CAPÍTULO 5: DESCRIPCIÓN DETALLADA DE LA SOLUCIÓN

En este capítulo se realiza un análisis comparativo de las características y limitaciones de los enfoques planteados en el capítulo 4 para la solución del problema. Además, con base en el análisis del *traceback* y la búsqueda modular se elige el algoritmo que permita acelerar el aislamiento de fallas en los modelos sintetizados y que mejor se adapte a las necesidades de la empresa. Finalmente, se describen las etapas desarrolladas en la implementación de la herramienta de software que ejecuta el algoritmo seleccionado.

5.1 Análisis de soluciones

En esta sección se realiza el análisis comparativo de las posibles soluciones para el problema de verificación lógica. Además, se justifica la elección del algoritmo de búsqueda modular como solución para el problema planteado.

5.1.1 Traceback

En la Figura 5-1 se muestra la representación gráfica del algoritmo *traceback*. Este algoritmo recibe como parámetros de entrada el puerto de salida del modelo y el vector de tiempo donde es observada la divergencia. El algoritmo inicia buscando la lógica que se encuentra antes del puerto de salida donde se observa la divergencia; para cada una de las señales encontradas en la búsqueda se procede a realizar un análisis.

El análisis consiste en determinando si la señal examinada se encuentra definida en el HDL; si la señal no se encuentre definida se debe repetir el proceso de búsqueda de la lógica anterior y si la señal es parte del HDL se inicia la simulación de la prueba funcional. La simulación se realiza hasta el vector de tiempo donde se observa la divergencia y con los datos generados por la simulación se procede a comparar el valor de la señal obtenido en el modelo sintetizado con el del HDL. En caso de que se detecten divergencias en esa señal el algoritmo se debe volver a ejecutar, únicamente que los parámetros de entrada del algoritmo deberán ser sustituidos por los de la señal analizada en ese momento; si no se detectan errores se procede a realizar la simulación de la prueba funcional tiempo atrás a donde se observa la divergencia.

Observar la simulación algunos vectores de tiempo antes de que se presenten las divergencias permite determinar si los de elementos de memoria almacenaron errores en estados anteriores.

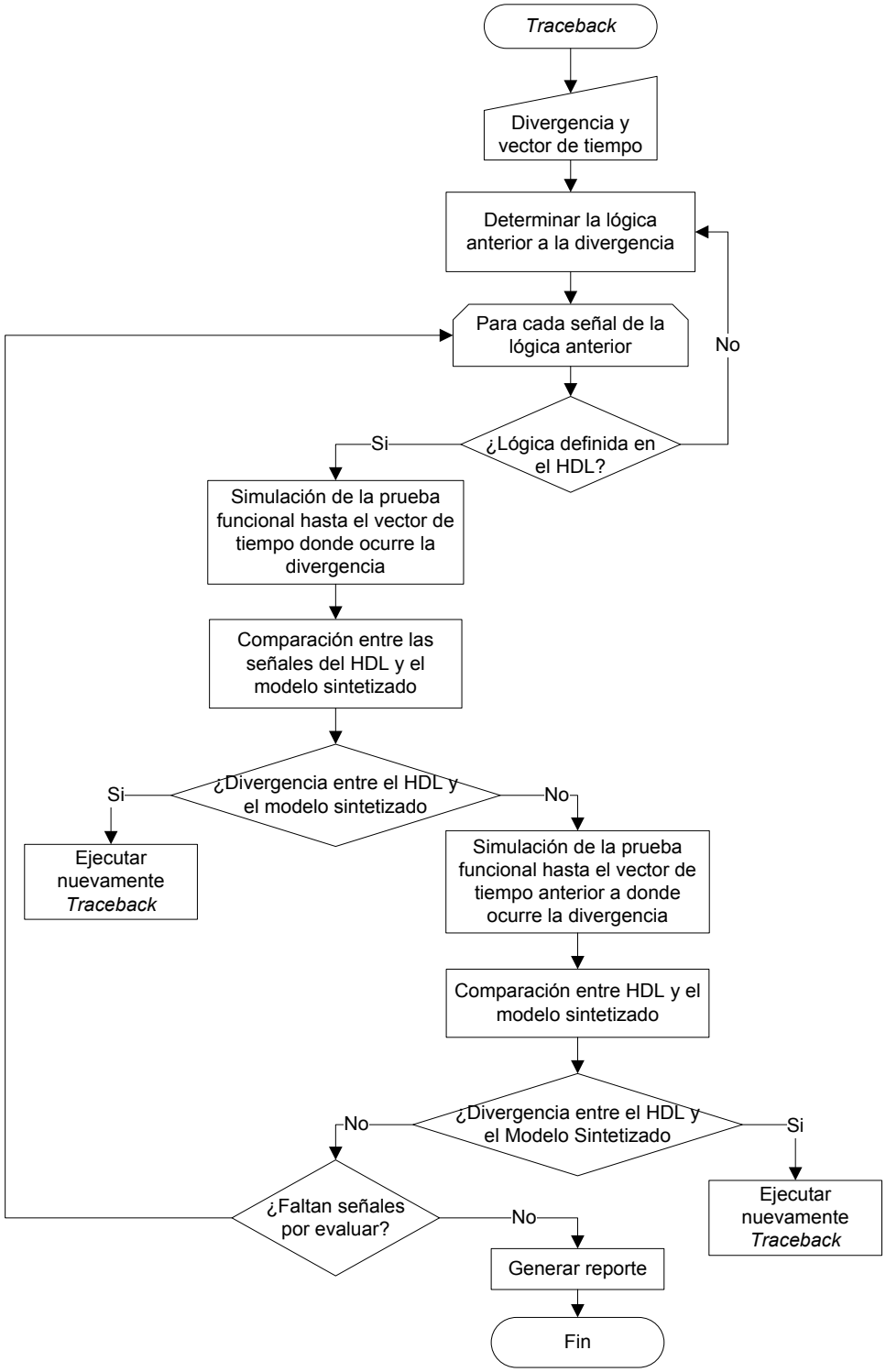


Figura 5-1 Algoritmo de *traceback*

Una vez finalizada el análisis de todas las señales anteriores involucradas en la divergencia se procede a generar un reporte donde se indique el camino que sigue la falla junto con los vectores de tiempo.

La principal ventaja de este algoritmo es que cuenta con información de análisis previos que servirían como referencia y permitirán prever posibles problemas, ya que los ingenieros al realizar manualmente el aislamiento de fallas en los modelo sintetizados ejecutan los pasos definidos en el algoritmo de la Figura 5-1. Sin embargo, el *traceback* presenta varias desventajas:

- No todas las señales definidas en el modelo sintetizado se encuentran especificadas en el HDL, por lo tanto no es posible realizar la comparación entre los valores de las señales observadas en el modelo sintetizado y el HDL. Para solucionar este inconveniente se debe repetir el proceso de búsqueda de señales anteriores a la señal con divergencia, aumentando así el tiempo y la lógica involucrada en el análisis.
- Para buscar en elementos de memoria posibles divergencias en estados anteriores se deben efectuar más simulaciones de la prueba funcional sobre el modelo sintetizado; esto hace que sean necesarios más recursos computacionales, los cuales son limitados.
- El *traceback*, como se explicó en la sección 4.2.1, es un algoritmo que determina el origen de una divergencia mediante el rastreo de la lógica anterior al nodo donde es observada la falla. Usualmente, una divergencia puede seguir varios caminos desde su origen para llegar a ser observada en un puerto de salida del modelo, por esta razón al realizar el rastreo de la lógica anterior se deben analizar todas las posibles señales que pueden producir un error, como señales de control, relojes o datos. Por ejemplo, si la compuerta que se encuentra antes del puerto de salida es una compuerta and, se deben verificar que el valor que se tiene en cada una de las entradas sea el correcto; en el caso de un multiplexor de 2 entradas el algoritmo debe de analizar la señal de selección y las dos entradas del multiplexor. Examinar las señales de cada compuerta provoca que el algoritmo deba analizar grandes conos de lógica asociados a una falla y que aumente el número de simulaciones producto de las sucesivas invocaciones del algoritmo.

5.1.2 Búsqueda modular

La búsqueda modular pretende analizar los circuitos de un módulo a partir de la observación de las entradas y/o salidas de los sub-módulos definidos para este. En la Figura 5-2 se muestra la representación gráfica del algoritmo de búsqueda modular.

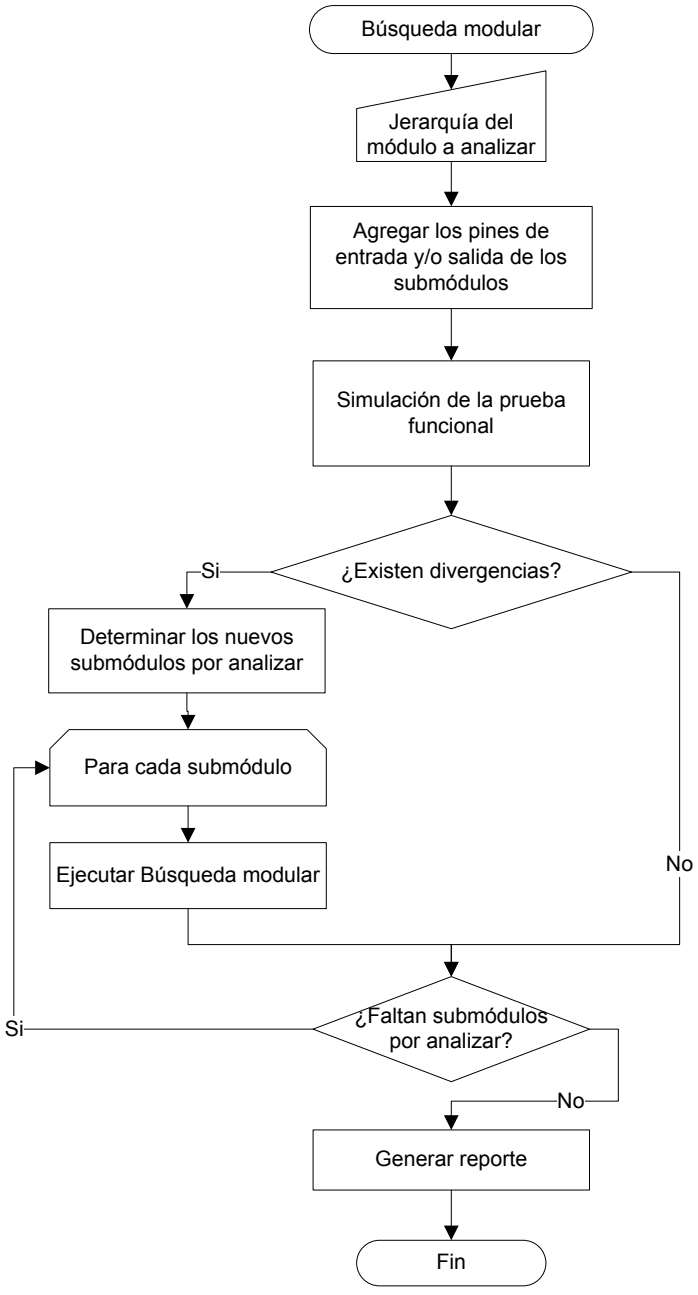


Figura 5-2 Algoritmo de búsqueda modular

El algoritmo planteando para la búsqueda modular recibe como parámetro de entrada la jerarquía del módulo que se pretende analizar; con base en este parámetro se buscan los puertos de entrada y/o salida de cada uno de los sub-módulos definidos para la jerarquía especificada. Estos puertos son agregados a la simulación de la prueba funcional como puntos de observación, por lo tanto, la herramienta de simulación va a detectar y detener la simulación cuando se propague la primera divergencia a alguno de los puertos de salida. A partir de este punto se buscan los sub-módulos de donde proviene el error y para cada uno de ellos se ejecuta de nuevo el algoritmo de búsqueda modular, usando como parámetro de entrada la jerarquía de lo sub módulo que presentan errores. En el caso de que ya no se detecten errores en ninguno de los sub-módulos analizados se procede a generar un informe con los resultados de la búsqueda.

Una de las principales ventajas de este algoritmo es el hecho de que la simulación de la prueba funcional se detiene en el momento en que se encuentra la primera divergencia, ya que en los casos en que existe retroalimentación de la lógica, el aislamiento de la falla iniciaría antes de que la divergencia sea realimentada. De esta forma el número de simulaciones que se realizan para aislar una falla deberían ser menor.

La desventaja del algoritmo de búsqueda modular se presenta cuando la verificación lógica se realice sobre un modelo sintetizado plano. Como se muestra en la Figura 5-3, el modelo sintetizado plano, aunque es poco común, es aquel modelo que está definido por pocos módulos que contiene mucha lógica asociada. Por lo tanto, cuando el algoritmo de búsqueda modular localiza el menor bloque de lógica definido en el modelo donde se encuentra la falla, se va a requerir más análisis de la lógica interna del módulo que para el caso de un modelo sintetizado modularmente.

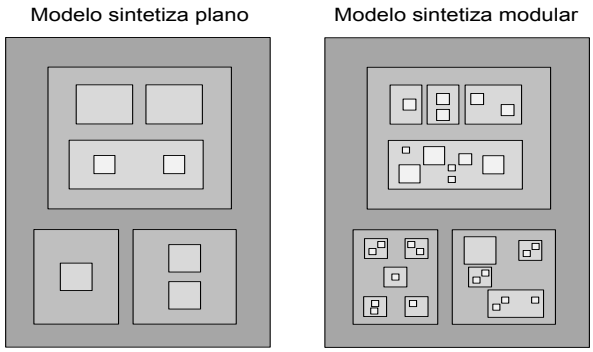


Figura 5-3 Modelo sintetizado plano y modulo sintetizado modular

5.1.3 Selección final

Para elegir el algoritmo que se implementó como solución al problema planteado en este proyecto se tomaron en cuenta los siguientes aspectos:

- La búsqueda modular permite que la localización de fallas se focalice en el módulo de lógica donde se origina el error; en el *traceback* no necesariamente sucede esto, ya que se debe pasar por varios módulos antes de encontrar el que origina la falla. Por lo tanto, la búsqueda modular es más eficiente y requiere de menos simulaciones que el *traceback*.
- Aunque los dos algoritmos utilizan la recursión como parte de la solución, la búsqueda modular utiliza un algoritmo más sencillo que el *traceback*.
- La búsqueda modular resulta ser una solución más general al problema de verificación lógica ya que se enfoca en localizar la primera falla que se detecta en el modelo; en cambio el *traceback* se utiliza para localizar una falla en específico observada en un puerto de salida del modelo.
- Aunque el *traceback* es un proceso ya probado por los ingenieros, la búsqueda modular parece ser una solución viable que permitiría localizar fallas de en menor tiempo.

Es en base a las consideraciones anteriores que se propone el algoritmo de búsqueda modular, dadas sus ventajas sobre el algoritmo de *traceback*.

Como se muestra en la Figura 5-4, el algoritmo de búsqueda modular tendrá como parámetros de entrada: la prueba funcional donde se han detectado las divergencias y el modelo sintetizado del circuito integrado. Con estos datos el algoritmo debe estar en la capacidad de localizar el vector de tiempo y el módulo de menor jerarquía definido en el modelo sintetizado donde se inicia la divergencia.

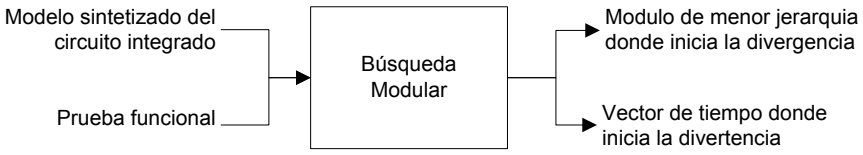


Figura 5-4 Parámetros de entrada y salida esperados del algoritmo de búsqueda modular

5.2 Evaluación de los algoritmos

La evaluación de los algoritmos implementados para la herramienta de software se realizará sobre el modelo sintetizado y el HDL de un circuito DMI (*Direct Media Interface),* el cual es parte del diseño de un microprocesador de la familia INTEL conocida como LCIA (*Low Cost Intel Architecture*). Además, se utilizará una prueba funcional enfocada en ejercitar la lógica del módulo DMI del circuito.

5.2.1 Pre-proceso

El algoritmo que encuentra las jerarquías y módulos definidos en el modelo sintetizado y el algoritmo de búsqueda de jerarquías definidas en el HDL serán evaluados por sus archivos de salida, los cuales deben contener el conjunto de jerarquías definidas para el modelo de prueba.

El algoritmo de comparación y traducción de jerarquías calcula la distancia de Levenshtein para determinar el porcentaje de similitud entre dos jerarquías y es con base en ese porcentaje se decide si se inicia la búsqueda de una traducción para el nombre de la jerarquía. La ejecución del proceso de traducción aumenta el tiempo de ejecución del algoritmo, por lo tanto, para el algoritmo de comparación y traducción de jerarquías se evaluará el porcentaje de similitud entre jerarquías que permita realizar la traducción de todos los nombres satisfactoriamente en el menor tiempo de ejecución. Además se evaluará el archivo de salida donde se encuentra la traducción entre el nombre de las jerarquías en el modelo sintetizado y el HDL.

5.2.2 Búsqueda de errores

La verificación lógica del modelo del DMI fue realizada y registrada previamente por ingenieros de la compañía, por lo que se cuenta con los datos necesarios para realizar la comparación entre el tiempo empleado por la herramienta implementada y los ingenieros. El algoritmo de búsqueda de errores se evaluará con base en:

- La correcta identifica de los módulos de menor jerarquía definidos en el modelo sintetizado donde se originan las divergencias del modelo.
- El tiempo de simulación que requiere la ejecución del algoritmo.
- El tiempo requerido por el ingeniero para corregir los errores en el modelo después de ejecutar el algoritmo e identificarse los módulos donde provienen los errores.

5.3 Pre-proceso

El algoritmo de búsqueda modular utiliza un pre-proceso encargado de ejecutar las fases o etapas de algoritmo que requieren de largos tiempos de simulación y son invariables para el modulo sintetizado de un circuito, con el fin de agilizar el proceso cuando se realizan simulaciones sucesivas. Como se muestra en la Figura 5-5, los algoritmos seleccionados para ser parte del pre-proceso son:

- Extracción de las jerarquías definidas en el HDL.
- Extracción de las jerarquías y módulos definidos en el modelo sintetizado, junto con los puertos de entrada y salida definidos para cada módulo.
- Generación de una tabla de traducción entre el nombre de una jerarquía en el HDL y el nombre en el modelo sintetizado.
- Listado de los pines de entrada y/o salida definidos para un sub módulo.

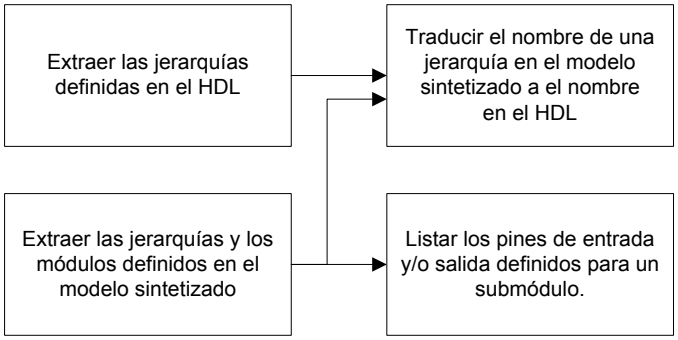


Figura 5-5 Algoritmos involucrados en el pre-proceso

A continuación se explica por medio de pseudocódigo cada uno de los algoritmos definidos en la Figura 5-5.

5.3.1 Jerarquías y módulos definidos en el modelo sintetizado

El algoritmo para extraer las jerarquías y módulos definidos en el modelo sintetizado del circuito fue implementado en el lenguaje de programación llamado TCL "*Tool Command Language"*, ya que este es el único lenguaje de programación que permite la comunicación con la herramienta de síntesis de modelos.

El algoritmo descrito por medio de la Figura 5-6, está compuesto por 2 procesos: instDistb y getChldInfo. El proceso llamado instDistb es el principal y se caracteriza por su ejecución recursivamente; tiene como parámetros de entrada una jerarquía nombrada como la variable padre y un módulo nombrado como hijo.

El proceso instDistb busca por medio de la subrutina getChldInfo si un módulo se encuentra definido para una jerarquía en específico; si este es el caso se procede a buscar las instancias definidas para este módulo. En un módulo puede estar instanciados tanto sub-módulos como compuertas lógicas, por lo que se debe identificar únicamente los sub-módulos para iterar sobre estos.

El nombre y la jerarquía donde se encuentra instanciado un módulo es agregado al archivo jerarquíasModelo.list para futura referencia. Además, para cada módulo encontrado se genera un archivo nombrado como /jerarquía/módulo donde se guardan los puertos de entrada y salida definidos.

```
instDistrib ( padre hijo ) {
    @instancia = getChldInfo ( padre hijo )
    Si ( $instancia(tipo) == "INSTANCE" ) {
        $nombre_instancia = decodificar $instancia(nombre)
        @módulo = modview find $nombre_instancia
        Si ( $módulo(tipo) == "MODULE" ) {
            Agregar $nombre_instancia al archivo jerarquías.list

            Buscar los puertos de entrada y salida en @módulo
            Crear archivo con el nombre $nombre_instancia
            Agregar puertos de entrada y salida al archivo

            Para cada $elemento de $instancia(instancia) {
                instDistrib ( $instancia(instancia) $elemento )
            }
        }
    }
}

getChldInfo ( padre hijo ) {
    $jerarquía = concatenar $padre y $hijo
    @info = ln $jerarquía
    @info_jerarquía = extraer información útil de @info
    retorna ( @info_jerarquía )
}
```

Figura 5-6 Algoritmo de búsqueda de puertos de entrada y salida módulos

En el algoritmo de la Figura 5-6 se utilizan dos instrucciones propias de la herramienta de síntesis: modview find y ln. La instrucción modview find provee toda la información referente a un módulo y la instrucción ln proporciona la información definida para una jerarquía.

5.3.2 Jerarquías definidas en el HDL

El algoritmo de la Figura 5-7 es el encargado de buscar las jerarquías definidas en el HDL para un circuito. El algoritmo utiliza un archivo en formato fsdb o vcd, el cual debe contener al menos las señales modelo sintetizado. En el caso de que el archivo recibido sea un fsdb, el algoritmo extraerá únicamente las señales contenidas en el modelo del circuito y convertirá el archivo de formato del fsdb a vcd.

Se utiliza el formato del vcd ya que este contiene en su encabezo un listado de las señales del circuito organizadas por módulo y jerarquía. Para recorrer el archivo vcd rápidamente e identificar las jerarquías definidas para un circuito se utilizó el lenguaje de programación Perl, ya que este permite manipular estructuras de datos complejas. El resultado de este algoritmo es un archivo llamado jerarquíasHDL.list que contiene las jerarquías definidas para el circuito modelado.

```
HDL_hierarchies ( fsdb/vcd, modelo ) {
        Si se recibe un fsdb{
                $modelo_fsdb = Extraer del $fsdb las señales del $modelo
                $vcd = Convertir el $modelo_fsdb en vcd
        }

        @vcd_file = abrir el $vcd
        Para cada $línea de @vcd_file {
                Si la $línea contiene "\$scope"{
                        Si $línea contiene "module" or "begin"{
                                Agregar a @jerarquía el nombre del módulo
                        }

                        Aumentar en 1 la variable $nivel_jerárquico
                }
                O si $linea contiene "\$upscope"{
                        $jerarquía = concatenar @jerarquía por medio de "/"
                        Agregar $jerarquía al archivo de salida
                        Expulsar el último elemento de @jerarquía
                        Decremento en 1 la variable $nivel_jerárquico
                }
        }
}
```

Figura 5-7 Algoritmo para extraer las jerarquías definidas en el HDL

5.3.3 Comparación y traducción de jerarquías

El nombre de una jerarquía en el modelo sintetizado no siempre es equivalente al nombre dado en el HDL, por lo que es necesario realizar una comparación y traducción de los nombres. Para realizar la comparación el algoritmo de la Figura 5-8 utiliza los 2 archivos generados por los algoritmos anteriores: jerarquíasModelo.list y jerarquíasHDL.list.

El procedimiento compara el nombre de cada jerarquía de una lista contra todas las jerarquías de la otra lista. Sin embargo, este proceso es ineficiente y para agilizarlo se procede a ordenar alfabéticamente las listas de nombres y se pretende utilizar la traducción realizada en la comparación anteriormente. Además se hace uso del algoritmo de Levenshtein antes de iniciar la búsqueda de la traducción para un nombre. En el caso de que no se encuentre traducción para una jerarquía, el nombre de esta se guarda en el archivo faltantes.cmp para posteriormente realizar un búsqueda manual.

```
comparación_jerarquías ( jerarquías_modelo, jerarquías_HDL ) {
    Ordenar alfabéticamente @jerarquías_modelo y @jerarquías_HDL

    Para cada $jerar_modelo de @jerarquías_modelo {

        Para cada $jerar_HDL de @jerarquías_HDL {

            Se verifica si la sustitución realizada en la jerarquía anterior pueda ser aplicada a esta jerarquía
            $LD = levenshtein($jerar_modelo, $jerar_HDL);

            Si ( $LD igual a 0 ) {
                La jerarquía en el HDL y en el modelo sintetizado tiene el mismo nombre
                Se guarda la traducción en el archivo de salida jerarquías.cmp
                Se sale del ciclo
            }

            $parámetro = (longitud de jerar_modelo - $LD) / jerar_HDL
            Si ( $parámetro mayor a 0.85 ){
                $bandera_traducción = traducción_jerarquías( $jerar_modelo, $jerar_HDL );
                Si ($bandera_traducción igual 1) {
                    Se guarda la traducción en el archivo de salida jerarquías.cmp
                    Se sale del ciclo
                }
            }
        }

        Las jerarquías no traducidas se guardan en el archivo faltantes.cmp
    }
}
```

Figura 5-8 Algoritmo de comparación y traducción de jerarquías

Al comparar el nombre de dos jerarquías únicamente se puede determinar si estas son iguales o no; por tanto se hace uso del algoritmo de Levenshtein el cual contabiliza el número necesario de caracteres que se deben sustituir, insertar y eliminar para lograr que el nombre de jerarquía sea igual a al otro. Determinado la distancia de Levenshtein se puede identificar en qué porcentaje se parece un nombre a otro, en los casos que este sea mayor a un 85%, se procede a buscar una traducción.

```
levenshtein ( cadena_caracteres_0, cadena_caracteres_1  ) {
    $tam_0 = número de caracteres de la cadena 0
    $tam_1 = número de caracteres de la cadena 1

    Se crea la @matriz de tamaño ($tam_0+1) por ($tam_1+1)

    # Inicializar la @matriz
    Para ($i = 1;  $i <= $tam_1;  ++$i) {
        Para ($j = 1;  $j <= $tam_2;  ++$j) {
            $matriz{$i}{$j} = 0;
            $matriz{0}{$j} = $j;
        }
        $matriz{$i}{0} = $i;
    }

    @cadena_0 = separar los caracteres de cadena_caracteres_0
    @cadena_1 = separar los caracteres de cadena_caracteres_1

    Para ($i = 1;  $i <= $tam_1;  ++$i) {
        Para ($j = 1;  $j <= $tam_2;  ++$j) {
            $costo = si ($cadena_0[$i-1] igual $cadena_1[$j-1]) se asigna el valor de 0 sino 1
            $0 = ( matriz{$i-1}{$j} + 1 ) - el valor de la celda anterior mas 1 -
            $1 = ( $mat{$i}{$j-1} + 1 ) - el valor de la celda a la izquierda mas 1 -
            $2 = ( $mat{$i-1}{$j-1} + $costo ) - el valor de la celda diagonal superior mas $costo -

            $matriz[$i:$j] = al mínimo valor de las siguientes variables $0, $1 y $2
        }
    }

    retorna ( $matriz{$tam_0}{$tam_1} )
}
```

Figura 5-9 Algoritmo para calcular la distancia de Levenshtein

Cuando se determina que dos nombres son significativamente similares se utiliza el algoritmo de traducción de jerarquías de la Figura 5-10, el cuál realiza una comparación carácter por carácter de las jerarquías. Cuando se encuentran caracteres diferentes se procede a verificar si aplicando sustitución de caracteres como "/" o "_", o agregando caracteres como "[" o "]" se puede realizar una traducción satisfactoria.

```
traducción_jerarquías( $jerar_modelo, $jerar_HDL ){
        @cadena_0 = separar los caracteres de $jerar_modelo
        @cadena_1 = separar los caracteres de $jerar_HDL

        #Comparación carácter por carácter
        $tamaño = # de caracteres de $jerar_modelo
        Para $i = 0 hasta $tamaño {
                Si ($cadena_0[$i] diferente $cadena_1[$i]){
                        Analizar posibles casos de traducción
                        Si existe retorna $traducción y sino retorna en conjunto vacío
                }
        }
}
```

Figura 5-10 Algoritmo para la traducción de jerarquías

5.3.4 Pines definidos para los módulos

En la simulación de la prueba funcional se observan los pines de entrada y de salida de los sub-módulos definidos para el módulo en análisis. Los pines que se analizarán se determinan por medio del parámetro de configuración que se dé al algoritmo de la Figura 5-11, ya que dependiendo de lo que esté buscando el usuario se pueden utilizar únicamente los pines de entrada, los pines de salida o ambos.

```
pinList ( módulo, -inputs, -outputs ) {
    Para cada $archivo generado en con el algoritmo de búsqueda de puertos {
        Abrir el $archivo
        Si ( -inputs activo ){
            Identificar los pines de entrada en el $archivo
            Traducir la jerarquía del pin en el modelo a la del HDL
            Agregar la señal y la traducción al $modulo .appp
        }
        Si ( -outputs activo ){
            Identificar los pines de salida en el $archivo
            Traducir la jerarquía del pin en el modelo a la del HDL
            Agregar la señal y la traducción al $modulo .appp
        }
    }
}
```

Figura 5-11 Algoritmo para listar los pines de los módulos

El algoritmo, escrito en el lenguaje Perl, modifica las señales para cumplir con el formato requerido por la herramienta de síntesis y identifica la traducción del nombre de la señal entre el modelo sintetizado y en el HDL. Toda la información recopilada por el algoritmo se guarda en un archivo que lleva el nombre de la jerarquía más el módulo.

5.4 Búsqueda de errores

La búsqueda de errores es la herramienta que permite rastrear y aislar una falla en el modelo sintetizado. Como se muestra en la Figura 5-12, el algoritmo de simulación se puede ejecutar en 3 modos:

- Simulación inicial: se utiliza para identificar si una prueba funcional detecta alguna falla sobre el modelo. Los pines analizados son los únicamente los principales del modelo y están configurados en la herramienta para ser observados por defecto. Este modo únicamente carga el módelo sintetizado del circuito integrado y simula la prueba funcional, los parámetros de entrada que requiere son el -fsdb <>, el -m <> que identifica el modelo que se va a utilizar y –initial que señala el modo de uso.
- Simulación inicial para una jerarquía: se utiliza para identificar si una prueba funcional detecta alguna falla sobre una jerarquía en específico. En este proceso se deben agregar los pines de la jerarquía por analizar como puntos de observación en la herramienta de simulación. Los parámetros de entrada que requiere son el -fsdb <>, el -m <>, –initial y –hierarchy <> que identifica la jerarquía que se va a analizar.
- Simulación recursiva: se utiliza para hacer la búsqueda de los errores entre las jerarquías del modelo sintetizado. Se le llama recursivo porque conforme encuentre errores se vuelve a ejecutar el proceso para profundizar en los sub-módulos que presenta posibles errores. Los parámetros de entrada que requiere son el -fsdb <> y el -m <>; el parámetro –v es opcional y se usa para indicarle a la herramienta de simulación a partir de que vector se deben observar los errores; el parámetro –hierarchy <> se utiliza para indicar a la búsqueda de errores si el análisis se debe hacer para una jerarquía en específico, en caso de que no aparezca se hace sobre el modelo completo.

El algoritmo de simulación se programó en el lenguaje conocido como csh o *C shell*, ya que permite llamar las subrutinas escritas en otros lenguajes de forma rápida y sencilla, además, permite interactuar en la consola de Unix con el usuario.

```
búsquedaErrores ( -fsdb <> -v <> -m <> -hierarchy <> -initial -help ){
    Si -help esta activo {
        Despliega instrucciones que indican cómo utilizar el programa
    }

    Si –initial está activo {
        Si hay una jerarquía especificada en -hierarchy < > {
            Pre-simulación (jerarquía)
            Simulación ( -fsdb < >, -m <> )
        }
        Sino {
            Simulación ( -fsdb < >, -m <> )
        }
    }
    Si –initial no está activo {
        SimulacionesRecursivas ( jerarquía )
        Generar reporte
    }
}

SimulacionesRecursivas ( jerarquía ) {
    Pre-simulación ( jerarquía )
    Simulación ( -fsdb < >, -m <> )
    @divergencias = Post-simulación ( jerarquía )
    Para cada divergencia {
        Determinar los sub-módulos que se deben analizar
        Para cada submódulo {
            nueva_jerarquía = concatenar jerarquía y el submódulo
            SimulacionesRecursivas ( jerarquía )
        }
    }
}
```

Figura 5-12 Algoritmo para la simulación de pruebas funcionales

El algoritmo de búsquedaErrores invoca a otros tres algoritmos: pre-simulación, simulación y post-simulación. A continuación se explica en detalla cada uno de ellos.

5.4.1.1 Pre-simulación

El proceso de pre-simulación es llamado por el algoritmo de búsquedaErrores cuando se va a examinar una jerarquía en específico. Se divide en 3 etapas, mostradas en la Figura 5-13, las cuales se aseguran de proveer a la herramienta de simulación todos los datos necesarios para analizar los sub-módulos de una jerarquía.

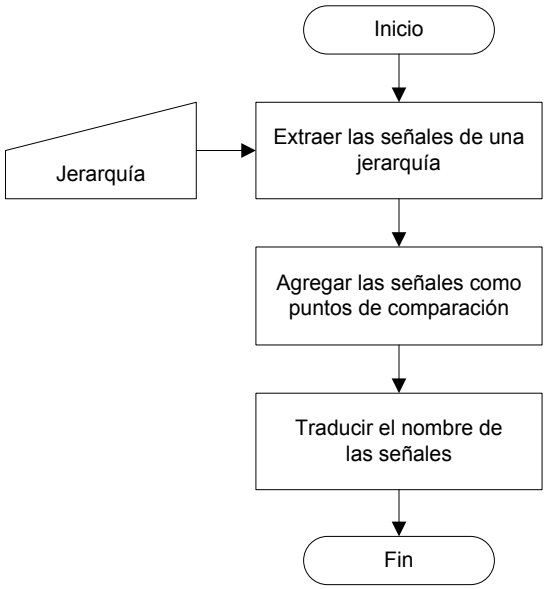


Figura 5-13 Etapas del algoritmo de pre-simulación

- Extraer las señales de una jerarquía: consiste en determinar los sub-módulos definidos para una jerarquía y buscar en los archivos generados por el algoritmo Figura 5-11 las señales que se van a observar en la simulación con el fin de guardarlas en el archivo modelo.appp.
- Agregar las señales como puntos de observación: se encarga de localizar en el modelo el número que identifica cada señal del archivo modelo.appp y agregarla al archivo modelo.ppp.
- Traducir el nombre de la señales: agrega al archivo modelo.scp las señales del archivo modelo.appp, solo que esta vez en el archivo debe aparecer la traducción de los nombres entre el modelo y el HDL. La traducción es utilizada por la herramienta de simulación para comparar los valores adquiridos por los pines en la simulación con los valores guardados en el fsdb.

El seudocódigo utilizado para el proceso de pre-simulación se muestra en la Figura 5-14 y fue programado utilizando Perl como lenguaje de programación. Cuenta con 3 subrutinas que equivalen a las 3 etapas explicadas por medio de la Figura 5-13.

```
Pre-simulación ( jerarquía ) {
      signalExtraction ( jerarquía_por_encontrar, modelo )
      cmpFile ( $módulo )
      spcFile ( $módulo )
}

signalExtraction ( jerarquía_por_encontrar, modelo ) {
     @jerarquías = Abrir el archivo jerarquías.list
     Para cada $jerarquía de @jerarquía {
          $num_submódulos0 = sub-módulos de $jerarquía_por_encontrar
          $num_submódulos1 = sub-módulos de $jerarquía + 1
          Si ( $num_submódulos0 es igual a $num_submódulos1 ) {
               Comparar los nombres de las jerarquías base
               Si los nombres de las jerarquías base son iguales {
                    Abrir el archivo $jerarquía generado por el algoritmo para listar los pines de los módulos
                    Agregar las señales encontradas al archivo $modelo.appp
               }
          }
     }
}

cmpFile (módulo ) {
      $archivo = abrir el archivo $modulo.appp
      Para cada $señal del $archivo {
           Localizar el número de identificación de la $señal en el modelo
           Agregar la $señal y el número de identificación al archivo $modelo.ppp
      }
}

spcFile ( módulo ) {
          $archivo = abrir el archivo $módulo.appp
          Para cada $señal del $archivo {
                    Identificar la jerarquía de la $señal
                    Buscar la traducción de la jerarquía en el archivo jerarquías.cmp
                    Agregar la $señal y la traducción de la $señal al archivo $módulo.spc
          }
}
```

Figura 5-14 Algoritmo del pre-proceso

5.4.1.2 Simulación

La simulación se encarga de verificar la existencia de los archivos necesarios para simulación y de cargar el modelo sintetizado para ejecutar la prueba funcional. La prueba funcional es simulada por la herramienta de síntesis y puede tardar un tiempo variado dependiendo de la complejidad de la prueba y del tamaño del modelo.

```
Simulación ( fsdb, modelo ) {
        Verificar la existencia de los archivos del modelo
        Agregar los archivos generados como parámetros de configuración
        Cargar el modelo del circuito sintetizado
        Simular la prueba funcional en el modelo del circuito sintetizado
}
```

Figura 5-15 Algoritmo para simular la prueba funcional en el modelo del circuito

5.4.2 Post-simulación

El proceso de post-simulación se encarga de analizar los resultados generados en la simulación, entre lo que se encuentran posibles errores en el proceso y las fallas observadas en los pines agregados como puntos de comparación. Cuando se encuentran divergencias se buscan y analizan los vectores y jerarquías donde se detectaron, y así definir los posibles sub-módulos donde se originan las fallas en el modelo.

```
Post-simulación ( jerarquía ) {
    Verificar que la simulación terminar con éxito
    @divergencias = abrir el archivo llamado $jerarquía.diff

    Para cada $divergencia de @divergencias {
        Determinar la jerarquía de $divergencia
        Determinar el vector donde ocurrió la $divergencia
        @errores = agregar la jerarquía y el vector donde ocurre la $divergencia
    }

    Retorna ( @errores )
}
```

Figura 5-16 Algoritmo para analizar las divergencias encontradas en la simulación

CAPÍTULO 6: ANÁLISIS DE RESULTADOS

Se realizaron pruebas a la herramienta de software desarrollada con el fin de comprobar el correcto funcionamiento y determinar si su utilización acelera el aislamiento de fallas en los modelos sintetizados de circuitos integrados. Los resultados más representativos se explican y analizan en este capítulo.

Para una mejor comprensión y análisis los resultados obtenidos se encuentran divididos en tres secciones: la primera muestra las pruebas realizadas a las partes que componen el pre-proceso, la segunda ejemplifica el funcionamiento la búsqueda modular; y la tercera sección presenta un análisis comparativo del tiempo empleado en la verificación de un modelo utilizando la herramienta desarrollada y realizándolo de forma manual.

6.1 Pre-proceso

En esta sección se exponen tres de las pruebas realizadas al pre-proceso, la primera pretende demostrar la funcionalidad del algoritmo de Levenshtein, la segunda procura encontrar el punto de operación donde el algoritmo de comparación y traducción de jerarquías tiene su mejor desempeño y la última demuestra el funcionamiento del pre-proceso en su totalidad.

6.1.1 Algoritmo de Levenshtein

El algoritmo de búsqueda difusa de cadenas implementado en la herramienta de software desarrollada fue el algoritmo de Levenshtein. Este algoritmo calcula la distancia de Levenshtein, la cual especifica el número de operaciones necesarias para convertir una cadena de caracteres en otra.

En la

Tabla 6-1 se muestran tres de los casos utilizados para comprobar el correcto funcionamiento del algoritmo de Levenshtein. En el caso #1, la distancia de Levenshtein es de 3 lo que corresponde a sustituir dos de los caracteres “_” por “[ ]” y agregar un “/”. El grado de

similitud entre las dos cadenas de caracteres del caso #1 es de un 94%. Este valor se calcula de acuerdo a la fórmula A.2-1 mostrada en el apéndice A2.

Tabla 6-1 Distancia de Levenshtein para casos de prueba

Caso # 1	
Cadena de caracteres # 1	/dmi_cck_CLK/dmi_cck_p_ulm/lane_lp_1_lane_clk_llx
Cadena de caracteres # 2	/dmi_cck_CLK/dmi_cck_p_ulm/lane_lp[1]/lane_clk_llx
Distancia de Levenshtein	3
Similitud	0.94
Caso # 2	
Cadena de caracteres # 1	/dft_unit_control_1x/b1_0_sync_rst_reset_to_flopspre_i
Cadena de caracteres # 2	/dft_unit_ control_1x/b1_0_sync_rst_reset_to_flops
Distancia de Levenshtein	5
Similitud	0.91
Caso # 3	
Cadena de caracteres # 1	/dft_unit_control_1x/b1_0_sync_rst_reset_to_flopspre_i
Cadena de caracteres # 2	/dft_unit_control_1x/b1[0]
Distancia de Levenshtein	31
Similitud	0.43

El caso #2, la distancia de Levenshtein es de 5 (equivalente a los cinco caracteres que deben ser agregados) y la similitud entre las cadenas es de un 91%. En el caso #3, son necesarias 2 sustituciones y agregar 29 nuevos caracteres y la similitud entre ambas cadenas es de un 43%.

La principal características de este algoritmo es que permite realizar comparaciones entre cadenas de caracteres de diferentes longitudes, ya que considera la inserción de caracteres. Por ejemplo, en el caso #1 la falta de un "/" no altera el cálculo de la similitud entre las cadenas, como si sucedería si se realiza una comparación carácter por carácter.

6.1.2 Comparación y traducción entre jerarquías

El proceso de comparar jerarquías hace uso del algoritmo de Levenshtein para determinar la similitud entre las dos cadenas de caracteres. A partir de este grado de similitud se

determina si se inicia una comparación entre los caracteres de la jerarquía. El algoritmo de comparación determina los caracteres que no son semejantes y busca una traducción para la jerarquía entre las previamente definidas en el algoritmo.

6.1.2.1 Similitud entre las jerarquías

Realizar las traducción de miles de jerarquías consume mucho tiempo de simulación y puede producir retrasos en los procesos subsiguientes, por lo tanto es necesario determinar el porcentaje de similitud entre dos cadenas que permite realizar la comparación y traducción de jerarquías en el menor tiempo. Para esto se procedió a ejecutar el algoritmo de comparación y traducción de jerarquías modificando el requerimiento de similitud entre cadenas, el cual desencadena el proceso de comparación y traducción de jerarquías.

Para la prueba el parámetro de similitud entre las dos jerarquías se varió entre el 70% y el 100%, en intervalos de 2%, y el número de jerarquías analizadas fue de 1401. El resumen de los resultados obtenidos se presentan en la Tabla 6-2 y el listado completo de los datos se presentan en el apéndice A.3, Tabla A.3-1.

Tabla 6-2 Comparación y traducción de las jerarquías para varios grados de similitud

Porcentaje de similitud	82%	84%	86%	88%	90%	92%	94%	96%	98%	100%
Comparaciones realizadas	1541	1436	1165	927	696	586	321	190	75	0
Comparaciones exitosas	290	309	326	332	332	326	292	182	68	0
Comparaciones insatisfactorias	1251	1127	839	595	364	260	29	8	7	0
Jerarquías traducidas	1378	1378	1378	1378	1378	1358	1312	1205	1065	386
Jerarquías sin traducción	23	23	23	23	23	43	89	196	336	1015
Tiempo de ejecución (min)	36.4	36.0	36.1	36.4	38.0	41.4	99.3	265.5	448.1	374.6

Con base en los datos presentados en la Tabla 6-2 se gráfica en la Figura 6-1 el número de transacciones ejecutadas por el algoritmo de comparación y traducción de jerarquías para los diferentes porcentajes de similitud entre las jerarquías analizados.

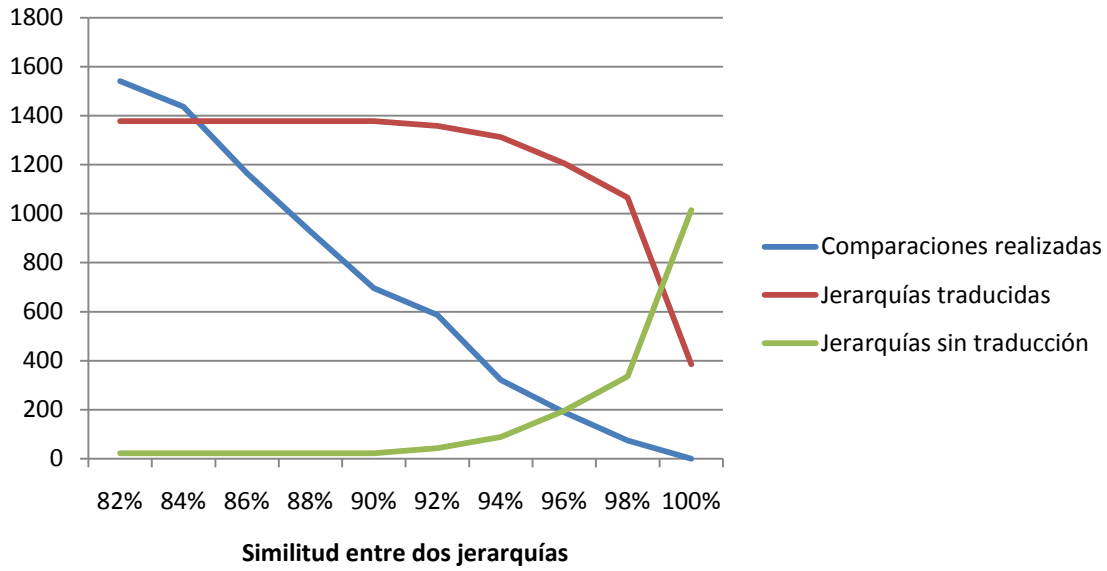


Figura 6-1 Relación entre el porcentaje de similitud dos jerarquías y los procesos ejecutados por el algoritmo de comparación y traducción de jerarquías.
FUENTE: Tabla 6-2

La gráfica en la Figura 6-2 presenta el tiempo de ejecución que requirió el algoritmo de comparación y traducción de jerarquías para diferentes grados de similitud entre las jerarquías analizadas.

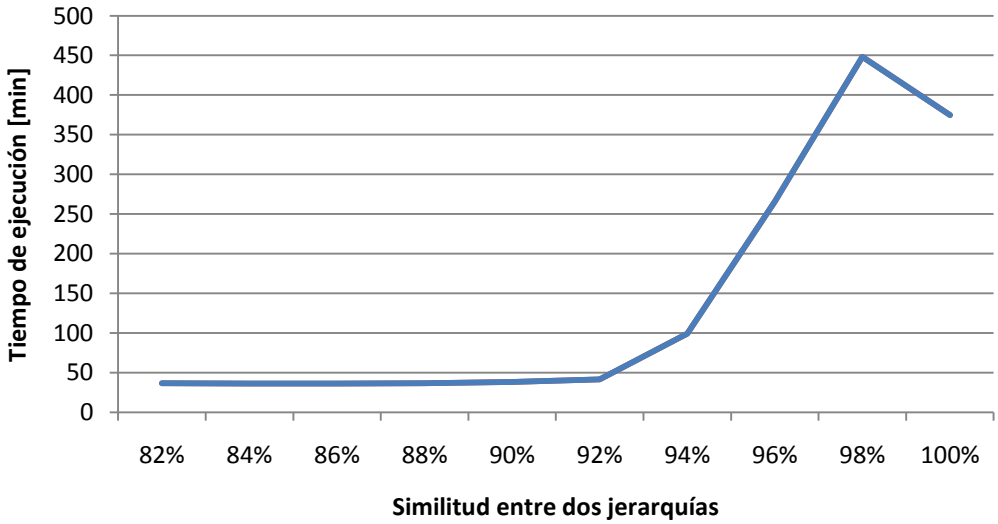


Figura 6-2 Relación entre el porcentaje de similitud dos jerarquías y el tiempo de ejecución del algoritmo de comparación y traducción de jerarquías
FUENTE: Tabla 6-2

Con base en los gráficos de la Figura 6-1 y Figura 6-2 se determinó que el algoritmo de comparación y traducción de jerarquías tiene su mejor desempeño cuando utiliza como condición para el inicio de la traducción una la distancia de Levenshtein de al menos un 84%-86%. Para los valores de 84%-85% no solo se tiene el menor tiempo de ejecución, si no que logra la mayor cantidad de traducciones con el menor número de comparaciones. Como se aprecia en la Tabla A.3-1, cuando la distancia de Levenshtein requerida para iniciar la traducción es menor al 80% se realizan comparaciones de innecesarias para lograr una misma cantidad de traducciones y cuando la distancia de Levenshtein es mayor al 90% se efectúan menos traducciones dado que se realizan menos comparaciones.

Estableciendo como condición para buscar la traducción de una jerarquía un porcentaje de similitud entre las dos jerarquías de un 85% se procede a ejecutar pruebas sobre los modelos de un circuito DMI y un circuito controlador de memoria.

6.1.2.2 Prueba en el modelo de un circuito DMI

La prueba realizada consistió en ejecutar el algoritmo de comparación de jerarquías sobre el modelo sintetizado de un circuito DMI, resultados se presentan en la Tabla 6-3. Las 1401 jerarquías definidas en el modelo sintetizado del circuito DMI se buscaron en las 5943 jerarquías definidas en el HDL; para el 98.3% de las jerarquías se encontró exitosamente una traducción.

Tabla 6-3 Resultados de la ejecución del algoritmo de comparación de jerarquías sobre el modelo sintetizado de un circuito DMI

Jerarquías definidas en el modelo	1401
Jerarquías definidas en el HDL	5943
Jerarquías traducidas	1378 (98.3%)
Jerarquías sin traducción	23 (1.6%)

Algunas de las 1378 traducciones realizadas por el algoritmo se muestran en la Tabla 6-4. En el caso #1 fue necesario agregar unos paréntesis cuadrados, en el segundo y tercer caso la traducción requirió de paréntesis cuadrados y algunos “/” para separar los niveles jerárquicos.

Tabla 6-4 Jerarquías traducidas en el modelo de un circuito DMI

Caso #1	
Modelo sintetizado	/dmi_cck/dmi_cck_p_ulm/lane_lp_0_lane_clk_mux
HDL	/dmi_cck/dmi_cck_p_ulm/lane_lp[0]/lane_clk_mux
Caso #2	
Modelo sintetizado	/dft_unit_controller/b1_0_sync_rst_reset_to_flops_pre_i
HDL	/dft_unit_controller/b1[0]/sync_rst_reset_to_flops_pre[i]
Caso #3	
Modelo sintetizado	/dmi_top/init16cmn_4/ltsctl_1_txrxafectl_1_rxsqulechexit_nlanes_0_rxsqulechexit
HDL	/dmi_top/init16cmn_4/ltsctl_1/txrxafectl_1/rxsqulechexit_nlanes[0]/rxsqulechexit

En la Tabla 6-5 se muestran dos jerarquías que no encontraron una traducción dentro de los parámetros establecidos en el algoritmo. En los dos casos, el nombre especificado en el HDL presenta una sección del nombre encerrada dentro de paréntesis cuadrados. Dado que este tipo de traducción no es común y la cantidad de caracteres dentro de los paréntesis puede variar se determinó que para estos casos debe realizarse una traducción manualmente. La traducción para una letra o un número dentro de paréntesis si se encuentra tipificada dentro del algoritmo.

Tabla 6-5 Jerarquías sin traducción en el modelo de un circuito DMI

Caso #1	
Modelo sintetizado	/dmi_top/dmi4ctrl/pcietl8dmis/fcregisters_fc_Latch_VC0_lcb_ClkFCCnsnVCFH_vc
HDL	/dmi_top/dmi4ctrl/pcietl8dmis/fcregisters/fc_Latch_VC0_lcb_ClkFCCnsnVCFH[vc]
Caso #2	
Modelo sintetizado	/dmi_top/dmi4ctrl/pcietl8dmis/data/buf_strm[1]_lcb_ClkBufAEnFH_stream
HDL	/dmi_top/dmi4ctrl/pcietl8dmis/data/buf_strm[1]/lcb_ClkBufAEnFH[stream]

6.1.2.3 Prueba en el modelo de un circuito controlador de memoria

En la Tabla 6-6 se presentan los resultados de la ejecución del algoritmo de comparación de jerarquías sobre el modelo sintetizado de un circuito controlador de memoria. En este caso de las 462 jerarquías definidas en el modelo sintetizado se tradujeron el 86.1%.

Tabla 6-6 Resultados de la ejecución del algoritmo de comparación de jerarquías sobre el modelo sintetizado de un circuito controlador de memoria

Jerarquías definidas en el modelo	462
Jerarquías definidas en el HDL	1550
Jerarquías traducidas	398 (86.1%)
Jerarquías sin traducción	64 (13.9%)

Las traducciones realizadas por el algoritmo en el modelo del circuito controlador de memoria son similares a las efectuadas en la prueba anterior, sin embargo, las razones por las que encontraron traducción a algunas jerarquías son diferentes a loa analizadas en el caso anterior. Los nombres de las jerarquías desplegadas en la Tabla 6-7 no presentan traducción ya que son módulos creados por la herramienta de síntesis y representan la unificación de dos o más módulos, por lo tanto en el HDL no se encuentra instanciadas con un nombre similar.

Tabla 6-7 Jerarquías sin traducción en el modelo de un circuito controlador de memoria

Nombre de la jerarquía
/core_0/DP_OP_51_357_3394
/core_0/i_Wunit_aiosfdown_0_aiosfdown_adapter0/aiosfdown_0_DP_OP_94_333_4474
/core_0/i_Wunit/arequp_0/DP_OP_98_331_6816
/core_0/i_Wunit/arequp_0/arequp_ah_0/sub_x_275_0
/core_0/i_Xunit_hctl0/hslave0_hcc0/add_x_272_0
/core_0/i_Xunit_hctl0/hslave0_ioq0/DP_OP_149_349_4549
/core_0/i_Wunit/amessage_0/amessage_config_0/add_x_278_0

6.1.3 Prueba del pre-proceso

Como se explicó en la Figura 5-5, el pre-proceso cuenta con tres fases: jerarquías definidas en el modelo sintetizado, jerarquías definidas en el HDL y traducción de jerarquías. Las pruebas realizadas en las tres fases del pre-proceso se ejecutaron sobre el modelo sintetizado de un circuito DMI.

6.1.3.1 Jerarquías definidos en el modelo sintetizado

El algoritmo de búsqueda de jerarquías en el modelo sintetizado del DMI encontró 1401 jerarquías, las cuales son ordenas alfabéticamente y almacenas en un archivo de texto plano

En la Figura 6-3 se presenta un extracto de las jerarquías encontradas, el cual se puede interpretar como:

- En el nivel 1, el "/" representa todo el modelo.
- En el nivel 2 están los sub-módulos principales que componen el modelo, uno de ellos es /dmi_top.
- El nivel 3 hay otros sub-módulos más pequeños que los de nivel 2, los sub-módulos de /dmi_top son /dmi_top/dmi4_pciepli4s y /dmi_top/dmi4ctrl8l4.

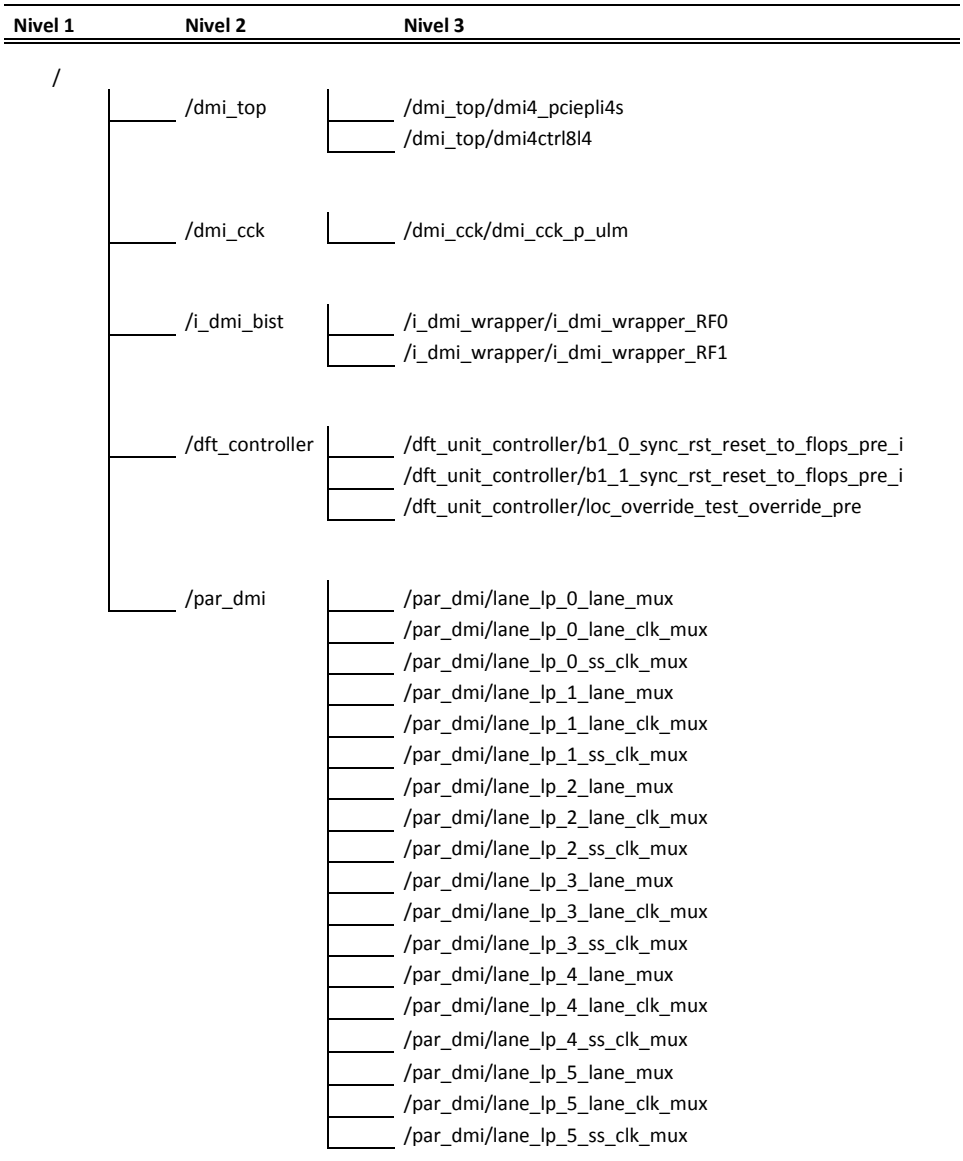


Figura 6-3 Principales jerarquías definidas en el modelo sintetizado de un circuito DMI

En la Figura 6-3 únicamente se presentan las jerarquías hasta un nivel jerárquico de tres, sin embargo el archivo de resultados contiene jerarquías con diez niveles de módulos.

6.1.3.2 Jerarquías definidas en el HDL

El algoritmo de búsqueda de las jerarquías definidas en el HDL se basa en un FSDB, generado a partir de la simulación de una prueba funcional sobre el HDL, y encuentra 5943 jerarquías en el HLD. En la Figura 6-4 se representan las principales jerarquías definidas en el HDL de un circuito DMI, el máximo nivel jerárquico encontrado fue de 12 módulos.

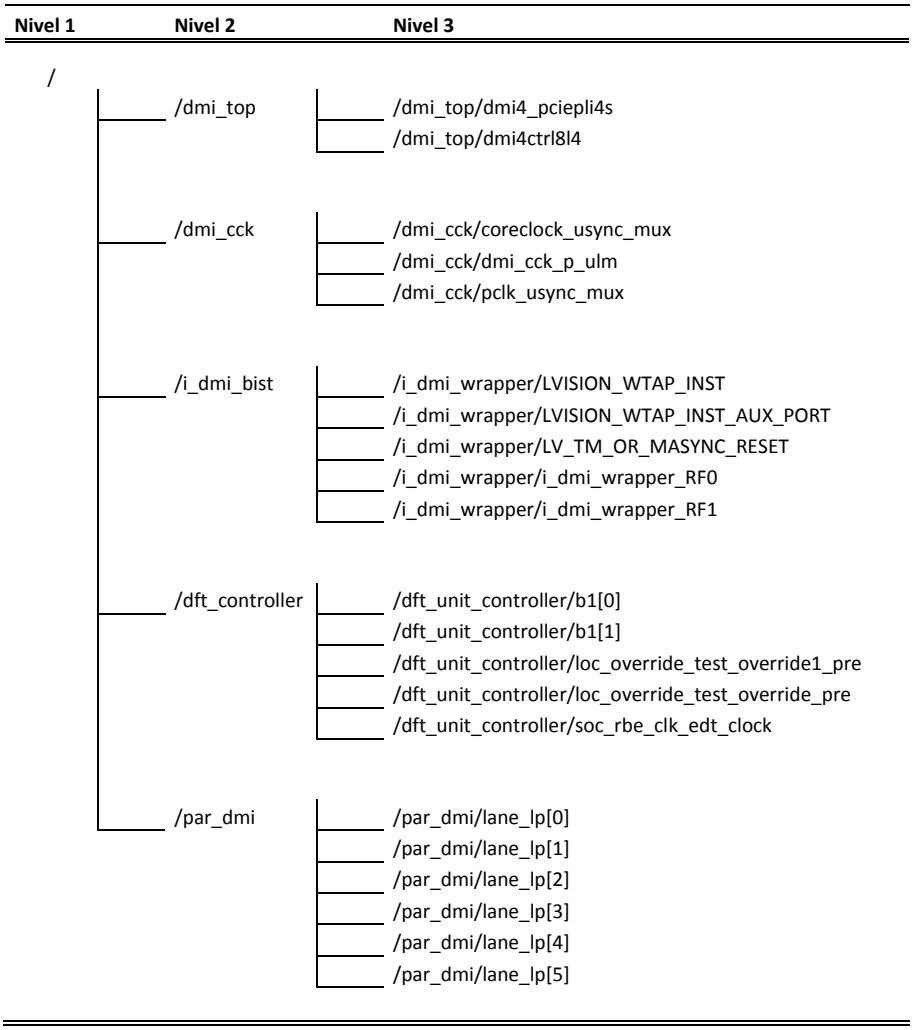


Figura 6-4 Principales jerarquías definida en el HDL para un circuito DMI

Si se comparan las jerarquías obtenidas en el nivel 3 del modelo sintetizado del DMI con las jerarquías obtenidas del HDL se puede observar grandes diferencias en cuanto al nombre

recibido por una misma jerarquía. Además, el número de módulos definidos para cada jerarquías de nivel 2 varía en cantidad, por ejemplo en módulo /par_mux_dmi en el modelo sintetizado tiene 15 sub-módulos definidos mientras que en el HDL solo se encuentran 5.

6.1.3.3 Traducción de jerarquías

El algoritmo de traducción de jerarquías es el encargado de encontrar equivalencias para los diferentes nombres dados a una misma jerarquía. En la Figura 6-5 se representa un extracto de la salida del algoritmo, donde en la primera columna se muestra el nombre recibido por una jerarquía en el modelo sintetizado y la segunda columna en el HDL. El archivo original contiene la traducción de 1378 jerarquías. Existe un archivo aparte donde se encuentra 23 jerarquías a las cuales no se encontró una traducción. Estas jerarquías necesitaron ser inspeccionadas manualmente.

Jerarquía en el modelo	Jerarquía en el HDL
/	/
/pcieutilrs	/pcieutilrs
/pcieutilrs/crmc_util_r	/pcieutilrs/crmc_util_r
/pcieutilrs/crmc_util_r_crmc_util	/pcieutilrs/crmc_util_r/crmc_util
/pcieutilrs/crmc_util_r/crmc_util/cr_block_right_block_cr_block	/pcieutilrs/crmc_util_r/crmc_util/cr_block/right_block/cr_block
/pcieutilrs/glob_dist_cr_block_0	/pcieutilrs/glob_dist/cr_block[0]
/pcieutilrs/glob_dist_cr_block_1	/pcieutilrs/glob_dist/cr_block[1]
/pcieutilrs/glob_dist_cr_block_2	/pcieutilrs/glob_dist/cr_block[2]
/pcieutilrs/dmi_msgpclk_ulm	/pcieutilrs/dmi_msgpclk_ulm
/pcieutilrs/dmi_msgpclk_ulm/lane_lp_0_lane_clk_mux	/pcieutilrs/dmi_msgpclk_ulm/lane_lp[0]/lane_clk_mux

Figura 6-5 Traducción de jerarquías para el modelo de un circuito DMI

6.2 Búsqueda de errores

En la sección 5.2 se explican los tres de modos de ejecución con que cuenta el algoritmo de búsqueda de errores. A continuación, se exponen las pruebas realizadas sobre los modos de ejecución llamados simulación inicial y simulación recursiva. Las pruebas efectuadas en el modo de ejecución llamado simulación inicial para una jerarquía no se presentan en esta sección, ya que este modo es una combinación de los anteriores y no presenta nuevos datos para el análisis.

6.2.1 Simulación inicial

En la verificación lógica se comparan los valores obtenidos a partir de la simulación de una prueba funcional, en los pines de salida del modelo sintetizado y del HDL. Determinar si una prueba funcional detecta divergencias en el modelo es el primer paso de la verificación lógica y puede ser realizado mediante la ejecución del modo de simulación inicial.

En la Figura 6-6 se presentan el reporte generado al ejecutar el algoritmo de búsqueda de errores, modo de simulación inicial, en el modelo sintetizado de un circuito DMI. Las primeras líneas del reporte especifican los comandos utilizados en la ejecución del algoritmo, por si es necesario repetir la prueba en el futuro, y la segunda parte del reporte detallan las divergencias detectadas por la prueba.

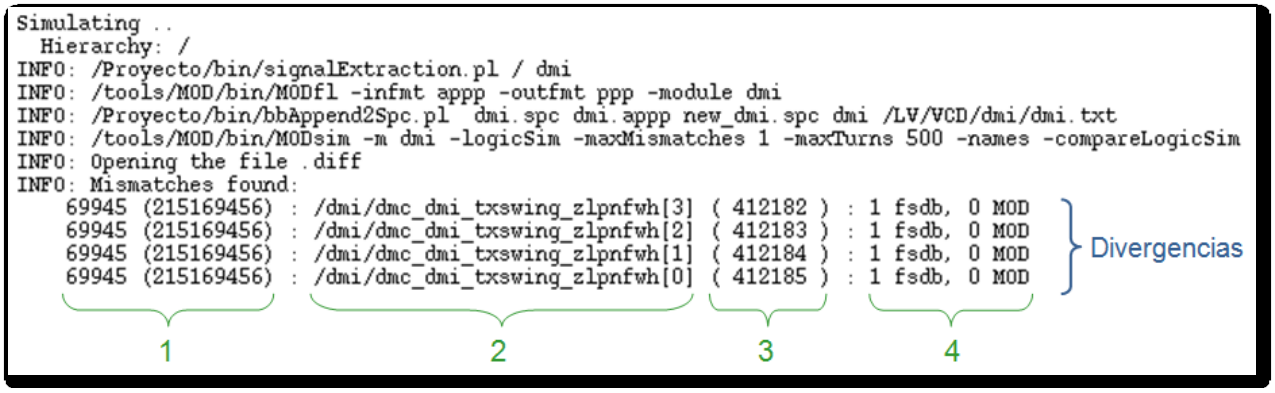


Figura 6-6 Reporte de ejecución del algoritmo de búsqueda de errores en modo: simulación inicial

Los números del uno al cuatro señalados en la Figura 6-6 pueden ser interpretadas de la siguiente forma:

5. Tiempo donde se detecta la divergencia: el primer número representa el tiempo de la simulación en el modelo y el número entre paréntesis es el tiempo de la simulación en el FSDB.
6. Nombre de la señal que presenta una divergencia.
7. Número que identifica la señal dentro del modelo sintetizado.
8. Causa de la divergencia: el primer número indica el valor que tiene la señal en el fsdb (equivalente a la simulación del HDL) y el segundo número valor en el modelo sintetizado.

La información provista por el reporte anterior, nombre de las señales con problemas y el tiempo donde suceden, se puede iniciar la verificación lógica del modelo. En este punto se puede decidir si se utiliza el algoritmo de búsqueda de errores en el modo recursivo o el procedimiento manual de *traceback.*

Únicamente las pruebas que detecten divergencias en el modelo sintetizado son de utilidad para la verificación lógica. El modo de simulación inicial permite descartar las pruebas que no detectan divergencias y las remite a la siguiente etapa de *fault grading* para calcular la cobertura de fallas de un circuito.

La razón por la que algunas pruebas detectan divergencias y otras no se debe al hecho de que cada prueba funcional está diseñada para ejercitar una función o característica específica del circuito. Por lo tanto, las pruebas funcionales que no detectan divergencias son las que ejercitan lógica del modelo que no presenta inconsistencias o problemas de síntesis.

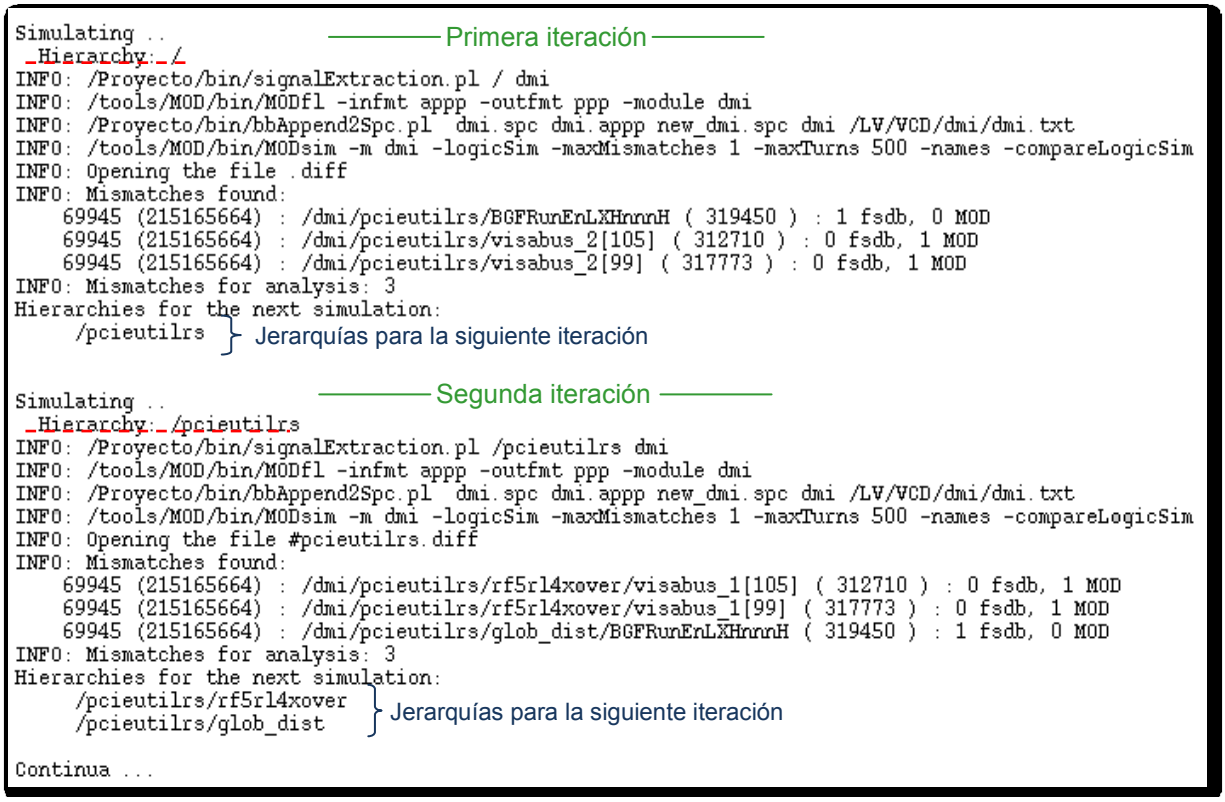
6.2.2 Simulación recursiva

La simulación recursiva permite comparar los pines de algunos módulos internos definidos en el modelo sintetizado con sus equivalentes en el HDL, con el fin de determinar el módulo de menor jerarquía definido en el modelo sintetizado donde se localiza un error.

La prueba efectuada para demostrar el funcionamiento del algoritmo de búsqueda de errores se realizó sobre el modelo sintetizado de un circuito DMI y como estímulo fue usada una prueba funcional enfoca en ejercitar la lógica relativa al circuito. En la Figura 6-7, se muestra una sección del reporte generado al ejecutar el algoritmo de búsqueda de errores, modo de simulación recursiva. El reporte completo se presenta en el apéndice A.4, donde se aprecia que fueron necesarias en total ocho iteraciones sobre diferentes módulos para identificar el módulo donde se origina el error.

En el reporte generado se observa que la primera iteración se realizó sobre la jerarquía principal “/” (señalada en la figura con la línea roja) y se detectaron 3 divergencias. Las tres divergencias encontradas se encuentran en la misma jerarquía por lo que la siguiente iteración se realiza únicamente sobre el módulo nombrado “/pcieutilrs”.

La segunda iteración realizada sobre la jerarquía “/pcieutilrs” localiza tres señales con divergencias en las jerarquías “/pcieutilrs/rf5rl4xover” y “/pcieutilrs/glob_dist”. Primero se analiza el módulo “/pcieutilrs/rf5rl4xover” y cuando no se encuentren más divergencias se inicia con el módulo “/pcieutilrs/glob_dist”.




```
Simulating ..
 Hierarchy: /
INFO: /Proyecto/bin/signalExtraction.pl / dmi
INFO: /tools/MOD/bin/MODfl -infmt appp -outfmt ppp -module dmi
INFO: /Proyecto/bin/bbAppend2Spc.pl  dmi.spc dmi.appp new_dmi.spc dmi /LV/VCD/dmi/dmi.txt
INFO: /tools/MOD/bin/MODsim -m dmi -logicSim -maxMismatches 1 -maxTurns 500 -names -compareLogicSim
INFO: Opening the file .diff
INFO: Mismatches found:
    69945 (215165664) : /dmi/pcieutilrs/BGFRunEnLXHnnnH ( 319450 ) : 1 fsdb, 0 MOD
    69945 (215165664) : /dmi/pcieutilrs/visabus_2[105] ( 312710 ) : 0 fsdb, 1 MOD
    69945 (215165664) : /dmi/pcieutilrs/visabus_2[99] ( 317773 ) : 0 fsdb, 1 MOD
INFO: Mismatches for analysis: 3
Hierarchies for the next simulation:
     /pcieutilrs

Simulating ..
 Hierarchy: /pcieutilrs
INFO: /Proyecto/bin/signalExtraction.pl /pcieutilrs dmi
INFO: /tools/MOD/bin/MODfl -infmt appp -outfmt ppp -module dmi
INFO: /Proyecto/bin/bbAppend2Spc.pl  dmi.spc dmi.appp new_dmi.spc dmi /LV/VCD/dmi/dmi.txt
INFO: /tools/MOD/bin/MODsim -m dmi -logicSim -maxMismatches 1 -maxTurns 500 -names -compareLogicSim
INFO: Opening the file #pcieutilrs.diff
INFO: Mismatches found:
    69945 (215165664) : /dmi/pcieutilrs/rf5rl4xover/visabus_1[105] ( 312710 ) : 0 fsdb, 1 MOD
    69945 (215165664) : /dmi/pcieutilrs/rf5rl4xover/visabus_1[99] ( 317773 ) : 0 fsdb, 1 MOD
    69945 (215165664) : /dmi/pcieutilrs/glob_dist/BGFRunEnLXHnnnH ( 319450 ) : 1 fsdb, 0 MOD
INFO: Mismatches for analysis: 3
Hierarchies for the next simulation:
     /pcieutilrs/rf5rl4xover
     /pcieutilrs/glob_dist

Continua ...
```

Figura 6-7 Reporte de ejecución del algoritmo de búsqueda de errores en modo: simulación recursiva

El reporte generado permite conocer el procedimiento utilizado por la herramienta para identificar los errores. También, existe otro archivo que resume los resultados obtenidos y los presenta en un formato como el que se muestra en la Figura 6-8. Para interpretar el reporte se debe considerar:

- El número precedido por “vec”, que representa el tiempo en la simulación donde se detecta el error.
- La información después de “hec”, que señala la jerarquía donde se detecta el error.
- Las señales que presentan el error se especifican después de “mis”.

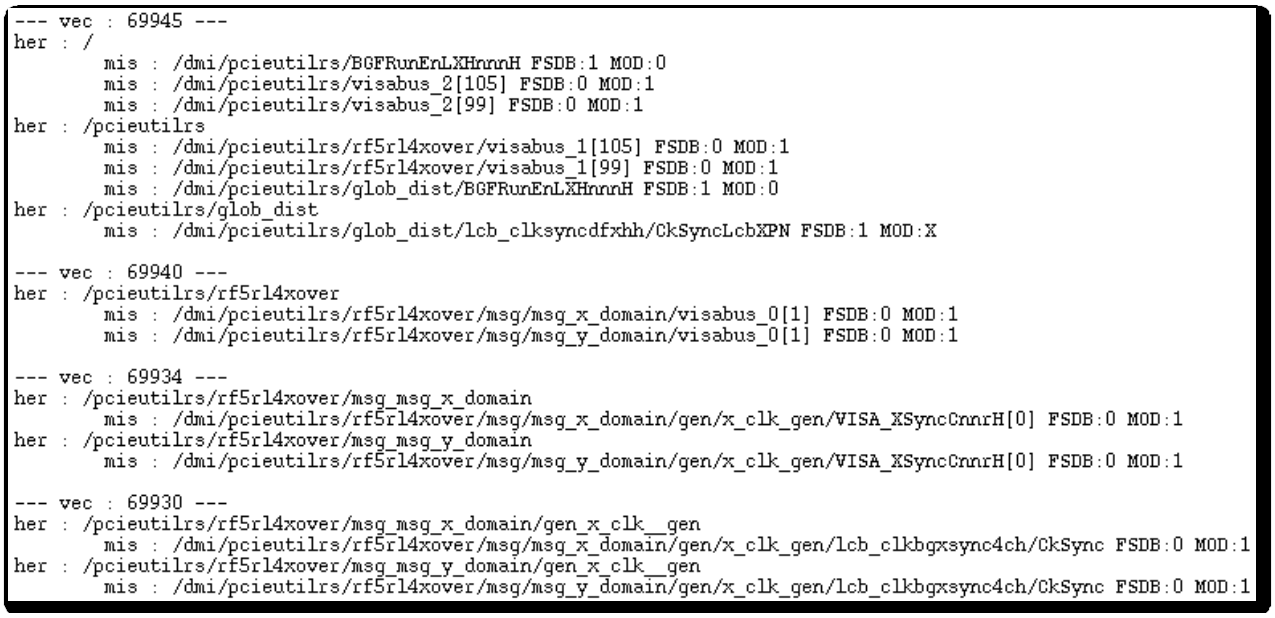


Figura 6-8 Archivo de resultados del algoritmo de búsqueda de errores en el modo de simulación recursiva

Los resultados de la ejecución del algoritmo de búsqueda modular son desplegados de acuerdo al tiempo donde se observa el error. Los errores que suceden primero en el tiempo se presentan al final del reporte y son los puntos de partida para determinar la fuente del error. Por ejemplo, en el tiempo 69930 se detecta un error en la señal “CkSync” del módulo en /dmi/pcieutilrs/rf5rl4xover/msg/msg_x_domain/gen/x_clk_gen/lcb_clkbgxsync4ch/, el error se propaga en la lógica del circuito y es observado en el tiempo 69934 en la señal “VISA_XSyncCnnrH[0]” del módulo /dmi/pcieutilrs/rf5rl4xover/msg/msg_x_domain/gen/ x_clk_gen/. La propagación de la falla continúa hasta ser detectada en los pines de salida del circuito en el tiempo 69945.

En Figura 6-9 se muestra que el módulo “x_clk_gen” tiene instanciados tres sub-módulos: “lcb_clksynch”, “lcb_clkbgxsync2ch” y “lcb_clkbgxsync4ch”. Estos sub-módulos no tienen definidos módulos más pequeños, por lo que se puede ser considerado como el módulo más pequeño definido en el modelo sintetizado para la jerarquía “x_clk_gen”.

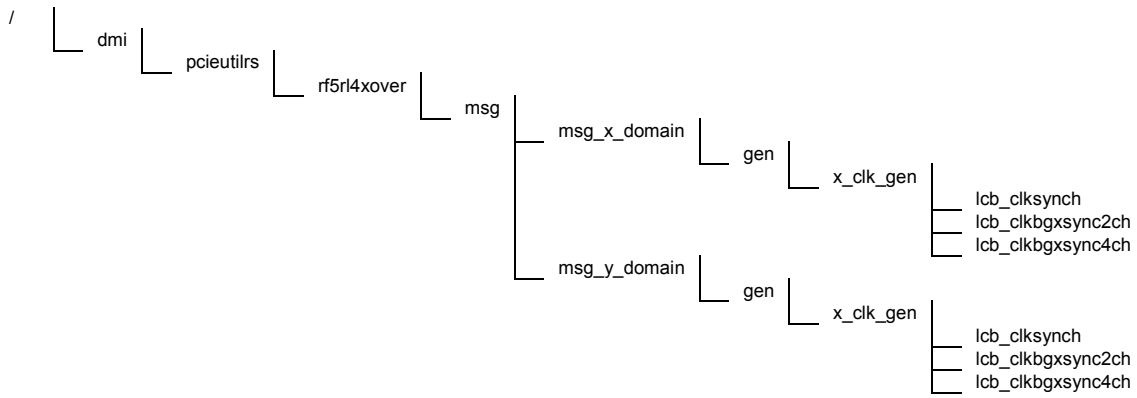


Figura 6-9 Definición jerárquica para el módulo msg

En la verificación lógica realizada previamente en el mismo modelo sintetizado por un ingeniero de la empresa, se identifican los siguientes módulos como los responsables de ocasionar las divergencias en el modelo:

- /dmi/pcieutilrs/rf5rl4xover/msg/msg_x_domain/gen/x_clk_gen/lcb_clkbgxsync4ch/
- /dmi/pcieutilrs/rf5rl4xover/msg/msg_y_domain/gen/x_clk_gen/lcb_clkbgxsync4ch/

Comparando los resultados mostrados en el reporte de la Figura 6-8 con los obtenidos en la verificación lógica manual se comprueba que el algoritmo de búsqueda de errores identifica los módulos de menor jerarquía donde se generan las divergencias en el modelo sintetizado de un circuito DMI.

6.3 Rastreo y localización de las fallas

En la sección 6.2.2 se muestran los resultados correspondientes al rastreo y localización de una de las fallas encontradas en el modelo sintetizado del circuito DMI. Para identificar todas las fallas del modelo del circuito DMI fue necesario ejecutar en tres ocasiones el algoritmo de búsqueda de errores. El tiempo de simulación utilizado en las tres ejecuciones del algoritmo de búsqueda errores fue contabilizado en la Tabla 6-8. Además en la misma tabla se muestra el tiempo empleado por un ingeniero en identificar la compuerta o nodo, perteneciente al módulo de menor jerarquía identificado por el algoritmo de búsqueda de errores, donde se origina la falla en el modelo. La última columna de la tabla registra el tiempo total empleado en la verificación funcional del modelo sintetizado del circuito DMI.

Tabla 6-8 Comparación del tiempo empleado con y sin la utilización de la herramienta desarrollada en la verificación lógica del modelo sintetizado de un circuito DMI

	Tiempo de simulación (horas)	Tiempo ingeniero (horas)	Tiempo total (horas)
Verificación lógica utilizando el algoritmo de búsqueda modular	20,5	10,3	30.8
Verificación lógica manual	120,5	120,5	120,5

La verificación lógica manual realizada en el modelo del circuito DMI requirió de total 120,5 horas de trabajo y como se aprecia en la Tabla 6-8 no existe una diferencia entre el tiempo de simulación y el tiempo de ingeniero, ya que la simulación realiza es interactiva. Una simulación interactiva significa que el ingeniero trabaja en el aislamiento de la falla y a su vez simula la prueba funcional en el modelo del circuito, por lo tanto no se puede separar del tiempo empleado en la simulación y el tiempo de trabajo del ingeniero de forma objetiva.

La Tabla 6-9 se presentan los tiempos empleados en la verificación lógica manual y en la verificación utilizando la herramienta desarrollada. La diferencia de tiempo entre las dos metodologías es de 89.7 horas, lo que equivale a una reducción del 74,43% en el tiempo requerido en la verificación del modelo sintetizado de un circuito DMI.

Tabla 6-9 Reducción en el tiempo empleado en la verificación lógica del modelo sintetizado de un circuito DMI haciendo uso del algoritmo de búsqueda modular

	Modelo sintetizado de un circuito DMI
Tiempo empleado en realizando la verificación lógica de forma manual (horas)	120,5
Tiempo empleado en la verificación lógica haciendo uso del algoritmo de búsqueda modular (horas)	30.8
Diferencia en el tiempo empleado para la verificación lógica (horas)	89.7
Porcentaje de reducción en el tiempo	74.43%

CAPÍTULO 7: CONCLUSIONES Y RECOMENDACIONES

En esta sección se presentan conclusiones y recomendaciones sobre el algoritmo implementado para agilizar y acelerar el rastreo y localización de una falla lógica en los modelos sintetizados de circuitos integrados.

7.1 Conclusiones

Al finalizar este proyecto se concluye:

1. Se diseñó el algoritmo de búsqueda modular para acelerar el rastreo y localización de fallas en los modelos sintetizados

2. El algoritmo de búsqueda modular fue seleccionado para acelerar el rastreo y localización de fallas en los modelos sintetizados, dadas las ventajas que presenta sobre el algoritmo de *traceback*.

3. La herramienta de software desarrollada utiliza el algoritmo de búsqueda modular para rastrear y aislar con éxito las fallas encontradas en el modelo sintetizado de circuitos integrados.

4. En las pruebas realizadas en el modelo sintetizado de un circuito DMI, se comprobó que el algoritmo de búsqueda modular localiza el origen de los errores lógicos en el bloque de menor jerarquía.

5. El tiempo empleado en la verificación lógica del modelo sintetizado de un circuito DMI mediante la utilización de la herramienta desarrollada fue de 30.8 horas. Si se compara con las 120.5 horas empleadas en la verificación manual del mismo modelo, se puede afirmar que la reducción fue de 74.43%.

6. Cuando se utiliza el algoritmo de búsqueda modular, la reducción en el tiempo empleado para localizar las fallas de los modelos sintetizados de circuitos integrados es mayor al 50%, por lo que se superan las expectativas planteados por la empresa Intel.

7.2 Recomendaciones

A continuación se citan algunas recomendaciones que permitirían el seguimiento de este proyecto:

- Adaptar el algoritmo de comparación de caracteres para reconocer diferencias entre letras mayúsculas o minúsculas, permitiría al algoritmo de traducción de jerarquías ser más versátil y aplicable a modelos sintetizados de nuevos circuitos.

- Almacenar y restaurar el estado lógico del modelo sintetizado para una prueba funcional y en tiempo específico, agilizaría el asilamiento de fallas. Con la implementación de *check points* en la herramienta de simulación se obtendría una disminución en el tiempo de simulación, para aquellos casos donde se ejecuta más de una vez la misma prueba funcional sobre el modelo.

CAPÍTULO 8: BIBLIOGRAFÍA

1. **Ferrero Martín, Francisco Javier.** Descripción de Circuitos Digitales mediante VHDL. *Dispositivos Lógicos Programables.* [En línea] 21 de Mayo de 1999. [Citado el: 16 de Junio de 2010.] http://www.ate.uniovi.es/campo/PLD/teoria/VHDL%20(Francisco%20Javier%20Ferrero).pdf.

2. **García Dopico, Antonio.** Biblioteca Universitaria Politécnica. *Distribución de carga y aumento del grado de paralelismo en simulación síncrona de lenguajes de descripción de hardware.* [En línea] Universidad Politécnica de Madrid, Setiembre de 2000. [Citado el: 16 de Junio de 2010.] http://oa.upm.es/123/1/10200014.pdf.

3. **Torres Valle, Francisco Javier.** *CAPÍTULO I: Lenguajes de descripción de hardware.* [En línea] [Citado el: 16 de Junio de 2010.] http://www.udistrital.edu.co/comunidad/profesores/jruiz/jairocd/texto/cirdig/vhdl/man2.pdf.

4. **Singer, Godi.** *Fault Grading Methodology.* Haifa, Israel : Training and Publciation Services Intel Corporation Santa Clara - California, 1990.

5. **Wang, Laung-Terng, Wu, Cheng-Wen y Wen, Xiaoqing.** *VLSI test principles and architectures: design for testability.* United States of America : Morgan Kaufmann, 2006. págs. 1-34.

6. **Azmi, Syazwani Binti.** Combinational Multiplier Fault Modeling and Test. *Universidad Tecnológica de Malasia, Facultad de Ingeniería Electrónica.* [En línea] Abril de 2008. [Citado el: 6 de Abril de 2010.] http://psm.fke.utm.my/libraryfke/files/61_SYAZWANIBINTIAZMI2008.pdf.

7. **Mylavarapu, Ajit Karthik y Athi, Sanjai Balakrishnan.** Patent application title: Methods, Systems, and Computer Program Products for Evaluating Electrical Circuits From Information Stored in Simulation Dump Files. *Patentdocs.* [En línea] [Citado el: 9 de Abril de 2010.] http://www.faqs.org/patents/app/20100070257.

8. **Chengyu, Mou.** Verification Avenue. *Boost Verification Performance by Introducing VCS.* [En línea] Agosto de 2002. [Citado el: 05 de Enero de 2010.] http://www.synopsys.com/company/documents/veriavenue/q302/verification_ave5.pdf.

9. **SourceIII, Support.** TRANSLATING PRINT-ON-CHANGE FSDB VECTORS TO CYCLE-BASED VECTORS FOR TESTERS. *SourceIII.* [En línea] [Citado el: 9 de Abril de 2010.] http://www.sourceiii.com/notes-POC_fsdb_vectors.html.

10. **Novas.** Linking Novas Files with Simulators to Enable FSDB Waveform Dumping. *CADfamily.* [En línea] [Citado el: 9 de Abril de 2010.] http://www.cadfamily.com/download/EDA/Novas/NOVAS%20linking.pdf.

11. **McLaughlin, Rich K.** *Quake User Guide.* s.l. : Intel Design & Technology Solutions , 2009.

12. *Computing the Levenshtein Distance of a Regular Language.* **Konstantinidis, Stavros.** 2005, IEEE ISOC ITW2005 on Coding and Complexity, págs. 113-116.

13. **Gilleland, Michael.** Levenshtein Distance, in Three Flavors. [En línea] Febrero de 2006. [Citado el: 15 de Junio de 2010.] http://www.merriampark.com/ld.htm.

14. **Yang, C. Han.** Chapter 4. Hamming Distance. *Department of Electrical Engineering Stanford University.* [En línea] Enero de 1998. [Citado el: 28 de Junio de 2010.] http://verify.stanford.edu/hyang/thesis/40HD.pdf.

15. Componentes Intel de Costa Rica. *Intel.* [En línea] 2009. [Citado el: 7 de Abril de 2010.] http://www.intel.com/costarica/costarica/.

APÉNDICES

A.1 Glosario

ASIC, significa Application-Specific Integrated Circuit, que se traduce como: circuito integrado para aplicaciones específicas. Es un circuito integrado hecho a la medida para un uso en particular, en vez de ser concebido para propósitos de uso general. 4

ATPG, significa Automatic Test Pattern Generation, que se traduce como: generación automática de patrones de prueba. Es un método de diseño de automatización electrónica / tecnología utilizada para encontrar una secuencia de entrada que, cuando se aplica a un circuito digital, permita distinguir entre el comportamiento del circuito correcto y el comportamiento del circuito defectuoso. 17

CUT, significa Circuit Under Test, que se traduce como: circuito bajo prueba. 8

DFT, significa Design for Testability, que se traduce como: diseño para pruebas. Es el nombre para las técnicas de diseño que añaden algunas características de prueba a un diseño de hardware de productos microelectrónicos. 17

DMI, significa Direct Media Interface. Es el circuito que realiza la interconeccion punto a punto entre el MCH (Memory Controller Hub) y ICH (I/O Controller Hub). 28, 36

falla, representación de un defecto que provoca que el circuito deje de operar en la forma requerida. 8

Fault Grading, es una medida de la calidad de detección de errores que proporciona un número de vectores de simulación dados. 6

HDL, significa Hardware Description Laguage, que se traduce como: lenguaje de descripción de hardware. Permite documentar las interconexiones y el comportamiento de un circuito electrónico, sin utilizar diagramas esquemáticos. 17, 18

JTAG, un acrónimo para Joint Test Action Group, es el nombre común utilizado para la norma IEEE 1149.1 titulada Standard Test Access Port and Boundary-Scan Architecture para test access ports utilizada para testear PCBs utilizando escaneo de límites. 17

LFSR, significa linear feedback shift register, que se traduce como: registro de desplazamiento con retroalimentación lineal. Es un registro de desplazamiento en el cual la entrada es un bit proveniente de aplicar una función de transformación lineal a un estado anterior. 17

netlist, archivo con la descripción de la conectividad de un diseño electrónico. 6

A.2 Cálculo del porcentaje de similitud de 2 cadenas de caracteres

$$\%\ de\ similitud = \frac{Número\ de\ caracteres\ de\ la\ cadena\ \#1 - Distancia\ de\ Levenshtein}{Número\ de\ caracteres\ de\ la\ cadena\ \#1} \quad (A.2\text{-}1)$$

A.3 Comparación y traducción de las jerarquías

Tabla A.3-1 Comparación y traducción de las jerarquías para varios grados de similitud

Porcentaje de similitud	70%	72%	74%	76%	78%	80%	82%	84%	86%	88%	90%	92%	94%	96%	98%	100%
Comparaciones realizadas	3645	3149	2454	1927	1771	1633	1541	1436	1165	927	696	586	321	190	75	0
Comparaciones exitosas	223	236	236	243	263	283	290	309	326	332	332	326	292	182	68	0
Comparaciones insatisfactorias	3422	2913	2218	1684	1508	1350	1251	1127	839	595	364	260	29	8	7	0
Jerarquías traducidas	1378	1378	1378	1378	1378	1378	1378	1378	1378	1378	1378	1358	1312	1205	1065	386
Jerarquías sin traducción	23	23	23	23	23	23	23	23	23	23	23	43	89	196	336	1015
Tiempo de ejecución (min)	36.5	36.5	36.4	36.4	36.4	36.4	36.4	36.0	36.1	36.4	38.0	41.4	99.3	265.5	448.1	374.6

A.4 Archivo de registro para el algoritmo de búsqueda de errores

```
Simulating ..
  Hierarchy: /
INFO: /Proyecto/bin/signalExtraction.pl / dmi
INFO: /tools/MOD/bin/MODfl -infmt appp -outfmt ppp -module dmi
INFO: /Proyecto/bin/bbAppend2Spc.pl  dmi.spc dmi.appp new_dmi.spc dmi /LV/VCD/dmi/dmi.txt
INFO: /tools/MOD/bin/MODsim -m dmi -logicSim -maxMismatches 1 -maxTurns 500 -names -compareLogicSim
/LV/Simulation/test/.diff -log /LV/Simulation/test/.log -fsdb /LV/VCD/test.fsdb -P zeros -ppp dmi.ppp -fsdbTop
/dmi -P zeros -spc new_dmi.spc -v 10 -dofile ~/Proyecto/bin/simulation.do
INFO: Opening the file .diff
INFO: Mismatches found:
    69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[3] ( 412182 ) : 1 fsdb, 0 MOD
    69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[2] ( 412183 ) : 1 fsdb, 0 MOD
    69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[1] ( 412184 ) : 1 fsdb, 0 MOD
    69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[0] ( 412185 ) : 1 fsdb, 0 MOD
    69945 (215165664) : /dmi/pcieutilrs/BGFRunEnLXHnnnH ( 319450 ) : 1 fsdb, 0 MOD
    69945 (215165664) : /dmi/pcieutilrs/visabus_2[105] ( 312710 ) : 0 fsdb, 1 MOD
    69945 (215165664) : /dmi/pcieutilrs/visabus_2[99] ( 317773 ) : 0 fsdb, 1 MOD
INFO: Mismatches for analysis: 3
Hierarchies for the next simulation:
     /pcieutilrs


Simulating ..
  Hierarchy: /pcieutilrs
INFO: /Proyecto/bin/signalExtraction.pl /pcieutilrs dmi
INFO: /tools/MOD/bin/MODfl -infmt appp -outfmt ppp -module dmi
INFO: /Proyecto/bin/bbAppend2Spc.pl  dmi.spc dmi.appp new_dmi.spc dmi /LV/VCD/dmi/dmi.txt
INFO: /tools/MOD/bin/MODsim -m dmi -logicSim -maxMismatches 1 -maxTurns 500 -names -compareLogicSim
/LV/Simulation/test/#pcieutilrs.diff -log /LV/Simulation/test/#pcieutilrs.log -fsdb /LV/VCD/test.fsdb -P zeros -ppp
dmi.ppp -fsdbTop /dmi -P zeros -spc new_dmi.spc -v 10 -dofile ~/Proyecto/bin/simulation.do
INFO: Opening the file #pcieutilrs.diff
INFO: Mismatches found:
    69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[3] ( 412182 ) : 1 fsdb, 0 MOD
    69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[2] ( 412183 ) : 1 fsdb, 0 MOD
    69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[1] ( 412184 ) : 1 fsdb, 0 MOD
    69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[0] ( 412185 ) : 1 fsdb, 0 MOD
    69945 (215165664) : /dmi/pcieutilrs/rf5rl4xover/visabus_1[105] ( 312710 ) : 0 fsdb, 1 MOD
    69945 (215165664) : /dmi/pcieutilrs/rf5rl4xover/visabus_1[99] ( 317773 ) : 0 fsdb, 1 MOD
    69945 (215165664) : /dmi/pcieutilrs/glob_dist/BGFRunEnLXHnnnH ( 319450 ) : 1 fsdb, 0 MOD
INFO: Mismatches for analysis: 3
Hierarchies for the next simulation:
     /pcieutilrs/rf5rl4xover
     /pcieutilrs/glob_dist


Simulating ..
  Hierarchy: /pcieutilrs/rf5rl4xover
INFO: /Proyecto/bin/signalExtraction.pl /pcieutilrs/rf5rl4xover dmi
INFO: tools/MOD/bin/MODfl -infmt appp -outfmt ppp -module dmi
INFO: /Proyecto/bin/bbAppend2Spc.pl  dmi.spc dmi.appp new_dmi.spc dmi /LV/VCD/dmi/dmi.txt
INFO: tools/MOD/bin/MODsim -m dmi -logicSim -maxMismatches 1 -maxTurns 500 -names -compareLogicSim
/LV/Simulation/test/#pcieutilrs#rf5rl4xover.diff      -log      /LV/Simulation/test/#pcieutilrs#rf5rl4xover.log      -fsdb
```

/LV/VCD/test.fsdb -P zeros -ppp dmi.ppp -fsdbTop /dmi -P zeros -spc new_dmi.spc -v 10 -dofile ~/Proyecto/bin/simulation.do
INFO: Opening the file #pcieutilrs#rf5rl4xover.diff
INFO: Mismatches found:
69940 (215165064) : /dmi/pcieutilrs/rf5rl4xover/msg/msg_x_domain/visabus_0[1] (317773) : 0 fsdb, 1 MOD
69940 (215165064) : /dmi/pcieutilrs/rf5rl4xover/msg/msg_y_domain/visabus_0[1] (312710) : 0 fsdb, 1 MOD
INFO: Mismatches for analysis: 2
Hierarchies for the next simulation:
/pcieutilrs/rf5rl4xover/msg_msg_x_domain
/pcieutilrs/rf5rl4xover/msg_msg_y_domain

Simulating ..
Hierarchy: /pcieutilrs/rf5rl4xover/msg_msg_x_domain
INFO: /Proyecto/bin/signalExtraction.pl /pcieutilrs/rf5rl4xover/msg_msg_x_domain dmi
INFO: tools/MOD/bin/MODfl -infmt appp -outfmt ppp -module dmi
INFO: /Proyecto/bin/bbAppend2Spc.pl dmi.spc dmi.appp new_dmi.spc dmi /LV/VCD/dmi/dmi.txt
INFO: tools/MOD/bin/MODsim -m dmi -logicSim -maxMismatches 1 -maxTurns 500 -names -compareLogicSim /LV/Simulation/test/#pcieutilrs#rf5rl4xover#msg_msg_x_domain.diff -log /LV/Simulation/test/#pcieutilrs#rf5rl4xover#msg_msg_x_domain.log -fsdb /LV/VCD/test.fsdb -P zeros -ppp dmi.ppp -fsdbTop /dmi -P zeros -spc new_dmi.spc -v 10 -dofile ~/Proyecto/bin/simulation.do
INFO: Opening the file #pcieutilrs#rf5rl4xover#msg_msg_x_domain.diff
INFO: Mismatches found:
69934 (215164664) : /dmi/pcieutilrs/rf5rl4xover/msg/msg_x_domain/x_clk_bubble_gen/VISA_XSyncCnnrH[0] (317773) : 0 fsdb, 1 MOD
INFO: Mismatches for analysis: 1
Hierarchies for the next simulation:
/pcieutilrs/rf5rl4xover/msg_msg_x_domain/gen_x_clk__gen

Simulating ..
Hierarchy: /pcieutilrs/rf5rl4xover/msg_msg_x_domain/gen_x_clk__gen
INFO: /Proyecto/bin/signalExtraction.pl /pcieutilrs/rf5rl4xover/msg_msg_x_domain/gen_x_clk__gen dmi
INFO: tools/MOD/bin/MODfl -infmt appp -outfmt ppp -module dmi
INFO: /Proyecto/bin/bbAppend2Spc.pl dmi.spc dmi.appp new_dmi.spc dmi /LV/VCD/dmi/dmi.txt
INFO: tools/MOD/bin/MODsim -m dmi -logicSim -maxMismatches 1 -maxTurns 500 -names -compareLogicSim /LV/Simulation/test/#pcieutilrs#rf5rl4xover#msg_msg_x_domain#gen_x_clk__gen.diff -log /LV/Simulation/test/#pcieutilrs#rf5rl4xover#msg_msg_x_domain#gen_x_clk__gen.log -fsdb /LV/VCD/test.fsdb -P zeros -ppp dmi.ppp -fsdbTop /dmi -P zeros -spc new_dmi.spc -v 10 -dofile ~/Proyecto/bin/simulation.do
INFO: Opening the file #pcieutilrs#rf5rl4xover#msg_msg_x_domain#gen_x_clk__gen.diff
INFO: Mismatches found:
69930 (215163456) : /dmi/pcieutilrs/rf5rl4xover/msg/msg_x_domain/x_clk_bubble_gen/lcb_clkbgxsync4ch/CkSyncLcbXPN (317755) : 0 fsdb, 1 MOD
INFO: Mismatches for analysis: 5
Hierarchies for the next simulation:
None

Simulating ..
Hierarchy: /pcieutilrs/rf5rl4xover/msg_msg_y_domain
INFO: /Proyecto/bin/signalExtraction.pl /pcieutilrs/rf5rl4xover/msg_msg_y_domain dmi
INFO: tools/MOD/bin/MODfl -infmt appp -outfmt ppp -module dmi
INFO: /Proyecto/bin/bbAppend2Spc.pl dmi.spc dmi.appp new_dmi.spc dmi /LV/VCD/dmi/dmi.txt
INFO: tools/MOD/bin/MODsim -m dmi -logicSim -maxMismatches 1 -maxTurns 500 -names -compareLogicSim /LV/Simulation/test/#pcieutilrs#rf5rl4xover#msg_msg_y_domain.diff -log

```
/LV/Simulation/test/#pcieutilrs#rf5rl4xover#msg_msg_y_domain.log -fsdb /LV/VCD/test.fsdb -P zeros -ppp dmi.ppp -fsdbTop /dmi -P zeros -spc new_dmi.spc -v 10 -dofile ~/Proyecto/bin/simulation.do
INFO: Opening the file #pcieutilrs#rf5rl4xover#msg_msg_y_domain.diff
INFO: Mismatches found:
   69934 (215164664) : /dmi/pcieutilrs/rf5rl4xover/msg/msg_y_domain/x_clk_bubble_gen/VISA_XSyncCnnrH[0]
                       ( 312710 ) : 0 fsdb, 1 MOD
INFO: Mismatches for analysis: 1
Hierarchies for the next simulation:
    /pcieutilrs/rf5rl4xover/msg_msg_y_domain/gen_x_clk__gen


Simulating ..
  Hierarchy: /pcieutilrs/rf5rl4xover/msg_msg_y_domain/gen_x_clk__gen
INFO: /Proyecto/bin/signalExtraction.pl /pcieutilrs/rf5rl4xover/msg_msg_y_domain/gen_x_clk__gen dmi
INFO: tools/MOD/bin/MODfl -infmt appp -outfmt ppp -module dmi
INFO: /Proyecto/bin/bbAppend2Spc.pl  dmi.spc dmi.appp new_dmi.spc dmi /LV/VCD/dmi/dmi.txt
INFO: tools/MOD/bin/MODsim -m dmi -logicSim -maxMismatches 1 -maxTurns 500 -names -compareLogicSim /LV/Simulation/test/#pcieutilrs#rf5rl4xover#msg_msg_y_domain#gen_x_clk__gen.diff -log /LV/Simulation/test/#pcieutilrs#rf5rl4xover#msg_msg_y_domain#gen_x_clk__gen.log -fsdb /LV/VCD/test.fsdb -P zeros -ppp dmi.ppp -fsdbTop /dmi -P zeros -spc new_dmi.spc -v 10 -dofile ~/Proyecto/bin/simulation.do
INFO: Opening the file #pcieutilrs#rf5rl4xover#msg_msg_y_domain#gen_x_clk__gen.diff
INFO: Mismatches found:
   69930 (215163456) : /dmi/pcieutilrs/rf5rl4xover/msg/msg_y_domain/x_clk_bubble_gen/lcb_clkbgxsync4ch/
                       CkSyncLcbXPN ( 312698 ) : 0 fsdb, 1 MOD
INFO: Mismatches for analysis: 5
Hierarchies for the next simulation:
        None


Simulating ..
  Hierarchy: /pcieutilrs/glob_dist
INFO: /Proyecto/bin/signalExtraction.pl /pcieutilrs/glob_dist dmi
INFO: tools/MOD/bin/MODfl -infmt appp -outfmt ppp -module dmi
INFO: /Proyecto/bin/bbAppend2Spc.pl  dmi.spc dmi.appp new_dmi.spc dmi /LV/VCD/dmi/dmi.txt
INFO: tools/MOD/bin/MODsim -m dmi -logicSim -maxMismatches 1 -maxTurns 500 -names -compareLogicSim /LV/Simulation/test/#pcieutilrs#glob_dist.diff -log /LV/Simulation/test/#pcieutilrs#glob_dist.log -fsdb /LV/VCD/test.fsdb -P zeros -ppp dmi.ppp -fsdbTop /dmi -P zeros -spc new_dmi.spc -v 10 -dofile ~/Proyecto/bin/simulation.do
INFO: Opening the file #pcieutilrs#glob_dist.diff
INFO: Mismatches found:
   69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[3] ( 412182 ) : 1 fsdb, 0 MOD
   69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[2] ( 412183 ) : 1 fsdb, 0 MOD
   69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[1] ( 412184 ) : 1 fsdb, 0 MOD
   69945 (215169456) : /dmi/dmc_dmi_txswing_zlpnfwh[0] ( 412185 ) : 1 fsdb, 0 MOD
   69945 (215169456) : /dmi/pcieutilrs/glob_dist/lcb_clksyncdfxhh/CkSyncLcbXPN ( 319033 ) : 1 fsdb, X MOD
INFO: Mismatches for analysis: 1
Hierarchies for the next simulation:
        None
```